4

Parallel Ray Tracing for Real Time Animation

by

Remigio Perales

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCiENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February, 1995

Signature of Author ....
Department of Electrical Engineering and Computer Science, Jan 20, 1995

Certified by ....
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Faculty Supervisor

Certified by ....
Daniel Chen
Thesis Supervisor (Texas Instruments, Inc.)

Accepted by ....
F.R. Morgenthaler, Chair, Department Committee on Graduate Students

Parallel Ray Tracing for Real Time Animation

by

Remigio Perales

Submitted to the Department of Electrical Engineering and Computer Science on Jan 21, 1995, in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science.

Abstract

Ray tracing is an image synthesis algorithm that projects light rays from a viewer location to the closest object in a user defined environment. At the intersection point between a light ray and the closest object, reflection and refraction rays are generated. The reflection and refraction rays are also projected into the environment, thus modeling the physics behind light interactions with the real world. This physical model of light interactions produces some of the most realistic computer generated images.

The realistic images created by ray tracing has made it very popular in computer graphics. However, the mathematical complexity involved in ray tracing and thus the time required for the generation of a single image (frame) has limited its use in animation. This may change as specialized processors (DSPs) become prevalent in computer graphics. The architecture of DSPs is designed for intensive mathematical computation. Parallel networks of such processors yield computational power comparable to supercomputers.

An algorithm developed for use on a parallel network of TMS320C40 DSPs is proposed as a solution for real time processing of ray traced images. The approach uses temporal coherence to eliminate unnecessary computation of pixels. Thus pixels remaining the same color in the following frame are not recomputed.

Tests were performed with 20 moving objects whose temporal coherence ranged from 4% to 10%. The processing time ranged from 1/20 to 1/15 of the time required assuming complete recalculation of the images on the parallel network. Since the algorithm does not require knowledge of the next frames' color, its possible applications include interactive games and military simulations.

Faculty Supervisor: Stephen Ward
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Daniel Chen
Title: Applications Engineer, Texas Instruments, Incorporated

Parallel Ray Tracing for Real Time Animation

by

Remigio Perales

Submitted to the Department of Electrical Engineering and Computer Science on Jan 21, 1995, in partial fulfillment of the requirements for the degrees of Bachelor of Science and Master of Science.

Abstract

Ray tracing is an image synthesis algorithm that projects light rays from a viewer location to the closest object in a user defined environment. At the intersection point between a light ray and the closest object, reflection and refraction rays are generated. The reflection and refraction rays are also projected into the environment, thus modeling the physics behind light interactions with the real world. This physical model of light interactions produces some of the most realistic computer generated images.

The realistic images created by ray tracing has made it very popular in computer graphics. However, the mathematical complexity involved in ray tracing and thus the time required for the generation of a single image (frame) has limited its use in animation. This may change as specialized processors (DSPs) become prevalent in computer graphics. The architecture of DSPs is designed for intensive mathematical computation. Parallel networks of such processors yield computational power comparable to supercomputers.

An algorithm developed for use on a parallel network of TMS320C40 DSPs is proposed as a solution for real time processing of ray traced images. The approach uses temporal coherence to eliminate unnecessary computation of pixels. Thus pixels remaining the same color in the following frame are not recomputed.

Tests were performed with 20 moving objects whose temporal coherence ranged from 4% to 10%. The processing time ranged from 1/20 to 1/15 of the time required assuming complete recalculation of the images on the parallel network. Since the algorithm does not require knowledge of the next frames' color, its possible applications include interactive games and military simulations.

Faculty Supervisor: Stephen Ward
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Daniel Chen
Title: Applications Engineer, Texas Instruments, Incorporated

Table of Contents

List of Figures

List of Equations

Acknowledgements

This thesis would not have been possible without the generosity Texas Instruments exhibited in funding the project. It also would have been impossible without the tremendous support of the TI employees. First, I would llike to thank my supervisor, Rose Marie Piedra, for always showing patience, and for teaching me how to properly use the TI tools. Next, I would like to thank Keith Larson for developing a communications kernel for the parallel network, and for answering numerous technical questions. I would also like to thank Daniel Chen, my company advisor, for supporting and advising me throughout my stay. In addition, I would like to thank Gene Frantz for displaying confidence in my abilities. Finally, Louise Mitchell and Mervin Galloway have always been there for me. Thanks to TI.

At MIT, I would like to thank Professor Stephen Ward for helping me decide the direction of my thesis. Also, thanks to the MIT 6-A graduate program.

Next, I would like to thank those who have influenced my life, thus my thesis. Where to begin?

With my parents, who taught me more than college ever could, and through whose eyes I've seen the important things in life. To my brother Ramon. For growing up with me, and understanding. To my sister Rossanna, for cookies and letters. To my sister Lauriza, with dreams of pets and flowers. To all my grandparents. I'm happy I'm coming home. Where to begin? With my friends. To Baker, for philosphy and basketball. To Matt, for entertainment. To Mark, for competition. To Tina, for conversation. To Kathy, for sweetness, and framemaker!. To Steve, for dreams. To Paul, for reality checks. To Jake, for politics. To Hung Chou, for principles. To Chris Chang, for enthusiasm. To Joe, for advice. To Juan, for humor. To Chris Reed, for hope. To the many other friends from MIT and Texas. Where to begin? With all my high school teachers...I did learn something.

Chapter 1

Introduction

Despite the realism of images produced by ray tracing (Figure 1)[1], its use in animation has been limited due to its time consuming nature. It is not uncommon for scenes to require hours to ray trace. If processing time for ray traced images (images produced by ray tracing) could be reduced, animators would have a powerful tool for realistic animation. The goal of this project is the production of ray traced images in real time (32 frames a second) on a parallel network.

1. Produced by M. Miller on the public domain ray tracer POVRay. Found on the World Wide Web at location http://acacia.ens.fr:8080/home/massimin/ray.ang.html.

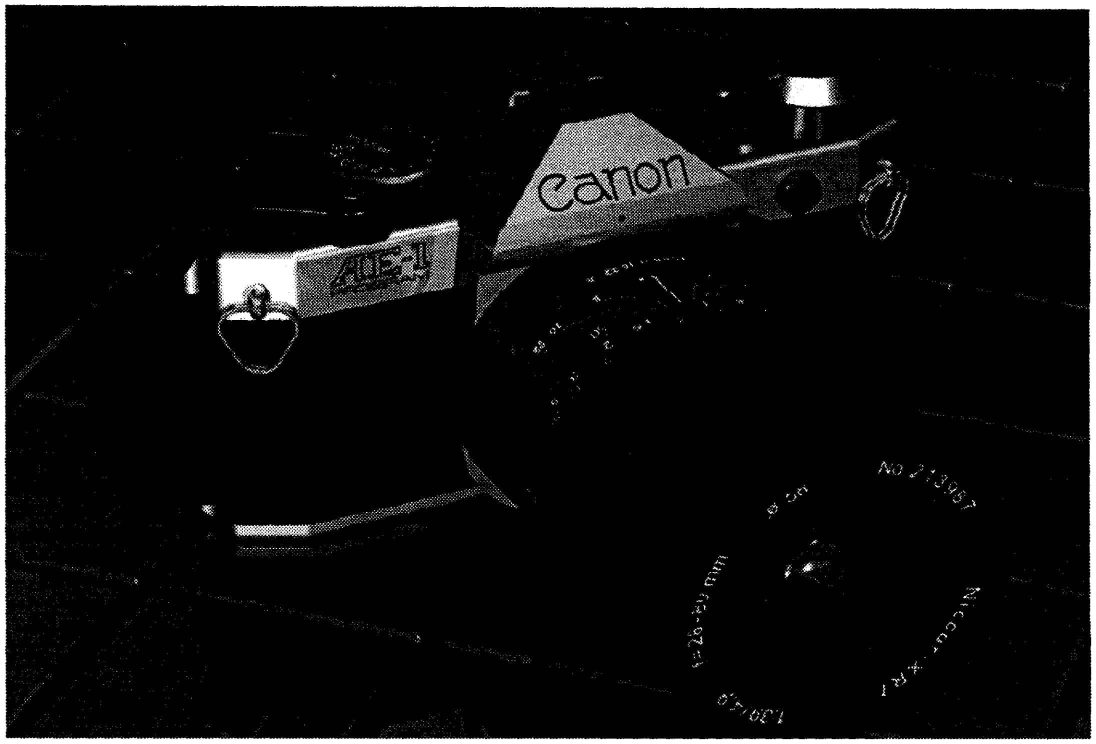


Figure 1: Ray Traced Image

1.1 Motivation

Computer processing speed has been increasing for a decade (Figure 2). However, physics rather than technology will be the ultimate barrier to continued increases in processor speed. The need for more processing power has been a catalyst for much of the current research involved in the field of parallel processing. Parallel processing is the division of a large task into smaller units, and the assignment of these smaller tasks to individual processors. The individual processors can then work concurrently on the smaller tasks. As interprocessor communication becomes less complex, the size of feasible multiprocessor networks will increase. These massive machines will form the basis for desktop supercomputers [18].

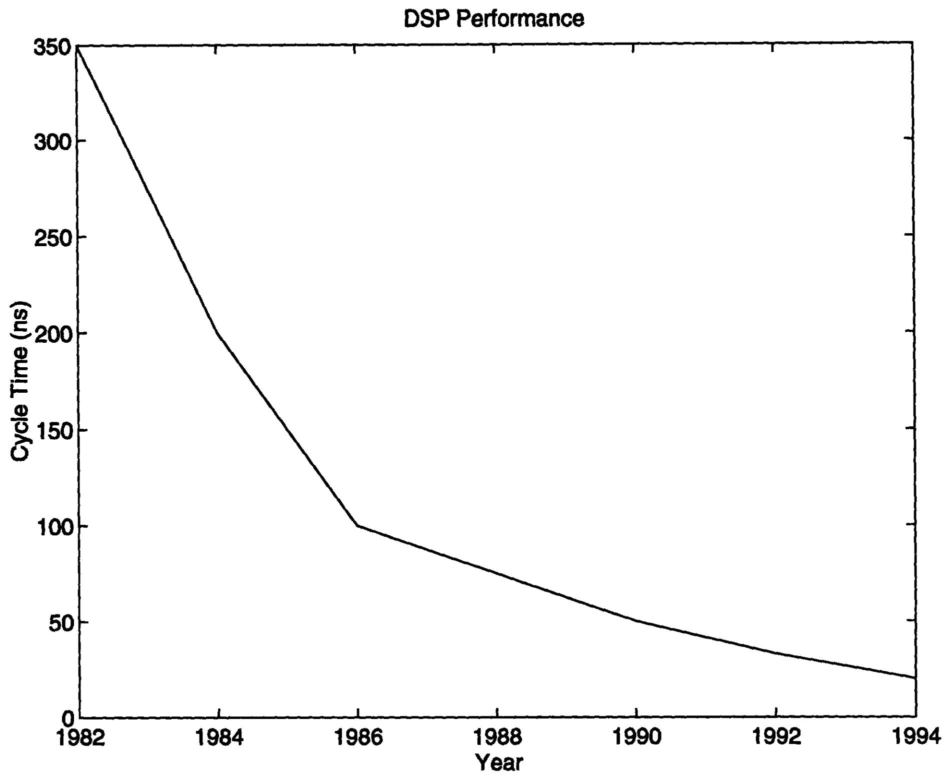


Figure 2: Processor Performance

One of the effects these supercomputers will have is to improve the speed of computer generated images. Different regions of an image may be calculated simultaneously. As a result, many computer graphics algorithms are ideally suited for parallel computation. One of the computer graphics algorithms whose speed improves through parallelization (Chapter 4) is ray tracing. Figure 3 depicts a pictorial explanation of ray tracing (refer to Section 3.2 on page 28 for details).

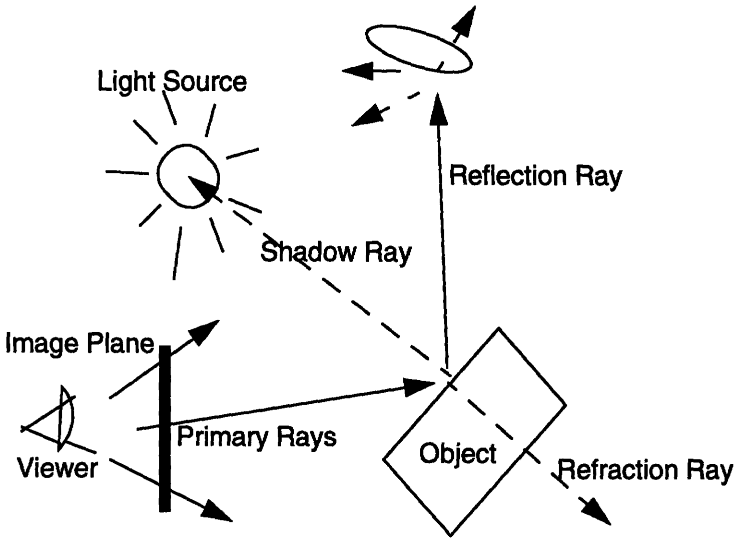


Figure 3: Ray Tracing

Image plane (dark line) is shown with rays from the viewer location intersecting. The points at which these rays intersect the image plane are pixels. This schematic illustrates a one ray per pixel model.

Ray tracing is a 3-D image synthesis routine which produces some of the most realistic computer graphics images. Therefore, it would be appropriate to use ray tracing for a project that required the creation of realistic images for animation.

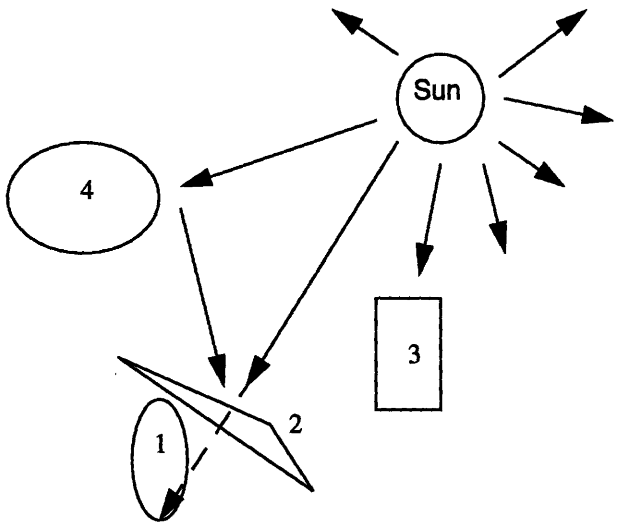
1.2 Purpose

The goal of the project was to produce ray traced images in real time on a parallel network. To accomplish this, an algorithm exploiting temporal coherence was developed, specific to ray traced scenes to be used in animation.

1.3 Motion Picture Coherence

Temporal coherence is the degree to which two successive frames in an animation are similar. Temporal (image space) coherence routines are used to eliminate unnecessary calculations of pixels, by attempting to identify which pixels in adjacent frames remain unchanged. Pixels which remain the same should not be ray traced again. The time savings which the developed algorithm provides are scene-dependent, but can be generalized by examining a 'typical' animation sequence. Previous research [7] of such typical scenes shows an average of 88% frame-to-frame coherence (88% of pixels remained the same color in two sequential frames). Pixel processing times for ray tracing are not constant. The primary light ray (Figure 3) generated for some pixels may not intersect anything. In these cases, the reflection and refraction rays of Figure 3 are note generated, and pixel color is therefore quickly determined. As a result of this variable pixel processing time, the 88% reduction in pixel calculations does not transfer directly to an 88% savings in time.

Motion picture coherence (temporal image space coherence) should therefore attempt to eliminate the processing of 88% of the pixels in a typical scene. The algorithm implemented in this project links all objects in the scene together, via an illumination and shadow network of lists (see Figure 4 and Figure 5). Thus each object contains a list of objects that it either shadows or illuminates.



Object 1 is shadowed by object 2, thus object 2 contains object 1 on its shadow list.

Object 4 is illuminating object 2. Thus object 4 contains object 2 on its illumination list.

Figure 4: Illumination and Shadow Network

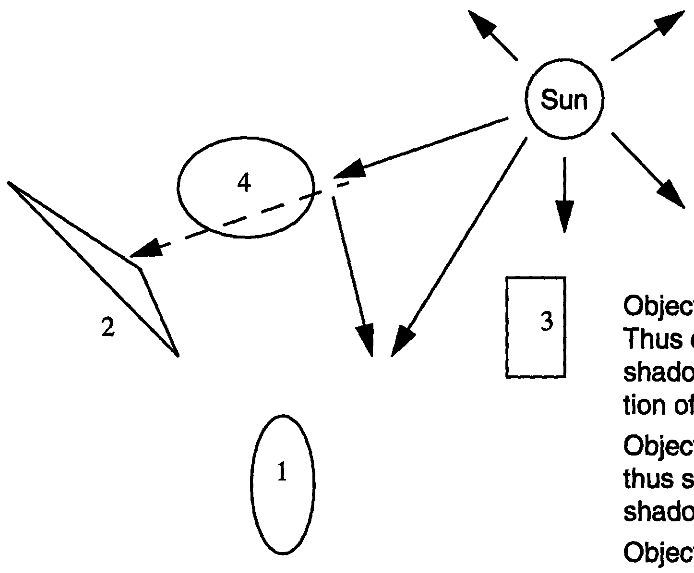


Figure 5: Modified Illum/Shadow Nets

Object 2 is no longer shadowing object 1. Thus object 2 removes object 1 from its shadow list. All pixels which contain a portion of object 1 are flagged for calculation.

Object 2 has moved behind object 4, and is thus shadowed. Object 2 is added to the shadow net of object 4.

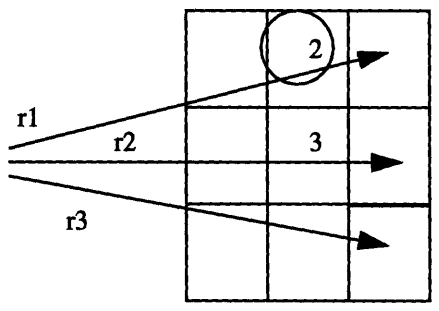
Object 2 has moved to a new location, thus all pixels which cross the path of object 2 are marked for recalculation.

For each frame, the network of lists (nets) must be updated. After the nets have been updated for the new frame, the pixels required for recalculation are determined as follows. First, a projection of moved objects onto the image plane (monitor) occurs. Every pixel which is within this projection is marked for recalculation (see Section 5.2 on page 52 for a further explanation).

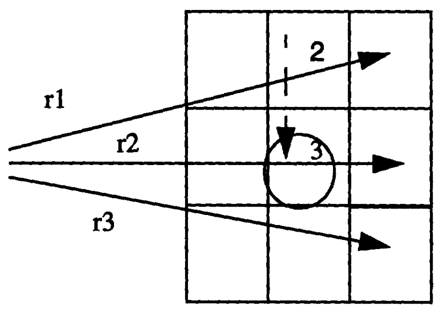
The uniqueness of this technique stems from the ability to use temporal coherence before the next frame in an image is known. Most other methods require either some frame in the future to be calculated and used for comparison, or they require an automatic recalculation of portions of the next frame to determine which pixels have changed color. However, there is one method [13] which attempts a similar strategy as the illumination and shadow nets described above. It achieves a 4-fold increase in processor performance over a standard ray tracing algorithm on a chosen data set. The motion picture coherence routines implemented in this project achieved up to 20-fold speed improvements[2].

2. The animation sequences tested on the two methods were not the same. In addition, Jevans ran his algoirhtm on a sequential processor, while the analysis here is on a parallel system. The parallel speedup (see Glossary) of the algorithm on 3 nodes was approximately 2.75.

The main difference between the two methods stems from the chosen network links. The previous work attempted a uniform subdivision of the world space. Each subdivision (voxel) contained a list of rays (pixels) which passed through it during ray tracing. In addition, the voxel contained a list of objects that was within its space. If an object within the voxel moved, then all rays (pixels) which passed through that voxel were marked for recalculation (see Figure 6) [13].



(a) Frame 1
The sphere is currently contained in voxel 2 and ray r1 is passing through this voxel. The first frame will require calculation of each primary ray: r1, r2, and r3.



(b) Frame 2
The sphere has moved out of the space occupied by voxel 2, and into the defined bounds of voxel 3. The voxels update which objects they contain. Rays r1 and r2 are ray traced in Frame 2.

Figure 6: Voxel Approach

1.4 Organization of Report

This thesis contains six chapters. The first chapter is simply a brief *introduction* to the material that will be covered through the rest of the report, and a goal statement. Chapter 2 briefly presents *The TI DSP and Hardware Platform* used to complete the project. Chapters 3, 4, and 5 describe the steps required for completion of the research: 1) working sequential ray tracer, 2) parallelization of ray tracer, 3) creation of motion picture coherence routines. Each chapter also includes background discussion. Chapter 3 covers *Image Synthesis Techniques,* and includes a discussion of the sequential ray tracing algorithm implemented.

Chapter 4 is *The Parallelization of Ray Tracing.* The last stage of this project was the creation of *Motion Picture Coherence* routines, Chapter 5. The final chapter of this thesis presents the *Results and Conclusions.*

Chapter 2

The TI DSP and Hardware Platform

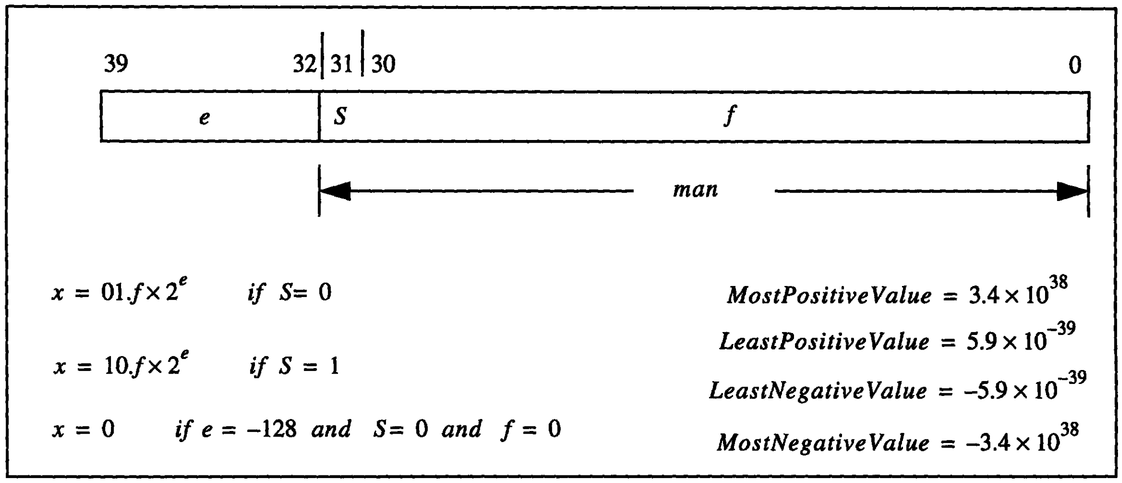
DSPs (Digital Signal Processors) are a broad category of processors whose uses range from control to communication systems. The TMS320C40 DSP (C40) was designed by TI to function in a parallel network. Characteristics such as a glueless communications interface to other C40s, and multiple communication and DMA channels make it ideal for development of algorithms in a parallel network [21].

2.1 The TMS320C40 DSP

2.1.1 Central Processing Unit

The register-based CPU contains a 40-ns instruction cycle. Other features of the CPU are its floating point/integer multiplier, internal busses, the CPU register file, and the CPU Expansion Register File.

Any processor working on ray tracing should perform efficient and precise arithmetic floating point operations[3]. The C40's floating point/integer multiplier is capable of single-cycle multiplications on 40-bit floating point (32-bit integer) values. The precision and format of these 40-bit floating point numbers is shown in Figure 7.



$x = 01.f \times 2^e$ if $S = 0$

$x = 10.f \times 2^e$ if $S = 1$

$x = 0$ if $e = -128$ and $S = 0$ and $f = 0$

$MostPositiveValue = 3.4 \times 10^{38}$

$LeastPositiveValue = 5.9 \times 10^{-39}$

$LeastNegativeValue = -5.9 \times 10^{-39}$

$MostNegativeValue = -3.4 \times 10^{38}$

Figure 7: Format of 40 bit Floating Point Value

The two 40-bit internal CPU busses and two 40-bit register file busses allow parallel operations in a single cycle, making programming more efficient. Two operands from memory and two from the register file are carried to the multiplier/ ALU thus performing parallel multiplies and adds/subtracts on the four integer/ floating point operands.

Another feature important to the programming of C40s is the CPU register file. The CPU register file holds the Interrupt Enable Register (IIE). The IIE enables/disables interrupts from internal timers, communication ports, and DMAs. In addition, the CPU register file holds other important registers such as the Status Register, Data-Page Pointer, System Stack Pointer, and Repeat Counter.

3. This preciseness is required for the intersection routines which must accurately determine which object is first intersected by a traversing ray.

The CPU Expansion Register File contains the Interrupt-Vector Table Pointer (IVTP), and the Trap-Vector Table Pointer (TVTP). The IVTP points to the start of the Interrupt Vector Table (IVT). The IVT contains pointers to the code that handles each communication port interrupt.

2.1.2 Memory Interfaces

The C40 contains an addressable memory range of 4 Gwords. This range includes the program memory (on chip ROM and RAM, and external memory). In addition, the control registers for the communication ports, DMA channels, and timers are also within this memory space. The memory map of Figure 8 shows the usual configuration with the upper 2 Gwords accessible by the local bus, and the lower 2 Gwords by the global bus [21].

The C40's memory map is shown with the size of each region in words displayed on the left. The lower 2 Gwords are accessible through the global bus. The address range is listed on the right.

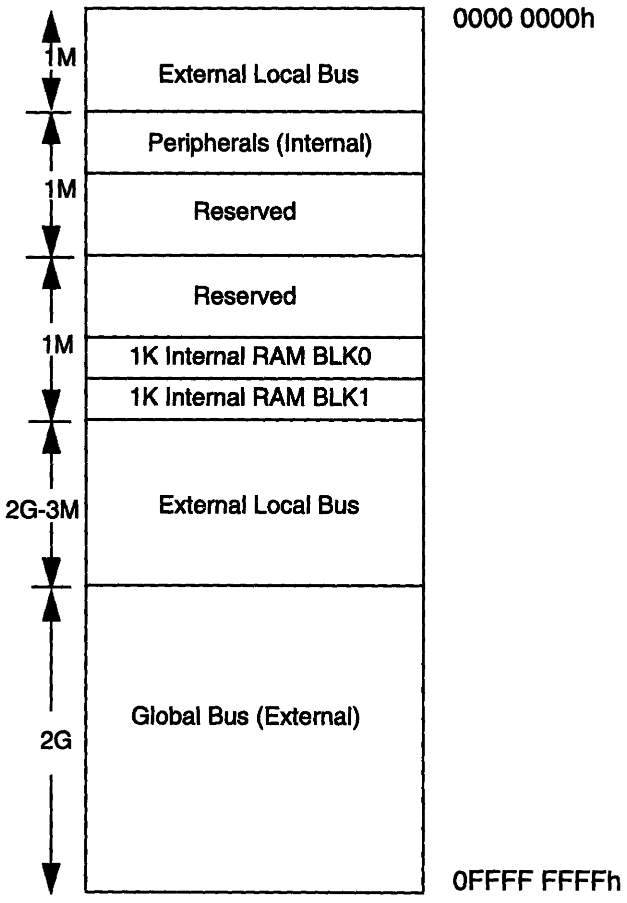


Figure 8: C40 Memory Map

A self-programmable direct memory access (DMA) coprocessor is capable of memory transfers to and from any portion of the C40's memory map without halting execution of the CPU. The DMA coprocessor controls six DMA channels which perform the data transfers. These DMA channels are initialized via nine registers (Figure 9) which determine when to begin the data transfer. In addition, these registers contain the source and destination address, and block size of the memory transfer. With the DMA handling much of the I/O, the CPU is free to compute thus increasing performance of the C40 to 275 million operations per second.

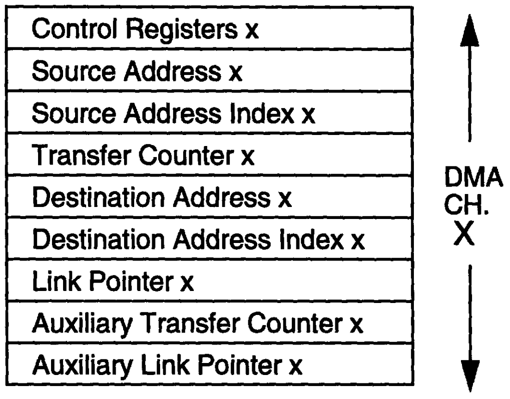


Figure 9: DMA Registers

2.1.3 Communication Ports

A large number of communication ports is required for a processor in a parallel network. These ports serve as communication links, through which the transfer of data between processors occurs. For a 3-D mesh as pictured in Figure 10, a minimum number of six communication ports is needed (the middle processor has six communication links).

This figure depicts a parallel network of processors arranged in a 3 dimensional grid. The processor in the middle of the structure is connected to 6 other processors.

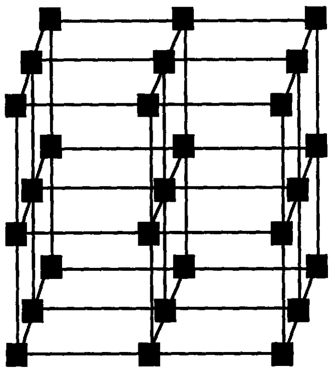


Figure 10: 3-D Parallel Grid

The C40 has six bi-directional communication channels. These make the C40 an ideal processing node in the 3-D grid of Figure 10. The C40 transfers data through each of its six communication ports at 20 megabytes per second. Each channel contains eight data lines (8 bits) and four control lines. The direction of transfer is handled by token passing. The processor with the token (control of the line) may transfer data. Through manipulation of the control lines, ownership of the token is determined.

The four control signals involved in interprocessor communication are the token request (CREQ), token acknowledge (CACK), data strobe (CSTRB), and data ready (CRDY). The basic protocol for data transfer is as follows: Assume Processor A has the token initially. Processor B wishes to transfer data to A. Thus Processor B asserts CREQ. Processor A will relinquish control of the line by asserting CACK. Processor B, obtaining ownership, may now place data on the lines and activate CSTRB. All that remains is for Processor A to signal reception of the data by assertion of CRDY.

The CPU may write directly to the FIFOs of the communication ports. However, it may be beneficial to setup the DMAs to transfer to the communication ports' FIFOs. Through proper setting of the control registers of Figure 9, the DMAs can be configured to transfer to/from a communication port's FIFO upon a communication port interrupt. Thus data transfers may occur between processors without interrupting the CPUs. This leads to a data throughput of 320 Mbytes / sec.

2.2 Hardware Platform

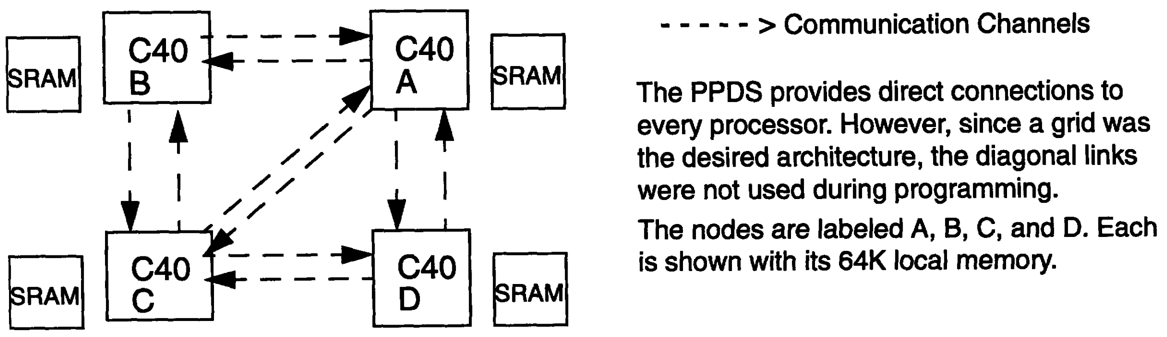
2.2.1 The Personal Computer

Initial research undertaken was performed on a 486DX with 16 Meg RAM and Super VGA. OS/2 was the required operating system since the TI parallel software development tools were only functional under this environment.

Further research was performed on a PC with a Pentium processor. The PC's duties were limited, thus avoiding the critical floating point error of the Pentium processor. The major duties of the PC involved communication with the Parallel Processing Development System (PPDS) via a printer port, and display of the pixel color information returned from the PPDS.

2.2.2 PPDS

The PPDS was the parallel network used during research. It consists of four C40s, all of which are directly connected to each other via one or two of the six communication channels (see Figure 11). In addition, eight external communication connectors allow communication to external devices, i.e. the PC.



The PPDS provides direct connections to every processor. However, since a grid was the desired architecture, the diagonal links were not used during programming.

The nodes are labeled A, B, C, and D. Each is shown with its 64K local memory.

Figure 11: PPDS Configuration

Another feature of the PPDS is its distributed memory capability (see Glossary). 64K x 32-bit words of zero wait-state SRAM are available to each processor's local bus. Shared memory capability also exists as 128K x 32-bit words of one wait-state SRAM is accessible through a shared global bus.

Loading of a program onto a C40 can take place through the communication channels or from a specified memory location. Thus each processor's local bus also has access to an 8K byte local EPROM which contains the code to be loaded on the processors upon reset[4].

2.2.3 Communication with PC/Complete System

The C40 was capable of communicating with a PC's bi-directional printer port through an interface card which emulated the printer port's protocol. A communication port of the root processor was connected to this interface card. The maximum rate of transfer through this interface was 166 Kbyte/second, thus limiting the minimum processing time of a standard image to 6 seconds[5]. The complete hardware system used for this project is pictured below in Figure 12.

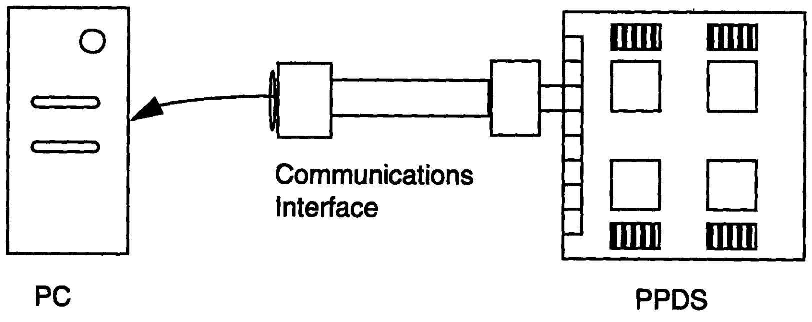


Figure 12: Complete System

4. The PPDS is configured by TI to automatically load each processor from EPROM. These hard-wired control signals were modified to allow processors to load from communication ports.
5. A standard image is 640*480 pixels in size. Each pixel has a red, green, and blue component, which translates to 3 bytes per pixel. Thus the size of a complete image transfer is 640*480*3, or approximately 1 megabyte. Temporal coherence reduces the number of pixels which must be transferred. Those pixels that are not recalculated, are not sent to the PC again.

Chapter 3

Image Synthesis Techniques

Ray tracing is not the only image synthesis technique in existence today. In fact, it is not even the most widely used approach. This chapter discusses other approaches, justifying the use of ray tracing for a realistic real time parallel animation system. Ray tracing is then explained, followed by a synopsis of its history. The final portion of this chapter presents the ray tracing algorithm chosen, and the platform used for creation of the sequential ray tracer.

3.1 Alternative Algorithms

3.1.1 Scan-line Approaches

An important issue in generating realistic synthetic images is the removal of surfaces which are not visible. This is known as Hidden Surface Removal (HSR). For example, the painter's algorithm begins by rendering objects furthest from the viewer (monitor). This algorithm accomplishes HSR by "painting" objects which are closest to the viewer last, thus occluding any objects which were previously painted [10].

Scan-Line approaches to rendering images are probably the most prominent form of image synthesis today. These techniques transform the vertices of all polygons in the geometrically described world to the coordinate system of the monitor. It is then possible to determine the ordering of the polygons along the horizontal and vertical axis of the monitors coordinate system. This ordering, along with a sorting of objects according to depth, leads to a systematic manner of rendering the image one scan-line at a time [8].

Rendering begins by searching the first scan-line for the first object's edge. When this is found, a shading algorithm is applied at this point on the object. The next point to be shaded is determined through an incremental change along the object corresponding to an increment of one pixel along the monitor's horizontal axis. For these scan-line algorithms, HSR can be accomplished through application of the painter's algorithm on each scan-line, or through z-buffering techniques (see Glossary of Terms).

The quality of images resulting from the above techniques varies depending on the exact implementation, and the shading algorithm chosen. For example, a basic shading algorithm used a Lambertian diffuse term (cosine dependence) to determine the illumination of an objects surface. This used the fact that a light source normal to the object exhibits a higher intensity than light at an angle. Surfaces which are at a 90 degree angle with the incoming light have no contribution from this diffuse term, and will appear black. As a result, an ambient lighting term is often included in the shading equation (see Equation 1) [10].

Equation 1: Shading Function

$$I = I_a k_a + I_p k_d (\bar{N} \cdot \bar{L})$$

(See Appendix A for description of parameters)

Phong [19] presented the computer graphics world with its most popular shading equation. He added a specular component which is related to the angle between the viewing direction (V) and the light source's reflection direction (R). Smaller angles produce greater contributions from the specular component [22].

Equation 2: Phong Shading

$$I = I_a k_a + I_p k_d (K_d \cdot (\bar{N} \cdot \bar{L}) + K_s \cdot (\bar{R} \cdot \bar{V})^n)$$

(See Appendix A for description of parameters)

These shading techniques are incapable of convincingly modelling reflection and refractions. Ray tracing improves on the above methods by realistically modelling reflections and refractions. In addition, the HSR problem is solved indirectly.

Before further exploration of ray tracing, another method of generating realistic images is briefly examined -- radiosity.

3.1.2 Radiosity

Radiosity subdivides the 3-D scene into surface patches of constant energy flow. Each patch generates a constant radiation of energy, which is a function of reflected and emitted energies. As the viewer moves around in the scene, the energy flux from each patch remains constant. Recalculation of a patches flux is not required. The only processing required as a viewer moves is the determination of which patch is visible. Patches visible to the viewer determine the color of pixels [10].

Some animation scenes are only changes in view. For example, a camera moving in for a close-up on a character does not require the character to move. The result is therefore only a change in view. If the motion is a change in an object's location, and not a change in view, the radiosity illumination algorithm must be applied to every frame in its entirety.

The research presented in this paper pertains to temporal (image space) coherence between frames. Temporal (image space) coherence can be used to eliminate large portions of the required pixel processing, when only a few objects have moved. Radiosity requires complete recalculation of the entire image when objects move. Therefore, radiosity could not benefit from the proposed image space temporal coherence algorithm.

Ray tracing, however, can benefit from temporal (image space) coherence. In addition, radiosity generally is an order of magnitude slower than ray tracing in image generation [22], and therefore is less feasible for any real time animation implementation. An explanation of ray tracing follows.

3.2 Description of Ray Tracing Algorithm

As shown in Figure 3, primary rays originate from a viewer location. These rays pass through an image plane (monitor) which is a specified distance from the viewer. The rays continue beyond the image plane until an intersection between some object in the environment occurs.

At the point of intersection, the primary ray is reflected and refracted. In addition, shadow rays are generated. Shadow rays are rays which have been sent out from the point of intersection, in the direction of every light source within the environment. Shadow rays, as their name implies, determine what light sources are illuminating the current point of intersection, and what light sources are occluded (obstructed) by objects. As the reflection and refraction rays (child rays) intersect other objects in the world, they also spawn child rays. The process of child rays generating child rays is known as recursive ray tracing. Child rays are generated until the level of the family tree has surpassed a program defined depth, which is three in the case of Figure 13.

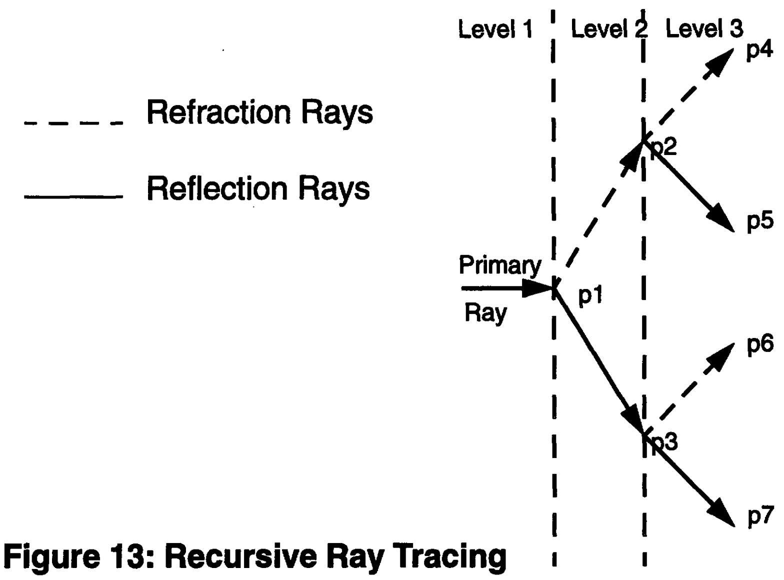


Figure 13: Recursive Ray Tracing

When this depth is reached, the color of the original point of intersection is determined in an inverse manner. A shading algorithm (see Equation 1), which determines the color of a point using object characteristics such as ambient (Ka) and diffuse (Kd) coefficients, is applied at each point in Figure 13, [23].

This shading equation is first applied to every point in level 3. The shading algorithm is then applied to every point in level 2. The color of every child ray is multiplied by a reflection (Kr) or refraction (Kt) coefficient, and then added to the color of its parent in level 2. Thus, the color of point 3 (p3) in Figure 13 is:

Equation 3: Ray Tracing

$$ColorOfp3 = I(p3) + (ColorOfp6) \cdot K_t + (ColorOfp7) \cdot K_r$$

The last point shaded is the intersection of the primary ray and the closest object, i.e. level one. For simple implementations, the color at the intersection point of a primary ray with an object corresponds to the color of the pixel located at the point where the primary ray crosses the image plane (monitor). Refer to Figure 3.

Due to the independence of each primary ray's color, simultaneous calculation of primary rays can occur. Thus parallelization would improve the speed of a ray traced image (Ideally, the speed would improve linearly with the number of processors in the network). The images produced by ray tracing are generally agreed to be some of the most realistic computer generated images. Since the goal of this project is to create realistic 3-D animated sequences on a parallel network (in real time), ray tracing is ideal.

3.3 History of Ray Tracing

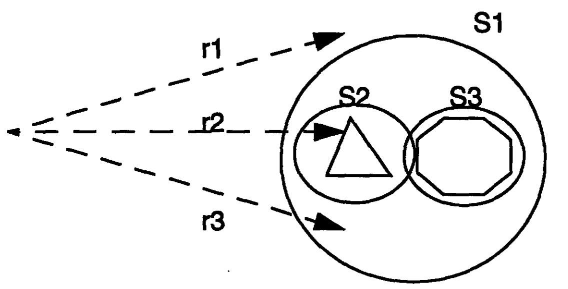
3.3.1 Early Models

The idea of casting a ray into space in order to simulate light interactions with the world was first presented in a paper by Appel [1]. However, it was not popularized until Whitted [23] introduced recursive ray tracing and produced impressive images. Some complex images require hours or even days to ray trace, thus researches have attempted to improve the efficiency of the ray tracing algorithm.

For non-trivial scenes, approximately 95% of processing time is spent computing ray-object intersections [23]. Thus early research focused on reducing the ray-object intersection cost. Techniques proposed by researchers included object bounding volumes [23] and hierarchical descriptions [20].

Bounding volumes simplified the cost of intersecting rays with objects by tightly surrounding complex objects with a simpler object (see Figure 14a). A bounding volume is the smallest volume which completely contains an object. Tight bounding volumes reduced the probability that a ray would intersect the bounding volume, and fail to hit the complex object within. Intersection with the complex object only occurred if the ray pierced the simple bounding volume.

Hierarchical approaches eliminated the number of objects that had to be intersected with each ray (see Figure 14b). When a ray failed to pierce the bounding volume (spheres in the case of Figure 14) of a node, all children of the node were eliminated from further processing, since they encompassed a subregion of the parent node. This reduced computation time from order(N) to order(logN), where N was the number of objects in the scene.



a) Ray r3 will be tested for intersection with S2. Since it does not intersect S2, the costly intersection with the complex object within S2 is avoided.

b) Ray r1 fails to pierce the bounding volume, S1. Thus no intersection with the child nodes, S2 and S3, is required.

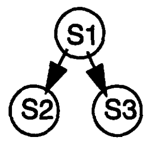
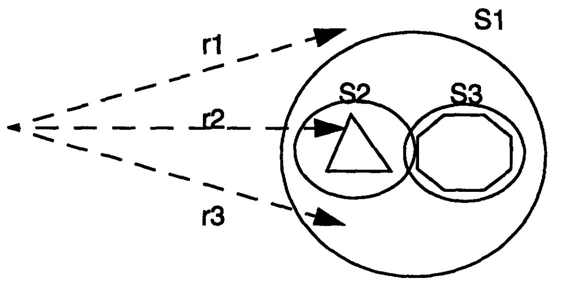


Figure 14: Hierarchies for Ray Tracing

3.3.2 Advanced Methods

In 1986, Kay [15] improved on the hierarchical approach by including a heap to store the closest ray-object intersection point. Rays travel through space penetrating bounding volumes that are closer before bounding volumes which are further (Figure 15). However, the space occupied by bounding volumes is not mutually exclusive, and it is therefore required that the next bounding volume within the same level of the hierarchy must also be checked for intersection. The purpose of the heap, as mentioned previously, was to store the closest ray-object intersection point. If the ray-object intersection point stored on the heap comes before the ray-bounding volume intersection, the intersection of the ray with each object within the bounding volume may be eliminated.



Ray r2 intersects the outer bounding volume, S1. As a result, the children must be intersected with the ray. Ray r2 first encounters child S2, which it also pierces. It hits the object within S2 and places the time until this intersection on the heap. The ray then intersects S3. Since the intersection time with S3 is greater than the time stored on the heap, no further processing is required.

Figure 15: Heap Approach

Another approach developed to reduce the number of intersections each ray must undergo is the octree method. It was first published in 1983 by Matsumoto [16]. The idea here is to divide space with a resolution dependent upon object density. Subdivision occurs where object density is above a prescribed level, and continues until the size of the subdivision (cell) reaches a defined minimum. Unlike the hierarchical approach, cells do not overlap, and thus when an intersection is found, no further processing is required.

The above techniques benefit from object coherence -- an object occupying a region of space is also likely to occupy space local to that region [14]. A uniform division of space into a 3-D grid -- spatial subdivision -- also exploits object coherence. An approach proposed by Kaplan [14], "adaptively subdivide(s) all of three-dimensional space based on objects in the scene." Rays are only intersected with objects lying in their path as they travel through this spatial division.

These are just a few of the algorithms which attempt to decrease the processing time of ray tracing. In fact, the algorithms mentioned have only touched upon one aspect of increasing the speed of ray tracing -- object (spatial) coherence. Algorithms seeking to exploit coherence in other forms exist. Ray and temporal coherence are examples.

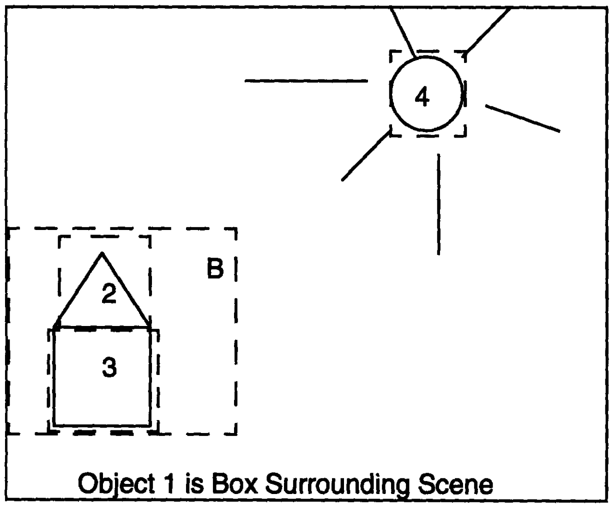
Further exploration of object and ray coherence (see Glossary) is not required. Time saving techniques exploiting both these methods may be used in conjunction with the algorithm developed here, which as mentioned previously is based on temporal coherence.

3.4 Ray Tracer Selection

The project began with the selection of a ray tracing algorithm that supported options necessary for the eventual animation of scenes. For example, one required option was a method of storing and organizing objects as they were initially loaded. In addition, animation demands the availability of such an object organizer. As objects leave and enter a processor's space during animation, new objects must be arranged, and objects no longer within the current processor's memory deleted (refer to Section 4.5 on page 44).

Hierarchical bounding boxes is the method of organization included on the chosen ray tracer (Figure 16). This approach creates a hierarchy of objects. The highest level of the hierarchy corresponds to the complete object space. Next, a subdivision of the complete object space is performed. New regions containing sub-objects are created (as seen in Figure 16). These regions are now children of the original upper level of the hierarchy. The process of dividing each of these children into smaller physical spaces continues until all objects in the scene are contained within an independent bounding box.

A hierarchical heap as described in Section 3.3.2 on page 31 also exists for improved performance.



Hierarchical Division of Space:

Division of object space continues until only a single object exists within each subdivision.

The tree created for this simple image is shown below.

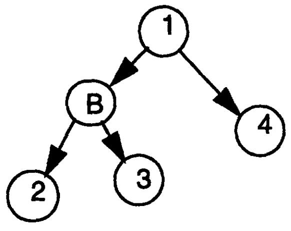


Figure 16: Hierarchical Bounding Boxes

3.5 Sequential Ray Tracer

After the ray tracer was chosen, porting of the chosen algorithm to a single processor platform was performed. This single processor implementation is known as a sequential ray tracer. The platform chosen for the sequential algorithm was a PC plug in board with a TI C40 and 64K local SRAM. The support for this system included a library of graphics functions which allowed for easy display of images.

Chapter 4

The Parallelization of Ray Tracing

Due to the independence of primary rays, ray tracing has been attractive to researchers of parallel processing. Issues explored include static and dynamic load balancing, distribution of the object space, types of message passing, and parallel architectures. Chapter 4 reviews current literature related to parallelization of the ray tracing algorithm. This is followed by the strategies implemented during the author's research.

4.1 Terms Common to Parallel Processing

A few terms common to papers related to parallel processing are briefly explained here to facilitate the understanding of this section.

Each vertex in the network of Figure 10 is a *node*. Each node is an independent processor. These processors together form a *parallel network*. The term *root,* as used in parallel processing, refers to the node in the network which is controlling the flow of the algorithm. Nodes are assigned addresses in the network. This address is known as the *node ID*.

4.2 Parallelization Issues

When Whitted [23] presented his paper, he suggested that the massive amount of computation involved in the intersection routines be performed by parallel processors, and that the shading be handled by a dedicated host. Due to the independence of primary rays, it was a natural extension to compute both the shading and intersections on nodes of a parallel network.

Primary rays (pixels) are the unit of computation distributed between nodes of a parallel network. Some primary rays may intersect reflective and refractive surfaces while others may intersect nothing. Thus the processing time for each primary ray varies widely. As a result, balancing the computational loading of a parallel network performing ray tracing is problematic.

Parallelization of both object space and primary rays (pixels) can occur. Parallel systems have been developed using object partitioning and ray partitioning (see Glossary). These systems attempt to balance interprocessor communication and load balancing.

Static and dynamic load balancing methods are usually employed to increase system performance. Intelligent initial distribution of pixels (or objects) to the nodes in the network for processing is known as static load balancing. Dynamic load balancing redistributes pixels (or objects), when some processors are more heavily loaded than others. Together, these methods attempt to eliminate processor idle time in a parallel network.

The interprocessor communication of a distributed memory system is related to the degree of object partitioning in the scene. Object partitioning distributes objects of a scene throughout a parallel network. This increases the complexity of images which can be processed by the network, but also increases the system message traffic.

The messages communicated between processors in distributed parallel ray traced systems varies. When a ray exceeds the spatial bounds of the local node's memory, two things can happen. Either the ray is passed to the node containing the portion of memory the ray has entered -- ray passing. Or the memory is requested from the remote node containing that physical space -- object passing.

Parallel ray tracing systems can be divided into 4 categories depending on the type of messages generated:

- Autonomous Systems
- Ray Passing Systems
- Object Passing Systems
- Combined Approaches

4.3 Previous Work

4.3.1 Autonomous Node Systems

- Nishimura, Ohno, Kawata, Shirakawa, & Omura [17]

These systems are the simplest of all parallel ray tracers and require the duplication of the entire data scene on each node, thus the transfer of rays or objects is unnecessary. A 64 processor system known as LINKS-1 was used to create a ray tracer of this type. The local node memory size limited the complexity of the scenes rendered. With all 64 nodes ray tracing an image, the parallel efficiency (see Glossary) of the system was 50%.

4.3.2 Ray Passing Systems

- Dippe & Swensen [9]

Dippe and Swensen, 1984, proposed a method of distributing objects through a uniform 3-D spatial subdivision scheme. Initially, each node of a 3-D grid was assigned a region of the divided space. Connected processors contained adjacent regions of space. They suggested that through dynamically adjusting the boundaries of the space stored within each node, load balancing could be achieved. There are several weaknesses to this approach. First, a finer subdivision of the space is required to further reduce the number of ray-object intersections. Another problem is the overhead of dynamically adjusting the spatial boundaries.

- Priol & Bouatouch [5]

Priol and Bouatouch, 1989, implemented a completely static load balancing technique which utilized both ray and object partitioning on an Intel iPSC Hypercube. First, the image was sub-sampled to determine which regions required heavier processing. Then, both object space and pixels were divided between processors in order to evenly distribute the load.

The rays which were sent to remote nodes for processing were not returned to the originating node. Instead, the contribution of that ray to the primary ray's pixel color was sent to the host processor which held the frame buffer. The received contribution was accumulated in the appropriate pixel location of the frame buffer. On a network of 64 processors, the observed parallel speedup was 19, thus the observed parallel efficiency of the network was 30%.

4.3.3 Object Passing Systems

- Priol & Badouel [2]

The work of Priol and Badouel, 1992, centered around creating a Shared Virtual Memory (SVM) system from a distributed memory system. A virtual memory system was implemented through a static allocation of contiguous portions of the object space to adjacent processors of an Intel Hypercube iPSC/2. This initial static allocation of memory is permanent. When a ray has travelled beyond the bounds of the local memory's space, the next region of memory required is requested from the appropriate remote node, and loaded into the local node's cache, thus creating the virtual memory system.

Static and dynamic ray partitioning is performed. Static partitioning is used to reduce remote accesses of memory by exploiting ray coherence. If a node is assigned primary rays which intersect objects within its local database, fewer requests for remote memories are required. In addition, if adjacent primary rays are computed sequentially, the efficiency of the cache is improved, further reducing remote requests.

Dynamic allocation of pixels improves load balancing. If the size of the work load is too small, excessive requests for work will be generated in the network, and the cost in communication may exceed the benefits of load balancing. Through experimentation, Priol and Badouel [2] found a 3x3 array of pixels to be the ideal task size for their system.

One of the benefits of this virtual memory system is the ability to ray trace images which exceed the memory capacity of a single processing node. Testing was undertaken on scenes which exceeded this single node capacity. As a result, the efficiency of the algorithm was difficult to measure against the performance on a single processor. However, analysis was performed to determine the overhead involved in the parallel communications. With a network of 64 processors, parallel efficiency was found to be better than 78%.

4.3.4 Combined Approaches

- Green, Paddon, & Lewis [11]

Green, Paddon, and Lewis, 1988, implemented an object partitioning strategy on a 10 transputer network (see Glossary). Eight processors as shown in the tree structure of Figure 17 were actually used for ray tracing. One processor served as the root, and another processor was dedicated to display.

R is the root processor which generates the initial work for the system -- primary rays. In addition, the root serves as an interface to the host computer.

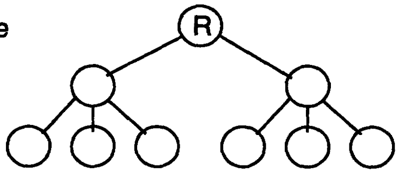


Figure 17: Transputer Network

The algorithm subdivided the scene via an octree. Each node contained the entire octree description. In addition, a portion of the complete database resided within each node's memory. Since the entire database was not present within each node, requests for objects from remote nodes were performed when a ray encountered an object not stored within local memory. Objects were stored in a cache which was updated by the Least Recently Used (LRU) approach.

Load balancing is performed as children request work from their parent. The work load of a parent is stored on a stack. Primary rays are initially on the work stack. Reflection and refraction rays (secondary rays) are placed on the stack as they are generated. Due to the illumination function, a primary ray's color cannot be determined until its secondary rays have been computed. A primary ray whose secondary rays are being computed by remote processors is placed on a *delayed work list*, where it stays until the secondary rays have been returned.

The size of the cache was varied during testing to study its usefulness. For the 8 transputer configuration of Figure 17 and with a cache size 0.5% of the complete database, parallel speedup of 4.04 was obtained.

4.4 Parallel Implementation

After the sequential ray tracer of Chapter 3 was completed, parallelization of the algorithm could begin. This section outlines the object passing parallel algorithm developed for this project. It is a combination of the previous algorithms described above with a virtual memory system similar to Priol and Badouel [2]. In addition, static and dynamic load balancing techniques are implemented.

4.4.1 System Configuration

The PPDS described in Chapter 2 was completely connected as depicted in Figure 18. The shared memory of the system was ignored in favor of the distributed memory. Bob Boothe [4] states that ray tracing on a shared memory system is “a nearly solved problem”.

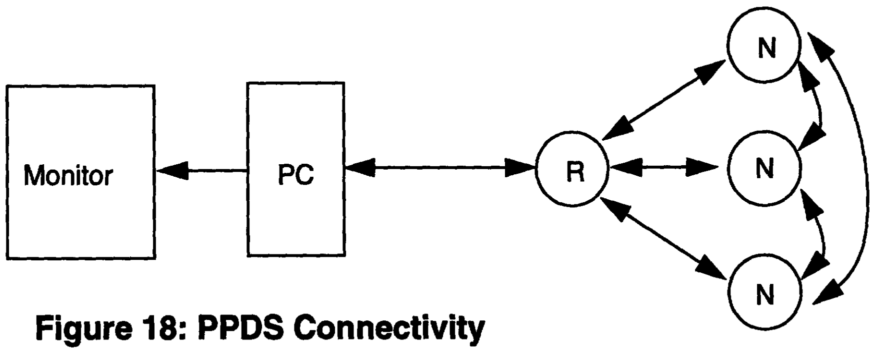


Figure 18: PPDS Connectivity

The root (R) is responsible for I/O to the PC, initial distribution of the scene, and dynamic load balancing. The three node processors perform ray tracing.

The node ID of each processor in the network is hard-coded. Communication is handled through inclusion of a header at the beginning of each message. The header contains the destination node ID, the source node ID, the length of the message, and type of message (see Appendix B, function *cpuint()* for a listing of message types). The source and destination node ID are unnecessary due to the direct connectivity of every processor, but allow for easier porting to a larger network[6].

4.4.2 Object Partitioning

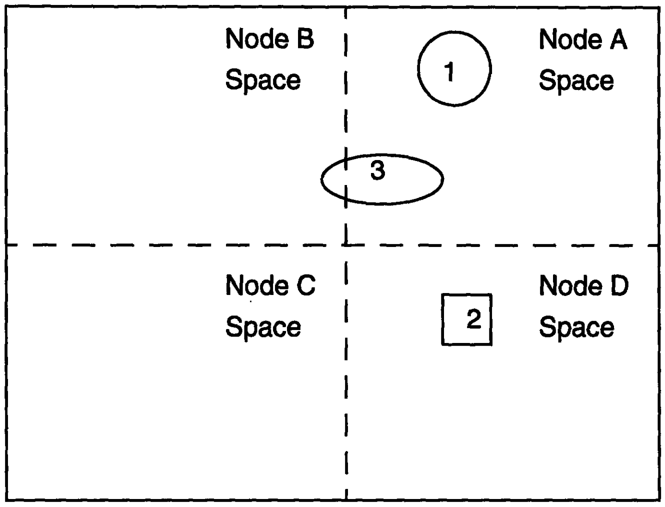
- Initial Distribution to Permanent Memory

A simple algorithm is implemented which assigns objects to nodes based on the objects location in the scene. This is shown in Figure 19. Objects are assigned to nodes depending on the bounds of the object, and the bounds of the space

6. Although the network was completely connected, the algorithm was programmed assuming a 3-D grid. Direct connections eliminated the necessity of forwarding messages.

designated as belonging to a node. Thus in Figure 19, Node A has been assigned the upper right hand region of physical space. Object 1 is located in this region, and therefore is given to Node A, to store as part of its permanent local memory.

This method leads to an unbalanced distribution of objects. During animation objects pass between the physical space assigned to each node, and thus must be added or deleted from a node's permanent memory. Therefore, attempting to balance the distribution of objects initially is useless. A dynamic attempt to equally distribute the objects would lead to massive communications overhead. The unequal distribution is then acceptable.



Each node in the network is assigned a portion of the object space to hold in local memory.

If an object is contained within 2 regions, then it is assigned to both nodes. For example, object 3 is located in the region assigned to Node A as well as the region assigned to Node B. It is therefore sent to both nodes A and B.

Figure 19: Distribution of Objects

- Virtual Memory System

A virtual memory system was implemented to increase the effective memory of the system. When a ray encountered space that was not present locally, the node which contained that space was determined and a request for memory issued to that node. The algorithm then falls into the object passing category of parallel ray tracers. In this implementation, the entire permanent memory of the remote node was transferred upon a request. Thus the virtual memory page size varied with the permanent storage of nodes.

For larger systems without direct connections between each node, the size of these memory transfers could deadlock the system. Ideally, object coherence would minimize the amount of memory transfers, reducing deadlock possibilities.

4.4.3 Task Allocation

- Static Load Balancing (First Frame)

The initial allocation of pixels to processors attempts to reduce the number of remote requests for memory. When possible, a pixel is assigned to the node which contains the object visible at that pixel. Ray coherence is exploited by allocating blocks of pixels, thus further reducing the remote requests necessary. The code was designed for a 3-D grid, but given the 2-D grid of the PPDS, the static allocation of pixels is shown in Figure 20.

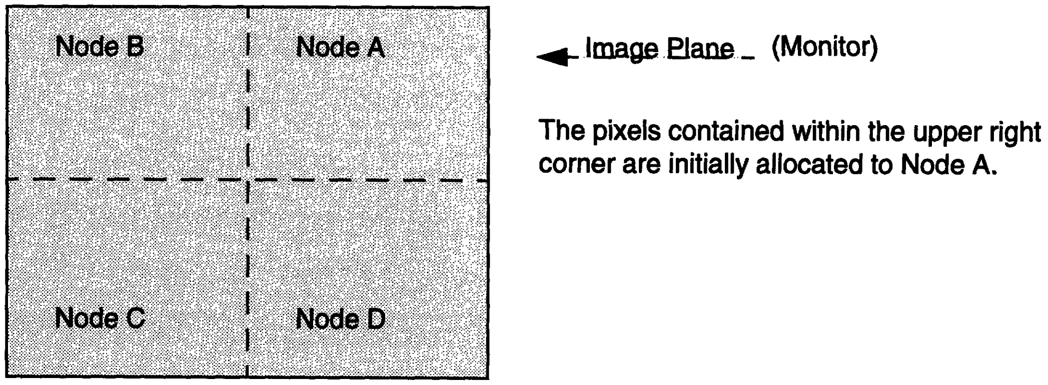

Node B
Node A
Node C
Node D
Image Plane (Monitor)
The pixels contained within the upper right corner are initially allocated to Node A.

Figure 20: Static Allocation of Pixels

- Dynamic Load balancing (First Frame)

Dynamic allocation is performed when a node has completed the processing of the pixels that were statically allocated to it. For example, Node A in Figure 21 has finished its block of pixels. Node A requests work from the root. The root realizes that Node A has finished processing its statically assigned pixels, and determines if a neighbor of Node A has not finished processing of the pixels statically assigned to it. Node B has not completed processing of its pixels. Thus, the root sends Node A a portion of the pixels previously assigned to Node B.

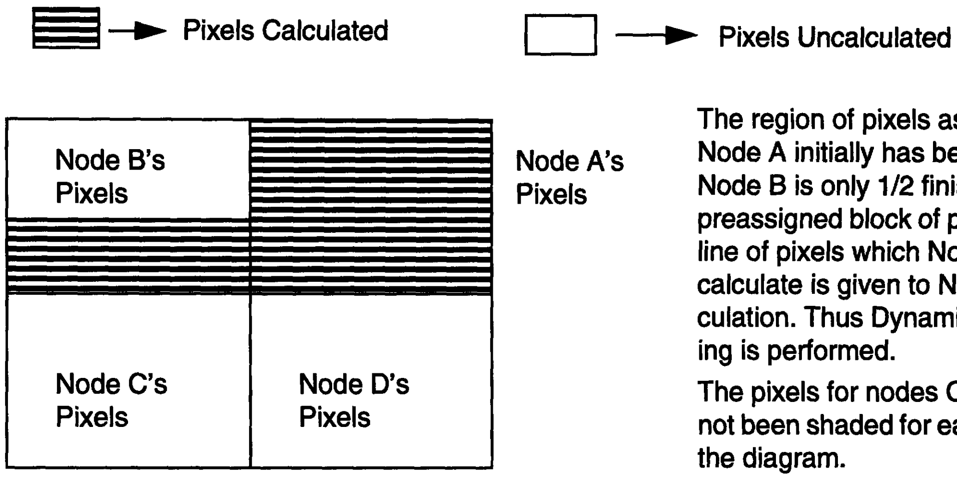

Pixels Calculated
Pixels Uncalculated
Node B's Pixels
Node A's Pixels
Node C's Pixels
Node D's Pixels
The region of pixels assigned to Node A initially has been completed. Node B is only 1/2 finished with its preassigned block of pixels. The next line of pixels which Node B was to calculate is given to Node A for calculation. Thus Dynamic Load Balancing is performed.
The pixels for nodes C and D have not been shaded for easier reading of the diagram.

Figure 21: Dynamic Allocation of Pixels

4.4.4 Root Processor Responsibilities

The examples in 4.4.2 and 4.4.3 (and in following sections) assume that 4 processors are available for handling of the algorithm. However, as mentioned in Section 4.4.1 on page 41, the root processor does not actually handle any of the ray tracing. Therefore only 3 processors are available for implementation of the algorithm. Explanations throughout this paper proceed assuming 4 child processors. This approach leads to a better understanding of the underlying algorithms.

The responsibilities of the root are distributed between the remaining 3 processors. Thus objects which would have been assigned to the root are sent to the nearest child, and pixels allocated for processing by the root will be dynamically allocated to the child nodes.

4.5 Animation

The next step of parallelization was to animate the scene.

- Movement of Objects

The emphasis of this project was not aimed at the realistic animation of real world phenomena, such as bouncing balls or walking figures. All that was required was the creation of animated sequences for testing of the motion picture coherence routines. Thus, a simple method of random movement of objects was created. The code shown in Figure 22 moves objects stored in the object buffer.

```
update_frame(void) { float data; int i, local, value;
    local = 0;
    value = (int) frame[local++];
while(value != 'E')     {  for(i=1;i<4;i++)
    {random = (float) rand()/((float) RAND_MAX) + random/
((float) RAND_MAX);
random = (float) M_PI*(2.0)*random;
    data = sin(random);
    data = frame[local] + (SPEED*data); /* modifying frame */
    frame[local++] = data;}
value = (int) frame[local++];
    } return;}
```

Figure 22: Translation of Objects

The preceding code belongs on the root. It is executed by the root at the beginning of each frame. The object buffer, *frame[ ]*, contains the identification number of the object moved, and the new x, y, and z locations of the center of the object. The new x, y, and z locations are determined by random movement, as displayed in the above code.

After updating the physical locations of objects, the root determines which nodes should receive these objects. The root maintains information about which nodes contain which objects. It uses this data to pass updated objects to the appropriate nodes.

- Updating Network

When the nodes have received this new object data, they undergo an intricate method of updating each other using semaphores (flags). The use of flags to stall execution of the processor results in a way to synchronize events. Proper synchronization eliminates the possibility for corruption of data. For example, if a node were to start the ray tracing of a scene before it received the new position of an object within its memory, it would produce the wrong image.

4.5.1 Object Partitioning During Animation

As mentioned in Section 4.4.2 on page 41, objects move between nodes as they traverse space. Objects may also move outside the bounds of the current scene, thus expanding the physical space occupied by the objects. It is necessary to expand the physical space allocated to the nodes as a result. In addition, the entire scene may have motion in a certain direction. Therefore the location of the physical space assigned to a node may expand or move.

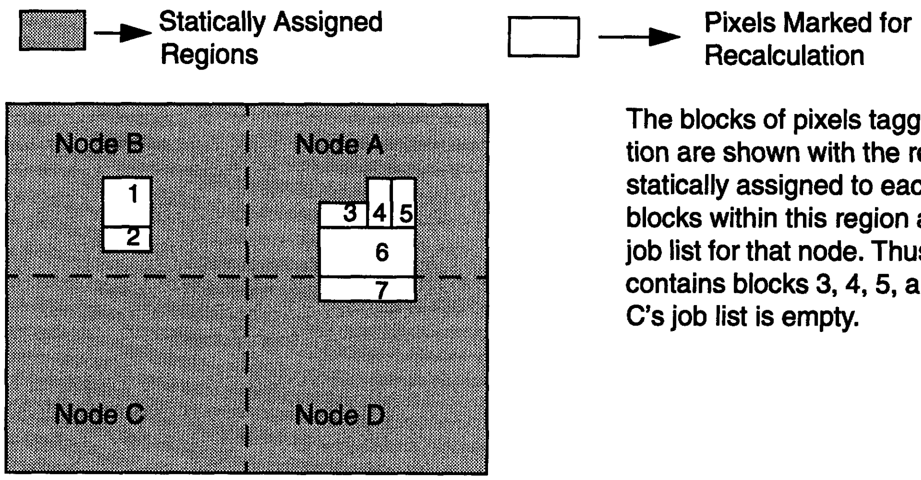
The algorithm implemented allows the physical space occupied by objects to expand. In addition, the entire database may move in any direction. The algorithm simply extends the bounds of the physical space controlled by a node if one of its objects moves into a region which is uncharted. This method is simple, but leads to uneven distribution of the database. A proposed solution was to reassign regions of physical spaces to nodes after a specified number of frames. A better solution is to reassign the database when the distribution of objects becomes detrimental to performance of the system[7].

4.5.2 Task Allocation During Animation

Section 4.4.3 on page 43 described the allocation of pixels for the initial frame. As explained in Chapter 5, the following frames will require only a subset of the image pixels to be calculated. Therefore, allocation of pixels to nodes after the initial frame is a modified version of the algorithm in Section 4.4.3.

7. Neither approach was implemented in this project due to time constraints.

Blocks of pixels are tagged for recalculation as explained in Section 5.2.2 on page 52. These blocks are placed in a job list for the nodes which were initially assigned to process them as shown in Figure 23. Thus a node may have an empty job list if no pixels in its statically assigned region are tagged for recalculation, i.e. node C.



The blocks of pixels tagged for recalculation are shown with the region of pixels statically assigned to each node. The blocks within this region are placed on a job list for that node. Thus Node A's job list contains blocks 3, 4, 5, and 6. While Node C's job list is empty.

Figure 23: Allocation of Pixels during Animation

Nodes begin processing of the scene by requesting pixels to ray trace from the root. The root sends each requesting node a block of pixels to calculate. If the job list for a requesting node is empty, the root will allocate pixels belonging to a neighbor.

4.6 Collection and Display of Images

The display of images is accomplished through the use of distinct code on the nodes, the root, and the PC. First, the nodes ray trace a block of pixels and store the color information. When the block is completed, it is sent to the root which passes it to the PC. The root never stores the complete image, frame buffer, within its memory. Its only purpose is to communicate the color information to the PC. The format of the block transfer is shown in Figure 24.

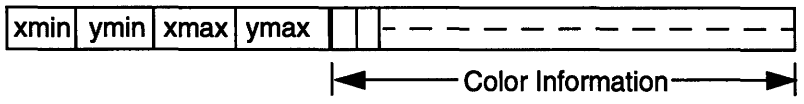


Figure 24: Color Transfer Format The length of the color info is $(x_{max} - x_{min}) \times (y_{max} - y_{min})$

The PC receives this block of information from the root and uses the x-y minimum and maximum pixel values to display the color information appropriately.

Chapter 5

Motion Picture Coherence

Motion Picture Coherence exploits image space temporal coherence. Temporal coherence exists in image space and object space. Methods have been developed utilizing both object and image space temporal coherence. In this chapter, previous methods in both image and object space are discussed. This is followed by the presentation of a new image/object space temporal coherence algorithm for ray tracing whose goal is to eliminate excessive pixel processing during animation.

5.1 Previous Work in Temporal Coherence

Image space temporal coherence as explained in Chapter 1 selects a subset of the complete image to recompute every frame. Object space temporal coherence uses the frame to frame relationships between objects in the environment to reduce computation of the rendering algorithm, although each pixel in the image may require some calculation.

5.1.1 Image Space Algorithms

- Sub-Sampling Algorithm -- Badt, 1988 [3]

Badt presented a method which uses a conventional ray tracer to compute the "base frame", first frame. Following frames are rendered by first randomly selecting an evenly spaced subset of the pixels. These pixels are then ray traced and if the color found for that pixel differs from the previous frame, a 3-D flooding algorithm floods that region in x-y image space and time with rays, thus determining the proper color for that pixel and surrounding pixels. Since this method requires flooding in time, every frame of the animation sequence must be present in memory. An alternative method proposed by Badt does not flood in time therefore does not require that the entire animation sequence be stored in memory. However, a larger subset of pixels must be recalculated. Badt did not actually implement these algorithms.

- Binary Frame Search -- Chapman & Calvert, 1992 [7]

Chapman and Calvert proposed a method similar to that of Badt [3] in that it compares a previous frame's pixel values to those of the current frame to determine which pixels require recalculation. This is accomplished through complete rendering of every Kth frame. A pixel whose value is identical in frames n and n+K will remain unchanged for all frames between n and n+K. If a pixel changes in value between frames n and n+K, the pixel is ray traced at frame n+(K/2). If it differs in color at frame n+(K/2), then it must be ray traced in frames n+(K/4) and n+ (3/4)K. In effect, a binary search for the frame in which each pixel changed color is performed. Once again, this method requires the entire animation sequence be present in memory. Since it requires information about future scenes, it is inappropriate for interactive animation.

For an animation with 95% frame-to-frame coherence and a value of K=5, a speedup over traditional ray tracing of about 4 was obtained.

5.1.2 Object Space Algorithms

- I-Net Method -- Fussell & Buckalew, 1990 [6]

This method does not use ray tracing. In fact, it is a modified radiosity approach which utilizes temporal coherence. A complete illumination network is built initially. Objects which interact through energy transfers are placed on this illumination network. Objects previously illuminating each other will with high probability illuminate each other during following frames. Thus the illumination network built initially will basically remain unmodified, exhibiting object space temporal coherence. New temporary connections may be made as objects move. These connections are also built into the illumination network initially, and given a temporal parameter. Only the network links which have a temporal parameter corresponding to the current frame are active.

The actual illumination of the scene must be determined each frame. The Illumination net simply allows a quick determination of which objects interact for the illumination algorithm. A few problems exist with this method. First, it requires redundant storage of objects since a moving object appears unique in each frame, due to its temporal parameter. The algorithm requires complete knowledge of the animation sequence. As a result, it is not suited for interactive environments, and it consumes enormous amounts of memory.

With the illumination nets in use, a speedup of almost 4 was achieved for each frame of a 4 second animation. The illumination network itself required 6M of memory.

- Voxel Approach -- Jevans, 1992 [13]

This is the alternative algorithm discussed in Chapter 1. A network of voxels is produced, and only primary rays passing through voxels containing moved objects are ray traced again. Unlike the previous methods mentioned above, future frames are not required for rendering of the current frame. A speedup of 4 over a traditional ray tracing algorithm was achieved for an animation sequence consisting of 361 frames and 6000 polygons.

5.2 Temporal Coherence Implementation

5.2.1 Uniqueness of Algorithm

As mentioned in Chapter 1, the proposed algorithm may be used for interactive animation purposes. No previous knowledge of future frames is required for calculation of the current frame. The algorithm is a hybrid model of the object space temporal coherence algorithms discussed above. For example, it exploits object space temporal coherence through the creation of an illumination net similar to that of Fussell and Buckalew [6]. However, it uses these nets to recompute only a portion of the image, similar to Jevans [13].

The algorithm transforms the object space locations which have been modified into image space coordinates. Thus the algorithm exploits image space temporal coherence by avoiding recalculation of pixels that remain the same. And it relies on object space temporal coherence for utility of the illumination and shadow networks.

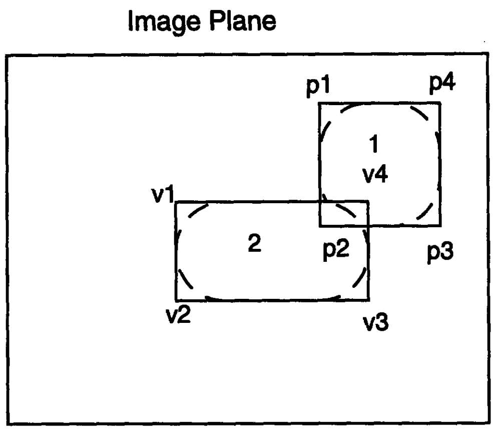
In addition, this researcher believes that this is the first project to attempt a temporal coherence ray tracing algorithm on a multiprocessing system.

5.2.2 Implementation

After animation of objects was properly working on the PPDS, it was possible to implement and test the proposed motion picture coherence algorithms. This section outlines the tasks these routines perform.

- Determination of Pixels to Recalculate

The pixels to be ray traced for the current frame must be calculated. The bounding box of every object which has been moved during the current frame is projected onto the image plane (Figure 25). In addition, the object in its previous location is projected onto the image plane. These projection areas correspond to the blocks of pixels which must be recalculated.



The projection of the objects is shown as dotted lines. The projection of the bounding box of that object is the solid line surrounding that object.

The vertices of these boxes are translated into pixel coordinates and thus a block of pixels which must be calculated is obtained.

The previous location of the objects are not shown for a more readable diagram.

Figure 25: Projection onto Image Plane

Each node goes through the list of objects in its memory which have moved during the current frame, and applies this projection routine to them. As shown in Figure 25, projections may overlap. This may cause the same pixel to be tagged for recalculation multiple times. A system of tests was implemented which eliminated this possibility. For example, in Figure 25, if the following relationships hold, then the bounding boxes are known to overlap:

$$v4.X \geq p2.X \qquad v4.Y \geq p2.Y \qquad v4.X \leq p4.X \qquad v4.Y \leq p4.Y$$

The vertices are defined in an x-y coord. system, thus v4.X implies the x location of the vertex.

In addition to the pixels directly related to moving objects, it may be necessary to recalculate other pixels. For example, an object may be shadowed in one frame and illuminated in the next (refer to Figure 4 and Figure 5). It is required that the portion of the image plane displaying this object be recalculated, otherwise the object will continue to appear shadowed. Thus all objects, which the algorithm determines may have changed in intensity or color, are applied to the projection routine, and the pixels corresponding to the projection are tagged for recalculation.

- Updating of Shadow and Illumination Networks (Nets)

Each object in the scene maintains a list of objects that it either shadows or illuminates. If a moved object is no longer shadowing or illuminating another object, the shadow and illumination lists must be updated. Figures 4 and 5 illustrate this method.

- Updating Nets in Remote Nodes

If an object resident in permanent memory of a remote node has illuminated or shadowed an object in local permanent memory, the local node must signal to the remote node this connection. This process occurs during the shading algorithm. For example, if a ray reflected from an object in remote permanent memory illuminates an object in local permanent memory above a threshold value, a connection exists. The local node must notify the remote node of this relationship. The remote node will then update the object's illumination list.

In Section 5.2.2 under *Determination of Pixels to Recalculate,* it was mentioned that each node uses the objects within its permanent memory, and their corresponding net lists, to determine which pixels to recalculate. As a result, objects which are present in local memory, but are resident in permanent memory on a remote node, need not have updated nets. In fact, no net is associated with these duplicated objects.

5.2.3 Summary

A parallel ray traced algorithm exploiting temporal coherence in both object and image space has been created. The previous chapters have outlined the histories of topics covered in this research, and presented the algorithms used for realization of the previously stated goal: production of ray traced images in real time on a parallel network. The degree to which this goal was met is explained in the final chapter, *Results and Conclusions.*

Chapter 6

Results and Conclusions

6.1 Performance

Unless otherwise specified, the results presented in Sections 6.1.1, 6.1.2, and 6.1.3 were obtained with a scene composed of 20 objects[8]. The image size was 640 by 480 pixels, with 8 bits per pixel[9]. The number of frames was limited to 10. Thus the animation described in Section 4.5 on page 44 is applied 9 times, creating ten frames. The frame-to-frame coherence is easily found by dividing the number of pixels which are not tagged for recomputation by the total number of pixels in the image.

6.1.1 Performance of Algorithm vs. Complete Ray Tracing of Scene

These graphs show the times required for the complete ray tracing of individual frames in a ten-frame animation sequence. The times achieved using motion picture coherence are superimposed on the same graphs. Figure 26 depicts the times for movement of objects limited to 1 unit per (speed 1) frame. Figure 27

8. All times on graphs are given in seconds.
9. The 8 bit RGB components were translated to a grey scale 8 bit value. A full 32 bit word was still transferred to the PC, with 24 of the bits zeros. Packing the bytes of color information should be performed for more efficiency.

illustrates the results of increasing the speed of moving objects to 5 units per frame (speed 5). The random motion generator described in Section 4.5 is in fact pseudo random. It generates the same sequence of random motions (for as long as the code runs). Since the directions of motion each time the algorithm runs are identical, it is possible to simply vary the speed of the movement in these directions to measure performance vs. speed.

Slower moving objects tend to improve the performance for the motion picture coherence method, since the number of new pixels required for recalculation is not as great[10].For example, a stationary sphere does not require any pixels to be recalculated, while a sphere of diameter 2 moving at 2 units per frame requires complete recalculation of its previous location, plus complete recalculation of its current location.

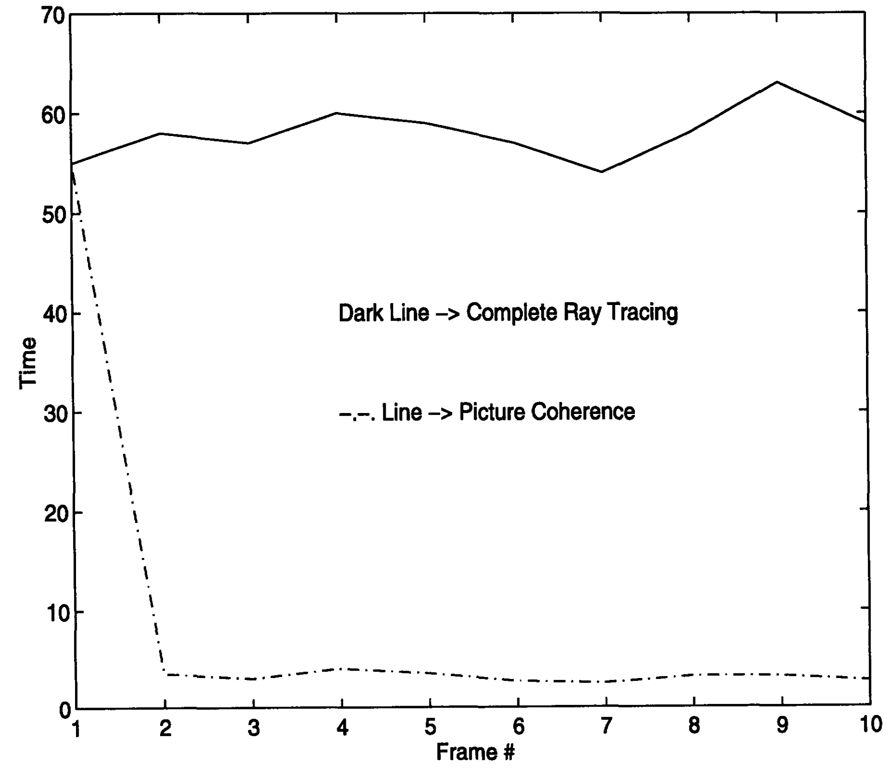


Figure 26: Picture Coherence vs. Standard Ray Tracing (Speed 1)

10. This can be seen by projecting the object at both its current and previous locations onto the image plane as explained in Section 5.2.2 on page 52

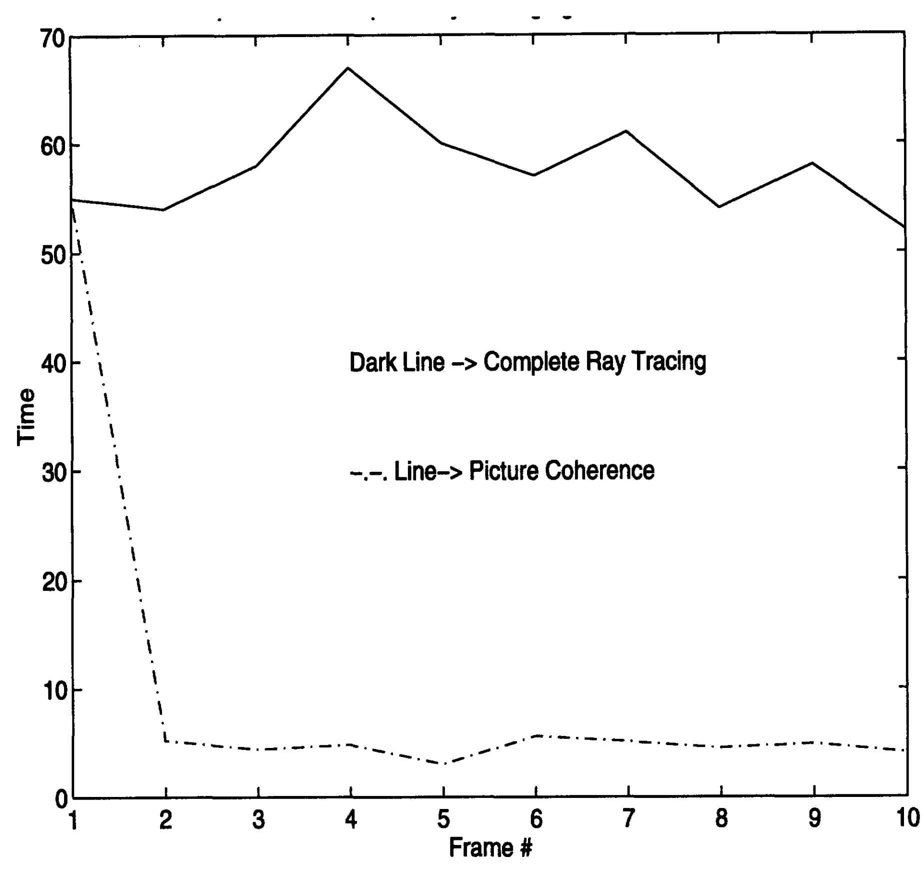


Figure 27: Picture Coherence vs. Standard Ray Tracing (Speed 5)

6.1.2 Scalability

Ideally, increasing the number of processors should linearly decrease the time for processing an algorithm. P processors should finish a task that took one processor Q seconds to complete in time Q/P. However, as the number of processors in a network is increased, interprocessor communication increases as well. Thus the computation time approaches an asymptotic value where it is no longer beneficial to increase the number of processors in the network[11]. The graph presented

11. The number of processors available was not enough to fully demonstrate this phenomenon.

below, Figure 28, was obtained from the same 10 frame sequence run at an animation speed of 1 unit per frame on one, two, and three processors[12]. With three processors, the parallel efficiency was 92%.

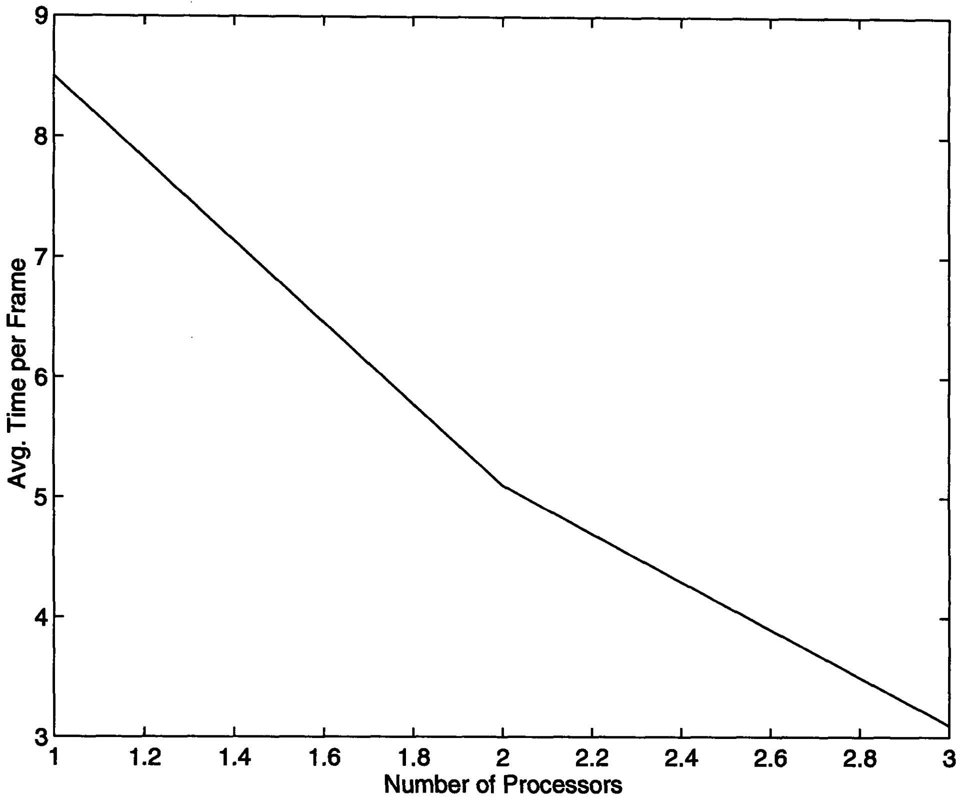


Figure 28: Algorithmic Performance vs. Network Size

The performance of a parallel network depends on the communications overhead as well as the parallelized algorithm running on the nodes. The time a parallel network will spend processing can be generalized by Equation 4,

Equation 4: Parallel Performance

$$Q = T/P + C$$

12. The time for calculation of the first frame was not included in the average times shown in Figure 28.

where Q is the total time to run the algorithm on the parallel network, T is the time the process would have taken if run on a single processor, P is the number of processors in the parallel network, and C is the time spent in communication. From the data collected, Q, T, and P are known. A model for variable C must be built to examine the scalability of the algorithm.

The communication overhead of this system is related to the transfer of memory, illumination and shadow network information, pixel color data, requests for work, and commands from the root to nodes. As the network size and the storage required for the scene becomes large, the transfer of memory will dominate communications. A model for communications is presented in Equation 5.

The following assumptions were made:

- Spatial coherence is exploited such that each node requests memory from a remote node at most once per frame.
- The cache hit rate is 0% for each node upon every initial request for remote memory.
- The cache hit rate is 100% for each node after a remote memory has been received during the current frame.

An estimation on the maximum number of memory transfers occurring each frame is obtainable from these three assumptions. If each node requests memory from every remote node, the number of requests per frame is P(P-1). The average size of each of these transfers is assumed to be the number of objects in the scene, Z, divided by P. The time required for these transfers is proportional to the average size of the transfer, thus the (Z/P)xP(P-1) term in Equation 5. The frame-to-frame coherence portion of the equation models the fact that a higher frame-to-frame coherence implies less communication. Finally, a scaling variable, S, is applied to model other factors left unattended.

Equation 5: Theoretical Performance

$$Q = T/P + S \times Z \times (P-1) \times (1 - FrameToFrameCoherence)$$

The variable S was empirically found[13]. This was possible due to the availability of all other variables in Equation 5. The expected value of S was 0.1209 with a scene size of 20 objects. The frame-to-frame coherence was set at 95%, and the value of T was 8.5 seconds. This resulted in the plot of Figure 29.

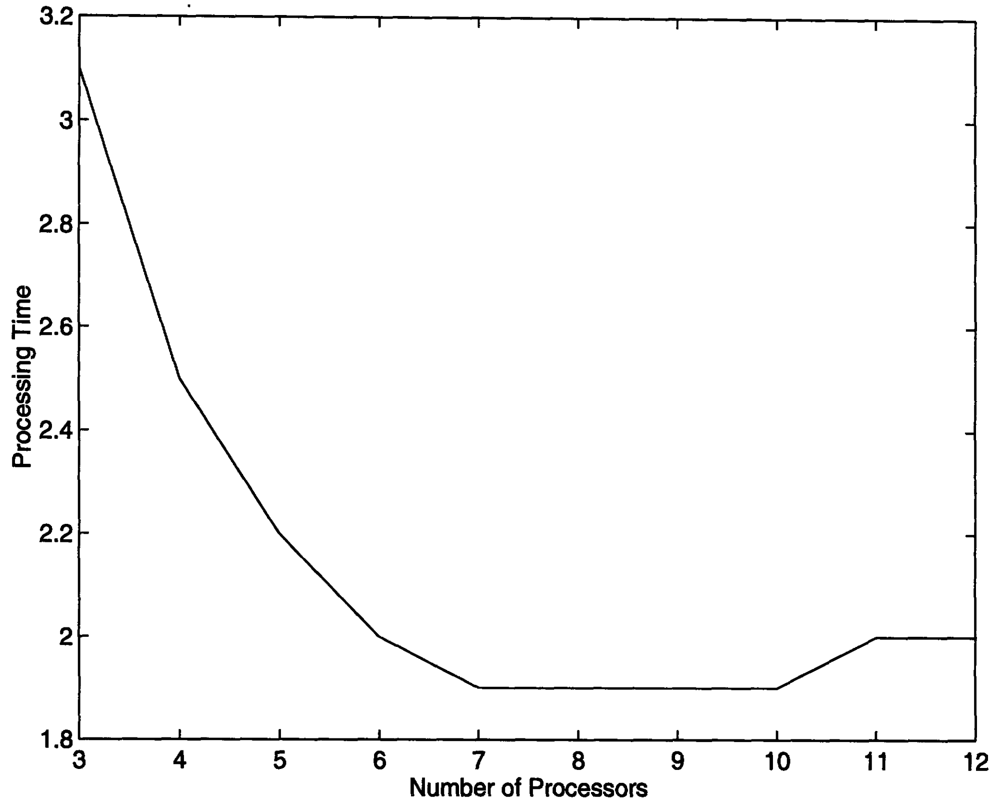


Figure 29: Theoretical Scalability

6.1.3 Performance vs. Frame-to-Frame Temporal Coherence

Frame-to-frame coherence is plotted against processing time in Figure 30. The processing time does not linearly improve with coherence. Since the ray tracing of individual pixels requires variable time, the computation of blocks of pixels of equal size must also require varying times. This implies that it is possible for smaller blocks of pixels to require more computation than larger blocks[14], as shown in

13. Through application of this equation 9 times with the data presented in this chapter, 9 values for S were found. The expected value of this variable was then taken, and used for the plot of Figure 29.

Figure 30. Frame-to-frame coherence for the 10 frame (speed 1) animation sequence used for the previous graphs is plotted here, along with the processing times for these frames.

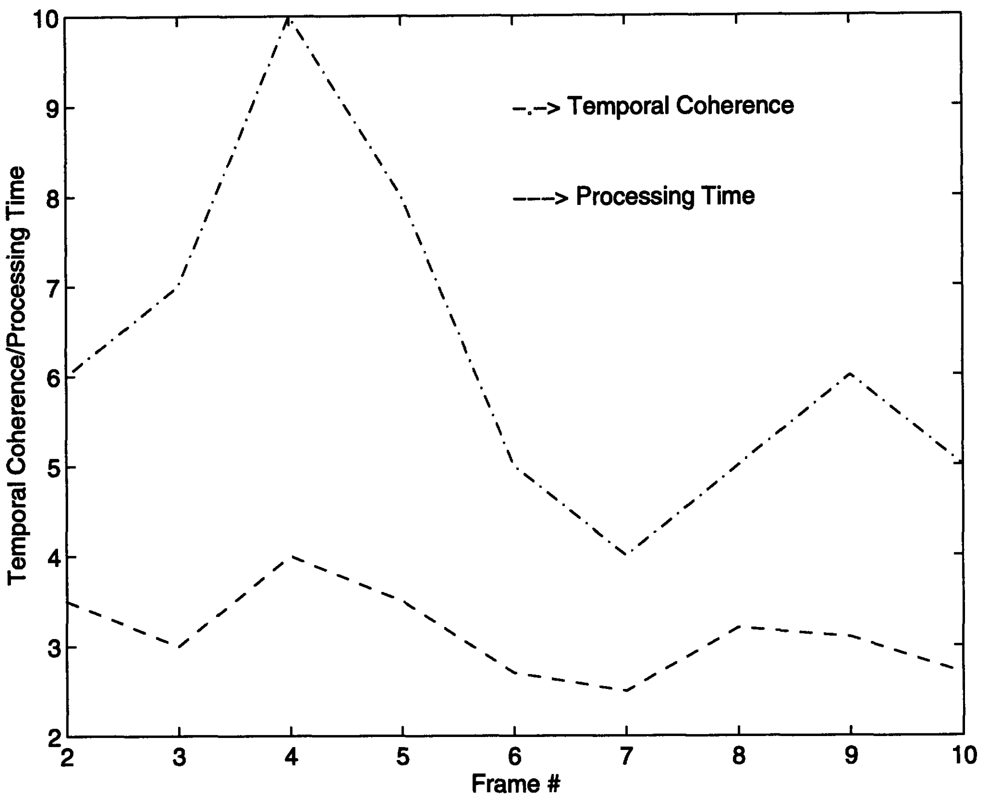


Figure 30: Algorithmic Performance vs. Temporal Coherence

6.1.4 Performance vs. Number of Moving Objects

The processing time of the algorithm increases as the number of objects in the scene increases (Figure 31). The increase in processing time with objects is not linear due to overlapping of objects as they move in the world. Overlapping of objects reduces the number of pixels marked for recalculation by the motion picture coherence routines.

14. Higher frame-to-frame coherence implies fewer pixels are tagged for recalculation. Temporal coherence as plotted on the vertical axis here is the opposite of frame-to-frame coherence, i.e. 6% temporal coherence implies 94% frame-to-frame coherence.

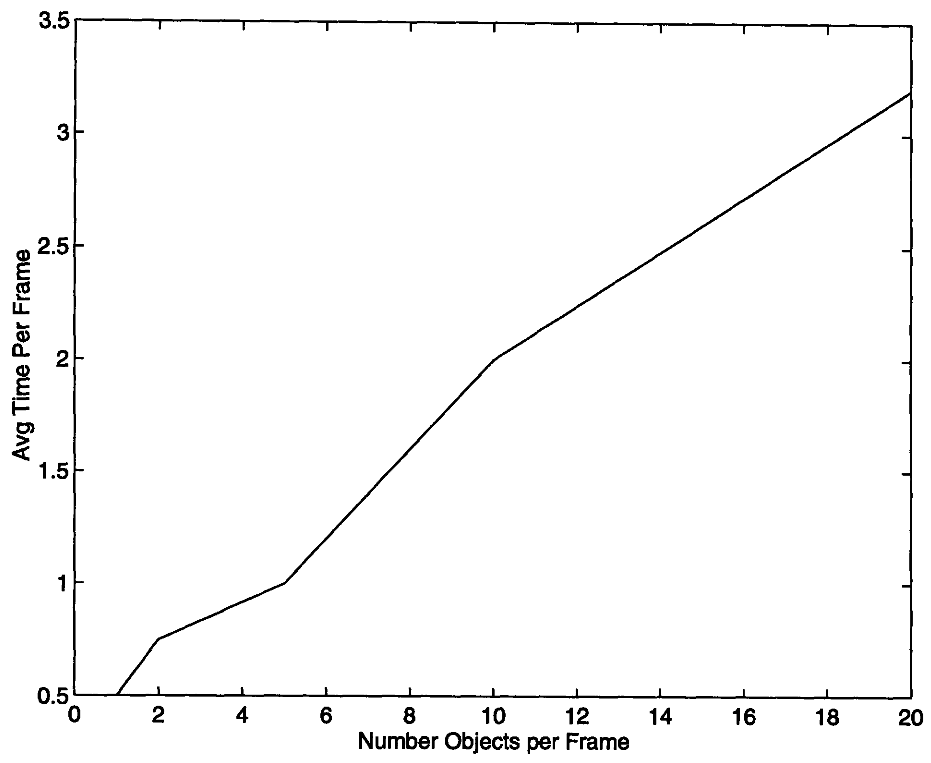


Figure 31: Performance as Number of Animated Objects Increases

6.2 Conclusions

6.2.1 Real Time

The data gathered on small scenes has proven that the proposed algorithm can reduce processing time. The scenes tested were composed of 20 objects. Each object was moving in a random direction at 1 unit per frame. For these scenes, the processing time averaged 3 seconds. Real time implies that a frame must be generated every 1/32 of a second. Thus, 3 second processing per frame is 92 times slower than real time.

6.2.2 Recommendations

To further enhance the performance of the algorithm, the code should be optimized. In general, optimization leads to a 50% reduction in processing time. This leaves a factor of 46 between real time and the observed processing time of these simple images.

Increasing the number of processors in the network would also decrease this time. A network of 27 C40 processors is available at TI. With minor modifications, the temporal coherence code would run on this network. Ideally, this would decrease the processing time observed on the PPDS by the ratio of processors available in the respective networks[15], 3/26. However, as the plot in Figure 29 demonstrates, the algorithm may not perform as desired for a larger network. Indeed, the theoretical analysis implies that no further improvements are achieved after 8 processors. The accuracy of this theoretical model should be tested and refined on larger networks.

Assuming no benefits are achieved by adding more than 7 processors, the algorithm would fall short of real time by a factor of 30 for the given animation[16]. It is possible that the remaining improvement in speed could be achieved through further advances in processor speed. These simple animation sequences could then be displayed in real time through ray tracing.

6.2.3 Extensions

As the number of moving objects increases, processing time increases. At some large number of moving objects, the results point to the algorithm exceeding the time required for ray tracing without picture coherence. Enough data was not collected to extrapolate the number of moving objects which would cause this to occur. Regardless of the number, the algorithm is beneficial at least for simple animations.

15. One processor for each network is dedicated for the root process.
16. This comes from 46*(2/3). The 2/3 ratio is the minimum theoretical time of Figure 29 divided by the avg. time found empirically.

6.2.4 Theoretical Accuracy

One of the limits to this approach is that the illumination net may grow very large for images with many reflective objects. Another concern is that the illumination net currently only stores objects which are directly illuminating each other. Indirect illumination may also change the color of objects and thus require recalculation of those indirectly illuminated areas also, see Figure 32. For theoretical accuracy, objects indirectly illuminating each other should also be added to the illumination networks. However, at some point, the overhead involved in updating these extended networks will eliminate any benefit of the proposed algorithm.

The algorithm therefore will perform satisfactory for images without conspicuous indirect illumination. For the images tested, no adverse visual effects were noticeable due to this problem.

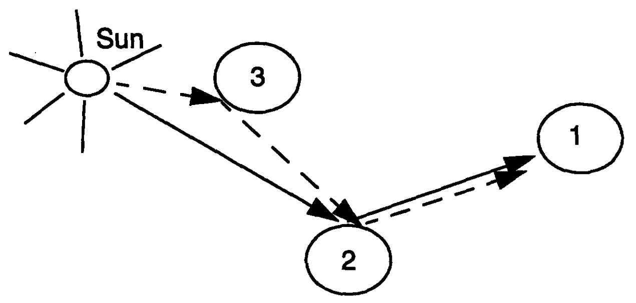


Figure 32: Indirect Illumination Problem

$\dashrightarrow$ Indirect Illumination. Object 3 indirectly illuminates object 1.

$\longrightarrow$ Direct Illumination. Object 2 directly illuminates object 1.

Problem: If object 3 moves, it could effect the shading of object 1. Currently, indirectly illuminating objects are not part of the illumination nets. Thus object 1 remains unchanged

6.2.5 Conclusion

Although real time was not achieved, the algorithm reduced the computation required in animations tested in the parallel network by a factor of 20. If implemented in a larger system, processing times for images would further decrease and make this a powerful tool for animators.

Glossary of Terms

Bounding Volume	A simple object which tightly surrounds a more complex object. The overall cost of intersecting rays with the complex object is reduced. If the ray fails to intersect the simple bounding volume, it will not intersect the contained object.
Deadlock	A situation, described below, in which each node of a network is unable to continue computation of the algorithm. Nodes of a parallel network may begin a process which requires completion before each processor may proceed. However, each node is unable to complete the process due to the unavailability of some resource. As a result, every processor in the network becomes stalled, as it waits for the resource to become free.
Distributed Memory	A parallel network with nodes featuring independent banks of memory is known as a distributed memory system.
Hypercube	A binary hypercube with 2^D nodes has dimension D. Each node is assigned a binary address with network links existing between those nodes which differ by only one digit in their binary address.
Image Space	The coordinate system of the image plane.
Load Balancing	The attempt to balance the computational load of processors in a parallel network with the goal to minimize idle time of processors.
LRU	A method of updating a cache which replaces the least recently used object in the cache when a new entry is required.
Motion Picture Coherence	The image space temporal coherence exhibited by successive frames of an animation.
Object Space	The world coordinate system of the scene.
Object Partitioning	The distribution of objects to nodes in a network in an attempt to balance the computation of each node in the network, and to increase the overall scene size capability of the network.
Parallel Efficiency	The ratio of parallel speedup to number of processors in the network.

Parallel Speedup	The ratio of sequential run time to parallel run time of an algorithm.
Ray Coherence	Rays similar in direction and point of origin are likely to follow similar paths, thus most likely intersecting common objects. A bundle of rays can be traced concurrently, exploiting ray coherence.
Ray Partitioning	The allocation of primary rays to nodes of a parallel network performing ray tracing to balance the load.
Rendering	The creation of computer images through physical models of the environment to be displayed.
Shared Memory	A parallel network with a single bank of memory accessible by every node of the network is known as a shared memory system.
Semaphores	A method of synchronizing processes which relies on two operations, wait and signal. These operations allow processes to begin at specific times
Speedup	The ratio of the time in which an algorithm previously took for completion to the time in which the improved algorithm took for completion.
Temporal Coherence	Similarities between successive frames of an animation sequence, either in image space or object space, which can be used to reduce computation of rendering algorithms.
Transputer	This INMOS microprocessor introduced in 1985 was a revolution in microchips in that it supported hardware multitasking as well as hardware communications between these processes and other processors. Thus it was suitable for parallel networks.
Virtual Memory	Virtual memory as used in a parallel system implies the assignment of portions of the database to the memory of each node. Each node contains this initial assignment of the database, along with memory it reserves for a cache. This cache is used to store portions of memory resident on remote nodes, when that memory is desired. In this way, the combined memories of the nodes mimics a single memory unit of a size potentially much larger than that resident on individual nodes.

Voxel	Spatial partitioning method which subdivides space into a regular grid of cells, each occupying equal volumes of space. These cells are known as volume elements, or voxels.
Z-Buffer	A technique which stores the distance to the closest object displayed at each pixel. This allows objects which are further than the distance stored by the z-buffer to have no contribution to the pixels color, while objects closer will update the pixel's color, and the distance stored in the z-buffer for that pixel. This is a visible surface determination algorithm.

REFERENCES

[1] Appel, A. "Some Techniques for Machine Rendering of Solids", AFIPS Conference Proc., 32, 1968. pp 37-45.

[2] Badouel, D. and T. Priol. "An Efficient Parallel Ray Tracing Scheme for Highly Parallel Architectures", Adv. in Comp. Graphics Hardware V. Rendering Ray Tracing and Visualization Systems, 1992. pp 93-106.

[3] Badt, S. "Two algorithms for taking advantage of temporal coherence in ray tracing", The Visual Computer. Vol. 4, no 3, 1988. pp 123-132.

[4] Boothe, B. "Multiprocessor Strategies for Ray-Tracing". Master's Project, University of California, Berkeley, Computer Science Division, Berkeley, CA, September 1989.

[5] Bouatouch, K. and T. Priol. "Static Load Balancing for a Parallel Ray Tracing on a MIMD Hypercube", The Visual Computer. Vol. 5, no 1-2, 1989. pp 109-119.

[6] Buckalew, C. and D. Fussell. "An Energy-Balance Method for Animation", Dept. of Computer Sciences, Univ. of Texas at Austin, April 1990.

[7] Calvert, T.W. and J. Chapman. "Exploiting Temporal Coherence in Ray Tracing", Proc Computer Graphics Interface '92. pp 196-204.

[8] Caspary, E. "Sequential and Parallel Algorithms for Ray Tracing Complex Scenes". Ph.D. Dissertation, University of California, Department of Electrical and Computer Engineering, Santa Barbara, CA, June 1988.

[9] Dippe, M. and J. Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis", SIGGRAPH '84. Vol. 18, no. 3. July 1984. pp. 149-158.

[10] Foley, J. and A. vanDam. Computer Graphics, Principles and Practices, Addison-Wesley Publishing Company 1990. pp 649-813.

[11] S.A. Green, E. Lewis and D.J. Paddon. "A Parallel Algorithm and Tree-Based Computer Architecture for Ray-Traced Computer Graphics", Parallel Processing for Computer Vision and Display, Addison Wesley Publishing Company, 1989. pp 131-142.

[12] Jensen, D.W. and D.A. Reed. "A Performance Analysis Exemplar: Parallel Ray Tracing", Concurrency: Practice and Experience. Vol 4, no 2. April 1992. pp 119-141.

[13] Jevans, A. D. "Object Space Temporal Coherence for Ray Tracing", Proc Graphics Interface '92, pp. 176-183.

[14] Kaplan, M.R. " The Use of Spatial Coherence in Ray Tracing", Techniques for Computer Graphics. David Rogers and Rae Earnshaw, ed., Springer-Veriag, 1985. pp. 173-193.

[15] Kay, T.L. and J.T. Kajiya. "Ray Tracing Complex Scenes", Computer Graphics. Vol 14, no. 4. August 1986. pp. 269-278

[16] Matsumoto, H. and K. Murakami. "Ray-Tracing with Octree Data Structure". Proc. 28th Information Processing Conf., Tokyo, 1983. pp.1535-1536.

[17] H. Nishimura, H. Ohno, T. Kawata, I. Shirakawa and K. Omura. " A Parallel Pipelined Multimicrocomputer System for Image Creation", Proceedings of the 10th Symposium on Computer Architecture. Vol. 11. SIGARCH, 1983. pp. 387-394.

[18] Parallel Processing Applications with the TMS320C4X, Applications Guide. Texas Instruments 1994. Ch. 1.

[19] Phong, B. "Illumination for Computer-Generated Pictures", Communications of the ACM 18, June 1975.

[20] Rubin, S.M. and T. Whitted. "A Three-Dimensional Representation for Fast Rendering of Complex Scenes", Computer Graphics, Vol. 14, 1980. pp 110-116.

[21] TMS320C4X User's Guide. 2564090-9761 rev C, Texas Instruments, Inc., 1993. Ch. 1-4.

[22] Watt, A. and M. Watt. Advanced Animation and Rendering Techniques, Addison-Wesley Publishing Company 1992. pp

[23] Whitted, T. "An Improved Illumination Model for Shaded Display", Communications of the ACM 23, June 1980. pp 343-349.

Appendix A (Shading)
Parameters from section 3.1.1 defined

1)

$$I$$

This is the total Illuminosity of the point. This is usually included with a subscript λ which signifies a function dependence on wavelength of the incident light.

2)

$$I_a$$

This is the ambient light component. Ambient light is produced by the cumulative effects of other objects in the world that are dispersing light in various directions.

3)

$$I_p$$

If light sources are modeled as points, then this parameter is the intensity of point lights in the environment. The shading equation must be applied for every light source in the environment.

4)

$$k_d$$

This is the material's diffuse reflection coefficient. Thus this models how much light is reflected by the object and how much is absorbed.

5)

$$\bar{N} \cdot \bar{L}$$

N is the normal at the point of intersection and L is the direction of the light source. NdotL gives the cosine of the angle between the two vectors. If NdotL is less than zero, then the light source is at an angle greater than 90 degrees from the normal, and thus no incident light can fall on the object.

6)

$$(\bar{R} \cdot \bar{V})^n$$

R is the direction the light source is reflected. V is the viewing direction. RdotV gives the cosine of the angle between the two vectors, which implies that smaller angles result in greater specular components. The exponent n varies the effect of the RdotV specular component.

Basic Ray Tracing Math

(Intersection of a light ray with a sphere)

The easiest object to intersect a ray with mathematically is the sphere. A ray is defined as a parameterized vector:

$$x = x_0 + t \cdot (x_1 - x_0) \qquad y = y_1 + t \cdot (y_1 - y_0) \qquad z = z_0 + t \cdot (z_1 - z_0)$$

where (x0, y0, z0) represent the origin of the ray and (x1, y1,z1) determine the direction. The equation of a sphere centered at (a, b,c) and with radius r is:

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$$

The intersection of the ray with the sphere is found by substituting the x, y, and z values of the parameterized vector into the equation for the sphere. The resulting equation is a quadratic in t, and thus the time, t, when the ray will hit the sphere is known. This time, t, can then be plugged into the parametric description of the ray to determine at what location this occurred.

Appendix B (Code)

Asmall portion of the code for this project is included here. The entire code listing would have taken over 200 pages. Thus only a few of the sections that were discussed in this paper are included. The following is a list of the functions inlcuded for the root and node code[1]. None of the PC side code was inlcuded. In addition, most of the ray tracing code was not included here[2].

Node Functions

- main(), pg 96
- dnext_pix(), pg 104
- next_scene(), pg 107
- message_init(), pg 98
- update_all(), pg 109
- update_netlist(), pg 110
- new_frame(), pg 125
- update_sphere(), pg 109
- remove_obj(), pg 110
- receiving_moved_net(), pg 111
- create_moved_net(), pg 112
- addtolist(), pg 113
- send_to_remote_net(), pg 114
- add_to_local_net(), pg 114
- add_to_shill(), pg 115
- delete_from_shill(), pg 116
- update_netlist(), pg 110
- nearest_neighbors(), pg 121
- still_contained(), pg 123
- common(), pg 124
- cpuint(), pg 118
- c_int08()->c_int28(), pg 118
- Raytrace(), pg 99
- Trace_a_ray(), pg 102

Root Functions

- main(), pg 90
- search_oid(), pg 75
- appends(), pg 75
- start_motion(), pg 77
- create(), pg 76
- box_send(), pg 77
- pixel(), pg 78
- pixalloc2(), pg 78
- add_to_no, pg 80de_list()
- motion(), pg 74
- pixalloc(), pg 93
- dpixalloc(), pg 94
- pixhandler(), pg 94
- update_frame, pg 76
- dpixalloc2(), pg 79
- cpuint(), pg 89
- c_int08()->c_int28()

1. The root node is commonly referred to as the HostC40 throughout the code.
2. The ray tracing code was ported from MTV's ray tracer.

Root Code

```
 /***
Move.c          7/17/94
***/
/*This file contains the code which passes the object id's
along with the new 'coordinate' information. In the future it
is possible that other parameters could be sent fairly easily
by simply expanding the TYPES of messages....but not now.
Simply need a method of transferring new scene information
right now */

/*'frame_num'==global  frame  count.  Used  for  debugging
purposes only. Actually, right now, only good for spheres and
other objects with a definite center        While nodes are
calculating pixel colors, root is determining which node will
get what objects next frame...The important thing here is not
how we bring in data, but how the object id is used to
determine what node to send this data to*/

extern float          frame[];
extern float       pbuff[];
```

void   motion(void)

```
{
intx, n, flag;
int data;
for(n = 0; n < P; n++)
{
if(n == my_node)
continue;
x = 0;
count = 0;
data = (int) frame[count];
      while(data != 'E')
/*->limited to 'E' nodes with this simple */
     {/* code */
flag = search_oid(n, data);
/* pass in object id to see in in node n*/
     if(flag != NULL)
             {
     appends(x, data)
/* More than one word of data per obj */
     x++;
     count = count +4;
     data = (int) frame[count];
     continue;}

     else
     {
     count = count + 4;/* store data 4 words at a time */
     }
     data = (int) frame[count];
     }
```

```
        store_size = x;
    /* Need to process across functions... */
        create(n, x);      /* put data in mail box */
    }
      update_frame(); /* Calc next frame ... */
    }
    /* Then new Frame time -- sends out array to remote nodes.
    Reason must check every object against every node space is
    cause very possible that object belongs in more than one node
    space.
    */
```

int search_oid(int node, int object_id)

```
    {
    HEAD    h, temp, prev;
    int     max, min, hit;
        if(nodes[node]->oid->hd->next == NULL)
        {
        return(0);    /* not a hit */
        }
             max = nodes[node]->oid->tail->offset;
             min = nodes[node]->oid->hd->offset;
             if(object_id < min)
             {
             hit = 0;
             return (hit);
             }
             if(object_id > max)
             {
             hit = 0;
             return (hit);
             }
             temp = nodes[node]->oid->hd;
             while(temp->offset < object_id)
             {
             temp = temp->next;
             }
             if(temp->offset == object_id)
             {
             hit = 1;
             return (hit);
             }
             else
             return(0);

    }
    /* could provide with previous link So that would accomplish
    a quicker  search cause could start from the end  at times.
    However, this would require  chains in the reverse direction
    which  adds to the memory and is not worth it. At least at
    this  time  it  doesn't  seem  worth  the  extra  work,memory
    */
```

appends(int array_off, int id)

```
    {
    int local = count+1; /* offset into current place in frame
    buffer */
    float data;
    int i;
```

```
if((storage[array_off]=
float*) calloc(4, sizeof(float)))==NULL)
    while(1);

store_count++;/* number of mallocs to store */
    storage[array_off][0] = (float) id;
    data = frame[local++];
    for(i=1;i<4;i++)
    {
    storage[array_off][i] = data;
    data = frame[local++];
    }
}
```

update_frame(void)

```
    {
    float data;
    int i, local, value;
    local = 0;

    value = (int) frame[local++];
    while(value != 'E')
    {
    for(i=1;i<4;i++)
    {
     random = (float) rand()/((float) RAND_MAX) + random/
((float) RAND_MAX);
   random = (float) M_PI*(2.0)*random;
   data = sin(random);
  data=frame[local] + (SPEED*data);/*modifying frame*/
    frame[local++] = data;
    }
    value = (int) frame[local++];
    }
   return;
}
```

create(int node_id, int size)

```
{
int     i;
if((mail_box[node_id]->mail=(float     **)     calloc(size+1,
sizeof(float *)))== NULL)
   while(1);

mailbox_count++;      /* # of calls to mail_box count */
    for(i=0;i<size;i++)
    {
    mail_box[node_id]->mail[i] = (float *) storage[i];
    }
if((mail_box[node_id]->mail[size] = (float*) calloc(1,
sizeof(float))) == NULL)
   while(1);
oneword++;      /* # of calls to mail_box count */

    mail_box[node_id]->mail[size][0] = NULL;
/* last word in box */
}
```

start_motion(void)

```
/* Sends out the new frame to nodes */
{
int i;
int x;
        for(i=0;i<P;i++)
                {
                if(i != my_node)
                {
                box_send(i);
/* send the mail box info of that node */
                }
                }

/* Now deallocate all storage */
   i = 0;
   while(i < store_size)
   {
   free((float *) storage[i]);  /* Deallocating 'storage' */
   i++;
   }
   for(i=0;i<P;i++)
   {
   if(i != my_node)
   {
   x=0;
   if(i != P-1){
   while(mail_box[i]->mail[x][0] != NULL)
        {
   free((float *) mail_box[i]->mail[x]);
   x++;
   }
   free((float *) mail_box[i]->mail[x]);/*freeing last one*/
   }
   else{ free((float *) mail_box[i]->mail[store_size]); }
   free((float **) mail_box[i]->mail);
   }
   }
}

/*  box_send();  */
/* With this, have to limit the size of new frame information
to the size of the input buffer, else reserve specified memory
just for this info on the nodes...*/
```

box_send(int i)

```
{
float *data = pbuff;
int size, q, z, x;
int dat;
   size = 4;
        x = 0;
        q = 0;
        dat =(int) mail_box[i]->mail[q][0];
        while(dat != NULL)
        {
        z=0;
        while(z < size)
        {
```

```
        data[x] = mail_box[i]->mail[q][z++];
        x++;
        }
        q++;
        dat =(int) mail_box[i]->mail[q][0];
              /* NULL Signifies last info*/
        }

   data[x++] = NULL;/* last object in list */
        send(i, data, x, MOVEDATA);
}



/*** Pict.c          08/03/94          ***/
/*
This file contains the code which collects the pixels to be
calculated from the children and then puts them into a
dynamically created array which can be read from during
dynamic allocation of the next frames pixels...
*/

extern float    pbuff[];
```

void pixel(int port, int len, int snode)

```
{
        float *data = pbuff;
        int i = 0;
        VECTOR pmax, pmin;
        in_msgk_float(port, data, 1, len);
        while(i < len)
        {
        pmin.x = *data++;
        pmin.y = *data++;
        pmax.x = *data++;
        pmax.y = *data++;
        i = i+4;
        add_to_node_list(pmin, pmax, 0, snode, 0);
        }
       len = pixarray[snode];

    if(len == 0)
      { nodes[snode]->pixels = 0;  }
    else
{
nodes[snode]->pixels = 1;
if((pix_list[snode]->pixels[len]    =    (float*)    calloc(1,
sizeof(float)))==NULL)
while(1);
oneword++;/* number of one word mallocs */
pix_list[snode]->pixels[len][0] = view.x_res;   /* Flag that
last pix */
}
}
```

void pixalloc2(void)

```
{
    int i, x, offset;
    int diffy, diffx;
    int minx, miny, maxx, maxy;
    float msg[5];
        for(i=0;i<P;i++)
        {
     pixarray[i] = 0;/* reset offset for allocation routines*/
        }
for(i=0;i<P;i++)
{
if(i != my_node)
{
                  if(nodes[i]->pixels != 0)
                  {
                  msg[0] =  (float)i;/*BOUNDS PASSED TO PROC*/
                   offset = pixarray[i];
                   minx =  pix_list[i]->pixels[offset][0];
                   miny =  pix_list[i]->pixels[offset][1];
                   maxx =  pix_list[i]->pixels[offset][2];
                   diffx = (maxx - minx) + 1;
                   diffy = MAX_PIXS/diffx;
                   /* Assuming rounds down here*/
                   maxy  = diffy + miny - 1;
                   if(maxy >= pix_list[i]->pixels[offset][3])
                   {
                   maxy = pix_list[i]->pixels[offset][3];
                   offset++;
                   pixarray[i] = offset;    /* next block */
                    }
                    else{
                   pix_list[i]->pixels[offset][1] = maxy+1;
                  pixarray[i] = offset;   /* stays the same */
                        }
                  msg[1] =  minx;
                  msg[2] =  miny;
                  msg[3] =  maxx;
                  msg[4] =  maxy;

                  send(i , msg, 5, PACK);

    if(pix_list[i]->pixels[offset][0]== view.x_res)
                   {
                  nodes[i]->pixels=0;
                  /*Dealloacte this array*/
                  x = 0;
                   while(x <= 1)
                   {
                   free((float *) pix_list[i]->pixels[x]);
                   x++;
                   }
                   pixarray[i] = 0;     /* reset offset to 0 */
                   }
                   }
                   else
                   {
                   pixhandler(i);
                   }
} } }
```

void dpixalloc2(int snode, int rnode)

```
{
int nymin;
float msg[5];
int maxy, maxx, miny, minx, i;
int diffy, diffx, offset;

offset = pixarray[snode];
i = 0;
/* BOUNDS PASSED TOPROC 'snode' */
msg[0] =  (float) snode;
minx   =  pix_list[snode]->pixels[offset][0];
miny   =  pix_list[snode]->pixels[offset][1];
maxx   =  pix_list[snode]->pixels[offset][2];
diffx = (maxx - minx) + 1;
diffy = MAX_PIXS/diffx;      /* Assuming rounds down here*/
maxy  = diffy + miny - 1;

if(maxy >= pix_list[snode]->pixels[offset][3])
{
   maxy = pix_list[snode]->pixels[offset][3];
   pixarray[snode] = offset+1;
   offset++;
   }
else
   {
   pix_list[snode]->pixels[offset][1] = maxy+1;
    }
   msg[1] =  minx;
   msg[2] =  miny;
   msg[3] =  maxx;
   msg[4] =  maxy;
   send(rnode,msg, 5, PACK);
   if(pix_list[snode]->pixels[offset][0] == view.x_res)
               {
               nodes[snode]->pixels = 0;
               /* Dealloacte this array */
               i = 0;
               while(i <= offset)
               {
               free((float *) pix_list[snode]->pixels[i]);
               i++;
               }
   pixarray[snode] = 0;     /* reset offset to 0 */
   return;
   }
}


/* Add_to_node_list() checks to make sure overlapping blocks
of pixels aren't tagged for recalculation */
```

```
void  add_to_node_list(VECTOR  pmin,  VECTOR
   pmax, int q, int snode, int tnode)
 {
   VECTOR tmax, tmin, tempmin, tempmax;
   int    len, x, pixlength, flag;
       int    tminx, tminy, tmaxx, tmaxy;
       int    pmaxx, pmaxy, pminx, pminy;
       len = pixarray[snode];
```

```
     pmaxx = (int) pmax.x;
     pmaxy = (int) pmax.y;
     pminy = (int) pmin.y;
     pminx = (int) pmin.x;
     flag = 0;
for(x=tnode;x<P;x++)
{
if((x != my_node)&&(x != snode)&&(nodes[x]->pixels != 0))
{
pixlength = pixarray[x];
while(q < pixlength)
{
tminx =  (int) pix_list[x]->pixels[q][0];
tminy =  (int) pix_list[x]->pixels[q][1];
tmaxx =  (int) pix_list[x]->pixels[q][2];
tmaxy =  (int) pix_list[x]->pixels[q++][3];
/*            Case 1       */
if((pmaxx >= tminx)&&(pmaxx <= tmaxx)&&(pminy<=tmaxy)
   &&(pminy>=tminy)&&(pminx <= tminx)&&(pmaxy >=tmaxy))
{
     /* First make sure actually different size */
    if(pminx < tminx)
    {
    tempmin.x = (float) pminx;   /* First new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) tminx - 1.0;/*fixing boundaries */
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if((pmaxy > tmaxy)&&(pmaxx > tminx))
    {
    tempmin.x = (float) tminx;   /* Second new block */
    tempmin.y = (float) tmaxy+1.0;    /* fixing bounds */
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*            Case 2       */
if((pmaxx <= tmaxx)&&(pminx >= tminx)&&(pminy>=tminy)
   &&(pminy<=tmaxy)&&(pmaxy >= tmaxy))
{
    if(pmaxy > tmaxy)
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) tmaxy+1.0;/* Fixing bouns */
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    return;
    }
    }
/*            Case 3       */
if((pmaxx >= tmaxx)&&(pmaxy >= tmaxy)&&(pminy>=tminy)
   &&(pminy<=tmaxy)&&(pminx >= tminx)&&(pminx <=tmaxx))
```

```
{
    if(pmaxy > tmaxy)
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) tmaxy+1.0;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if((pmaxx > tmaxx)&&(pminy < tmaxy))
    {
    tempmin.x = (float) tmaxx+1.0;/* Second new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*            Case 4        */
if((pmaxx >= tmaxx)&&(pminx >= tminx)&&(pminx<=tmaxx)
   &&(pminy>=tminy)&&(pmaxy <= tmaxy))
{
    if(pmaxx > tmaxx)
    {
    tempmin.x = (float) tmaxx+1.0;   /* First new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    return;
    }
    }
/*            Case 5        */
if((pmaxx >= tmaxx)&&(pminy <= tminy)&&(pminx>=tminx)
   &&(pminx<=tmaxx)&&(pmaxy >= tminy)&&(pmaxy <=tmaxy))
{
    if((pmaxx > tmaxx)&&(pminx < tmaxx))
    {
    tempmin.x = (float) tmaxx+1.0;   /* First new block */
    tempmin.y = (float) tminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(pminy < tminy)
    {
    tempmin.x = (float) pminx;    /* Second new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*            Case 6        */
if((pmaxy >= tminy)&&(pmaxy <= tmaxy)&&
     (pminy<=tminy)&&(pminx >= tminx)&&(pmaxx <=tmaxx))
{
    if(pminy < tminy)
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    return;
    }
    }
/*            Case 7        */
if((pminy <= tminy)&&(pminx <= tminx)&&(pmaxx>=tminx)
   &&(pmaxx<=tmaxx)&&(pmaxy >= tminy)&&(pmaxy <=tmaxy))
{
    if((pminx < tminx)&&(pmaxy > tminy))
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) tminy;
    tempmax.x = (float) tminx-1.0;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(pminy < tminy)
    {
    tempmin.x = (float) pminx;    /* Second new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*            Case 8        */
if((pmaxx >= tminx)&&(pmaxx <= tmaxx)&&(pmaxy<=tmaxy)
   &&(pminy>=tminy)&&(pminx <= tminx))
{
    if(pminx < tminx)
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) tminx-1.0;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    return;
    }
    }
/*            Case 9        */
if((pmaxx <= tmaxx)&&(pmaxy <= tmaxy)&&(pminx>=tminx)
   &&(pminy>=tminy))
{
```

```
              return;
}
/*Case 10 */
if((pmaxx >= tmaxx)&&(pmaxy >= tmaxy)&&(pminx<=tminx)
   &&(pminy<=tminy))
{
    /* replace previous values with encompassig bbox */
     if(pminx < tminx)
     {
    tempmin.x = (float)pminx;    /* First new block */
    tempmin.y = (float)pminy;
    tempmax.x = (float)tminx-1.0;
    tempmax.y = (float)pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
     if(pmaxy > tmaxy)
     {
    tempmin.x = (float) tminx;    /* Second new block */
    tempmin.y = (float) tmaxy-1.0;
    tempmax.x = (float) tmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
     if(pmaxx > tmaxx)
     {
    tempmin.x = (float) tmaxx+1.0;   /* Third new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
     if(pminy < tminy)
     {
    tempmin.x = (float) tminx;    /* Fourth new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) tmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
     if(flag == 1)
     { return; }
}
/*           Case 11 */
if((pmaxx >= tminx)&&(pminx <= tminx)&&(pmaxy>=tmaxy)
   &&(pminy<=tminy)&&(pmaxx <= tmaxx))
{

   if((pmaxy > tmaxy)&&((pmaxx > tminx)||(pminx < tminx)))
    {
    tempmin.x = (float) pminx;    /* First new block */
    tempmin.y = (float) tmaxy+1.0;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(pminx < tminx)
    {
    tempmin.x = (float) pminx;   /* Second new block */
    tempmin.y = (float) tminy;
    tempmax.x = (float) tminx-1.0;
    tempmax.y = (float) tmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
   if((pminy < tminy)&&((pmaxx > tminx)||(pminx < tminx)))
    {
    tempmin.x = (float) pminx;   /* Second new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*          Case 12 */
if((pmaxx <= tmaxx)&&(pminx >= tminx)&&(pmaxy>=tmaxy)
   &&(pminy<=tminy))
{
    if(pmaxy > tmaxy )
    {
    tempmin.x = (float) pminx;   /* First new block */
    tempmin.y = (float) tmaxy+1.0;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(pminy < tminy)
    {
    tempmin.x = (float) pminx; /* Second new block */
    tempmin.y = (float) pminy;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) tminy-1.0;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
    }
    if(flag == 1)
    { return; }
    }
/*          Case 13 */
if((pmaxx >= tmaxx)&&(pminx >= tminx)&&(pmaxy>=tmaxy)
   &&(pminx<=tmaxx)&&(pminy <= tminy))
{
   if((pmaxy > tmaxy)&&((pmaxx > tmaxx)||(pminx < tmaxx)))
    {
    tempmin.x = (float) pminx;   /* First new block */
    tempmin.y = (float) tmaxy+1.0;
    tempmax.x = (float) pmaxx;
    tempmax.y = (float) pmaxy;
    add_to_node_list(tempmin, tempmax, q, snode, x);
    flag = 1;
```

```
        }
        if(pmaxx > tmaxx)
        {
        tempmin.x = (float) tmaxx+1.0;    /* Second new block
*/
        tempmin.y = (float) tminy;
        tempmax.x = (float) pmaxx;
        tempmax.y = (float) tmaxy;
        add_to_node_list(tempmin, tempmax, q, snode, x);
        flag = 1;
        }
       if((pminy < tminy)&&((pmaxx >tmaxx)||(pminx < tmaxx)))
        {
        tempmin.x = (float) pminx;    /* Second new block */
        tempmin.y = (float) pminy;
        tempmax.x = (float) pmaxx;
        tempmax.y = (float) tminy-1.0;
        add_to_node_list(tempmin, tempmax, q, snode, x);
        flag = 1;
        }

        if(flag == 1)
        { return; }
        }
    /*           Case 14 */
    if((pmaxx >= tmaxx)&&(pminx <= tminx)&&(pmaxy<=tmaxy)
       &&(pmaxy>=tminy)&&(pminy <= tminy))
    {
        if((pminx < tminx)&&(pmaxy > tminy))
        {
        tempmin.x = (float) pminx;    /* First new block */
        tempmin.y = (float) tminy;
        tempmax.x = (float) tminx+1.0;
        tempmax.y = (float) pmaxy;
        add_to_node_list(tempmin, tempmax, q, snode, x);
        flag = 1;
        }
        if(pminy < tminy)
        {
        tempmin.x = (float) pminx;    /* Second new block */
        tempmin.y = (float) pminy;
        tempmax.x = (float) pmaxx;
        tempmax.y = (float) tminy-1.0;
        add_to_node_list(tempmin, tempmax, q, snode, x);
        flag = 1;
        }
        if((pmaxx > tmaxx)&&(pmaxy > tminy))
        {
        tempmin.x = (float) tmaxx+1.0;    /* Second new block
*/
        tempmin.y = (float) tminy;
        tempmax.x = (float) pmaxx;
        tempmax.y = (float) pmaxy;
        add_to_node_list(tempmin, tempmax, q, snode, x);
        flag = 1;
        }
        if(flag == 1)
        { return; }
        }
```

```
   /*           Case 15 */
   if((pmaxx >= tmaxx)&&(pminx <= tminx)&&(pminy>=tminy)
      &&(pmaxy<=tmaxy))
   {
       if(pminx < tminx)
       {
       tempmin.x = (float) pminx;   /* First new block */
       tempmin.y = (float) pminy;
       tempmax.x = (float) tminx-1.0;
       tempmax.y = (float) pmaxy;
       add_to_node_list(tempmin, tempmax, q, snode, x);
       flag = 1;
       }
       if(pmaxx > tmaxx)
       {
       tempmin.x = (float) tmaxx+1.0;    /* Second new block
*/
       tempmin.y = (float) pminy;
       tempmax.x = (float) pmaxx;
       tempmax.y = (float) pmaxy;
       add_to_node_list(tempmin, tempmax, q, snode, x);
       flag = 1;
       }
       if(flag == 1)
       { return; }
       }
       /*        Case 16 */
   if((pmaxx >= tmaxx)&&(pminx <= tminx)&&(pmaxy>=tmaxy)
      &&(pminy>=tminy)&&(pminy <= tmaxy))
   {
       if((pminx < tminx)&&(pminy < tmaxy))
       {
       tempmin.x = (float) pminx;   /* First new block */
       tempmin.y = (float) pminy;
       tempmax.x = (float) tminx-1.0;
       tempmax.y = (float) tmaxy;
       add_to_node_list(tempmin, tempmax, q, snode, x);
       flag = 1;
       }
       if(pmaxy > tmaxy)
       {
       tempmin.x = (float) pminx;   /* Second new block */
       tempmin.y = (float) tmaxy+1.0;
       tempmax.x = (float) pmaxx;
       tempmax.y = (float) pmaxy;
       add_to_node_list(tempmin, tempmax, q, snode, x);
       flag = 1;
       }
       if((pmaxx > tmaxx)&&(pminy < tminy))
       {
       tempmin.x = (float) tmaxx+1.0;    /* Second new block
*/
       tempmin.y = (float) pminy;
       tempmax.x = (float) pmaxx;
       tempmax.y = (float) tmaxy;
       add_to_node_list(tempmin, tempmax, q, snode, x);
       flag = 1;
       }
       if(flag == 1)
```

```
        { return; }
        }
      /* Do same check for any previous pix in array */
       }
       }
       q = 0;/* --> checking next list in array */
       }
/*If make it to the end -> no intersections, then add the box*/
        if(len == 0)   /* if first time, --> now have pixels */
           {
           nodes[snode]->pixels = 1;
           }
pix_frame = pix_frame + (pmaxx+1-pminx)*(pmaxy+1-pminy);
if((pix_list[snode]->pixels[len]  =  (float  *)  calloc(4,
sizeof(float)))==NULL)
while(1);

pixlist_count++; /* # of mallocs for pixel_list */
pix_list[snode]->pixels[len][0] = (float) pminx;
pix_list[snode]->pixels[len][1] = (float) pminy;
pix_list[snode]->pixels[len][2] = (float) pmaxx;
pix_list[snode]->pixels[len++][3] = (float) pmaxy;
pixarray[snode] = len;
}


/*
 * File Hinstall.c
 *
 * Functions c_int08, c_int12, ...and cpuint(int port)
 * The c_intnn functions are the interrupt sevice routines
 * which call function cpuint(). Function cpuint() is passed
 * the port of the incoming data which it then uses to obtain
 * the incoming data. The incoming message is handled by
 * branching to a subroutine dependent on the message received
 *
 * 1/19/94                    RP
 */
/*
 *Hinstall.c    --- ISR install for the host processor
 */

/*********************************************
 SETTING UP INTERRUPT SEVICE ROUTINES FOR EACH COMMMPORT
*********************************************/

void c_int08(){cpuint(0);}/*int 0E*/

void c_int12() {  cpuint(1); }

void c_int16() {  cpuint(2); }
/*SHOULD SET THIS UP SO CALLS COMMAND KERNEL*/

void c_int20() {  cpuint(3); }  /* 1A  */

void c_int24() {  cpuint(4); }  /* 1E  */
void c_int28() {  cpuint(5); }  /* 22  */

void cpuint(int port)
   {
   int type, dnode, snode, len;

   /* here follows all the routing stuff necessary to do this
   correctly */
      dnode = in_word(port);
      if(dnode != my_node)
      {
      forward(dnode, port);/* if not yours, forward message */
      }

      else
      {
      type  = in_word(port);  /* Get incoming info */
      snode = in_word(port);
      len   = in_word(port);


   switch (type) {

   case GO           : ready++;   /*Just incrementing number of */
                      break;      /* Processors ready  */
   case IMAGE        : store_image(len, snode, port);
                     /* storing image*/
                     break;
   case RQ           : pixhandler(snode);
                     break;
   case MOVEDATA      : update(snode, port, len, MOVEDATA);
                     /* moved objects */
                     break;
   case UPDATE       : update(snode, port, len, UPDATE);
                     break;

   case PIXLIST    : pixel(port, len, snode);
                           break;
   case PARAM      : tempor(port, len);
                          break;
   case BBOXS      : minmax(port, len);
                          break;
   case START      : starter = 1;  /* command from PC to begin */
                          break;
   case 'F'        : c_parse(len, port, 'F');
                           break;
   case  'A'       : c_parse(len, port, 'A');
                           break;
   case  'U'       : c_parse(len, port, 'U');
                           break;
   case  'G'       : c_parse(len, port, 'G');
                           break;
   case  'r'       : c_parse(len, port, 'r');
                           break;
   case  'L'       : c_parse(len, port, 'L');
                           break;
   case  'B'       : c_parse(len, port, 'B');
                           break;
case 's'        : c_parse(len, port, 's');
                        break;
case  'C'       : c_parse(len, port, 'C');
                       break;
case  'P'       : c_parse(len, port, 'P');
                       break;
case  'S'       : c_parse(len, port, 'S');
                       break;
case  'H'       : c_parse(len, port, 'H');
                     break;
case  'R'       : c_parse(len, port, 'R');
                       break;
case  'Q'       : c_parse(len, port, 'Q');
                       break;

default     : while(1); /* -->error in communications */
}
   }
}


/* The following main() is for the root */
```

void main(void)

```
{

   void                        *isr;
   int                         x;
   int                         index;
   int                         shift;
   int                         i, xres, yres, oport;
   x = 0;

   random = rand();

   /* global debug variables */
   oneword = 0;
   head_count = 0;
   mailbox_count = 0;
   store_count = 0;
   pixlist_count = 0;


   while(x < P){     /* For PPDS case */

   if(x != my_node)
   {
   start(x);     /* tell nodes to wait for data to trace */
   x++;
   }
   else{
   x++;
   }
   }

        for(x=0;x<P;x++)
        {
```

```
   pix_list[x]   = (PIXSTR *) malloc(sizeof(PIXSTR));
   pix_list[x]->pixels   =   (float   **)   calloc(MAX_BLOCKS,
sizeof(float *));
   pixarray[x]   = 0;          /* initializing count */
        }

/*Dynamically  creating  Nodes'  array  which  holds  pixel
information*/

   for(x=0;x < P; x++)
   {
   nodes[x] = (BOUNDS *) calloc(1, sizeof(BOUNDS));
   }
/* This is the info that is used to handle moving objects */
   hb = (HEADER *) malloc(sizeof(HEADER));

        for(x=0;x < P; x++)
        {
        nodes[x]->oid = (LIST *) malloc(sizeof(LIST));
        nodes[x]->oid->hd = (HEAD) malloc(sizeof(HD));
        nodes[x]->oid->tail = (HEAD) malloc(sizeof(HD));
        nodes[x]->oid->tail->offset = MAX_PRIMS;
        nodes[x]->oid->tail->next = NULL;
        nodes[x]->oid->hd->next = NULL;
        }
   storage = (float **)calloc(MAX_BLOCKS, sizeof(float *));
   for(x=0;x<P;x++)
   {
   mail_box[x]   = (MAIL *) malloc(sizeof(MAIL));
   }
/* Now installing interactive portion of the algorithm*/
                 set_ivtp(DEFAULT);

                 isr = (void *) c_int08;
                 install_int_vector(isr, 0X0E);

                 isr = (void *) c_int12;
                 install_int_vector(isr, 0X12);

                 isr = (void *) c_int16;
                 install_int_vector(isr, 0X16);

                 isr = (void *) c_int20;
                 install_int_vector(isr, 0X1A);

                 isr = (void *) c_int24;
                 install_int_vector(isr, 0X1E);

                 isr = (void *) c_int28;
                 install_int_vector(isr, 0X22);

asm(" LDI @_iieval, iie"); /* enabling all ICRDY inter */
asm(" LDI 2000h, ST");          /* GIE == 1 */

   while(starter == 0);   /* wait until all the info */
                          /* has been received from the PC */

   i = 0;
   while(i < P)
```

```
    {
      if(i != my_node)
       {
       oport = routing(i);
       out_word(i,     oport);  /* Destination ID */
       out_word(NFRAME,  oport); /* Type of message */
       out_word(my_node,  oport); /* Source ID */
       out_word(i,oport); /* Trash sent to finish comm */
       }
      i++;
    }

while(ready != P-1) /*Won't receive an increment from itself*/
    {
    NULL;
    }

    pixalloc(bk_side);

    pix_calc = 0;
    xres = view.x_res;
    yres = view.y_res;

    pix_frame = xres*yres;/* for first frame */
    animation = 1;          /* animation on */

    while(animation == 1)
         {
NF = 0;
motion();/* Begin determining what to send for next frame*/

    while(NF != 1)
/* Waiting until all pixels for prev frame done*/
    {
    i = 0;
    for(x=0;x<P;x++)
    {
    if(nodes[x]->pixels == 1)
    {
    i++;
    }
    }               /* Don't hard code this part */
    if((i == 0)&&(pix_calc >=pix_frame))
    {             NF = 1; }     /* no more pixels */

    }

pix_calc = 0; /* reset flag */
pix_frame=0;/*reset number of pixels to redo for next frame*/

start_motion();/*Start sending objects to appropriate nodes*/

ready = 0;
while(ready < P-1);
/*Waiting until nodes prepared for next frame*/

    frame_cnt++;      /* increment frame counter */
    pixalloc2();      /* Start allocation of pixels */
    }
```

```
   }

void pixalloc(int bk_side)
   {

   int nid;
   int x, tcols, c, r;
   int trows, ymax, x_pix, y_pix;
   int nymin;
   float msg[5];
   void *isr;

   int d2_p = netsize*netsize2;/* number of processors in the 2D-
   mesh*/
   int planes   = P/d2_p;    /* number of planes in network */

   tcols = planes*netsize2;
   trows = netsize;

   y_pix  =  view.y_res/trows; /* vertical pixels per node */
   x_pix  =  view.x_res/tcols;/* horiz. pixels per node */


   if(bk_side > y_pix)
   {
   exit(0);
   }
   c = 0;
   r = 0;

      while(c < trows)          /* offset into rows     */
      {
      while(r < tcols)        /* offset into columns */
      {
      planes =  (r/d2_p);    /* The plane node resides in */
      nid =(planes*d2_p) + (netsize*c + (r+netsize2)%netsize2);

      msg[0] =  (float) nid;
      msg[1] =  (float) c*x_pix;
      msg[2] =  (float) r*y_pix;
      msg[3] =  (float) c*x_pix + x_pix-1;
      msg[4] =  (float) r*y_pix + bk_side-1;

      /* The following handles the case when assinging to */
      /* processor P pixels */

      if(nid == (P-1))
      {
      nodes[nid]->p_max.x = view.x_res-1;
      nodes[nid]->p_max.y = view.y_res;
      }
      else
      {
      nodes[nid]->p_max.x = msg[3];  /* max x for node space */
      ymax = r*y_pix + y_pix;
      nodes[nid]->p_max.y = ymax  ;/* max y for node space */
      }
      nodes[nid]->p_min.x = msg[1];/* min x for node space */
      if(nid != my_node)
      {
      nymin = r*y_pix + bk_side;/* new miny for space   */
      nodes[nid]->p_min.y  = nymin;
      }
      else
      {
      nodes[nid]->p_min.y = r*y_pix; /* don't increment here */
       }
      nodes[nid]->pixels = 1; /* Flag that more pixels to do */

      if(nid != my_node)    /* don't let hostc40 trace anything*/
                   {
                   send(nid, msg, 5, PACK);
                   /*sending the packet*/
                    }
                    ++r;
                   }
      r = 0;
      ++c;
      }
   }
```

void dpixalloc(int snode, int rnode)

```
   {

   int nymin;
   floatmsg[5];
   int maxy;
   int diffy;

       msg[0] =  (float) snode;
      /* Bounds passed to Porcessor */
       msg[1] =  (float) nodes[snode]->p_min.x;
       msg[2] =  (float) nodes[snode]->p_min.y;
       msg[3] =  (float) nodes[snode]->p_max.x  ;
       maxy        =  nodes[snode]->p_max.y;

       diffy  =  maxy - msg[2];

       if(diffy <= bk_side)
       {
      nodes[snode]->pixels = 0;
      msg[4] = (float) maxy - 1.0;
       }
       else
       {
       msg[4] = (float) (msg[2] + bk_side-1);
       nodes[snode]->p_min.y = msg[4]+1;
       }
     send(rnode, msg, 5, PACK);  /* sending the packet*/
    }
```

void pixhandler(int snode)

```
      {

   int d2_p;     /* number of processors in the 2D-mesh   */
```

```
int myplane;  /* plane where node id belongs to        */
int dnodep,my_nodep, col, mycol, row, rows, nid, planes;
float trash[1];

if(nodes[snode]->pixels != 0)
    {
    if(frame_cnt == 0)
    {
    dpixalloc(snode, snode);
    }
    else{
    dpixalloc2(snode, snode);
        }
    }

    else
    {

    d2_p = netsize*netsize2; /* Proc. per plane */
    planes = (P/d2_p)- 1;    /* Watch this carefully*/
    myplane  = snode/d2_p ;
    my_nodep = snode%d2_p ;
    rows = my_nodep/netsize2;     /* done */
    mycol = my_nodep%netsize2;
    mycol = myplane*netsize2 + mycol;
    row = rows;
    col = mycol + 1;

    nid = (planes*d2_p) + (netsize2*col) +
((row+netsize2)%netsize2);

    if((col < netsize2) && (nodes[nid]->pixels != 0))
    {
    if(frame_cnt == 0)
    {
    dpixalloc(nid   , snode);
    }
    else{
    dpixalloc2(nid, snode);
        }
    }

    else{
    col = mycol - 1;
    nid = (planes*d2_p) + (netsize2*col) +
((row+netsize2)%netsize2);

    if((col >= 0) && (nodes[nid]->pixels != 0))
    {
    if(frame_cnt == 0)
    {
    dpixalloc(nid   , snode);
    }
    else{
    dpixalloc2(nid, snode);
        }
    }

    else{
```

```
    col = mycol;
    row = rows + 1;
    nid    =  (planes*d2_p) + (netsize2*col) +
((row+netsize2)%netsize2);

    if((row < netsize) && (nodes[nid]->pixels != 0))
    {
    if(frame_cnt == 0)
    {
    dpixalloc(nid  , snode);
    }
    else{
    dpixalloc2(nid, snode);
        }
    }

    else{
    row = rows - 1;
    nid    =  (planes*d2_p) + (netsize2*col) +
((row+netsize2)%netsize2);

    if((row >= 0) && (nodes[nid]->pixels != 0))
/* Order in which check */
     {
    if(frame_cnt == 0)
    {
    dpixalloc(nid  , snode);
    }
    else{
    dpixalloc2(nid, snode);
       }
/* Comparisons is important. If do the*/
/* nodes[nid] check first and the nid is*/
/* invalid, then you are gonna have a*/
/* problem */

else{
trash[0] = WAIT;

/*  send(snode, trash, 0, WAIT); */ /* Telling node to */
/* Wait for the next scene    */
/* Not necessary, node will wait anyway  */
    }
    }
    }
    }
    }

}
```

Node Code

main()

```
 {
 long          timest, timeend;
 int i;
 int *data;
```

```
double temp;

temp = MIN_T;
convf64f32(&min_t, temp);

hb = (HEADER *) malloc(sizeof(HEADER)); /* Header for send
routie*/

new_pixlist = (float **) calloc(MAX_BLOCKS, sizeof(float *));

    for(i=0;i<P;i++)
    {
    pows[i] = (TEMPLIST *) malloc(sizeof(TEMPLIST));
    pows[i]->hd =  (HEAD) malloc(sizeof(HD));
    pows[i]->tail = (HEAD) malloc(sizeof(HD));
    }

    /* initialization of global list for data to be retraced */
    global_list = (TEMPLIST *) malloc(sizeof(TEMPLIST));
    global_list->hd = (HEAD) malloc(sizeof(HD));
    global_list->tail = (HEAD) malloc(sizeof(HD));
    global_list->tail->id = 0;
    global_list->tail->next = NULL;
    global_list->hd->id = 0;
    global_list->hd->next = NULL;
          for(i=0; i < P; i++)
    {
    box[i] = (BX *) calloc(1, sizeof(BX));
/* Don't wanna have too many objs. else this calloc fails */
    }
    box[my_node]->mem = 1;/*--> Local Memory always present */
       /* actually set to 1 when receive NFRAME */
    memory[0] = my_node;
    memory[1] = 'E';

    for(i =0;i < P;i++)/* Should not be necessary, but this */
    { /* array is not initializing to zero */
    primobj[i] = 0; /* properly, thus do this for now.   */
    }

    for(i=0;i<4;i++)
    {
    illumcache[i] = 0;
    /*Should also be taken care of statically*/
    }

    for(i=0;i<4;i++)
    {
    shadcache[i]  = 0;
    }

    for(i=0;i< P; i++)
    {
    hitlist[i] = (HIT *) calloc(1, sizeof(HIT));
    /* Initializing here */
    }

color_table();  /* Sets up the color table */
message_init(); /* This routine will set up the commport
```

```
interreupts */
NF = 0;
goes = 0;

asm(" LDI @_iieval, iie");/* Enabling All ICRDY Interrupts */
asm(" LDI 2000h, ST");          /* GIE = 1*/

while(NF == 0); /* wait for last initial object from root*/

   for (i = 0; i < nlights; i++)
   {
   lights[i]->intensity = sqrt ((double) nlights)/ (double)
nlights;
   }
/* Preparing for ray tracing */
VecSub(view.look_at, view.from, view.look_at);
VecNormalize(&view.look_at);
VecNormalize(&view.up);
new_frame();

   animation = 1;
   NF = 0;

   while(animation == 1)
   {
   raytrace = 0;
   while(NF == 0)          /* NF == 0 -> same frame */
    {
   if(raytrace == 1)      /* stays here                    */
     {
   Raytrace(); /* Set raytrace == 0 in Raytrace()     */
     }                       /* and do other things here    */
   }

   for(i=0;i<4;i++)
        {
        illumcache[i] = 0;/* reset caches after each frame*/
        }

        for(i=0;i<4;i++)
        {
        shadcache[i]  = 0;
        }
      NF = 0;              /* Need here as flag             */
   cap = 1;
/*could lead to race conditions,although rare*/

frame_cnt++;/* now can inrement the frame counter */
net_updater = 0;/* global control flag */
next_scene();          /* Start creation of next scene      */
cap = 0; /* next_scene calls new_frame()    */
changes = 0; /* changes == flag used by next_scene */
}
while(1);             /* After finished with animation, loop*/
}
```

void message_init(void)

```
    {
```

```
    void *isr;
/* Now installing interactive portion of thealgorithm*/
                set_ivtp(DEFAULT);
                isr = (void *)c_int08;
                install_int_vector(isr, 0X0E);

                isr = (void *)c_int12;
                install_int_vector(isr, 0X12);

                isr = (void *)c_int16;
                install_int_vector(isr, 0X16);

                isr = (void *)c_int20;
                install_int_vector(isr, 0X1A);

                isr = (void *)c_int24;
                install_int_vector(isr, 0X1E);

                isr = (void *)c_int28;
                install_int_vector(isr, 0X22);
}
```

Raytrace()

```
{ /* Converting to 64 bit precision */
RAY ray;
float64 xr, yr, x_step, y_step, x_pw, y_pw, temp1, temp2;
float64 x_rand, y_rand, temp, xres, yres, view2angle;
int x, y, snode;
VECTOR2 hor, ver;
COLOR col, scol;
COLOR q;
VECTOR2 view2from, view2look, view2up;
int i,s, flag;
int y_start2, size;
INTERSECT       inter;
static  int   pbuff[BKSIZE];
float *data     = in_buff2;
i = 4;

raytrace = 0;           /* Global control variable */
    flag = 0;
    snode       = (int) *data;
    data++;
    x_start = (int) *data;
    pbuff[0] = x_start;
    data++;
    y_start = (int) *data;
    pbuff[1] = y_start;
    data++;
    x_end       = (int) *data;
    pbuff[2] = x_end;
    data++;
    y_end       = (int) *data;
    pbuff[3] = y_end;
    y_start2 = y_start;
    if(snode == my_node)
    {
    if(pobjects[snode] == 0)
    {
    return;/* Don't Raytrace if have no objects in memory */
    }
    }

    if(snode == HOSTC40)
    {
    snode = my_node;
    /* Currently assuming Host has no memory of scene */
    flag = 1;
    }/* But local nodes will have all of the scene present*/

         convf64f32(&view2up.x, view.up.x);
         convf64f32(&view2up.y, view.up.y);
         convf64f32(&view2up.z, view.up.z);

         convf64f32(&view2look.x, view.look_at.x);
         convf64f32(&view2look.y, view.look_at.y);
         convf64f32(&view2look.z, view.look_at.z);

         convf64f32(&view2from.x, view.from.x);
         convf64f32(&view2from.y, view.from.y);
         convf64f32(&view2from.z, view.from.z);

         convf64f32(&view2angle, view.angle);

         convf64i32(&xres, view.x_res);
         convf64i32(&yres, view.y_res);
/*calculate the viewing frustrum -- partly done in nframe.c*/
VECCROSS64(&hor, &view2up, &view2look);
VECNORM64(&hor, &temp);  /* horizontal screen vector */
VECCROSS64(&ver, &view2look, &hor);
VECNORM64(&ver, &temp);         /* vertical screen vector*/

/* The placing of this is suspicious */

divf32f64(&x_step, 2.0, &xres);
divf32f64(&y_step, 2.0, &yres);

VECCOPY64(&ray.pos, &view2from);
if(x_end == view.x_res)
    x_end = view.x_res - 1;
if(y_end == view.y_res)
    y_end = view.y_res - 1;

/* OK, start tracing */
    convf64i32(&temp, y_start);
    mpyf64(&yr, &y_step, &temp);
    subf32f64(&yr, 1.0, &yr);
    convf64i32(&temp, x_start);
    while(y_start  <= y_end)
    {

    mpyf64(&xr, &x_step, &temp);
    subf32f64(&xr, 1.0, &xr);

    for (x = x_start; x <= x_end; x++)
    {
    /*Setup the ray*/
    if (sample_cnt == 1)
    {
    mpyf64(&temp1, &xr, &view2angle);
    mpyf64(&temp2, &yr, &view2angle);

    VECCOMB64(&ray.dir, &temp1 , &hor, &temp2, &ver);
    VECADD64(&ray.dir, &view2look, &ray.dir);
    VECNORM64(&ray.dir, &temp1);
/* Trace that Ray!!*/

    col = Trace_a_ray(&ray, 0, snode, flag, inter);
    }
    else
    {
    /* Not bothering with this for now */

    col.r = col.g = col.b = 0.0;
    for (s = 1; s < sample_cnt; s++)
    {
        /*x_rand = (xr * view.angle) + (x_pw * RAND());
    y_rand = (yr * view.angle) + (y_pw * RAND());

    VecComb(x_rand, hor, y_rand, ver, ray.dir);
    VecAdd(ray.dir, view.look_at, ray.dir);
    VecNormalize(&ray.dir);        */
    /*
     * printf("ray.dir = (%lg %lg
     * %lg)\n", ray.dir.x, ray.dir.y,
     * ray.dir.z);
     */
    scol = Trace_a_ray(&ray, 0, snode, flag, inter);

    col.r += scol.r;
    col.g += scol.g;
    col.b += scol.b;
    }

    col.r /= sample_cnt;
    col.g /= sample_cnt;
    col.b /= sample_cnt;
    }

/*
 * Write pixel to output buffer
 */
     q = Write_pixel(&col);
       /* find the value associated with */
       /* the RGB color */

     pbuff[i++] = grey_scale(q);
        /* pbuff[i++] = color_sort(q); */

    subf64(&xr, &xr, &x_step);
    }
    subf64(&yr, &yr, &y_step);
    y_start = y_start+1;

    }
    size = (y_end+1-y_start2)*(x_end+1-x_start)+4;
```

```
        INT_DISABLE();  /* Don't wanna lose data here */
        send(HOSTC40, pbuff, size, IMAGE);
        INT_ENABLE();  /*Start accepting again*/

    }
```

COLOR Trace_a_ray(ray, n, nid, flag, inter)

```
    RAY *ray;
    int n;
    int nid;
    int flag;
    INTERSECT inter;
    {
        OBJECT          *obj;
        COLOR            col;
        VECTOR2          ip;
        double           mc;
        int hits, retval, hold;
        int flag2, nid2, x, temp, temp2;
        float64 t1, t2;
        ++n_rays;

    /*
    *Check to see if this ray will intersect anything.If not, then
    * return a proper background color. Else, apply the proper
    * illumination model to get the color of the object.
    */

    /*
     * If ray doesn't hit anything in the current node space which
     * travelling through, check
     * surrounding node space for intersections
     */

    flag2 = 0;
    _convf64f32(&t1, tmax);
    _convf64f32(&t2, tmax);

    inter.flag = 0;
    temp = Intersect(ray, &inter, nid);

            if(temp == 1)
            {
        _cpyf64(&t1, &inter.t);
        obj = inter.obj;
            }

        if ((temp == 0)||(flag == 1))
        {
    /*Use flag for case that doing HOSTC40 space due to improper
    partitioning...*/
    /* snode is gotten from &inter */
    /* Probably don't need to pass box since global */

        hits = check_bv(ray, nid);
        temp2 = inter.flag;/* take the bv's and check for */
    /* hit with ray */
```

```
if(hits != 0)
{
if(hits > 1)
{/* Ordering routine if more than 1 hit */
   close(nid, hits);
}

   /* This routine is supposed to order them distance */
   /* from the nid */
   x = 0;

/* This implies that Intersect checks to see if in memory */
   nid2 = hitlist[x]->nid;
   while(x < hits)
        {
        retval  = Intersect(ray, &inter, nid2);
        if(retval == 1)
   {
   flag2 = 1;
   if(_lssf64(&inter.t, &t2) && (inter.obj != obj))
   {
   _cpyf64(&t2, &inter.t);
   obj  = inter.obj;
   hold = nid2;
   }
   }
   x++;
   nid2 = hitlist[x]->nid;
   }
   nid2 = hold;

   if((flag2 == 0)&&(temp2 == 0))/* If hit none of the bv in
hit list */
   {
   return (Background_color(ray));
   }
   }
   else
   {
   if(temp2 == 0)
   return(Background_color(ray));
   }
   }
   ++n_intersects;
/*
 * calculate the point of intersection and pass it to the shade
 * function
 */

        if(temp == 1)
        {
       if(flag2 == 1)
        {
        if(_lssf64(&t2, &t1))
        {
        nid = nid2;
        _cpyf64(&t1, &t2);
        }
```

```
            }
        }
        else
        {
        if(flag2 == 1)
        {
        nid = nid2;
        _cpyf64(&t1, &t2);
        }
        }

        VECADDS(&ip, &t1, &ray->dir, &ray->pos);

        col = Illuminate(&inter, ray, &ip, n, nid, flag);
        /*
         * If colors have overflown, normalize it.
         */

        return (col);
    }
```

void dnext_pix(int len, int snode)

```
          {
          RAY2        ray;
          int i, x, p, objid, primitives, loops;
          double xrmax, xrmin, yrmax, yrmin;
          VECTOR  min, max, prevmin, prevmax, point, start, end;
          double  diffx, diffy, diffz, pdiffx, pdiffy, pdiffz;
          double xr, yr, scale, va, vlax, vlay, vlaz;
          float *data = in_buff2;
          double x_step, y_step, numdir, denom;
          VECTOR hor, ver;
          int nullist[1];
          SHADOW illst, shlst;
    /* the variables used by trace to do it's creation of rays
    must be global else will have to redo all that math here, and
    since have to end up doing it everytime trace is called, see
    no reason why just don't make them global...*/
        VecCross(view.up, view.look_at, hor);
        VecNormalize(&hor);/* horizontal screen vector */
        VecCross(view.look_at, hor, ver);
        VecNormalize(&ver);/* vertical screen vector */
        x_step = 2.0 / view.x_res;
        y_step = 2.0 / view.y_res;
          va = view.angle;
          vlax = view.look_at.x;
          vlay = view.look_at.y;
          vlaz = view.look_at.z;
        primitives = primobj[my_node];
        for(x=0;x<len;x++)
          {
          objid = (int)  *(moved[snode]+x);
          i = 0;
          while(box[my_node]->obj[i]->id != objid)
          {
          i++;
          if(i > primitives)
          { while(1);   }    /*error if this occurs */
          }
```

```
      /* setting flags in obj->lists to 1 for checks */
      if(snode == my_node)
      INT_DISABLE();
p = addtolist(global_list,objid);/*check if already in list*/

      if(snode == my_node)
      INT_ENABLE();
      if(p == 1) /* if already in list, continue  */
      continue;

      illst = box[my_node]->obj[i]->illum->hd;
      shlst = box[my_node]->obj[i]->shads->hd;
      while(illst->next != NULL)
      {
      illst->flag = 1;
      illst = illst->next;
      }
      if(illst->id != 0)
      illst->flag = 1;
      while(shlst->next != NULL)
      {
      shlst->flag = 1;
      shlst = shlst->next;
      }
      if(shlst->id !=0)
      shlst->flag = 1;
      min = box[my_node]->obj[i]->b_min;
      max = box[my_node]->obj[i]->b_max;
      prevmin = box[my_node]->obj[i]->pmin;
      prevmax = box[my_node]->obj[i]->pmax;
      diffx = max.x - min.x;
      diffy = max.y - min.y;
      diffz = max.z - min.z;
      pdiffx = prevmax.x - prevmin.x;
      pdiffy = prevmax.y - prevmin.y;
      pdiffz = prevmax.z - prevmin.z;
      p = 0;
      point = max;
      VecCopy(view.from, ray.pos);

      while(p < 2)
      {
    loops = 1;
      /* points 1->8 */
      while(loops <= 8)
    {
    if(loops == 2)
      point.x = point.x - diffx;
    if(loops == 3)
      point.z = point.z - diffz;
    if(loops == 4)
      point.x = point.x + diffx;
    if(loops == 5)
      point.y = point.y - diffy;
    if(loops == 6)
      point.z = point.z + diffz;
    if(loops == 7)
      point.x = point.x - diffx;
    if(loops == 8)
```

```
     point.z = point.z - diffz;

     VecSub(point, ray.pos, ray.dir);
     scale = VecNormalize(&ray.dir);

      numdir = -vlax*hor.z/(va*hor.x);
scale = (-vlay/va-ray.dir.x*hor.y/hor.x)/(ver.y-ver.x*hor.y/
hor.x);
scale  = numdir - scale*ver.x*hor.z/hor.x;
numdir    =    ver.z*(-vlay/va-ray.dir.x*hor.y/hor.x)/(ver.y-
ver.x*hor.y/hor.x);
scale  = (scale+numdir)*va;

denom=(ray.dir.z-ray.dir.x*hor.z/hor.x)-ray.dir.y*ver.z/
(ver.y-ver.x*hor.y/hor.x)+ray.dir.y*ver.x*hor.z/
(hor.x*(ver.y-ver.x*hor.y/hor.x));

scale = scale/denom + vlaz/denom;

yr = ( ((ray.dir.y*scale-view.look_at.y)/view.angle) -
((ray.dir.x*scale-view.look_at.x)*hor.y/view.angle*hor.x))/
(ver.y - ver.x*hor.y/hor.x);

xr  =  (ray.dir.x*scale-view.look_at.x)/(hor.x*view.angle)  -
yr*ver.x/hor.x;

     if((p == 0)&&(loops == 1))
     {
     yrmin = yr;/* initialization of bounds */
     yrmax = yr;/* on first pass through loop */
     xrmin = xr;
     xrmax = xr;
     }
     else{
     if(yr > yrmax)
     yrmax = yr;
     if(yr < yrmin)
     yrmin = yr;
     if(xr > xrmax)
     xrmax = xr;
     if(xr < xrmin)
     xrmin = xr;
    }
     loops++;
     }
     point = prevmax;
     diffx = pdiffx;
     diffy = pdiffy;
     diffz = pdiffz;
     p++;
     }
/* Now we have the view angle bounds for this moved object */
        if(xrmax>1)
        xrmax = 1;
/* checking for objects that have gone off screen */
        if(xrmin<-1)
        xrmin = -1;
        if(yrmax>1)
        yrmax = 1;
        if(yrmin<-1)
```

```
                yrmin = -1;

            end.x = -(xrmin/x_step) + view.x_res/2;
            start.x   = -(xrmax/x_step) + view.x_res/2;
            end.y = -(yrmin/y_step) + view.y_res/2;
            start.y   = -(yrmax/y_step) + view.y_res/2;

            if(snode == my_node)
            INT_DISABLE();
            add_to_pix_list(start, end, 0);
            /* Determines new pix to calc */
            if(snode == my_node)
            INT_ENABLE();
            }
        }
```

next_scene()

```
        {
        int x, value, i, type, id, flag, primitives;
        float *data=in_buff2;
        int list0[P+1], list1[P+1], list2[P+1];
        int requests;
        int oport;
        HEAD release, prev;
        /*'list' needs to be larger than required cause possibility
        of*/
        /*just as many connections as nodes, and then the 'E' flag..*/
                        x = 0;
                        value = (int) *data;
                        data++;
                        primitives = primobj[my_node];

        /*deallocates space from previous bv for all nodes*/
                        free_bounding_box();
                       nearest_neighbors(list1);
        /* Don't wanna do this in loop*/
        /* Determine which nodes to check for passage of objects */

                        while(value != NULL)
                        {
                        i = 0;    /* assuming id in memory */
                        while(i < primitives)
                        {
                        id = box[my_node]->obj[i]->id;
                        if(id == value)
                        {
                        type = box[my_node]->obj[i]->type;
                        break;
                        }
                        i++;
        /* move this block to encompass all --- */
                        }

            flag = still_contained(i, list1, list2, 0);
        /* create list of node hits */
            x=0;
            while(list2[x] != 'E')    /* Storing list for comparison */
                        {
```

```
                list0[x] = list2[x];
                x++;
                }
        list0[x] = 'E';


        update_all(i, data, type, my_node); /* updating object */
        flag = still_contained(i, list0, list2, 1);
    /* Check if object*/
    /* has moved into a new node space, if so, send it to node*/

        if(flag == 0)    /* Implies has moved out of local space */
        {
        remove_obj(i, list2);
        /* only remove if present elsewhere*/
        }
         data = data+3;
         value = (int) *data;
         data++;
         }

        send_changes_to_remote();/* sends all the info */
                  /* to remote nodes containing local memory  */
        if(frame_cnt != 1)
        {
        INT_DISABLE(); /* don't wanna take a chance*/
        update_netlist();          /* update shad/illum net */
        INT_ENABLE();
        }

        requests = 0;
        for(i=0;i<P;i++)
        {
        if(box[i]->mem == 1)
         {requests++;  }
        }
        for(i=0;i<P;i++)
        {
        if((i!=HOSTC40)&&(i != my_node))
        {

        send_dma(i, x, 0, GO);/* control loop */
    }
    }

        while(changes < requests-1);
        /* wait till have received */
        /* all changes */
        send(HOSTC40, newdata, append, UPDATE);
        send(HOSTC40, remdata, remove, MOVEDATA);
        new_frame();
        release = global_list->hd;/* freeing up global list */
        release = release->next;/* and initializing again */
        while(release != NULL)
        {
        if(release == global_list->tail)
        {
        global_list->tail->next = NULL;
        global_list->tail->id= 0;
        break;
        }
        prev = release;
        release = release->next;
        INT_DISABLE();/* not actually interruptible here */
        free(prev);
        INT_ENABLE();
        }
        global_list->hd->next = NULL;
        global_list->hd->id   = 0;

    }
```

update_all(int id, float *data, int type, int snode)

```
    {
     switch(type)
            {

        case T_CONE     : update_cone(id, data, snode);
                                 break;
        case T_QUADRIC  : update_quad(id, data, snode);
                                 break;
        case T_SPHERE   : update_sphere(id, data, snode);
                                 break;
        case T_HSPHERE  : update_hsphere(id,data, snode);
                                 break;
        case T_RING     : update_ring(id, data, snode);
                                 break;
        default        :  return;     /* Error somewhere */
            }
     }
```

update_sphere(int id, float *data, int snode)

```
    {
    SPHERE *s;

           s = box[snode]->obj[id]->obj;
           s->center.x  = *data;
           data++;
           s->center.y  = *data;
           data++;
           s->center.z  = *data;

    /*
     * Setup of bounding box, along with previous bbox.
     */
        box[snode]->obj[id]->pmin = box[snode]->obj[id]->b_min;
        box[snode]->obj[id]->pmax = box[snode]->obj[id]->b_max;
        box[snode]->obj[id]->b_min.x = s->center.x - s->radius;
        box[snode]->obj[id]->b_min.y = s->center.y - s->radius;
        box[snode]->obj[id]->b_min.z = s->center.z - s->radius;
        box[snode]->obj[id]->b_max.x = s->center.x + s->radius;
        box[snode]->obj[id]->b_max.y = s->center.y + s->radius;
        box[snode]->obj[id]->b_max.z = s->center.z + s->radius;
    }
```

remove_obj(int object_id, int list[])

```
 {
int x, primitives, id;
OBJECT *obj;
 primitives = primobj[my_node];
 if((pix_flag == 1)||((list[0] != 'E')&&(primitives > 1)))
 {/* present elsewhere? If so, remove... */
  /* if so, delete from current memory */
  /* as long as will have at least i object */


   if(pix_flag == 1)
   {
   obj = box[my_node]->obj[object_id];
   free(obj->obj); /* free up primitive */
   free(obj);    /* free up object struct */
   primitives = primobj[my_node] -1;
   primobj[my_node] = primitives;

   for(x = object_id; x < primitives; x++)
   {
   box[my_node]->obj[x] = box[my_node]->obj[x+1];
   }                    /* rearrange the array */
   }
   else{
   id  = box[my_node]->obj[object_id]->id;
   remdata[remove++] = (float) id;
   /* Notify others of change */
   }

   }
}
```

update_netlist()

```
 {

 LISTS *lst;
 SHADOW  first, prev;
 int q, z, x, objid, prims;
 int oldlength;
                /* list created by      */
                 prims = primobj[my_node];
                 for(q=0;q<P;q++)
                 {
                  if(q == HOSTC40)
                  continue;

                   oldlength = movesize[q];
                  if(oldlength==0) /*don't do anything if==0*/
                  continue;
                 x=0;
                 while(x < oldlength)
                 {
                  objid = (int) *(moved[q]+x);
                 x++;
                 z=0;
                 while(box[my_node]->obj[z]->id != objid)
                 {
                    z++;
                    if(z >= prims)
                    while(1);       /* error if reaches here */
                    }
                    z=0;
                    while(z < 2)
                    {
                    if(z == 0)
                    {
                    first =  box[my_node]->obj[z]->illum->hd;
                    lst   =  box[my_node]->obj[z]->illum;
                    }
                    if(z == 1)
                    {
                    first = box[my_node]->obj[z]->shads->hd;
                    lst = box[my_node]->obj[z]->shads;
                    }
                    while(first != NULL)
                    {
                    prev  = first;
                    first = first->next;
                    if(prev->flag == 1)
                     {
                     INT_DISABLE();
                     delete_from_shill(lst,prev->id,
                     prev->rnode);
                     INT_ENABLE();
                     }
                    if(prev == lst->tail)
                    break;
                    }
                    z++;
                    }
                    }
                    /* now can free old moved[snode] array */
                    free(moved[q]);
                    }
}
```

receiving_moved_net(int   port,int   len,int snode)

```
{
 float *data=in_buff1;
 movesize[snode] = len;
 if(len != 0)
 {
 moved[snode] = (float *) calloc(len,sizeof(float));
 data = moved[snode];
 in_msgk_float(port,data,1,len);
 /*now should have data in buffer */
 /* moves[snode]+len = NULL; */
 /*setting last element in array to 0*/
 dnext_pix(len,snode);
 }
 net_updater++; /*increment this global flag count  */
 if(net_updater == 0) /* debugging code */
 while(1);
 return;
}
```

int create_moved_net(void)

```
{
float *data = in_buff2;
TEMPLIST *lst;    /* make declaration global */
SHADOW lst1, lst2;
HEAD hst;
int locobj, value, prims, x;
            locobj = 0;
            for(x=0;x<P;x++)
            {
            if(x == HOSTC40)
            continue;
       pows[x]->hd->id = 0;
       pows[x]->hd->next = NULL;
       pows[x]->tail->id = 0;
       pows[x]->tail->next = NULL;
            }
value = (int) *data;
while(value != NULL)
{
data = data+4;
prims = primobj[my_node];
x=0;
while(box[my_node]->obj[x]->id != value)
{
if(x >= prims)          /* break if exceeds max count */
break;
x++;
}

value = (int) *data;
if(x == prims)
{ continue; } /* obj not present ! */

lst1  = box[my_node]->obj[x]->shads->hd;
lst2  = box[my_node]->obj[x]->illum->hd;

while((lst1->next != NULL)&&(lst1->id != 0))
{
lst = pows[lst1->rnode];
INT_DISABLE();
x = addtolist(lst, lst1->id);
INT_ENABLE();
lst1 = lst1->next;
}
if(lst1->id != 0)             /* just in case only 1 obj in
list */
{
lst = (TEMPLIST *) pows[lst1->rnode];
INT_DISABLE();
x = addtolist(lst, lst1->id);
INT_ENABLE();
}
                          /* might be doing extra test case */

while((lst2->next != NULL)&&(lst2->id != 0))
{
lst = pows[lst2->rnode];
INT_DISABLE();
x = addtolist(lst, lst2->id);
INT_ENABLE();
lst2 = lst2->next;
}
if(lst2->id != 0)/* just in case only 1 obj in list */
{
lst = (TEMPLIST *) pows[lst2->rnode];
INT_DISABLE();
x = addtolist(lst, lst2->id);
INT_ENABLE();
}
}
hst = pows[my_node]->hd;
while(hst->next != NULL)
{
locobj++;
hst = hst->next;
}
if(hst->id != 0)/* for last obj, and first...if only 1 */
locobj++;
return(locobj);
/* Send these moved data in function next_pix() */
/* and deallocate entire list after read and send data */
}
```

int addtolist(TEMPLIST *LIST, int id)

```
{
int max, min;
HEAD h, temp, prev;
int flag;
flag = 0;
        if(LIST->hd->next == NULL)
        /* For First Time Through */
        {
        LIST->hd->id = id;
        LIST->hd->next   = LIST->tail;
        return(flag);
        }
        /* need case for when second in list */
      if((LIST->hd->next==LIST->tail)&&(LIST->tail->id== 0))
        {
        if(LIST->hd->id == id)
        return(1); /* already in list */
        LIST->tail->id = id;
        LIST->tail->next = NULL;
        return(flag);
        }
        if((h =   (HEAD)  malloc(sizeof(HD))) == NULL)
        while(1);
        h->id = id;
/* put it in order to save time for check if already in list*/
        max = LIST->tail->id;
        min = LIST->hd->id;

        if(id < min)
        {
        h->next      = LIST->hd;
        LIST->hd     = h;
        return(flag);
        }
        temp = LIST->hd;
        while((temp != NULL)&&(temp->id <= id))
        {
        if(temp->id == id)
        {
        free(h);         /* Already in list */
        return(1);
        }
        prev = temp;
        temp = temp->next;
        }
        prev->next = h;   /* append to list */
        if(temp == NULL)  /* last in list */
        {
        h->next = NULL;
        LIST->tail = h;
        }
        else{
        h->next    = temp;
        }
        return(flag);
}
```

send_to_remote_net(int did, int dnode, int rid, int type, int rnode)

```
 {
   float message[3];
   sendcounter++;
   message[0] = (float) did;
   message[1] = (float) rid;
   message[2] = (float) rnode;
   send(dnode, message, 3, type);
   return;
 }
```

add_to_local_net(int port, int len, int snode, int type)

```
{
LISTS *lst;
int id, rid, rnode, x, prims;
float *data = in_buff1;

 netcounter++;
 in_msgk_float(port, data, 1, len);
 id = (int) *data++;
 rid = (int) *data++;
 rnode = (int) *data;

x = 0;
prims = primobj[my_node];
 while(box[my_node]->obj[x]->id != id)
{
x++;
if(x >= prims)
while(1);
/* if greater than size of # primitives, error somewhere */
}
  if(type == SHADOWS)
{ lst = box[my_node]->obj[x]->shads; }
  else
{ lst = box[my_node]->obj[x]->illum; }
 add_to_shill(lst, rid, rnode);
 }
/*
* takes extra time this way, but saves 2 SHADOWs per obj which
* is quite a
* considerable chunk of memory, so must do it like this.
*/
```

void add_to_shill(LISTS *LIST, int id, int rnode)

```
{
int     max, min;
SHADOW   h, temp, prev;

if(LIST->hd->next == NULL)       /* For First Time Through */
        {
        LIST->hd->id = id;
        LIST->hd->next   = LIST->tail;
        LIST->hd->rnode = rnode;
        LIST->hd->flag  = 0;
        return;
        }
        /* need case for when second in list */
       if((LIST->hd->next==LIST->tail)&&(LIST->tail->id==0))
        {
        if((LIST->hd->id == id)&&(LIST->hd->rnode == rnode))
        return;          /* already in the list */
        LIST->tail->id = id;
        LIST->tail->rnode = rnode;
        LIST->tail->next = NULL;
        LIST->tail->flag = 0;
        return;
        }
        max = LIST->tail->id;
        min = LIST->hd->id;

        if((h =   (SHADOW)  malloc(sizeof(SLIST))) == NULL)
        while(1);

        h->id = id;
        h->rnode       = rnode;
        h->flag        = 0;

        if(id < min)
        {
        h->next        = LIST->hd;
        LIST->hd       = h;
        return;
        }
        temp = LIST->hd;
```

```
            while((temp != NULL)&&(temp->id <= id))
            {
            if((temp->id == id)&&(temp->rnode == rnode))
            {
            temp->flag = 0;
            /* resetting flag so knows that hit again */
            free(h);        /* Already in list */
            return;
            }
            prev = temp;
            temp = temp->next;
            }
            prev->next = h;   /* append to list */

            if(temp == NULL)  /* last in list */
            {
            h->next = NULL;
            LIST->tail = h;
            }
            else{
            h->next     = temp;
                }
            return;
    }
```

void delete_from_shill(LISTS *LIST, int id, int rnode)

```
    {
    SHADOW  temp, prev;
    /* the following is for the delete case */
    temp = LIST->hd;
    prev = LIST->hd->next;

    if(temp->id == 0)/* all objects already deleted */
    return;

    /* Checking for special case of only 1 object in list */
    if(prev->id == 0) /* only one object ! */
    {
    temp->next = NULL;  /* Now uninitialized again */
    temp->id = 0;
    return;
    }
    /* checking for case that only 2 objs in list */
            if(LIST->hd->next == LIST->tail)
            {
            if((temp->id == id)&&(temp->rnode == rnode))
            {
            LIST->hd = LIST->tail;
            LIST->tail = temp;
            LIST->tail->id = 0;
            LIST->tail->next = NULL;
            LIST->hd->next = LIST->tail;
            return;
            }
            else
            {
            LIST->tail->id = 0;
```

```
         return;
         }
         }
   /* Checking for case that id is at the head of the list */
      if((temp->id == id)&&(temp->rnode == rnode))
   {
      LIST->hd = prev;/* points to next item in list */
      free(temp);
      return;
      }

      while((temp->id != id)||(temp->rnode != rnode))
         {
         prev = temp;
         temp = temp->next;
         if(temp == NULL)
         { return; }/* Object not present in this mem space */
         }
         prev->next = temp->next;
         if(temp->next == NULL)
         {
         LIST->tail = prev;      /* resetting tail */
         }
   free(temp); /* deallocate memory */
   }

   /* must update netlist before send out objs to redo for next
   frame */
```

update_netlist()

```
   {
      LISTS *lst;
      SHADOW  first, prev;
      int q, z, x, objid, prims;
      int oldlength;
                    /* list created by      */
                     prims = primobj[my_node];
                     for(q=0;q<P;q++)
                     {
                     if(q == HOSTC40)
                     continue;
                     oldlength = movesize[q];
                     if(oldlength == 0)
                     /* don't do anything if == 0 */
                     continue;
                     x=0;
                     while(x < oldlength)
                     {
                     objid = (int) *(moved[q]+x);
                     x++;
                     z=0;
                     while(box[my_node]->obj[z]->id != objid)
                     {
                     z++;
                     if(z >= prims)
                     while(1);       /* error if reaches here */
                     }
                     z=0;
                    while(z < 2)
                    {
                    if(z == 0)
                    {
                    first =  box[my_node]->obj[z]->illum->hd;
                    lst   =  box[my_node]->obj[z]->illum;
                    }
                    if(z == 1)
                    {
                    first = box[my_node]->obj[z]->shads->hd;
                    lst = box[my_node]->obj[z]->shads;
                    }
                    while(first != NULL)
                    {
                    prev  = first;
                    first = first->next;
                    if(prev->flag == 1)
                    {
                    INT_DISABLE();
                  delete_from_shill(lst, prev->id, prev->rnode);
                    INT_ENABLE();
                    }
                    if(prev == lst->tail)
                    break;
                    }
                    z++;
                    }
                    }
                    /* now can free old moved[snode] array */
                    free(moved[q]);
                    }
}

/*********************************************
 SETTING UP INTERRUPT SEVICE ROUTINES FOR EACH COMMMPORT
*********************************************/


void c_int08() {  cpuint(0); }

void c_int12() {  cpuint(1); }

void c_int16() {  cpuint(2); }

void c_int20() {  cpuint(3); }

void c_int24() {  cpuint(4); }

void c_int28() {  cpuint(5); }

void cpuint(int port)
  {
  int type, dnode, snode, len, k;
  /* here follows all the routing stuff necessary to do this
  correctly */
  k = 0;
```

```
dnode = in_word(port);
if(dnode != my_node)
{

forward(dnode, port); /* if not yours, forward message */
else
{
   type  = in_word(port);/* Get incoming info */
   snode = in_word(port);
   len   = in_word(port);

   switch (type) {

                case FROM : command(len, port, FROM);
                             break;
                case AT   :  command(len, port, AT);
                             break;
                case UP :    command(len, port, UP);
                             break;
                case ANGLE : command(len, port, ANGLE);
                             break;
                case RES:     command(len, port, RES);
                             break;
                case LITE :   command(len, port, LITE);
                             break;

                case BKGND : command(len, port, BKGND);
                             break;
                case SURF:   solid(len, port, snode, SURF);
                             break;
                case CON:    if(cap == 1)
                              { self = 1; snode = my_node; }
                              solid(len, port, snode, CON);
                             break;
                 case CONR:   self = 0;
                              solid(len, port, snode, CON);
                             break;
                case POLY:   if(cap == 1)
                              { self = 1; snode = my_node; }
                              solid(len, port, snode, POLY);
                             break;
                case POLYR:   self = 0;
                              solid(len, port, snode, POLY);
                             break;
                case SPHRE:   if(cap == 1)
                             { self = 1; snode = my_node;}
                             solid(len, port,snode, SPHRE);
                             break;
                case SPHRER:  self = 0;
                             solid(len,port, snode, SPHRE);
                             break;
                case HSPHRE:  if(cap == 1)
                               { self = 1; snode = my_node;}
                            solid(len, port, snode, HSPHRE);
                             break;
               case HSPHRER :  self = 0;
                            solid(len, port, snode, HSPHRE);
                             break;
                case RNG:     if(cap == 1)
```

```
                          { self = 1; snode = my_node;}
                          solid(len, port, snode, RNG);
                         break;
               case RNGR :  self = 0;
                          solid(len, port, snode, RNG);
                         break;
               case QUAD:   if(cap == 1)
                          { self = 1; snode = my_node;}
                          solid(len, port, snode, QUAD);
                         break;
               case QUADR:   self = 0;
                          solid(len, port, snode, QUAD);
                         break;
               case BV :     b_volume(len, port, snode);
                         break;

               case PACK:    pack(len, port);
                         break;
               case REQMEM :  reqmem(snode);
                         break;
               case LOBJ:     last_obj(snode);
                         break;
               case PARAM:   params(port);
                         break;
               case NFRAME:  NF = 1; /* New Frame ! */
                         break;
               case MOVEDATA:read_in(port, len, snode);
                         break;
               case DELETEOBJ:delete_obj(port, len, snode);
                         break;
               case GO :     goes++;
                         break;
               case LUXES:
               add_to_local_net(port, len, snode, type);
                         break;
               case SHADOWS:
               add_to_local_net(port, len, snode, type);
                         break;
               case NETMAP:
               receiving_moved_net(port, len, snode);
                         break;
              case FLASH   :  flash++;
                         break;
              }
    }
}


/*   8/22/94      MP
*The function of nearest_neighbors() is to determine the nodes
*in the network which are directly connected to the given node
*in an arbitrary 3-D grid archtitecture. It then stores them
*in a list which is used by other functions to determine if an
*object has moved into or out of another node.
*This nearest neighbor approach assumes that objects will not
*move further than 1 node away in a given frame. This will in
*general be true for any realistic motion detectable by the
*human eye across the screen. However, for very large networks
*this could break down if say the physical space assigned to
each node was quite small.
```

```
*Function  still_contained()  basically  determines  if  the
*bounds  of  the  passed  object  id  (therefore,  object)  are
*contained within the node space of those node ids contained
*within a passed list. If they are contained, then the nodes
*in which it is contained are stored in another list which is
*required as a check against a list of nodes in which it was
*contained before it was moved. The 2 lists are compared to
*determine if an object has moved into a new node. If it has
*moved into a new node, then the object in its entirety
*is send to the node. In addition a check is done to determine
*whether the object is still contained within the local node.
*If this is the case, a value of 1 is returned.
*Function common() just takes 1 list(array) and an integer
*value and checks for inclusion of the integer within the
*array. A value of 1 is returned if not included.
*/

/*The following code is cumbersome for smaller networks. But
*for a large number of nodes, it cuts down drastically on the
*amount of computation. It only checks the nearest neighbors
*for passing of objects. These are the only ones that would
*realistically be passed an object anyway.
*/
```

nearest_neighbors(int LIST[])

```
  {
  int nid, x, myplane;
  int d2_p = netsize*netsize2;
  /*number of processors in the mesh -- d2_p*/
  int planes    = P/d2_p;/*number of planes in network*/
  int my_nodep, col, row, rows;
     x = 0;
     LIST[x++] = my_node;  /* Need for comparison */
     planes         = (P/d2_p)- 1;    /* Watch this carefully*/
     myplane  = my_node/d2_p;
     my_nodep = my_node%d2_p;
     rows = my_nodep/netsize2; /* done */
     col = my_nodep%netsize2;
     col = myplane*netsize2 + col;

     row = rows+1;
     nid    =  (planes*d2_p) + (netsize*row)
        + ((col+netsize2)%netsize2);

     if(row < netsize)
     {
     LIST[x] =  nid;
     x++;
     }
     col = col+1;
     nid    =  (planes*d2_p) + (netsize*row)
      + ((col+netsize2)%netsize2);

     if((col < netsize2)&&(row < netsize))
     {
     LIST[x] =  nid;
     x++;
     }
     row = row - 1;
     nid    =  (planes*d2_p) + (netsize*row)
       + ((col+netsize2)%netsize2);
```

```
    if(col < netsize2)
                        {
                        LIST[x] = nid;
                        x++;
                        }
                        row = row - 1;
    nid    =  (planes*d2_p) + (netsize*row)
     + ((col+netsize2)%netsize2);

    if((row >= 0)&&(col < netsize2))
                        {
                        LIST[x] = nid;
                        x++;
                        }
    col = col -1;
    nid    =  (planes*d2_p) + (netsize*row)
    + ((col+netsize2)%netsize2);

    if(row >= 0)
                        {
                        LIST[x] = nid;
                        x++;
                        }
    col = col -1;
    nid    =  (planes*d2_p) + (netsize*row)
     + ((col+netsize2)%netsize2);

    if((row >= 0) &&(col >= 0))
                        {
                        LIST[x] = nid;
                        x++;
                        }
                        row = row+1;
    nid    =  (planes*d2_p) + (netsize*row)
     + ((col+netsize2)%netsize2);

    if(col >= 0)
                        {
                        LIST[x] = nid;
                        x++;
                        }

                        row++;
    nid    =  (planes*d2_p) + (netsize*row)
     + ((col+netsize2)%netsize2);

    if((col >= 0) && (row < netsize))
                        {
                        LIST[x] = nid;
                        x++;
                        }

                        LIST[x] = 'E';  /* end of list */
}
```

```
int still_contained(int obj_id, int list1[],
      int list2[], int still)
   {
   double xmin,  xmax,  ymin,  ymax;
   double xtmin, xtmax, ytmin, ytmax;
   int type, nid, i, x, flag, t;
   int *list;
   OBJECT *obj;

           x = 0;
           flag = 0;
             xmin =       box[my_node]->obj[obj_id]->b_min.x;
             ymin =       box[my_node]->obj[obj_id]->b_min.y;
             xmax =       box[my_node]->obj[obj_id]->b_max.x;
             ymax =       box[my_node]->obj[obj_id]->b_max.y;
             x     = 0;
             i     = 0;


             if(still == 1)
             /* still == 1 -> if moved send object */
             {
             list = list2;
             }
             else
             {   list = list1;
             /* Put node 0 in initialization of list1 */
             }
            while(list[i] != 'E')
            {

              nid  =  list[i];
               i++;

              xtmin = box[nid]->min.x;
              xtmax = box[nid]->max.x;
              ytmin = box[nid]->min.y;
              ytmax = box[nid]->max.y;

   /*Begin checking for containment within node object space*/
       if((xmin >= xtmin) && (xmin <= xtmax)
       &&(ymin >= ytmin) && (ymin <= ytmax))
       {
              list2[x++] = nid;
       }
       else{

       if((xmax >= xtmin) && (xmax <= xtmax)
      &&(ymax >= ytmin) && (ymax <= ytmax))
       {

              list2[x++]= nid;
       }
       else{
       if((ymax >= ytmax) && (xmax <= xtmax)
      &&( xmax >= xtmin) && (ymin <= ytmax))
       {
              list2[x++] = nid;
       }
      else{
      if((ymax >= ytmax) && (xmax >= xtmax)
     &&(ymin <= ytmax) && (xmin <= xtmax))
      {
             list2[x++] = nid;
      }
      else{
      if((ymax <= ytmax) && (xmax >= xtmax)
     &&( ymax >= ytmin) && (xmin <= xtmax))
      {
             list2[x++] = nid;
      }
      }
      }
      }
      }
      }
       list2[x] = 'E';
 /* FLAG INDICATING LAST OF THE PROCESSORS TO SEND IT
    TO--> can't have more than 'E' processors!!*/
                  if(still == 1)
                  {
                  /* Compare the 2 lists here */
                  t=0;
                  obj = box[my_node]->obj[obj_id];
                  type = obj->type;

                  while(list2[t] != 'E')
                  {
                  if(list2[t] == my_node)
                  {
                  flag = 1;/* still within local memory */
                  t++;
                  continue;
                  }
                  x = common(list1, list2[t]);
                  if(x == 1)            /* New node ! */
                  {
                  i = list2[t];
                  locality  = 1;
                  /*flag so that uses special_send*/
                  transfer(type, obj, i, 0);
                  /* Send data to new node */
                  locality  = 0;
                  }
                  t++;
                  }
                  }
                  else{
                      flag = 0;
                      }
                 return (flag);
 }
```

common(int list[], int value)

```
   {
   int     x =0;
```

```
          while(list[x] != 'E')
          {
          if(list[x] == value)
          {
          return(0);   /*   same as previos scene */
          }
          x++;
          }
         return(1);  /*   implies new node contains object */
}
```

void new_frame()

```
{
int   i, type, node;
float bv[6];/* Array for bounding volume */
VECTOR  min, max;
int x;
OBJECT  *obj;
SURFACE *surf;
x=0;
/*
 * Build the bounding box structures.
 */
          if(frame_cnt != 0)
          {
          while(append > 0)
          {
          surf = (SURFACE *) forward_box[append*2-1];
          obj = (OBJECT *) forward_box[append*2-2];
          type = obj->type;

          while(memory[x] != 'E')
          {
          node = memory[x];
          /* Sending to those nodes which */
          /* contain 'my' memory          */
          if(node != my_node)
          {
          send_surf(surf, node);
          transfer(type, obj, node, 1);
          /* to let know that belongs to */
          /* my local memory space */
          }
          x++;
          }
          append = append-1;
          }

          for(x=0;x<P;x++)
          {
          if((x == my_node)||(x==HOSTC40))
          continue;
          send_dma(x, x, 0, FLASH);/* control loop */
          }
          while(flash < P-2);  /* waiting for all new objs */
          flash = 0;
          }
          x=0;
```

```
         INT_DISABLE();
        while(x < P)
        {
        node = x;
             if(box[node]->mem != 0)
        {
        pobjects[node] = primobj[node];
         /* need pobjects[] for build_b...*/
         Build_bounding_slabs(node);
         }
         x++;
         }
             INT_ENABLE();
        min = box[my_node]->root->b_min;
        max = box[my_node]->root->b_max;

        box[my_node]->min = box[my_node]->root->b_min;
        box[my_node]->max = box[my_node]->root->b_max;

        bv[0] = min.x;
        bv[1] = min.y;
        bv[2] = min.z;
        bv[3] = max.x;
        bv[4] = max.y;
        bv[5] = max.z;
        i = 0;

        if(NF == 0)
        {
        pixlength = 0;             /* used for recalc of pixels*/
        while(goes < P-2);/* Wait for other nodes...*/
        }
        goes = 0;
        /* Determine pixels to recalculate */
        if(frame_cnt != 0)
        {
        next_pix();
        }

        while(i < P)
        {
        if((i != my_node)&&(i != HOSTC40))
        {
        send_dma(i, bv, 6, BV);
        }
        i++;
        }
}
```

5231.44