RAY TRACING COMPLEX SCENES ON A MULTIPLE-INSTRUCTION STREAM MULTIPLE-DATA STREAM CONCURRENT COMPUTER

By

MICHAEL BRANNON CARTER

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, Oklahoma

1987

Submitted to the Faculty of the Graduate
College of Oklahoma State University
in partial fulfillment of the
requirements for the Degree of
MASTER OF SCIENCE
December, 1989

RAY TRACING COMPLEX SCENES ON A MULTIPLE-INSTRUCTION STREAM MULTIPLE-DATA STREAM CONCURRENT COMPUTER

Thesis Approved:

Keith A. Teague
Thesis Advisor

Richard L. Cummins

C. M. Bacon

Norman N. Durham
Dean of the Graduate College

PREFACE

The Ray Tracing technique generates perhaps the most realistic looking computer-generated images. It does so at the cost of a great deal of computer time. Many algorithms have been developed to speed up the ray tracing procedure, but it still remains the most CPU-intensive realistic image synthesis method. To date, ray tracing has remained largely in the realm of serial computers. The research in this thesis takes ray tracing strongly into the parallel computing domain and deals effectively with all of the central issues surrounding the parallelization of this procedure. Results from the "Hypercube Ray Tracer" are collected and compared against other ray tracing systems. A new technique for ray tracing Constructive Solid Geometry objects is also developed and implemented.

The inspiration for this project came from two places at once. My advisor, Dr. Keith A. Teague, provided the spark that got me interested in the parallel processing field. Dr. Samuel P. Uselton, then of the University of Tulsa, infected me with the computer graphics bug. To these individuals goes my thanks for opening new horizons. Thanks are also due Ron Daniel, Chris Schuermann, Mark Vasoll, Gregg Wonderly, Roland Stolfa, and Eric Blazek, my friends, for their constant criticism, encouragement, and "why don't you do this" genre of questions. Many features and refinements to the Hypercube Ray Tracer sprang from their ideas and suggestions. To Mr. Ron Daniel, especially, I owe tremendous thanks for his experience, expertise, and constant attention. More than a few bugs were chased out of the ray tracer with his help.

Finally, my deepest debt of gratitide and respect must go to my parents, Everett and Murrel Carter. Without their years of patient upbringing, understanding, and wisdom, I would never have had the chance to learn.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

What is the Ultimate Goal of Computer Graphics

The last ten years of computing has witnessed an ever increasing demand for higher quality computer graphics. Computer graphics is now used not only for charts and graphs, but for much more complex applications such as medical imaging, architectural design, and flight simulation. In each of the latter cases, the highest possible degree of realism is desired to achieve maximum visual and creative impact. And, at least in the flight simulation case, real-time rendering is required. Even though not all applications for computer graphics require such speed and realism, all could benefit from it. High speed and realism represent the Holy Grail of computer graphics.

Computer graphics has always been computationally intensive. The additional burden of realism makes computational cost grow by orders of magnitude. Realistic images can now be rendered in a few minutes, rather than hours, due in large part to algorithm improvements. If real time image synthesis speeds are to be realized, these minutes must be compressed into milliseconds – a speedup of roughly 10^5. Clearly, there is much work yet to be done.

Already, single-processor computers are approaching processing speeds which are predominantly limited by the speed of light – not technology. From this realization has sprung the concept of parallel processing. To date, parallel processing has been applied only weakly to the problem of realistic image synthesis [Goldsmith 88, Carter 89]. If realistic image synthesis is to move into the real-time realm, then new approaches are needed. Parallel processing is one approach that promises significant performance gains in return for a modest increase in software complexity.

Purpose and Motivation

All image synthesis techniques have several things in common. All have a set of objects that are to be rendered. All have a point from which this "scene" is to be viewed and a "camera model" that models the optical characteristics of the imaginary viewer. Finally, all have an algorithm that renders an image based on the scene and some viewpoint. This rendering algorithm is the key to the quality and speed of the image synthesis technique. Usually, speed must be traded off against realism.

It is the purpose of this research to take one realistic image synthesis technique, known as ray tracing, into the parallel domain. This thesis concentrates on the implementation of a modular ray tracer on a specific distributed-memory parallel architecture. Much of the wisdom gained applies equally well to other parallel architectures, as well as to serial computers of all types.

There are many publications dealing with the fundamentals of ray tracing [Whitted 80, Phong 75, Foley 84]. These early publications, of course, do not deal with any of the parallel aspects of ray tracing. Later publications optimize various parts of the ray tracing process [Kay 86, Arvo 87, Fujimoto 86, Glassner 84, Goldsmith 87, Kajiya 83, Cook 84], but only a few have addressed problems specific to parallel implementation [Goldsmith 88]. The major thrust of this research is to explore and find solutions for these unaddressed problems. Toward this goal, many algorithms had to be modified to operate in the parallel environment.

As with all parallel implementations, the architecture of the target machine heavily influences the software architecture. If an inappropriate software architecture is chosen, performance will suffer. Also, if a software architecture is too complex, then its maintainance will be complicated. Conversely, the software architecture chosen must be flexible enough to accommodate future expansion. If it does not, the ray tracer's functionality will be limited. Several software architectures are considered for the ray tracer, and each is evaluated for its suitability.

All this talk of custom algorithms is not lost on the ultimate goal of this project – a fast, parallel ray tracer. In all cases, the utmost consideration has been given to the speed and efficiency of algorithms, either borrowed or developed, in the ray tracer. In numerous cases, special optimizations have been made, and each is described fully in the following chapters.

Overview

Work is presented in this thesis that has been performed in constructing a fast, usable, parallel ray tracer – the Hypercube Ray Tracer. The scope of this work includes such things as: using and modifying existing algorithms for the ray tracing process, developing new algorithms where necessary to implement new features or optimize for speed, and selecting an overall software architecture suitable for parallel implementation.

The School of Electrical and Computer Engineering at Oklahoma State University is privileged to have been selected as a beta-test site for the Intel iPSC/2 hypercube concurrent computer system. It is primarily upon this architecture that this work has been done, but consideration has been made for other parallel architectures. These considerations are noted in the subsequent chapters. The iPSC/2 is a distributed memory, medium grained parallel computer with a hypercube interconnect, and fast message routing hardware. It is one of the least expensive parallel computers available on a per node performance basis. This makes it an ideal testbed for the development of parallel ray tracing.

Chapter 2 presents an indepth discussion of the ray tracing process and its evolution to the present. Ray tracing is not the only realistic image synthesis technique, however. A method known as radiosity also produces high quality images. It too has certain strengths and weaknesses. Where ray tracing produces superior results on mostly specular scenes, the radiosity approach is best at scenes involving diffuse lighting. Neither technique is clearly superior to the other in a general sense, and implementing both techniques in a parallel fashion is beyond the scope of this thesis.

The techniques differ considerably in the way they generate images. Ray tracing is a strictly procedural technique involving many geometric calculations, and database searching operations. Radiosity, on the other hand, uses an iterative mathematical technique involving many matrix calculations. Also, as we shall see later, ray tracing is easily parallelizable on the pixel level. Since ray tracing is a more irregular type of procedure not involving the already well-studied parallel matrix methods, I chose it. Scope and limitation related topics are discussed at length in Chapter 3.

In chapter 4, tradeoffs are discussed that involve the ray tracer's implementation, and its major functional blocks. There are three major functional blocks in the hypercube ray tracer; these are the scene compiler (Rayd), the object database hierarchy generator (Hiergen), and the ray tracer itself (Ray). The scene compiler takes a human-readable description of the scene and converts it into the low-level format suitable for the hierarchy generator. A complete description of the scene description language is given in Appendix A. The hierarchy generator then takes this raw, unstructured list of objects and builds an efficient hierarchical representation of the scene that is directly read by the ray tracer. Finally, the ray tracer itself reads the hierarchical representation of the object database and renders a realistic image from it. The hierarchical database representation is discussed in Chapter 4, also.

The list of objects to be rendered is called the object database. When the number of objects to be rendered becomes too large, a single computing node can no longer hold them all. At this point, several nodes must cooperate to store all of the objects. This so-called object database distribution has been performed by Goldsmith, but in a different way and on a different parallel machine [Goldsmith 88]. Major changes to the basic ray tracer were necessary to implement the distributed object database. Chapter 5 discusses the approach taken, and its differences from the technique of Goldsmith.

Chapter 6 gives performance data, and compares the Hypercube Ray Tracer against other parallel and serial ray tracers. A number of scenes of varying complexity are used in order to give representative data over a wide range of input parameters.

Finally, Chapter 7 concludes and chapter 8 suggests topics that should be studied further.

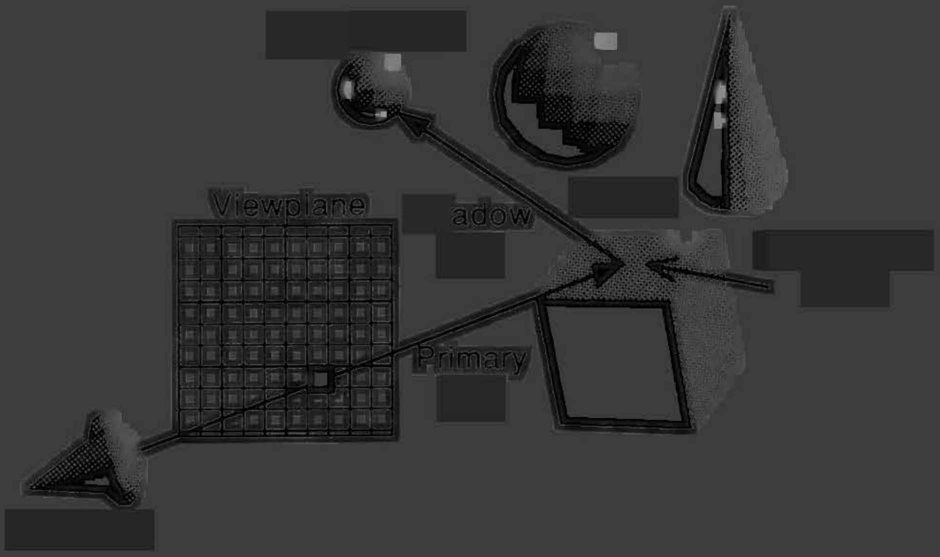
CHAPTER II

BACKGROUND

Outline of Ray Tracing Procedure

In order to go any further with the discussion of the Hypercube Ray Tracer, it is appropriate to review the evolution of the ray tracing technique. The basic parallel techniques thus far applied to ray tracing will also be discussed.

Ray tracing emerged as a realistic image synthesis technique in the mid 1970's [Whitted 88]. Its milieu consists of an observer, a viewplane, and a set of objects called the scene. (See Figure 1) The observer is a point in 3D space from whose perspective the scene is to be rendered. The viewplane is an imaginary rectangle through which the observer sees the scene. It is divided into a number of boxes, each of which represents one pixel in the final image. It is then the task of the ray tracing procedure to find the light intensity present at each pixel. The scene is composed of a (potentially large) number of three-dimensional geometric figures called "primitive objects," or just "primitives." Primitives can be as simple as a sphere or cube to as complex as a fractal mountainside [Kajiya 83]. One helpful realization is that any light falling on a given pixel must have come from the direction along a ray from the observer to the pixel in question. This direction is antiparallel to the actual direction of light propagation. If one traces backward along this line of propagation into the scene, the surface from which the light was scattered can be discovered.



[redacted]

[redacted]

[redacted]

[redacted]

Examples here are the reflection in a mirror, and the highlights on the shiny surface of an apple. Transmitted (refracted) light is also modeled in most ray tracers. Ray tracer shading started with just these basic effects [Whitted 80].

Reflected light and refracted light are handled by straightforward applications of the law of reflection and Snell's law, respectively. The new ray thus formed is ray traced just like a primary ray, and its intensity contribution added to the other components of the shading model.

No mention, as yet, has been given as to where light comes from in a scene. Some objects in the scene are designated as light sources. Although they are thought of as "sources", rays are never traced outward from them, only toward them. There are an infinite number of possible ray paths outward from each light source. Clearly, all of them cannot be traced in a finite time. Here lay one of the major flaws in ray tracing. Light is not allowed to propagate outward in all directions from the light sources into the scene and interact as it does in reality. With ray tracing, such features as razor sharp shadows are common, whereas in reality, they are not. A light source is brought into play only when the shading model needs to know how much light is falling on the point in question. These shading models do not, as a rule, take into account global illumination effects such as shadow penumbrae.

A given light source may or may not be occluded by some other primitive as viewed from the point in question. In order to determine this, the shading model will fire a ray from the point in question toward the light source. If the ray hits any object other than the light source, the point is not illuminated by that light source. In this case, the intersection point being shaded is in the shadow of the occluding object.

Primitives

In order to ray trace a primitive, one must know how to do two things. First, a way must be found to determine all of the primitive's points of intersection with a ray.

Second, the normal vector at these intersection points must be computed. These are the only two geometrical pieces of information needed by a shading model to shade an intersection point.

Primitives vary widely in geometrical complexity. One of the simplest useful primitives in ray tracing is the sphere. It is described by a simple implicit equation and is readily intersected against a ray by solving a quadratic equation. (See Appendix C) The surface normal vector is trivially obtained by constructing a vector from the center of the sphere to the point of intersection, and normalizing. This illustrates the simplest intersection and normal vector calculation.

Other primitives whose intersections have direct solutions are cylinders, cones, and polygons. Although a polygon is not a solid, a polyhedron may be formed from several polygons. Any surface defined by a second degree equation or lower also has a direct solution. This includes parabolas, hyperbolas, and all nonuniformly scaled versions of spheres, and cylinders. Higher-order surfaces require iterative numerical intersection algorithms.

The Parallel Nature of Ray Tracing

The brightness of each pixel on the viewplane is completely independent of its neighbors. That is to say, the process by which a pixel's brightness is calculated in no way depends on the adjacent pixels. Clearly, correlation does exist between adjacent pixels, but the calculations themselves are independent. Pixel independence gives ray tracing the fine-grained parallelism that makes it suitable for fine-, medium-, and coarse-grained parallel architectures. The viewplane may be broken up into some number of pixel groups, and each group assigned to a processing element for ray tracing. This is an image decomposition approach to parallel ray tracing [Carter 89, Orcutt 88]. This strategy has the distinct advantage of simplicity. It is easy to visualize some number of processors, each ray tracing a portion of an image. Once all processors have finished their portions, the complete image is assembled from the pieces, and the ray tracing is complete.

Image decomposition is not the only option for decomposing the ray tracing task. One may break up the ray tracing algorithm into a pipeline of stages [Gaudet 88]. These stages might be labeled as: ray initiation, ray-object intersection, shading, and pixel storage. Each pipeline stage would be implemented on a single computing element (node), and data would be passed between stages as needed. The ray initiation stage would be responsible for initiating all primary, reflected, and refracted rays. Clearly, this stage must communicate with the shading stage. The second stage's function is self-explanatory. The shading stage would take intersection points, found by the second stage, and apply the shading model to them. Obviously, this stage must request that reflected, refracted, and shadow rays be initiated by the first stage. Once the third stage completes, it sends its results on to the fourth and final stage for conversion into RGB triples and storage in the frame buffer. This decomposition is better suited to specialized hardware rather than to homogeneous parallel computers.

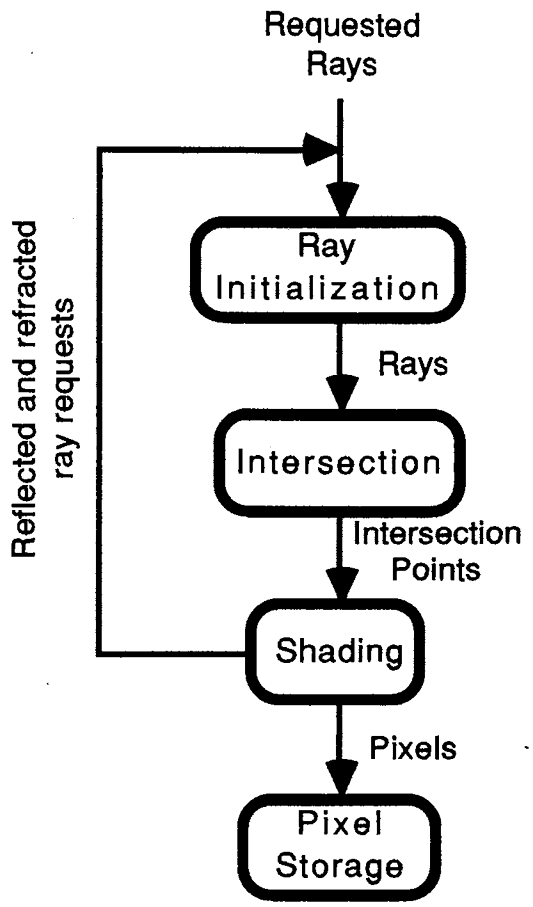


Figure 2: Flow of Data in the Hypothetical Pipelined Ray Tracer

Still another decomposition technique distributes the object database (ODB), and passes rays, in various stages of completion, among the nodes [Goldsmith 88]. This might be labeled database decomposition. During the intersection process, a ray may need to be intersected against a primitive that does not reside on that node. In this event, the node sends the ray off to the node that does have that primitive. When the ray arrives at the new node, the intersection process is picked up where it left off. This "database miss" may happen more than once during the intersection process. It is assumed, here, that it is more efficient to move a ray between nodes than to move portions of the ODB. For finer grained machines with smaller node memories, this is indeed the case.

These are but a few of the possible ways that the ray tracing process could be decomposed onto a parallel architecture. But before one of these methods, or another method entirely, is selected for the Hypercube Ray Tracer, the specific target architecture needs to be considered further.

How The Technique Has Improved Since Its Conception

Camera Models

The observer and viewplane constitute what is called a "camera model." This is the simplest possible camera model – that of a pinhole camera. It has an infinite depth of field, so all object viewed are in sharp focus [Foley 84]. This is neither like our eye nor like a camera: it is unrealistic. Better camera models than the pinhole camera model have been developed [Potmesil]. Potmesil presents a model based on a lens and aperture. The model accurately reproduces the effects seen in a real camera such as focus and depth of field, but is not easily adapted to ray tracing. It is much better suited to other rendering techniques such as Z-buffers or scanlines [Foley 84].

Shading Models

Better shading models were developed as experience in realistic image synthesis grew and illumination engineers came onto the ray tracing scene [Whitted 80, Cook 82]. These shading models superseded the more simplistic models of [Phong 75] and [Blinn 77]. The model of [Cook 84] is particularly interesting since it includes wavelength-dependent effects. One such effect is the color shift seen on a metallic surface as the viewing angle with the surface becomes small. Although this model opened new vistas of realism, it further taxed the already slow ray tracing procedure with yet more computations. Most shading models use just three spectral bands to render images – red, green, and blue. But since these colors are just three discrete samples along a continuous frequency spectrum, they cannot adequately model the way light interacts with the environment.

Torrance's model uses many wavelengths along the visible light spectrum. In this way, the behavior of many different wavelengths may be modeled separately. These samples can be combined, via color-science principles, to give the classical RGB tristimulus values that our eye perceives.

Intersection Philosophy

Discovering a ray's closest intersection point is the major consumer of time in the ray tracing process. The simple solution is to merely intersect the ray with all objects in the scene, and select the closest. This algorithm has the unfortunate attribute of being O(n) in the number of objects in the scene. For example, if one wishes to render a scene composed of 1000 primitive objects at a resolution of 512 by 512 pixels, over a quarter of a billion ray-object intersections would be required in the worst case. This number can be greatly reduced by organizing the object database (ODB) into some better structure and using a more efficient intersection strategy.

Most intersection acceleration techniques go hand in hand with some specific ODB organization philosophy. There are two broad classes of ODB philosophies; object subdivision [Weghorst 84, Kay 86] and space subdivision [Glassner 84, Fujimoto 86, Arvo 87, Uselton 89]. Each presents its own unique set of advantages and disadvantages. To date, neither has shown itself to be clearly superior over the other. The fastest algorithms in each class are roughly the same speed as one another. Furthermore, the overhead for each class of algorithm in terms of time and memory is almost identical.

Space subdivision techniques attempt to divide 3D space into a number of disjoint volumes, each of which wholly or partially contains a number of the scene primitives. To intersect a ray against this structure involves stepping through the subvolumes along the path of the ray intersecting against each primitive associated with that subvolume. Space is usually divided using an adaptive, variable depth octree [Fujimoto 86, Glassner 84], or slabs [Uselton 89]. One notable exception to this rule is the strategy developed by Arvo and Kirk [Arvo 87]. They describe a technique called "ray classification" which is based on 5-dimensional adaptive space subdivision, rather than the usual three dimensional space subdivision. From the starting position and direction of a ray, the ray classification algorithm derives a small set of primitives against which the ray must be checked. Position and direction of a ray constitute the five degrees of freedom which are used as the basis for a 5D hypercube which encompasses all possible ray origins and directions. Associated with each subcube is a list of all possible objects which a ray in that subcube might intersect – a candidate set. At first, this 5D hypercube is very coarsely subdivided. As rays are cast, it is successively subdivided into smaller hypercubes which contain fewer and fewer candidate primitives. Space subdivision techniques have the advantage that they query primitives in the order they occur along the ray, but they also have the disadvantage of splitting up the ODB.

Object subdivision techniques impose a structure on the ODB rather than space itself. They organize the primitives into a structure of a more classical nature – a hierarchy resembling a tree [Weghorst , Goldsmith 87, Kay 86]. To intersect a ray against this type of structure involves traversing the hierarchy from the root, down to the leaves while checking the ray against each node. The ray may sometimes intersect more than one subtree of a given node. Therefore, more than one primitive must sometimes be tested.

Object Hierarchies and Bounding Volumes

Many strategies have been developed that improve the intersection performance to $O(\log_2 n)$ or better [Rubin 80, Weghorst 84, Kay 86, Arvo 87, Fujimoto 86]. The technique in [Weghorst] suggests a hierarchical organization of the object database and judicious use of bounding volumes. Bounding volumes have been the topic of much research. Their function and their advantages are discussed below.

Bounding volumes are very simple geometric objects, such as spheres and cubes, that may be placed around primitive objects. Since bounding volumes are simpler objects than most primitives, it is much faster to intersect against a bounding volume than a primitive. In this way, one may test a ray against a bounding volume first. If the ray misses the bounding volume, then it must also miss the primitive. When this happens, the ray does not have to be intersected against the primitive at all. Since many rays do, in fact, miss the object they are being tested against, a substantial time savings is realized. This time savings is maximized when a primitive's bounding volume fits very tightly around it. As the size of the enclosing bounding volume shrinks, more rays will miss it, and thus avoid the potentially costly ray- primitive intersection.

Through the use of bounding volumes, one can speed up the intersection process by a constant factor. However, by reorganizing the object database into a tree and developing a new searching technique, one can realize logarithmic speedup! The hierarchy is formed by partitioning the object database into small groups of primitives, placing a bounding volume around each of these groups, and continuing recursively until there is a single bounding volume enclosing the entire scene. Using the following algorithm, one may traverse the hierarchy to find the closest primitive [Weghorst 84].

```
Given a ray
elements = children of root node
While elements is not empty
   While not at end of elements
      c = current element
      If c has been tested
         Advance to next element in elements
      Else
         If ray intersects element c
            If element c is not a primitive
               Replace c with its children in elements
            Else if distance to intersection > 0.0
               Advance to next element in elements
            Else
               Remove c from elements
            Endif
         Endif
      Endif
   Endwhile
   If elements is not empty
      e = element in elements with least intersection distance
      If e is not a primitive
         Replace e with its children in elements
      Else
         Return e
      Endif
   Endif
Endwhile
Return nil
```

Algorithm 1: Weghorst Hierarchical Intersection

Weghorst states that the hierarchy should be constructed carefully, but gives no objective measures by which to judge the quality of a hierarchy. It should be fairly clear that a group of primitives in close spatial proximity to one another will give a smaller bounding volume that a group with even one outlier. The tightness with which bounding volumes fit is a crucial issue in hierarchy creation. Performance of the hierarchical intersection procedure may degrade to **O**(n) for a worst-case hierarchy.

Salmon and Goldsmith have developed an algorithm which generates a nearly optimal hierarchy [Goldsmith 87]. It constructs a hierarchy one primitive at a time given a partially shuffled list. Since there are n objects to be inserted into the hierarchy, the algorithm complexity is **O**(n log n). The algorithm decides where to place an additional primitive based on a simple cost function. The cost function for a node is based on the increase in its bounding volume area if the primitive is added to that node. In other words, the primitive is placed such that the increase in bounding volume area of all nodes is minimized. The order in which the primitives are inserted into the hierarchy has an impact on the optimality of the hierarchy, but practice has shown that excellent results are obtained when partially shuffled modeler order is used. Modeler order is the spatially coherent order in which the objects in the scene are usually modeled.

Intersection

Not only has the object database structure improved, so have the ways in which it is traversed to find the closest ray-object intersection. A little thought will disclose the fact that the Weghorst algorithm intersects a ray with a potentially large number of primitives before it finally finds the closest one. Primitive intersections are precisely the types of intersections that bounding volumes are designed to reduce. What if we could intersect the ray with primitives in its path in the order that they occurred? This would cut down considerably on the number of ray-primitive intersections performed per call. Just such a technique has been developed by [Kay 86], and is presented here.

```
Initialize heap to empty
Initialize t_nearest = +∞     { Distance to nearest primitive }
Initialize p_nearest = nil    { Pointer to nearest primitive  }
While heap is not empty and distance to top node < t_nearest
   Extract candidate from heap
   If the candidate is a primitive
      Compute ray-primitive intersection
      If ray hits candidate and distance < t_nearest
         t_nearest = distance
         p_nearest = candidate
      Endif
   Else
      For each child of the candidate
         Compute ray-bounding volume intersection
         If the ray hits the bounding volume
            Insert the child into the heap
         Endif
      Endfor
   Endif
Endwhile
```

Algorithm 2: Kay Hierarchical Intersection

One interesting thing to note about the Kay algorithm is the use of a heap. The heap always keeps the closest distance to a primitive- or bounding-volume intersection at its root. In this way, the primitives are intersected with the ray in the order they occurred along the ray. This idea of intersecting the ray against primitives in order along the ray is largely responsible for the greatly-improved efficiency of the Kay algorithm over previous hierarchy-intersection algorithms.

Bounding Volumes

Bounding volumes are another potential target for improvement. Although a sphere is a very simple and easy to use geometrical figure, intersecting a ray against it requires a quadratic equation be solved. If the bounding volume intersection could be simplified, then we would benefit greatly. Note that we really don't need the exact point of intersection with the bounding volume, rather we need only know *if* the ray intersects. One solution is to use polyhedra formed by the intersection of three or more "slabs" [Kay 86]. A slab is the infinite volume of space contained between two parallel planes. An arbitrarily complex convex polyhedron can be formed by intersecting a number of slabs. For example, a right rectangular prism is formed by intersecting three orthogonal slabs. Once certain simple preliminary calculations are made, intersecting a ray with a slab involves only two multiplies, two subtracts, and a comparison. Compare this with 10 additions 16 multiplications, and a square root for a sphere.

Primitives

Simple primitives have already been mentioned, but their usefulness for representing real scenes is limited. Seldom do we see a perfectly spherical rock, or a perfectly smooth surface. More complex primitives are needed to make a scene look more realistic.

Once the equation of a surface goes above second degree, it no longer has a direct algebraic solution, and we are forced to resort to numerical techniques. Some primitives whose intersection is handled by numerical methods are the superquadric [Barr 86], the generalized cylinder [Bronsvoort 85], bicubic patches [Kajiya 83], algebraic surfaces [Hanrahan], and swept surfaces [Kajiya 83, VanWijk 84]. Swept surfaces are a class of procedurally-defined objects. The generalized cylinder is actually a swept surface, but its complexity warrants mention on its own.

Fractals are another type of procedural object [Mandelbrot 77, Kajiya 83]. There are two ways of ray tracing a fractal surface. One may fully evolve the surface and instantiate each facet with a single polygon, or one may elect to treat the surface as a single primitive. In the first option, a database of many thousands of polygons would result. This would waste much memory because many of the polygons would not be visible from a given viewpoint. The second option would result in only one primitive in the object database, but it would have an intensive, and possibly very costly, intersection procedure. Kajiya gives a full and elegant treatment of just such an intersection scheme. The algorithm evolves the surface in tandem with the intersection, and its performance is surprisingly good. In this same paper, Kajiya also describes techniques for intersecting a ray with prisms and surfaces of revolution. A prism is a two-dimensional, closed curve swept along a straight path, and a surface of revolution is a two-dimensional, closed curve swept along a circular path.

Constructive Solid Geometry

A crucial distinction needs to be made at this point between some of the aforementioned primitives. Some, like the sphere and polyhedron, are solids. Others, like a polygon or bicubic patch, are not solids. They are just a two dimensional sheet warped in three-space and do not enclose any volume. If we turn them into solids by completing an enclosed space somehow, then they become candidates to be used with a technique known as constructive solid geometry (CSG) [Goldstein 71, Roth 82, Yossef 86].

CSG refers the process of applying boolean operations on the members of a set of solid objects in order to produce a new solid object. For example, we might construct a nut by subtracting a cylindrical hole from the middle of a square block. A CSG expression for this operation might look like, "A and not B," where A is the square block and B is the cylinder meant to be the hole.

CSG is one technique that can be applied to existing primitives to build new and different objects. It is not the only one, however. Certain deformations such as tapering, bending, and twisting may also be applied to primitives [Barr 84]. There are two approaches to the ray-deformed primitive intersection problem. One may either intersect a straight ray against a deformed primitive, or one may intersect a deformed ray against an undeformed primitive [Barr 84]. Further research by Barr has led to the development of methods suitable for intersecting a ray with any differentiable surface [Barr 86]. Differentiable surfaces include both parametric surfaces and implicit surfaces. This encompasses twisting, bending, and tapering deformations, as well as the primitives themselves. Therefore, Barr's methods may be used to intersection against arbitrarily deformed surfaces. These methods are numerical in nature and, of course, much slower than direct ray intersection solution.

Antialiasing

All image synthesis techniques suffer from aliasing unless specific measures are taken to eliminate it. Aliasing manifests itself as jagged edges, and moiré patterns. It is a result of sampling the image on a regular grid. Many researchers have proposed methods to combat this problem [Lee 89, Heckbert 86, Mitchell 87, Whitted 80, Amanatides 84, Abram 85, Dippe 85, Cook 84, Kajiya 83]. The methods proposed by Lee and Mitchell are discussed briefly below. They represent a good cross-section of the methods proposed by the above group of researchers.

The method of Lee is an adaptive technique based on the variance of many rays cast through a single pixel. To start with, a number of rays are traced through a pixel. The variance in brightness of these rays is then examined to determine if more rays should be traced. If the variance is sufficiently low, then no more rays are traced, and the existing samples are averaged in some way. Different theories exist for choosing the distribution for tracing rays within the pixel, and the method for averaging them. One way is to trace the rays in a uniform distribution, and take a uniformly weighted average of the resulting samples. Another way is to cast the sample rays in a Gaussian distribution about the center of the pixel, and take a uniformly-weighted average of the samples. Finally, there is the converse of the last method – cast samples in a uniform distribution and take a Gaussian weighted average. All of these methods are types of low-pass spatial filters, and produce similar visual results. Since this method deals only with samples from one pixel, it preserves ray tracing's pixel-level parallelism.

Mitchell's method is also an adaptive pixel subsampling technique, but with different sampling criterion and a different sample averaging technique. Pixels are initially sampled once each in a nonuniform "jitter" pattern. These initial samples are then examined, and areas of high contrast are supersampled. Supersampling is also done nonuniformly. When the whole image has been sufficiently supersampled, the samples are combined using a four stage, ever coarser box filter. The result of this method, of course, is a uniformly resampled antialiased image. Since the supersampling criterion is based on information from a number of neighboring pixels, the pixel-level parallelism is destroyed. Although initial sampling rays may be cast independently of one another, supersampling rays depend on the results of other rays. This method in its present form, therefore, is unsuitable for the Hypercube Ray Tracer.

Distributed Ray Tracing

Distributed ray tracing [Cook 84] solves many basic problems with the images generated with classical ray tracing. Distributed ray tracing (DRT) is not to be confused with the distributed object database I have implemented. The problems of shadow penumbrae, motion blur, depth of field, and other fuzzy phenomena are addressed by DRT. No more rays are required than with standard antialiasing techniques. The rays used are distributed according to various distribution functions. For example, shadow rays are distributed across the solid angle subtended by the light source in question. Reflected rays are distributed according to the object's specular reflectance function. A similar distribution is performed on refracted rays. Depth of field is produced by distributing the primary ray origins over the theoretical camera lens. Finally, motion blur is produced by distributing the primary rays over the interval of time encompassed by the current frame. In this case, the positions of all objects in motion must be recomputed for each ray.

When used with the antialiasing strategy outlined by Lee, the pixel-level parallelism of ray tracing is preserved. This makes DRT an ideal candidate for the Hypercube Ray Tracer.

Architecture of the iPSC/2

We will now focus our attention upon the iPSC/2 – the parallel computer on which the Hypercube Ray Tracer was developed. The iPSC/2 system consists of two subsystems: the system resource manager (SRM, sometimes called the "host"), and the hypercube itself (sometimes called the "cube" or "tower"). The SRM is a standalone microcomputer connected to one node of the hypercube. Its purpose is to act as a software development platform, as well as the administrator of the hypercube. All user interaction with programs running on the hypercube is handled through the SRM.

The hypercube portion of the iPSC/2 is a homogeneous array of computing nodes connected in a binary n-cube. The hypercube interconnect is implemented by 2.8 megabyte per second communication links and special hardware that optimizes message routing. Each node is composed of an 80386 microprocessor, 1-16 MB of RAM, a floating point coprocessor, and communications hardware. These features place the iPSC/2 into the medium grained, distributed memory, MIMD (Multiple Instruction Stream, Multiple Data Stream), hypercube interconnect class of parallel computers.

iPSC/2 programs are generally written in two parts: a host part that runs on the SRM, and a node part that runs on each node of the cube. Although this structure is not mandatory, it provides a convenient paradigm from which to work. Usually, the host program handles such non-parallel functions as terminal I/O, disk I/O, or network access. The node program or programs, meanwhile, handle all problem specific processing. This model has no serious drawbacks, and does not impose any severe limitations on the Hypercube Ray Tracer as we shall see later.

CHAPTER III

DESIGN CHOICES AND IMPLEMENTATION

Scope of Project and Statement of Goals

To reiterate, the main purpose of this research is to efficiently parallelize the ray tracing process. This task does not stand alone, however. Certain other issues must be addressed before one can begin to think about ray tracing, proper. The first question that comes to mind is, "What are we going to ray trace, and more specifically, how are we going to represent it to the computer?" Some method of scene specification is needed in addition to the ray tracer.

A number of ray-object intersection methods have already been discussed. For the purpose of this research, I have chosen to implement one technique. The programming task involved in implementing one intersection model is formidable enough in itself to render others beyond the scope of this thesis. This will become evident in subsequent discussion.

I have also chosen to implement just one relatively simple shading model. The choice of shading models does not impact the parallelism of the ray tracing process, so the choice is largely arbitrary. I have selected the model proposed by [Phong 75] since it is easy to implement. The illumination equation is as follows:

$$\mathbf{I} = \mathbf{I_a} + \mathbf{I_d} \sum_{i=1}^{m} (\underline{\mathbf{N}} * \underline{\mathbf{L}}) + \mathbf{I_s} \sum_{i=1}^{m} (\underline{\mathbf{R}} * \underline{\mathbf{V}})^n \qquad (1)$$

Where:

$\mathbf{I}$	= The total light intensity falling on a point.
$\mathbf{I_a}$	= The constant ambient illumination.
$\mathbf{I_d}$	= The diffuse reflection characteristic.
$\mathbf{I_s}$	= The specular reflection characteristic.
$\underline{\mathbf{N}}$	= Unit normal vector at the point in question.
$\underline{\mathbf{L}}$	= Unit vector in the direction of the i'th light source.
$\underline{\mathbf{R}}$	= Unit vector in the direction of maximum specular reflection. This is the mirror direction of L.
$\underline{\mathbf{V}}$	= Unit vector in the direction of the viewer.
n	= "Specularity" exponent.
m	= The number of light sources.

The variable **I** in the above equation is, in fact, a vector quantity. Since we are modeling light reflection in terms of the RGB tristimulus values, all illumination variables and constants are actually triples. $\mathbf{I_a}$ is the light intensity that is thought of as constant and falling on all surfaces. It is called the "ambient light" intensity. $\mathbf{I_d}$ may be thought of as the color of the surface in question. For purposes of simplicity, the specular reflection characteristic is defined as $\mathbf{I_s} = k_s * \mathbf{I_d}$, where k_s is called the "specular reflection coefficient and ranges between 0 and 1. This way, only one surface color need be specified for each primitive.

A new, more realistic shading model can be added at a later date with relative ease, and virtually no impact on the program structure as a whole. As the choice of shading models does not contribute to the parallel aspects of ray tracing, it is not emphasized as a major design choice.

The choice of primitives will have a major impact on the ray tracer's performance. This, in turn, will influence how credible the Hypercube Ray Tracer's performance data is in relation to that from other ray tracers. A set of primitives must be selected that is complex enough to be useful for solid modeling. They must not be so complex that ray-primitive intersection time is always large, though. A compact, but representative sample of popular primitives is called for.

Three possible parallelization strategies have been described. Each one has its own merits and liabilities. It is easy to see that the decomposition model chosen will have the most wide reaching effects on the ray tracer's program structure. Each decomposition would require a completely different program structure. For this reason, I have chosen a single decomposition model.

Design Choices

Scene Description

How do we efficiently describe a scene to the computer? First, let us state the qualities a good scene description language has. It should be easy for a user to write and modify; a text file would be ideal. Second, the scene description should be intuitive; the user should not have to memorize special codes, or formats. Third, the description language should be powerful and flexible; it should not leave any feature of the Hypercube Ray Tracer inaccessible. Fourth, it should be expandable. As new features are added to the ray tracer, they must be made available to the user through the scene description language.

My solution is a 'C'-like language called 'Rayd.' It satisfies all the criteria given above. A full BNF (Backus-Naur Form) [Aho 86] description of RAYD is given in Appendix ~~B~~.A

The decision to construct a new scene description language was not made lightly. There were, at the time, already existing scene description languages such as PHIGS+ and Renderman. NFF, however was not flexible enough for the demands of the Hypercube Ray Tracer, and Renderman was not yet publicly available. This drove me to the only other alternative available – writing my own.

Intersection Method

Intersection method and ODB organization is the next major design choice to address. Several very different intersection algorithms have been presented, and all work well. Only the methods of Fujimoto and Kay were available when this decision was made, therefore, we shall concentrate on them. The method in Fujimoto (ARTS -- Accelerated Ray Tracing System) is a space subdivision technique based on the octree. The octree is traversed using a 3DDDA. (3 Dimensional Digital Differential Analyzer) ARTS claims good results, but the octree traversal algorithm is complex, and the ODB is split up into a large number of nondisjoint units. It would be highly desirable if the ODB could be easily split up over the nodes of a parallel processor. This does not appear to be the case for ARTS' ODB organization.

The method of Kay is an object subdivision technique based on a hierarchy. The hierarchy is traversed by an efficient algorithm already given in chapter 2. Kay's method also includes a new, and more efficient type of bounding volume. Excellent results are obtained, and the hierarchy may be distributed by subtrees across processing nodes if need be. Kay's algorithm 1) keeps the ODB in a form that is easily distributable, 2) has better performance than ARTS, 3) is relatively easy to implement, and 4) gives us a more efficient bounding volume. Any one of these reasons is enough to choose Kay's algorithm.

Hierarchy Generation

Choosing a method for generating the hierarchy to be used by the Kay algorithm was a relatively simple task. The research presented in [Goldsmith 87] presents definitive comparisons between three hierarchy construction methods: model order construction, median-cut construction, and a heuristic tree search method. The heuristic tree search method, henceforth called the Goldsmith method, yielded the best overall results. The Goldsmith method is O(n log n) in the number of objects in the scene. For purposes of comparison, the median cut method is also O(n log n), but produces a hierarchy inferior to the Goldsmith method. The model order construction is O(n), but can produce a very poor hierarchy. Since the Goldsmith method is O(n log n), it is in the same asymptotic complexity class with the intersection algorithm. From this we can deduce that the Goldsmith method is probably economical in terms of the time it will take to construct the hierarchy. Clearly, the hierarchy construction time must not exceed the amount of time that will be saved by using such a hierarchy, otherwise, there will be a net loss in performance. For all of the aforementioned reasons, I chose to implement the Goldsmith method for the Hypercube Ray Tracer. Hierarchy generation is performed as a serial preprocess, however, not in parallel. This is because the hierarchy needs to be constructed only once, and it is the same for every node in the hypercube. The hierarchy construction could, conceivably, be parallelized, but it would not contribute significantly to the ray tracer's performance. (The hierarchy constructions would be faster, but the ray tracer would not) For this reason, I opted for the slightly easier option of a serial hierarchy constructor.

Primitives

Choice of primitives, as mentioned before, will heavily influence the relative performance of the Hypercube Ray Tracer. If the primitives are too simple, the ray tracer's flexibility will suffer. If they are too complex, then ray-primitive intersections will dominate rendering time, and absolute performance will suffer. I decided on a representative mix of primitive complexities. The Hypercube Ray Tracer supports spheres, cylinders, cubes, polygonal prisms, and convex superquadric ellipsoids. All may be scaled arbitrarily along each of the three coordinate axes. The sphere, cylinder, and cube provide primitives which are easily intersected against. The prism and superquadric provide more flexible primitives which are harder to intersect. As per the design goals of the Hypercube Ray Tracer, the software has been structured such that new primitives may be added with a minimum of effort.

Constructive Solid Geometry

To complement these primitives and enhance their flexibility, I decided to implement CSG. Publications dealing with ray tracing CSG objects, however, are very scarce. At least two have undertaken this task, with varied results [Roth 82, Yossef 86]. Roth's CSG intersection method is $O(n^2)$ in the number of objects in the CSG construct -- very costly indeed. Yoseef's method is $O(n)$ in the number of objects, but the algorithm given is complex and based on a form of space subdivision. Since I had already decided on an object subdivision approach for the Hypercube Ray Tracer, this was a major stumbling block. Therefore, I developed my own CSG representation and intersection methods. My CSG intersection method is called the "truth table" method.

CSG is a boolean expression on a number of volumes. Let us consider, for a moment, all the primitives in a CSG construct without the boolean expression applied to them. If we now consider the path of a ray through this collection of primitives, we can envision many intersection points along its path with the various primitives. Some of these intersection points, however, are not a part of the CSG object; they are excluded by the boolean expression performed on the primitive volumes. If a way could be found to detect these "false" intersection points, they could be thrown out of the intersection process as they were found. The resulting algorithm would then find only the intersection points which actually lay on the surface of the CSG object.

Any 3D point may be tested for membership in the CSG volume by first determining whether it lies within each of the primitive volumes, and then evaluating the boolean CSG expression with these truth values substituted. As a ray passes through the CSG primitives, it encounters the surfaces of zero or more of these primitives. The CSG membership of each of these *surface* intersections must be determined. This is not quite the same as testing an arbitrary 3D point, since the intersection point in question lies on the surface of a primitive. The solution is to test two points, one on either side of the surface. Evaluating the CSG expression based on the positions of these two points, we obtain two answers. Either both points are inside the CSG object, both are outside, or one is inside and one is outside. Clearly, if both points are outside the CSG object then the surface intersection is not on the surface of the CSG object. If one point is inside and one is outside, then the surface intersection is on the CSG object. This, then, is a CSG intersection point that we wish to keep. There is one other case: both points inside. This case could only happen if a ray were being refracted through a transparent CSG construct. Since a visible interior interface would be undesirable, we must exclude this intersection point. Reviewing the four cases and the desired truth values of each, we find that the exclusive OR function on the two test points yields the proper truth value of the surface intersection point.

Nothing has yet been said about how to determine whether or not a 3D point lies inside a primitive. Rather than evaluating a costly inside/outside function, we may make one simple observation. When a ray encounters the surface of an object, it changes its "insidedness" state with respect to that object. When a ray in free space hits the surface of a sphere, it goes from being outside the sphere to being inside the sphere. Since a ray may enter an arbitrary number of objects (CSG objects or primitives), each ray has a list associated with it specifying which objects it currently lies within. In this way, there is very little cost associated with keeping track of which objects a ray lies within. If the ray has to be checked against each object in the CSG construct each time the CSG expression were evaluated, then the CSG intersection procedure would become $O(n^2)$.

A method for representing the CSG expression must also be specified. One way to represent a boolean function is by using a parse tree [Aho 86]. A parse tree can be easily evaluated by interpreting it with a recursive traversal algorithm. Although this approach is simple, it is not very fast, nor is the representation particularly compact. Another way to represent a boolean function is by a truth table. A truth table associates each possible input state with a corresponding output value. If we assume an order for the input states, then we need only store the corresponding output values. Since each of these output values is simply a true or false, they may be stored as a bit vector.

A truth table for n possible inputs has 2^n entries, so a complex expression may require a truth table of considerable size. If we keep the number of objects participating in a CSG expression down to a reasonable number (8 or so), then the truth table is of reasonable size (256 bits). Note the use of the word "objects"; a CSG object may contain other CSG objects as well as primitives. In this way, generality is maintained, and CSG objects may encompass any number of primitives.

Extending the truth table intersection method to a hierarchy of CSG constructs is trivial. Since each CSG node has its own CSG expression, an intersection with any of its children may be tested for validity simply by consulting the truth table. The following original algorithm implements the Hierarchical Truth Table Method (HTTM).

```
Given an intersection point IP on primitive P
Let C be the CSG node to which P belongs
While C is a CSG node
   If IP is not a valid CSG intersection for C
      Return FALSE
   Endif
   P = C
   C = parent of C
Endwhile
Return TRUE
```

Algorithm 3: HTTM Candidate IP Membership Test

The Hierarchical Truth Table Method traverses up the CSG hierarchy checking for CSG membership at each level. If the intersection point in question is found to be invalid at any point in the traversal, the intersection fails. If the intersection point clears all CSG nodes up to and including the root, it is a valid intersection for the CSG hierarchy.

Using HTTM, a prospective intersection point may be quickly checked for validity. HTTM is O(n) in the number of objects in the CSG construct for purposes of intersection. This is clearly superior to the method of [Roth 82], which is $O(n^2)$. HTTM is at least as fast of the method of [Yossef 86] which is O(n). HTTM has the extra advantage of independence from ODB organization strategy; it will work with any object or space subdivision scheme.

Antialiasing and Distributed Ray Tracing

Of the two antialiasing methods presented in chapter 2, only the method of Lee is suitable for our parallel environment. (It would work in a shared-memory parallel computer) It is also extensible to distributed ray tracing. Since DRT does not affect the parallelism of the Hypercube Ray Tracer in any way, it was omitted from the initial design. The Lee antialiasing method, however was implemented since its implementation was trivial.

Parallel Decomposition

With all of the strictly ray tracing specific issues decided, we now turn to the parallel issues. Of primary importance in the parallelization process is the problem decomposition. What, in the ray tracing milieu, shall be decomposed across the nodes of the hypercube? There are two basic decomposition methods in classical parallel processing; domain decomposition and control decomposition. Domain decomposition focuses on distributing an algorithm's central data structure evenly among computing nodes. It is used whenever the type and amount of processing to be done on each datum is roughly the same. This paradigm is used extensively in finite element analysis and matrix methods. Control decomposition focuses on distributing an algorithm's control structure. It is used in programs which have irregular data structures or an unpredictable flow of control.

Which decomposition should be used on ray tracing? First, let us try to identify the central data structure in ray tracing. A first guess might be the ODB. Although the ODB is certainly the largest data structure, each node must have access to the whole database. Therefore, if the ODB is statically decomposed, the nodes will not be able to complete the ray tracing operation without extensive communication. In the distributed memory environment of the iPSC/2, communication is a relatively expensive operation. Performance, to say nothing of program simplicity, would suffer because of this decomposition. Another data structure must be found to decompose. As discussed earlier, each pixel in the frame buffer is independent of all other pixels. By virtue of this property alone, the frame buffer is an ideal choice for decomposition. Each node may be set to ray trace a portion of the final image, and the pieces later recombined to form the complete image.

This type of image decomposition leads to several very important points of overall program structure. If each node is going to ray trace a portion of the image, then it must execute a functionally complete ray tracing program. If each node is going to function as a complete ray tracer, then it must have access to the entire ODB. This does not necessarily mean that each node must have a complete copy of the ODB. However, if each node did not have a copy, then they would have to communicate with other nodes to get the data they needed. Due to the complexity of a distributed ODB, the initial version of the Hypercube Ray Tracer stores a complete copy of the ODB on every node. Distributed ODB extensions to the Hypercube Ray Tracer are discussed in chapter 5.

There is a vital question about image decomposition that needs to be addressed: in what manner will the image plane be divided? Should it be divided by pixels, rasters, blocks, strips, or some other method? Just because nodes are assigned equal size parts of the image does not mean that they will take the same length of time to render them! A little thought will disclose that the time taken to render a pixel is highly variable depending on what kind of objects the primary ray and its subsequent secondary rays intersect. A ray that misses everything will take much less time to shade than a ray that hits a transparent or reflective surface. Furthermore, if a node's share of an image is largely composed of "hard" pixels, it will take much longer to complete than a node tracing "easy" pixels. This can lead to a poor load balance and low efficiency. If a decomposition could be found that allocated equal numbers of hard and easy pixels to all nodes, efficiency would be maximized. One such decomposition is known as the "comb" decomposition because the portion of an image assigned to a particular node resembles the teeth of a comb. Starting with the first raster of an image, rasters are assigned to successive nodes until a raster has been allocated to every node. The process continues until all rasters have been allocated. Experimental evidence from the Hypercube Ray Tracer shows that this decomposition gives a very good load balance, and an excellent efficiency. See chapter 6 for specifics.

Summary of Design Choices

At this point, several things about the Hypercube Ray Tracer's overall program structure are decided. Each node in the hypercube runs a complete ray tracer with a complete copy of the ODB. Later modifications will distribute the ODB among the nodes. The "comb" image decomposition is used to divide the ray tracing load evenly among the nodes. Both of these are parallel concerns, and by no means constitute the complete Hypercube Ray Tracer.

In addition to the node programs, a host-based program is responsible for loading the node programs, sending the ODB to them, and reassembling the completed image when the nodes finish. Once the final image is complete, it is written in standard HIP (Hypercube Image Processor) format [Daniel 89]. HIP is an iPSC/2 based image processing program authored at Oklahoma State University. Images may be viewed, printed, or further processed easily using HIP. Figure 3 shows the overall flow of control in the ray tracing process.

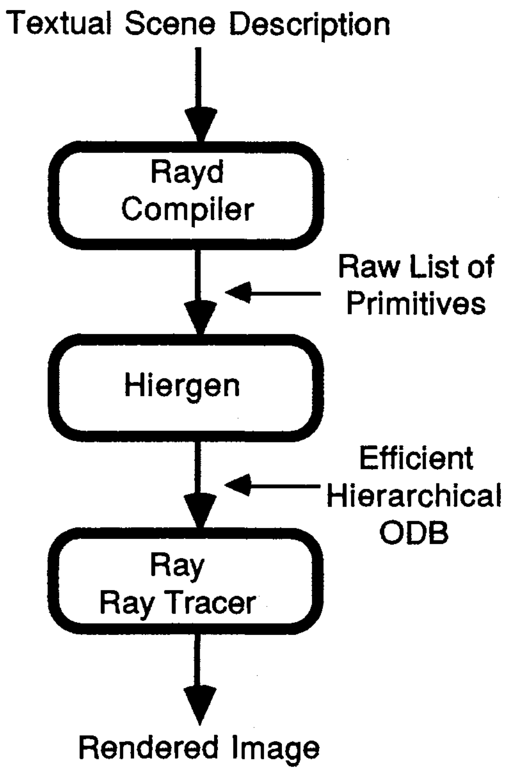


Figure 3: Flow of Control in Hypercube Ray Tracer

CHAPTER IV

COMPONENTS OF THE HYPERCUBE RAY TRACER

Rayd – The Scene Description Compiler

Rayd is a language and a compiler. Both are discussed here in turn. As stated earlier, Rayd must be intelligible, flexible, and expandable. Toward this end, the Rayd language borrows many features from the 'C' language. Among these features are block structure and syntax elements such as braces and semicolons. The Rayd compiler uses two standard UNIX^{TM} compiler building tools: lex and yacc. Once the Rayd compiler has parsed an entire scene description, it produces a complete but unstructured list of primitives. First, we will discuss the Rayd scene description language.

The Rayd Language

A Rayd scene description consists of three sections: object definitions, a single scene definition, and a single observer definition. Object definitions may be hierarchical. The following is a simple Rayd description.

```
/* Define two spheres by the name 'balls' */
define object balls {
   /* Big green ball */
   object sphere (
      position  = (9.0 0.2 0.2);     /* (x,y,z) position */
      color    = (0.4 1.0 0.1);      /* RGB surface color */
      size      = (0.9 0.9 0.9);     /* (x,y,z) size */
   );
   /* Small red ball */
   object sphere (
      position = (10.0 -0.8 0.0);
      color    = ( 0.8  0.0 0.2);
      size     = ( 0.5  0.5 0.5);
   );
};

/* Define a single light source by the name 'lumen' */
define light lumen {
   light (
      position  = (1.0 1.0 2.0);
      color      = (1.0 0.5 1.0);
      brightness = 10;
      size       = (0.3 0.3 0.3);
   );
};

/* All objects in the scene go here.  The balls and the light. */
define scene {
   object balls (
      position = (0 0 0);
   );
   light lumen (
      position = (0 0 0);
   );
};

/* Define the observer's position, viewing direction, etc. for */
/* rendering.                                                   */
define observer {
   position = (5 0 0);     /* Eye position.                 */
   viewdir  = (74 -11 0);  /* Direction we are looking.     */
   updir = (0 0 1);        /* Dir. toward "top" of viewplane.*/
   flen  = 5;              /* Dist. from eye to viewplane.   */
   vrectsize = (0.4 0.4);  /* Viewplane size in world coords.*/
   recursion   = 8;        /* Max. recursion depth of a ray. */
   resolution = (512 512); /* Pixel resolution of image.     */
};
```

Figure 4: Sample Rayd Scene Specification

Each section is clearly visible in the above example. Note that a light source definition is also an object definition. The only restrictions on the order of definitions is that to invoke an object, such as in the scene definition or in another object definition, it must have already been defined. For this reason, the scene definition is usually near the end of a Rayd scene description.

Braces are used to enclose the bodies of all definitions, and parentheses enclose object invocations and parameter lists. A semicolon is mandatory at the end of all brace or parenthesis enclosed blocks. Semicolons are also mandatory following all parameters, pseudodefinitions, and compiler directives. All of these constructs are defined in the next section.

Object Definitions

Member objects in an object definition have several parameters that may be set. The number and types of these parameters vary according to what type the member object is. Generally, these parameters take the form of a single number, a tuple, or triple. Tuples and triples are enclosed in parentheses and elements are not comma separated. If the member object is not a primitive, then only its position, size, and rotation may be manipulated. If the member object is a primitive then these along with surface parameters and other type dependent parameters may be changed. Standard surface parameters are color, specular reflectance coefficient, specular reflectance exponent, reflectivity, transmissivity, and refractive index. An example of type dependent parameters are the shape controlling exponents for a superquadric. A full list of these parameters is given with the Rayd BNF in Appendix A.

As far as Rayd is concerned, an object is an arbitrary collection of primitives and previously defined objects all referred to by a single name. In the above example, we saw the following structure:

```
define object balls {
   object sphere ( ... );
   object sphere ( ... );
};
```

This groups the two primitives (spheres in this case) into a single object called 'balls'. This new object has its own coordinate system and may be rotated, scaled, and translated at each future reference. For example, suppose we wanted to define an object consisting of three 'balls'. The definition might look like this:

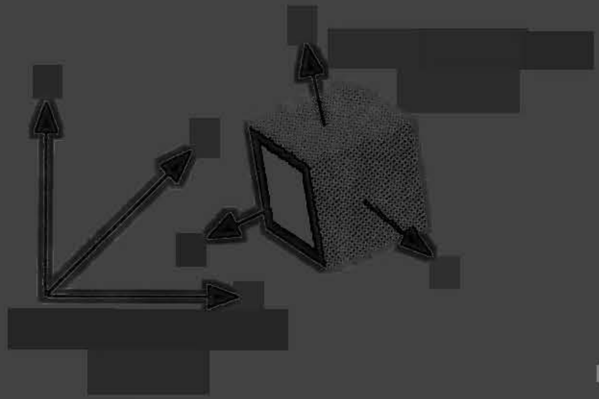
```
define object six_spheres {
   object balls (
      position = (4 3 2);
   );
   object balls (
      position = (3 6 0);
   );
   object balls (
      position = (0 0 0);
      size     = (2 1 1);
      rotation = (45 30 0);
   );
};
```

Figure 5: Rayd Specification for a Cluster of Six Balls

We now have an object, 'size_spheres' consisting of three 'balls'. The first two invocations of 'balls' simply place two copies at (4 3 2) and (3 6 0) respectively. No resizing or rotation is performed. Note that the position (4 3 2) is relative to the local coordinate system of the 'six_spheres' object (Figure 5).

Formatting is arbitrary in a Rayd description; the above example is formatted as it is for clarity. The third invocation of 'balls' is a bit different. It creates a 'balls' object scaled twice along the X axis, and then rotated 45 degrees about the X axis, and 30 degrees about the Y axis. Again, all scaling and rotations are done with respect to the named object's local

[redacted]



[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

Scene Definition

The scene definition is very much like an object definition in form, but much different in function. The function of the scene definition is to let the Rayd compiler know which objects are part of the final scene to be rendered. Not all of the objects defined may be intended to be part of the final scene. Indeed, when a high level object is constructed from many lower level objects, one may only want to render the high level object. All of its constituent objects are irrelevant. The scene definition provides a convenient way of resolving this ambiguity. In general, the structure of the scene definition is as follows:

```
define scene {
   ... objects and lights ...
};
```

Objects and lights are specified in the format already given. At least one object or light source must be specified in the scene definition. There is no upper limit to the number of lights and objects that may be specified.

Observer Definition

Once all of the objects and light sources to be rendered are defined, only the viewpoint remains to be specified. This is the function of the observer definition. It includes not only the position of the observer, but the viewing direction, viewplane size and orientation, and other miscellaneous rendering information. A complete observer definition is given in Figure ~~2~~4. All of the listed parameters are mandatory.

Pseudodefinitions and Compiler Directives

A number of other features are included in Rayd for convenience. One is the ability of attach a textual label to an RGB color representation. This is done in the following manner:

define color cyan = (0 1 1);

A color definition can appear anywhere outside other definitions. Thereafter, the label 'cyan' may be used anywhere the RGB triple (0 1 1) would otherwise have been used. Similarly, the surface definition allows the user to attach a textual label to a set of surface properties. The following is a hypothetical surface definition:

```
define surface shiny_cyan {
   color   = cyan;  /* Surface color   */
   reflect = .6;    /* Amount of light reflected*/
   spec    = .5;    /* Specular reflection coeff.    */
   phong   = 20;    /* Specular reflection exponent. */
};
```

After the surface has been defined, it may be invoked in any primitive definition by using the following form:

```
define object shiny_ball {
   object sphere (
      position = (1 2 3);
      surface  = shiny_cyan;
   );
};
```

In this way, the surface properties of a large number of objects may be changed simply by modifying one surface definition in the Rayd file. Also, the size of the scene description file is frequently reduced since a large number of primitives usually have the same surface characteristics.

Another convenient feature of the Rayd language is the ability to include other Rayd files. The ability to bring in outside source files enables the user to construct libraries of frequently used objects. It also lets the user logically organize a very large scene description into multiple files. The syntax of the include directive is:

```
include "filename";
```

Include directives may be nested up to eight levels deep, but they may not form a loop.

The Rayd Compiler

The Rayd compiler works as a two-phase process. Phase one parses the Rayd description and constructs an internal representation of the scene. Phase two traverses the internal data structure and produces Ray Tracer format primitives as it goes. The exact representation of primitives will be discussed at length later.

As mentioned earlier, the Rayd compiler is constructed using the UNIX™ tools lex and yacc. Lex reads a file describing a number of lexicographic units, or tokens, that are to be recognized. It produces a program which scans a stream of characters looking for the tokens. This program is called a lexical analyzer. The particular lexical analyzer used in the Rayd compiler recognizes all legal keywords and strings for the Rayd language. The lexical analyzer effectively transforms a scene description into a stream of tokens which are parsed by the next stage of the Rayd compiler – the parser. See Figure 7 for an overall view of the Rayd compiler's internal structure.

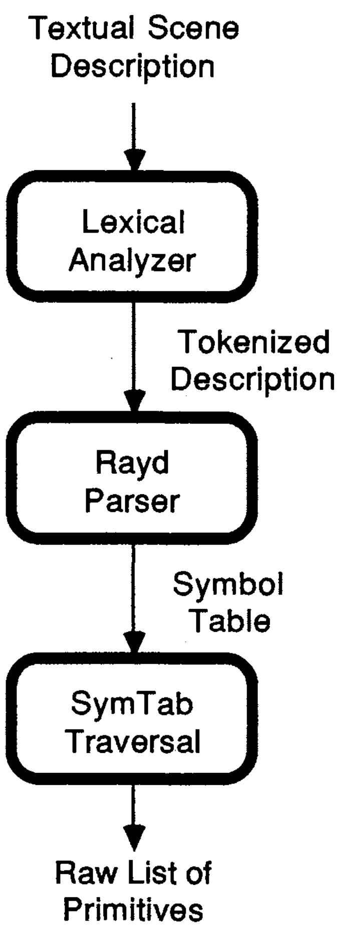


Figure 7: Internal Block Diagram of Rayd Compiler

Just as there are grammatical rules for the English language, there are also rules of grammar for the Rayd language. These rules can be formulated in standard BNF notation. The full BNF description of the Ray language can be found in Appendix A. This BNF description is read by the yacc parser generator. Yacc stands for "Yet Another Compiler Compiler." After reading the input grammar, Yacc generates a parsing program which must be compiled and linked against the lexical analyzer built by Lex. Actions, written in C, may be included with each rule in the grammar. The actions are executed when the parser recognizes the construct represented by the associated rule. For example, there is a rule in the Rayd BNF description which recognizes a triple of numbers enclosed in parentheses. The action code associated with this rule takes the three numbers in the triple and packs them into an array for future reference by other rules.

Overall, these actions comprise the actual compiler part of the Rayd compiler. Their ultimate goal is to construct a data structure of primitives and the relationships between objects. This data structure is called the "symbol table," and each element comprising it is called a "symbol." A symbol may contain one of many different things. Table 1 is a list of the possible types of symbols.

TABLE 1
SYMBOL TABLE ENTRY TYPES

Type	Description
surface	The result of a surface definition, this symbol stores all of the surface properties associated with the specified name. This symbol is only found at the root level of the symbol table.
primitive	This symbol contains a full description of one primitive. Stored here are the type and dimensions of the primitive as well as its orientation and name as specified in the scene description. Surface characteristics are also stored here.
macro	This symbol is a link to another object. It implements the hierarchical structure of compound objects. Stored here are position, rotation, and size of the object that it points to. If the this construct is a CSG object, the CSG expression is also stored in this symbol.
light	Points to a list of light_elements. Similar to a macro symbol, except a light symbol can only point to light_elements. Position and sizing information is stored in the symbol to be applied to the component light_elements. This symbol is only found at the root level of the symbol table.
light_element	This is a special type of primitive that is modeled as a light emitter. It is assumed to be spherical. Position, size, orientation, color, and intensity are stored in this symbol.

TABLE 1 (Continued)

Type	Description
scene	Similar to the light type, this type of symbol points to a list of scene_elements. This symbol implements the scene definition section of a Rayd scene description. This symbol is only found at the root level of the symbol table.
scene_element	This symbol is very similar to a macro symbol in that it points to a list of primitives or other macros. Each scene_element represents one object specified in the scene definition section.
ambient	This symbol stores the ambient light intensity specified in the Ray scene description. This symbol is only found at the root level of the symbol table.
observer	This symbol store all of the observer specific parameters. It corresponds with the observer definition section of a Rayd scene description. This symbol is only found at the root level of the symbol table.
color	The result of a color definition, this symbol stores the name of the color, and its RGB representation. This symbol is only found at the root level of the symbol table.

One important thing needs to be stressed regarding the properties of the macro and scene_element symbols. Each of them has associated with it position, size, and rotation (orientation) information. This information applies to the symbol's target *in addition to* any such information stored there, not *in place of* it. Note that the target may be a compound object. In this case, the geometrical manipulation applies to *all* of the object's constituents be they primitives or other compound objects. In this way, multiple translation, scaling, and rotations are possible. This is the way a local coordinate system is imposed on each primitive and hierarchical object.

Symbol Table Traversal

Once the symbol table has been constructed, it must be converted into the list of primitives it represents. The only objects which must be converted are the ones pointed to by scene_element symbols. All of these symbols are either primitives or macros. Since the linkage among the macro symbols is acyclic, a depth first traversal is ideal [Reingold 83]. Even though a symbol may be pointed to by more than one macro symbol, the traversal will still function correctly since the graph is acyclic.

As the graph is traversed, it is also necessary to calculate the primitives' positions, sizes, and orientations. Such calculations are greatly simplified by using geometric transformation matrices [Foley 84]. A theoretical discussion of geometric transformation matrices is beyond the scope of this thesis, and the reader is referred to [Foley 84] for a tutorial. The Hypercube Ray Tracer requires two transformation matrices associated with each primitive. One transforms a ray or point in global coordinates into a coordinate system centered on the unit primitive. This is called the global to local transform. The inverse of this matrix transforms a ray or point in object local coordinates and transforms it into global coordinates. This is called the local to global transform. These two matrices may be calculated as the hierarchy is traversed. The following is the pseudocode representation of the graph traversal:

```
Given a pointer S to a symbol
Save the aggregate transformation
Compose aggregate transformation with S
If S is a macro or scene_element
   For each symbol P under S
      Recurse for P
   Endfor
Else
   Transform S by the aggregate transformation
   Write out primitive S in standard format
Endif
Restore the aggregate transformation
Return
```

Algorithm 4: Rayd Symbol Table Traversal Algorithm

Now that the scene primitives can be generated from the symbol table, a suitable format is necessary for their storage. Not only must the primitive data structures themselves be written out, but also any CSG tree information. The following information is kept in each primitive data structure:

TABLE 2

FIELDS IN PRIMITIVE DATA STRUCTURE

Field	Description
1.	What type of primitive this object is. (Sphere, prism, etc.)
2.	A flag which is true if this primitive is a light source.
3.	Unique identifier associated with this primitive.
4.	The primitive's (x,y,z) global position.
5.	The primitive's local (x,y,z) sizes.
6.	RGB surface color.
7.	Specular reflectance coefficient and exponent.
8.	Transmittance of primitive's surface. 0 is opaque, 1 is transparent.
9.	Reflectance of surface. 0 = no reflected light,1 = mirrorlike reflection.
10.	Primitive's refractive index, if transparent.
11.	Local-to-Global transform matrix.
12.	Global-to-Local transform matrix.
13.	Bounding volume.
14.	Optional shape parameters is primitive is a superquadric
15.	Pointer to a list of 2D points if primitive is a prism.

Note that no information pertaining to CSG membership is kept with each primitive. Since the CSG truth table is of significant size relative to that of the primitive data structure, much memory would be wasted in non-CSG primitives. For this reason, CSG objects are stored as a tree with a CSG node at the root, and primitives or other CSG trees hanging below it. Just as primitives are nodes in the ODB hierarchy, so are CSG nodes. A CSG node contains the following information:

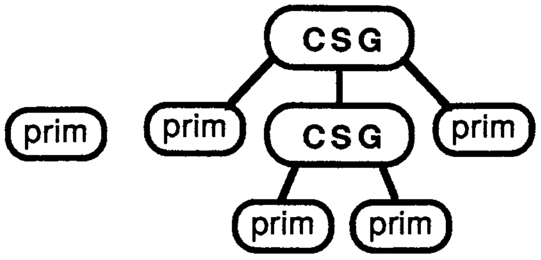
TABLE 3

FIELDS IN CSG NODE DATA STRUCTURE

Type	Description
1.	References to the objects in this CSG construct
2.	Bounding volume around the whole CSG construct.
3.	A unique ID associated with this CSG node.
4.	CSG truth table.
5.	Pointer to CSG ancestor, if this node is the child of a CSG node.

Enough information is now known about the primitives' data structures and CSG data structures to proceed. The output from the Rayd compiler begins with the ambient light intensity and the observer. It continues with the list of primitives and CSG objects. In this list, primitives are preceded by a special marker. This strategy facilitates easier interpretation of the file. CSG objects, being hierarchical in nature, have a slightly different format. They are preceded by an OPENCSG marker plus the CSG truth table. Following these come the CSG node's children, be they primitives or other CSG trees. Closing the CSG node is a CLOSECSG marker. See Figure 8 for an example of this structure. Note the nested use of the OPENCSG – CLOSECSG construct in the example.

```
Object Database Structure:
```



```
File Format: PRIM primitive OPENCSG csg-info PRIM primitive
             OPENCSG csg-info PRIM primitive PRIM primitive
             CLOSECSG PRIM primitive CLOSECSG
```

Figure 8: File Format Example From Rayd Compiler

We have now completely described the functionality of the Rayd compiler. The scene description is tokenized by the lexical analyzer and passed to the parser. The parser reads these tokens, determines their validity, and constructs a hierarchical symbol table from them. This symbol table is then traversed and an unstructured list of primitives and CSG trees is generated along with the observer parameters.

Hiergen – The ODB Hierarchy Constructor

Hiergen has one function only -- to take the list of primitives generated by the Rayd compiler and organize them into an efficient hierarchy that can be used by the Hypercube Ray Tracer. Before we launch into the functionality of Hiergen, let us first turn our attention to the elements in such a hierarchy.

Elements of the Hierarchy

In all subsequent discussion, the word 'node' will be used to refer to any type of node in a Ray Tracer hierarchy. This includes primitives, CSG nodes, and the yet-to-be-introduced 'Hnode.' Leaf hierarchy nodes will always be primitives, and the body is composed of CSG nodes and Hnodes. The function of Hnodes are to act as linkage nodes in the hierarchy. Each Hnode contains the following information:

TABLE 4

FIELDS IN HNODE DATA STRUCTURE

Type	Description
1.	Pointers to 1 to 8 subtrees.
2.	A unique ID.
3.	A bounding volume enclosing the whole subtree.

The decision was made early on for all nodes to have the same maximum branching ratio. This greatly simplifies hierarchy traversal by eliminating special-case nodes. As the branching factor grows, the CSG truth table size becomes very large. As the branching factor shrinks, fewer subtrees can be culled by the Kay algorithm at each node in the hierarchy. Experimental evidence has shown that a branching factor of 8 gives the best balance between CSG truth table size and intersection performance. See chapter 6 for performance versus branching factor data.

Organization of the Hierarchy

Figure 9 shows a very simple hierarchy. Toward its top, Hnodes give it its structure. At the leaf level, only primitives are present. Lying between the leaves and Hnodes are CSG nodes in certain locations. As previously discussed, the CSG nodes organize groups of primitives into CSG trees. These CSG trees are viewed as a single object.

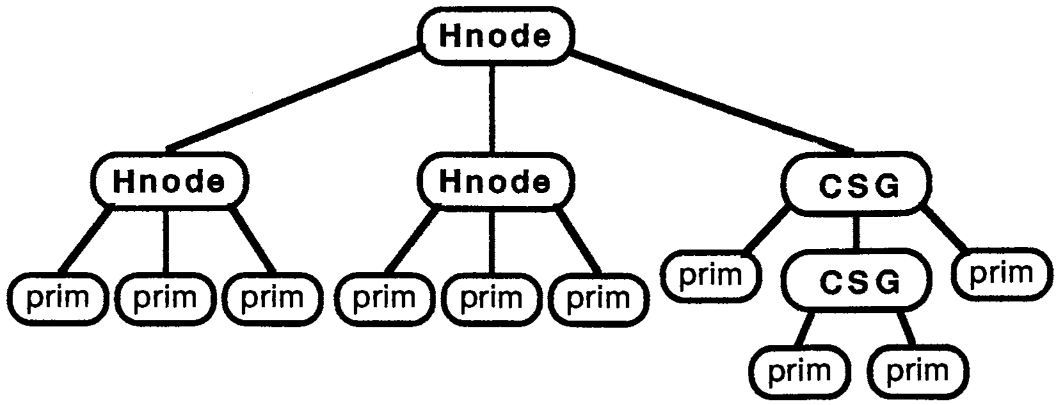


Figure 9: Organization of ODB Hierarchy

The Goldsmith Algorithm

As per design choice, the Goldsmith algorithm is used to construct the hierarchy from a list of primitives. The Goldsmith algorithm constructs the hierarchy one node at a time basing its placement of each object on a cost-based heuristic tree search. As each new node is considered, the existing hierarchy is searched to find the position where, if the new node were inserted, the increase in local bounding volume area would be minimized. Bounding volume area is used instead of volume because it is the silhouette of the bounding volume, as viewed from the ray, that determines how probable an intersection is. The following pseudocode implements the Goldsmith algorithm:

```
For each object O in the ODB
   Let N be the root hierarchy node
   While N is a full Hnode
      Select the child, C, of N whose bounding volume increases
        least when O is merged with C
      N = C
   Endwhile
   If N is an Hnode
      Insert O below N
      Merge O's BV with N's and continue to the hierarchy root
   Else
      /* N is a primitive or CSG tree. */
      Replace N with a new Hnode, H
      Insert N and O below H
      Merge BV's to the hierarchy root
   Endif
Endfor
```

Algorithm 5: Goldsmith Hierarchy Construction Algorithm

In the above code, a "full" Hnode is one in which there are the maximum number of children allowed by the branching ratio . The concept of merging bounding volumes is also in need of amplification. When a new child is inserted under an Hnode, that Hnode's bounding could possibly grow in size. The process of recomputing a new bounding volume that will fit around the node's new child is called "merging." All ancestors of the Hnode in question may be affected in the same way. Therefore, the merging process must be continued up the hierarchy until either the root is reached or an Hnode's bounding volume does not grow. CSG objects are thought of as a single object, and are added to the hierarchy as a single object.

Output File Format

Once Hiergen has constructed the hierarchy, it must be written to a file. An extension of the format used by the Rayd compiler is used. Two new marker types are introduced, the OPENHNODE mark, and the CLOSEHNODE mark. The Hnode is similar to the CSG node in that it is hierarchical. Its format in the output file is also very similar. Since this output file must contain all information about the hierarchy, all information contained in the Hnode must be written along with the OPENHNODE mark. This is true for CSG nodes as well. The following is a representation of what the Hiergen output file looks like for the structure in Figure 9.

```
Hierarchy :  See Figure 9
Output    :  OPENHNODE hnode-info
    OPENHNODE hnode-info PRIM prim PRIM prim PRIM prim CLOSEHNODE
    OPENHNODE hnode-info PRIM prim PRIM prim PRIM prim CLOSEHNODE
    OPENCSG csg-info PRIM prim OPENCSG csg-info PRIM prim PRIM
    prim CLOSECSG CLOSEHNODE
```

Figure 10: Hiergen File Format Example

This format provides the Hypercube Ray Tracer with full information about a hierarchy. It is also a convenient form from which to build the hierarchy. Each node can reconstruct the hierarchy from this stream quickly and easily. This will be discussed in the next section.

Ray – The Hypercube Ray Tracer

Ray is the third and final part of the Hypercube Ray Tracer. Ray is not a single program, but rather two programs – one that runs on the SRM, and one that runs concurrently on the nodes of the hypercube. Each program is responsible for very different parts of the rendering process. Both programs are described in detail below, but first, an overview of Ray is in order.

Software Architecture

The function of the host part of the Hypercube Ray Tracer is mainly that of an administrator. When the Ray Tracer is invoked, the host program is executed. It loads the node program onto all of the nodes, and downloads the ODB hierarchy to them. The node programs then have all the information necessary to render their respective portions of the final image. After the nodes have finished rendering, the host program uploads their portions of the frame buffer and reassembles them into a complete image. It then writes this image to disk, and shuts down. Figure 11 shows a schematic diagram of the flow of data and control about the various elements of the Hypercube Ray Tracer.

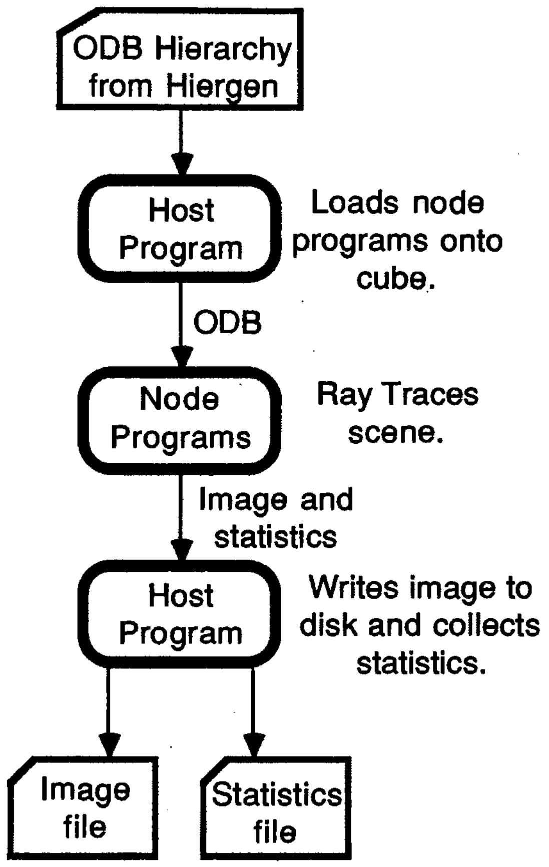


Figure 11: Flow of Data and Control

The function of the node program is somewhat simpler in overall concept, though not in execution. Its job is to accept the ODB hierarchy from the host, render its portion of the image, and return that portion to the host. Note that even though the same program runs on each node, they ray trace different portions of the image. Since a node program can figure out which node it is running on, it may base which portion it ray traces on its node number. The node program, here, is of the standard SPMD (Single Program Multiple

Data) programming model. The difference between the MIMD and SPMD software architectures is slight. In the MIMD model, each computing node may run a different program. In the SPMD model, each node runs the same program. This does not mean that SPMD programs must do exactly the same thing, only that the code is identical from node to node.

The Host Program

As stated above, the host part of Ray is the administrator of the whole process. The following is a pseudocode representation of the flow of control in the host program:

```
1.  Allocate a hypercube of nodes
2.  Download the node program to the hypercube
3.  Download the ODB to the hypercube
4.  Wait for the nodes to complete
5.  Upload and reassemble the frame buffer
6.  Upload timing and statistics from each node
7.  Write frame buffer to disk
8.  Print out the timing and statistics
9.  Deallocate the hypercube
```

The timing and statistics data is kept by each node as it is ray tracing the scene. It is useful for determining such things as load balance, and the efficiency of the intersection algorithm. Listed below in Table 5 are the various counts and times that each node accumulates.

TABLE 5

PERFORMANCE METRICS KEPT BY NODE PROGRAMS

Type	Description
1.	Number of floating point operations performed
2.	Time taken reconstructing the ODB
3.	Total rendering time
4.	Number of rays cast
5.	Number of rays that hit a primitive
6.	Number of BV intersections
7.	Number of primitive intersections

It can be easily see that floating point operations per second (FLOPS) can be computed from items 1 and 3 in Table 5. A "hit rate" metric can be obtained by dividing item 7 by item 5. This gives the fraction of all rays intersected against primitives that actually intersect. If the ray-ODB intersection algorithm performs poorly, this number will be small. The host program also sums up these metrics to form aggregate metrics. In this way, the performance of the whole hypercube can be measured.

The Node Program

Aside from the algorithms for ray intersection and shading model evaluation, the node program control structure is rather modest in complexity. Below is the pseudocode representation of the node program control flow:

```
Initialize data structures and variables
Download ODB from the host
For each responsible pixel, P
   Construct a ray from the observer through P on the viewplane
   Find the closest intersection point of ray with scene
   Evaluate the shading model, casting reflected, refracted
    and shadow rays if necessary.
   Move P's intensity (color) into the local frame buffer
Endfor
Send the local frame buffer to the host
Send node timing and statistics to the host
```

Algorithm 6: Node Program Pseudocode

A node's *responsible* pixels are those that lie in the rasters assigned to that node by the comb decomposition. These responsible pixels, once ray traced, are stored in a *local frame buffer* for later transmission back to the host program. The body of the `for` loop above can be thought of as the quintessential ray tracing algorithm. Note that it is a recursive algorithm – the shading model will frequently call for reflected and refracted rays to be traced. These rays are traced in exactly the same way as primary rays.

The ODB Reconstruction Algorithm

The ODB comes out of Hiergen in a special format, as discussed earlier. This data is relayed, unchanged, to the node programs by the host program. It is up to the node programs to reconstruct the ODB hierarchy from the Hiergen file format. The following original algorithm implements this ODB reconstitution:

```
Let RootNode be a pointer to the hierarchy root node
Initialize the current "hanging location" D = address of RootNode
Initialize the "hanging location" stack, HS, to empty
Initialize the CSG stack, CS, to empty
Initialize the Hnode stack, NS, to empty
Push a NULL onto HS
Push a NULL onto CS
While top of HS is not NULL
   Get a token, T, from INPUT
   Switch on T

      Case PRIMMARK     /* Got a primitive from INPUT */
         Allocate a new primitive pointed to by P
         Get primitive information from INPUT
         Place P at location pointed to by D
         Advance D to next child pointer

      Case OPENHNODE  /* Build an Hnode subtree */
         Allocate a new Hnode pointed to by H
         Get Hnode information from INPUT
         Push D onto HS
         Push H onto NS
         Let D = address of H's first child pointer

      Case OPENCSG/* Build a CSG subtree */
         Allocate a new CSG node pointed to by C
         Get CSG node information from INPUT
         Push D onto HS
         Push C onto CS
         Let D = address of C's first child pointer

      Case CLOSEHNODE   /* End an Hnode */
         Pop D from HS
         Pop H from NS
         Place H at location pointed to by D

      Case CLOSECSG     /* End a CSG subtree */
         Pop D from HS
         Pop C from CS
         Place C at location pointed to by D
         Set father pointer of C to top item on CS

   Endswitch
Endwhile
```

Algorithm 7: Hiergen File to ODB Reconstruction Algorithm

This algorithm reconstructs the ODB hierarchy one node at a time in the same order that it was traversed. Any subtree of the ODB may be packaged in the above format. This property will be useful when distributing the ODB. Furthermore, this packing method does not rely on information which cannot be transferred from one node to another, such as pointer values. This unpacking algorithm also has the advantage of being fast; since it is a simple tree copying operation, it is O(1) in the number of objects to be unpacked.

The Modified Kay Intersection Algorithm

The Kay algorithm for intersecting a ray with a homogeneous hierarchy was presented in Chapter 2, Algorithm 2. Now, it must be modified to work with the CSG subtrees found in the Hypercube Ray Tracer's hierarchy. This involves the addition of the Hierarchical Truth Table Method (HTTM) loop around the Kay intersection algorithm. In addition, CSG nodes must now be traversed as well as Hnodes. Given in pseudocode format below is the standard Kay algorithm with three modifications: it traverses both CSG nodes and Hnodes, it is able to intersect a ray with the ODB starting from some arbitrary point t_0 along the ray, and it returns a pointer to the father node of an object if it was part of a CSG construct. The second and third modifications are critical to the HTTM intersection algorithm which will be presented after the modified Kay algorithm. Only intersection points along the ray *beyond* the point t_0 will be reported by the modified Kay algorithm. The variable t_0 is used as a sliding starting point for primitive intersections. Initially, t_0 is 0, and the Kay intersection algorithm behaves normally. If the first intersection point found by the Kay algorithm is a false CSG intersection point, then the next intersection point beyond it must be checked, etc.

```
Given $t_0$, the minimum allowable intersection distance
Initialize heap to empty
Initialize $p_{near}$ = nil { Pointer to nearest primitive }
Initialize $p_{father}$ = nil { Pointer to father of primitive }
Initialize $t_{near}$ to infinity { Distance to closest primitive }
Insert hierarchy root node into heap
If recursion level of ray > MAX_RECURSION
   Return nil
Endif

While heap is not empty and distance to top node < $t_{near}$
   Extract candidate with closest int. distance from heap
   If the candidate is a primitive
      Compute ray-primitive intersection
   Endif
   If ray hits candidate and distance < $t_{near}$ and distance > $t_0$
      $t_{near}$ = distance
      $p_{near}$ = candidate
      $p_{father}$  = father of candidate
   Endif
   Else if candidate is an Hnode
      For each child of the candidate
         Compute ray-bounding volume intersection
         If the ray hits the bounding volume
            Insert the child and distance into the heap
         Endif
      Endfor
   Else   { It must be a CSG node }
      For each child of the candidate
         Compute ray-bounding volume intersection
         If the ray hits the bounding volume
            Insert the child, distance, and father of child node
              (candidate) into the heap
         Endif
      Endfor
   Endif
Endwhile
Return $p_{near}$, $t_{near}$, $p_{father}$
```

Algorithm 8: Inner Loop of the Modified Kay Intersection Algorithm

The outer loop of the modified Kay algorithm is not as tidy as the inner one. The inner loop's job is to find the closest primitive intersection point with a ray past some point t_0 on the ray. This *does not* involve any CSG membership evaluation. This is the outer loop's function. Prospective primitive intersection points found by the inner loop are examined by the outer loop for CSG membership. If the intersection point is found to be part of the CSG construct, then the point is accepted and the outer loop terminates with success. Otherwise, the point is discarded, and the inner loop is called again with a larger t_0. In this way, the outer loop slides the minimum intersection distance, t_0, outward from the ray origin until the first "real" intersection point is found. Algorithm 9 shows the outer loop of the modified Kay algorithm.

```
Given a ray
Let $t_0$ = 0
Let Foundint = FALSE
Do
   $t_{near}$= $t_0$
   { Find the next primitive intersection point.}
   Call Kay inner loop for closest intersection past $t_{near}$
      (Returns $t_{near}$, $p_{near}$)

   { If ray missed everything, then return failure. }
   If $p_{near}$ = nil
      Return nil
   Endif

   { If it is a non-CSG primitive, then return success. }
   If $p_{near}$ is not part of a CSG construct
      Return $p_{near}$, $t_{near}$
   Endif

   { Otherwise, we must traverse up the CSG hierarchy checking }
   { for validity at each CSG node.  }
   Let O = $p_{near}$
   Do
      If O is not valid for the CSG object
         Let Hitcsgobj = FALSE
      Else
         Let Hitcsgobj = TRUE
         O = father of O
      Endif
   While not at top of CSG hierarchy and Hitcsgobj = TRUE

   { If we made it all the way to the top of the CSG hierarchy }
   { and the intersection point was a member of every CSG }
   { object, then the primitive intersection point is valid. }
   If at top of CSG hierarchy and Hitcsgobj = TRUE
      Foundint = TRUE
   Else
      $t_0$ = $t_{near}$
   Endif
While Foundint = FALSE
Return $p_{near}$, $t_{near}$
```

Algorithm 9: Outer Loop of Modified Kay Algorithm

The Ray Tracer uses many algorithms to perform its function: the modified Kay intersection algorithm, primitive intersection algorithms, the Phong shading model, geometrical transformations, the Goldsmith hierarchy construction algorithm, the HTTM CSG intersection algorithm, as well as others. Most of these algorithms have already been discussed in preceding sections. The algorithms presented in this chapter tie the basic ones together into a complete ray tracing system.

CHAPTER V

THE DISTRIBUTED OBJECT DATABASE

Rationale

In the preceding chapters, the development of a functionally complete parallel ray tracing system is documented. It has one key shortcoming, however. Each computing node must be able to hold the entire ODB no matter how large or small it is. In a hypercube with n nodes, n copies of the ODB are stored – a gross waste of precious memory. A way needs to be found to drastically reduce this waste of memory if large numbers of objects are to be ray traced.

The ODB is largest data structure in the Ray Tracer for all but the most trivial scenes. It, therefore, needs to be considered for parallel decomposition. On an iPSC/2 with 4 megabyte (MB) nodes, about 3.3 MB is available for ODB storage. The balance of memory is taken up in operating system (400 KB), the Ray Tracer node program (200 KB), and overhead data structures (100 KB). A primitive takes 240 bytes to store, an Hnode 80 bytes, and a CSG node 116 bytes. In an 8-ary tree, each subsequent level has 8n nodes where n is the number of nodes on the higher level. In a b-ary tree with L levels, there are

$$N = \sum_{i=1}^{L-1} b^i \qquad (2)$$

total nodes above the leaf level. Knowing that there are b^L total nodes in a complete b-ary tree, we can compute P, the fraction of nodes above the leaf level as

$$P = \lim_{L \to \infty} \frac{N}{b^L} = \lim_{L \to \infty} \sum_{i=1}^{L-1} b^{i-L} = \sum_{i=1}^{\infty} b^i \qquad (3)$$

By substituting b=8 into (3), we find that P ≈ 14.3% for a general ODB hierarchy. We may now formulate an expression for the total memory, M, taken by a hierarchy of n primitives.

$$M = S_p\, n + P\, n\, (f\, S_h + (1-f)\, S_c) \qquad (4)$$

Where:

S_p	= size of a primitive
S_h	= size of an Hnode
S_c	= size of a CSG node
P	= fraction of nodes above the leaf level
f	= fraction of hierarchical nodes which are Hnodes
n	= number of primitives
M	= total memory taken by the hierarchy

If we now solve for n, we have an expression in M, the available node memory, that gives how many primitives that node may store.

$$n = \frac{M}{S_p + P\ (\ f\ S_h + (1-f)\ S_c)} \qquad (5)$$

Evaluating (5) for M=3.3 MB and f=0.9, we obtain n = 13733. (Using f=.9 is an empirical estimate from the scenes rendered so far, and in any case, makes little difference in the final answer due to the closeness of Sh and Sc and the small magnitude of P) This means that each 4 MB node can hold an ODB of 13733 primitives – only a moderately complex image by today's standards. Moreover, if p processors could each hold n different objects, then a 16 node hypercube could hold an ODB of n p = 219728 primitives! Note that this figure does take into account any overhead for the hierarchy infrastructure. As will become apparent in Chapter 6, a significant number of duplicate primitives must be stored across the hypercube for performance reasons. This duplication, though not nearly as severe as duplicating the entire ODB, reduces the total effective number of primitives that can be stored in the hypercube ensemble.

As more processors are added, the maximum number of primitives in the ODB would increase rather than stay the same. Thus, the computer's parallelism could be used to increase the number of primitives as well as the speed at which they are rendered. This is precisely the goal of ODB distribution.

What Has Gone Before

Salmon and Goldsmith [Goldsmith 88] decomposed their ODB in a static manner across processing nodes. Parts of it were unable to move from one processor to another. As their ray tracer was implemented on a finer grained machine than the iPSC/2, this was the most attractive arrangement for them. However, this decomposition has a number of highly undesirable features. The most serious of these is the issue of intersection. What happens when the intersection process can no longer proceed on a given processor? When this happens, the intersection state and the ray must be shipped off to a processor that contains the correct part of the ODB. The ray may be shuttled between many processors before it finally completes the intersection process. All of this internode communication is costly and greatly slows down the ray tracing process. Furthermore, nodes which contain frequently queried primitives will constantly be swamped with rays from other nodes. This leads to a potentially poor load balance unless care is taken to distribute the ODB in such a manner that the "popular primitives" are evenly distributed. If these popular primitives are in close spatial proximity to one another and they are split up onto different nodes, then more ray swapping traffic will result. This springs directly from the searching nature of the intersection process.

Salmon and Goldsmith chose a static ODB decomposition with swappable rays. Their method takes no natural advantage of the coherence with which the ODB is queried by the intersection process. As a result, performance suffers [Goldsmith 88].

A Fresh Look at ODB Decomposition

With the coarser grained architecture of the iPSC/2 comes the freedom to experiment with a different decomposition method. I have chosen a dynamic ODB decomposition where primitives are traded between nodes rather than rays. Initially, the ODB is split evenly across the nodes, just as with Salmon and Goldsmith's method. This is where the similarity ends. The node to which a primitive is initially assigned is called its "home node," and that node will always store a copy of the primitive. Once a node discovers that it does not have a part of the ODB it needs, it requests that part from its home node. This is called an "ODB miss."

When the primitive is checked for intersection, it is not thrown away; it is kept on that node until memory is exhausted and space is needed for another primitive. In this way, a node stores its share of the ODB plus some number of transitory primitives. Transitory primitives are thrown away as needed to accommodate new transitory primitives needed in intersection. Since the transitory primitives are essentially a primitive-cache, it is appropriate to use the least-recently-used (LRU) cache replacement method to select which transitory primitives are no longer needed. Since only the least recently used transitory primitives are thrown away, the more heavily used ones remain on the node. This greatly reduces the message traffic between nodes, and more closely approaches the ideal condition of having the whole ODB resident on each node.

This method of ODB decomposition has the best of both worlds. It has the ability to distribute a very large number of primitives across a number of nodes, and the load balance is kept much more even. What's more, the ODB distribution adjusts itself to give much better performance than a static decomposition.

Changes to the Hierarchy

Sending messages from one hypercube node to another is a costly process. There is a heavy overhead time penalty to set up a message route plus a modest penalty for each byte transferred. In order to defray the high startup cost, large messages are preferred over short ones. A single primitive, the result of an ODB miss, would make a very short message. It is desirable to send several primitives at once when swapping is required. But which primitives should be picked? It would be most helpful to send additional primitives which are likely to be intersected against. Indeed, the Kay algorithm usually tests all of the children of a given Hnode. It, therefore, makes sense to send all siblings of the requested primitive as they will likely be tested. Thus, we move from the concept of swapping individual primitives to swapping all primitives associated with a certain Hnode.

A number of changes to the structure and content of the hierarchy is required to support this ODB decomposition and the swapping scheme. The hierarchy is composed of two basic entities: the group of Hnodes which comprise the infrastructure of the hierarchy, and the primitives. As demonstrated above, the Hnode infrastructure is only responsible for 14.3% of the total number of nodes in the ODB. And since an Hnode takes only one third the memory space to store as a primitive, the Hnode infrastructure effectively is only responsible for about 5% of the total size of the ODB. Thus, it is economical for each node to store the ODB infrastructure, and just swap groups of primitives. This also allows the Kay algorithm to go all the way to the leaf level before an ODB miss is possible. Thus, each Hnode must contain information about whether or not its child primitives are resident.

Hnodes must also keep track of the LRU reference word for cache replacement purposes. The new Hnode structure contains the following information. Note the addition of the two new fields to the previous structure of an Hnode.

TABLE 6

REVISED FIELDS IN THE HNODE DATA STRUCTURE

Type	Description
1.	Pointers to 1 to 8 subtrees.
2.	A unique ID.
3.	A bounding volume enclosing the whole subtree.
4.	LRU reference word.
5.	A flag which is true if this Hnode's child primitives are not resident.

One thing to note is that not all Hnodes have child primitives. Some Hnodes will have only other Hnodes as children. These interior Hnodes are unswappable, and do not take part in the ODB distribution process. The balance of the Hnodes are called "swappable Hnodes," and do take part in the distribution process. Stated another way, an Hnode is a swappable Hnode if and only if at least one of its children is a primitive or CSG tree.

As stated above, a certain portion of the ODB must remain resident on each computing node. Rather than thinking of this portion as a set of primitives, we shall think of it as a set of swappable Hnodes. The swappable Hnodes are divided evenly among the processors rather than the primitives directly. In this way, the child primitives of a swappable Hnode are never split between two computing nodes. In a scene with a large number of primitives, the unevenness in the distribution of primitives caused by this method is negligible.

As primitives are swapped in from other nodes as groups, so are they swapped out as groups. When a node's memory is exhausted and it needs more primitives to complete an intersection, space must be made for the new primitives. The LRU replacement algorithm targets the Hnode whose LRU reference word is smallest for replacement. All child primitives of the target Hnode are freed, and the Hnode is marked as "swapped." The targeting and freeing operations are repeated until enough space is available for the incoming primitives. Note that a CSG subtree is considered as a single primitive, and is treated as such. Thus, many Hnodes could possibly be freed just to make room for one CSG subtree. Although the CSG nodes could be swapped just as Hnodes are, experimental evidence has shown this to be unnecessary. CSG trees are sufficiently small in comparison to the whole ODB that they do not make a great impact on the swapping action.

Changes to the Ray Tracing Loop

Now that ODB distribution has been addressed, we must now address the problems this causes in the ray tracing loop. Since parts of the ODB can be missing on each node, the Kay intersection algorithm may fail. When it does fail, a request for the missing primitive must be formulated and sent to the primitive's home node. (A primitive's home node is based on the unique ID number assigned to the Hnode parent of the primitive.) The interruption in the intersection process raises a number of questions. One may be stated as follows, "what happens when the requested primitive comes back and is inserted into the ODB?" Should the Kay algorithm be restarted from the beginning, or from where it stopped? The first option is unacceptable for two reasons. First, it is grossly inefficient to repeat the hierarchy traversal done before the ODB miss. Second, a different ODB miss may cause thrashing. The intersection might never complete. The option of restarting the intersection process is clearly desirable, but it cannot be done without paying a considerable cost in terms of program complexity.

Once an ODB miss occurs, what happens while the node is waiting for primitives from another node? It may do one of two things: wait, or work on another ray. Considering the cost of sending a message to another node, and waiting for it to reply, waiting is clearly out of the question. The time to send a 2 KB message from one node to another is about 1.2 ms [Intel]. Doubling this and adding another 2 ms latency at the other end yields up a round-trip time of about 4.4 ms *per ODB miss*. Experiments show that a ray takes 10 - 20 ms to ray trace if no ODB miss occurs. Clearly a 4.4 ms delay per ODB miss would cripple the performance of the Ray Tracer. Therefore, it must occupy this time doing something constructive; processing another ray is an ideal choice. This means that the entire state of the ray tracing process must be saved when an ODB miss occurs. The ideal place to save this information is in the same data structure as the offending ray. Below is the new structure for a ray. Many of the fields in this data structure have not yet been explained, but will be shortly.

TABLE 7

FIELDS OF THE RAY DATA STRUCTURE

Type	Description
1.	Origin of ray.
2.	Direction of ray.
3.	Recursion level.
4.	Total distance ray has traveled.
5.	List of objects ray is currently inside. (For HTTM)
6.	Kay intersection heap.
7.	List of objects ray has intersected. (For coincident intersection point disambiguation.)
8.	Space for temporary variable used in Kay algorithm.
9.	Ray type. (Shadow ray or shading ray)
10.	Ray state (See Table 8).
11.	Pointer to parent ray if spawned by another ray.
12.	Pointer to child ray is one has been spawned.
13.	Pixel coordinates if primary ray.
14.	Intensity of pixel if primary ray.
15.	Unique ID number.
16.	Temporary variables used by shading model.

Now that it has been decided that intersection may be stopped and restarted, we must consider the other steps in the ray tracing process, namely the shading step. The shading model casts shadow rays every time it is evaluated, and optionally casts reflected and refracted rays. These secondary rays must also be ray traced. Since they may also cause ODB misses, the shading model evaluation must be made interruptible, too! To complicate matters further, the shading model may be interrupted in no less than three different locations: once for each light source when casting a shadow ray, once for the reflected ray, and once for the refracted ray! Now, the ray tracing loop has become a very complex choreography of interruptible states, spawning of subrays, and resumption of control. The following FSA (Finite State Automaton) is the solution to this control problem. In Table 8, we see that rays are divided into a number of different types: primary rays, secondary rays, and shadow rays. The only difference between the types is the way in which the shading model operates. For primary rays, the full shading model is evaluated, and the result is stored at the appropriate pixel coordinates in the local frame buffer. Secondary rays execute the full shading model, but pass their intensity to their parent ray rather than the frame buffer. Shadow rays need not be shaded at all, only intersected with the ODB.

TABLE 8

RAY STATES

Type	Description
1.	Ready to intersect – This state means that a ray is set up and ready be be intersected against the ODB.
2.	Pending object from another node – Here, the ray has failed the intersection process due to an ODB miss, and is waiting for the required primitives to be sent from elsewhere. This state has no action function.
3.	Shadow ray setup – This is the first step of the shading model. Shadow rays are set up and spawned from this state. A shadow ray to a different light source is spawned each time this state is entered until all shadow rays have been cast.

TABLE 8 (Continued)

Type	Description
4.	Pending on shadow ray – Once a shadow ray has been spawned, the parent ray must wait for it to complete. This state has no action function as there is nothing to do but wait.
5.	Process shadow ray – When a shadow ray has completed, control comes to this state. The result of the shadow ray intersection are stored and control is passed back to the "shadow ray setup" state to cast more shadow rays.
6.	Shading – This step in the shading model performs all operations that depend only on the results of the shadow rays. i.e. ambient, diffuse, and specular components.
7.	Reflective shading – If the surface of the primitive in question is reflective, this state spawns a reflected ray.
8.	Pending reflected ray – Control comes here to wait on a reflected ray to be traced. This state has no action function.
9.	Process reflected ray – The contribution of the reflected ray is added into the overall shading in this state.
10.	Transmissive shading – If the surface of the primitive in question is transmissive, this state spawns a refracted ray.
11.	Pending transmitted ray – Control comes here to wait on a refracted ray to be traced. This state has no action function.
12.	Process transmitted ray – The contribution of the refracted ray is added into the overall shading in this state.
13.	Forward results – The ray has been fully evolved, and the results are ready to be passed on. Depending on the ray type, the results are either put in the local frame buffer (primary ray), or forwarded to the parent ray (shadow or secondary ray).

TABLE 9

RAY STATE TRANSITION EVENTS

Type	Description
1.	ODB miss – This event is posted by the "ready to intersect" state when an ODB miss occurs.
2.	Object received – This event is posted when primitives arrive from another node as the result of an ODB miss.

TABLE 9 (Continued)

Type	Description
3.	Spawn – Posted whenever a state had to spawn a subray. This happens when shadow rays, reflected rays, and refracted rays are spawned.
4.	Complete – This event is posted to a parent ray when a child ray has completed.
5.	Done – Posted by state action functions, this event signals that the state completed successfully, and the ray is ready to move on to the next state.
6.	Backtrack – If a shadow ray intersects a transparent object, it is not necessarily occluded. This event is used to restart the intersection process to find the next intersection point along the shadow ray.
7.	Missed – If the intersection process misses all objects in the ODB, this event is posted to initiate a shortcut straight to the "Ray done" state.

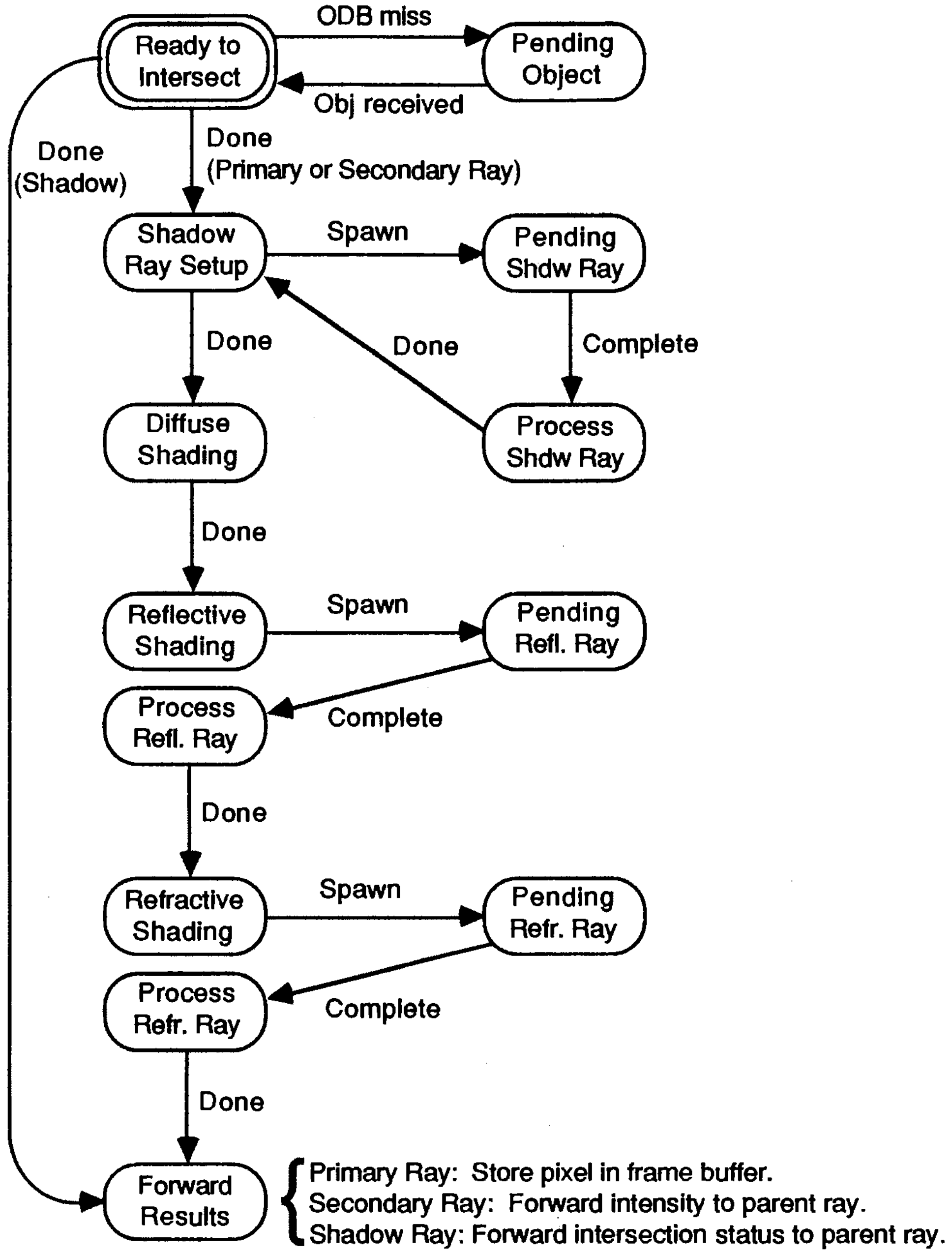


Figure 12: Control Flow for Primary Rays

Transitions between ray states are caused by "events." (See Table 9) These events are based on the result of the "action" associated with each state. These actions perform the various steps in the ray tracing process. For example, the action associated with the first state in the FSA is to try to intersect the ray with the ODB. If the intersection fails, the action function posts an "ODB miss" event for the ray, and terminates. The result of this event is to place the ray in the "pending object from another node" state. If the intersection succeeds, the action function posts a "function complete" event and terminates. The result of this event is to place the ray into the "shadow ray setup" state. The concept of state driven ray tracing complicates the classical ray tracing loop, but beautifully modularizes it into an interruptible series of atomic operations. Although some of the states presented above could be merged, they are left separate for clarity. Performance is affected negligibly because of this.

As stated earlier, multiple rays must be allowed so time is not wasted waiting for ODB misses to be resolved. Indeed, multiple rays are already allowed by virtue of the state driven structure of the ray tracing loop. Since there can be a number of pending rays equal to the maximum recursion depth, a way is needed to keep track of all of these rays. A way is also required to keep track of events destined for a particular ray. The solution is a "ray queue" to keep the rays, and an "event queue" to keep track of the events. As an event is intended for a specific ray, it is necessary to store a pointer to the target ray as well.

As new rays are created, they are pushed onto the ray queue to begin their journey through the states that will ray trace them. Similarly, ray-event tuples are pushed onto the event queue for evaluation. A *scheduler* is responsible for driving the FSA from the rays and events. The scheduler sits at the top of the control structure for the new node ray tracing loop. Below is the pseudocode representation for the new node program.

```
Initialize data structures and variables
Initialize ray and event queues
Download ODB from the host
While there are still pixels to ray trace and ray queue not empty

   /* Service all events in the event queue. */
   While the event queue is not empty
      Pop the event queue and determine the next state of the ray
   Endwhile

   /* Service any requests for primitives from other nodes. */
   If there is a ODB request from another node
      Pack up the requested portion and send it
   Endif

   /* If another node has responded to an ODB request sent */
   /* by this node, add the new primitives to the ODB.*/
   If there is an ODB request reply
      Receive the message
      Unpack it into the local ODB
      Notify all rays pending on this reply by posting events
   Endif

   /* Add a new primary ray if there is room. */
   If there is room for another ray on the ray queue
      Construct and initialize a new primary ray
      Push it onto the ray queue
   Endif

   /* Execute a ray's state function. */
   Pop a ray from the ray queue
   Execute its state function
Endwhile

Send the local frame buffer to the host
Send node timing and statistics to the host
```

Algorithm 10: Scheduler for State Driven Node Program

One will notice the striking resemblance between the above algorithm and any standard round-robin task scheduler. In the ray tracer's case, the analog for a process is the ray.

Changes to Image Decomposition

When the leap is made from a duplicated ODB to a distributed ODB, many things change. The structure of the ODB changes from a fully intact hierarchy to a hierarchy missing some or most of its leaves. The hierarchy nodes themselves become more complex. Ray-ODB intersection becomes an interuptable, re-entrant process rather than classical straight-line code. Even the ray tracing loop itself changes from a regimented and easy to understand loop into a complex scheduler driving a thirteen state FSA.

After such a drastic change to the basic ray tracing loop, the suitability of the comb image decomposition needs to be reassessed. There is one basic problem associated with the comb decomposition – that of locality. The pixels ray traced by a single node using the comb decomposition are fairly evenly scattered over the entire image plane. This is just the effect we want with a copied ODB to give a good load balance. It is disastrous to a distributed ODB. If the DODB is to perform well, then the rays tested against it should be fairly localized with respect to their positions and directions within the scene. This locality of reference keeps the number of ODB misses down, and the performance up. If widely varying rays are intersected against the DODB, then there will be a much higher miss rate, and correspondingly lower performance. Experiments verify not only the lower performance of the comb decomposition, but also show a very poor load balance. It is therefore desirable to invent a new image decomposition to solve the load balance and locality problems simultaneously.

The solution used by the Hypercube Ray Tracer is what is generally called the "block" decomposition. The image plane is divided into a large (usually 1024 in this case) number of rectangular blocks which are handed out dynamically to processors. Each block encloses a number of pixels that one node will be responsible for ray tracing. When a processor finishes ray tracing all of the pixels in its block, it is assigned a new block spatially close to the previous one. In this method, the dynamic block assignment solves the load balancing problem, and new blocks are chosen close to old blocks to give heightened locality.

Block assignments are kept track of by the host program running on the SRM. As compute nodes complete their blocks, they send the block's frame buffer to the host where is is copied into the global frame buffer. After this is done, the host assigns a new block to the node as close to the old one as possible. This continues until all blocks have been rendered.

CHAPTER VI

RESULTS, TIMING, AND PERFORMANCE METRICS

Many things impact the ultimate performance of the Hypercube Ray Tracer. One of the most important of these is the suitability of the object database hierarchy. Its branching ratio is critical to the speed of the modified Kay intersection algorithm. Shown below is a graph of rays cast per second versus the maximum branching ratio of the ODB hierarchy. The term 'maximum branching ratio' is used advisedly here since not all hierarchy nodes are guaranteed to have the maximum number of children. The Hiergen hierarchy generator often constructs hierarchy nodes of less than maximum branching ratio due to efficiency considerations.

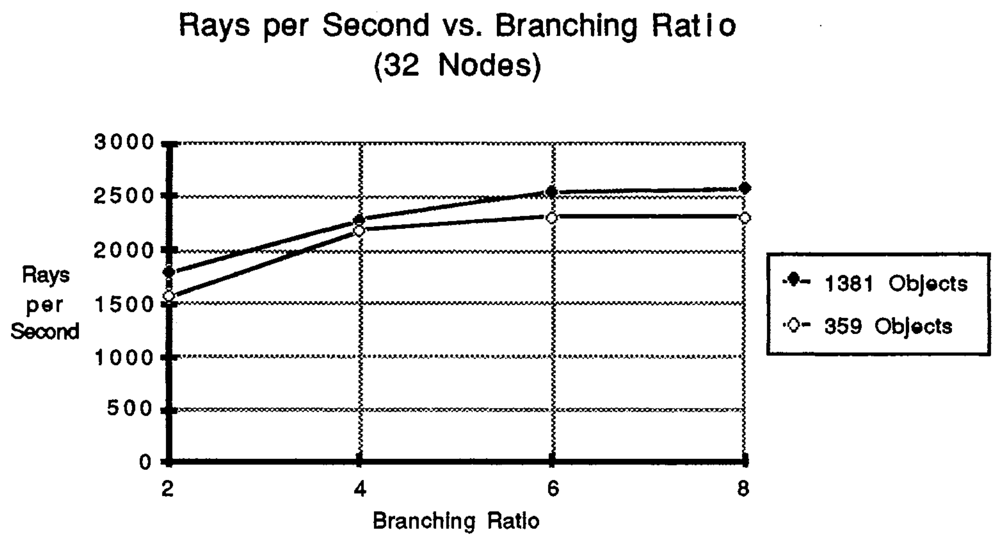


Figure 13: Hypercube Ray Tracer Speed vs. ODB Branching Ratio

Figure 13 show us that branching ratios of eight and six provide better performance than that of lower branching ratios. Branching ratios below six start to degrade performance significantly. Performance on an ODB with a branching ratio of two is approximately 25% less than that of an ODB with a branching ratio of six for the test cases shown in figure 13. In all subsequent data, the branching ratio is eight. No data of this type is available against which to compare these figures.

Shown below in figure 14 is a graph that profiles the performance of the Hypercube Ray Tracer over a wide range of ODB sizes. The scenes used to gather the following data consisted of a number of randomly placed constant sized spheres with no specular reflection. A random spatial distribution was chosen so that each object contributes a constant amount to the rendering time of the image. If a high degree of spatial coherence exists between a group of objects, then they tend to contribute less to overall rendering time than the same objects placed farther apart.

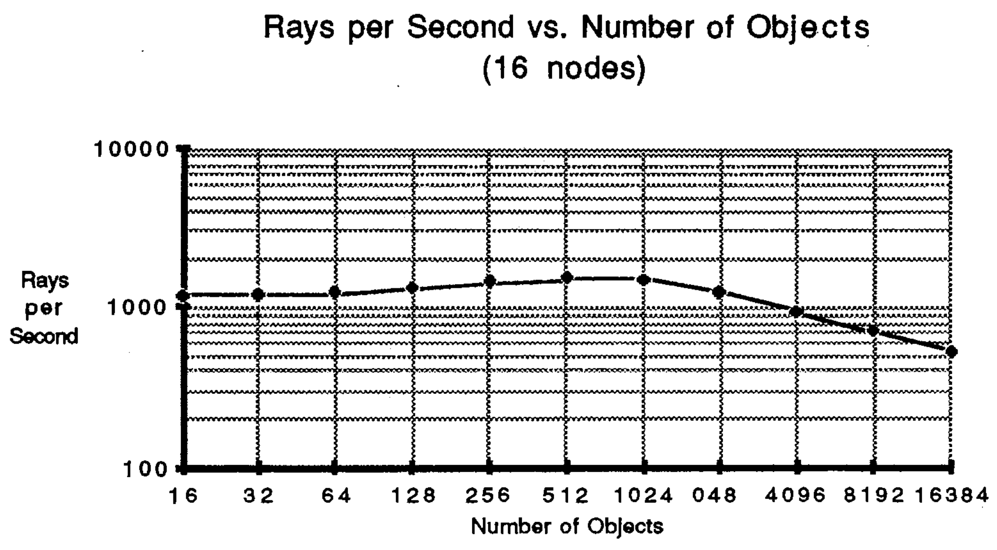


Figure 14: Hypercube Ray Tracer Speed vs. Number of Objects

The shape of the performance curve in figure 14 is classical in nature. The Hypercube Ray Tracer exhibits relatively constant performance up to approximately 1000 objects. At that point, performance drops off in a logarithmic manner. The performance dropoff is just as expected for the modified Kay algorithm searching the hierarchical ODB. Logarithmic search time is expected from the Kay algorithm. The flat area from 32 objects to 1024 objects is due to constant overhead in the ray tracer. There are two types of overhead associated with the ray tracing procedure: constant overhead for the whole process, and constant overhead per ray. The former overhead is responsible for the general flat shape of the performance curve from 32 to 1024 objects, and the latter overhead is responsible for the slight increase in speed at 512 objects.

Figure 15 shows speedup plotted against number of processing nodes for a scene consisting of 1024 randomly scattered spheres. Speedup is defined here as:

$$S_p = \frac{N\, t_{slow}}{t_{host}}$$

where S_p is speedup, N is the number of nodes, t_{slow} is the longest single node running time, and t_{host} is the time taken by the host. Of prime importance in figure 15 is the very nearly linear speedup of the Hypercube Ray Tracer.

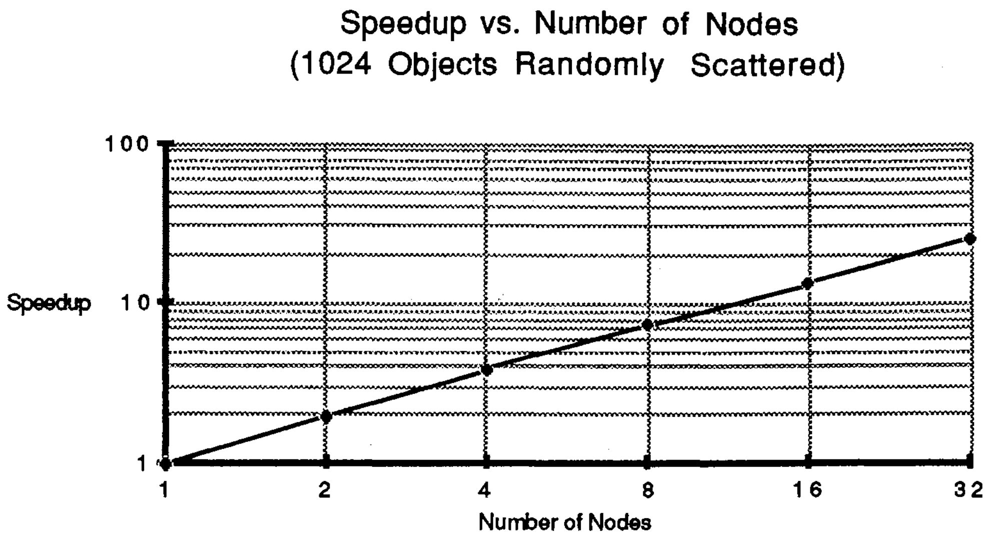


Figure 15: Hypercube Ray Tracer Speedup Curve

Although the above performance metrics are useful for general characterization of the Hypercube Ray Tracer, they vary greatly from scene to scene. To illustrate this fact, data is presented below for three different scenes with roughly the same number of objects all run on 32 nodes.

Scene	Objects	Total Time	Rays/Sec.
1	1024	2112	1492
2	1381	17093	2595
3	1310	11837	2170

Figure 16: Ray Tracer Speed for Similar Scenes

The major differences between these scenes is the placement, and surface properties of the various objects. Scenes 2 and 3 are more representative of a realistic scene than 1. Kay's ray tracer, run on an IBM 4381 minicomputer, logs 40 to 50 rays per second (for rays that hit something). A 4381 is approximately equivalent performance to one iPSC/2 computing node. Arvo's ray tracer, run on an Apollo DN570 microcomputer, logs 60 to 100 rays per second on scenes of roughly the same complexity. A DN570 is roughly 30% the performance of one iPSC/2 node. Recall that Arvo is using a different intersection algorithm that that of Kay and the Hypercube Ray Tracer. The Hypercube Ray Tracer logs 60 to 100 rays per second per node on scenes of this complexity. Goldsmith's ray tracer, implemented on the NCUBE hypercube, logs about 45 rays per second per node. Each NCUBE node is approximately 75% the performance of an iPSC/2 node.

Another metric of the Hypercube Ray Tracer's performance is how well it tolerates large distributed ODB's. How does performance suffer as the ODB become much larger than the nodes' local memory? This can be shown by plotting rays per second versus the largest fraction of the ODB that a node may store.

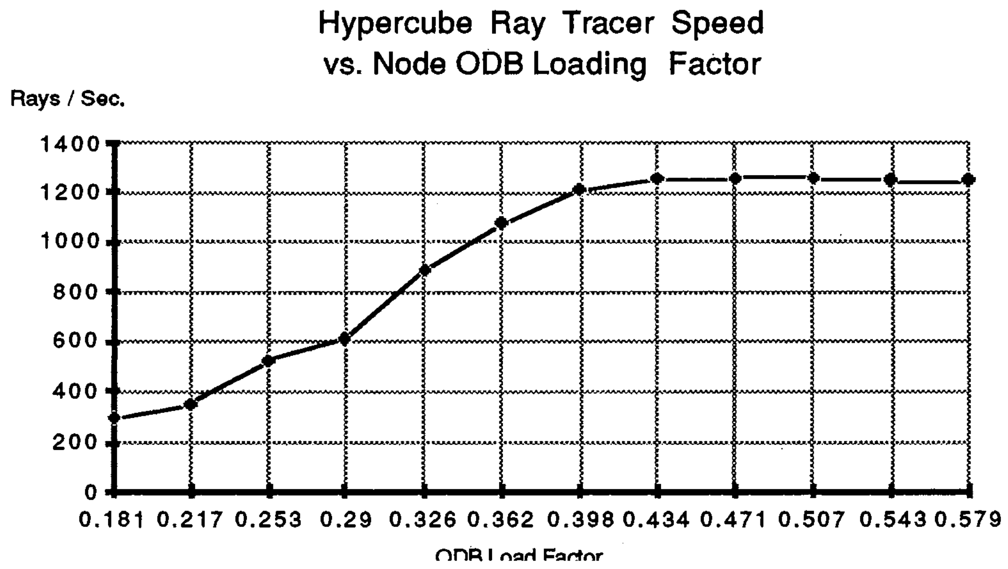


Figure 17: Ray Tracer Speed vs. Node Overloading Factor

From this graph, we can see that performance is reduced substantially when each node can store less than 36% of the total ODB. This would not seem to bode well for the scalability of ray tracing on a distributed memory machine. However, we have only to notice that in Figure 15, we see over 16000 objects ray traced without this catastrophic performance degradation. The data for Figure 17 was collected from a scene of about 1300 objects by reducting the total number of objects each node may store. More testing is necessary, but it would seem reasonable to conclude that the distribution of objects in the ODB influences distributed ODB ray tracing performance. This has already been shown to be true for a non-distributed database.

From these comparisons, we can see that the Hypercube Ray Tracer is on an even par with other ray tracers on a per node basis. When we consider it parallelism, however, it far outperforms most serial rays tracers. When compared against the hypercube ray tracer of Goldsmith, it is about equal or slightly superior when run on the same number of nodes.

CHAPTER VII

CONCLUSIONS

The Ray Tracing Algorithm

We have seen the development of a parallel ray tracing system in preceding chapters. Results show that performance scales excellently with the number of computing nodes attacking the problem on a medium grained distributed memory parallel computer. This is to be expected since the ray tracing process itself exhibits fine grained parallelism on the pixel level. Furthermore, little or no performance penalty is incurred by decomposing the image plane for parallel rendering. All measures of parallel program performance (speedup, efficiency, effectiveness) show the Hypercube Ray Tracer to be nearly ideal.

Scene Specification Language

The Rayd language and compiler have proven themselves to be effective scene description and development tools in practice. Many scenes have been developed using Rayd and no serious conceptual flaws have been uncovered. It has fulfilled all of its original design goals. This is easily verified since each of them have been extensively exercised over the Hypercube Ray Tracer's development cycle. Although no replacement for RenderMan, Rayd does provide a simple and easy-to-use front-end for the Hypercube Ray Tracer.

Object Database Distribution

Problems associated with large object database size must be confronted in the distributed memory computing environment when one wishes to render very large scenes.

These problems include: how to decompose the ODB across multiple computing nodes, how to allow all nodes access to the distributed ODB, and how to keep internode communications low.

The Hypercube Ray Tracer deals effectively with each of these problems with a novel ODB decomposition method and cacheing strategy, dynamic load balancing, and spatially coherent load assignment. The unfortunate side-effect of a much more complex ray tracing loop and interruptable ray-ODB intersection is a complicated program structure. This imposes little performance penalty, however. Near ideal parallel performance is maintained until the object database size exceeds about five times the number of primitives that one computing node can store. At this point, internode communications brought about by a decreased hit ratio starts to dominate the time spent doing useful work. This effect would seem to indicate a performance plateau dictated not by the computing power or number of nodes, but rather by the amount of memory contained on each. This phenomenon is highly variable with the type and properties of the objects modeled in the scene and is evident from the graphs in Chapter 6.

Constructive Solid Geometry

Constructive Solid Geometry (CSG) is incorporated into the Hypercube Ray Tracer with the aid of special nodes in the object database hierarchy and a novel intersection scheme (HTTM). Although the cost of CSG intersection is high, it is much less than previous methods. Furthermore, the HTTM intersection method is independent of object database organization -- it may be used in space subdivision ray tracers as well as other object subdivision ray tracers. Although CSG was not ostensibly a design goal, its inclusion greatly enhanced the flexibility of the Hypercube Ray Tracer.

CHAPTER VIII

FUTURE WORK

There are many new directions in which the Hypercube Ray Tracer could be expanded. The most interesting of these changes, as far as the Hypercube Ray Tracer is concerned, involve enhancements to parallel operation. Other enhancements are more generic in nature. Such generic enhancements might include a *RenderMan* user interface, more primitive types, and a more efficient antialiasing method. All of these problems, however, are only superficially influenced by Hypercube Ray Tracer's parallelism.

A more interesting problem to tackle deals with the efficiency of the distributed ODB. Although experimental evidence shows that it works well, much inefficiency still exists. Some nodes are burdened with considerably more ODB requests than others. Although this does not create a load imbalance, it does slow that node's response time to the ODB request. If a method could be found to predict which objects are likely to be requested more than others, then they could be spread out. This would improve the overall response time by more evenly distributing the ODB requests.

Mentioned in Chapter 7 is the apparent relationship of absolute maximum ODB size to a single node's memory size rather than total distributed memory size. This phenomenon is not well behaved, and has neither been characterized nor rigorously studied. An understanding of the causes and consequences of this phenomenon might give insight into the question of whether ray tracing is best attacked by coarse grained or fine grained parallel computers. Even though ray tracing has been done on all three classes of parallel computers, not enough information is available to make a judgement about which type of architecture is best suited to ray tracing.

The Hypercube Ray Tracer shuttles objects about the hypercube. This is in contrast to Salmon and Goldsmith's method. Even though shuttling objects works well, this does not preclude the swapping of rays among nodes in certain circumstances. If it could be determined that another node contained a preponderance of primitives likely to be needed in the intersection process and not present on the current node, then it might be more economical to transfer the ray rather than the primitives. This type of hybrid swapping arrangement might provide a modest gain in performance.

Presently, the scene specification parsing and ODB hierarchy generation are implemented as serial processes, and as such, suffer from poor performance. Even though the ODB hierarchy needs to be generated only once, it takes a rather large amount of time in comparison to the rendering process. Most of this poor performance can be attributed to relatively slow disk I/O. If this process were ported to the hypercube where the entire ODB could be kept in distributed memory, a dramatic speedup would be realized.

BIBLIOGRAPHY

[Abram 85] Abram, Greg, Lee Westover, and Turner Whitted, "Efficient Alias-free Rendering using Bit-masks and Look-up Table," Computer Graphics, Vol. 19, No. 3, July 1985, p. 57.

[Aho 86] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman, Compilers, Principles, Techniques, and Tools, Addison-Wesley Publishing Co., Reading, Mass., pp. 215-217.

[Amanatides 84] Amanatides, John, "Ray Tracing with Cones", Computer Graphics, Vol. 18, No. 3, July 1984, pp. 129-135.

[Arvo 87] Arvo, James, and David Kirk, "Fast Ray Tracing by Ray Classification," Computer Graphics, Vol. 21, No. 4, July 1987, pp. 55-64.

[Barr 84] Barr, Alan H., "Global and Local Deformations of Solid Primitives," Computer Graphics, Vol. 18, No. 3, July 1984, pp. 21-30.

[Barr 86] Barr, Alan H., "Ray Tracing Deformed Surfaces," SIGGRAPH '86, Vol. 20, No. 4, August 1986, pp. 287-296.

[Blinn 77] Blinn, Jim, "Models of Light Reflection for Computer Synthesized Pictures," Computer Graphics, Vol. 11, No. 2, Summer 1977, pp. 192-198.

[Bronsvoort 85] Bronsvoort, William F., and Fopke Klok, "Ray Tracing Generalized Cylinders," ACM Transactions on Graphics, Vol. 4, No. 4, October 1985, pp. 291-303.

✓ [Carter 89] Carter, Michael B., and Keith Teague, "The Hypercube Ray Tracer", to appear in Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications (HCCA4), March 6-8, 1989, Monterey, CA.

[Cook 82] Cook, Robert L., and Kenneth E. Torrance, "A Reflectance Model for Computer Graphics," ACM Transactions on Graphics, Vol. 1, No. 1, January 1982, pp. 7-24.

✓ [Cook 84] Cook, Robert L., Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing", Computer Graphics, Vol. 18, No. 3, July 1984, pp. 137-145.

[Daniel 89] Daniel, Ronald E., Michael B. Carter, and Keith A. Teague, "A Parallel Image Processing System for the iPSC/2", to appear in Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications (HCCA4), March 6-8, 1989, Monterey, CA.

√ [Deguchi 86] Deguchi, Hiroshi et. al., "A Tree-Structured Parallel Processing System for Image Generation by Ray Tracing", Systems and Computers in Japan, Vol. 17, No. 12, February 1986, pp. 51-62.

[Dippe 85] Dippe, Mark A. Z., and Erling Henry Wold, "Antialiasing Through Stochastic Sampling," Computer Graphics, Vol. 19, No. 3, July 1985, pp. 69-78.

[Foley 84] Foley, J. D., and A. Van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company, Reading, Mass., pp. 255-261.

[Fujimoto 86] Fujimoto, Akira, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray Tracing System," IEEE Computer Graphics and Applications, Vol. 6, No. 4, April 1986, pp. 16-26.

√ [Gaudet 88] Gaudet, Severin, Richard Hobson, Pradeep Chilka, and Thomas Calvert, "Multiprocessor Experiments for High-Speed Ray Tracing," ACM Transactions on Graphics, Vol. 7, No. 3, July 1988, pp. 151-179.

[Glassner 84] Glassner, Andrew S., "Space Subdivision for Fast Ray Tracing," IEEE Computer Graphics and Applications, Vol. 4, No. 10, Oct. 1984, pp. 15-22.

[Goldsmith 87] Goldsmith, Jeff, and John Salmon, "Automatic Creation of Object Hierarchies for Ray Tracing", IEEE Computer Graphics and Applications, Vol. 7, No. 5, May 1987, pp. 14-20.

√ [Goldsmith 88] Goldsmith, Jeff, and John Salmon, "A Hypercube Ray-tracer", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3), Vol. II, Jan. 1988, pp. 1194-1206.

[Goldstein 71] Goldstein, E. and R. Nagle, "3D Visual Simulation," Simulation, Vol. 16, Jan. 1971, pp. 25-31.

[Greenberg 89] Greenberg, Donald P., "Light Reflection Models for Computer Graphics", Science, Vol. 255, April 1989, pp. 166-173.

√ [Hanrahan 83] Hanrahan, P., "Ray Tracing Algebraic Surfaces," Computer Graphics, Vol. 17, No. 3, 1983, pp. 83-90.

[Heckbert 86] HeckBert, Paul S., "Survey of Texture Mapping," IEEE Computer Graphics and Applications, Vol. 6, No. 11, November 1986, pp. 56-67.

[Intel 88] iPSC/2 User's Guide.

√ [Kajiya 83] Kajiya, James T., "New Techniques for Procedurally Defined Objects," Computer Graphics, Vol. 17, No. 3, July 1983, pp. 91-102.

[Kay 86] Kay, Timothy L., and James T. Kajiya, "Ray Tracing Complex Scenes," ACM SIGGRAPH 1986, Vol. 20, No. 4, August 1986, pp. 269-278.

√ [Lee 85] Lee, Mark, Richard A. Redner, Samuel P. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," Computer Graphics, Vol. 19, No. 3, July 1985, pp. 61-67.

[Lee 89] Lee, Mark, Personal Communication.

[Mandelbrot 77] Mandelbrot, Benoit, Fractals: Form, Chance, and Dimension, W. H. Freeman, San Francisco, 1977.

[Mitchell 87] Michell, Don P., "Generating Antialiased Images at Low Sampling Densities," Computer Graphics, Vol. 21, No. 4, pp. 65-72.

✓ [Orcutt 88] Orcutt, David E., "Implementation of Ray Tracing on the Hypercube", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications (HCCA3), Vol. II, Jan. 1988, pp. 1207-1210.

[Phong 75] Phong, B. T., "Illumination for Computer Generated Pictures," Communications of the ACM, Vol. 18, No. 6, June 1975, pp. 311-317.

[Potmesil 82] Potmesil, Michael, and Indranil Chakravarty, "Synthetic Image Generation with a Lens and Aperture Camera Model," ACM Transactions on Graphics, Vol. 1, No. 2, pp. 85-108, April 1982.

[Reingold 83] Reingold, Edward M., and Wilfred J. Hansen, Data Structures, Little, Brown and Company, Boston, MA., pp. 181-185.

✓ [Roth 82] Roth, S. D., "Ray Casting for Modeling Solids," Computer Graphics and Image Processing, Vol. 18, No. 2, Feb. 1982, pp. 109-144

✓ [Rubin 80] Rubin, S., and T. Whitted, "A Three-Dimensional Representation for Fast Rendering of Complex scenes," Computer Graphics, Vol. 14, No. 3, July 1980, pp. 110-116.

[Uselton 89] Uselton, Samuel P., Personal Communication.

✓ [VanWijk 84] van Wijk, Jarke J., "Ray Tracing Objects Defined By Sweeping Planar Cubic Splines," ACM Transactions on Graphics, Vol. 3, No. 3, July 1984, pp. 223-237.

[Weghorst 84] Weghorst, Hank, Gary Hooper, and Donald P. Greenberg, "Improved Computational Methods for Ray Tracing," ACM Transactions on Graphics, Vol. 3, No. 1, January 1984, pp. 52-69.

[Whitted 80] Whitted, Turner, "An improved Illumination Model for Shaded Display," Communications of the ACM, Vol. 23, No. 6, June 1980, pp. 343-349.

✓ [Yossef 86] Youssef, Saul, "A New Algorithm for Object Oriented Ray Tracing," Computer Vision, Graphics, and Image Processing, No. 34, 1986, pp. 125-137.

APPENDIXES

APPENDIX A

BNF FORM OF RAYD SCENE DESCRIPTION LANGUAGE

```
/* Top level syntax of a Rayd scene description */

infile       :  infile include
         |  include
         |  infile define
         |  define
         |  infile ambient
         |  ambient

/* Include format */

include      :  INCLUDE qstring SEMI
         |  error

/* Define format */

define       :  DEFINE COLOR word EQUAL triple SEMI
         |  DEFINE SURFACE word surf_block
         |  DEFINE PLANECURVE word pcur_block
         |  DEFINE OBJECT word obj_block
         |  DEFINE SCENE scn_block
         |  DEFINE OBSERVER obs_block
         |  DEFINE LIGHT word lite_block

/* Ambient format */

ambient      :  AMBIENT EQUAL triple SEMI

/* Planar curve definition syntax for prisms */

pcur_block   :  pcur_lb pcur_elem_list RBRACE SEMI
         | error

pcur_lb      :  LBRACE

pcur_elem_list:   pcur_elem_list pcur_element
         |  pcur_element

pcur_element   :  CURVE pcur_spec LPAREN pcur_parm_list RPAREN
            SEMI
```

```
pcur_spec    :  word

pcur_parm_list:   pcur_parm_list pcur_parm SEMI
         |  pcur_parm SEMI

pcur_parm    :  POINTS EQUAL double_list

/* Object definition syntax */

obj_block    :  obj_lb obj_elem_list RBRACE csg_expr SEMI
         |  obj_lb obj_elem_list RBRACE SEMI
         |  error

obj_lb       :  LBRACE

obj_elem_list: obj_elem_list obj_element
         |  obj_element

obj_element : OBJECT obj_spec LPAREN obj_parm_list RPAREN SEMI
         |  OBJECT obj_spec LPAREN obj_parm_list RPAREN word SEMI
         |  error

obj_spec :  word

obj_parm_list: obj_parm_list obj_parm SEMI
         |  obj_parm SEMI

obj_parm :  POSITION EQUAL triple
         |  ROTATION EQUAL triple
         |  COLOR EQUAL triple
         |  SIZE EQUAL triple
         |  COLOR EQUAL word
         |  SURFACE EQUAL word
         |  REFLECT EQUAL NUMBER
         |  TRANSMIT EQUAL NUMBER
         |  RINDEX EQUAL NUMBER
         |  SPEC EQUAL NUMBER
         |  PHONG EQUAL NUMBER
         |  E1 EQUAL NUMBER
         |  E2 EQUAL NUMBER
         |  SQR EQUAL NUMBER
         |  CURVE EQUAL word
         |  error

/* Scene definition syntax */

scn_block    :  scn_lb scn_elem_list RBRACE SEMI
         |  error

scn_lb       :  LBRACE

scn_elem_list: scn_elem_list scn_element
         |  scn_element

scn_element : OBJECT scn_spec LPAREN scn_parm_list RPAREN SEMI
         |  LIGHT scn_spec LPAREN scn_parm_list RPAREN SEMI
         |  error
```

```
scn_spec :  word

scn_parm_list: scn_parm_list scn_parm SEMI
         |  scn_parm SEMI

scn_parm :  POSITION EQUAL triple
         |  ROTATION EQUAL triple
         |  SIZE EQUAL triple
         |  error

/* Light source definition syntax */

lite_block   :  lite_lb lite_elem_list RBRACE SEMI
         |  error

lite_lb      :  LBRACE

lite_elem_list: lite_elem_list lite_elem
         |  lite_elem

lite_elem    :  LIGHT lite_spec LPAREN lite_parm_list RPAREN SEMI

lite_spec    :  word

lite_parm_list:   lite_parm_list lite_parm SEMI
         |  lite_parm SEMI

lite_parm    :  POSITION EQUAL triple
         |  ROTATION EQUAL triple
         |  COLOR EQUAL triple
         |  COLOR EQUAL word
         |  SIZE EQUAL triple
         |  BRIGHT EQUAL NUMBER
         |  TYPE EQUAL word
         |  ANGLE EQUAL NUMBER
         |  error

/* Observer definition syntax */

obs_block    :  obs_lb obs_parm_list RBRACE SEMI
         |  error

obs_lb       :  LBRACE

obs_parm_list: obs_parm_list obs_parm SEMI
         |  obs_parm SEMI

obs_parm :  POSITION EQUAL triple
         |  VIEWDIR EQUAL triple
         |  FLEN EQUAL NUMBER
         |  UPDIR EQUAL triple
         |  RESOLUTION EQUAL double
         |  RECURSION EQUAL NUMBER
         |  VRECTSIZE EQUAL double
         |  error

/* Surface definition syntax */
```

```
surf_block  :  surf_lb surf_parm_list RBRACE SEMI
         |  error

surf_lb     :  LBRACE

surf_parm_list:   surf_parm_list surf_parm SEMI
         |  surf_parm SEMI

surf_parm   :  COLOR EQUAL triple
         |  COLOR EQUAL word
         |  REFLECT EQUAL NUMBER
         |  TRANSMIT EQUAL NUMBER
         |  RINDEX EQUAL NUMBER
         |  SPEC EQUAL NUMBER
         |  PHONG EQUAL NUMBER
         |  error

/* Syntax for ordered pairs and triples */

triple      :  LPAREN NUMBER NUMBER NUMBER RPAREN

double_list :  double_list double
         |  double

double      :  LPAREN NUMBER NUMBER RPAREN

/* Syntax for a CSG expression */

csg_expr :  csg_and OR csg_expr
         |  csg_and

csg_and     :  csg_and AND object
         |  object

object      : word
         | NOT object
         | LPAREN csg_expr RPAREN
         | error

/* Miscellaneous syntax elements */

word     : WORD

qstring     : QSTRING
```

APPENDIX B

SAMPLE RAYD SCENE DESCRIPTION

```
/* This is the Rayd scene description for a scene containing  */
/* a variety of objects.  A number of Rayd's features are      */
/* used in constructing this scene.  Most notably the color    */
/* definition.  Reflectivity, transmissivity, and specularity */
/* illustrate the use of shading model parameters.             */

/* Specify the ambient light intensity. */
ambient = (1 1 1);

/* Define some colors to be used in object definitions. */
define color lite_red = (0.3 0 0);
define color med_blue = (0 0 0.6);
define color red      = (1.0 0 0);
define color green    = (0 1.0 0);
define color blue     = (0 0 1.0);
define color black    = (0 0 0);

/* Define the main body of the scene. */
define object balls {
   /* Background cube */
   object box (
      position = (20 0 0);
      color    = lite_red;
      size     = (1 20 20);
   );
   /* Tabletop cube */
   object box (
      position = (10 0 -1.5 );
      color    = (.8 .8 .6);
      size     = (30 5 1);
      rotation = (0 -1.5 0);
   );
   /* Sky cube */
   object box (
      position = (10 0 10.1);
      color    = med_blue;
      size     = (20 20 1);
   );
   /* Right side cube */
   object box (
      position = (10 -10.1 0);
      color    = lite_red;
      size     = (20 1 20);
   );
   /* Left side cube */
   object box (
      position = (10 10.1 0);
      color    = lite_red;
      size     = (20 1 20);
   );
   /* Background cube */
   object box (
      position = (-1 0 0);
      color    = lite_red;
      size     = (1 20 20);
   );
   /* Big green ball */
   object sphere (
      position = (9 .2 .2);
      color    = green;
      size     = (.9 .9 .9);
   );
   /* Small red ball */
   object sphere (
      position = (10 -.8 0);
      color    = red;
      size     = (.5 .5 .5);
   );
   /* Background mirrored ball */
   object sphere (
      position = (10 -1.1 1.1);
      color    = black;
      size     = (.5 .5 .5);
      reflect  = 1;
   );
   /* Yellow specular ball in green ball */
   object sphere (
      position = (8.3 .4 .4);
      color    = (.7 .7 .2);
      size     = (.5 .5 .5);
      spec     = .5;
      phong    = 10;
   );
   /* Blue cylinder */
   object cylinder (
      position = (8 -.5 -.4);
      color    = blue;
      size     = (.3 .3 .8);
      rotation = (0 -15 0);
   );
   /* Clear ball on top of cylinder */
   object sphere (
      position = (7.9 -.5 .18);
      color    = black;
      size     = (.3 .3 .3);
      transmit = 1;
      rindex   = 2;
   );
};
```

```
/* Light sources */
define light lumen {
   light l1 (
      position = (1 1 2);
      color    = (1 .5 1);
      brightness = 10;
      size     = (.3 .3 .3);
   );
   light l2 (
      position = (1 -2 2);
      color    = (1 .5 .5);
      brightness = 10;
      size     = (.3 .3 .3);
   );
};

/* Put all of the elements of the scene together. */
define scene {
   object balls (
      position = (0 0 0);
   );
   light lumen (
      position = (0 0 0);
   );
};

/* Specify a front view. */
define observer {
   position   = (5 0 0);
   viewdir    = (1 -.05 .05);
   updir      = (0 0 1);
   flen       = 5;
   vrectsize  = (3.7 3.7);
   recursion  = 8;
   resolution = (512 512);
};
```

APPENDIX C

INTERSECTION OF A RAY WITH A SPHERE

We are given a ray, the vector equation for which is expressed by:

$$\vec{r} = \vec{o} + t \cdot \hat{d} \qquad (1)$$

where $\vec{r}$ = Any position along the ray in question.
$\vec{o}$ = The origin of the ray.
$\hat{d}$ = The unit vector in the direction of the ray.
t = The parametric distance along the ray.

This vector equation may be broken up into three simultaneous scalar equations, one for each of the three coordinate axes:

$$\begin{aligned} r_x &= o_x + t\, d_x \\ r_y &= o_y + t\, d_y \\ r_z &= o_z + t\, d_z \end{aligned} \qquad (2)$$

These three equations can now be substituted into the implicit equation for a sphere,

$$x^2 + y^2 + z^2 = r_s^2, \qquad (3)$$

Note that this equation is that of a sphere positioned about the origin. All intersection calculations in the Hypercube Ray Tracer assume that the primitive being intersected lies at the origin, is of unit dimensions, and has no rotation. All three of these assumptions are made valid by transforming the ray in global coordinates into the primitive's local coordinate system. This transformation removes all scaling, translational, and rotational components from the primitive thereby allowing a simplified intersection procedure at the expense of doing two geometrical transformations. The equation resulting from the substitution into (3) is second degree in terms of t, therefore the quadratic equation can be used to solve for t.

$$t = \frac{-B \pm \sqrt{B^2 - 4 \cdot A \cdot C}}{2 \cdot A} \qquad (4)$$

where $A = r_x^2 + r_y^2 + r_z^2$
$B = 2(o_x\, r_x + o_y\, r_y + o_z\, r_z)$
$C = o_x^2 + o_y^2 + o_z^2 - r_s^2$

This yields t_0 and t_1, which represent the distance to the two intersection points referenced from the ray's origin. If the ray misses the sphere, then the values of t_0 and t_1 will be complex. To obtain the exact points of intersection from real values of t, we simply evaluate equation (1) with $t = t_0$ and $t = t_1$. Usually, the closer intersection point will be chosen. This corresponds to the lesser of t_0 or t_1 which is greater than 0. Calculating the points of intersection with other second degree surfaces is just as easy since it involves only using a different equation in place of equation (3).

2✓

VITA

Michael Brannon Carter

Candidate for the Degree of

Master of Science

Thesis: RAY TRACING COMPLEX SCENES ON A MULTIPLE-INSTRUCTION STREAM MULTIPLE-DATA STREAM CONCURRENT COMPUTER

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Sulphur, Oklahoma, April 3, 1965, son of Everett and Murrel Carter.

Education: Graduated from Davis High School, Davis, Oklahoma, June 1983; received Bachelor of Science Degree in Electrical Engineering (Computer Option) from Oklahoma State University, Stillwater, Oklahoma; completed requirements for the Degree of Master of Science at Oklahoma State University, December, 1989.

Professional Experience: Graduate Research Assistant, Department of Electrical and Computer Engineering, December 1987 to date; Teaching Assistant, Department of Electrical and Computer Engineering, July 1988 to August 1988; Junior Research Engineer, Amoco Tulsa Research Center, June 1987 to August 1987; System Software Engineer, M.A.N. Systems, September 1985 to August 1986.