THE UNIVERSITY OF CALGARY

An Implementation of Ray Tracing Using Multiprocessing and Spatial Subdivision

by

Andrew P. Pearce

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

OCTOBER, 1987

ISBN 0-315-42448-6

The University of Calgary

Faculty of Graduate Studies

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "An Implementation of Ray Tracing Using Multiprocessing and Spatial Subdivision" submitted by Andrew P. Pearce in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor,
Dr. John G. Cleary
Department of Computer Science

Prof. David R. Hill
Department of Computer Science

Dr. Lawrence Turner
Department of Electrical Engineering

Dr. Brian Wyvill
Department of Computer Science

Sept. 25, 1987

Abstract

An Implementation of Ray Tracing Using Multiprocessing and Spatial Subdivision

Abstract

Ray tracing is an image synthesis algorithm which produces highly realistic pictures. The ray tracing algorithm is very CPU intensive, but fortunately, ray tracing lends itself easily to parallel implementations. As a result, research into a multi-processor ray tracing algorithm has been undertaken to speed the ray tracing process.

A first implementation of a multi-processor algorithm is described and the results examined. The space of the scene to be ray traced is divided between the processors and rays are passed as messages between the various processors mimicing their traversal through scene space. To determine when the multi-processor array has completed ray tracing, a new method for determining when all processors are idle is presented. As well, an original method of overlapping processor subvolumes is presented to balance the load between the multi-processor nodes.

In addition to multi-processing, each processor element subdivides it's own space to further speed the ray tracing process. Space is subdivided into uniform subvolumes and an implementation of a fast algorithm for sequentially accessing these subvolumes is discussed. Since a uniform subdivision approach uses significant amounts of storage, the effects of storing the subvolumes using a new, fast hash table

method are examined. A method for saving space by sharing object references between subdivisions is also introduced.

This is the first implementation which uses a combination of both multi-processing and spatial subdivision on each processor to speed up the process of ray tracing.

Acknowledgements

Many people helped me get through this thesis, and I would like to thank all of them. Primarily my supervisor, John Cleary, who always extended ideas to their fullest potential faster than I could, by an order of magnitude. His comments and guidance were invaluable through this writing.

My interim supervisor, Dr. Vasudevan, who showed me what a thesis was, how it takes form, and who convinced me that I could actually write one, deserves heady words of praise.

Brian Wyvill for providing a graphical programming environment. It was his enthusiasm for computer graphics that infected me and got me started on this project.

A special word of thanks goes to Murray Peterson who wrote the multi-tasking kernel for the workstations used in this thesis. Without his helpful suggestions, and a timely re-writing of the network communication software, this thesis would not be.

Darcy Grant, my friend and office partner who was always there with a push to get me started and a drink to land in when I fell. I also would like to thank two people who actually read rough drafts of this thesis and provided useful critisisms, Jeff Allan and Roy Brander, you know who you are.

Without the finacial support of my parents this thesis could not have been. I thank them for their love and support, as I thank Jacquie Bede for hers.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Ray tracing is an image synthesis algorithm which has become popular in computer graphics because it can produce highly realistic images incorporating reflections, transparency and shadows, as natural aspects of the rendering process. The main drawback to ray tracing is that it requires enormous amounts of processing time. This has prevented ray tracing from becoming the standard rendering algorithm in industry.

In recent years, researchers have been investigating ways of reducing the processing time required to ray trace an image. These methods focus on two main areas, multi-processor algorithms and space sub-division. This thesis will detail an implementation which uses both of these approaches.

1.1. Realism in Computer Image Synthesis

The aim of *computer image synthesis*, or *computer graphics*, is to take a geometric description of a surface or an object and produce a realistic image. There are many applications for the end products of computer graphics, such as training simulations for aircraft and ships, viewing design prototypes, architecture, and animation.

The more realistic an image, the more understanding the viewer gains of the scene being depicted. But in producing realistic images, there are two main difficulties; accurately modelling the surfaces and accurately modelling the effect of

light sources on these surfaces. Since increasing the accuracy of the model causes an increase in computational expense, there is usually be some tradeoff between execution time and image realism.

In recent years the complexity of the object models has increased to a point where very few objects cannot be modelled. A *fractal* [Mandelbrot 1983] approach has been used to model mountains and clouds. *Particles* have been used to model fire, grass, leaves, smoke, hair and water [Reeves 1983; Reeves and Blau 1985; Novacek 1985]. objects which deform constantly because of forces imposed on them by their surroundings, such as bouncing balls, flowing liquids or skin [Wyvill, Wyvill and McPheeters 1985]. but it is the rendering process and the lighting model that will determine how realistic these surfaces look.

Most rendering algorithms use a *perspective* transformation on the objects in a scene before rendering to reduce the 3-D sorting problem to a 2-D sort and to utilize some form of coherence. This operation basically projects the objects onto a two dimensional plane as a function of the viewing position and the viewing direction. The visibility of objects is then determined by sorting the objects based on a modified *depth* value produced by the perspective transformation. This modified depth value is not linear with the actual distance of the the object from the view point, and the amount of depth information decreases the further the object is from the view point. This means that it is difficult to determine the order of (and thus the visibility of) objects which are far from the view point.

The perspective transformation also makes it very difficult to model reflection, refraction and shadows. These are all effects which occur in 3 dimensions. The perspective transform, while reducing visible surface determination to a sorting problem, also reduces the ability of the lighting model to determine light interaction

between objects within the scene. In general, approaches which project the surfaces into 2-D before rendering use very approximate lighting models [Newman and Sproull 1973]. Shadows and specular effects are treated as a special case and can require a large amount of extra code.

In order to render these complex surfaces convincingly, a different approach should be taken. Ray tracing provides an alternative. The ray tracing model does not project surfaces into 2-D, it operates in 3 dimensions. Imaginary ''rays'' of light are traced backwards from the view point into the scene. The visible surface is determined by identifying the object that the ray strikes first. The lighting model is an integral part of the rendering process in ray tracing; to determine the shading and intensity of the object struck by the ray, new rays are generated and the tracing routine is called recursively. These rays determine ambient light, diffuse illumination, specular reflection, refraction and shadows. Of the rendering algorithms available, only ray tracing calculates these lighting effects elegantly and consistently.

1.2. The Ray Tracing Algorithm

Ray tracing of a sort was first introduced by [Appel 1968] and [MAGI 1968] for rendering engineering designs, but the algorithm did not come into widespread use until [Whitted 1980] and [Kay 1979] introduced ray tracing models which incorporated specular reflections from other surfaces, refractions, and shadows. This simple addition allowed for extremely realistic modeling of light interactions within a scene.

These additional effects did not mean that ray tracing became practical. It required an enormous amount of computation. If the algorithm is *very* naive, every ray traced is tested for intersection with every surface in the scene. Whitted pointed

out that for simple scenes 75% of the total CPU time is spent testing for intersections between rays and surfaces. As the scene increases in complexity, this figure can jump to 90%. Before discussing techniques used to reduce the number of intersection calculations, let us examine ray tracing more closely and see why it is so computationaly expensive.

A simple way to visualise the ray tracing process is to imagine a rectangular screen grid some distance between the view point and the scene. A primary ray is sent through each *hole* in the rectangular screen grid, originating at the view point and continuing out into the scene. Each of these *holes* correspond to a pixel on the raster screen. At some point after passing through the pixel, the ray may intersect with a surface. At such a time, depending on the properties of the surface, several new rays may be generated which are used to calculate the intensity of that surface.

The Lighting Model

The intensity equation has three components; diffuse, reflected and refracted intensities. Once a primary ray has struck an object in the scene, secondary rays are spawned from the ray's point of intersection with the object. The processing of each of these rays is described in this section.

To compute the intensity of the diffuse illumination and determine if the intersection point is in shadow, a shadow ray is traced from the point of intersection to each light source in the scene. If a shadow ray reaches a light source without striking another surface, then the intensity contribution from that light source is computed. However, if a shadow ray hits another surface before it reaches a particular light source, then the point of intersection is in shadow with respect to that light source. The shadowed light source will not contribute to the intensity of the

intersection point. Note that this type of ray tracing does not take into account intensity from reflected or refracted light sources.

In addition to the diffuse intensity, the intensity of specular reflections and refractions must be calculated. This is done by spawning a secondary ray in each of the reflected and refracted directions. The algorithm is recursively applyed to these secondary rays. The only time a ray will not spawn secondary rays is when a light source or a completely diffuse surface is hit, or when the ray leaves the object space entirely. Once all of the reflected, refracted and shadow rays have been traced to completion, their intensities are summed and the result is the intensity of the pixel which the primary ray passed through.

This process is repeated for each pixel of the frame being rendered. Typical image sizes range from 512×512 pixels to 2048×2048 pixels. This means that for a single image anywhere from $\frac{1}{4}$ million to 4 million primary rays *alone* must be traced. Even if only half of these rays generate secondary and shadow rays the number of rays grows quite quickly to be in the order of 10's of millions.

A diagram of the ray tracing process for a primary ray is presented in Figure 1-1.

Recently researchers have expanded on Whitted's lighting model to account for more accurate diffuse reflections [Dubetz 1985], , soft edged shadows, depth of field and motion blur [Cook, Porter and Carpenter 1984]. While these effects serve to increase the realism of the picture, they also increase the amount of computation that is needed to render an image.

Even more recently a very accurate method for calculation of lighting environments has been presented [Goral, Torrance, Greenberg and Battaile 1984]

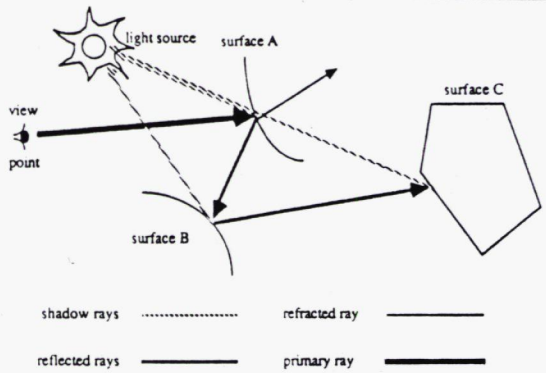


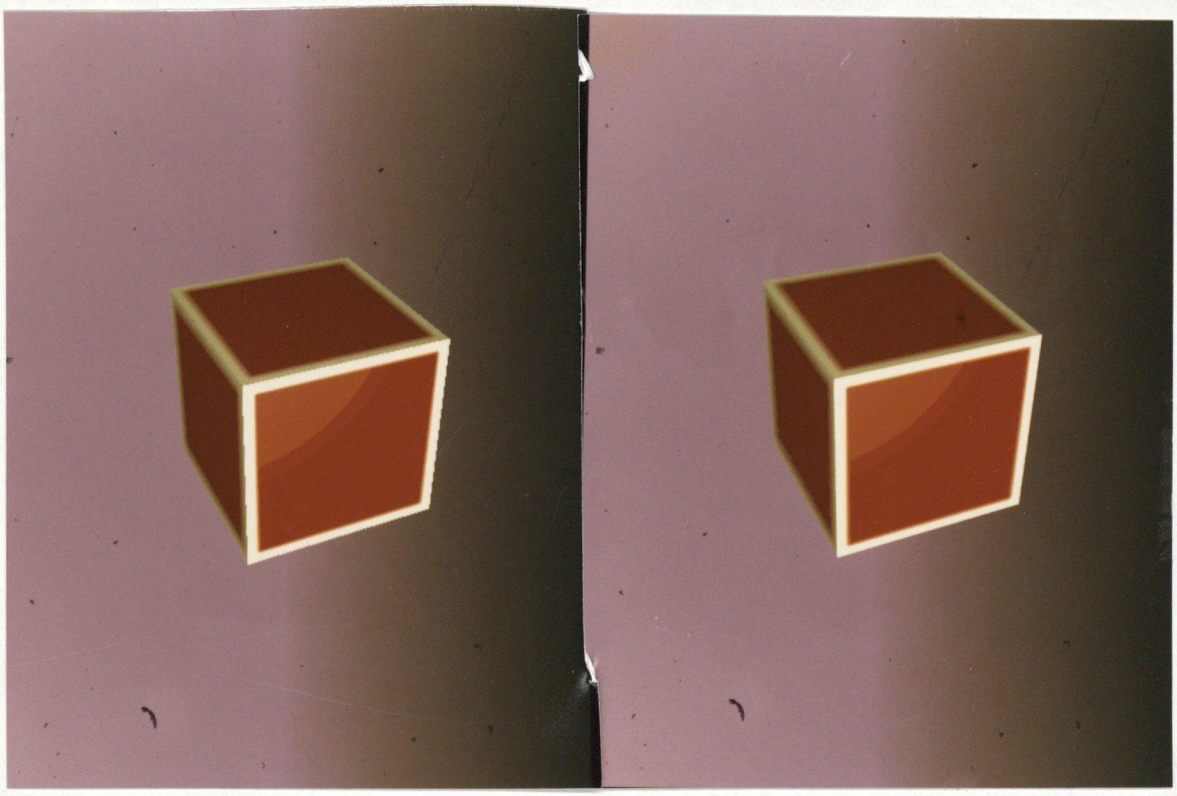
Figure 1-1. Ray tree generated by a single ray.

The technique has been termed radiosity. Instead of simply point sampling the specular, diffuse and ambient terms, radiosity performs an integration of all light arriving at the patch from all directions to produce the intensity reflected. This has produced highly realistic images [Kajiya 1986; Immel, Cohen and Greenburg 1986; Nishita and Nakamae 1986; Rushmeier and Torrance 1987] but it is computationally more expensive than ray tracing. In fact ray tracing is being employed in tandem with this technique to compute the view dependent information [Wallace, Cohen and Greenberg 1987] so the speed of ray tracing remains an important issue.

Aliasing

When an image is completed, the ray tracing algorithm has effectively point sampled the scene space, which introduces the problem of *aliasing*.

Aliasing is the result of low frequency sampling of a high frequency signal. As an example, aliasing usually occurs along the edge of a surface, resulting in the jagged, staircase-like artifacts demonstrated in Figure 1-2. There is no way to eliminate aliasing completely, but the amount of aliasing can be greatly reduced by sampling the scene at a higher frequency than the screen resolution, then passing the image through a filter before displaying it. This process is called *super-sampling* and



Aliased picture Anti-aliased picture

Figure 1-2. Aliasing.

is usually implemented by tracing more than one ray per pixel, with the rays having just slightly different orientations from each other. The filter is then applied to the multiple intensity values in order to produce the single pixel intensity. Super-sampling increases the number of rays anywhere from 4 to 64 times the number of pixels, depending on the frequency of over sampling.

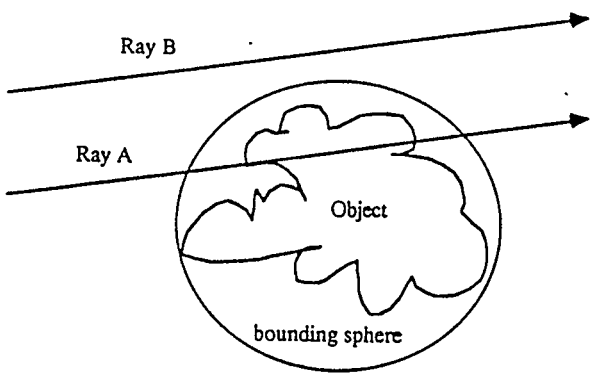
Summary of the Ray Tracing Process

Ray tracing determines which surface is visible from the view point by determining which surface the ray strikes first. Those surfaces which *are* intersected by the ray are sorted based on the distance of the intersection point from the origin of the ray. The intersection point nearest to the origin of the ray is determined. From that point, a ray is traced to each of the light sources to determine shadow information, a ray is traced in the reflected direction (if the surface is reflective) and a ray is traced in the refracted direction (if the surface is transparent). The secondary rays themselves may spawn secondary rays to compute the intensity of surfaces hit by the first secondary rays, and so on. The intensities determined by tracing these secondary rays are added to the pixel intensity, scaled by the reflective or transmittive properties of the surface, and the final pixel intensity is place in the frame buffer.

In this manner, each primary ray can spawn a tree of secondary rays in order to calculate the intensity of a pixel. Since the naive algorithm tests every ray for intersection with every surface, more objects in the scene means a slower throughput of rays. With the number of rays possibly being in the order of millions, it is desirable to have rays processed as quickly as possible.

1.3. Improving Ray Tracing Algorithm Performance

When ray tracing was first introduced, Whitted noted that the intersection calculation was the most time consuming aspect of the algorithm. To speed processing, reducing the number of intersections is of great importance. At the time ray tracing was introduced, Whitted used *bounding* spheres to limit the number of fully detailed intersection calculations per ray. A bounding sphere fully encloses and surrounds a surface. There are two good reasons for using a bounding sphere; the test for intersection with a sphere is fast, and it is a simple matter to find a sphere which completely and tightly surrounds any fully defined surface. If a ray does not intersect a surface's bounding sphere, then it cannot possibly intersect the surface itself. However, even simple sphere checks can consume a great amount of time when there are a large number of objects in the scene.



Ray **A** *is tested for intersection with the object, while ray* **B** *is not.*
Figure 1-3. A bounding sphere.

Image coherence has been exploited to speed up the rendering of images in algorithms such as the depth buffer [Watkins 1970] and Warnock's algorithm [Warnock 1969]. Image coherence is the observation that most scenes will be locally similar; that is, if a given pixel is covered by a surface, then the neighbours of that pixel will likely be covered by the same surface. These same principles can be expanded to three dimensions and recently researchers have been applying *spatial coherence* to ray tracing. Spatial coherence refers to the fact that only those surfaces in close proximity to a ray's path will have any possibility of being intersected by the ray. Surfaces which are distant from the path of the ray are ignored when testing for ray/surface intersections.

1.4. Spatial Coherence

To exploit spatial coherence, Rubin and Whitted [Rubin and Whitted 1980] suggested using arbitrarily oriented rectangular parallelepipeds as bounding boxes (see Figure 1-4). Every surface in the scene is fitted with one of these bounding boxes. Spatial clusters of these bounding boxes are enclosed in higher level bounding boxes, recursively, until some arbitrary upper limit is reached. Once the object space is hierarchically decomposed in this manner, rays are tested for intersection with the top level bounding boxes. If a bounding box is intersected, the bounding boxes contained within it are tested. Eventually, the ray will either pass through all of the bounding boxes or the ray will intersect a surface at the lowest level.

Due to the reduced number of surfaces considered for intersection, the speedup over bounding spheres is great when there are a large number of surfaces. Rubin and Whitted noted that for a bi-cubic surface display, the improvement is about one hundred to one.

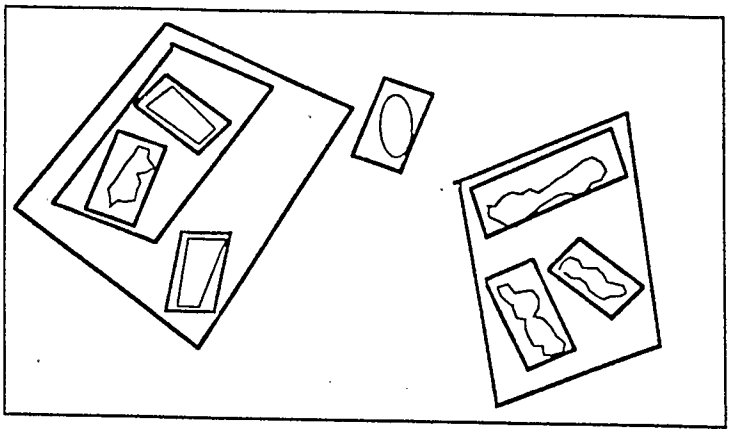
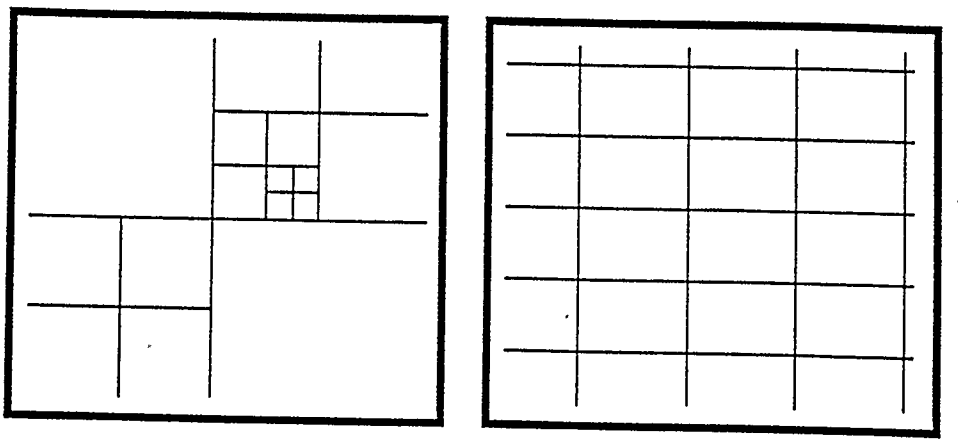


Figure 1-4. Two dimensional hierarchical bounding boxes.

Octree Space Subdivision

The main drawback to bounding boxes is that the expense of determining the next bounding box is quite great. A large number of the boxes may have to be examined and tested for intersection at each level. To improve on this, Glassner [Glassner 1984] proposed instead that the object space be recursively subdivided into subvolumes or *voxels* using an octree structure. Spatial decomposition begins with a bounding box orthogonal to the three axes. This bounding box contains all of the surfaces in the scene. If more than some fixed small number of surfaces (S) are contained within the box then it is subdivided into 8 equally sized subvolumes. The process is recursively applied at each subdivision until all subvolumes have at most S surfaces or until an arbitrary recursion limit is reached. The algorithm focuses on areas of the scene with a large number of surfaces, allowing empty space to be described by a single subspace. A two dimensional example of this type of subdivision is presented in Figure 1-5.



recursive subdivision uniform subdivision

Figure 1-5. Two dimensional subdivision.

Rays are traced through these subvolumes by determining the subvolumes the ray passes through and examining them in the order encountered. The ray is only tested against the surfaces contained within the current subvolume. Determining the next subvolume is done by generating the ray's intersection with the three most distant defining planes of the current subvolume. The intersection closest to the origin of the ray is chosen, and computation continues in the next subvolume in that direction. Since all subspaces are orthogonal to the three axes, the intersection calculation is quick, but still requires several floating point operations.

Since it is not known in advance how deep any subdivision branch of the octree will become, storage for each subdivision must be allocated dynamically. During ray tracing, pointers in the octree must be traversed to find the address at which the next

voxel is stored, and this slows the process of moving to the next voxel. The computational cost of determining and finding the next voxel is high, however, this method requires very little extra memory for storing the octree structure since large empty volumes can be described in a single subspace.

Uniform Space Subdivision

At nearly the same time, another approach was presented by Vatti [Vatti 1984]. Vatti decomposes space into a three dimensional array of equally sized subspaces, or *voxels*, regardless of the density of surfaces in any particular area. The advantage of this method is the speed with which the actual memory address of any voxel can be found. Since the number of voxels ($n \times n \times n$) is known before hand, the array of voxels can be allocated in contiguous memory, and finding the next voxel is simply a matter of adding (or subtracting) 1, n or n^2 to the current index, once the direction of the next voxel has been determined. Since space is subdivided to the maximum granularity, regardless of the number of surfaces in a volume, this method uses much more memory than the octree if there are large empty volumes within the scene.

Fast Octree Voxel Movement

Fujimoto [Fujimoto, Tanaka and Iwata 1986] improved the speed of octree addressing by using a hybrid of octree and uniform subdivision methods. Although traversal of the octree still requires pointer traversal, the *name* (distinct from it's address) of the next subspace is generated quickly using an incremental method based on a technique for drawing arbitrarily oriented lines on a raster display. Each ray is traced through a regular three dimensional grid the size of the subspaces at the current level of the octree. When it comes time to traverse the octree vertically

because of differing levels of subdivision, the parameters of the incremental equation are halved or doubled, depending on the direction of traversal. Since integers are used, this halving or doubling can be done very quickly using a low level *shift* instruction. According to Fujimoto, this next voxel method is 13 times faster than Glassner's. There still exists the problem of finding the actual memory address of the next voxel through a variable number of memory references during the traversal of the octree. Fujimoto implemented both an octree and a uniform subdivision and compared the two. Although his results are fairly ambiguous, they do seem to favour a uniform approach.

The incremental next voxel method used by Fujimoto, is similar to, but distinct from the one implemented in this thesis. The differences will be further detailed in Chapter 2. The method used in this thesis is based on the regular subdivision method presented in [Cleary, Wyvill, Vatti and Birtwistle 1983; Cleary, Wyvill, Birtwistle and Vatti 1986] and uses the fast incremental next voxel calculation proposed in [Cleary and Wyvill 1987]. To reduce the amount of storage needed for regular space subdivision, a hash table representation of the voxels is also implemented and the same fast next voxel calculation is modified to produce the correct hash index iteratively [Cleary and Wyvill 1987]. This thesis describes the first implementation of these methods.

1.5. Multi-processing

In addition to using spatial subdivision, a multi-processor algorithm also offers advantages over the naive algorithm. In most ray tracing algorithms, the results of the rays are independent of each other, making them ideally suited to a parallel implementation. Even algorithms which do exploit ray to ray coherence, such as

beam tracing [Heckbert and Hanrahan 1984; Hanrahan 1986], still have a large amount of parallelism and can benefit from a multi-processor approach.

Image Space Division

There are several methods for applying multiple processors to the ray tracing problem. The simplest to implement is a simple division of image space. Each of N processors is assigned $\frac{1}{N}$th of the pixels, and the whole of object space. The method is simple because the only inter-process communication required is the controller process sending every processor a description of object space and a block of pixels to trace. When the ray tracing is complete on all processors, the blocks of rendered pixels are sent back to the controller for placement in the frame buffer [Sequent 1985]. The method has a slight variation for animation sequences, wherein each processor is assigned a different frame of the sequence, rather than a portion of a single image [Whitted 1985].

A rudimentary form of load balancing can be effected by having the pixels of an image divided into S sections, where $S>N$. When a processor is done it's current section, it asks the controller for another, until the sections are exhausted. This almost guarantees that all processors will do a similar amount of work, since those processors which receive simple sections of the image will eventually render *more* pixels than those processors with more complex sections. This method is so simple and effective, it has been implemented many times [Sequent 1985; Peterson 1983; Williams, Buxton and Buxton 1985].

The main draw back is that each processor requires a large amount of memory to be able to store the entire object space description. In order to have a large number of processors available, and hence a larger speed up, the cost of each processor

should be relatively low. In terms of current technology, this means that each processor might have less than 1 megabyte of RAM. Average scenes currently being rendered at the University of Calgary range from 1 to 3 megabytes for object space descriptions alone, with the largest being near to 24 megabytes. Since this number is not likely to decrease, neither here nor in industry, an alternate approach must be taken.

Pipelined Processors

A multiprocessor approach that has received a lot of attention is the use of pipeline arrays. The Toyo LINKS project [Nishimura, Ohno, Kawata, Shirakawa and Omura 1983] uses an array of 64 processors with subgroups of processors linked together to form pipelines, each pipeline working in parallel with the others. A pipeline is formed out of three processors which perform ray tracing, sorting intersections, and shading respectively. Each pipeline contains all of object space and is responsible for a subset of the image space. Again, this limits the complexity of the scene since every object in the scene must be able to fit into every processor.

Vectorization

Pipelined vector computers, such as the CDC Cyber 205, have also been applied to ray tracing [Plunkett and Bailey 1985]. A large number of rays are stored in a ray queue and sent down the pipeline to be intersected against an object using vector code. The results are stored and the process is repeated for every object in the scene. Once every object has been tested, the intersection points are sorted, the first intersection points for each ray determined, and any new rays (for shadows, reflection, refractions, etc.) are placed in the ray queue. This continues until there are

no more rays to be traced. While the speedups obtained are impressive, the ray tracing algorithm used is very naive. A pipelined processor's power comes from doing a large number of identical operations very fast, such as intersecting a large number of rays against a single object. Better algorithms, such as spatial subdivision, do not translate well since their purpose is to reduce the number of objects which rays are tested against. By reducing the number of rays tested against each object, the ray vector becomes shorter, and the algorithm becomes limited by the speed with which a vector operation can be started. Further, the availability of such high speed *supercomputers* is severely limited due to their high price.

Processor Array with Object Space Subvolumes

Ullner [Ullner 1983] examined several parallel algorithms and parallel architectures for ray tracing. Beyond vectorization of the code and hardware, Ullner suggested a different approach to ray tracing. In his approach, every processor is assigned some sub-volume of the scene being rendered. Algorithms based on this method require a more sophisticated inter-process communication than ones previously described because a ray being traced will pass out of the current processor's subspace at some point. It is then possible that the ray will enter the subspace of a different processor, and the ray must be passed to that processor.

This algorithm and the nature of ray tracing dictate to a certain extent the required physical layout of a ray tracing processor array. Each processor in the array need only communicate with those processors which have been assigned neighbouring sub-volumes of scene space. While this ignores the problems of inputting information into the array and getting pixel intensities out, it does strongly suggest a parallel architecture for the processor array. Since each processor must

communicate with the processors containing adjacent subvolumes, the physical layout should have these logically neighbouring processors close to each other with direct communication channels between them. A two or three dimensional mesh suits this requirement well. Each processor has short physical distance to it's neighbours, making the hardware implementation relatively simple. A broadcast network is undesirable since as the number of processors grows, it will become the bottleneck. Indeed, with the microprocessor network used to simulate this mesh implementation, network collisions were a major problem with only 4 nodes. With an actual mesh, the communication channels can be dedicated to the machines they are connected to, making for more reliable and faster communication.

An implementation of this sort is described by Dippe and Swensen [Dippe and Swensen 1984]. They chose a three dimensional mesh and an adaptive subdivision algorithm for allocating subspaces to the processors. The algorithm attempts to subdivide scene space so that an equal number of surfaces are assigned to each processor much like the octree method. Further, as ray tracing continues, if any processor becomes too busy, boundary shifting is initiated so that the less busy neighbouring processors can take over some of the work. To avoid having to redistribute the load on many processors (as would happen if the subvolumes were constrained to be orthogonal parallelepipeds) they allow the processor's subvolumes to become general cubes. In that way, only one corner of a subspace has to be moved, involving only that processor and the 7 neighbours which share that corner.

Two-dimensional versus Three-dimensional Processor Arrays

The reason that Dippe and Swensen chose a three dimensional processor mesh is not clear, both Cleary [Cleary, Wyvill, Birtwistle and Vatti 1986] and Ullner [Ullner

1983] favour the two dimensional mesh. Cleary shows that the three dimensional mesh will provide a speedup of $N^{\frac{2}{3}}$ for N processors, however the two dimensional array initially provides a speedup of N and approaches $N^{\frac{1}{2}}$ as N increases. The two dimensional array initially performs better, but eventually will lose to the three dimensional array at some N. Cleary then shows that the N where this occurs is at least 10,000. Ullner points out that with the three dimensional case, "the number of processors in the array increases as the cube of the number of subdivisions...". However he also shows that the number of non-empty subvolumes increases only quadratically. This means that with a large N most of the processors in a three dimensional mesh will have only empty space and thus be wasted. A three dimensional mesh also provides physical difficulties with cooling the central processors and accessing the central processors for repair.

Load Balancing

A further difficulty with Dippe and Swensen's method is the adaptive redistribution process. While load distribution is an important consideration, with a three dimensional mesh the load will initially be shifted towards where the rays enter the scene, as ray tracing continues, the load will once again have to be shifted back towards where the rays are leaving the scene. All this shifting requires processing time which takes away from the ray tracing process.

The issue of load balancing is an important one, even when considering a two dimensional mesh. The busiest processor will determine the amount of time taken to ray trace a scene fully. To keep the speedup as close to N as possible, it is desirable to keep each processor equally busy. Load balancing is an area that requires attention. The results presented in Chapter 4 support this. With 9 processors, the difference in

node completion times can vary as much as 75%. There is no reason why Dippe and Swensen's redistribution algorithm could not be adapted to two dimensions with good results. The problem with the initial load shifting to the front and subsequently to the rear of the mesh would not occur in two dimensions. However the use of general cubes makes space subdivision on each node difficult.

To avoid this difficulty, Chapter 5 presents a new load balancing algorithm that uses overlapping subvolumes to even the load between the nodes.

1.6. Summary

A uniform subdivision approach was choosen because of the slight bias towards it shown in Fujimoto's work and because the voxel passing algorithm presented by Cleary was faster than Fujimoto's. It was thereby hoped that this implementation would fair even better.

A 2 dimensional mesh approach to multi-processing was choosen because it was one of the few methods with any sort of theoretical study done on it. A 2-D mesh has a better intial speedup predicted, and has less empty space processors as the number of nodes increase.

1.7. Current Work

This thesis presents an implementation on a square two dimensional torus mesh. Uniform subdivision is used on each node of the mesh to enhance performance further. The mesh approach was chosen because the processor nodes are relatively inexpensive and readily available in today's market. Processors such as the Motorola 68000 series can be used in sufficient numbers to produce reasonable speedups and their co-processor facilities allow for high individual node performance. A full

discussion of the multiprocessor algorithm used is presented in Chapter 3.

CHAPTER 2

Object Space Subdivision

In my work, the uniprocessor ray tracing algorithm on each node of the multi-processor uses equally sized subdivisions to form voxels through which rays will be traced. This chapter will consider only a uniprocessor implementation of this algorithm. It is then applied to each individual node of the multi-processor array as discussed in Chapters 3 and 5. This chapter will detail the algorithm used to subdivide object space, the structures used to store the voxels, and the method used to determine the next voxel. Two separate methods of voxel storage were implemented in this thesis; an array structure and a hash table structure. Both are described in this chapter, and a comparison based on results from actual scenes is presented in Chapter 4.

Spatial Coherence

As was described in the previous chapter, the number of intersection tests performed on each ray can be drastically reduced by exploiting spatial coherence. This means that only surfaces which lie near the ray's path are tested for intersection, and the surfaces which are nearest to the origin of the ray are tested first. This is effected by using a uniform subdivision of object space in 3 dimensions.

The division of object space begins by determining the scene extents. The scene is assumed to lie within some finite bounds in X, Y and Z which can be determined by examining the extents of each object. This is an $O(N)$ operation for N objects, but this examination can be paired with the input of the surfaces, incurring minimal

overhead.

Once the scene extents are known, the number of subdivisions in each of the X, Y and Z direction will dictate the size of the voxels. The current implementation allows the number of divisions to be chosen by the user, but there is no reason why some form of heuristic based on object distribution could not be used to arrive at these numbers. However, such a heuristic is beyond the scope of this thesis and was not investigated.

After the voxel size is fixed, the surfaces in the scene must be assigned to the voxels through which they pass. Although the algorigthm used to do this is not detailed in this thesis, it is important to note here that only the *surface* of an object is assigned to the voxels. For example, those voxels which are inside a sphere, but which do not contain part of the sphere's surface, will not contain a reference to the sphere. If an object's surface area is small in comparison to it's volume, this can save an enormous amount of storage.

2.1. Using an Array for Voxel Representation

The voxels themselves are represented by an array of pointers to chains of object references. An empty voxel is represented by a *NIL* pointer. If the pointer is not *NIL* then it will point to an object reference, which consists of an object identifier and a pointer to another object reference. If there is more than one object in the voxel, then the objects hang off the voxel in a linked list. No special ordering of multiple object references is performed.

Tracing Rays Through the Voxels

When a ray is traced through the scene, it will travel through some sequence of voxels. A ray travelling through uniform voxels will generally encounter more voxels than if it were travelling through an octree simply because of the manner in which object space is partitioned by each algorithm. So to outperform the octree method, identifying and referencing each voxel must be a fast operation. To do this, two versions of the next voxel calculation presented in [Cleary and Wyvill 1987] are used. The next voxel calculation is similar to the one described in [Fujimoto, Tanaka and Iwata 1986] in that it is incremental. However, Fujimoto checks and updates two *error* terms on two separate planes, using a modified Digital Differential Analyser (DDA) originally developed for drawing lines on a raster display. While this method is fast, it is coupled with an octree subdivision, and referencing the next voxel may require an octree traversal to another node. Checking for the traversal condition, and the octree traversal itself make the number of instructions for the next voxel calculation much larger than the number of instructions needed for uniform space subdivision. The method presented in [Cleary and Wyvill 1987] is also incremental, but requires many fewer instructions.

Initializing the Next Voxel Calculation

Using uniform subdivision, the distance a ray travels between intersections of voxel walls in any one of the three directions (X,Y or Z) is a constant for that ray. For example, if we consider the voxel divisions in the X direction, each one is equally spaced, so a ray will travel a constant distance between each intercept of a voxel division in the X direction (and similarly in Y and Z). Since this number is a constant, but different for each ray, it can be calculated at the outset of each ray (see Figure 2-

1). Let us call these distances Δ_i, where i can take on a value of x, y, or z.

Since the origin of the ray is necessarily known at the outset, the distance to the next intercept of a voxel wall from the origin of the ray is easily calculable. Let these values be labeled $dist_i$, where i can take on a value of x, y, or z. See Figure 2-2.

If we then let (i,j,k) be the indices into the voxel array, then each time a next voxel is picked, we can increment or decrement (depending on the ray's direction of travel) the appropriate voxel index by ± 1 depending on the direction of the ray's travel.

However, this requires that the index to $voxel[i][j][k]$ be decoded each time *voxel* is referenced. Since three dimensional arrays are kept in contiguous memory, the real index is determined by $i*v^2 + j*v + k$ where *voxel* is declared as an $v \times v \times v$ array. Speed can be gained if *voxel* is declared as a single linear array, $voxel[V]$ where $V = v*v*v$. To step to the next voxel, the voxel index is incremented by $\pm v^2$,

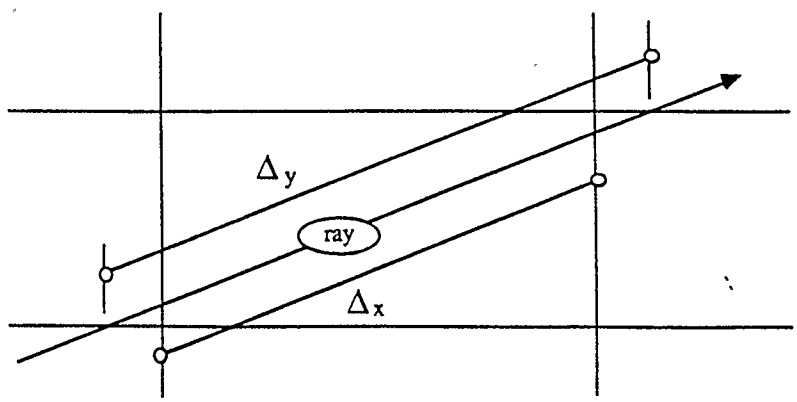


Figure 2-1. Initializing "delta" values.

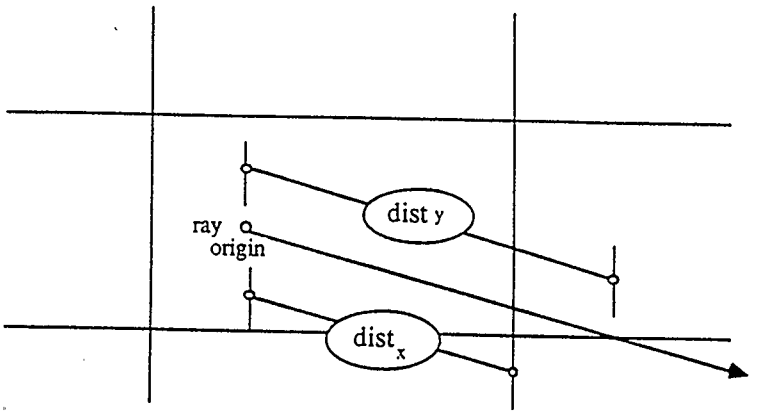


Figure 2-2. Initializing "dist" values.

$\pm v$ or ± 1 depending on which index must be updated. Let these values be labeled p_i, where i can take on a value of x, y, or z. The value of *small* is set simply by determining the smallest $dist_i$.

Calculating the Next Voxel

Once these values have been initialized, determining the next voxel is done by determining the smallest $dist_i$, incrementing it by the appropriate Δ_i, and incrementing the voxel index by the appropriate p_i. The array next voxel algorithm is presented in Figure 2-3.

A further speedup may be gained by noting that once the $dist_i$ and Δ_i values have been determined, they are used only to compare with each other, so they can be converted to integer values by normalizing them.

```
{ "small" has been initialized to the index of the smallest dist_i }

{ update the voxel index and increment the next voxel variables }
voxel_index = voxel_index + p_small;
dist_small = dist_small + Δ_small;

{ now reset small }
if (dist_x < dist_y) small = x;
else small = y;
if (dist_z < dist_small) small = z;

{ and finally, check the voxel to see if there is anything in it }
voxel_full = voxel[voxel_index];
```

Figure 2-3. The array next voxel calculation.

Using the C programming language, the next voxel calculation can be reduced to 4 lines of code, or about 12 VAX assembler instructions. A more detailed account on how this can be achieved is presented in [Cleary and Wyvill 1987]. More elaborate assembler instructions can further reduce this to an average $7\frac{1}{2}$ instructions.

2.2. Using a Hash Table for Voxel Representation

The spatial subdivision method presented achieves it's goal of reducing the number of intersection calculations per ray as presented in Chapter 4. However, uniform space subdivision uses a large amount of memory storing empty voxels. In [Cleary and Wyvill 1987] a hash table voxel representation is presented to reduce the storage requirements of a uniform subdivision approach. The hash table method

was also implemented in this thesis, and this section describes the differences from the array version.

Instead of using a V length voxel array, a hash table of length M is used (where $M < V$). When a voxel is accessed, it's index (idx) is hashed to create a hash table index. By using a simple hashing function such as (idx **mod** M), the next voxel operation is only slightly changed. After each next voxel operation, idx is hashed into M. In this implementation, M is chosen to be a power of 2 so that a simple (and fast) masking operation can replace the **mod** operation. The results from using this type of hashing function are presented in Chapter 4.

Initializing the Next Voxel Operation

The values of p_i discussed in the last section must be modified to p_i **mod** M if the ray is travelling in a positive direction in that dimension, or $M - (p_i$ **mod** $M)$ if travelling in a negative direction. This produces only positive numbers as hash increments (h_i) and so no check for hash index underflow is needed in the next voxel calculation. The other values, Δ_i and $dist_i$, are left unchanged from the array version. The modified algorithm, with the additional check needed for hash index overflow, is presented in Figure 2-2.

Using a Bit Table to Assist the Next Voxel Operation

When tracing a ray through the voxels, each time a non-empty hash table entry is encountered, the next voxel loop is left and the object references in the hash table entry are examined. But since each hash table entry has multiple voxels hashed to it, there may not be an object reference which is in the current voxel. All of the references may just be in the same hash table entry by coincidence. To avoid leaving

the next voxel loop until there actually are objects in the current voxel, a bit table is employed. There are V bits in this table, one bit for each voxel. When a voxel contains a reference to an object, the corresponding bit representing that voxel is set to 1 and an object reference is placed in the hash table. When tracing the rays, both the bit index and the hash index are updated, but instead of checking for a **NIL** pointer in the hash table, the bit table is examined. Only when a 1 is found in the bit table is the next voxel loop left. Extra code is added to the next voxel calculation, but since the bit table index is identical to the voxel index, the extra code consists of 2 shifts and 2 mask operations.

Using the hash table method, each object reference contains an extra field which identifies the voxel which it was hashed from. When a 1 is found in the bit table, the actual voxel index is used to check the object references in the current hash table entry. In this way only the objects which are in the current voxel are tested for intersection, the objects which are coincidently hashed to the same hash table entry are ignored.

The Next Voxel Operation Using the Hash and Bit Tables

The hash table next voxel calculation is given in Figure 2-4. In the figure, **SHIFT_RIGHT** *and* **SHIFT_LEFT** mean the bit patterns on the left of the operand are shifted right or left (respectively) the number of bits indicated on the right of the operand. The instruction, **AND** , means do a logical ''and'' on the two operands. The bit pattern, *M_MASK*, is the appropriate mask for the hash table. This mask operation takes the place of a **mod** operation since the length of the hash table is chosen to be a power of two.

```
{ "small" has been initialized to the index of the smallest dist_i }
hash_index = hash_index + h_small;  { increment hash index }
hash_index = hash_index AND M_MASK;  { do the hash function }
voxel_index = voxel_index + p_small;  { increment the bit index }
dist_small = dist_small + Δ_small;  { update the distance to next voxel wall }

{ reset small }
if (dist_x < dist_y) small = x;
else small = y;
if (dist_z < dist_small) small = z;

{ calculate the bit table index }
bit_shift = voxel_index AND 07;  { get value of lower 3 bits }
bit_mask = 01 SHIFT_LEFT bit_shift;  { make bit_mask point at correct bit }
bit_index = voxel_index SHIFT_RIGHT 3;  { divide by 8 }

{ and finally, check the bit table }
voxel_full = bit_table [bit_index] AND bit_mask;
```

Figure 2-4. Next voxel operation using the hashing function.

2.3. Determining When a Ray Leaves Object Space

When using spatial subdivision it is important to determine when a ray has left the volume which are described by the boundaries of the voxels. Failure to identify when a ray leaves object space will cause the voxel (or hash) index to indicate an erroneous voxel or be outside of the voxel array bounds all together. Since it is undesirable to add extra code to the next voxel calculation, an extra layer of voxels is placed around the scene and bounding polygons are placed in these extra voxels so as to fully enclose the processor's subvolume. When a ray enters a voxel containing a bounding polygon, that ray is terminated.

The algorithm presented in this chapter can be used not only on a uniprocessor, but it may be used on each node of a processor array as well. In the multiprocessor approach, the ray is leaving the object space of the current processor and it is passed on to the next appropriate processor (if any) to continue being traced. If there is not a processor responsible for the subvolume that the ray is entering (it has left all of scene space), the ray is tested to see if it intersects a light source and any colour contribution it carries is returned to the appropriate pixel.

This particular approach does not require any extra code within the next voxel calculation and adds very little code elsewhere. When a ray is in a voxel containing a bounding polygon, no intersection calculation with the bounding polygon is needed since it is guaranteed to be leaving the processor's space. This means that care must be taken when starting the ray, it must be started within the processor's subvolume, past any initial bounding polygons. This simply requires a small number of next voxel calculations (usually one) at the outset of tracing a ray which is entering the subvolume from outside.

Determining the Next Processor

With a multi-processor approach, when a ray encounters a bounding polygon, it is leaving the subvolume of the current processor. The next subvolume (and thus the next processor) which the ray enters is determined using the bounding polygon references. Each bounding polygon reference contains a unique id number. This id number identifies which bounding polygon is encountered by the ray, and the direction of the next processor is easily determined.

An alternate method of determining the next processor is to use the values in the next voxel calculation. The bounding polygon that the ray encountered can be

determined from the state of the next voxel calculation variables. The voxel wall through which the ray has just passed can be reconstructed by subtracting Δ_i from $dist_i$ for all three directions (x, y and z). This produces three values, the largest of which will indicate the direction that the ray just moved.

2.4. Avoiding Multiple Intersection Tests

Spatial subdivision reduces the number of intersection calculations that are performed, but it introduces the possibility of testing an object for intersection more than once. This can occur because, with any significant number of voxels, there will be surfaces which are assigned to more than one voxel (see Figure 2-5). To avoid testing a ray against the same object in multiple voxels, it must be determined if a particular ray has been previously tested against an object. To do this, each ray is

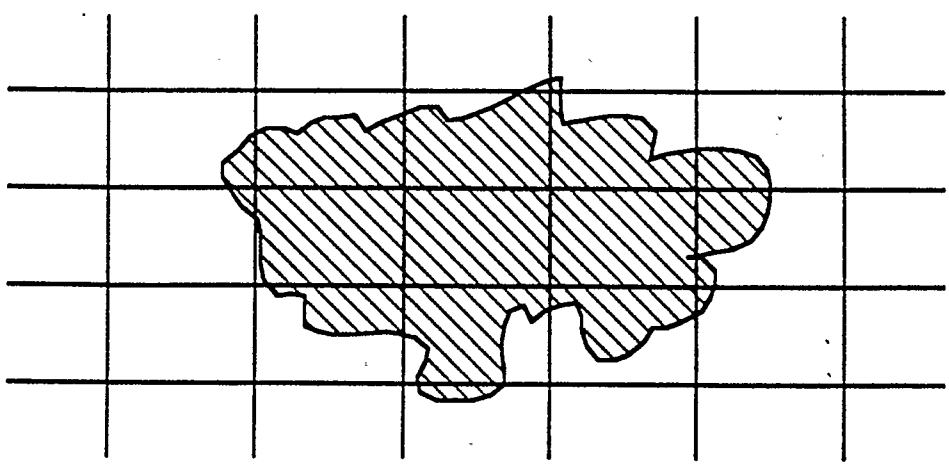


Figure 2-5. A surface assigned to multiple voxels.

assigned a unique identifying number, with each reflected, refracted and shadow ray's number being different from it's parent's. When a ray is tested for intersection with a surface, the unique ray number is stored in that surface descriptor. At each voxel, the ray's unique number is checked against the number stored in each of the surface descriptors in that voxel. If the numbers match, then the ray has already been tested against that surface and no further processing on that surface is performed. This requires an extra field be stored with each object descriptor.

Intersections Which Do Not Occur in the Current Voxel

Since a ray is tested against an object only in the first voxel where it is encountered, a case can arise where an intersection is found, but that intersection occurs in a different voxel. The processing of the ray cannot stop at this point because a nearer intersection may be found in the intervening voxels between the voxel where the intersection was calculated and the voxel where it actually occurs. Figure 2-6 shows such a case. The ray is tested for intersection with surface **1** in the first voxel where it is encountered, voxel **A**. An intersection is found, but that intersection does not occur until voxel **C**. If the processing of the ray is halted in voxel **A**, then the real nearest intersection with surface **2**, in voxel **B** is not found. To avoid this, the voxel where the intersection actually occurs and the intersection point are stored in global variables. Only one set of these global variables is needed because only the nearest intersection point has to be stored. If a nearer intersection point is found in the intervening voxels, then the saved intersection point can be over-written.

Once the voxel and intersection point are stored, tracing of the ray continues until the ray either arrives in the voxel where the intersection occurs, or a nearer intersection is found. Upon arriving in the intersection voxel, all of the surfaces in

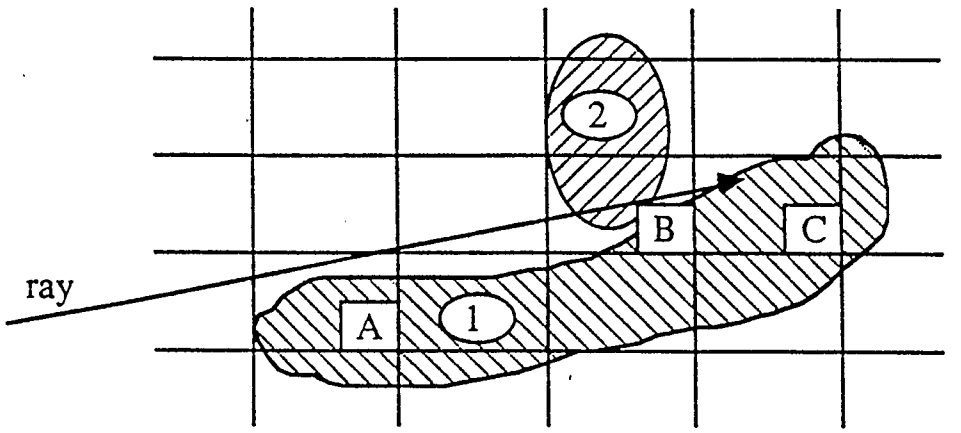


Figure 2-6. A ray finding an intersection that is not in the current voxel.

that voxel are checked (unless already checked) and if no nearer intersection is found, the stored intersection point is used. This means a ray will be intersected against any particular object a maximum of one time.

2.5. Reducing the Storage Requirements Further

As presented, the hash table representation for the voxels reduces the storage requirements for this algorithm significantly. There are further storage savings possible for both the array representation and the hash table representation. This section details the space saving methods used for both of these approaches.

Saving Space on Object Array Voxel Assignment

Cleary and Wyvill [Cleary and Wyvill 1987] have analysed the storage requirements for uniform space subdivision. They assume that every voxel is assigned a unique object reference for *each* object which intersects it. However, space can be saved at some small speed cost during assignment of objects to voxels. If an object reference does not point to another object reference, then it can be shared with all of the voxels which also contain that object reference, but do not yet point to other object references. To do this, initially only one object reference is created for the current object. If a voxel already contains an object reference, then a new object reference must be allocated and prepended to that voxel's list. Otherwise, if the voxel is empty, the initial object reference may be used, since no other object reference is yet pointed to from this voxel. Since this algorithm is followed for every object, the first object reference assigned to every voxel will be shared with every other voxel where that object is the first assigned, and space is saved since an object reference is not allocated for every voxel. This "shared" algorithm can be extended to perform the same operation for voxels which contain the same multiple objects, (such as cases where two or more polygons meet through many voxels) but the algorithm becomes quite difficult and slow. The single object reference method is presented in Figure 2-7. The results from this method are presented in Chapter 4, and they show a large reduction in the number of object references stored.

However, this method will not work with the hash table. A single hash table entry has multiple voxels hashed to it, so each object reference must have an extra field which identifies which voxel it is really in, thus making it impossible to share object references between voxels.

```
shared_object_reference = NIL ;
for (all_voxels_intersected) do
        if (voxel[i][j][k].pointer ≡ NIL )
                { There is no object in this voxel yet so use the }
                { shared object reference. }
                if (single_object_reference ≡ NIL )
                        { The shared object reference has not been }
                        { allocated yet, so create it. }
                        allocate (single_object_reference);
                        single_object_reference.id = current_id;
                        single_object_reference.pointer = NIL ;
                endif ;
                { Put the shared object reference in the voxel. }
                voxel[i][j][k].pointer = single_object_reference ;
        else
                { The voxel already has an object reference in it, }
                { so a new one must be allocated. }
                allocate (new_object_reference);
                new_object_reference.id = current_id;
                new_object_reference.pointer = voxel[i][j][k].pointer;
                voxel[i][j][k].pointer = new_object_reference ;
        endif ;
enddo ;
```

Figure 2-7. Assigning object references to voxels.

Reducing the Storage Required for Bounding Polygons

The drawback to bounding polygons is the extra memory required to store both the references to the bounding polygons and the extra voxels themselves. While storage for the extra voxels *must* be allocated, it is possible to save some space by having all the non-edge voxels on each side point to the same bounding polygon descriptor (6 references, one for each bounding polygon). The edge and corner voxels where more than one bounding polygon meet are a more difficult case. However,

since rays can not move to a next voxel diagonally, the edge and corner voxels do not need to have accurate bounding polygon references. There needs to be *some* bounding polygon reference in these voxels to allow rays entering the scene to determine when they have passed all of the initial bounding polygons, but so long as there is *at least* one reference, it does not matter which bounding polygon is referenced. Thus only 6 references are used to represent all of the bounding polygons in the voxels regardless of the size of v. Further, since processing of the ray stops once a bounding polygon *reference* is encountered, there does not have to be an actual bounding polygon allocated.

Eliminating Bounding Polygons with the Bit Table

Using the hash table method, each bounding polygon reference needs to be unique, so that the original voxel from which it was hashed can be identified. Yet this means that $v^2 \times 6$ references to bounding polygons must be used. To avoid this extra storage, the bit table, can be manipulated so that no bounding polygon references need to be used at all. When assigning the bounding polygons to voxels, only the bit table entry is modified to be 1, and nothing is placed in the hash table. When tracing the rays, if the bit table is set to 1 but there is no corresponding reference for the current voxel in the hash table, then a bounding polygon has been hit and the appropriate action is taken. The bit table was intended to speed up processing, however in this case it helps save space as well.

While the bit table requires extra storage, if each object reference is 12 bytes using the hash table method, the size of v where the bit table starts to cost more than the number of object references is :

$$\frac{v^3}{8} = (v^2 \times 6) \times 12 \tag{2.1}$$

that is:

$$v = 576 \tag{2.2}$$

Since results indicate that the best v for most scenes is less than 100, it is unlikely that the bit table will ever use more storage than the bounding polygon references.

2.6. Summary

Both the hash table and the array method of representing voxels have been introduced and discussed. The hash table and the space saving assignment reduce the amount of storage needed for a uniform subdivision approach. This spatial subdivision method is used to improve the individual node performance of the multi-processor array as discussed in the next chapter.

.pn 39

CHAPTER 3

Multi-processor Ray Tracing

The idea of using a multi-processor approach to ray tracing was discussed in the first chapter, this chapter will detail the algorithm and hardware used in this implementation. There are new difficulties introduced when using a multi-processor approach which are also examined in this chapter. One of the most notable problems is changing the state of the processor network. A new *distributed synchronization* state changing algorithm is presented to overcome the state change problem in an elegant way.

The multi-processor algorithm assigns each node of a two dimensional processor network some subvolume of total object space, and each node ''knows'' about every object which intersects it's assigned subvolume. That node is then responsible for tracing every ray which passes through it's subvolume. Once a ray is terminated, pixel intensities are returned to a host processor for addition to a frame buffer. The processor network consists of low cost processors with fast communication channels to their neighbours. Since none of the nodes have backing store, one of the nodes is connected to a host processor which has backing store, but is not part of the processor network. All of the computed pixel values are sent to the host via the one processor connected to the host. The host processor is also responsible for downloading the scene description to the nodes.

3.1. Dividing Object Space

To subdivide object space between the nodes, the host processor starts by assigning each node a roughly equal number of pixels. If the size of the pixel array is $P = R \times C$, where R stands for 'row' and C stands for 'column', then each processor is assigned an $\frac{R}{k} \times \frac{C}{k}$ pixel rectangle, where the number of nodes is $K = k \times k$. The number of pixels will not always be evenly divisible by k, so a processor may be assigned one extra row or column of pixels, but if P is large, this difference should not be significant. At the outset of ray tracing, every node will have a nearly equal load, that is, an equal number of rays to trace.

Once the number of pixels for each node is determined, the master process on the host subdivides object space into K subvolumes. The sizes of the subvolumes are determined by the $\frac{R}{k} \times \frac{C}{k}$ pixel rectangles in the following manner. The primary rays which pass through a pixel rectangle must all enter the same subvolume, V_K so that they are all traced by the same node. This is because creating rays which will immediately be passed to a node with the appropriate subvolume creates more overhead. To avoid this unwanted overhead, the pixel rectangles are paired with the subvolumes so that all primary rays enter the subvolume of the node they are assigned to. So to determine the size of the subvolume for a processor, a test ray is traced to the front of object space from the four corners of the pixel rectangle. The intersection points generated from the intersection of these test rays with the front of object space define the (X, Y) extents of that processor's subvolume.

To provide a margin for error, these subvolumes are *grown* by 1% in each direction to allow a slight overlap between node subvolumes. This overlap avoids floating point precision problems near subvolume boundaries. The idea of growing

the subvolume boundaries is extended further in Chapter 5 to provide a load balancing algorithm.

To choose which node is assigned a particular subvolume, the master process determines which nodes are neighbours and then assigns the subvolumes to the nodes so that neighbouring nodes contain contiguous subvolumes. This allows rays to pass through the entire object space with minimum communication overhead.

Downloading Objects

Once the subvolumes have been assigned to the nodes, objects are transmitted from the host to the primary processor. The current implementation only ''knows'' about three types of objects; polygons, spheres and sources. However, adding further object types could be done with relative ease.

When a node receives an object, it must determine if the object intersects the subvolume of space assigned to it. If the object is a polygon, it is clipped against the node's subvolume, and anything clipped off is sent to the processor in the appropriate direction. Any portion of the polygon that is left after clipping is kept on the node.

Spheres cannot be clipped, so if the sphere intersects the processor's subvolume, the entire description is kept. If any of the sphere extends beyond the node's subvolume, the full description of the sphere is sent to the processor(s) assigned to the neighbouring subvolume(s).

Since the possibility of an endless loop exists with this type of algorithm, object start in the lower left corner of the processor array and once clipped, they can only be passed up or right until they reach the upper right corner of the processor array. In addition, each object has an associated and unique identifying number with to prevent a node from receiving a single object twice.

Light sources are the only objects which are not clipped. It is necessary to keep all of the light sources on each node so that the direction of shadow rays can be determined. There are usually less than 10 light sources so this poses minimal storage overhead.

There is a drawback to downloading objects in this manner. The primary processor must clip every object in the scene regardless of which node(s) the object will eventually be stored on. This creates a bottleneck at load time. However, the startup time for scenes in this implementation was so small in comparison to the trace time (less than 1%) that a more optimal algorithm was not investigated. A better method is presented in [Vatti 1984].

Once all of the polygons, spheres and sources have reached the proper nodes on the mesh, ray tracing begins. Determining when the mesh has finished clipping the objects is a difficult problem, but it is resolved by using a *distributed synchronization* state change which is described in Section 3.4.

3.2. The Ray Model

After the objects have been downloaded, ray tracing starts by creating primary rays through the pixels assigned to each node. To keep the storage requirements as low as possible, only one primary ray is created at a time. The ray is then placed in a *ray queue* to await processing. If a ray arrives from another processor it is also placed in this queue.

Tracing the rays is then very simple. This process is detailed in Figure 3-1. The messages queues are checked for any incoming messages. If any rays are received they are placed in the ray queue. Should the ray queue still be empty after checking the message queues, then a new primary ray is created and placed in the ray queue.

```
while (there_are_still_busy_nodes) do
        check_messages ();
        if (ray_queue_is_empty)
                make_new_ray ();
                { this function does nothing if there are no more new }
                { rays to be made }
        endif ;
        current_ray=get_ray_off_queue ();
        if (current_ray ≠ NIL)
                trace_ray (current_ray);
        endif ;
        if (shadow_ray_queue_is_not_empty)
                shadow_ray = get_shadow_ray_off_queue ();
                trace_shadow_ray (shadow_ray);
        endif ;
enddo ;
```

Figure 3-1. Main ray tracing loop.

The first ray of the queue is then removed and traced through the node's subvolume, and the process is repeated. The condition "there_are_still_busy_nodes" is determined by using the state change algorithm described in Section 3.4.

Shadow rays are slightly different from other rays, both in structure and in the way they are traced. For this reason they are maintained in a separate ray queue.

Carrying Colour Information With the Rays

This section deals with how rays carry colour intensity information with them as they travel through the processor mesh. The collection of colour information once the ray is finished being traced is detailed in Section 3.3.

To carry the colour information, a ray has several fields in addition to the origin and direction. These fields and their function are shown in Table 3-1. The colour intensity ($C_{r,g,b}$) of a primary ray is initially set to 0. When a ray intersects an object, colour information from the object is added to the ray. This colour information consists of 3 terms; ambient, diffuse, and specular (reflected and refracted light). Pseudo-code for this method is presented in Figure 3-2. The intensity of the surface due to ambient light is calculated first and this value is placed in the ray. Then the diffuse intensity is determined. This may happen in two ways. If shadows have not

Ray Structure	
Field	Function
$\vec{D}_{l,m,n}$	The ray's direction vector.
$O_{x,y,z}$	The ray's origin.
$Pixel_{x,y}$	The index of the pixel through which the primary ray passed. If it is a secondary ray, this field is copied from the ray that spawned it. This field indicates where the colour intensity from the ray will be placed in the frame buffer.
$C_{r,g,b}$	The colour intensity accumulated by the ray so far. (initially 0)
$Contribution_{r,g,b}$	A floating point value describing what percent of total pixel intensity will be contributed by any object which the ray hits. (initially 100%)
Level	The number of predecessors that this ray has. Allows for a limit on the number of secondary rays spawned. (initially 0)
Medium	Describes the medium in which the ray is travelling (ie. inside an object or outside in the 'air'). Important for calculating the direction of the refracted ray.

Table 3-1. The ray structure.

```
trace_ray_through_voxels (ray); { as described in chapter 2 }
if (ray_hits_an_object)
        calculate_ambient_lighting (ray.C_r,g,b, object_hit);
        create_shadow_rays (); { shadows and diffuse intensity }
        if (object_is_reflective)
                create_reflected_ray (reflected_ray);
                { transfer the colour to the reflected ray }
                reflected_ray.C_r,g,b = ray.C_r,g,b;
                { and set the incident ray's colour to 0 so that it is }
                { not also transferred to the refracted ray (if it exists) }
                ray.C_r,g,b = 0;
                reflected_ray.Contribution = ray.Contribution × object_reflectivity;
                { transfer other information... }
                add_ray_to_queue (reflected_ray);
        endif;
        if (object_is_refractive)
                create_refracted_ray (refracted_ray);
                if (ray.C_r,g,b ≠ 0)
                        { if the colour was not transferred to the reflected }
                        { ray, transfer it to the refracted ray }
                        refracted_ray.C_r,g,b = ray.C_r,g,b;
                        ray.C_r,g,b = 0;
                endif;
                refracted_ray.Contribution = ray.Contribution × object_refractivity;
                { transfer other information... }
                add_ray_to_queue (refracted_ray);
        endif;
        if (ray.C_r,g,b ≠ 0)
                { there was not a reflected or refracted ray, the object was a }
                { total diffuser so return the colour information to the host }
                send_colour_to_host (ray.C_r,g,b);
        endif;
endif;
```

Figure 3-2. Transferring ray information.

been requested by the user, then the diffuse contribution from each light source is simply added to the ray. If shadows have been requested by the user, then a shadow

ray is spawned towards each light source. If this occurs, then there is no colour information added to the ray, since the shadow ray will carry the diffuse contribution to be made by the light source. The tracing of shadow rays is detailed later in this section.

Once the ambient and diffuse intensities have been calculated, the specular component of the ray is determined. If the object struck is reflective or refractive, then reflected and refracted secondary rays are spawned. In the serial implementation the algorithm recurses on each of these secondary rays. Upon return from the recursion, the colour values of these secondary rays have been determined. The intensities of the secondary rays are summed with the colour intensity of the incident ray and the resultant intensity returned. However, a subvolume based multi-processor algorithm can not use this method. The secondary rays can leave the node's subvolume and continue to be traced on another processor. During this time the node which spawned the secondary rays would be idle, waiting for the colour values of the secondary rays to be returned. Since having idle processors wait for results while there are still more rays to be traced is nonoptimal, and bound to cause dead-lock (processors waiting for results from processors waiting for results, etc.), a different tack must be taken.

In this multi-processor implementation, when a ray hits a surface, the colour information which the incident ray carries is transferred to one of the secondary rays and the primary ray is discarded. The colour is only transferred to one of the secondary rays. If the object is both reflective and refractive, then the reflected ray is arbitrarily chosen to receive the incident ray's colour information. The secondary rays are then added to the ray queue and treated identically to primary rays. Using this method, the processor does not have to wait for the secondary rays to be traced.

Any colour which was carried in the primary ray is passed to the secondary rays. Any further colour which is determined by the secondary ray is added to the colour it carries, scaled by the amount of contribution this secondary ray makes to the final pixel intensity.

The serial algorithm uses a recursion cut-off to limit the number of secondary rays produced, the reasoning being that past some limit, creating further secondary rays will add little to the the final image. The same is done with the multi-processor algorithm. The *Level* field is incremented for each secondary ray. For example, if a ray with a level of 4 hits an object and spawns secondary rays, their level will be 5. Rays which exceed an arbitrary limit (eg. 8) are terminated and any colour intensity carried is returned to the host processor. The rays are also terminated if the $Contribution_{r,g,b}$ becomes too small.

If the object struck is neither reflective nor refractive, or if no object is struck at all (ie. it first strikes a bounding polygon), then the ray is terminated and it's colour intensity is returned to the host processor in much the same manner as rays which exceed the recursion limit.

Shadow Rays

A shadow ray is a special case of a ray. When an object is intersected, a shadow ray is created in the direction of each light source. If there are L light sources in the scene, then L shadow rays are generated from *each* intersection of a ray with an object. The colour information carried in a shadow ray is based on the intensity and colour of the light source it is directed towards. If the shadow ray reaches the light source without intersecting an object, then the colour information in the shadow ray is returned to the host processor. A shadow ray will terminate immediately if it hits

an object before reaching the light source. The shadow ray is then discarded, since the point of intersection is in shadow with respect to that light source.

The amount of processing needed to trace a shadow ray will generally be less than that needed for a primary or secondary ray since no other rays must be spawned and no lighting calculation is involved when the ray hits the light source.

Shadow rays are traced through the processors in the same way that primary and secondary rays are, but they are maintained in a separate shadow ray queue since their structure and processing is slightly different from that of the other rays.

Passing Rays Between Processor Nodes

A ray may not strike any of the objects in the node's subvolume. At that point, the ray is either leaving object space, or entering the subvolume of another node. If the ray is leaving object space, the ray is checked for intersection with the light sources and then the colour information is removed from the ray and returned to the host processor for placement in the frame buffer. However, if the ray is just crossing a subvolume boundary, the ray is passed, unmodified, to the node in charge of the next subvolume.

When a processor receives a message containing a ray, the ray is placed in a ray queue. The incoming ray is placed at the head of the ray queue. This is done for two reasons. The incoming ray has been partially traced on another node which means it will likely terminate sooner, or at least produce fewer secondary rays, than a primary ray. The second reason is that it gives incoming rays precedence over primary rays so that the incoming message queues (the shared memories) are less likely to become full. When a message queue becomes full, a node which tries to send to that queue will block, causing processing time on the sending node to be wasted. When the

nodes begin to block on sends, deadlock may occur. By attempting to finish existing rays before creating any new primary rays, the nodes avoid the deadlock problem. This approach appears to work well, since no deadlocking occurred durring all of the test runs.

3.3. Returning Pixel Values

Up to this point, when discussing the termination of a ray, I have done some hand-waving and said that the ray's colour intensity is returned to the host processor. This section will describe the method used to send pixel values back to the host.

Pixel values consist of two fields, a colour field ($C_{r,g,b}$) and a home pixel address field ($P_{x,y}$). These are extracted from the ray once the ray has been terminated. However, sending every pixel value individually causes a plethora of messages in the mesh. To avoid this, a number of pixel values are accumulated by each node before sending them to the host. These groups of pixel values are called a *pixel packet.* The number of pixel values in a pixel packet is limited by the maximum size of a message, 21 pixel values in this implementation.

When a pixel packet is received by a node, it is forwarded to the primary processor by the shortest possible route. The primary processor is the processor which has the connection to the host. The primary processor, upon producing or receiving a pixel packet sends it to the host. Rays whose colour is black (0,0,0) are not placed in the pixel packet since they do not contribute anything to the final pixel intensity.

3.4. Changing State

To determine when an image is finished, and the rendering of a new scene can begin, it is important to determine when there are no more rays in the processor array. The ray tracing of a scene is complete when all nodes report that they have no more rays in their queue, and no more initial rays to compute. As with any multi-processor approach, determining and changing the state of the processor array is a difficult problem. This is an area that has not received a great deal of attention with respect to this type of ray tracing.

If nodes send *idle* messages to the host when they become idle, the information in the message may become obsolete by the time it reaches the host. For example, after a node reports that it is idle, it may receive rays from another node, and become busy again. An idle message method requires a complex and repeated handshaking to ensure that all of the nodes are indeed idle. In [Dippe and Swensen 1984] this problem is avoided by allowing new frames to be started while old ones are not yet complete. However, this assumes that the available memory on each node is quite large since multiple versions of object space have to be stored. This extra memory requirement increases the cost of individual nodes and therefore limits the number of processors which can be applied to the problem. Further, if a large number of frames are being computed, the least loaded processors may become far enough ahead that there may be more than 2 frames being stored simultaneously.

Alternatively, a token value can be associated with each primary ray [Vatti 1984]. When the ray spawns secondary rays, the value in the token is divided between them. As ray values are returned, the host sums the token values into a corresponding *token* buffer. When all of the entries in the token buffer are equal to the initial ray value, then all of the rays have been traced to completion and the frame

is finished. However, each ray and pixel packet must have an extra token value field which increases the size of the messages. This also means that every ray must produce a pixel packet, regardless of whether it has anything to contribute to the final pixel intensity or not. This method increases both the size and number of messages which must be passed through the mesh.

Distributed Synchronization Method

This thesis uses a new method for determining the end of ray tracing. It is related to a tight time driven distributed synchronization scheme presented in [Peacock 1979] It was suggested to me by John Cleary. Peacock suggested this distributed synchronization scheme for use with simulations that run on multi-processors. Each node executes for some length of time independently of any other node, he calls this length of time a 'tick'. When a node has completed a tick, it must stop and wait until all of the nodes have completed the current tick before advancing to the next tick. This type of synchronization is needed for the ray tracing network which, before it can start or stop ray tracing, must be certain that all of the other

Field	
Current_state	The current state of the processor array.
Last_busy_node	The ID of the last busy node which the state packet encountered on the path around the processor array.

Figure 3-2. The state packet.

processors are ready to change state as well.

Peacock presented a two phase method for synchronization. However, the two phase method assumes that each node has a fixed amount of work, and will not become busy again once it has reported that it is idle. This is not the case with the ray tracing algorithm. A node can receive a ray from a neighbour at any time and thus oscillate between busy and idle. This problem can be resolved by noting that the speed of the multi-processor algorithm will be limited by the speed of the busiest node. By taking advantage of this fact, a state change algorithm which incurs very little overhead can be implemented.

The state change algorithm implemented in this thesis uses only one state packet. The fields of the state packet and their function are presented in Table 3-2. The state packet is sent around the nodes in a fixed loop, so that it touches each processor once on each pass. Each node maintains an ''activity'' flag which indicates if the node has had any activity. In this case, activity is defined as doing anything other than handling the state packet. When a node receives the state packet, the activity flag is examined. If there has been activity on the node, the Last_busy_node field in the state packet is updated to indicate the current node. The activity flag is reset to zero and the state packet is kept until the node becomes idle. When the node eventually becomes idle, the state packet is sent to the next node in the loop.

Should the node's activity flag be zero when the state packet is received, then the Last_busy_node field is compared with the node's ID. If the two fields do not match, then the state packet is passed to the next node, unmodified. If the Last_busy_node and the node's ID do match, then the state packet has been passed around the entire loop without encountering a busy node. This means that all of the nodes are idle, and a state change can be initiated.

```
if (my_ID = PRIMARY_PROCESSOR) start_state_packet();
wait_for_state_packet();
while (my_state ≠ END_OF_RAY_TRACING) do
        have_state_packet = TRUE;
        read_state_packet(State_packet);
        if (no_activity ≡ TRUE)
                if (State_packet.Last_busy_node ≡ my_ID)
                        { The state of the array can now be changed. }
                        State_packet.Current_state = State_packet.Current_state + 1;
                endif;
                if (my_state ≠ State_packet.Current_state)
                        { The state of the node should be changed. }
                        State_packet.Last_busy_node = my_ID;
                endif;
                my_state = State_packet.Current_state;
                send_state_packet();
                have_state_packet = FALSE;
        else
                { Activity has occurred, do not change state. }
                no_activity = TRUE;
                State_packet.Last_busy_node = my_ID;
        endif;
        flag = TRUE;
        while (flag = TRUE) do
          switch (my_state)
                { These functions only return when receiving the State_packet, }
                { or if this node has the State_packet and is now idle. }
                { They also set no_activity to FALSE if they do any work. }
                case CLIPPING : clip_objects();
                case RAYTRACE : trace_rays();
                case FLUSHPIX : flush_remaining_pixel_values();
                case END_OF_RAY_TRACING : wait_for_state_packet();
          endswitch;
          if (have_state_packet = TRUE)
                { Node has held the State_packet, and now it is idle. }
                State_packet.Last_busy_node = my_ID;
                send_state_packet();
                have_state_packet = FALSE;
                flag = TRUE; no_activity = TRUE;
          else
                { This node just received the State_packet. }
                flag = FALSE;
          endif;
        enddo;
enddo;
```

Figure 3-3. The state packet algorithm.

The state change is effected by incrementing the Current_state field. The state of the node which initiates the state change is modified, and then the state packet is once again sent around the loop.

When a node receives the state packet and the Current_state field does not match the state of the node, the node changes state and forwards the state packet. When the node which initiated the state change receives the state packet again, the process of holding the state packet until the node becomes idle is repeated. This process is presented in Figure 3-3.

It should be noted that a node can be idle and still have activity. For example, with this implementation, a node is considered idle when its ray queue is empty and it has no more pixels to trace. At that point it may still have activity due to rays being passed to it from neighbouring nodes. A node will only forward the state packet when it has traced a ray through each of its assigned pixels. From this it can be seen that when nodes begin to become idle, the state packet will eventually arrive at the busiest node. The busiest node in this case will be the one which is slowest at creating it's primary rays through the pixels. Due to the way primary rays are created on a node, as was described in Section 3.2, this node will have had the most rays passed to it from other nodes (or the slowest clock). The busiest node will retain the state packet until it becomes idle. The state packet will then be sent around the loop again. Since the state packet should not encounter any nodes which have not yet completed their primary rays, the state packet should only have to travel around the loop a few times per state. This is well supported by results from the actual implementation. Over all of the test runs, each node received the state packet a maximum of 4 times in each state.

This method has a disadvantage. When the number of nodes becomes large, the propagation of the state change through the processor array will be slow. This can be remedied by allowing a node to change to the next state if it receives a message that could only come if the sending node was already in that next state. Of course, if the node has had any activity since the last state packet, some sort of error has occurred with the state packet, or with the sending node, but this did not occur with this implementation. This method allows the new state to propagate through the mesh as quickly as possible, which may be faster than the state packet can propagate through the mesh.

3.5. Message Passing

The number of messages in the processor network is large. Therefore the communication channel must be reliable, and fast. The mesh in this thesis uses shared memories to implement an asynchronous communication channel. Each shared memory is logically broken into 2 sections, half for incoming messages, and half for outgoing messages. There are two outgoing message calls, *open* and *send*. The *open* call allocates a variable sized buffer in one of the outgoing shared memories. Information is placed in the buffer and *send* is called to signal that a message is waiting for the processor at the other end of the shared memory. This signal interrupts the receiving processor only long enough to set a bit to register the send, unless the receiving processor is waiting for a message. If there is not enough space in the shared memory when *open* is called, then the process may request to either block until enough space becomes available, or to fail immediately.

The two incoming message calls, *get* and *release*, act in a similar manner. The *get* call gets the next available message out of the shared memory in the requested

direction, information is taken out of it and *release* is called to remove that message from the shared memory. When *release* is called, if there is a process blocked on an *open*, it is signaled to continue if there is now enough space. As with *open*, *get* can be requested to block until a message arrives, or requested to fail if there are no messages. Further, *get* can be called without specifying a shared memory, whereupon all 4 of the incoming shared memories are examined and the first message found is returned.

This method of using the shared memory for communication has several advantages. The communication channel is dedicated to the two nodes which it links. A broadcast communication channel would suffer from contention problems as the number of nodes increased. Shared memory is also more reliable as communication channels, since *noise* levels during transmission of the message are greatly reduced. Another feature is that the communication can be asynchronous, assuming the shared memories do not become full. To send a message to a node, the message is simply placed in the appropriate shared memory, and unless the receiving node is explicitly waiting for a message, it only executes a small interrupt routine to note that there is a message waiting in that shared memory. The main drawback is the limited global communication. Sending a message to a node which is not a neighbour becomes more difficult.

Message Routing

The shared memory allows for fast, reliable communication between neighbouring nodes. There is, however, only one communication line to the host. A node which is distant from the primary processor must route pixel packets through other nodes to send them to the host. To do this a fast and simple routing algorithm

was developed by Murray Peterson and myself.

The routing algorithm sets up a shortest path table at the outset of the ray tracing process. The table contains one entry for each node of the mesh. Each entry contains the address of one of the four shared memories. When a message must be sent to a specific processor, the corresponding entry in the table is consulted, and the message is sent to the indicated shared memory. When a node receives a message that is to be sent to another node, it consults its unique shortest route table.

Since the processor network used in this thesis is a two dimensional mesh, the shortest route to a destination node, $D_{row,column}$, from the sending node, $S_{row,column}$, is calculated by comparing the row and column indexes of S and D. The row indexes, S_{row} and D_{row}, are compared, if they differ the algorithm decides which row should be sent to. This decision is made more difficult because the mesh is a torus. This means that if the mesh is $n \times n$, nodes in row 0 and row n are connected to one another. Because of this, the shortest route to a node with a smaller row index may be to send it to a row with a higher row index. The method to determine the shared memory to send to for differing rows is presented in Figure 3-4 and it is trivially extendible to columns.

Since this procedure is followed on every node, when a message is received which is not for the receiving node, the route table is consulted to find the shortest route to get the message to the intended node. A further enhancement was made so that if there is more than one shortest route to a certain destination node, the routing algorithm alternates sending messages between the different shortest routes. Based on the test results, the message loads were much more balanced using this alternating message routing method.

```
if (S_row = D_row and S_column = D_column)
        route_to [D_row,column] = ME;
endif;
if (S_row ≠ D_row)
        diffA = D_row − S_row;
        if (diffA < 0)diffA = diffA + n;
        diffB = S_row − D_row;
        if (diffB < 0) diffB = diffB + n;
        if (diffB < diffA) route_to [D_row,column] = Down_a_Row;
        else route_to [D_row,column] = Up_a_Row;
endif;
```

Figure 3-4. Calculating the shortest route to a processor.

3.6. The Multi-processor Array

The algorithm presented in this thesis was designed to run on a toroidal, 4-connected processor mesh. Inter-processor communication is effected via 4KB blocks of shared memory between the processor and its neighbours.

One node in the processor mesh is connected to a host machine. This machine is known as the *primary processor*. The only distinction between this node and the others is this connection to the host. Figure 3-5 shows a conceptual diagram of a 3×3 toroidal mesh. The primary processor is pictured in the bottom left corner.

Processor Array Hardware

The nodes of the mesh consist of M68000's with 512KB DRAM, 2 on-board 4KB dual-ported memories for communication with neighbours, an independent clock, and interrupt lines from each of the 4 neighbours. One of the nodes has a 1Mbit/second connection for communication with the host processor (VAX 11/780

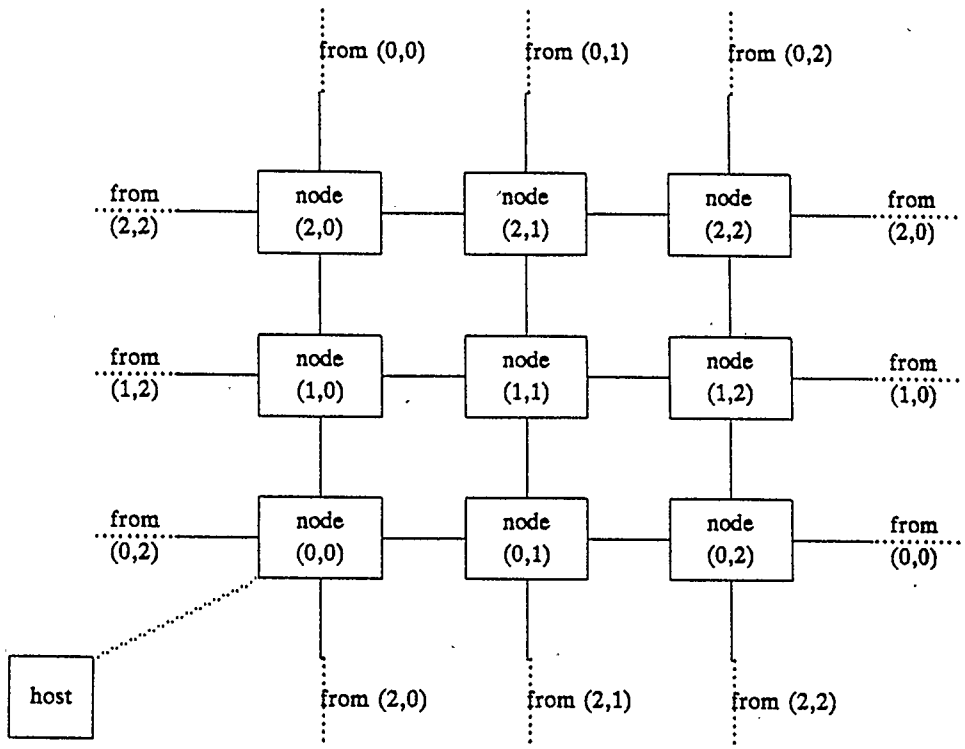


Figure 3-5. A 3 by 3 processor mesh.

running UNIX 4.2bsd).

The shared memory provides a high speed, highly reliable, asynchronous communication channel between neighbouring processors. Unfortunately the hardware was not complete at the time of this writing so a network of workstations was used to simulate the hardware. Shared memory was not available using the workstation network, so the asynchronous communication was simulated using a

synchronous protocol, JIPC, part of the JADE project, developed at the University of Calgary [Unger, Birtwistle, Cleary, Hill, Lomow, Neal, Peterson, Witten, Wyvill 1984].

Since JIPC is a synchronous protocol, and the algorithm calls for asynchronous communication, the simulation requires that each process have an associated *memory* process. The memory process only receives *get* and *release* requests from its parent processor, *open* and *send* requests from the four neighbouring processors. Since the memory process is dedicated to its task, and independent of the actions of the ray tracing process, the communication between any two neighbouring ray tracers becomes asynchronous.

The network used for simulating the mesh consists of 9 Corvus Concept workstations, each having an M68000 processor, 512KB DRAM and a 1 Mbit/second net connection. Since dedicated communication channels were not available, broadcast collisions were a major source of difficulty during testing runs. End to end acknowledgement at the software level had to be implemented before the simulation could be run to completion without losing message packets. The end to end acknowledgement caused the simulation to run much slower than the actual mesh hardware would. This extra overhead is examined more thoroughly in Chapter 5.

Accuracy of the Simulation

The workstation's multi-tasking kernel uses a little under half of the available memory leaving about 300K free for the ray tracing process. The required memory simulation process uses about 40K, leaving close to 260K free for the ray tracer. The amount of memory is slightly less than will be available on the mesh hardware. The mesh's multi-tasking kernel is obstensibly the same as workstation's, although

slightly smaller, because the nodes on the mesh do not need screen and keyboard code. On the mesh itself, the memory simulation process will not be present which further increases the available memory. So the simulation will use more memory and will have a slower speed of communication.

The ray tracer itself is approximately 90K in size, so 170K is left for the objects and voxels. The ray tracer knows of only three types of objects. These types and their sizes are summarized in Table 3-3. The listed size of a polygon assumes that a polygon will have only 4 vertices, the number of bytes for a polygons is $88 + (13 \times V)$ where V is the number of vertices. The column labeled "Approximate Maximum Number" refers to the number of that object type which will fit into 170K if all of object space is contained in one voxel and no other objects are used. In practice the maximum number of objects will be much less because of the space needed for the voxels and the object references.

Objects Sizes		
Object	Number of Bytes used for storage	Approximate Maximum Number of objects
polygon	140	1200
sphere	60	2800
light source	28	6050

Table 3-3. Size of objects used in the ray tracer.

Summary

This chapter has presented a multi-processor algorithm which has been coupled with the spatial subdivision method presented in Chapter 2. The results from simulating this algorithm are presented in the next two chapters.

CHAPTER 4

Results from Uniprocessor Spatial Subdivision

The spatial subdivision algorithm presented in Chapter 2 was implemented on both a multi-processor and a uni-processor system. This chapter will present the results obtained from using the spatial subdivision algorithm on the uni-processor, although the results are equally applicable to the performance of the individual nodes in the multi-processor array. Chapter 5 will present results from the multi-processor algorithm along with discussion of how the results relate to spatial subdivision.

The purpose of a spatial subdivision algorithm is to reduce the number of intersection tests performed and thereby speed the process of ray tracing. This chapter examines the effect a regular subdivision has on reducing this number.

A regular subdivision requires a large amount of space to represent the voxels in memory. The use of a hash table was introduced in Chapter 2 to reduce this requirement. The quality of the hashing function used, the amount of memory saved and time used are also examined.

The Test Scenes

To test the effectiveness of the spatial subdivision algorithm, three scenes were ray traced: 40 randomly placed, reflective spheres; a barroom corner with a table and two stools made entirely of polygons; and a human face described by a polygonal mesh. The scenes were chosen because they are fairly typical of the scenes being rendered at the University of Calgary. The scenes which could be rendered were limited in size by the available memory on the individual nodes of the processor

array. Photographs of these scenes are presented in Chapter 5.

The three scenes have different features which are worth pointing out before discussion of the results. The sphere scene has very few objects in it, which means that the time needed to ray trace it without spatial subdivision is relatively small. However, due to the large number of reflections, rays are well distributed throughout scene space.

The barroom scene has three conical ''lamp shades'' which are formed by 78 polygons, all of which meet at a point. The voxel which encompasses this point will contain at least 78 objects regardless of the level of voxel subdivision.

The ''face'' scene does not contain such irregularities but the polygonal mesh which defines the face is more complex towards the centre of the scene, due to the eyes, nose and mouth areas. The total number of objects in each scene is detailed in Table 4-1.

Scene	Number of Objects
Spheres	40 spheres
Barroom	367 polygons
Face	954 polygons

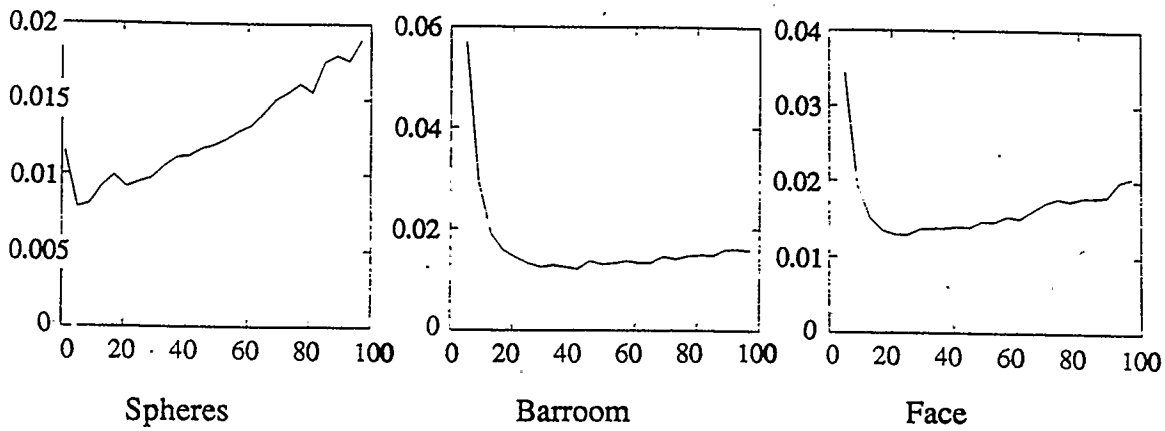
Table 4-1. Number of objects in each scene.

4.1. Speedup Provided from Spatial Subdivision

The main advantage provided by a spatial subdivision algorithm is that the number of objects tested for intersection is reduced. Provided that the cost of moving between voxels is not great, the reduced number of intersection tests should increase the speed with which each single ray can be traced, and thus increase the total overall speed of ray tracing an image. This section will examine the speedup gained as the number of *voxel subdivisions* increases. The term *voxel subdivisions* is used to refer to the cube root of the total number of voxels, or more intuitively, the number of voxel boundaries in each of the 3 major axis directions.

The X axis of every graph presented in this section is the number of voxel subdivisions. They are increased in steps of 4, starting with a $1 \times 1 \times 1$ voxel array, and ending with a voxel array of $97 \times 97 \times 97$ in 25 steps. The hash table graphs are the only exception to this, the number of voxel subdivisions for the hash table graphs were increased in steps of 2, from 17 to 97 since any number smaller than 17 took so little space, it seemed inefficient to use a hash table.

The speedup from using spatial subdivision on the three scenes is presented in graph form in Figure 4-1 (the ratio speedups obtained over the naive algorithm are presented later in Table 4-2). In all cases, the time to ray trace the scenes is initially reduced as the number of voxel subdivisions is increased. Eventually, increasing the number of voxel subdivisions increases the trace time. This turn around occurs when the number of ray/object intersection tests is not reduced significantly by increasing the number of voxel subdivisions. Increasing the number of voxels past this point increases the amount of computation needed for each ray by adding more next voxel calculations without reducing the other times significantly. The number of intersection tests will still decrease slightly as more voxel subdivisions are added, but



vertical axis - Time in seconds (VAX 11/780).
horizontal axis - Number of voxel subdivisions.

Figure 4-1. Average time per ray with increasing voxel subdivisions.

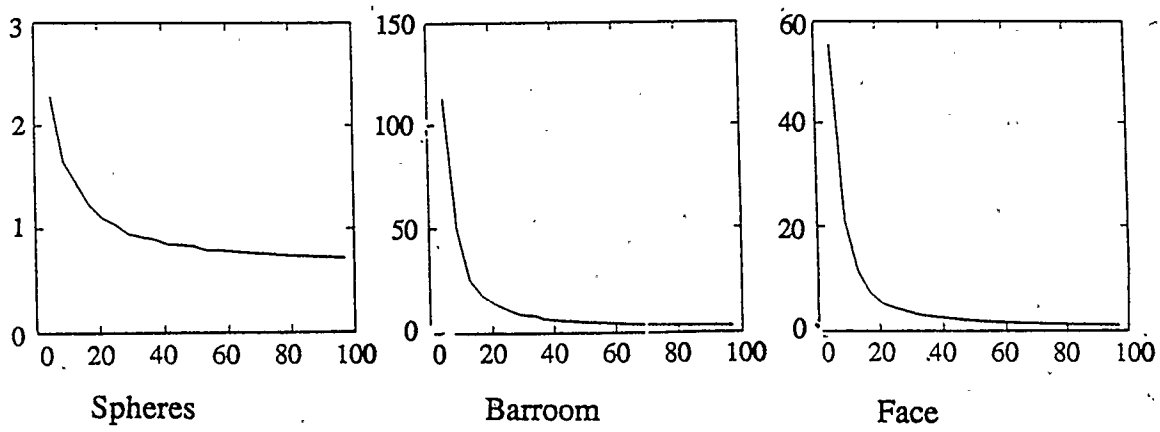
the time to perform the additional next voxel calculations will outweigh any benefits from the reduced number of intersection tests.

The sphere scene suffers most noticeably from this added cost. The trace time for the spheres eventually climbs so high that the time needed to trace the spheres at a voxel subdivision of 97 is greater than the time needed with only 1 voxel subdivision (which is equivalent to the naive algorithm). This is a result of the small number of objects in the scene. The initial speedup due to spatial subdivision is nearly 200% for the spheres, but the turn around point is reached almost immediately as N increases above 5.

The Reduction in the Number of Intersection Tests

The spatial subdivision technique was introduced in order to reduce the number of intersection tests performed and thus speed up the tracing of each individual ray. As can be seen in Figure 4-2, the number of intersection tests per ray decreases quickly as the number of voxel subdivisions is increased. As expected, the number of ray/object intersections are not greatly reduced beyond the level of voxel subdivision where the trace times start to increase again in Figure 4-1. Since the number of intersection tests are not being greatly reduced, the time spent travelling through the additional voxels becomes the dominant cost.

The level of voxel subdivision which produced the fastest times are presented in Table 4-2. The table also shows the average number of intersection tests per ray at



vertical axis - Average number of intersections tests per ray.
horizontal axis - Number of voxel subdivisions.

Figure 4-2. The reduction in the number of intersection tests.

Scene	Approximate optimal number of voxel subdivisions	Average number of intersection tests per ray	Percentage speedup over single voxel
Spheres	5	2.29	198%
Barroom	41	6.00	2223%
Face	25	4.39	3267%

Table 4-2. Approximate optimal number of voxel subdivisions.

this level of subdivision. At the point where the trace time begins to increase the number of intersection tests is relatively small. This point will be called *the optimal number of voxel subdivisions* since that number of voxels produces the fastest timings for that screen resolution.

Speed of the Next Voxel Calculation

When using spatial subdivision, the time cost of moving between voxels is added to the ray tracing process. The time cost of this operation should be small enough to allow a large number of subdivisions before significant time penalties are incurred. If the time cost is too large, then a ray which does not intersect an object will take longer to process than a ray which does simply because it must traverse more voxels.

The spatial subdivision implementation in this thesis was presented in Chapter 2. The average speed of the next voxel operation is presented in Table 4-3. For interest, the times for the floating point *3DDDA* given in [Fujimoto, Tanaka and Iwata 1986] are also presented. These results are not directly comparable since

Voxel Representation	Next Voxel Operation (per voxel)	Next Voxel Operation Initialization (per ray)
Array	0.059	0.684
Hash Table	0.090	0.807
Octree (3DDDA VAX 11/750)	0.072	0.591

timings taken on a VAX 11/780, times are in milliseconds

Table 4-3. Average speed of the next voxel operation.

different machines types were used and it is not known if floating point hardware was used in the *3DDDA* implementation. The timings taken on this implementation did not use floating point hardware. Only the *3DDDA* floating point timings are presented here since the next voxel operation implemented uses floating point and not integer arithmetic. The times in this table were averaged over all the test runs, with a maximum deviation of 0.02 milliseconds, which is near to the difference between these an Fujimoto's results. The times may be slightly slow due to the presence of the timing mechanism in the next voxel code, which will reduce the chance of a cache loop.

In determining which method is faster, several factors must be considered. The speed of moving from one voxel to another is slightly better for the uniform array subdivision, and the actual memory address is found with one integer addition to add the array offset to the array base, but a higher startup cost is incurred. The cost of finding the actual memory address of the next voxel using the octree approach is much higher since anywhere from 2 to $log(N)$ pointers must be chased (assuming a

balanced tree), where N is the number of voxels. This address calculation cost is not included in the times for the octree, whereas it is for the uniform method. On the other hand, the octree method will likely have to perform fewer next voxel operations. To compare the two methods would require an equitable implementation on identical hardware.

The times needed for an intersection test and a full intersection calculation are given in Table 4-4. These timings were not carried out on the multi-processor nodes, but similar ratios are expected.

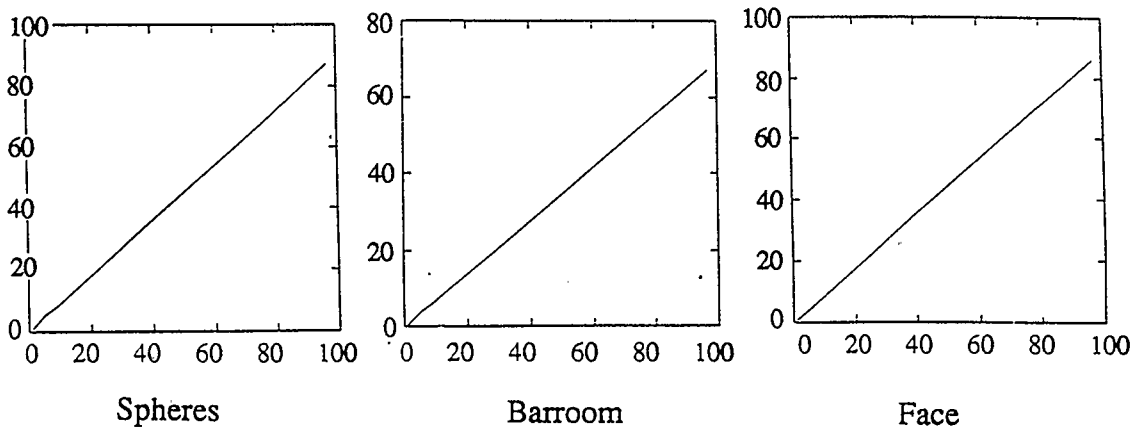
The Average Number of Voxels Encountered per Ray

The length of each ray remains constant as the number of voxel subdivisions increases. This means we should expect the average number of voxels which a ray is traced through to increase linearly with the number of subdivisions. This in fact did occur. The graphs in Figure 4-3 depict the average number of voxels encountered per ray.

Object	Average Time Intersection Test	Average Time Intersection Operation
Sphere	0.218	1.301
Polygon	0.320	1.362

timings taken on a VAX 11/780, times are in milliseconds

Table 4-4. Speed of the intersection operations.



vertical axis - Number of voxels encountered.
horizontal axis - Number of voxel subdivisions.

Figure 4-3. The average number of voxels encountered per ray.

4.2. The Quality of the Hashing Function

The hashing function is a simple "modulus" based function. The use of this function ideally would produce a random distribution of the number of full voxels hashed to a given location in the hash table. To test the quality of the hashing function, the predicted and actual results where compared.

If the hashing function is random, the mean number of full voxels hashed to a given location is given by:

$$\rho = \frac{F}{H} \tag{4.1}$$

where H is the number of hash table locations, and F is the total number of full voxels.

Of interest is the time spent searching for the correct voxel once a hash table location has been identified. If this search time is too great, then the hash table representation will suffer greatly in terms of speed when compared to the array representation. With the array representation, the search time is 1 since there is always only one voxel at any given location. The search time for a hash table representation should be close to 1 as well.

If a bit table is used to screen out searches for empty voxels, then the expected or average search time ($E(time)$) for a voxel in the hash table is given by:

$$E(time) = \frac{1}{2F}\sum_{i=1}^{H} n_i^2 + \frac{1}{2} \tag{4.2}$$

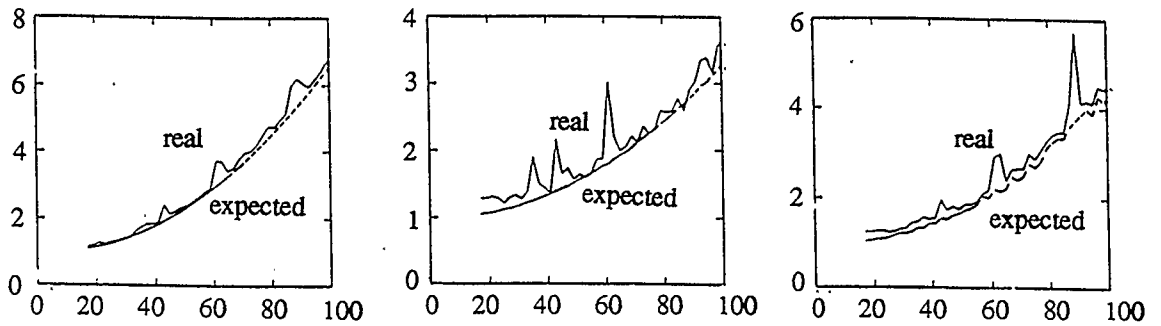
where n_i is the number of full voxels hashed to the ith location, F is the total number of full voxels and H is the number of hash table locations.

Of interest is the result of this function compared to the expected search time for a completely random distribution which is given by:

$$E(time) = 1 + \frac{1}{2}\rho \tag{4.3}$$

Figure 4-4 plots these two functions. The two functions match well, with the exception of some odd spikes in the curves. These spikes are due to our hashing function which is not completely random. At certain levels of voxel subdivision, the hashing function ends up assigning a more than average number of full voxels to a single hash table location. To guard against this, a useful preprocessing step may be to assign objects to voxels and test the distribution in the hash table. The relative merits and trade-offs to this method are presented later in this chapter.

Table 4-5 shows the expected (from equation 4.3) and real (from equation 4.2) average search time at the optimal level of subdivision. The real average search times are slightly larger than the expected times, but they are still very close, which means



Random Spheres Barroom Face

Figure 4-4. Expected vs. real search times for full voxels.

Expected Search Times for a Full Voxel		
Scene	Expected average search time (4.3)	Real average search time (4.2)
Spheres (17^3)	1.11	1.18
Barroom	1.34	1.35
Face	1.13	1.24

average number of full voxels checked before the correct voxel is found
Table 4-5. Expected vs. real average search times.

that our hashing function is not unacceptably worse than truly random hashing at these levels of subdivision. It is encouraging that the search times are very near to 1, since this means that the hash table representation will not spend too long searching for the correct full voxel in the hash table. As mentioned earlier, the search time for

the array representation will be exactly 1. Because of the small search times, the hash table representation will deliver only slightly slower times than the array representation.

The Cost of the Bounding Polygon Search in the Hash Table

As mentioned earlier, the bit table screens out searches for empty voxels. However, the voxels containing the bounding polygons are indicated as full in the bit table yet they do not have a corresponding entry in the hash table. A ray can encounter such a case exactly once, since upon hitting a bounding polygon it leaves scene space and all processing on the ray is terminated. In complex scenes, a large number of rays will terminate by intersecting an object rather than encountering a bounding polygon. So $\rho \times \phi$, (where ϕ is the time needed to check if the voxel in the hash table is the one we are searching for), is an upper bound on the time added for unsuccessful searches per ray. In the test scenes, ρ never exceeded 10.9 and it never exceeded 0.7 at the optimal level of subdivision.

So long as space is a more important consideration than speed, the hash table representation will provide reasonable results, since the hashing function is well distributed and it is not much slower than the array representation. Table 4-6 shows the speed of the hash table representation versus the array representation at the optimal level of subdivision. Only the face and the barroom scene are shown since the sphere scene required only 125 voxels (5×5×5) and using a hash table is inefficient for such a small number.

Scene	Hash table representation	Array representation
Barroom	844	744
Face	317	301

timings taken on a VAX 11/780, times are in seconds
both scenes traced at optimal level of subdivision
Table 4-6. Hash table vs. array representation speed.

Overhead in Assigning Objects to Voxels

In practice voxel assignment is substantially faster than the ray tracing process. For the three test scenes used, the maximum percentage of time spent assigning objects to voxels was 13.6%, but at the optimum level of subdivision, the maximum was only 5.6%.

It should be noted, however, that the number of pixels was relatively small in the scenes from which the voxel assignment statistics were taken (150×150). If high quality images are to be rendered, the time taken to ray trace will increase, but the time taken to assign objects to the voxels will remain constant since it is a function of the voxels and the objects, not the number of rays. This means that the percentages in the previous paragraph are high. As a test, the barroom scene was rendered at high quality (512×512 pixels, or more than 3/4 million rays) and the percentage time taken to assign the objects to the voxels was 0.01%. The time taken to assign the objects to the voxels was constant. From these figures, assigning objects in a pre-pass test for quality of the hash table distribution is feasible without a large speed degradation.

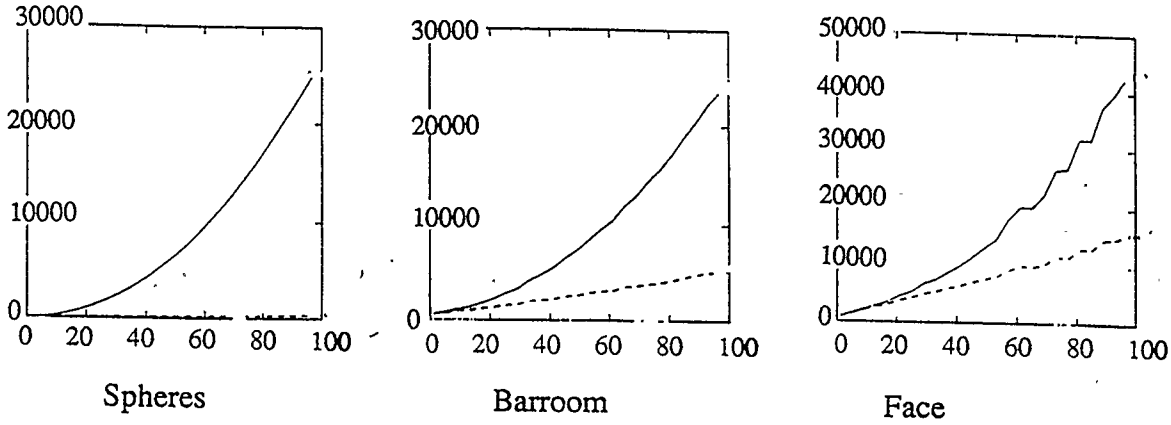
4.3. Storage Requirements for the Scenes

In Chapter 2, the amount of storage which uniform spatial subdivision required was discussed. To reduce this requirement, two techniques were presented. With the array representation of the voxels, it was noted that voxels with a single object reference could share that reference with all of the other voxels which also had only that one object reference. A hash representation was also presented to reduce the amount of space needed to store the voxels themselves. This section will examine the space that can be saved with both methods, and then compare the two methods.

Shared Object References

The simple "shared" object reference method produced substantial space savings over the straight forward method of using a separate object reference for each voxel the object is assigned to. These results can be seen in Figure 4-5. In these charts the upper curve is the straight forward method of assignment, while the lower curve is the new "shared" reference method presented in Section 2.5. It can be seen that the savings in the number of references are substantially better than the straight forward reference assignment as the number of voxel subdivisions is increased.

Each object reference consumes 8 bytes, so for every 128 references saved, 1*K* bytes are saved. At the optimal level of voxel subdivision this amounts to 39*K* at best (for the face scene) or about 5% of the total space required by both program and data. The other two scenes saved less than 4%, so while the graphs are impressive, the actual savings provided using the "shared" reference method are not very significant.



vertical axis - Number of object references in voxels.
horizontal axis - Number of voxel subdivisions.

Figure 4-5. Number of object references in voxels.

Space Savings with the Hash Table versus the Voxel Array

As was mentioned in Chapter 2, the hash table method was introduced in order to reduce the amount of space needed for storing the voxels. This subsection examines the space used in the hash table representation versus the space used when the "shared" object reference method is used with the array representation.

The number of bytes (B) needed for the array representation is given by:

$$B = [(n^3) \times 4] + [(a\_refs - shared\_refs) \times 8] \tag{4.4}$$

where:

n^3 is the number of voxels.

a_refs is the number of object references in voxels (there may be more than one reference per object).

shared_refs is the number of times a reference is not allocated because of sharing between voxels.

The 4 and the 8 represent the number of bytes required for each voxel, and each object reference, respectively.

The number of bytes needed for the hash table representation is given by:

$$B = (M \times 4) + (a\_refs \times 12) \tag{4.5}$$

where:

M is the length of the hash table.

a_refs is the number of object references in voxels.

Note that the byte size of an object reference (12) in the hash table representation is 4 more than for the array representation since the ID of the actual voxel must be stored to distinguish between voxels which may have been hashed to the same hash table location.

From (4.4) and (4.5) we can write:

$$\beta = 4[\,(n^3 - M) + ((a\_refs - shared\_refs) \times 2) - (a\_refs \times 3)] \tag{4.6}$$

where:

β is the number of bytes difference between the two methods.

If β is positive, then using the hash table representation is more economical, if it is negative, the "shared" object reference array representation should be used. Since these values can be determined without ray tracing, deciding on the best representation to use can be done by assigning objects to voxels using both methods, and comparing. As mentioned earlier, this incurs less cost than ray tracing, but the hash table representation will be more economical in most cases.

Applying this formula to the three scenes at the optimal level of voxel subdivision, we get the figures in Table 4-7. The hash table size was held constant at

4096 hash table locations.

The table shows that except in the case of the spheres, using the hash table method is more economical in terms of storage. The sphere scene is exhibits a poor β because the hash table was larger than needed to represent such a small number of voxels.

4.4. Summary

The spatial subdivision algorithm provides significant speedups over the naive ray tracing algorithm by reducing the number of tests for intersection significantly. The optimal level of subdivision is a function of the scene and the number of rays. No method for choosing the optimal number of voxel subdivisions before ray tracing the scene exists, but the speedups are large enough that a reasonable level of subdivision should provide good results. Based on the results a uniform spatial subdivision can provide a speedup of 1 to 3 orders of magnitude for reasonably complex scenes.

A trade-off between speed and storage can be made if a hash table representation is used for storing the voxels. While the hashing function used in this

Scene	β	v^3	hash *a_refs*	array *a_refs*	% space saved over array
Spheres	-4123	125	70	29	N/A
Barroom	52676	68921	5591	2312	55%
Face	4001	15625	5400	4336	9%

Table 4-7. Difference in storage space - Equation (4.7).

implementation is not random, the expected search time for a full voxel is well behaved and the time cost incurred by the search does not appear prohibitive. The hash table access and performance is only slightly slower than the array representation.

CHAPTER 5

Results from the Multi-processor Algorithm

The multi-processor algorithm was described in Chapter 3. This chapter will examine the results from tracing the 3 test scenes using this algorithm on a varying number of nodes.

When using a multi-processor algorithm, the primary area of interest is the speedup attained, which is ideally close to or greater than N for an N processor array. A primary reason for not achieving this result is an unbalanced workload between the processors. A new load balancing scheme is presented with some preliminary results.

This chapter will examine the speedups obtained from several runs and try to identify why the speedup was or was not close to N.

The Simulation Procedure

Before discussion of the results a brief description of the simulation procedure is in order. The multi-processor algorithm was simulated on both a single processor using multi-tasking to simulate multi-processors, and a small network of workstations. The uni-processor had a process limit which restricted the number of simulated processes which could be run. The workstations were limited in physical availability. Due to this, the multi-processor simulations were limited to a maximum dimension of 3×3. Further description of the hardware and run-time environment is provided in Chapter 3.

Each scene was run with a varying number of processors, or processes on the uni-processor simulation. The processor arrays are constrained by the current

implementation to be square, so the number of processors on the various runs were 1, 4 and 9.

5.1. The Speedup as the Number of Processors is Increased

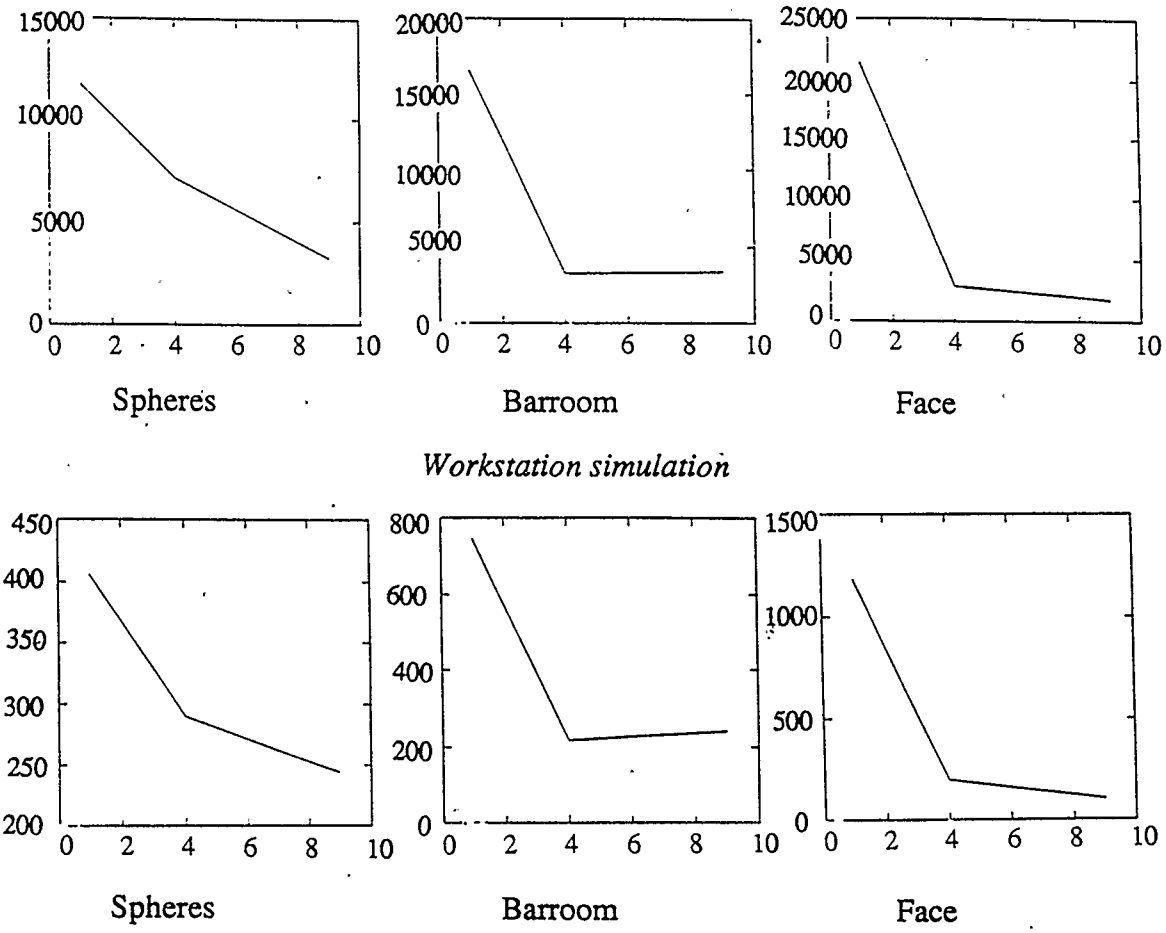
A multi-processor ray tracing approach takes advantage of the lack of ray to ray coherence in most ray tracing algorithms. Since each ray can be traced independently of the others, each node of the processor array may process rays in parallel. This seems to indicate that if N processors are applied to the task, a speedup of N should be attained. However, even assuming that each node processes an equal number of rays, the amount of time needed to process a ray is not constant. The factors which contribute to this are:

- the length of the ray
- the number of voxels passed through in the node's subvolume
- the number of objects tested for intersection
- the complexity of the surface struck
- the surface characteristics of the surface struck

In addition there are ''per ray'' variables introduced by a spatial subdivision approach to multi-processor ray tracing:

- the number of processor subvolumes which the ray passes through
- the time needed to pass a ray between processors

These ''per ray'' variations mean that even if each node has an equal number of rays to trace, the time needed to trace these rays will not be equal. This will tend to reduce the speedup possible. To further complicate the issue, it is likely that each



based on the busiest processor

vertical axis - Time in seconds.
horizontal axis - Number of processors.

Figure 5-1. The speedup gained via the multi-processor algorithm.

processor will not have an equal number of rays to trace. This is caused by:

- a difference in the number of primary (pixel) rays assigned
- a difference in the number of secondary rays spawned
- a difference in the number of rays received from other nodes

Some combination of these factors will conspire to create an uneven workload across the nodes. The graphs in Figure 5-1 show the timings obtained from ray tracing the three test scenes on 1, 4 and 9 processors. The speedups gained are presented as ratios in Table 5-1. These ratios are based on the time taken to ray trace the scenes on a single processor.

The ''face'' scene was too large to be ray traced on a single (1) workstation and so the ratios have been estimated based on the time taken to ray trace it on the uni-processor. However, all other timings for the ''face'' are real.

The ratios in the table show that the mainly diffuse scenes (the barroom and the face) achieved very good speedups, while the sphere scene did not. Table 5-2 shows

Scene	2×2 processors		3×3 processors	
	Wrkstn.	Uni	Wrkstn.	Uni
Spheres	1.64	1.40	3.61	1.66
Barroom	5.11	3.43	4.93	3.09
Face *(workstation ratios estimated, see text)	*6.04	6.05	*9.88	11.50

ratios based on the busiest processor compared to single processor

Table 5-1. Speedup gained using the multi-processor algorithm.

the difference between the busiest and least busy nodes. The processing times for the sphere scene vary wildly between nodes indicating that the cause of the poor speedup is an uneven workload.

The test runs with the largest discrepancy between the busiest and least busy node times are the test runs which exhibit the least amount of speedup. When the difference between the busiest and the least busy node is slight, the speedups are much better. By balancing the load, better performance can be gained. A new algorithm for load balancing, along with some results, is presented at the end of this chapter.

Scene	2 × 2 processor array			3 × 3 processor array		
	% of rays	Total time wrkstn.	Total time uni	% of rays	Total time wrkstn.	Total time uni
Spheres						
Busiest	27.2%	7139	289	14.9%	3253	243
Least busy	22.8%	2994	222	8.2%	954	101
Barroom						
Busiest	25.0%	3244	217	14.0%	3364	241
Least busy	25.0%	2095	120	8.4%	486	33
Face						
Busiest	25.3%	2855	196	10.5%	1747	103
Least busy	24.9%	2134	150	10.5%	755	65

Table 5-2. Busiest and least busy nodes.

Concerning the Busiest Node

The amount of work done by a node will be determined by the number of rays which enter it's subvolume as well as the number of objects within the subvolume, but the number of objects should not affect the amount of work as much as the number of rays. This is especially true since spatial subdivision is being used on each node. With the appropriate level of voxel subdivision, the number of intersection tests performed on a ray is greatly reduced, as discussed in Chapter 4.

In general, then, it is expected that the load on each node will strongly depend on the number of rays it has to trace, assuming that the time spent forwarding pixel packets is small. Table 5-3 pairs the busiest node with the number of rays it traced, and compares that to the largest number of rays traced by any node in the array. In 4 of the 12 cases, the node with the greatest number of rays was also the busiest processor. In all but 2 of the cases, the busiest node had more than the average number of rays. The discrepancy between the node with the greatest number of rays and the busiest is due to a poor level of voxel subdivision for the number of objects on that node.

The Number of Objects per Node

The optimal number of subdivisions is dependent upon the number of objects, the object's placement, the speed of the next voxel calculation, the speed of other operations such as intersections, the number of rays and the distribution density of the rays. Since the actual optimal number is a complex function of these factors, it is very difficult to calculate. However, to provide the fastest trace time, the number of voxels will increase as the number of rays and the number of objects do. Using this assumption, if the busiest processor does not trace the most rays, then it will likely have a poor level of subdivision for the number of objects which it contains.

Number of nodes	Percent of rays on busiest node	Largest percent of rays traced on a node
4 (uni)		
spheres	26.4	27.3
barroom	25.0	25.0
face	25.3	25.3
9 (uni)		
spheres	14.9	14.9
barroom	14.4	15.3
face	10.5	12.2
4 (multi)		
spheres	27.2	27.2
barroom	25.0	25.0
face	25.0	25.3
9 (multi)		
spheres	14.9	14.9
barroom	12.6	13.4
face	10.5	12.2

percent refers to the percentage of the total number of rays

Table 5-3. Busiest node vs. the largest percentage of rays.

Table 5-4 shows the number of objects contained on the busiest nodes. Only 4 of the 12 busiest processors did not correspond to either having the most objects or the most rays. In all 4 cases the nodes had more than the average number of rays and more than the average number of objects.

Discussion of the Results

The barroom scene is slower with 9 processors than it is with 4. This is due to the fact that few rays were passed when 4 nodes were used. Using 9 nodes, the central

Number of nodes	Number of objects	Most objects?	Most % of rays?	Nodes with more objects
4 (uni)				
spheres	7	no	no	1
barroom	73	no	no	2
face	352	yes	yes	0
9 (uni)				
spheres	7	no	yes	1
barroom	70	no	no	2
face	356	yes	no	0
4 (multi)				
spheres	7	no	yes	1
barroom	74	no	yes	2
face	352	yes	yes	0
9 (multi)				
spheres	7	no	yes	1
barroom	70	no	no	2
face	356	yes	no	0

Table 5-4. Number of objects and rays on the busiest node.

processor passed numerous rays to the outside processors and the number of rays which outside nodes had to trace increased. Since these processors had more rays to trace, an uneven load resulted.

The timings for the barroom and the face are quite surprising. The speedups obtained are greater than N in some cases. However the timings should only be taken as a strong indication of the speedups to be gained. The timing methodology contains many inaccuracies as discussed in Section 5.2.

The Level of Voxel Subdivision on Each Node

The multi-processor algorithm uses spatial subdivision on each of the nodes to further increase the performance of the processor array. The number of voxel subdivisions through out the scene was held relatively constant regardless of the number of nodes. With one processor, a voxel subdivision level of 32 was used, with 4 processors a level of 16 was used on each processor and with 9 processors a level of 8 was used. This should serve to limit the effects of voxel subdivision on the multi-processor results so that they may be examined separately. In practice it will be desirable to subdivide the subvolume on each node by a different amount, based on the number of objects which intersect the subvolume.

If each processor has an optimal number of voxel subdivisions for its subvolume, then the ray tracing process will be at its fastest. The results in Chapter 4 show only a weak time dependency on the number of voxel subdivisions beyond some initial level of subdivision. However, it is still desirable to keep the speed as fast as possible. Even small differences in the average amount of time to trace a ray can become significant when the number of rays is in the order of millions.

Object space is not evenly divided between the nodes when 9 processors are used. As was mentioned in Chapter 3, each node is assigned a nearly equal number of pixels. However, object space will generally extend beyond the boundaries defined by the pixels to be traced (see Figure 5-2). This means the outside nodes will be assigned more of object space since the pixels are generally focused on a subsection of the scene. Scenes which are more centrally complex will exhibit a greater speedup. This is because the central nodes have less space assigned, but have the same number of voxel subdivisions as every other node. Therefore, the volume of the voxels on the central processors will be smaller than the voxel volumes on the outside processors of

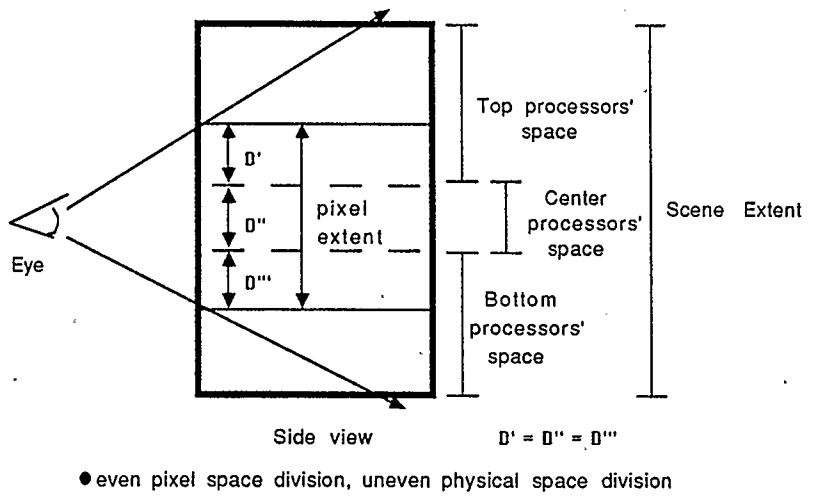


Figure 5-2. Object space boundaries resulting from pixel boundaries.

the array. This creates an octree-like situation where object space is subdivided finer in areas of greatest complexity. Since the face is centrally complex, it exhibits a speedup due to this effect.

This occurrence indicates that by choosing a good subdivision of object space between array nodes, greater than N speedups may be achieved. This idea was not pursued further in this thesis due to time constraints.

5.2. The Timing Methodology

As was mentioned in Chapter 3, the actual mesh hardware was not yet complete at the time of this writing, so a simulation of the mesh was run on a network of workstations. The shared memories for message passing were simulated by a separate

process running on each of the workstations. These ''shared memory'' simulations had the same interface as the functions on the actual mesh hardware. Unfortunately, they did not act in the same way. The only communication channel between the workstations was a packet based network. This caused significant amounts of time to be spent doing collision detection and sending re-tries on the network.

The mesh hardware would not have this difficulty due to it's use of shared memories for a communication channel. The amount of time to send a message would be small and nearly constant; the time needed to write the message to the shared memory and generate an interrupt. The results presented in this chapter ignore the communication overhead introduced by the network. The time each node spent ray tracing is calculated by taking the amount of time spent sending messages and subtracting it from the total amount of time spent in the ray tracing state. Then a small time penalty is added, based on the number of calls to the shared memory. This time penalty is an estimation of the amount of time needed to write 100 bytes to memory, generate an interrupt and do 2 M68000 context switches. This was estimated to be 2 milliseconds.

This procedure is not wholely accurate but it should provide an upper bound on the length of time the mesh hardware will take. It is an upper bound because the time the workstation spent receiving messages is not known. The underlying communication channel on the workstation has to actively participate in receiving a message by pulling it off of the packet network, copying it into an internal buffer, and sending an acknowledgement to the sender. All of these actions may precipitate retries on both ends. The time spent doing a locally initiated ''send'' or ''receive'' was timeable; but the time the underlying communication channel spent doing physical receives from the network was hidden from the ray tracing program, and the

timing mechanism on the workstation was not sophisticated enough to filter this processing time out.

5.3. Pixel Communication Overhead

A major concern when implementing this thesis was that the communication overhead from forwarding pixels would degrade the performance of the primary processor since it must forward every pixel packet computed by the mesh back to the host. Fortunately, this overhead did not significantly affect the performance of the primary processor. This can be seen in Tables 5-3, and 5-4 where the primary processor (node 0) was only the busiest node when it had the largest number of rays, the largest number of objects or some combination of both. This is due to the relatively small amount of time it takes to forward a pixel packet compared to starting and tracing a ray.

Time Spent in Other States

The ray tracing process consists of several states:

- downloading scene space and clipping objects against subvolumes
- assigning objects to voxels
- ray tracing
- flushing any remaining pixel values not yet returned
- collection of timing information and statistics

The assigning of objects to voxels was included in the time taken to ray trace the scene, and in all cases the time spent in states other than ray tracing was less than 0.3% of the total time. Due to message propagation and other factors, this figure will

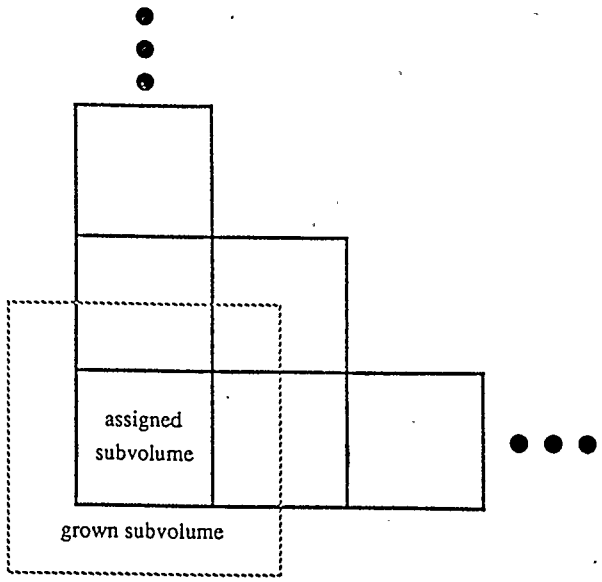


Figure 5-3. A node with a 100% grown subvolume.

increase as the number of processors grows, but not significantly.

5.4. A Proposed New Load Balancing Algorithm

Based on the idea of ''growing'' the subvolume boundaries to avoid floating point precision problems (presented in Chapter 3), John Cleary suggested that the subvolumes might be grown by significantly more than 2% of their initial volume. This means that each node's subvolume will overlap its neighbour's subvolumes by some amount (see Figure 5-3). While this requires more storage on each node, it still does not require that the entirety of object space be stored.

With the grown subvolumes the load will become more balanced. Rays can then be traced further on each individual node, meaning that fewer rays will be passed between nodes. Further load balancing can be achieved by having each node start tracing primary rays from the centre of its pixel array, and spiraling outwards. An idle node asks it's neighbours if there are any primary rays which they have not yet traced. The queried node examines the pixels it has yet to trace and if there is an unstarted row of pixels that enter the overlapped subvolume, the row of pixels is passed to the asking node.

This algorithm was implemented late in this thesis, so very few test runs were done. However, based on the results presented in Table 5-5, this algorithm looks very promising. The table shows the results from runs with the subvolumes grown by 24%

Processor	Processor 1	Processor 2	Processor 3	Processor 4	Speedup
Workstation					
2%	3683	3124	7139	2945	1.64
24%	3410	2758	3772	2823	3.11
100%	2931	2403	3236	2674	3.63
Uni-processor					
2%	241.1	190.7	211.1	172.1	1.40
24%	245.5	198.8	226.9	195.6	1.65
100%	194.3	166.7	183.1	159.4	2.09
simulation					

Sphere scene on a 2×2 processor array
Times shown are in seconds

Table 5-5. Load balance using the proposed load balancing algorithm.

and 100% over their initial volume allocation. There were no unstarted pixels passed to neighbouring nodes in any of the test runs. This is due to each node being able to trace each of its rays further. Table 5-6 shows how the number of rays leaving each processor is reduced as the subvolume is grown. Since a more equal load distributions is achieved, there is no need for untraced pixels to be transferred between nodes. The load is more evenly balanced between the nodes and the speedup approaches N.

As the number of processors used is increased, the likelihood of a node being assigned an empty subvolume increases. By growing the node's subvolume with this method, these nodes will relieve some of the workload from neighbouring processors. In this manner a more efficient utilization of the available processor nodes can be achieved.

This serves as a good indicator that a 2 dimensional processor array can perform at a speedup of close to N as predicted by Cleary and by Ullner.

Growth percentage	Processor 1	Processor 2	Processor 3	Processor 4
2%	1794	763	2427	1926
24%	1079	513	1470	1472
100%	440	212	583	964

Sphere scene on a 2×2 *processor array*

Table 5-6. Number of rays passed from each processor.

Summary

The multi-processor algorithm does provide a speedup over a uni-processor algorithm. The resultant speedup is determined by the distribution of the load and the communication overhead. Since the communication overhead is small, at least with a small number of nodes, the main problem with the approach presented in this thesis is the varying loads which the nodes receive. However, the proposed load balancing method indicates that these problems can be overcome to provide a fast, practical ray tracing algorithm.

With the load balancing method presented, it seems likely that a 2 dimensional architecture can perform at a speedup of close to N.

Figure 5-4. Random spheres.

Figure 5-5. Barroom scene.

Figure 5-6. Face scene.

References

Appel 1968
Appel, A., "Some Techniques for Shading Machine Renderings of Solids," *AFIPS 1968 Spring Joint Computer Conf.*, 1968, pp. 37-45.

Cleary, Wyvill, Birtwistle and Vatti 1986
Cleary, J., Wyvill, B., Birtwistle, G. and Vatti, R., "Multiprocessor Ray Tracing," *Computer Graphics Forum No. 5*, 1, (March 1986), pp. 3-12.

Cleary, Wyvill, Vatti and Birtwistle 1983
Cleary, J., Wyvill, B., Vatti, R. and Birtwistle, G., "Design and Analysis of a Parallel Ray Tracing Computer," *Proceedings Graphics Interface '83*, May 1983, pp. 33-34.

Cleary and Wyvill 1987
Cleary, J. and Wyvill, G., "Fast Ray Tracing Using Uniform Space Subdivision," *to be published in Visual Computer*, 1987.

Cook, Porter and Carpenter 1984
Cook, R., Porter, T. and Carpenter, L., "Distributed Ray Tracing," *Computer Graphics (SIGGRAPH '84 Proceedings) No. 18*, 3, (July 1984), pp. 137-145.

Dippe and Swensen 1984
Dippe, M. and Swensen, J., "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics (SIGGRAPH '84 Proceedings) No. 18*, 3, (July 1984), pp. 149-158.

Dubetz 1985
Dubetz, M., "*Ray Tracing Algorithms for Computer Graphics*," PhD Thesis, Department of Computing Science, University of Alberta, 1985.

Fujimoto, Tanaka and Iwata 1986
Fujimoto, A., Tanaka, T. and Iwata, K., "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications No. 6*, 4, (April 1986), pp. 16-26.

Glassner 1984
Glassner, A., "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications No. 4*, 10, (October 1984), pp. 15-22.

Goral et al. 1984
Goral, C., Torrance, K. E., Greenberg, D. P. and Battaile, B., "Modeling the Interaction of Light Between Diffuse Surfaces," *Computer Graphics (SIGGRAPH '84) No. 18*, 3, (July 1984), pp. 213-222.

Hanrahan 1986
Hanrahan, P., "Using Caching and Breadth-First Search to Speed Up Ray-Tracing," *Graphics Interface '86 Proceedings*, 1986, pp. 56-61.

Heckbert and Hanrahan 1984
Heckbert, P. and Hanrahan, P., "Beam Tracing Polygonal Objects," *Computer Graphics (SIGGRAPH '84 Proceedings) No. 18*, 3, (1984), pp. 119-127.

Immel, Cohen and Greenburg 1986
Immel, D., Cohen, M. F. and Greenburg, D. P., "A Radiosity Method for Non-diffuse Environments," *Computer Graphics (SIGGRAPH '86) No. 20*, 4, (August 1986), pp. 133-142.

Sequent 1985
Sequent Computer Systems, Inc., "Parallel Ray Tracing Case Study," *TN-85-09 (rvp) Rev 1.0, Internal Report*, October 1985.

Kajiya 1986
Kajiya, J., "The Rendering Equation," *Computer Graphics (SIGGRAPH '86) No. 20*, 4, (August 1986), pp. 143-150.

Kay 1979
Kay, D., "*Transparency, Refraction, and Ray Tracing for Computer Synthesized Images*," Masters Thesis, Cornell University, January 1979.

MAGI 1968
MAGI, "3-D Simulated Graphics," *Mathematical Applications Group Inc., Datamation*, February 1968.

Mandelbrot 1983
Mandelbrot, B., "The Fractal Geometry of Nature," *W.H. Freeman and Company (First Edition 1977)*, 1983.

Newman and Sproull 1973
Newman, W. and Sproull, R., "*Principles of Interactive Computer Graphics*," McGraw-Hill, Inc., 1973.

Nishimura et al. 1983
Nishimura, H., Ohno, H., Kawata, R., Shirakawa, I. and Omura, K., "Links-1: A Parallel Pipelined Multimicrocomputer System for Image Creation," *Proc. 10th Annual Int. Symposium on Computer Architecture (SIGARCH)*, 1983, pp. 387-394.

Nishita and Nakamae 1986
Nishita, T. and Nakamae. E., "Continuous Tone Representation of Three-Dimensional Objects Illuminated by Sky Ligth," *Computer Graphics (SIGGRAPH '86) No. 20*, 4, (August 1986), pp. 125-132.

Novacek 1985
Novacek, M., "*Computer Graphics: Particles and Fractals*," Masters Thesis, Deptartment of Computer Science, University of Calgary, 1985.

Peacock 1979
Peacock, J., "*Distributed Simulation Using a Network of Processors*," PhD Thesis, Department of Mathematics (Computer Science), University of Waterloo, 1979.

Peterson 1983
Peterson, J., "A System for High Quality Moving Image Synthesis," *CS 653, Department of Computer Science, University of Utah*, June 1983.

Plunkett and Bailey 1985
Plunkett, D. and Bailey, M., "The Vectorization of a Ray-Tracing Algorithm for Improved Execution Speed," *IEEE Computer Graphics and Applications No. 5*, 8, (August 1985), pp. 52-60.

Reeves 1983
Reeves, W., "Particle Systems - A Technique for Modelling a Class of Fuzzy Objects," *Computer Graphics (SIGGRAPH '83) No. 17*, 3, (July 1983), pp. 359-376.

Reeves and Blau 1985
Reeves, W. and Blau, R., "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems," *Computer Graphics (SIGGRAPH '85) No. 19*, 3, (July 1985), pp. 313-322.

Rubin and Whitted 1980
Rubin, S. and Whitted, T., "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics (SIGGRAPH '80 Proceedings) No. 14*, 3, (July 1980), pp. 110-116.

Rushmeier and Torrance 1987
Rushmeier, H. and Torrance, K. E., "The Zonal Method for Calculationg Light Intensities," *Computer Graphics (SIGGRAPH '87) No. 21*, 4, (July 1987), pp. 293-302.

Ullner 1983
Ullner, M., "*Parallel Machines for Computer Graphics*," PhD Thesis, California Institute of Technology, 1983.

Unger, Birtwistle, Cleary, Hill, Lomow, Neal, Peterson, Witten, Wyvill 1984
Unger, B., Birtwistle, G., Cleary, J., Hill, D., Lomow, G., Neal, R., Peterson, M., Witten, I. and Wyvill, B., "Jade: a simulation and software prototyping environment," *Proc. SCS Conference on Simulation in Strongly Typed Languages*, San Diego, California, February, 1984, pp. 66-73.

Vatti 1984
Vatti, B., "*Multiprocessor Ray-Tracing*," Masters Thesis. Deptartment of Computer Science. University of Calgary, May 1984.

Wallace, Cohen and Greenberg 1987
Wallace, J., Cohen, M. F. and Greenberg, D. P., "A Two-pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," *Computer Graphics (SIGGRAPH '87) No. 21*, 4, (July 1987), pp. 311-320.

Warnock 1969
Warnock, J., "A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures," *Tech. Rep. 4-15*, 1969.

Watkins 1970
Watkins, G., "A Real-Time Visible Surface Algorithm," *UTEC-CSc-70-101, Computer Science Department, University of Utah*, June 1970.

Whitted 1980
Whitted, T., "An Improved Illumination Model for Shaded Display," *Communications of the A.C.M. No. 23*, 6, (June 1980), pp. 343-349.

Whitted 1985
Whitted, T., "The Hacker's Guide to Making Pretty Pictures," *SIGGRAPH '85 Image Rendering Tricks seminar notes.*, July 1985.

Williams, Buxton and Buxton 1985
Williams, N., Buxton, B. and Buxton, H., "Simultaneous Ray Tracing Computer Graphics," *16,887A Project Report*, January 1985.

Wyvill, Wyvill and McPheeters 1985
Wyvill, G., Wyvill, B. and McPheeters, C., "Soft Objects," *Research Report No. 85/215/28, Deptartment of Computer Science, University of Calgary*, October 1985.