1995

MIRAGE : a system for distributed image generation on workstation clusters

Darrin Weber
Lehigh University

AUTHOR:

Weber, Darrin L.

TITLE:

Mirage: A System for Distributed Image Generation on Workstation Clusters

DATE: May 28, 1995

MIRAGE: A System for Distributed Image Generation on Workstation Clusters

by

Darrin Weber

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Department of Electrical Engineering and Computer Science

Lehigh University

Bethlehem, Pennsylvania 18015

June 1, 1995

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science.

Feb. 14, 1995
Date

Thesis Advisor

Thesis Advisor

Chairperson of Department

Table of Contents

List of Figures

Abstract

This thesis describes an efficient and robust implementation of the "Supervisor-Worker" model for distributing and coordinating the parallel tasks of image generation[1]. Image generation by ray-tracing is well suited for effective distributed processing due to its intensive computations and its ability to be divided into independent components. A system for utilizing clusters of interconnected workstations as a distributed platform for generating photo-realistic images and animations is presented. Additionally, enhancements to the ray-tracing technique of image generation are presented. Specifically, an "adaptive" octree method for organizing the objects within a scene is discussed. Utilizing the presented distributed application and enhancements to the ray-tracing technique, a near linear performance increase is experienced for certain characteristic images. Although near linear performance can be observed for certain images, analysis of the presented model reveals several factors which limit the speed-up which can be achieved by distributing image generation across a cluster of workstations. Correlations between performance and image complexity, image size, and communication overhead of the presented model are examined. Further, the communication overhead and its effect on performance is evaluated in terms of the network medium used to connect the workstations of a cluster. A high-speed

switch designed specifically for clustering workstations is compared to Ethernet to analyze the performance impact on image generation, utilizing the presented "Supervisor-Worker" model.

Chapter 1. Introduction

The increasing presence of clusters of interconnected workstations has inspired research into exploiting local area networks as viable distributed processing environments. Workstation clusters, common to both educational and business environments, provide a flexible, powerful distributed processing environment. Workstation clusters are flexible in the cost, type, and number of workstations that constitute a particular network configuration. The flexibility of workstation clusters allows computing environments to be tailored to specific processing requirements as well as expanded for future needs. The supervisor-worker model for distributed image generation presented in this paper efficiently utilizes the power available from workstation clusters. The high computational cost of image generation makes it well suited for distributed processing[2,3,4]. The image generation technique of ray-tracing is implemented due to its suitability for application in distributed environments.

1.1. Ray-Tracing Distribution

Ray-tracing generates photo-realistic images by tracing the interaction of light rays and objects within a three-dimensional computer model. Rays are traced backward from a viewer perspective into the model according to the physical

and optical laws of nature. The intersections of the traced rays and objects in the scene determine the image produced. Recursive rays determine shadows, reflectivity, and transparency. Each primary ray that is fired into the scene corresponds to one pixel in the final image. Each primary ray can be traced independently and in parallel. Distributed processing techniques can be applied easily by partitioning the ray-tracing process across a cluster of workstations. Although each ray can be traced independently, the overhead incurred in distributing each ray outweighs the computation time for individual rays. The approach taken by Mirage is the distribution of individual scan lines of the final image to the worker nodes of a cluster. This distribution strategy allows coherence between the pixels of the same scan line to be exploited. By distributing scan lines instead of individual rays, the distribution overhead is minimized compared to the computation time.

This technique does set a physical maximum number of workstations that can be used effectively for ray-tracing an image. Clearly, no benefit can be gained by utilizing a number of workstations exceeding the number of scan lines in the final image. By distributing the lines of an image to a cluster of workstations to be rendered, the expected performance increase should be nearly linear. However, three factors, setup time, communication overhead, and

collection time, will limit the performance increase experienced by utilizing the distributed system. Setup time involves the time required by the worker nodes to acquire and organize all of the resources required for image generation. Communication overhead is the direct cost as a result of assigning lines to be rendered to the workstations of a cluster. Messages are passed between a supervisor and worker nodes for coordinating the distributed image generation. Collection time results from the transference of rendered lines from the worker nodes to the Supervisor for composition into the final image.

1.2. Computing Environment

Implementation of Mirage for image generation was done in ANSI C. Portable system constructs and network protocols, namely Berkeley's socket interface and TCP/IP, were used in the implementation[5,6]. The testing environment consisted of eight RISC workstations connected by an Ethernet and IBM's Allnode high-speed switch[7]. The high-speed switch is specifically designed for clustering workstations. This environment, with two independent network transports, provided versatility in evaluating the effect of communication overhead on the overall performance of the image generation system.

1.3. Organization

Even though the technique of ray-tracing forms the basis for Mirage's image generation, the functional details and imaginary world used to represent the scene to be rendered are unique. Chapter 2 contains the functional details on the image generation process. The process for describing the three-dimensional world to be rendered, the format of the output image, and the status and performance reports are discussed. The resources required for rendering an image are detailed, as well as the input and output for the entire image generation process.

Many papers have discussed the theoretical distribution of image generation but they often lack implementation details and characteristics such as portability, efficiency, and fault tolerance. Chapter 3 presents a highly portable, efficient system based on the supervisor-worker paradigm for distributing the generation of images amongst a cluster of workstations. Implementation details, fault tolerance strategies, and effective debugging methods for the distributed system are presented. Additionally, an efficient communication protocol, utilized between the supervisor and worker nodes, is examined.

Chapter 4 discusses the enhancements to the basic ray-tracing technique for maximizing the performance of image generation. An "adaptive" octree technique for efficiently organizing the objects in a scene is presented. An

efficient organization strategy reduces the number of computations required for detecting ray-object intersections. In addition to the organization strategy presented, shadow caches and optimized object functions are detailed as enhancements to the ray-tracing technique.

Following the presentation of ray-tracing enhancements, the performance of Mirage is analyzed. Relations between performance and image characteristics, such as image complexity, image size, and communication cost, are evaluated. Sample images are rendered to analyze the performance of the presented image generation system.

The evaluation of IBM's Allnode high-speed switch is presented in Chapter 6. The performance and suitability of the switch as it relates to Mirage and a similar class of distributed applications is analyzed and predicted.

Chapter 7 presents some conclusions regarding the use of the presented system among a cluster of workstations for a similar class of computationally intensive applications. Current research focuses on network mediums with the bandwidth and speed that will allow an increasingly larger number of distributed workstations to be interconnected. With larger geographical regions being interconnected due to technological advances in network hardware, distributed applications similar to the presented system will provide the power computing platform of the future.

Chapter 2. Image Generation Overview

The Mirage image generation system is designed to ray-trace an environment of three-dimensional objects to produce photo-realistic images. Light rays are traced through the three-dimensional world and an image is computed based on the physical and optical laws of nature. Mirage calculates shadows, reflections, and transparencies of objects through their interactions with light rays. Multiple light sources of differing types are used to illuminate a particular model. For realism, the three-dimensional objects that compose a scene are assigned specific surface characteristics that simulate their real world textures. Constructive solid geometry is employed to model complex objects by using basic geometric constructs. Constructive solid geometry allows the intersection, union, and clipping of basic geometric shapes by planes and conics. The presented image generation system incorporates natural camera motion along with other variables such as camera focal length, tilt, and field of view. Animation is implemented through object transformations and the use of "key framing". The user defines certain "key" frames along a path and the system calculates the "tween" frames for smooth animation.

2.1. Image Generation Input

The three-dimensional environment used to generate an image is described by three text files. The three input files describe the complete environment and parameters utilized to ray-trace it to produce a photo-realistic image. Resources of the image generation process includes the Studio, Model, and Material files. The Studio file defines the placement of the camera(viewer perspective point), global lighting, key-frames, object animations and transformations, image quality and size, and global rendering options. There exists an entry in the Studio file for every "key" scene in the animation. The Material file contains definitions for surface characteristics that are assigned to the objects within a scene. Surface characteristics such as reflectivity, transparency, and roughness are defined for particular materials. The Model file contains the layout of the three-dimensional environment to be ray-traced. The Model file defines the graphic primitives, constructive solid geometry, lighting, grouping of objects, and the transformations of groups.

The three text files provide the resources for generating the final photo-realistic image. Text files are used as the input for rendering to allow direct portability of the image resources between heterogeneous hardware platforms. The text files also serve as a base for translating models from CAD applications for use in the presented image generation

system. Appendix A contains sample source files for images discussed in this paper.

Mirage contains a compiler for converting the source input files into binary counterparts. The binary versions ultimately provide the input to the image generation system. Binary format files are used in order to speed up the rendering process by removing the parsing overhead which can be significant for large projects. The binary files consist of variable length records that can be read very quickly compared to the time consuming process of parsing a textual file. Additionally, the input files can be compiled separately. The capability of compiling the resource files independently, facilitates the debugging of the production of an image or animation. Only the resource file that is modified needs to be recompiled, thus, decreasing the cycle for generating test images. One important point of Mirage is that the same resource files are used for both the distributed and single-processor implementations. The portability of the image resources results from utilizing the same rendering engine in both the distributed and single-processor implementations.

2.2. Image Generation Output

Mirage produces true-color (24 bits per pixel) images in the industry standard Targa 2 file format[8]. The resolution and quality of the rendered image is adjustable

and specified by the user. The image can be of any resolution, only constrained by the increased time and memory required to ray-trace larger images. The quality of the final rendering is selected by the user in terms of the level of anti-aliasing that is applied during the image generation process. Anti-aliasing is the process of over-sampling and averaging of the pixels that compose an image. The higher the level of anti-aliasing, the better the quality of the resulting image and the greater the rendering time. In the case of animations, output images are sequentially numbered to facilitate single-frame recording or interactive viewing.

2.3. Image Generation Feedback

During image rendering and at its completion, statistics are reported by Mirage. The statistics include timings of particular segments of the rendering process and memory usage. The memory usage statistics indicate the general complexity of the scene. Memory usage reflects the number of objects in the scene and their spatial organization. The setup, rendering, and collection times are tracked as well as the contribution of each worker to the final image. The data presented can be used to determine the number of workstations required to generate images in an acceptable time frame. The reported data also helps the user to determine the best mix of rendering

parameters, such as anti-aliasing, to apply for the highest quality image with the available resources.

Chapter 3. Distributed Image Generation Model

Mirage follows the "Supervisor-Worker" paradigm for distributed computing. A "supervisor" process manages the distribution of the computational task and collects the results from the "worker" processes. Worker processes distributed across the cluster, complete parallel computational tasks as assigned and coordinated by the supervisor.

During the image generation, a supervisor process manages the resources required to ray-trace a particular image. The supervisor distributes the resources required for image rendering, such as three-dimensional models, materials in the scene, and animation specifications to all of the worker nodes. Following the distribution of scene resources, the supervisor assigns individual scan lines to the workers to render as requested. The supervisor then collects all of the rendered scan lines from each worker for final assembly into a completed image.

The resources required for image generation are distributed through several communication mechanisms. Communications between the supervisor and worker nodes include both control and data messages. Figure 1 presents an overview of the communications between the supervisor and workers. The scene information and rendering parameters, such as image quality and size, are read from the Studio

file by the supervisor and transferred via data messages to the workers. Control messages are utilized to synchronize the distribution of other resources. Through control messages, the supervisor transfers the network location of the Model and Material resource files to the workers for access via a Network File System(NFS). The Model and Material resources do not need to be manipulated by the supervisor as the Studio file does, therefore, the worker nodes read them directly via NFS. NFS is inherently well suited for efficient, simultaneous access by workers for reading the Model and Material resource files. NFS employs

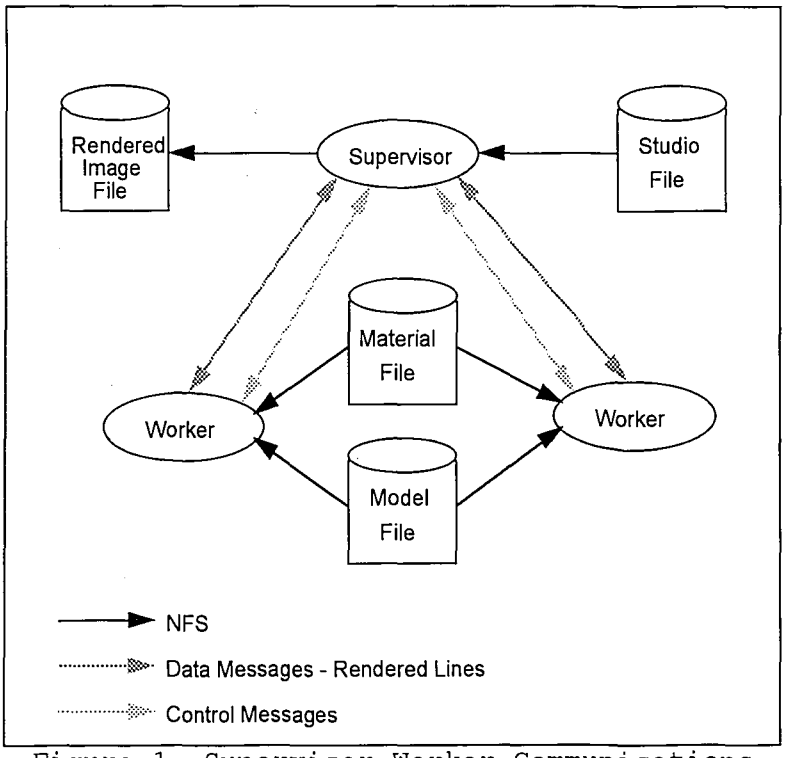


Figure 1. Supervisor-Worker Communications

multiple processes and disk caching in order to handle simultaneous disk I/O efficiently.

After distributing scene resources, control messages between the supervisor and workers coordinate the assignment of scan lines to render. The supervisor sequentially assigns individual lines to worker nodes as requested by the workers. Completed image scan lines are accumulated locally on the worker node and transferred to the supervisor only after all lines have been rendered. Completed scan lines are transferred to the supervisor at the end of current processing by the worker for two important reasons. First, by transferring the completed lines at the end, they can be transferred together as a large data stream. Transferring a worker's set of rendered lines as a single data stream avoids the typical startup latency of sending multiple messages individually. Secondly, overall image generation performance can be increased by not intermixing control and data messages to be handled by the supervisor. Workers would be idle, waiting for line assignments, as other workers transferred completed lines to the supervisor. Actual implementation supports the maintaining of only control messages during image rendering and transferring completed lines as a data stream, following the completion of all lines. By collecting all rendered lines, the supervisor can easily detect missing lines, thus, determine that a worker has failed. Missing lines can then be

reassigned to all remaining workers. The detection and reassignment of the missing lines of an incomplete image provides a simple, yet robust, fault tolerance for Mirage.

The supervisor's coordination and distribution of the image generation process is simplified by the fact that all workers maintain all the resources required to generate any portion of the final image. The partitioning of the image generation computations simply involves the assignment of the next sequential scan line remaining to be rendered to the requesting worker node. Additionally, once a failure is detected, as described above, recovery involves the reassignment of any lines not collected by the supervisor after a reasonable time interval. Some distributed image generation techniques discussed in the literature partition the data and computations to particular nodes[2][3]. These schemes may gain a minimal increase in performance over the presented model, but this gained acheived at the cost of increased complexity in identifying and resolving failures. The failure of individual workstations of a cluster in today's environment are not uncommon. With individual users controlling the availability of their own workstations, reboots are not uncommon. These interruptions in today's computing environments further stresses the advantages of applications designed with fault tolerance strategies.

A particular advantage of Mirage is its implementation utilizing standard, highly portable software architectures.

Communications between the supervisor and workers are implemented using Berkeley's socket interface and TCP/IP for the network transport protocol. The modules are coded to ANSI C standards to facilitate porting the software to different hardware platforms. This implementation encourages the use of this image generation system in networks of heterogeneous workstations. Additionally, clusters of workstations with varied computational capabilities are well suited for the presented distributed processing system due to its method of partitioning tasks. Mirage dynamically assigns scan lines to workers as they are completed. The worker nodes will receive computational tasks dependent on the completion of previously assigned tasks, therefore, each worker will be used to its capacity. In a heterogeneous environment, workstations will simply contribute toward the generation of an image according to their individual computational capability. Thus, Mirage's distribution strategy performs equally well in both clusters with homogeneous and heterogeneous workstations.

3.1. Supervisor

The supervisor is responsible for partitioning the computations for image rendering and for assembling completed scan lines from the workers into a final photo-realistic image. As presented by the supervisor-worker model, the supervisor does not actually take part in any of

the computations, rather it controls the distribution and collection of the computations. Further, the supervisor takes a passive role in distributing the computations to be completed by assigning tasks only at the request of a worker. In Mirage, the supervisor manages the distribution of the resources required for image generation, distributes individual scan lines to be rendered, and assembles results into the final image.

The supervisor process is user-activated with the rendering parameters and the series of scenes specified. A user starts the distributed image generation process by initiating the supervisor process on a network which has at least one worker process active. Due to the low computational requirements of the supervisor, a worker may execute on the same node as the supervisor process. At least one worker must be active for the supervisor to begin the distribution of the image generation. Worker nodes may be located anywhere, as long as the supervisor can establish a reliable communication channel and the worker can utilize NFS to access the required file resources.

3.1.1. Supervisor-Worker Communications

A crucial component of any distributed application is a reliable communication mechanism. The supervisor and worker of Mirage, communicate over a connected, reliable transport implemented by the use of Berkeley's socket interface and

the TCP/IP protocol. The socket interface allows a reliable communication path to be established between two processes on the same or different workstations on a network. Even though the socket connections behave like streams, allowing any amount of data to be passed in either direction, Mirage communicates via a defined message structure. By encapsulating all communications within messages, the same communication modules can be used by both the supervisor and worker processes. The message structure is designed to be flexible and efficient in handling both control and data messages over the connection. Figure 2 details the message structure used in the supervisor-worker protocol.

```
struct {
      void    *msgPtr0;            /* pointers for accessing data   */
      void    *msgPtr1;            /* maintained as lists by super  */
      int     msgIndex0;           /* integer vals for control msgs */
      int     msgIndex1;
      int     msgType;             /* message type                  */
      int     msgLen;              /* length of message in bytes    */
      union {
        char     data[MAX_SIZE]    /* message data section          */
        STUDIO   studio;           /* scene info & parameters       */
        GROUP    group;            /* transformation & animation    */
      } msgData;
} TCP_MSG;
```

Figure 2. Message Data Structure

Data and control messages are distinguished by the field in the message header that specifies the length of the data following the header. A zero length data specified in the header indicates a control message. Utilizing this scheme

to distinguish control messages allows efficient message handling functions to be implemented that transfer only the amount of data required. This message scheme allows the establishment of an efficient protocol between the supervisor and workers based on message types. The supervisor establishes and maintains connections to each worker for the duration of the image generation process. The connection information is maintained in a table by the supervisor that contains communication data for each established connection. Socket identifiers, workstation node names and addresses, and worker's process states are maintained in the communication table. Figure 3 contains the message types and descriptions of the supervisor-worker protocol.

INVALID	Signals that an invalid message type received
ECHO	Echo message type used during testing phase
CLOSE	Indicates completion of entire series of images
ACTIVATE	Activation signal sent to all connected workers
DONE	Signals the completion of current image
WAIT	No more scan lines to assign
RESUME	Resume rendering due to a worker failure
REQ_DSTUDIO	Request Studio parameters from supervisor
SND_DSTUDIO	Send Studio parameters to worker
REQ_MOX_PATH	Request Model File network location
SND_MOX_PATH	Send Model File network location to worker
REQ_MAX_PATH	Request Material File network location
SND_MAX_PATH	Send Material File network location to worker
REQ_FRST_GROUP	Request first transformation group from supervisor
SND_FRST_GROUP	Send first transformation group
REQ_NEXT_GROUP	Request next transformation group from supervisor
SND_NEXT_GROUP	Send next transformation group
REQ_LINE	Request scan line to render
SND_LINES	Signal the sending of completed lines to supervisor
SND_READY	Signal supervisor's acceptance of transferred lines

Figure 3. Supervisor-Worker Message Types

Synchronization between the supervisor and the workers is required during certain points in the image generation process. Worker nodes cycle through three distinct phases during the image generation process. Figure 3 illustrates the processing phases of the worker and the supervisor message types used to change phases. Specific messages from the supervisor determine the phases of the worker process. As illustrated in Figure 4, a point of synchronization is before the start of the rendering of an image. Each worker awaits an activation message from the supervisor before initiating the image generation process. This synchronization point allows worker processes to be continually active within the workstation cluster while waiting for an image to process.

Synchronization is particularly useful in detecting common failures in the presented distributed model. The supervisor detects the failure of a worker by recognizing the absence of rendered scan lines once all active workers transfer their completed lines. Worker nodes transfer all their completed lines after a request for their next task which results in a signal from the supervisor indicating that all image scan lines have been assigned. Following the transfer of rendered lines to the supervisor, the workers must wait for the supervisor to determine if it receives all of the assigned scan lines. If the supervisor does not receive all scan lines within a reasonable time interval, a

failure is assumed and the missing lines are reassigned. For Mirage a reasonable time interval of 10 seconds was utilized in worker failure detection. Therefore, each worker synchronizes at the point after which it has rendered and transferred the lines assigned to it. At this point, worker nodes await a signal from the supervisor indicating the rendering of reassigned lines following a worker failure, or the end of processing for the current image.

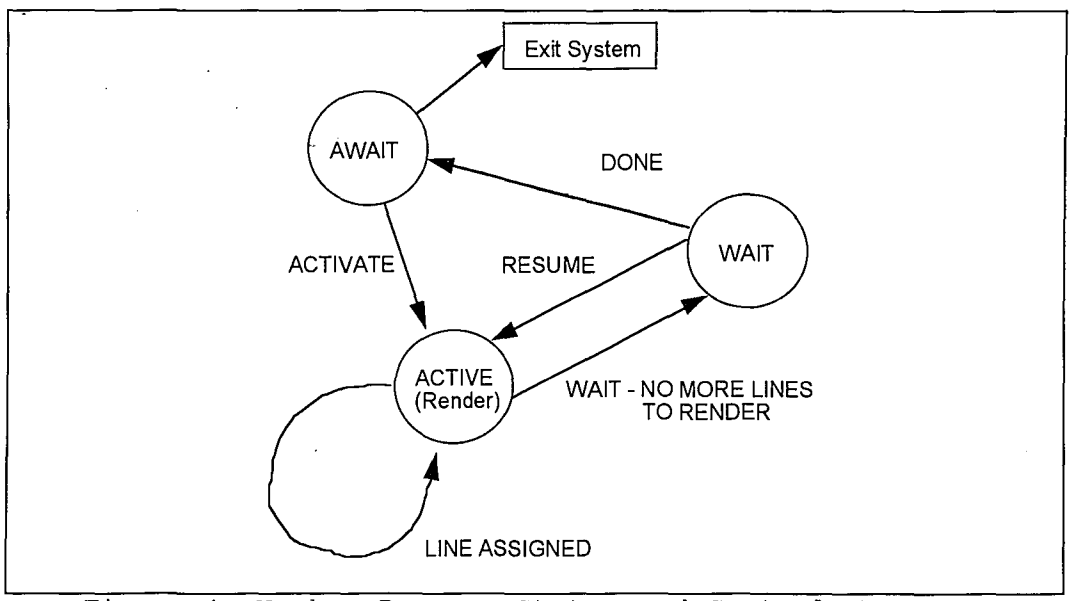


Figure 4. Worker Process States and Control Messages

3.1.2. Supervisor Implementation

An advantage of Mirage is the simple implementation of the supervisor. The supervisor is implemented as a single process and not a system of spawned child processes as is typical in supervisor implementations. Typical supervisor or server implementations utilize spawned child processes to

handle messages received from clients. The supervisor in the presented image generation system requires very little computation to respond to worker messages due to the large percentage of messages simply assigning sequential scan lines for rendering. Therefore, a single process implementation of the supervisor component is capable of sustaining an effective worker message throughput. By implementing the supervisor as a single process, the usual system dependent constructs, such as semaphores and shared memory, can be avoided.

Figure 5 details the pseudo code for the supervisor implementation. The supervisor initializes a communication log and a statistics log. The communication log records all messages received and sent through the supervisor and is used as an aid in debugging the distributed system as described below. The statistics log captures elapsed rendering times, worker contributions, and detailed rendering data, such as the number of object intersections. The statistics log helps determine the optimal rendering parameters and the number of workers for a particular image generation.

Following the initialization of the log files, the supervisor reads and organizes the scenes contained in the Studio resource file. The scene data and transformations read from the Studio file are stored in an ordered list in preparation for transmittal to workers. The supervisor

```
initialize communication log & statistics
read Studio file for scenes & transformations
open sockets
initialize active worker connections table
while (current scene < last scene)
     initialize scan line table
     ensure all resource files exist and are valid
     send activation signal to workers
     while (not all scan lines received for current image)
          select on communication input
          if (time-out on input) and (no workers active)
               exit because no workers are active
          if (time-out on input) and (all workers are waiting)
               send resume message to workers
               continue
          log incoming message to communications log file
          if (connection request)
               establish reliable connection to worker
               update active worker connections table
          process incoming message according to type
          log outgoing message to communications log file
     write image to disk
     send done message to all connected workers
send close message to all connected workers
```

Figure 5. Supervisor Implementation

initializes communications by opening sockets and preparing to accept connections from workers at a dedicated port number. The table used to maintain the data and statistics on active worker connections is initialized in conjunction with the setup of the communications.

The main loop controls the rendering of a series of images. A series of images are generated from a start scene to an end scene. Prior to distributing the processing for each scene, several functions are performed by the supervisor. The supervisor initializes a table which contains the status of every scan line in the desired image. Values are assigned based on the status of particular scan lines. Status values are set based on the assignment of the

line to a worker, receipt of the completed line, and the reassignment following a worker failure. Next, the supervisor validates and verifies the existence of all of the resource files required for image generation. Following the successful verification of the necessary resource files, the supervisor sends an "activation" signal to all workers. The activation signal initiates the workers to begin the process for generating the specified image.

The inner loop of the supervisor process represents the main message processing loop for the current image. Messages are processed from the workers until all completed scan lines for the current image has been received. Worker messages are received by multiplexing all of the active connections contained in the communication table though the use of the select system function[9]. If no input is detected for a reasonable time interval two possible failures may have occurred. The first failure checked is the situation where there exists no active worker nodes. With no worker nodes active, the image cannot be completed indicating a fatal condition that aborts the image generation process. The second possible failure detected is the situation where all active workers have transferred their completed lines to the supervisor and are waiting, but the supervisor detects missing scan lines. The second failure situation is resolved by sending a "resume" message to all workers and reassigning the missing lines.

When input is detected by the select system function, a message is read from the specified socket. The message read is logged in the communication log file. The input message is then processed according to its type. Following the processing of the worker's message, a reply message is formulated and sent. The outgoing message is tracked in the communication log. Connection requests are recognized by detecting input on the supervisor's socket assigned a specific port number. Connection requests are accepted to establish a reliable communication path between the requesting worker and the supervisor. The active communication table is then updated to reflect the establishment of a new connection to a worker.

Following the completion of the current image, as evidenced by the collection of all rendered lines from the workers, the image is written to disk. After the supervisor writes the image to disk, a "done" message is sent to all worker nodes. The supervisor and worker now prepare for rendering the next image in the current series. When all images of the current series have been rendered, a "close" message is sent to all workers. The "close" message indicates that the image generation is complete and workers can free all resources acquired for the current series of images.

3.1.3. Distributed Debugging Methods

Debugging distributed systems usually proves to be a difficult task. Mirage incorporates several techniques to facilitate the debugging of the supervisor process and associated communication protocol. The invocation of the supervisor provides a switch to enable the detailed display of the actions performed by the supervisor during execution. During execution this display allows the user to monitor the results of particular functions and operations as they execute. Secondly, the supervisor logs all incoming and outgoing messages to a communication log file. The communication log is examined to verify the correctness of the messages passed between the supervisor and worker. Additionally, an interactive utility was implemented to verify the correct operation of the supervisor. The interactive utility allowed the testing of the supervisor implementation and design. Specifically, the utility allowed the interactive testing and debugging of the supervisor's message handling. The utility allowed different message types to be sent to the supervisor and displayed the response messages interactively.

3.2. Worker

The worker component of Mirage performs the image generation computations as directed by the supervisor. The worker ray-traces its assigned scan lines and stores them

locally until they are transferred as a group to the supervisor. Worker processes may be started manually, however, they are usually incorporated into the startup process of workstations. This automated startup of the worker process ensures that a designated number of worker nodes exist in the network. Incorporating the worker into the startup script also guarantees the execution of the worker following a workstation failure. While this technique allows the worker process to remain active continually, very little resources are consumed. The worker remains idle awaiting an activation signal from a supervisor. The execution of the worker requires no user intervention once started. All rendering parameters and scene data are provided by the supervisor.

There exists no inherent limit on the number of workers that may be active on a given cluster of workstations at anytime. As discussed in Chapter 5, performance issues dictate the optimal number of active workers for a particular image. Depending on image complexity, a large number of workers will actually degrade overall image generation performance due to the increased communication overhead. Due to the intensive computational resources required during the rendering of an image, only one worker process may be active on a single workstation.

The presented worker design includes no explicit scheduling strategy. Mirage utilizes worker nodes to their

potential with no regard to current workstation load. Scheduling strategies on the worker nodes could be employed to limit their availability for image generation to certain days and times as warranted by the particular computing environment. For evaluating the system's performance, maximum participation by the workers is desired, therefore, no scheduling strategy is implemented. Although no explicit scheduling strategy is implemented by the workers, a passive load balancing occurs as the supervisor dynamically assigns scan lines. Workers with more computational power will contribute a greater percentage of the overall image.

3.2.1. Worker Implementation

The implementation of the worker follows a modular design strategy. The modular design of the worker allows the worker to use the same basic message handling functions utilized by the supervisor. Additionally, the design of the worker allows many modules to be used from the single-processor version. Specifically, the worker utilizes the same rendering engine as the single-processor version. A common rendering engine between a single-processor version and the worker component in the distributed model allows accurate performance analysis in regard to communication and distribution overhead. The pseudo code of the worker is presented in Figure 6. After a general initialization, the worker establishes a User Datagram Protocol(UDP) socket. The

UDP socket is used to receive the activation signal broadcasted by the supervisor at the start of a series of images. The worker receives an activation signal in one of two ways. If the worker is not already connected to the supervisor from a previous image rendering, the UDP connection will be examined for the broadcast signal. Worker nodes that are already connected to the supervisor will await an activation signal across an already established connection. If the worker is activated via the broadcast to the UDP socket, the worker then establishes a reliable connection to the supervisor for subsequent communications. Following the receipt of an activation signal from the supervisor, the worker acquires the scene information and rendering parameters for the current image. Once the general rendering information for the current image is received from the supervisor, the transformations and animations are requested. Next, the network location of the Material and Model resource files are acquired. After determining the network location of the Material and Model resources, the worker utilizes NFS to load the scene's materials and objects. Objects contained in the scene are then transformed according to the animation parameters previously received from the supervisor. The objects composing the scene are organized into an octree to speed up the ray-tracing algorithm as discussed in the next chapter on the rendering algorithm enhancements.

```
initialize global variables
open UDP port for receiving activation signal from supervisor
while (1)
     if already connected to a supervisor
          await activation signal from supervisor on connected socket
     else
          await activation signal on UDP socket for broadcast
          establish reliable connection to supervisor
     get scene information from supervisor
     get first group transform for scene from supervisor
     while not last group transform
          get next group transform from supervisor
     get Material file network location from supervisor
     open and read materials into tree structure
     get Model file network location from supervisor
     open and read objects
     transform all objects according to group transforms
     organize objects and build octree
     request scan line to render
     while done message not received
          render assigned scan line
          request a scan line to render
          if wait message received
               notify supervisor ready to send completed lines
               await ready signal from supervisor
               stream completed lines to supervisor
          receive message from supervisor
```

Figure 6. Worker Implementation

The inner loop of the worker implementation controls the rendering of the assigned scan lines. Line assignments are requested from the supervisor. Assigned lines are rendered in turn and stored in a local cache of completed scan lines. If the worker receives a "wait" message instead of a line assignment, the worker realizes that all lines have been assigned by the supervisor and there are no more to render. After receiving a "wait" signal, the worker sends the completed lines from its local cache to the supervisor. The completed scan lines are sent to the supervisor as a continuous data stream. By transferring the

completed lines as a data stream, the overhead of sending the lines as individual messages is avoided. Subsequent to sending the completed scan lines, the worker continues to await a message from the supervisor indicating that the current image is done. At this point, either a "done" or "resume" message is received from the supervisor. A "resume" message indicates that a failure has occurred and lines need to be reassigned and rendered. Upon receiving a "resume" message, the worker repeats the cycle of requesting and rendering lines. A "done" message indicates that the current image is finished and the worker now awaits an activation signal in order to begin the next image.

The worker maintains the reliable connection to the supervisor throughout the generation of a series of images. By maintaining the connection, the worker eliminates the latency involved in establishing a new connection for each image rendered for the same supervisor. The connection between the supervisor and worker is dismantled after the rendering of the series of images is completed. The completion of a series of images is indicated by receipt of a "close" message from the supervisor.

Chapter 4. Rendering Algorithm Enhancements

The technique of ray-tracing for producing photo-realistic images is inherently computationally intensive. Several enhancements have been developed to increase the performance of the basic ray-tracing algorithm. A significant amount of literature has been devoted to increasing the performance of image rendering based on the technique of ray-tracing[20,21,22]. The ray-tracing algorithm utilized by the presented image generation model incorporates many of the recent improvements. A few unique enhancements used by the presented image generation model are presented. Specifically, an enhancement to the octree method of organizing objects in a scene is discussed. An "adaptive octree" is presented that benefits from the standard octree method, but requires less memory resources. This chapter discusses the enhancements to the basic ray-tracing technique utilized by the presented model that gives the most performance benefits.

4.1. Shadow Cache

Calculations for determining the shadows in a particular image consumes a significant portion of computation time. Typical images devote 10-20% of computation time to calculating shadows. Clearly, a decrease in required shadow computations will impact overall

image generation performance. Shadows are detected by firing shadow rays from the current object intersection point toward a light source. If the shadow ray reaches the light source without intersecting another, opaque object, the original object is not in shadow. However, if the shadow ray intersects an opaque object on its way toward the light source, then the object must be in shadow. If the object intersected by the shadow ray is not completely opaque, such as glass, a partial shadowing results due to the attenuation that light rays would experience by passing through the intersected object.

It is very costly to calculate the intersections of the shadow ray as it tracks through the entire scene. A shadow cache is employed to eliminate the firing of many shadow rays[10,11]. In theory, if a point on an object is computed to be in shadow by a particular object, adjacent points on that object have a high probability of being shadowed by the same object. The shadow cache stores the last opaque object which caused a shadow. Before tracking the shadow ray through the entire scene, the object in the shadow cache is tested for intersection with the shadow ray. If the shadow ray intersects the object in the cache, a shadow exists and no further calculations are performed. If the shadow ray does not intersect the cached object, shadow ray calculations proceed as normal. The shadow cache is further

expanded to include entries for shadow rays of different recursion depths.

In a scene with reflection and transparency rays being spawned, shadow rays occur at different depths of recursion. The shadow cache is examined according to the recursion depth of the current shadow ray being computed. This depth ordering in the shadow cache expands the likelihood that one of the cached objects casts a shadow for the current shadow ray. To further improve performance of the shadow cache, a separate shadow cache is maintained for each light source. The hit ratio, and thus performance benefit, of the shadow cache is directly proportional to the number of lights and complexity of the scene being rendered. In the sample images rendered for performance analysis, an average hit ratio of 10% for the shadow cache was experienced. At first this hit ratio might seem low, however, it must be noted that the majority of shadow rays cast reach the light source, indicating no shadow exists. The shadow cache only attempts to short circuit shadow ray tests that eventually result in the existence of shadows.

4.2. Adaptive Octree and Object Organization

The time required to ray-trace an image can be significantly reduced by the efficient organization of the objects in the scene. An efficient organization allows only the objects that could possibly intersect the traced ray to

be tested, rather than every object in the scene. Ray-object intersection tests constitute a major portion of the computations utilized in the ray-tracing technique. Approximately, two-thirds of the computations performed by the ray-tracing algorithm are devoted to determining ray-object intersections. The octree, a data structure commonly used in ray-tracing algorithms, efficiently organizes the objects of a scene for intersection testing. In an octree scheme, the three-dimensional space of a scene is uniformly sub-divided. The objects of the scene are placed into the sub-divisions depending on their spatial location. A ray is tracked through the octree as it passes from one sub-division to the next. Objects within the octree sub-divisions are tested for intersection as the ray moves forward through the octree.

Several enhancements to the basic octree strategy of object space organization are presented. First of all, the octree organization is made "adaptive". During octree construction, space sub-division only occurs when the density of objects in a particular area exceeds a maximum threshold, rather than uniformly. Object space is recursively sub-divided until objects are partitioned in a manner that yields a number of objects per octree sub-division that is less than a threshold value. Additionally, the recursion of the octree sub-division is limited based on the nature of the scene in order to conserve memory where

further sub-division yields little benefit. The "adaptive octree" recognizes that certain sub-divisions do not markedly decrease the density of objects in certain areas, therefore, further sub-division is avoided. This adaptive approach yields advantages of decreased memory usage and more efficient object organization as compared to the uniform sub-division implemented by most octree strategies. The algorithm for organizing the objects of a scene with the presented "adaptive octree" is illustrated in Figure 7.

```
octree.level= 0
octree.numObjects= all objects in the scene
root octree= extent of entire scene
push root octree
while stack not empty
     octree= pop stack
     if (octree.numObjects > max objects allowed per octree)
        and (octree.level < max recursion level)
           subdivide octree into 8 regular children
           partition all objects from octree into children nodes
           increment octree.level
           for i= 1 to 8
                 if (child[i].numObjects > max objects per octree)
                    and (child[i].numObjects < octree.numObjects/RES)
                       push child[i]
```

Figure 7. Adaptive Octree Build Algorithm

In addition to efficient object organization, a significant performance benefit may be realized by enhancing the algorithm for tracing a ray through the octree for determining its nearest intersection point. The octree is recursively examined to determine the objects intersected by a ray as it traces through a scene. The sub-divisions of

the octree through which the ray passes are examined starting at the root of the octree. The ray is traced from one sub-division to the next until a leaf node is encountered. The leaf nodes contain a list of objects which lie within the leaf sub-division. Once a leaf node is encountered, all objects within the leaf node are tested for intersection with the ray. If an intersection occurs, the algorithm exits having found the nearest, intersected object, otherwise the tracking continues through the octree. It is important to note that the octree sub-divisions are searched in the order of forward travel of the ray with the sub-division closes to the ray origin being searched first.

A key enhancement possible with the octree method of object organization lies in the algorithm for tracking a ray through the sub-divisions of the octree. The algorithm utilized by the presented image generation model reduces the number of calculations required for tracking a ray through

```
if ray does not intersect octree root then exit
push octree root
while stack not empty
     octree= pop stack
     if (octree.child[0] is NULL)
          test ray intersection with all objects in octree
          if (ray intersects at least one object)
               return the closest intersected object
     else
          determine intersection of ray with octree subdivision planes
          determine trace of ray through children subdivisions
          push subdivisions on stack according to trace order
```

Figure 8. Tracking Ray through Octree

an octree and uses a local stack for implementing the recursive search. The pseudo code for the enhanced algorithm for tracking a ray through an octree is presented in Figure 8. The enhanced algorithm for determining the order in which the sub-divisions of an octree are intersected is explained easier in two-dimensions. Figure 9 represents the intersection of a ray with the four sub-divisions of a quadtree(two-dimensional equivalent of an octree).

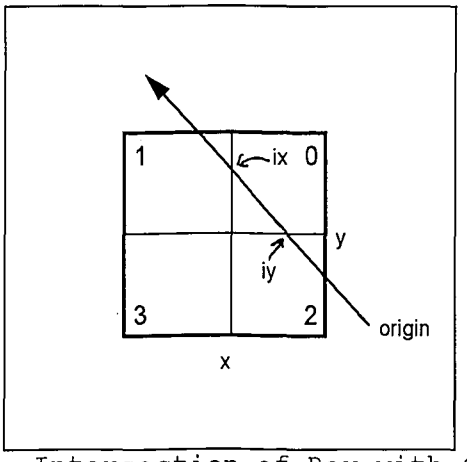


Figure 9. Intersection of Ray with Quadtree

The sub-divisions are numbered in such a way as to facilitate the tracking of a ray from one sub-division to another, depending on the ray's direction. Movements between sub-divisions in the X direction occur by exclusive-or'ing the current sub-division with 1(0001 binary). Conversely, movement between sub-divisions in the Y direction are achieved by exclusive-or'ing the current sub-

division with 2(0010 binary). Tracking a ray through the sub-divisions requires the relative direction of ray travel to be determined. The intersection distance between the ray and the sub-division planes, X and Y, are calculated. The plane intersection distances are compared to determine the order in which the sub-divisions are encountered. In the figure, ix and iy are the distances from the ray origin to the sub-division planes. Since iy is less than ix, it can be deduced that the ray must travel in the Y direction first, followed by the X direction. In order to track the sub-divisions intersected by the ray, the start sub-division is determined. In Figure 9, the start sub-division is 2. Since it was determined that the ray travels through the sub-divisions in the Y direction first, the start sub-division, 2, is exclusive-or'd with 2, resulting in 0. It is clear from Figure 9 that the sub-division labeled 0 is in fact the next sub-division intersected by the ray. Next, sub-division 0 is exclusive-or'd with 1, for movement in the X direction, to get the sub-division labeled 1. Therefore, the order in which the ray tracks through the sub-divisions is 2, 0, 1. The sub-divisions intersected by the ray are pushed onto the local stack in reverse order. The sub-divisions are pushed onto the stack in reverse order so that the sub-divisions closest to the ray origin will be examined first. The described algorithm for tracking a ray through the sub-divisions of an octree is easily extended to three-

dimensions. In three dimensions, the Z sub-division plane is utilized and the sub-divisions are labeled as represented in Figure 10.

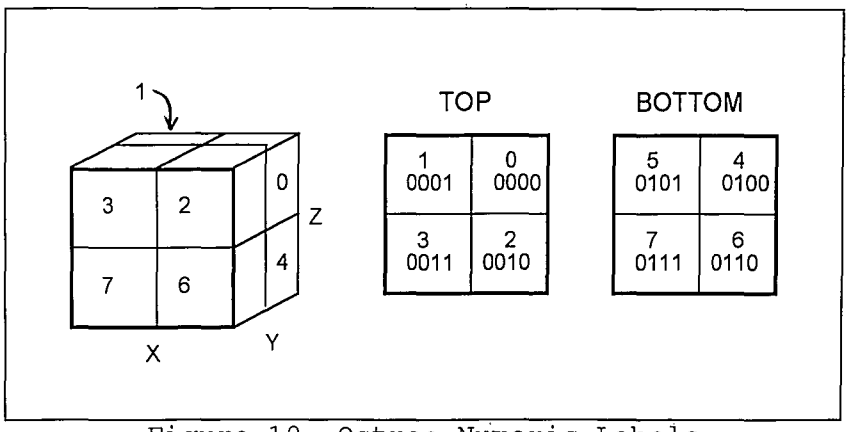


Figure 10. Octree Numeric Labels

The presented algorithm for tracking a ray through the sub-divisions of an octree increases overall performance by limiting the number of floating point calculations required and using a local stack for implementing recursion. Additionally, an enhancement was employed for eliminating the redundant intersection tests for objects that span more than one octree sub-division. Objects that span more than one octree sub-division are tested for intersection multiple times with the same ray as it tracks through the octree. A mailbox scheme eliminates the multiple intersection tests of the same ray and object[12]. The mailbox scheme involves tagging objects with a unique numeric identifier associated with the ray the first time the object is tested for intersection. Before the ray-object intersection test is

performed, the ray identifier and object identifier are compared. If the identifiers are equal, indicating that the object has already been tested with the current ray, the intersection test is not performed.

4.3. Optimized Object Functions

Particular calculations on objects and rays are used heavily in the technique of ray-tracing. Specifically, the intersection and normal calculations of rays with objects are used extensively. The discussed image generation system utilizes optimized intersection and normal calculation functions developed individually for each object type. Overall performance is enhanced by binding pointers to the optimized functions within the object data structure. By maintaining pointers to the optimized functions within the object structure, the overhead of branch and switch constructs can be avoided in the ray-tracing implementation. The optimized functions are accessed directly as objects are manipulated by the ray-tracing algorithm.

Chapter 5. Performance Analysis

Ideally, a task that is distributed among a cluster of workstations should yield a performance speedup equal to the number of participating workstations. For most distributed applications a linear speedup proves unattainable due to the overhead incurred in distribution and collection tasks. Several factors specific to the presented model limit the performance gained by distributing the image generation amongst a workstation cluster. Additionally, certain image characteristics impose an upper bound on performance speedup regardless of the number of workstations utilized. An upper bound on the number of workstations that can be utilized in the presented system exists due to the employed distribution strategy. Since the distribution scheme involves the assignment of individual scan lines to the workers, no performance gain can be realized by utilizing more workstations than the vertical resolution of the final image. This limitation on the number of participating workstations does not adversely affect the typical image generation process. Typical images consist of at least 500 lines vertically, which does not impose an artificially low bound on speedup. However, the presented distribution strategy imposes an overall maximum speedup which can be achieved, equal to the number of vertical scan lines of the final image.

The presented image generation model incurs distribution overhead, attributed to the cost of communicating with the worker nodes. The communication cost consists of message passing for assigning scan lines, sending render parameters, and collecting completed lines. Processing messages for the successful distribution of the image generation process requires additional computations by both the supervisor and worker. Therefore, the communication cost which results indirectly from distributing the image generation affects the performance speedup achieved.

Another factor that affects the performance of the distributed model is the setup time required by each worker. Setup involves the time to acquire the scene resources, namely the objects and materials, and the time to organize the objects into an octree. The setup time for the generation of an image relates directly to the complexity of the scene being rendered. The complexity of a particular scene is defined as the number of objects contained within the scene. Therefore, complex scenes, containing thousands of objects, require longer setup times than less complex scenes containing only a few objects. By increasing the number of participating workstations, a corresponding increase in the setup time required for each worker occurs. The setup time required per node imposes an upper bound on the performance speedup that can be realized. Workers can

be added to the image generation system until the point at which an additional worker increases the setup time per node to exceed the render time per node. Incorporating additional workers into the distributed model past the point of saturation further increases the setup time per node so that a diminishing speedup is realized. Note, the combination of the three factors discussed directly affects the setup time required per node. The performance of the network file system, image complexity, and the number of workers affect the setup time per node.

Although the presented image generation system does not achieve a linear speedup, its performance speedup is proportional to the number of worker nodes when utilized within the discussed bounds. The performance of the presented system is examined with regards to the distribution overhead and the setup time required per node. Specifically, the effects of image complexity and number of workers are analyzed in relation to their effect on overall performance.

The distribution overhead of the system is examined with regard to image size, thus the amount of data distributed and collected, and number of worker nodes. The performance of the distributed image generation system was benchmarked against a single-processor implementation which utilized the same basic rendering engine and identical input resource files. The single-processor version provided a

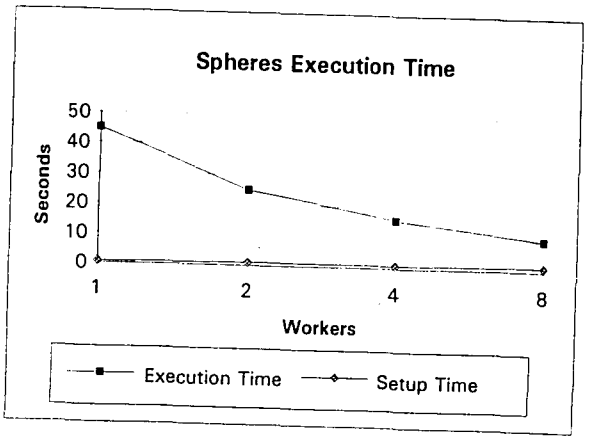
basis for performance comparison with the distributed model since it utilized the same rendering algorithm. The image generation system was tested on a cluster of RISC-based workstations interconnected via Ethernet. Various configurations, in terms of number of workers, of the image generation system were utilized. The tested configurations consisted of one supervisor and 1, 2, 4, and 8 worker nodes.

For each of the configurations, performance speedup was measured with three test scenes. The test scenes were utilized for their wide range of complexity and varied image size and are representative of a broad spectrum of images for evaluating the image generation system. A scene consisting of seven spheres of various material characteristics was utilized for evaluating system performance of simple scenes. Color plate 1 contains the image generated by the spheres scene. The second scene analyzed consists of the classic Utah Teapot model[13]. The teapot contains 1500 phong-shaded polygons and two light sources. The Utah teapot model was utilized because of its medium complexity and its informal acceptance as a benchmark for image generation. The image generated from the teapot model is presented in color plate 2. The third scene utilized for performance analysis consisted of a complex model of spheres generated by a fractal, known as Menger's Sponge[14]. The sponge model utilized in this scene consisted of 18,000 spheres. Additionally, two polygons, to

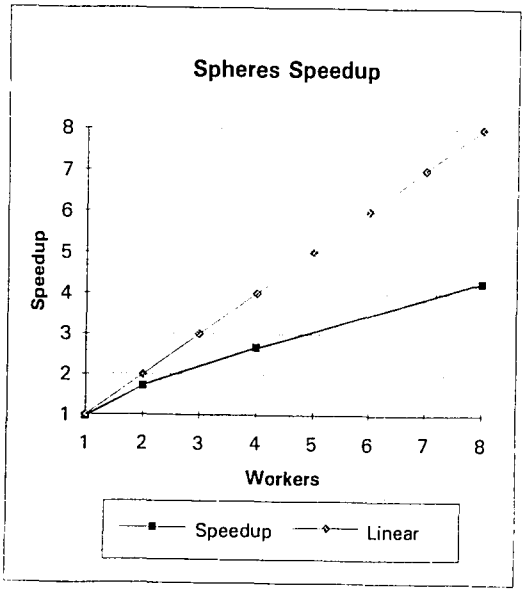
highlight the shadows cast, and two light sources were placed in the test scene. Color plate 3 contains the image generated from the sponge model.

4.1. Spheres Scene

The spheres model, the least complex scene containing only several objects, required the least amount of time to render. The single-processor implementation required 43 seconds to render the image with a setup time of only 1 second. Distributing the sphere scene with the presented image generation system yielded a decrease in rendering times as indicated in Figure 11a. Figure 11a also illustrates the setup time per node for each configuration of workers. The setup time per node increased from 1 second for the single worker configuration to 1.5 seconds for the eight node configuration. This change in setup time as larger worker configurations were utilized represents a 50% increase in setup time per node. Figure 11a highlights the fact that, as rendering times decrease and setup times increase for larger workstation configurations, performance gains diminish. Figure 11b contains the observed speedup of the distributed generation of the spheres model in relation to a linear speedup. The speedup of the distributed system is calculated in comparison to the execution time of the single-processor implementation. For the spheres scene a significant divergence from a linear speedup was observed.



(11a)



(11b)

Figure 11. (a) Spheres Distributed Execution Time and (b) Speedup

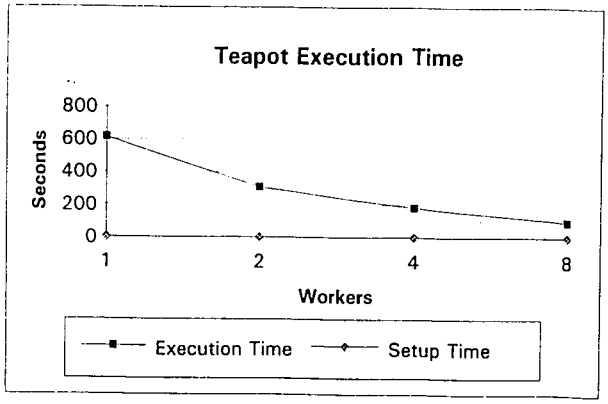
This divergence in performance speedup is attributed to the simplistic nature of the scene and thus, its short rendering times. Since the spheres model was not complex and contained only a few objects, the render time compares closely to the setup time and communication overhead.

Therefore, a lower performance gain was observed, resulting from the fact that the setup time and communication overhead form a greater percentage of the overall execution time. The speedup of 4.26 measured for the spheres scene was the lowest of the three test scenes for configurations of eight workstations. The spheres scene data implies that less complex images benefit from smaller configurations of workstations but, a lower overall speedup in performance is observed.

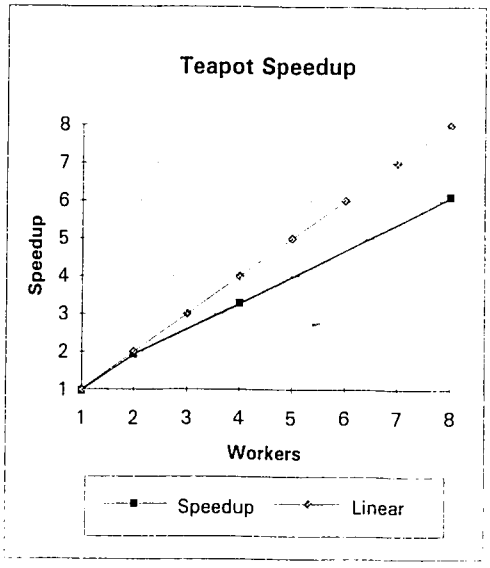
4.2. Teapot Scene

The teapot model was rendered in 598 seconds with the single-processor implementation. Figure 12a illustrates the rendering and setup times for the teapot model utilizing the distributed image generation system with configurations of 1, 2, 4, and 8 workstations. As observed from Figure 12a, a significant decrease in rendering times occurred as larger configurations of workers were employed. In contrast to the spheres scene, the setup time did not exhibit a substantial increase as more workers were utilized. The setup time increased from 1.6 seconds for the single worker configuration to 1.9 seconds for the eight node configuration. Additionally, the setup time constituted only 0.26% of the render time for the single worker configuration and 2% for the eight node configuration. Figure 12a indicates the more substantial performance

benefit from distributing a scene of medium complexity, as exhibited by the teapot model. The greater performance benefit of the teapot model compared to the spheres model, is indicated by the slower convergence of the render time to the setup time as the number of workers increased.



(12a)



(12b)

Figure 12. (a) Teapot Distributed Execution Times and (b) Speedup

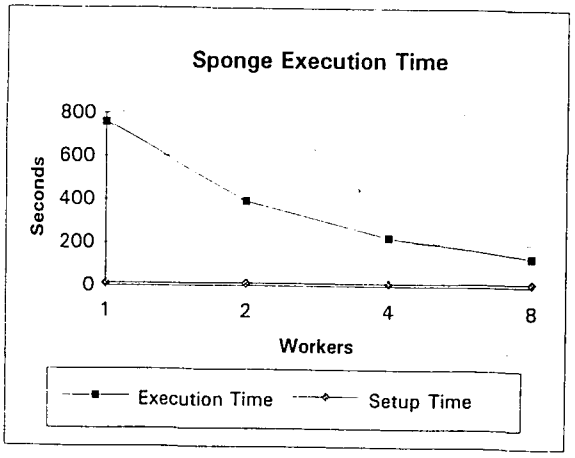
Figure 12b illustrates the speedup observed by distributing the image generation of the teapot model. The speedup observed for the teapot model approached the ideal, linear performance. The "near" linear performance can be attributed to the teapot model's relatively large render time compared to the small setup time per node. The steep slope of the observed speedup indicates that further increases in performance are attainable over the measured 6.1 for the eight node configuration. The teapot model exhibited the best performance speedup of the three test scenes evaluated with the distributed image generation model. The "near" linear speedup observed for the teapot model relates the balance between setup time, communication overhead, and image complexity.

4.3. Sponge Scene

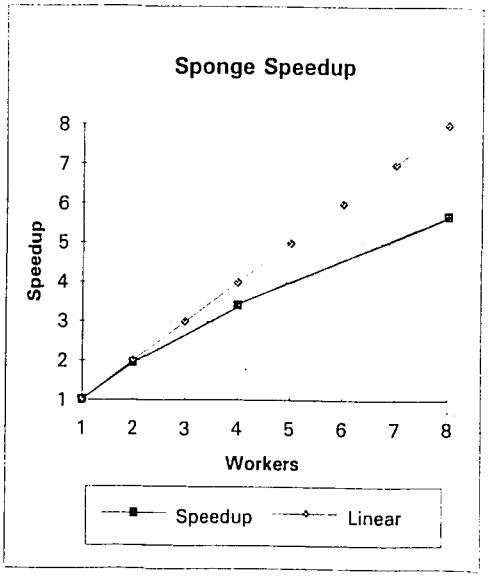
The sponge scene represents the most complex scene tested with the distributed image generation system. With more than 18,000 objects, the sponge model took the longest time to render with the single-processor implementation, requiring 760 seconds. The setup time for the sponge model was also the greatest at 8.5 seconds per node. Figure 13a contains the rendering times for the sponge model when distributed. The decrease in rendering times for the sponge model were very similar to the teapot model. However, the

setup time for the sponge model increased from 8.5 seconds for the single worker configuration to 15 seconds for the eight node configuration. This increase in setup time is in stark comparison to the teapot model's uniform setup time required per node. This dramatic increase can be attributed to the fact that the sponge scene contained in excess of ten times as many objects. Therefore, the network file system was unable to cache all of the information for the sponge model during the simultaneous access by the workers. Additionally, the setup time component accounted for 1.1% of the render time for the single worker configuration and 11% for the eight node configuration. This ratio of setup time to render time represents a ten-fold increase as compared to the teapot model. The increase in setup time indicates a faster convergence of render time and setup time, therefore, indicating a lower overall speedup as displayed in Figure 13b.

Figure 13b demonstrates the substantial speedup observed by distributing the sponge model. The speedup observed for the eight node configuration was 5.8. As Figure 13b indicates, the sponge scene stands to benefit from increasing the number of workers in excess of the eight node configuration. The speedup of the sponge model implies that the setup time plays a larger role in overall performance for scenes of increasing complexity.



(13a)



(13b)

Figure 13. (a) Sponge Distributed Execution Times and (b) Speedup

For the three test scenes a substantial speedup in performance was observed, indicating the efficiency of the presented image generation system. A speedup approaching linear was observed for the two more complex scenes. Low

communication overhead and minimal impact on the setup time per node was experienced for the tested configurations, contributing to the substantial performance increases. Communication overhead was minimal when compared to the computation time for even the simplest images. Additionally, the communication overhead is minimized by the fact that no complex task partitioning was employed for distributing the image generation. Although the setup time per node increased as larger configurations of workers were tested, the increase was not proportional to the number of workers added and not significant when compared to render times. This minimal impact on setup time per node is attributed to the stable performance of NFS in the test environment even under increasing levels of simultaneous access.

Although not linear, the performance of the distributed model achieved a substantial speedup due to its low distribution overhead. The observed speedup was proportional to the number of workers utilized while operating within the stated bounds. The overall performance of the presented system demonstrates its suitability for distributed image generation amongst interconnected workstations.

Chapter 6. High-Speed Switch

The effect of the network hardware used to interconnect the workstations was evaluated for the presented image generation system. The test environment contained two independent network architectures for clustering the workstations. Ethernet connected the workstations in typical local area network fashion, which provided the communication platform for the previous chapter on performance analysis. In addition to Ethernet, a high-speed switch provided a communication mechanism between the cluster of workstations. Each workstation interfaced with the switch through a copper wire cable. The twisted pair cable contained one unidirectional channel for receiving messages and a second unidirectional channel for sending messages from the workstation to the switch. This dual channel construction allows for messages to be sent and received from the switch simultaneously. The hardware architecture of the switch provided a high bandwidth and low latency, resulting in an efficient network mechanism for clustering workstations.

The switch provided support for the socket interface utilized by the distributed image generation system. The same workstation was assigned two different addresses in order to distinguish the network hardware used for communicating with it, either Ethernet or the high-speed

switch. Therefore, the address used to access a particular workstation determined the network hardware that was utilized to communicate with that workstation. This arrangement allowed the socket interface, implemented in the image generation system, to be utilized without modification except for the addressing of the workstations. An option was provided during the invocation of the image generation process that directed all communications between the supervisor and worker nodes to utilize the switch. The performance benefit afforded by the switch was analyzed by rendering the test models presented in the previous chapter. The test models were rendered utilizing the switch for communications within the eight node configuration. Figure 14 displays the overall execution times for the three models utilizing both Ethernet and the switch. Figure 14 indicates the negligible impact on performance experienced when utilizing the switch versus Ethernet.

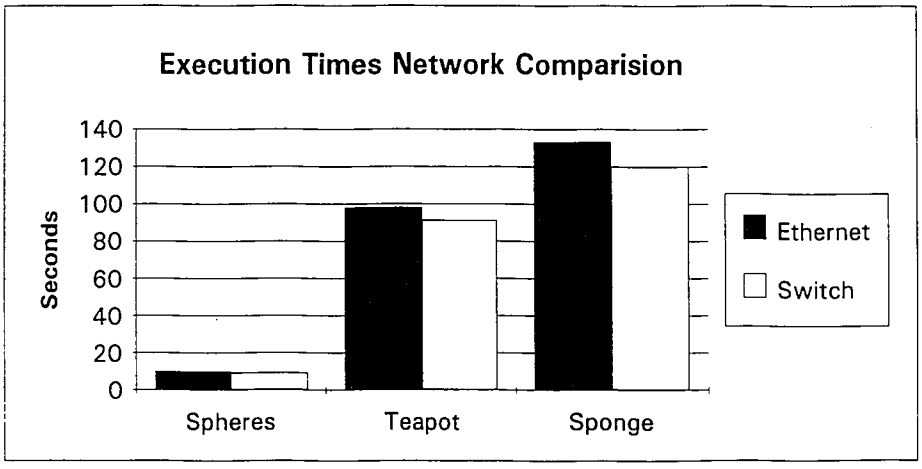


Figure 14. Execution Times for 8-node Configuration

There existed no significant difference in execution times when the switch was used for inter-process communication. Several factors contribute to the disappointing results when utilizing the switch hardware with the distributed image generation system. Even though the switch supports a socket interface, a proprietary, low-level interface provides better performance. The majority of distributed applications are designed to be portable, thus, the use of proprietary, low-level interfaces are not implemented, which ultimately results in a performance disadvantage. Likewise, the feature of portability was paramount in the design of the presented image generation system.

Secondly, the rendering of photo-realistic images with the ray-tracing technique is compute-bound rather than communications bound. The communication time comprises only a minimal percentage of overall execution during ray-tracing, therefore, increases in communication performance impact overall execution very little. Additionally, the small number of workstations in the tested configuration do not generate an excessive communication overhead during the rendering of images. Perhaps as larger configurations of workers were utilized, the switch would provide a greater performance benefit due to the increased communication overhead.

Thirdly, the protocol utilized by the supervisor and worker of the presented image generation system contained a majority of control messages. With the exception of the transfer of completed lines, communications between the supervisor and worker consisted of small control messages. A performance gain should not be observed due to the high volume of control messages, unless the startup latency for passing a message via the switch is much smaller than standard Ethernet.

Additional tests were conducted utilizing the switch in order to support the assumptions about its performance. Tests in which large data streams were transferred between workstations simultaneously were performed. Data stream tests were examined in order to calculate the performance of the switch during high-volume data transfers in contrast to series of control message transfers. Figure 15 illustrates the data rates measured when transferring large blocks of data between the workstations of a cluster. Figure 15 compares the transfer rates for Ethernet and the switch as the number of simultaneous transfers increases. The large data stream test, as indicated in Figure 15, implies the suitability of the switch over Ethernet for high-volume, simultaneous data transfers between nodes. The switch experienced significantly less degradation in performance as the number of simultaneous transfers were increased. The switch performed well in the transfer and routing of large

data streams simultaneously across the network cluster. The performance of the switch under these conditions suggest its applicability for communication intensive applications. The switch provides greater performance benefits for high-volume

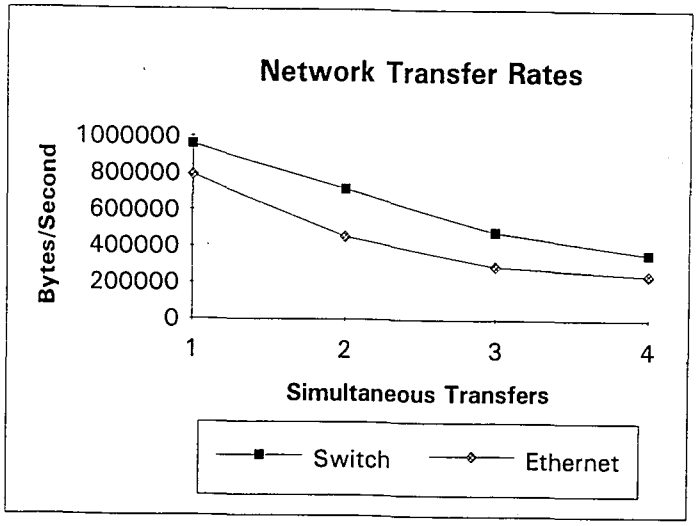


Figure 15. Comparision of Network Transfer Rates

data communications applications, such as digital video and video conferencing, rather than compute-bound tasks. As indicated by performance analysis, applications similar to the presented image generation system are not markedly affected by the underlying network architecture used for communications.

Chapter 7. Conclusion

The increase in interconnected workstations provides an environment that fosters the development of distributed applications. The image generation system presented in this thesis demonstrates the significant speedup of computationally intensive programs that can be attained by utilizing a cluster of interconnected workstations as a distributed processing platform. Its model can easily be generalized for similar, computationally intensive tasks. The speedup observed by the presented model indicates the performance gain possible for similar distributed applications utilizing a "Supervisor-Worker" architecture. Advantages of the presented distributed system are its scalability, portability, and ease of implementation for heterogeneous computer environments. Additionally, the presented image generation system required no modifications to individual workstations or the network for implementing the distributed processing platform. Distributed applications, like the one presented, can be easily integrated into today's networks of workstations for immediate performance gains without incurring additional costs.

Algorithms and models for distributed computing will play an increasingly important role as distributed processing provides the cost-effective, power computing of

the future. As network hardware technology advances, an increasing number of distributed workstations will be interconnected. Vast distributed platforms will be viable as workstations, separated by large geographic distances, are connected with technologically advanced network hardware. Large distributed processing environments, consisting of hundreds or thousands of workstations, will rely on efficient distributed applications for harnessing the computing power available in the future.

List of References

[1] J. N. Magee and S. C. Cheung, "Parallel Algorithm Design for Workstation Clusters", *Software - Practice and Experience*, Vol. 21(3), 1991, pp. 235-250.

[2] R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing", *Computer Graphics*, Val. 18, no 3., 1984, pp. 137-145.

[3] D. W. Jensen and D. A. Reed, "Ray Tracing on Distributed Memory Parallel Systems", *Performance Evaluation Review*, Vol. 18, Iss. 1, 1990, pp. 251-252.

[4] O. Vorberger, R. Feldmann, and P. Mysliwetz, "A Local Area Network Used as a Parallel Architecture", Technical Report 31/1986, Univ. of Paderborn, Germany.

[5] G. Bourbigot and F. Vandewiele, *TCP/IP Tutorial and Technical Overview*, IBM Technical Document GG24-3376, IBM Corporation, 1989.

[6] D. Comer, *Internetworking with TCP/IP, Second Edition, Volume I*, Prentice-Hall, Inc., 1991.

[7] IBM Allnode Switch for Clusting RISC System/6000 and PS/2, IBM Product Offering Document, IBM Corporation, 1994.

[8] A. Mangen, "RAY: A Ray-Tracing Program in C++", *Dr. Dobb's Journal*, #216 July 94, pp. 42-43.

[9] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., 1990.

[10] C. D. Watkins, S. B. Coy, and M. Finlay, *Photorealism and Ray Tracing in C*, M&T Publishing, Inc., 1992.

[11] A. Woo, P. Poulin, and A. Founier, "A Survey of Shadow Algorithms", *IEEE Computer Graphics and Applications*, Vol. 19(6), pp. 13-22.

[12] B. Arnaldi and T. Priol, "A New Space Subdivision Method for Ray-Tracing CSG Modelled Scenes", *Visual Computer*, Vol. 3(2), pp. 98-108.

[13] M. E. Newell, R. G. Newell, and T. L. Sancha, "A New Approach to the Shaded Picture Problem", *Proc. ACM National Conf.*, pp. 443-50.

[14] B. Mandlebrot, *The Fractal Geometry of Nature*, Freeman, Inc., 1982.

[15] A. Watt, *3D Computer Graphics, Second Edition*, Addison-Wesley, 1993.

[16] B. Kernighan and D. Ritchie, *The C Programming Language, Second Edition*, Prentice-Hall, Inc., 1988.

[17] K. Sung, "A DDA Octree Traversal Algorithm for Ray Tracing", *Proceedings of Eurographics '91*, pp. 73-85.

[18] T. Kay and J. Kajiya, "Ray Tracing Complex Scenes", *Computer Graphics (SIGGRAPH '86 Proceedings)*, Vol. 20(4), 1986, pp. 269-278.

[19] J. D. Foley, A. vanDam, S. K. Feiner, and J. F. Hughes, *Computer Graphics Principles and Practices*, Addison-Wesley, 1990.

[20] A. S. Glassner, "Space subdivision for fast ray tracing", *IEEE Computer Graphics and Applications*, Vol. 4(10), 1984, pp. 15-22.

[21] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated ray tracing system", *IEEE Computer Graphics and Applications*, Vol. 6(4), 1986, pp. 16-26.

[22] M. R. Kaplan, "Space Tracing, a constant time ray tracer", *SIGGRAPH 85 Course Notes*, San Francisco CA, 1985

Vita

Darrin Weber was born in Beacon, New York, on April 9, 1969 to Donald Lee Weber and Donna Lee Peters. He earned his Bachelor of Science degree in Computer Science from Bucknell University, Lewisburg, Pennsylvania, in June of 1991. Darrin is currently employed as an Information Technology Specialist for Air Products and Chemicals, Inc. in Trexlertown, Pennsylvania.

Appendix A. Sample Resource Files

A.1. Spheres Studio Resource File

```
/* ---------------------------- */
/* BALLS.STU                    */
/*                              */
/* Studio File for BALLS.MOD    */
/* Test Model for Timing Tests  */
/*                              */
/* Author: Darrin L. Weber      */
/* Date:   Oct.  1994           */
/* ---------------------------- */

#include "balls.col"

1     {
      studio  {
            eye             0 -1000 0
            target          0.0  0.0  0.0
            up              0.0  0.0  1.0
            background      black
            height          512
            width           512
            angle           64
            aspect          1.333
            maxlev          5
            maxobj          4
            boxtop          boxtop.tga
            boxfront        boxfront.tga
            boxback         boxback.tga
            boxbottom       boxbot.tga
            boxleft         boxleft.tga
            boxright        boxright.tga
      }
      group copper_sphere {
                  rotate {
                        axis   -1 0 1
                        deg    0
                        origin 0 0 0
                  }
      }
      group chrome_sphere {
                  rotate {
                        axis   1 0 1
                        deg    0
                        origin 0 0 0
                  }
      }
}
```

A.2. Spheres Model Resource File

```
/* ---------------------------- */
/* BALLS.MOD                    */
/*                              */
/* Model File for BALLS.MOD     */
/* Test Model for Timing Tests  */
/*                              */
/* Author: Darrin L. Weber      */
/* Date:   Sept. 1994           */
/* ---------------------------- */

/* Front Point Light Source */
light  {
        type      point
        source    500 -1000 500
        color     1.0 1.0 1.0
        radius    1000
        intensity linear
        shadows   natural
        group     front_light
}

/* Back Point Light Source */
light  {
        type      point
        source    -500 1000 500
        color     1.0 1.0 1.0
        radius    1000
        intensity linear
        shadows   natural
        group     back_light
}

/* Surrounding colored spheres that pulsate */
sphere {
       radius    100
       center    -250 0 250
       material  red_plastic
       group     red_sphere
}

sphere {
       radius    100
       center    250 0 250
       material  blue_plastic
       group     blue_sphere
}

sphere {
       radius    100
       center    250 0 -250
       material  green_plastic
       group     green_sphere
}

sphere {
```

```
        radius    100
        center    -250 0 -250
        material  yellow_plastic
        group     yellow_sphere
}

/* Center glass sphere */
sphere {
        radius    150
        center    0 0 0
        material  glass
}

/* Left orbiting silver sphere */
sphere {
        radius    50
        center    -106 0 106
        material  chrome
        group     chrome_sphere
}

/* Right orbiting copper sphere */
sphere {
        radius    50
        center    106 0 106
        material  copper
        group     copper_sphere
}
```

A.3. Spheres Material Resource File

```
/* ---------------------------- */
/* BALLS.MAT                    */
/*                              */
/* Material File for BALLS.MOD  */
/* Test Model for Timing Tests  */
/*                              */
/* Author: Darrin L. Weber      */
/* Date:   Sept. 1994           */
/* ---------------------------- */

#include "balls.col"

green_plastic {
                ambient       0.15
                diffuse       0.85
                diff_color    light_green
                specular      0.80
                spec_color    white
                reflect       0.15
                transparent   0.0
                ior           1.0
                finish        100
}

red_plastic {
                ambient       0.15
                diffuse       0.85
                diff_color    light_red
                specular      0.60
                spec_color    white
                reflect       0.15
                transparent   0.0
                ior           1.0
                finish        50
}

blue_plastic {
                ambient       0.15
                diffuse       0.85
                diff_color    light_blue
                specular      0.40
                spec_color    white
                reflect       0.15
                transparent   0.0
                ior           1.0
                finish        10
}

glass {
                ambient       0.00
                diffuse       0.05
                diff_color    gray
                specular      0.05
                spec_color    white
                reflect       0.10
```

```
                transparent   0.90
                ior           1.4
                finish        10
}

yellow_plastic {
                ambient       0.15
                diffuse       0.65
                diff_color    light_yellow
                specular      0.70
                spec_color    white
                reflect       0.15
                transparent   0.0
                ior           1.0
                finish        4
}


chrome {
                ambient       0.025
                diffuse       0.10
                diff_color    chrome_col
                specular      0.63
                spec_color    chrome_spec
                reflect       0.76
                transparent   0.0
                ior           1.0
                finish        100
}

copper {
                ambient       0.025
                diffuse       0.53
                diff_color    copper_col
                specular      0.40
                spec_color    copper_spec
                reflect       0.15
                transparent   0.0
                ior           1.0
                finish        50
}
```

Appendix B. Sample Color Images

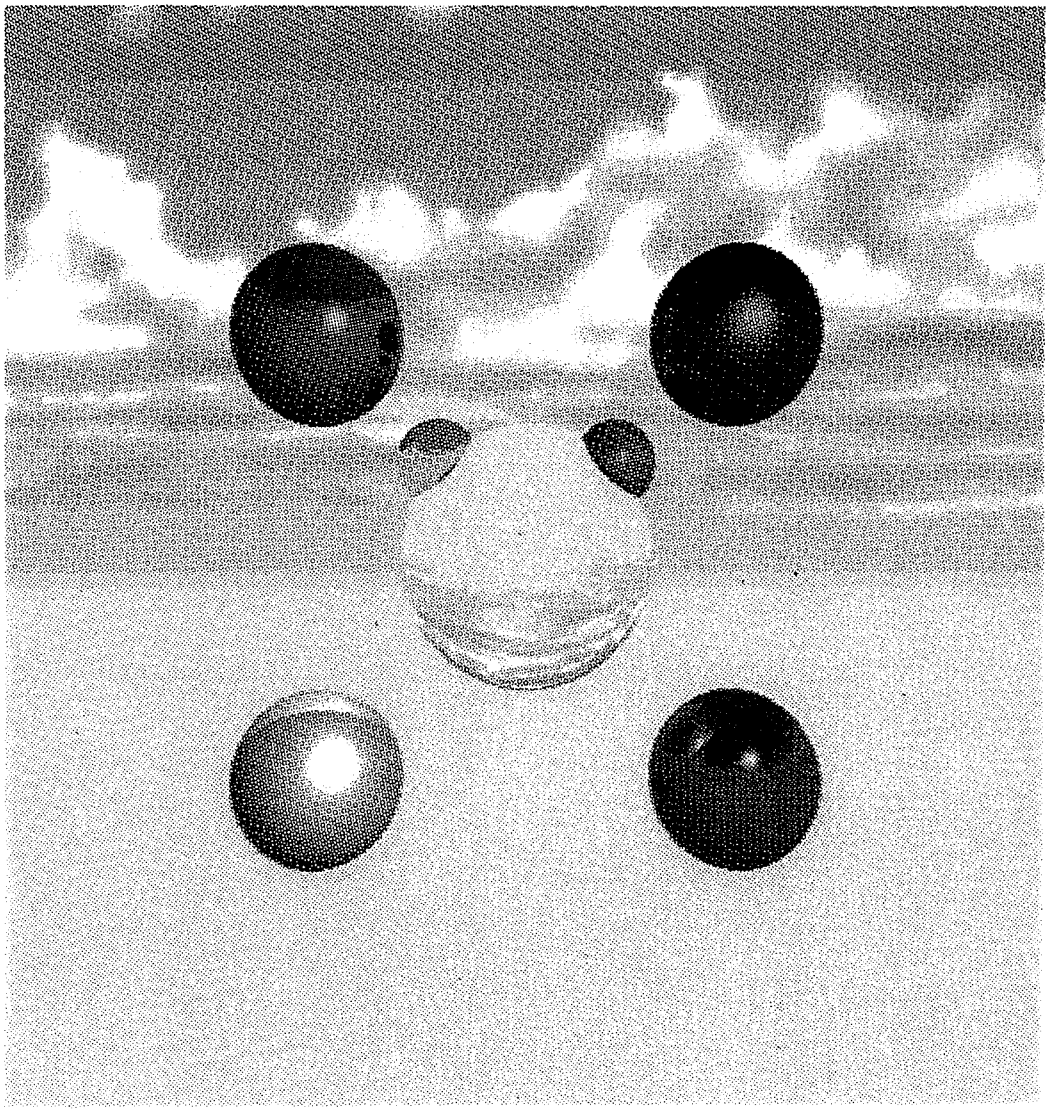


Plate 1. Spheres Scene

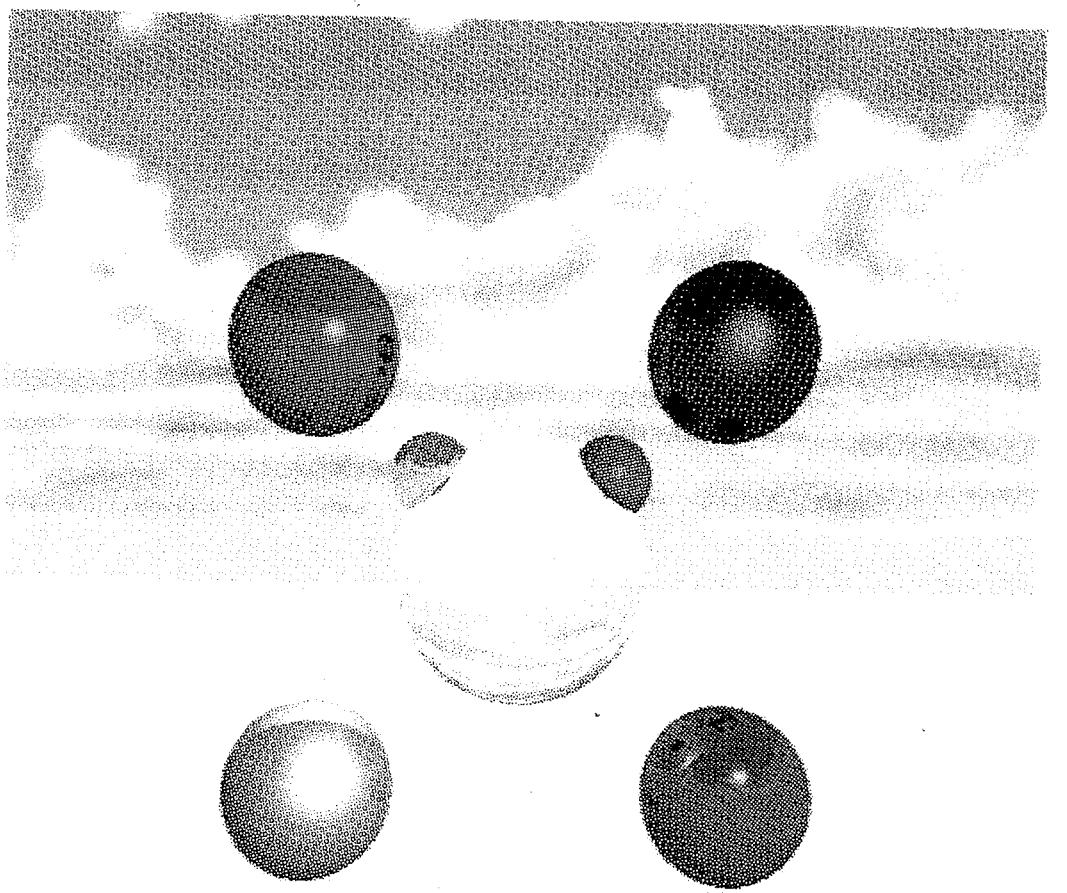


Plate 1. Spheres Scene

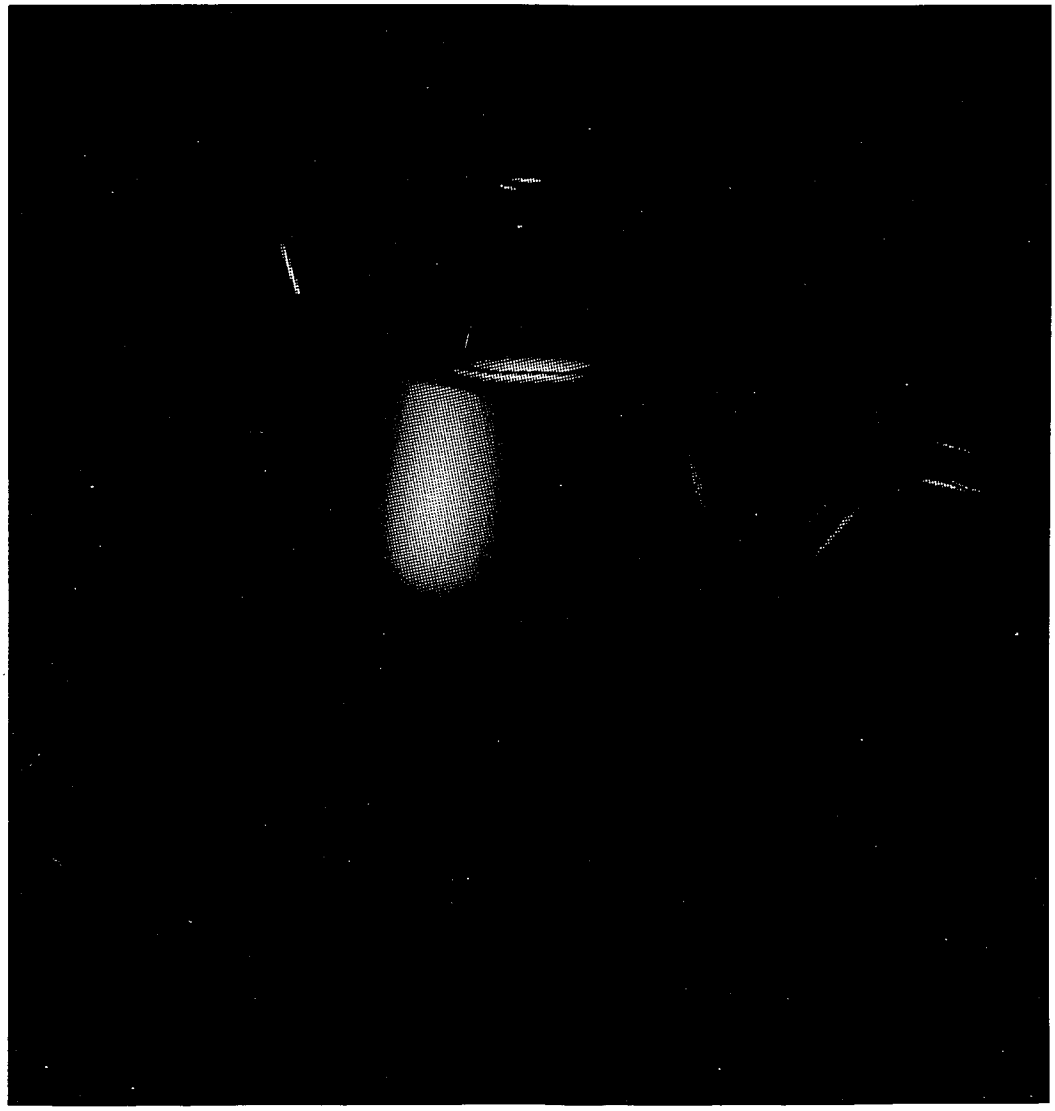


Plate 2. Utah Teapot Scene

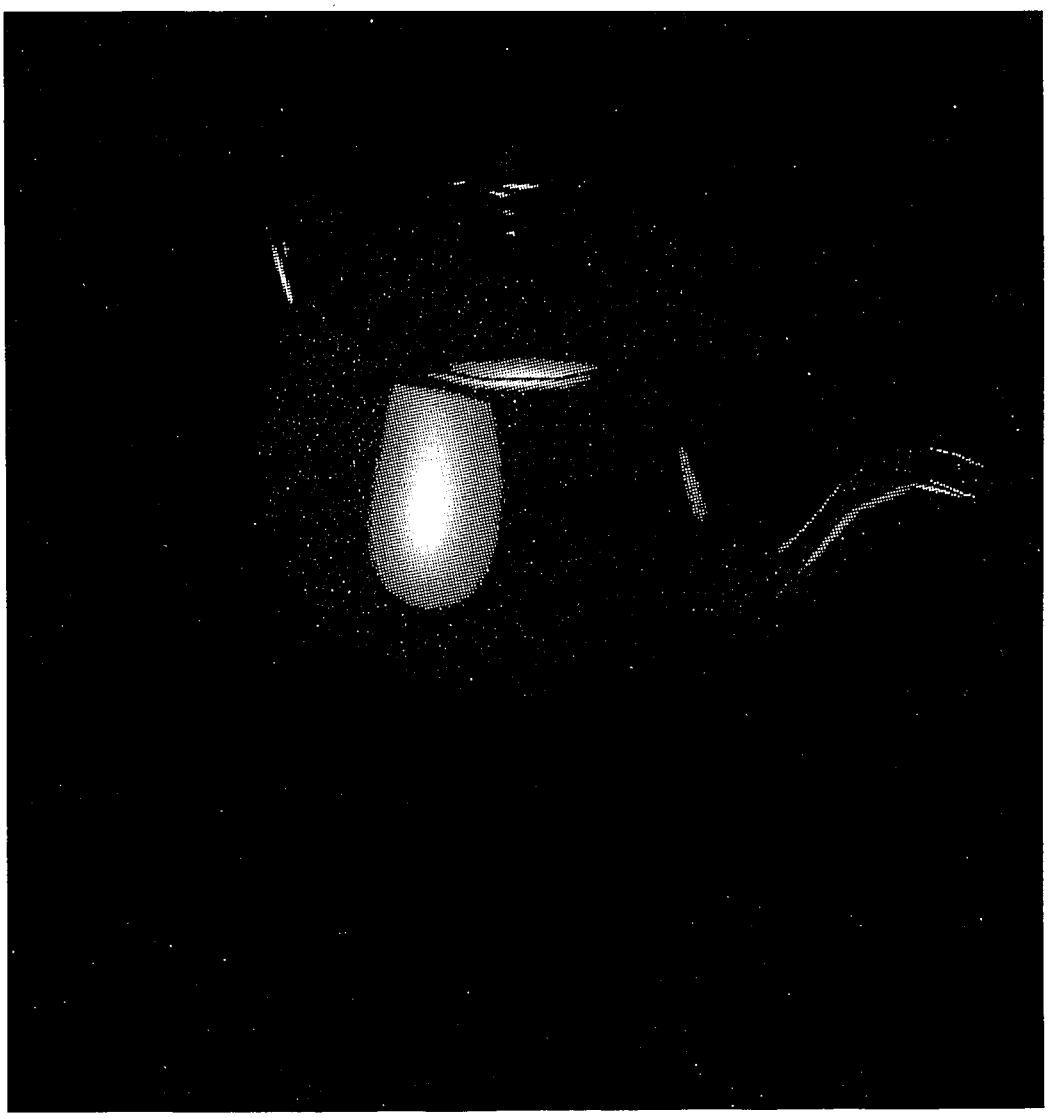


Plate 2. Utah Teapot Scene

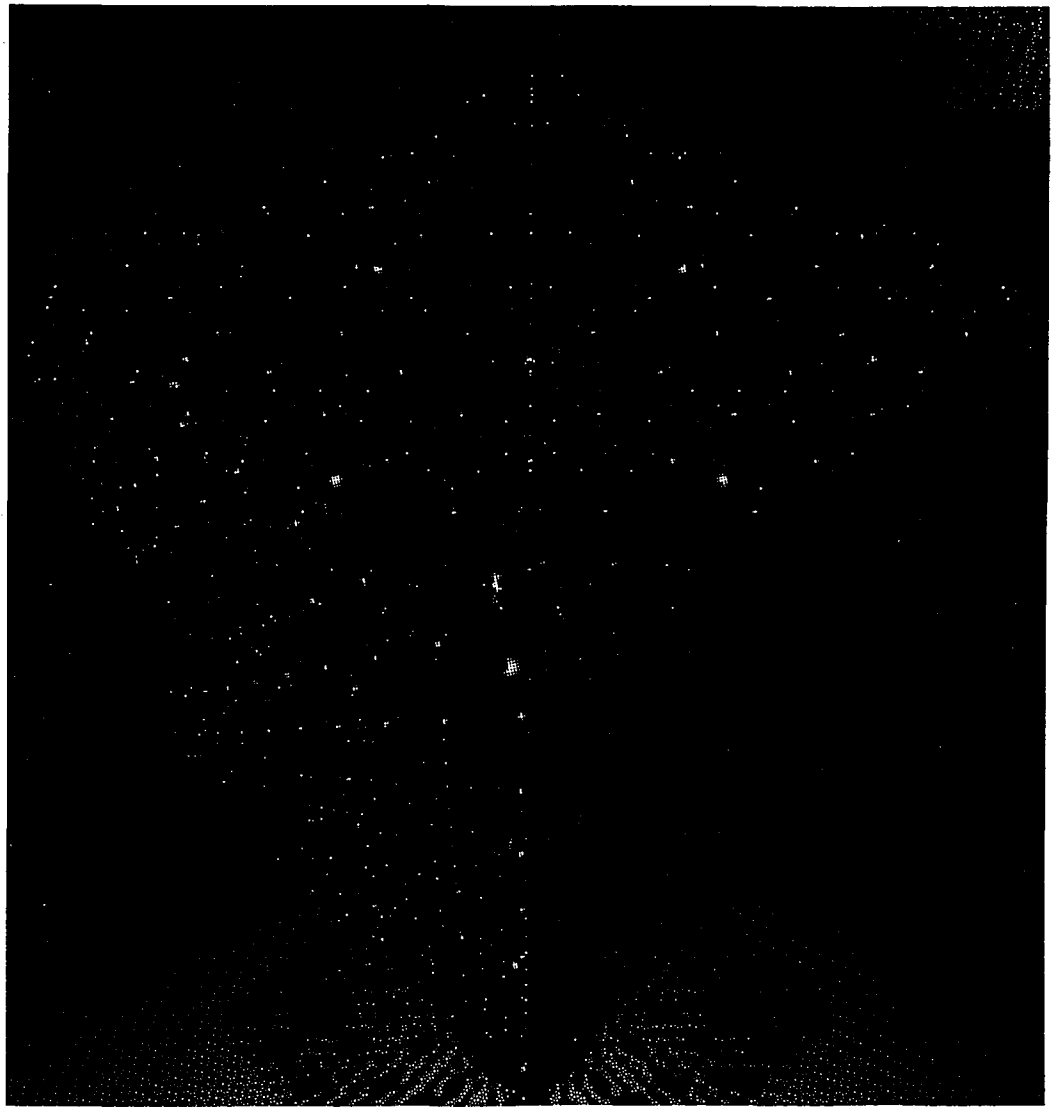


Plate 3. Menger's Fractal Sponge Scene

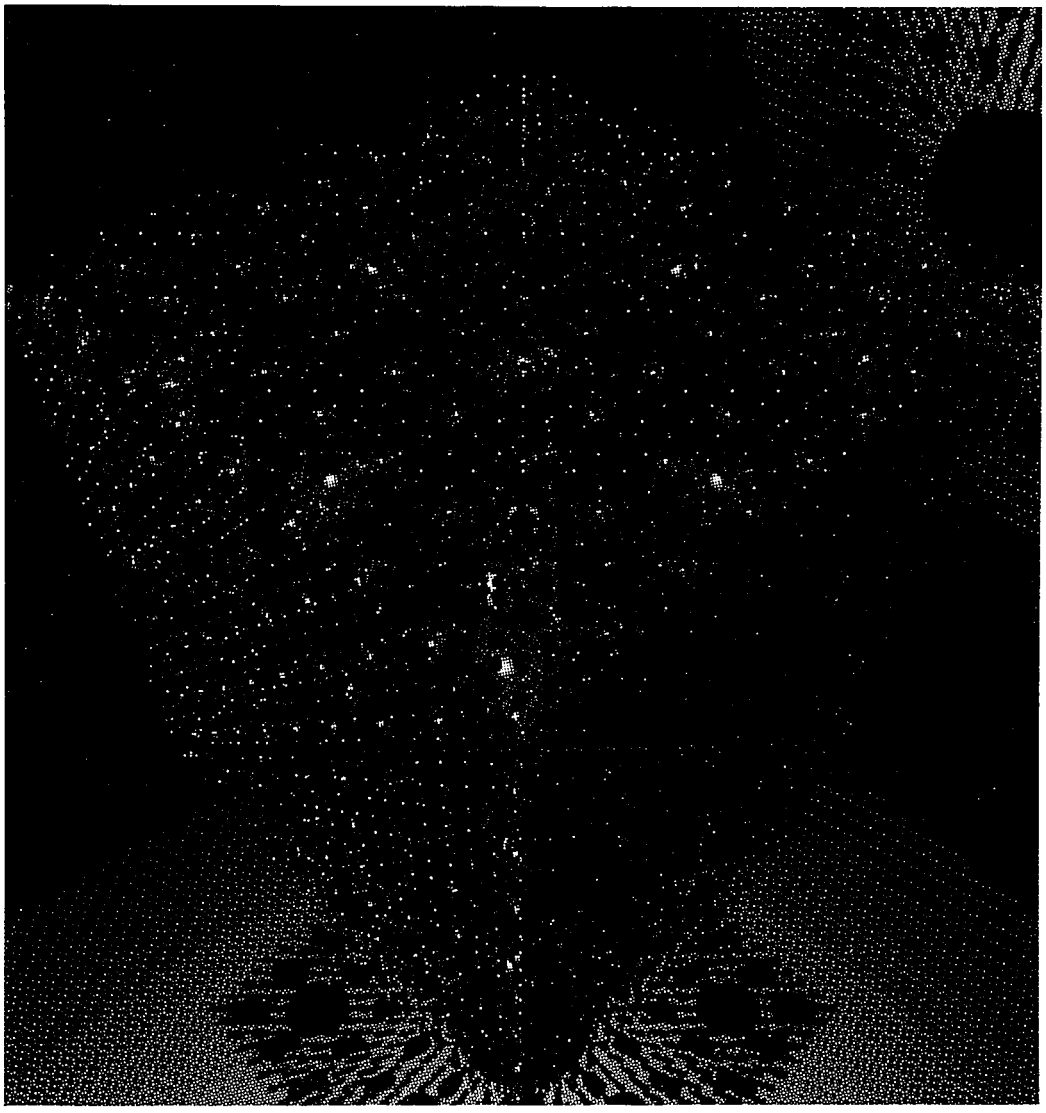


Plate 3. Menger's Fractal Sponge Scene

END OF TITLE