UC-405
LBL-27517

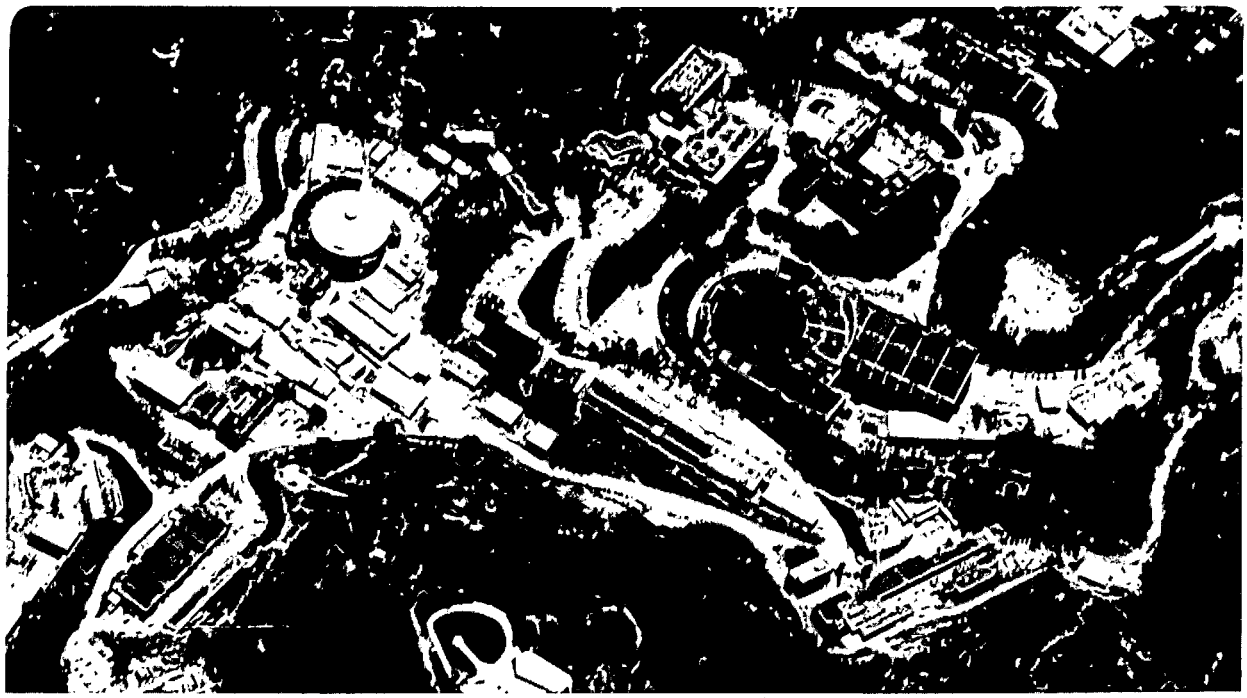
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Information and Computing Sciences Division

Distributed Processing in Ray Tracing

M.T. Tran
(M.S. Thesis)

August 1989



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-27517

DISTRIBUTED PROCESSING IN RAY TRACING

Master of Science
Thesis

Mydung Thi Tran
Advanced Development Projects
Information & Computing Sciences Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, California 94720

August 1989

This work was supported by the Director, Office of Energy Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

DISTRIBUTED PROCESSING IN RAY TRACING

A thesis submitted to the faculty of
San Francisco State University
in partial fulfillment of the
requirements for the
degree

Master of Science
in
Computer Science

by

Mydung Thi Tran

San Francisco, California

August, 1989

ACKNOWLEDGEMENTS

I wish to extend my sincere gratitude to all the members of the Thesis Advisory Committee: Dr. McDonald, Mr. Johnston, Dr. Kroll, and Mr. Greg Ward of Lawrence Berkeley Laboratory, whose time, patience, understanding, and invaluable criticisms and suggestion had contributed a great deal in the completion of this thesis, and the past and current members of the Information and Computing Science Division, Advanced Development Group at Lawrence Berkeley Laboratory for their help.

This work was supported by the Director, Office of Energy Research, U.S. Department of Energy, under Contract No. DE-AC03-76SF00098 with the Lawrence Berkeley Laboratory, University of California.

TABLE OF CONTENTS

List of Tables viii

List of Figures ix

List of Appendices x

Chapter

1. Introduction 1

    Goal 1

    Environment 2

2. Ray Tracing 4

    What is Ray Tracing? 4

    Advantages versus Disadvantages 8

    Different Methods Developed to Improve Ray Tracing Speed 8

3. Distributed Processing 19

    Different Classes of Coupled Processors 19

    Remote Procedure Call 20

    Sun Microsystems' Network File System 26

    External Data Representation 26

4. Design and Implementation 30

    Major Issue 30

    Major Components/Processes 30

    Flow of Control 35

    RPC Programming Issues 41

5. Result and Effectiveness Analysis 44

Performance of an Asynchronous Distributed Environment... 44

Factors that might effect Performance..... 45

Models used in Estimating Results 48

Estimated Time versus Obtained CPU Results 49

Estimated Time versus Obtained Real-time Results..... 49

Justification for Differences in Comparison 49

6. Conclusion 54

Meet design goals 54

Further improvements..... 54

References..... 57

Appendices..... 61

LIST OF TABLES

Table

3.1: XDR types 5

5.1: Time results (seconds) 46

5.2: Time results (speed-up ratios) 46

5.3: CPU profile (seconds)..... 50

5.4: CPU profile (percentage)..... 50

LIST OF FIGURES

Figure

2.1: Ray tracing 5

2.2: Parallel Processing of Object Space - System Architecture 11

2.3: Program and Data Relationship in Radiance 15

2.4: Space Division for Octree Structure 17

3.1: Sun's RPC client-server paradigm 22

4.1: Distributed Architecture 36

4.1.1: Distribution client 37

4.1.2: Ray tracing server 38

4.1.3: Collection server 39

4.2: Flow of control 40

5.1: Ray-traced Gold Sphere 53

LIST OF APPENDICES

Appendices

A. Program Listings

Notice of Copyright 62

color.h 63

distribute.h 64

distribute_client.c 66

raytracing_server.c 77

collect_server.c 91

B. Program Profiles

distribute.p 99

raytrace.p 101

collect.p 104

Chapter 1

Introduction

1.1. Goal

As the demand for computing power keeps increasing scientific and industrial applications push computer and programming technology to its limit. Semiconductor technology has also been advancing dramatically. While the cost of hardware has been decreasing large and complex multiprogrammed uniprocessors are being manufactured. Multi-processor systems have become easier to purchase and in fact are much cheaper to own. Local Area Computer Network technology has also developed very quickly, a computing environment consisting of workstations connected by a high-speed network is becoming more and more popular. This concept of distributed processing has grown widespread. A computing job which has certain parallel characteristics is an excellent application for distributed processing. The process of image generation using "ray tracing" fits this category of computing applications. The purpose of this thesis is to investigate the issues of distributed processsing in ray tracing. The work required to ray trace a still image is distributed among several loosely coupled processors, connected by a local area network, using the DARPA Internet Protocols. Improvement in process time, the interval from the moment that the first ray is traced to the moment the last ray result is collected, over serial processing is demonstrated for the proposed method.

1.2. Environment

1.2.1. Operating System and Programming Language

This thesis work was carried out at Lawrence Berkeley Laboratory, under the supervision of Mr. William E. Johnston. Available as tools for my work are:

- Radiance, a tool developed by Greg Ward, at Lawrence Berkeley Laboratory, is a ray tracing method for calculating luminance, and producing realistic images of a complex lighting environment. It was developed as a research tool for predicting the distribution of visible radiation in illumminated spaces. Light rays are traced backward, from the image plane to the source(s) [WardGr86]. Radiance is written in the C programming language, in the 4.3BSD Unix environment.

- In addition, Sun Microsystems' Remote Procedure Call (RPC) mechanism is also available. It provides communication between heterogeneous environments (i.e. machine architecture, programming languages and operating systems) distributed across a computer network.

1.2.2. Hardware available

VAX 11-780 's and Sun Workstations (ranges from SUN 3-50 to 3-110) provide the computing resources. They are connected by a local area network and run versions of the Unix (4.3BSD) operating system. In addition, the Sun workstations are connected to file servers using Sun's NFS (Network File System) Protocol. NFS performs operations such as storing and retrieving information from a file upon reveiving requests from the workstations.

I choose to run the control process, the distribution client, which acts as a supervisor in distributing the work on the VAX 11-780. Sun workstations are my ray tracing servers, the workers who carry out the actual task. The VAX 11-780 is the host where all results generated by the ray tracing servers are sent.

Chapter 2

Ray Tracing

2.1. What is Ray Tracing?

Ray tracing is one of the most exciting area in computer graphics. To understand the basic idea of the technique of Ray tracing, one must think about how our eyes normally work. Light rays from a light source illuminate an object and are reflected or transmitted through surfaces in many directions. Some of these light rays eventually reach our eyes, impact on the retina and are relayed to our brain. From each light source there are an infinite number of rays, and these rays most often do not reach the viewer. As the result, the process of tracing rays from the sources is computationally inefficient. In 1968, Appel originated the idea of tracing rays from the opposite direction, i.e. from the viewpoint as we look out to the scene[Appel68]. The raster screen of the computer monitor is treated as an imaginary plane positioned in front of the scene [Figure 2.1].

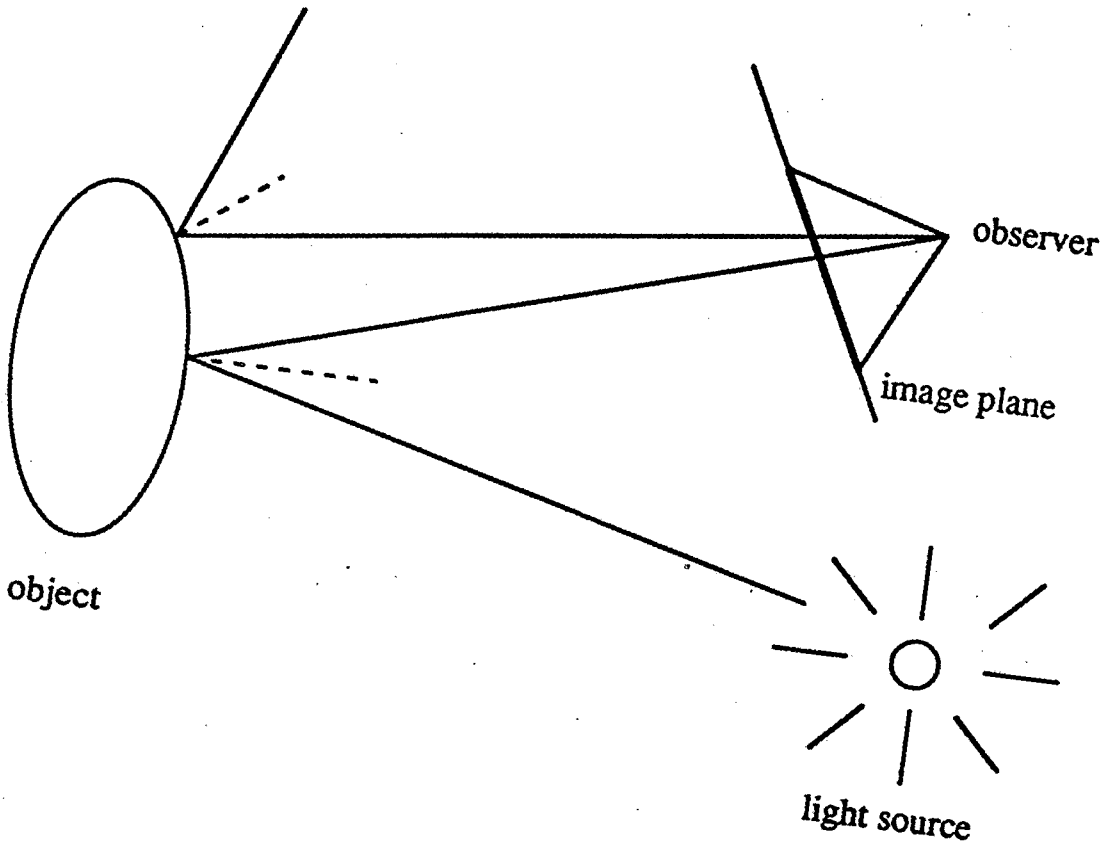


Figure 2.1 : Ray tracing

Appel's ideas brought a new horizon to image synthesis. Since then many ray-tracing algorithms have been developed. Whitted implemented an algorithm in conjunction with global illumination models, and true reflection, refraction, transparency and shadowing are accounted for in this model. Anti-aliasing is also included [Whitted80].

In ray-tracing algorithms luminance is computed by following light backwards from the point of measurement to the source(s). Each "light ray" can be thought of as the luminance value that results either directly from an emitting surface or indirectly from a reflecting surface. The basic steps of ray-tracing method are:

1. Determine surfaces that rays intersect.

2. Calculate luminance in the ray's direction, tracing newly spawned rays, if any.

3. Return the computed luminance value.

In step 1, for a scene of multi-surfaces, an individual intersection test must be performed for each surface in the ray's path, until the closest point of intersection, with respect to the viewer, is encountered. Radiance, the ray- tracing tool used, applies the octree sorting method developed by Glassner, as the quick way of identifying those surfaces that are in the path of a ray. The scene space is recursively divided into cubes, each tree cube, branches into eight sub-cubes. A leaf cube, one that does not subdivide, contains a list of the surfaces that penetrate this leaf cube. To construct the octree for a scene start with an empty cube which completely contains the scene, add to it surfaces, one by one, using the following algorithm:

```
AddSurface(Cube, Surface):
    if Intersect(Cube, Surface) then
        if not Leaf(Cube) then
            for each SubCube in Cube :
                AddSurface(SubCube, Surface)
        else
            if (SetSize(Cube.Set) < N) OR (CannotDivide(Cube) then
                AddElement(Cube.Set, Surface)
            else
                SubDivide(Cube)
            for each SubCube in Cube:
                        for each Element in Cube.Set:
                            AddSurface(SubCube, Element)
                    AddSurface(SubCube, Surface)
```

where *Intersect*() is the test for whether a cube contains a surface. N is the user-defined maximum set size for a leaf. *CannotDivide*() prevents cubes from being made smaller than a certain size.

For step 2, as mentioned earlier, there are two components: direct and indirect illumination.

1. The contribution of direct illumination is computed using locations and sizes of the light sources in the scene to determine whether the surface is in shadow. If the surface is completely in shadow, the contribution is zero. If the surface is illuminated by a source, the size, distance, brightness, direction of the source, and the surface reflectance distribution are used in calculating the contribution. If the surface is partially illuminated, the luminance is fractional. A Monte Carlo technique is used to determine the contribution due to penumbra phenomenon (the "soft shadowing" due to a finite size light source).

2. Illumination arriving at a surface that did not travel directly from any light source (i.e. indirect illumination) is computed by sampling re-radiated

luminance values over a hemisphere defined by the surface element position and normal direction. The bidirection reflectance distribution function could be divided into two components: diffuse and specular [Ward86].

2.2. Advantages versus Disadvantages

Ray tracing is one of the techniques for hidden surface removal. David Rogers has compared it with others such as: list priority algorithms, Z-buffer algorithm, spanning scanline algorithms, etc...[Rogers85].

Ray tracing is used as a method to do shadowing for surfaces that transmit as well as reflect light [Foley82].

"Ray tracing is one of the most elegant techniques in computer graphics. Many phenomena that are difficult or impossible with other techniques are simple with ray tracing, including shadows, reflections, and refracted light." [Cook84]. Cook also developed the technique of Distributed Ray tracing, which provides easy solutions to some previously unsolved or partially solved problems, such as motion blur, depth of field, penumbra, translucency and fuzzy reflections.

Ray tracing has the capability of producing very realistic images. It uses a global lighting model to calculate reflection, refraction, shadow. In addition, it can handle different geometric primitives. Although the algorithm is very simple, it has a major drawback: it is time-consuming.

2.3. Different Methods Developed to Improve Ray Tracing Speed

Since its introduction in the field of Computer Graphics, many improvement methods have been developed for ray tracing.

2.3.1. Ray Casting

Roth describes a method for solid modeling using ray casting. Blocks and cylinders are combined to model solid objects. Virtual light rays are cast as probes to visualize and analyze the composite solids modeled [Roth82].

2.3.2. Space Subdivision

A method which reduces the number of time-consuming object-ray intersection calculations that have to be made was developed by Glassner [Glassner84]. The space of a three-dimensional scene is divided into small cubes. A list of all the objects residing in each of these cubes is maintained. For each ray traced, determine the cube from which the ray originated. Follow the ray and compare only against the objects it hits in that cube. If the ray passes through one or more objects, the return value for the ray is the color of the first object the ray hits. If the ray does not hit any object in this cube, project the ray into the next cube and repeat the process.

The algorithm to get to the next cube is based on the idea of finding a point that is guaranteed to be in the next cube. Statistics for this algorithm gathered with code written in C, running under Unix, on VAX 11-780 indicates speed-up ratios ranging from 4 up to 27 times, depending upon the image complexity [Glassner84].

The octree technique which describes the breaking up of space into cubes was developed by Jackins and Tanimoto and Meagher [Meagher82]. Space is dynamically divided into cubes of decreasing volume until each cube contains less than a minimum number of objects.

2.3.3. The Light Buffer: A Shadow-Testing Accelerator

Light buffers, cubes surrounding light sources, are generated to partition the environment. This partitioning is used in the process of shadow testing to quickly determine a small subset of objects for intersection testing. Depending upon the resolution of the light buffer and the scene complexity, this approach, proposed by Haines and Greenberg, has a speed-up ratio of 4 to 30 [Haines86].

2.3.4. An Adaptive Subdivision Algorithm & Parallel Architecture

Dippe and Swensen subdivide the three dimensional space of a scene into subregions, more or less uniformly, and load them with object descriptions. Rays are traced in these subregions and tested for intersection only with those objects within a subregion. Those rays which do not hit objects are traced into neighboring subregions. To maintain balance in the work load, the space is redistributed among subregions as computational loads are determined. The algorithm is implemented using independent computers, each responsible for one or more subregions, and each communication is with a few neighbors using messages [Dippe84].

2.3.5. Parallel Processing of an Object Space

A hardware approach is presented by Kobayashi, Nakamura and Shigei. There are five components to the system architecture [Figure 2.2].

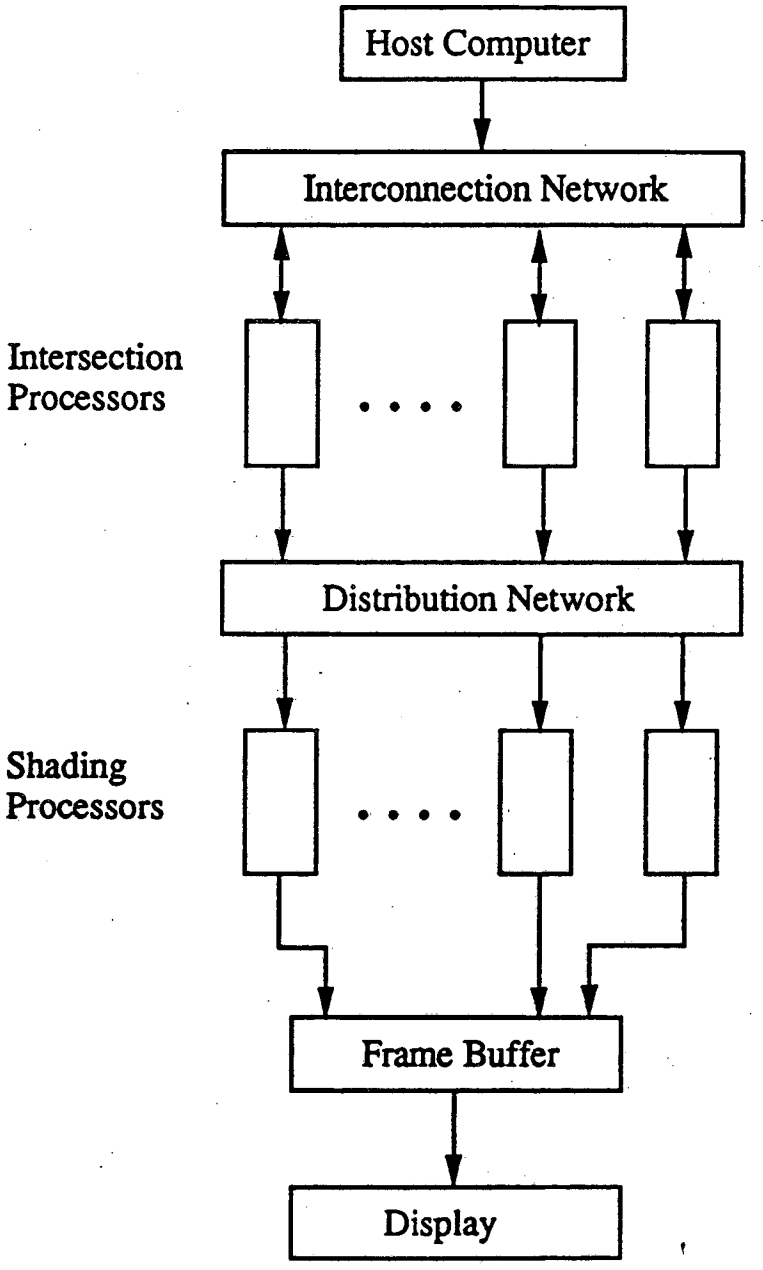


Figure 2.2 : Parallel processing of object space - System architecture

In this method, the host computer subdivides the object space and initial rays and sends them to IP (Intersection Processors). If the rays intersect any objects within the subspace, the local intensity is calculated and passed to the SP (Shading Processors). If the rays do not intersect an object, they are passed to a neighbor IP. SPs calculate the global intensity of pixels. As the number of objects of a scene increases, the total processing time of intersections decreases [Kobayashi87].

2.3.6. Distributed Processing - The Chosen Method

Parallel processing is the key issue of this thesis. The proposed method is based on the observation that rays can be traced independently. Radiance provides a routine named "*rtrace*", which takes as input a scene description, an octree structure generated from the scene description, and a file which contains the rays' origin and direction. From these, "*rtrace*" traces the rays providing a color for each pixel of the raster image.

The scene description file is a three-dimensional environment in Cartesian world coordinates. The surfaces and materials that make up the specific environment of the image to be generated are listed in this scene description file. This is an Ascii file with the following format:

xmin ymin zmin size

#comment

modifier type identifier

n S1 S2 S3 ... Sn

0

m R1 R2 R3 ... Rm

!command

The bounding cube containing all surfaces is defined by the four numbers on the first line. The second line, which start with a pound sign (#), is an example of a comment. Groups of four lines follow. Each of these groups describes a scene description primitive. A scene description primitive can be either a surface or a modifier, and is described by the following:

– modifier: an identifier, a previously defined primitive, or void

– type: the following are supported:

* surfaces: source, sphere, bubble, polygon, cone, cup, cylinder, tube and ring
* textures: Texfunc, a perturbation of the surface normal, which is specified by a function
* materials: light, illum (secondary light source), glow, plastic, metal, dielectric, glass, clip (a material which acts to cloak other surfaces from view)
* patterns: colorfunc, brightfunc (monochromatic), colordata(data map interpolation), brightdata(monochromatic data map interpolation), text

– identifier: any string of non-blank characters.

The arguments of the primitive can be strings or real numbers. These arguments make up the second, third, and fourth lines of the primitive description. The command line, which starts with the exclamation mark, is executed by the shell.

The following is the scene description file of the image of a gold sphere used:

```
# 0    0      0      10
#
# A room, a ball, and a lightbulb
#

void plastic light_blue
0 0
5      .6     .6     .7     0      .1

!genbox light_blue room 10 5 8 -i

void metal gold
0 0
5      .45    .25    .02    .9     0

gold sphere ball
0 0
4      2      1      2      1

void light incandescent
0 0
3      1000   1000   1000

incandescent sphere lightbulb
0 0
4      5      4.5    4      .4
```

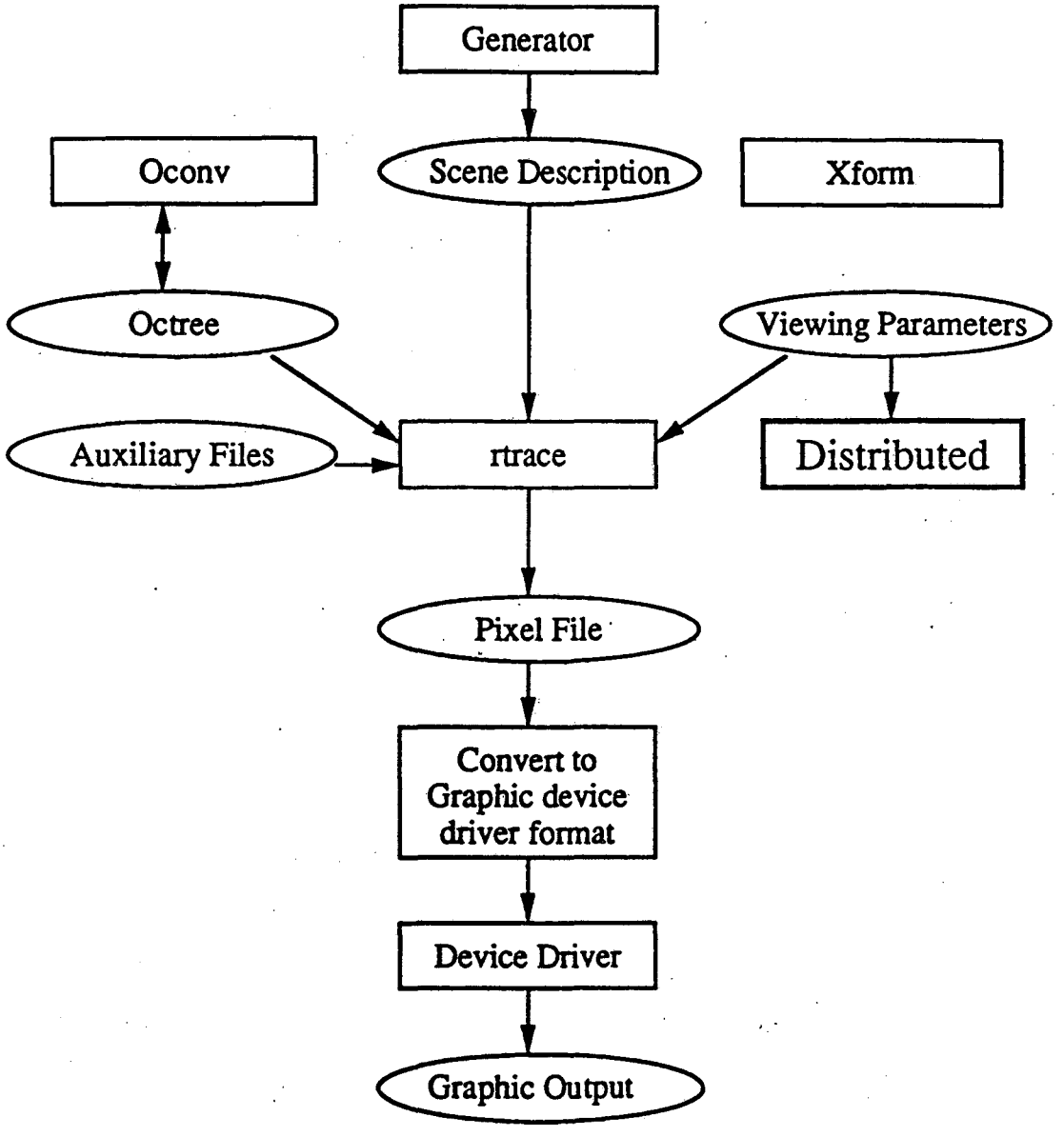


Figure 2.3 : Programs and Data Relationship in Radiance

Figure 2.3 describes the relationship between data files and programs. The oval boxes represent the data files while programs are illustrated using rectangles.

"*Generators*" are programs that produce a scene description as output. For examples: *genbox* produce a parallelopiped given width, height and depth, *gensky* generates a Radiance description of the sky, etc...

"*Oconv*" takes a scene description and creates an octree structure.

"*Xform*" transforms a Radiance scene description (i.e. translates, rotates, mirrors the scene about the xy, xz, yz plane, etc..).

The octree structure is used for geometric modeling. Arbitrary 3-D objects are represented to any specified resolution in a hierarchical 8-ary tree structure or "*octree*". Solid objects may be concave, convex, or consist of disjoint parts. Boolean operations such as union, intersection and difference are used to combine primitive solids, such as blocks and cylinders, into solid modeling objects. Octree encoding technique makes real time analysis and manipulation of highly complex objects possible, due to the fact that it does not require floating-point operations, integer multiplications, or integer divisions [Meagher82]. While an octree is the input to the ray tracing program, it also directs the use of a scene description [Ward86]. In an octree structure, cubic space is divided into 2x2x2 = 8 basic cells by halving each sides. Each of the cell is called an octant (also called voxel). For the first division, a cube is partitioned into 8 octants. Each of these first generation octants could be further divided into smaller octants, or suboctants [Figure 2.4].

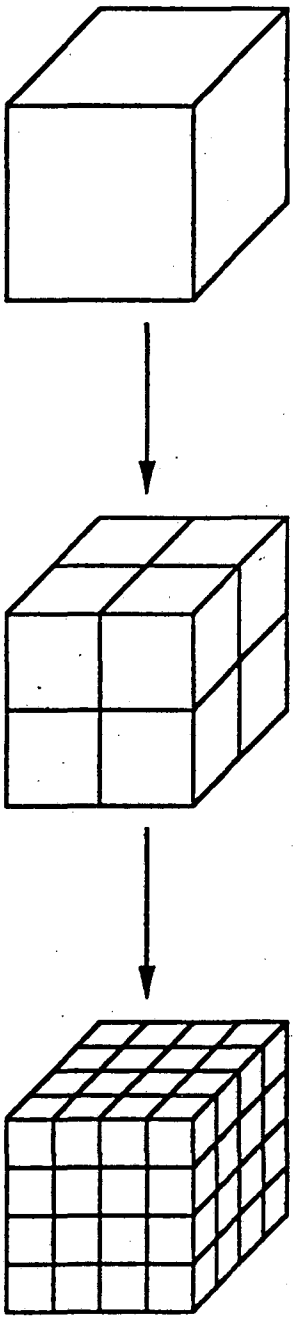


Figure 2.4 : Space Division for an Octree structure

The dividing process continues until a desired resolution is reached. This resolution increases with the number of the objects in a scene. Space is typically divided 3-7 times for images generated using Radiance. In addition, there is a trade-off between the resolution and the time taken to determine the intersection points (where light rays hit objects). The higher the resolution, the less time taken to determine intersection points, i.e. smaller space volume results in less effort for finding intersection points.

Ray tracing is extremely time-consuming. Most of the computing time is spent in calculating the intersections between objects and rays. The number of these intersections is enormous. The number of rays for a given image is proportional to the number of pixels and the number of light sources. Approximately one million rays are traced for the test image, the image of the gold sphere, using one light source. Efficiency in data structure management allows fast access to geometrical information. The Octree structure is utilized to improve the data access time.

Chapter 3

Distributed Processing

In the past, a collection of small computers which had the equivalent capability of a large computer has cost more than the large computer. That has changed in the past few years, and distributed systems have become an increasingly important trend in the computer world. A distributed system is a collection of computers (linked by shared memory, communication lines, etc...) or processing elements working closely together to solve a single problem or problems. Distributed processing is a technique that implements a set of processing tasks across several computers. Each computer performs some part of the total processing required.

In this chapter a brief explanation of different classes of processor coupling will be presented followed by the remote procedure call facility with its paradigm, layers, binding protocol, and data representation in a heterogeneous environment.

3.1. Different Classes of Coupled Processors

Systems can be categorized by the degree of coupling between processors. In general, there are three classes:

3.1.1. Tightly Coupled

In this class, the processors are physically close. Shared memory is used for data transfer and for program storage. Tight coupling is required when there is a

high degree of interprocess, or even more commonly intraprocess communication. These systems are classified as multiprocessors.

3.1.2. Moderately Coupled

These systems are characterized by high levels of intercomputer activity using either high speed serial lines, 50 Kbytes/sec or greater, parallel data busses or shared disks.

3.1.3. Loosely Coupled

Processors of these systems can be either co-located or separated by large distances. Serial lines and relatively low transmission speeds are among the characteristics of these systems. The amount of interprocess activity is relatively low, and the degree of error checking is high.

Available for my work, as described in the previous section, is an environment consisting of loosely coupled processors.

3.2. Remote Procedure Call

Communication among processes plays an important role in this project. Shared primary memory is not common in distributed systems. Communication using semaphores and monitors is not applicable. Message-passing is the method, and the preferred model in this area is the client-server model using Remote Procedure Call (RPC) for interprocess communication.

3.2.1. The RPC Paradigm : the Client-Server Model

For the local procedure model, the caller passes arguments to a procedure, typically by pushing them onto to an in-memory stack. The control is then

transfered to the procedure, and eventually the caller will gain back control so that it can continue. RPC is similar to the local procedure model, except there are two processes - the caller, or client, and the server processes - in the thread of control.

In Sun's RPC the call message has three unsigned fields: a remote program number, a remote program version number, and a remote procedure number. These three fields uniquely identify the procedure to be called. When a program first becomes available on a machine, it registers itself with the port mapper on the same machine. The port mapper program maps RPC program and version numbers to UDP/IP or TCP/IP port numbers. This port mapper program makes dynamic binding of remote programs possible. When a program becomes unavailable, it should unregister itself with the port mapper program on the same machine [SunPRC86]. This is desirable since the number of potential remote programs is very large and the range of reserved port numbers is very small. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

The caller process initiates the remote procedure call by sending a call message, which contains the procedure's parameters, to the server process, and waits for a reply message. When the caller process receives the reply, the results of the procedure are extracted and the control returns to the caller [Figure 3.1].

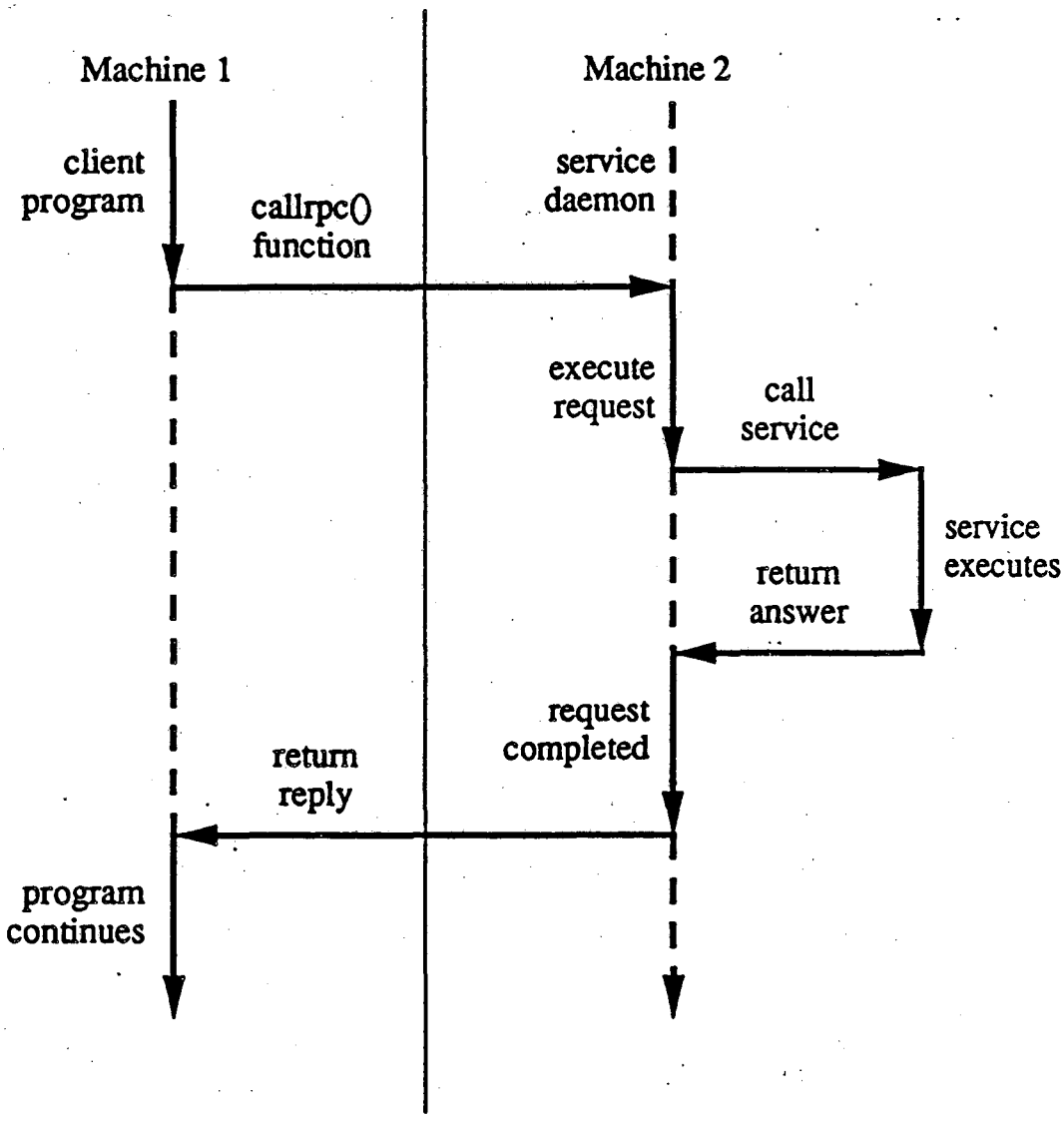


Figure 3.1 : Sun's RPC client-server paradigm

The caller process resides on the client machine while the library of called routines resides on the server machine. The server process, when started, registers all RPC calls it will handle, and then goes into a wait state waiting for service requests, i.e. remains dormant until the arrival of the call messages (request). When the call message arrives, the procedure's parameters are extracted, and the server calls a dispatch routine to perform the requested service. When the called procedure is completed a reply message is sent back to the caller. In Sun's RPC model, the conventional procedure call semantic is exactly emulated only one of the two processes is active at any given time, and multi-threading of the caller or server is not supported. Riche semantics are supported by other RPC mechanisms.

3.2.2. Different Layers of RPC

Sun RPC interface is divided into three layers, according to the degree of transparency [SunRPC86]:

3.2.2.1. Highest Layer

Total transparency is provided to the programmer in this layer. The call, issued by the programmer, looks like a regular library call with the name of the remote server machine name as argument. This layer consists of system services such as: requesting numbers of users on remote machine (*rnusers*), requesting information about users (*rusers*), determining if remote machine has disk (*havedisk*), getting performance data from a remote kernel (*rstat*), writing to a specified remote machines (*rwall*), geting name of name server master (*getmaster*), geting RPC port number (*getrpcport*), updating user password in name server

database (*yppasswd*), etc.

3.2.2.2. Intermediate Layer

There are two routines in this layer: *callrpc*, which resides in the caller process (the client machine), and *registerrpc* which resides in the server process.

- *callrpc* requires eight parameters: the name of the remote machine, the program number, the version number, the procedure number, the type and address of the input argument, the type and the address of the result.

- *registerrpc* establishes what procedure corresponds to each RPC procedure number. It is called with six arguments: the program number, the version number, the procedure number, the name of the dispatch routine, the type of the input to, and output from the procedure.

If successful, callrpc returns with a zero, non-zero otherwise.

3.2.2.3. Lowest Layer

The programmer must know about sockets, the basic building block for communication, in this layer. Transport handles must be established on both sides, in the client and the server, and if bound to a port, this port number must match on both sides . Transport protocol must be specified according to the requirements of the application.

All three layers of RPC are used for this project. As described in the later chapter, workstation status check will utilize a routine of the first layer of RPC. Register and unregister ray-tracing service, as well as result sending service, will

make use of the second and third layers.

3.2.3. Different Message Passing Protocols

Processes communicate with each others through communication channels. One type of InterProcess Communication offerred by 4.3BSD UNIX is sockets. Sockets are end-points of communication. Normally, sockets communicate with sockets of common properties. Depending upon properties which are visible to the user, sockets are typed as follows [SunIPC86]:

* **Stream Socket** : bidirectional, reliable, sequenced and unduplicated flow of data without record boundaries are characteristic of this type of socket. When the supporting network is the Internet, the TCP/IP protocol supports stream sockets.

* **Datagram Socket** : bidirectional flow of data is supported. However, duplication of messages is possible, the order in which the messages are received could be different from the order in which they are sent, and messages may be lost. In the Internet environment the UDP/IP protocol supports Datagrams.

* **Raw Socket** : access to the underlying communication protocol that supports socket abstraction are provided. Sockets of this type are used in developing new communication protocol.

* **Sequenced Packet Socket** : in addition to the properties posessed by *Stream* sockets, record boundaries are also preserved for this type of sockets.

* **Reliable Delivered Message Socket** : this is Datagram socket with reliable delivery.

The Sun's RPC Protocol is independent of transport protocols. It does not care how a message is passed among processes. It only deals with the specification and interpretation of messages. For an application which does not care about reliability, message passing using UDP/IP (User Datagram Protocol) can be specified. In constrast, RPC using TCP/IP (Transmission Control Protocol) must be used to guarantee reliability. UDP is implemented using Datagram sockets while TCP is built on Stream sockets. Currently, Sun's RPC only supports UDP and TCP transports [SunRPC86].

3.3. Sun Microsystems' Network File System

4.3BSD Unix is not a distributed operating system. Nonetheless, one of the goals of my thesis is to be able to work in a heterogeneous computing environment. Information sharing is neccessary. Fortunately, Sun Microsystems' Network File System is available. This allows file access transparency. Machine, operating system, network architecture, and transport protocol independence are provided by this NFS protocol [SunNFS86].

Through the use of remote procedure call primitives built on top of eXternal Data Representation, this independency is achieved.

3.4. EXternal Data Representation

3.4.1. Justification

Data format incompatability often occurs in a heterogeneous environment. Whenever data is passed back and forth among two or more machines of

different architecture, there is a need for translating data formats. Sun's approach to this is with an intermediate, in this case a "*standard*", network format. Data coming out from machine A, before being sent to machine B, is converted from machine A's format to the network standard format. When received at machine B, conversion is made from the network standard format to machine B's format. Sun provides an eXternal Data Representation (XDR) Protocol for data-portability purpose.

3.4.2. Sun XDR

The dual purpose of XDR is to provide an architecture independent representation of data types, and to provide an encoding for data structures that are passed as arguments. XDR uses a complementary set of procedures to encode an arbitrary C data structure to a byte stream and a matching procedure to decode the byte stream back to a data structure.

Among the arguments of a call in the second or third layer of Sun RPC are the XDR procedures and addresses of the input to, and output from, the service routine as described in an earlier section. Using RPC, the types of input and output arguments are specified in terms of filters supplied by the Sun XDR library. Users can also construct special-purpose filters to fit their own needs, using the available primitives. See table 3.1 for examples of available primitives.

Table 3.1 - XDR Types		
XDR Type	C Primitive	Data type
xdr_int	integer	32-bit integer
xdr_long	integer	64-bit integer
xdr_short	integer	16-bit integer
xdr_u_int	unsigned	32-bit unsigned
xdr_u_long	unsigned	64-bit unsigned
xdr_u_short	unsigned	16-bit unsigned
xdr_float	float	32-bit real
xdr_double	double	64-bit real
xdr_enum	enum_t	enumerated integers
xdr_bool	bool_t	boolean integer(0/1)
xdr_string	string	string
xdr_bytes	string	8-bit char
xdr_array	array	arrays
xdr_opaque	opaque	uninterpreted data
xdr_union	union	union
xdr_reference	pointer	pointer chasing structures
xdr_void	-	no data

The XDR standard is independent of operating systems and hardware architecture. This eXternal Data Representation standard assumes that bytes, quantities of eight bits of data, are portable, i.e. byte's meanings across hardware boundaries are preserved. In transmitting data accessed by different types of machines XDR must be used.

The basic block size is four bytes. They are numbered 0 through n-1, where (n mod 4) = 0. These bytes are read from or written to a stream in increasing order, i.e. starting with 0 and going up to n-1. An XDR signed integer is represented in two's complement notation. Its least significant byte is numbered 3 while the most significant is 0. Floating point numbers, are encoded using the IEEE standard:

– bit 0, the most significant, is the sign bit,

– bits 1-8 (float), or 1-11 (double), are the base 2 exponent,

– bits 9-31 (float), or 12-63 (double), are the base 2 fractional part of the mantissa.

On the client side, when a service is requested, the XDR input procedure will serialize the arguments, and on the server side deserializing will take place in the XDR input procedure, upon receiving the service request. Upon completing the request and returning to the client/caller, the XDR output procedure will serialize the return data, and deserializing will be handled by the XDR output procedure on the client side, upon getting the reply [SunXDR86].

Chapter 4

Design and Implementation

4.1. Major Issue

Parallel processing is the key issue for my proposed method. Providing concurrency within the semantics of synchronous procedure calls is the goal.

4.2. Major Components/Processes

To accomplish the work of ray tracing an image , the following tasks are identified:

- search for an available processor,
- distribute the units of work,
- trace rays, (the actual work),
- collect results, (the output from ray tracing),
- check progress, and recover work lost if necessary.

The goal here is to be able to execute the ray tracing routine concurrently on several processors. Using remote procedure calls the client will issue requests to the servers to trace rays. In order to establish concurrency the client, after issuing a request, should not have to wait for the result. Instead, the initiating RPC should return with the server sending a void message as soon as the request is received, so that the client can regain control and continue distributing pieces of the job. The results generated by the ray tracing servers will be sent to one common destination. This destination must have the information on how to

reconstruct the entire image, since the result could be collected in a random order. Three major components/processes will be described in this chapter:

– the distribution client,

– the ray tracing server,

– the collection server.

4.2.1. The distribution client

This is the main control process. Its functions are:

4.2.1.1. Checking Status of Workstation

The ray tracing servers and the collection server were started before the kistribution client. Before work is given out to a workstation its availability must be checked. A remote procedure call of the first layer type will be issued to the next workstation in a list of available machines to get back the idle time. These machines are connected to the network. A threshold of idle time value is used to determine availability. The percent of idle time is calculated based on the result returned by this RPC call. The returned value will be compared to the threshold value.

4.2.1.2. Distributing Units of Work

Once the status of the workstation is determined as being available, a unit of work will be distributed to this processor. Otherwise, the status of the next processor in the list will be checked. A unit of work is a block of scanlines. Work is given out in order, starting with block 0, and ending with the last block. The blocksize is a compilation constant in the distribution client.

4.2.1.3. Checking Progress and Recovering Errors

After all work has been distributed, the distribution client will start checking progress by issuing another RPC to the process that collects the ray tracing results. The collection server will be discussed as the third process, in a later section. The collection server responds to this progress query by sending a list of the result units that have been collected. If this list indicates that there are still some result units that have not arrived, then the time stamped by the distribution client when these units were distributed will be checked against the current time. An estimate for completing a unit of work will be used to determine that the result may have been lost, or a workstation became unavailable, etc... If the estimated time for completing units of work is much less than the elapsed time since the work was distributed, then these work units will be redistributed.

4.2.1.4. Initiating Image Reconstruction Process

Upon receiving the indication that all scanline results have been collected from the collection server, the distribution client will issue a call to the Collect server to construct the completed image.

4.2.1.5. Unregistering Services

When the collection server finishes the image reconstruction, it will notify the distribution client. Upon getting this notification, the distribution client will start the cleaning up process before exiting, by issuing calls to unregister services on the ray tracing servers and the collection server, and terminating processes on the servers.

4.2.2. The ray tracing server

The ray tracing server carries out the actual work of tracing rays. Multiple processors are used as ray tracing servers for this project. All of these servers have the same set of functions:

4.2.2.1. Generating Input for Ray Trace Routines

From the user-defined viewing parameters, a file consisting of viewpoint and ray directions are generated. Among viewing parameters are view point, view up, view direction vectors, view angles, view plane, etc....

4.2.2.2. Tracing Rays

The ray trace routines are executed for the block of scanlines requested. Results are written into a temporary file.

4.2.2.3. Sending Results to the collection server

As soon as the ray tracing server finishes tracing the block of scanlines, it will initiate an RPC call to the collection server (discussed in the next section) to send the result to the common destination.

4.2.3. The collection server

In the RPC paradigm, while the thread of control is intertwined between two processes, only one process can be active at a time. If the Distribute server is to wait for the ray tracing servers in order to collect the result, it could not do any other job, such as continuing the work distribution. Hence, the ray trace server upon receiving the processing request, has to reply immediately, i.e. send a void message to the distribution client, before starting to perform the actual requested

work.

The result, generated by the ray tracing server, is stored on the server side as separate files for each unit of work done. A record of the completed scanlines is kept in order to allow the user to reconstruct the final image. This could be cumbersome for the user. In addition to the two processes just described, a third one is introduced, the collection server, as an answer to this issue. Among its functions are:

4.2.3.1. Collecting Results

The ray tracing servers, when finished with the work requested, will send the results to this collection server. These results will be written into a file in the order they come in. (The results of blocks of scanlines do not necessarily arrive at the collection server in the same order they are given out by the distribution client.)

4.2.3.2. Updating Progress Report

A list of the blocks of scanlines collected will be kept as the progress report. When a request arrives from the distribution client querying the progress, this list is sent as the reply.

4.2.3.3. Reconstructing Image

A data structure which contains information such as the starting and ending position in the result file for each block of scanlines is maintained by this server for image reconstruction. Run-length encoding is used by the ray trace routine, therefore, for blocks of the same number of scanlines, the number of bytes needed to store the results could be different in length. Upon receiving the

progress report from this collection server, with a success result indicating that all results have been collected, the distribution client will issue an RPC call to this collect server to reconstruct the final image.

4.3. Flow of Control

The following two figures serve as a more detailed description of the distributed architecture. The communication among the three components is summarized in Figure 4.1. Figure 4.2 illustrates the control thread of the entire task in which concurrency is provided within a synchronous procedure call paradigm and how error recovery is accomplished.

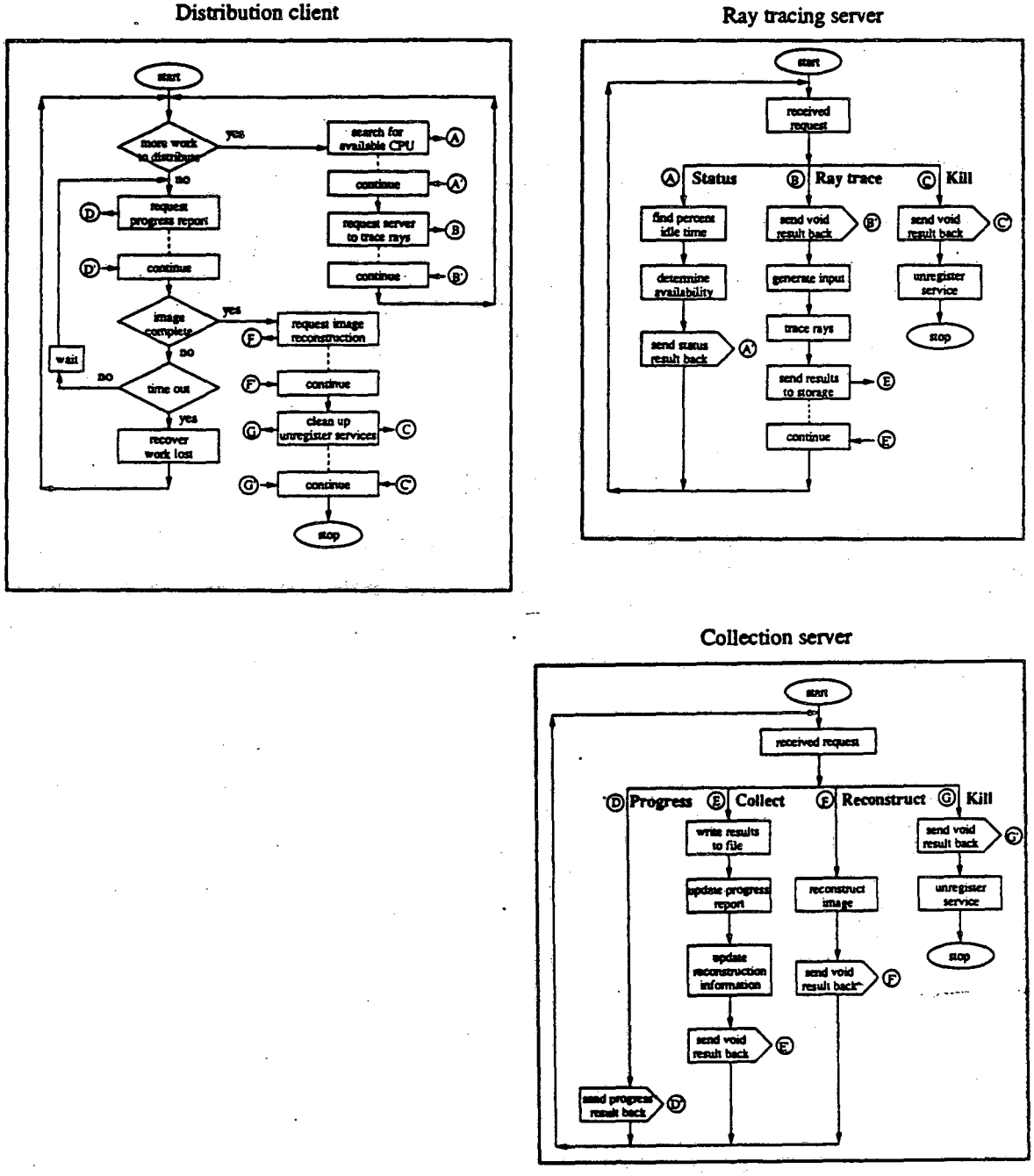


Figure 4.1 : Distributed Architecture

Distribution client

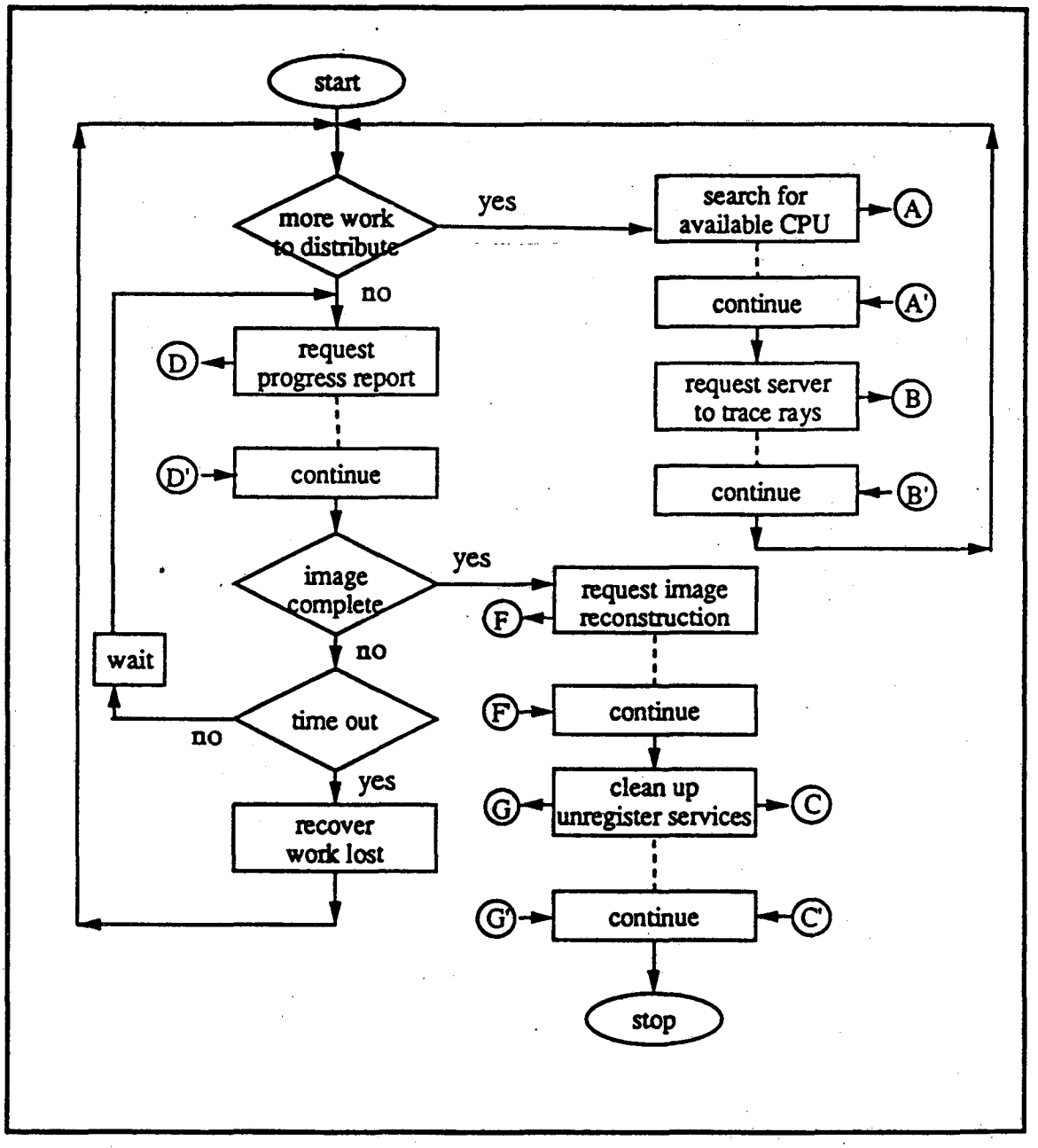


Figure 4.1.1

Ray tracing server

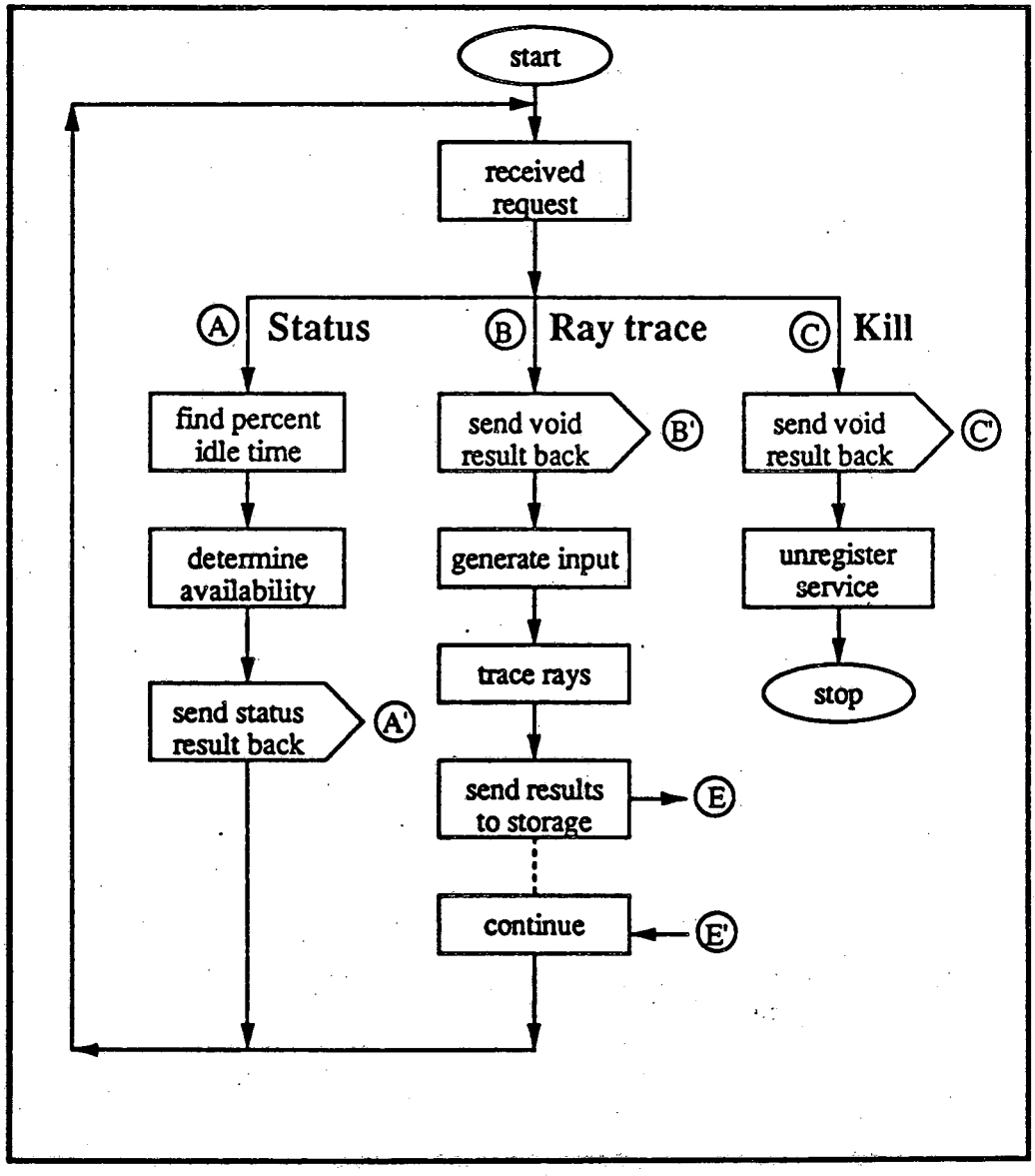


Figure 4.1.2

Collection server

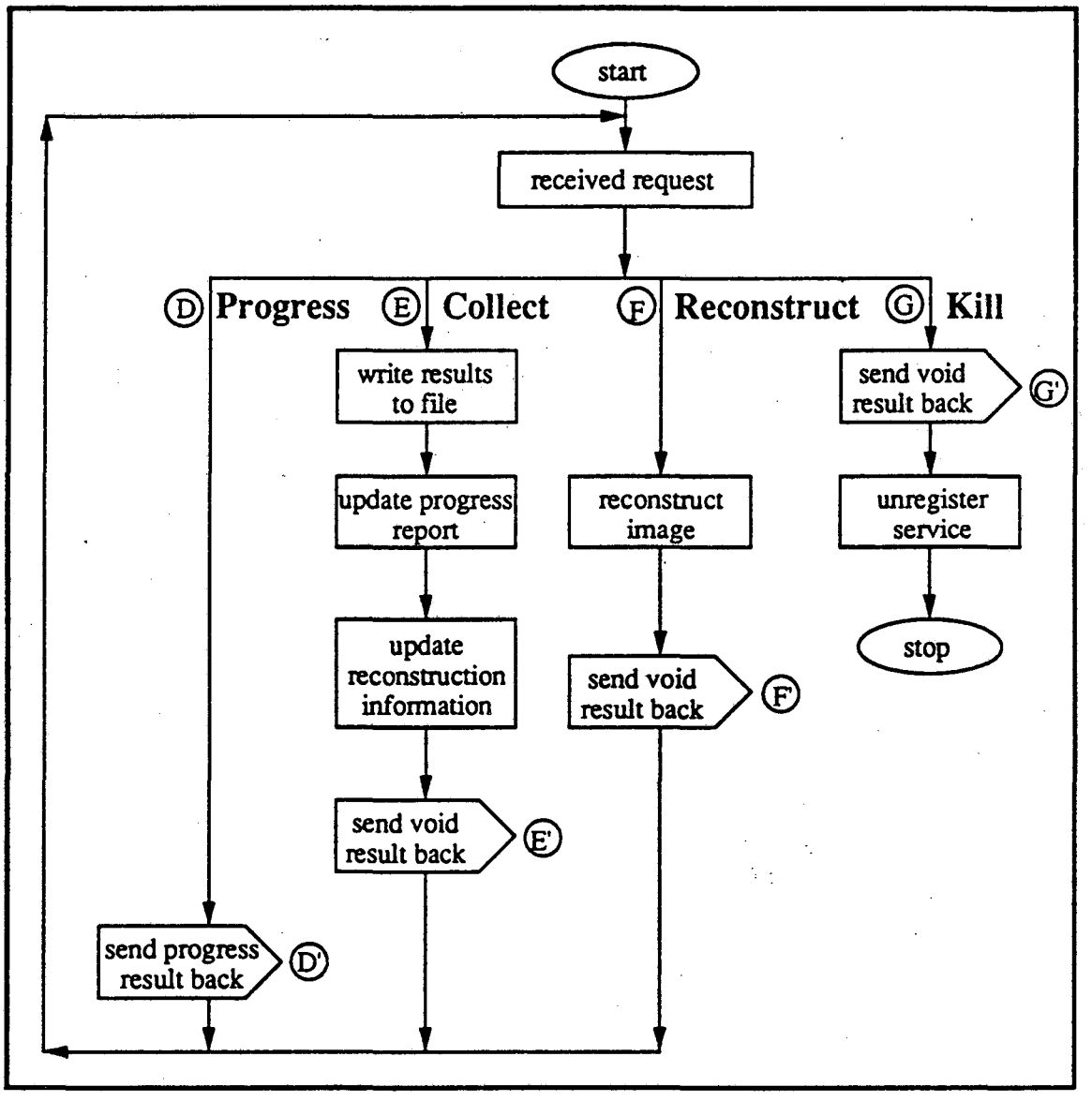


Figure 4.1.3

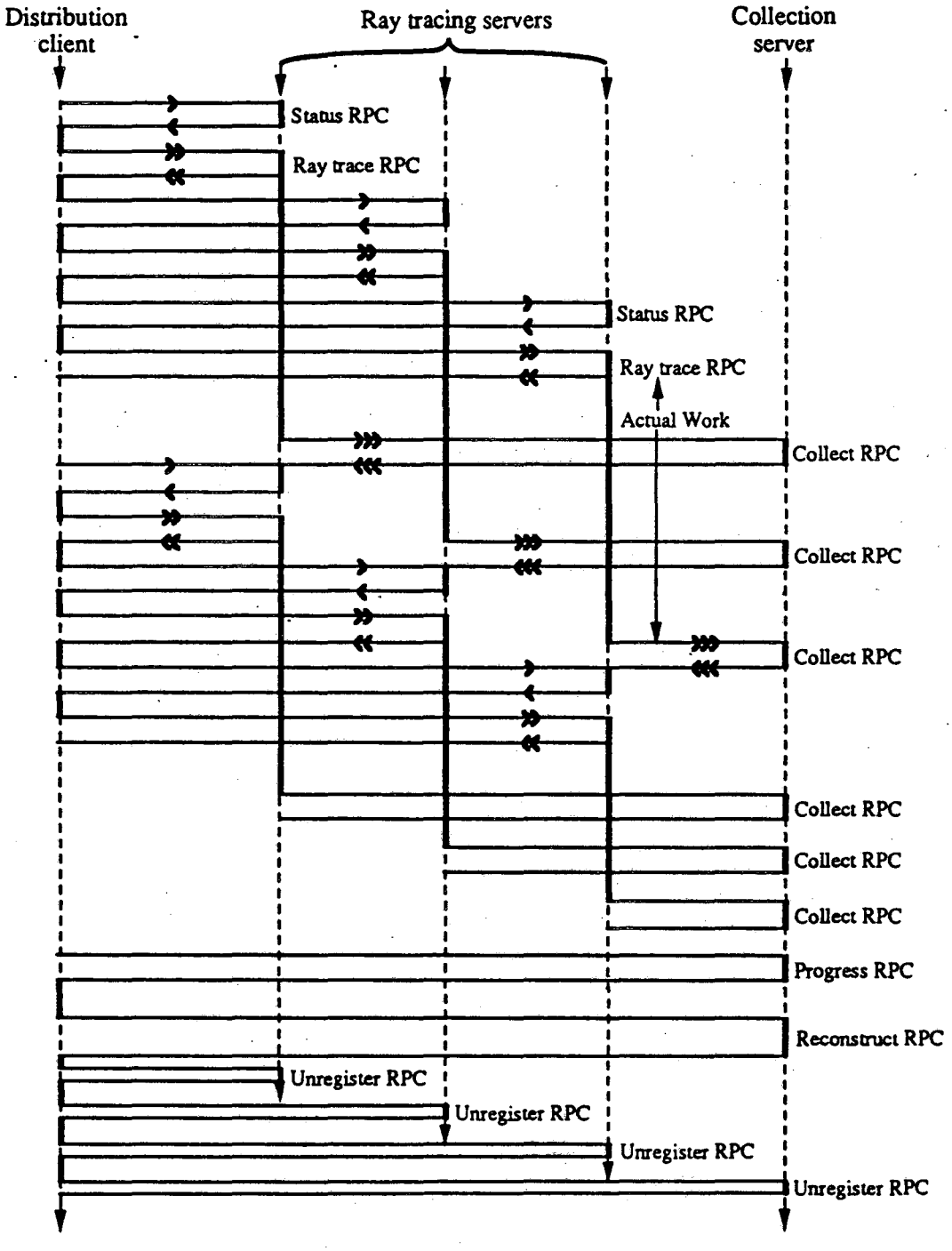


Figure 4.2 : Flow of control

4.4. RPC Programming Issues

4.4.1. Chosen Protocol

Among the five different network communication protocols, Sun's RPC only supports two of them: datagrams and streams. Less overhead is incurred by datagrams (UDP). Datagram communication does not use connections, each message is addressed individually. If the address is correct, the message will usually be received. There is a system dependent limitation on the size of the message. The upper limit for message length is 1Kbyte on the VAX. Depending on the size of the blocks of scanlines and the format of the output, the result from the ray tracing server could be longer than the upper limit for UDP. The datagram protocol is not reliable. Lengthy messages are sent using the reliable stream protocol, provided by TCP/IP (Transmission Control Protocol). TCP is also chosen for its reliability, any messages received with errors will be retransmitted, and for the fact that the timeout per try, as well as the number of tries for each call, can be controlled.

4.4.2. Chosen layer of RPC

Since socket manipulation is necessary, the third layer of RPC is utilized.

4.4.2.1. On the distribution client

There is a limitation on the number of open socket descriptors, i.e. the number of communication channels. (According to the library function description "*servers - inet server data base*", at the time this thesis was written, Sun limits this number to 27). One has to adapt to one of the two alternatives: release the socket after every call, or leave the communication channel open once it is

established and only close it upon the last call to the server. With the first alternative, the communication will still linger for at least 4 seconds after a close call was issued to shutdown the socket. For this reason the second alternative was chosen; the client handles established for the first call will be saved and reused until the last call is made to unregister the service. At that time these handles will be destroyed.

4.4.2.2. On the ray tracing servers

Due to the synchronous nature of RPC, a void reply is sent to the Distribute client immediately after the ray tracing servers receive messages to trace rays, so that concurrency can occur.

4.4.3. Chosen XDR

A standard network data format plays an important role in a heterogenuous environment. Three different XDR routines are utilized to accomplish this. These routines are built using the existing XDR library.

4.4.3.1. Input to Ray Trace Routines

For the call to request the ray tracing server to trace rays, an XDR routine is used to serialize the argument data structure (input). The input is sent as a byte stream, and it consists of the scanline block number, the names for the octree scene description file, and the viewing parameter file. XDR for this input structure utilized the integer and string XDR primitives.

4.4.3.2. Output from Ray Trace Routines

The output from the ray tracing server is transfered to the collection server as another data structure. This output consists of a scanline block number, the

number of bytes of the result, and the pixels that make up the scanlines. Integer XDR, and opaque XDR are used.

4.4.3.3. Progress in Collecting Results

The last type of data structure which is sent over the network is the progress report structure. This structure is an array whose index is the scanline block number. Each element of this array is of type boolean, which indicates the status of the block: collected or not. Only boolean XDR is needed.

4.4.4. Miscellaneous

Two further issues were encountered during the design:

4.4.4.1. Broadcast RPC

This option of RPC could be used in searching for more than one available processors in one call. Broadcast RPC expects more than one reply, while normal RPC expects only one. For simplicity, one mechanism of RPC, the normal RPC, is used through out the entire project. Mismatch of RPC library versions between client and servers are not notified by broadcast RPC due to its implementation which treats unsuccessful reponses as garbage, and filters them out.

4.4.4.2. Batching RPC

Normal RPC is designed such that the clients wait for the servers to reply, i.e. clients must be silent while servers process a call. Batching facilities provide the possibility for clients to continue computing while waiting for replies. No messages will be sent by the servers, so failures are not notified. As a result, clients have to provide error checking. Batching was not used.

Chapter 5

Result and Effectiveness Analysis

5.1. Performance of an Asynchronous Distributed Environment

The Unix *time* command was used to obtain the elapsed CPU and the real time for the timing alnalysis.

Data was collected for ray tracing a specific picture, an image of a metallic gold sphere (Figure 5.1). Using the ray tracing program, in a uni-processor environment, data was collected on different CPUs. Time taken to generate this image ranged from 1.72 to 3.31 hours for a group of Sun 3 workstations. These non-distributed timings were used to estimate the speed up ratios for the distributed case.

For the same image, data was collected using different numbers of ray tracing servers and different work unit sizes. CPU and Real time estimates were done using the time taken for the least powerful server in each case, using the non-distributed method, plus the estimated overhead. The overhead were estimated based upon the result of running *prof* on the executables. *prof* is the Unix command for obtaining procedure level execution time profiles of programs. The differences between the results collected and estimated were tabulated using the estimates as the bases.

Statistics in seconds are listed in table 5.1. In table 5.2, these data were interpreted in speedup ratios. Note that in most cases the speed-up ratios are in

between n-1 and n, where n is the number of servers used in each case. Data was collected for up to 7 servers, the limit of servers available to the work of this thesis.

Table 5.1 Time results (seconds)							
WorkUnit	# servers	Real time			CPU time		
		Est	Act	% diff	Est	Act	% diff
20 sl	2	4107	4090	-0.4	4047	3898	-3.7
	3	3699	4228	+10.1	3619	3531	-2.4
	4	2666	3176	+8.1	2586	2574	-0.5
	5	2170	2286	+16.1	2090	2146	-16.5
	6	2162	2087	-3.6	2082	1918	-7.9
	7	1666	1753	+5.0	1586	1501	-5.1
10 sl	2	4180	4265	+2.0	4100	3930	-4.1
	3	3483	3929	+11.3	3403	3304	-2.9
	4	2748	2790	+1.5	2668	2575	-3.5
	5	2187	2194	+0.3	2107	1998	-5.2
	6	1920	1955	+1.8	1840	1753	-4.7
	7	1678	1755	+4.4	1598	1522	-4.8
5 sl	4	2762	2898	+4.7	2682	2568	-4.2
	5	2242	2478	+9.5	2162	2082	-3.7
	6	1963	2071	+5.2	1883	1744	-7.4
	7	1705	1857	+8.2	1625	1543	-5.0

Table 5.2 Time results (speed–up ratios)							
WorkUnit	# servers	Real time			CPU time		
		Est	Act	% diff	Est	Act	% diff
20 sl	2	1.94	1.94	+0.0	1.97	2.04	+3.4
	3	2.75	2.40	-14.6	2.87	2.88	+0.3
	4	3.81	3.20	-19.1	3.93	3.94	+0.4
	5	4.68	4.44	-5.4	4.86	4.73	-2.6
	6	4.71	4.87	+3.3	4.87	5.29	+8.6
	7	6.09	5.79	-5.2	6.41	6.78	+5.7
10 sl	2	1.90	1.86	-2.2	1.94	2.02	+5.7
	3	2.92	2.58	-13.2	2.98	3.07	+3.1
	4	3.70	3.64	-1.6	3.81	3.94	+3.5
	5	4.65	4.64	-0.2	4.82	5.08	+5.4
	6	5.29	5.19	-1.9	5.53	5.80	+5.0
	7	6.05	5.79	-4.5	6.35	6.67	+5
5 sl	4	3.68	3.50	-5.1	3.93	3.95	+0.6
	5	4.53	4.10	-10.5	4.70	4.88	+3.7
	6	5.17	4.90	-5.5	5.39	5.82	+8.0
	7	5.97	5.48	-8.9	6.26	6.59	+5.5

5.2. Factors that might effect Performance

There are several factors that effect the time taken to generate an image. These are:

5.2.1. Number of ray tracing servers

It is only logical that this number has to be greater than one. Together with the number of scanlines that makes up the block (unit of work), the number of servers and the blocksize determine the maximum number of work units which each server will be requested to do in order to complete an image of a given resolution. For example, for a 512 X 400 image, if the blocksize is 20, there will be 20 work units to be distributed. If there are 5 servers available, and if there is no error recovery needed, then each server will be requested to perform 5 work units. In the case where work balancing is not done ideally, some servers will need to perform more work than others, for example, if there are 6 servers, and there are 20 work units, if there is no error recovery the time required to complete the image is the time taken for those servers that received requests for 4 units.

The number of active descriptors or the number of communication channels is limited to 28, less one for the Collect server, due to a reason mentioned earlier. The number of accessible servers are also limited. The maximum number servers used is seven.

5.2.2. Size of Work Units

The number of scanlines that makes up a block, a unit of work, is another factor that affects the performance. Small blocks take less time to be

accomplished, and if work loss occurs less time is required to redo one unit of work. More overhead for making RPC calls will be needed since the number of calls increases. RPC overhead will be discussed in a later section. Work balancing is one of the keys for choosing the blocksize.

5.2.3. Differences in Individual Server Performance

A group of Sun workstations were used as ray tracing servers. These workstations possess different computing power, hence the time taken to finish one work unit varies from server to server.

5.2.4. Work Balancing

Among the factors that affect the efficiency of the method implemented is the need to obtain an even work distribution for all servers, i.e. all servers optimally receive the same number of work units.

5.3. Models Used in Estimating Performance

For performance guidelines, three models were used:

1. A single CPU in a non-distributed environment.
2. More than one CPU in a distributed environment with infinitely fast data transfer rate.
3. A data transfer rate of 10 KBytes per second across the network was assumed in the last model. Each pixel required 4 bytes.

For the chosen picture, the resolution is 512 X 400, and a total of 800 KBytes were transfered across the network. The estimated data transfer time was 80 seconds. In model two, programs' profiles were used to estimate the RPC and I/O overhead. In addition to this overhead, the waiting time between calls to

obtain the servers' status had to be included.

5.4. Estimated Time versus Obtained CPU Results

The obtained CPU results were compared against the time taken for the least powerful server among the group of servers to perform the work requested. The estimated data was calculated using the CPU times of the slowest servers and the overhead described in model 2. The measured time was consistently lower than the estimated CPU time.

5.5. Estimated Time versus Obtained Real-time Results

The obtained Real-time results were compared against the time taken for the least powerful server among the group of servers to perform the work requested, plus the time taken for the server availability checks, plus the data transfer time described in model 3. As expected, the differences in the estimated and collected results were larger than the CPU results.

5.6. Justification for Differences in Comparison

In calculating the estimated data, it is assumed that the servers will be dedicated for ray tracing. However, in reality, the Sun workstations were also being used for other tasks. As the result, the results collected could be significantly different from the estimate values.

For the distributed environment, one must take into account the overhead of RPC, XDR, I/O etc.... Files' profiles (see appendix B), obtained using *prof* were analyzed and the results were tabulated in table 5.3 (in seconds) and table 5.4 (in percentage). As expected, the overhead increases as the number of servers

increases. In addition, as the size of work unit grows, the overhead grows.

Table 5.3 CPU profile (seconds)							
WorkUnit	#servers	RPC+XDR	XDR	IO	Wait	Overhead	Total
20 sl	2	5	1	11	50	67	3989
	3	6	0	12	35	53	3531
	4	6	1	12	25	43	2574
	5	8	1	12	20	39	2046
	6	8	1	13	20	41	1918
	7	6	1	13	15	34	1505
10 sl	2	7	1	13	10	120	3930
	3	9	1	13	70	91	3304
	4	9	1	14	50	72	2575
	5	8	1	12	40	60	1998
	6	12	1	10	35	57	1753
	7	11	1	13	30	54	1522
5 sl	4	17	2	15	100	132	2568
	5	16	2	14	80	110	2082
	6	19	2	14	80	11	1744
	7	20	1	14	60	94	1543

Table 5.4 CPU profile (percentage)							
WorkUnit	#servers	RPC+XDR	XDR	IO	Wait	Overhead	Total
20 sl	2	0.13	0.02	0.28	1.25	1.68	100.00
	3	0.16	0.01	0.35	0.99	1.50	100.00
	4	0.23	0.02	0.47	0.97	1.67	100.00
	5	0.38	0.04	0.57	0.98	1.91	100.00
	6	0.41	0.03	0.68	1.04	2.14	100.00
	7	0.39	0.04	0.88	1.00	2:26	100.00
10 sl	2	0.17	0.02	0.34	2.54	3.05	100.00
	3	0.27	0.02	0.38	2.11	2.75	100.00
	4	0.33	0.05	0.53	1.94	2.80	100.00
	5	0.41	0.05	0.61	2.00	3.00	100.00
	6	0.68	0.06	0.57	2.00	3.25	100.00
	7	0.70	0.08	0.87	1.97	3.55	100.00
5 sl	4	0.63	0.07	0.56	3.89	5.14	100.00
	5	0.76	0.08	0.67	3.84	5.28	100.00
	6	1.07	0.10	0.80	4.01	5.91	100.00
	7	1.27	0.08	0.92	3.89	6.09	100.00

5.6.1. RPC Overhead

Time taken to establish communication links between clients and servers is part of the overhead. Up to 15.00% of the total overhead, or up to 1.30% of the total time taken to finish tracing the picture, was spent as RPC cost. As expected, the RPC overhead increases as the size of work unit decreases due to the increase in the number of RPC calls made.

5.6.2. XDR Cost

Decoding and encoding the input and output arguments by the client and servers contributes to the diferences between the estimates and the collected results. Approximate 10.00% of the RPC cost was spent in eXternal Data Representation encoding and decoding. In table 5.3 and 5.4, XDR cost was included in the RPC cost.

5.6.3. Sleeping Periods between Status Calls

The output of the RPC calls that check for status availability is in cummulative numbers, hence it is required to make two calls and take the difference in the two results. For each status availability check a sleeping interval of 5 seconds (real time) is used. These waiting periods are shown in the wait column of table 5.3 and 5.4.

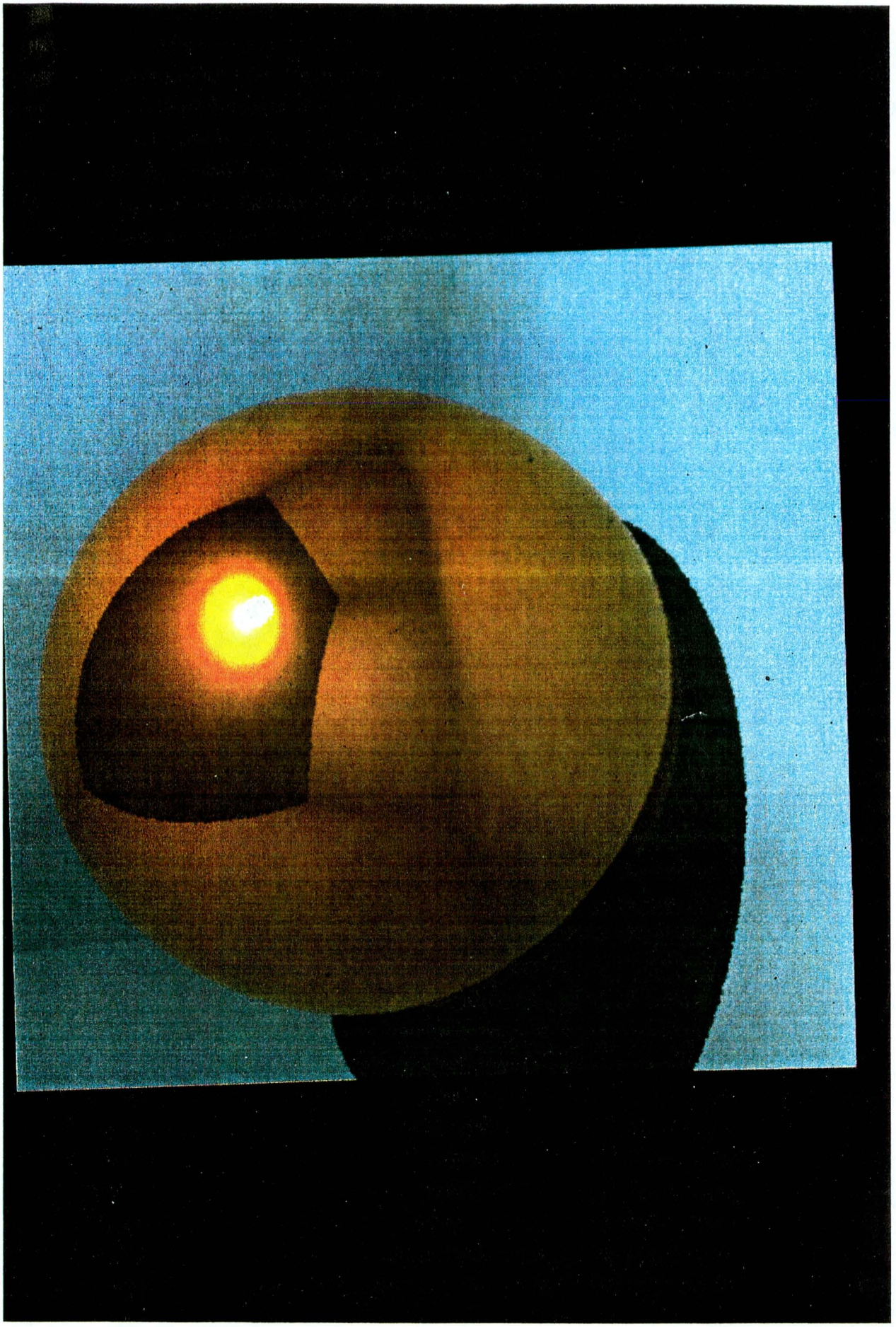
5.6.4. I/O Operations

A significant portion of the time taken to accomplish an image is spent on input and output operations. Viewing parameters are read in from a file. Input generated for the ray tracing routine are written into a file and later are read in.

Output from the ray tracing routine is written into a file before being sent to the Collect server. On the Collect server, at first, the result is written into a file in random order. Finally, after receiving the request to reconstruct the file, I/O operations are invoked for data transfer. This I/O cost could taken up to 1.00% of the total CPU time collected to trace the chosen picture.

5.6.5. Progress Checking

Error checking is important. Fortunately, during the data collection stage there was no incidence in which a ray tracing server went down in the middle of carrying the requested work. Error checking could play a significant roll in performance evaluation, especially when work lost occurs often and recoveries are frequently needed.



Chapter 6

Conclusion

6.1. Meet Design Goals

Improvements were achieved using the proposed method. CPU time speed-up ratios of up to 6.78 were obtained in the case where seven servers were used. Distributed processing was proven to be effective in ray tracing. The work done on this thesis could be improved to achieve even higher speed up ratios.

6.2. Further Improvements

6.2.1. Upper limit on the number of servers

Due to the availability of the number of Sun workstations, the maximum number of servers used was limited to seven. This limit could be higher. With more servers available, the number of distributing cycles would be reduced because the CPU time to do the work completely dominates the overhead. An ideal situation could be met in which each server would have to perform the work on one work unit.

If the number of servers increased the size of the work unit could be reduced so that all available servers would be utilized.

6.2.2. Screen Saver for Status Check

The workstation status check could be done using a different method, so that the time spent in the sleeping periods between to RPC status calls could be eliminated. A good alternative for checking the status is a screen saver program, used to monitor the time elapsed since the keyboard was touched.

6.2.3. Work Balancing

The distributing process as set up is synchronous, i.e. work is given out to server1, server2,...., servern (last server in list), then back to server1. Due to the variation in performances among workstations, there were servers that sat idle while waiting for the next request.

To minimize the idle time among servers, i.e. to achieve even higher speed up ratio, it is worthwhile to consider the two alternatives:

1. Using servers of the same performance.

2. Keeping tracks of the pool of available servers: As soon as the ray tracing server finished sending the result to the collection server, it should notify the distribution client of its availability. The distribution client would then put this server in the pool. After the first cycle, work would be given to the first available server in this pool.

6.2.4. Eliminate Image Reconstruction

A significant amount of time is spent for reconstruction of the final image, due to the asynchronous nature in collecting results. If servers of similar

performance are used, there is a good chance that the result will be received in order, hence no reconstruction would be needed.

References

[Appel68]

Appel, A., Some techniques for Shading Machine Renderings of Solids. AFIPS 1968 Spring Joint Computer Conference, pp 37-45.

[Cook84]

Cook Robert L., T. Porter, L. Carpenter, Distributed Ray Tracing. ACM SIGGRAPH Conference Proceedings, July 23-27, 1984 Minneapolis, Minnesota, vol. 18, n. 3, pp 13-144.

[Dippe84]

Dippe, M., J. Swensen, An adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. ACM SIGGRAPH Conference Proceedings, July 23-27, 1984 Minneapolis, Minnesota, vol. 18, n. 3, pp 148-158.

[Foley82]

Foley, J. D., and A. van Dam, Fundamentals of Interactive Computer Graphics. Addison-Wesley Publishing Company Inc., 1982. pg 585. 1982.

[Glassner84]

Glassner, A. Space Subdivisions for Fast Ray Tracing, IEEE CG&A, vol. 4, n. 10, October 1984, pp 15-22.

[Haines86]

Haines, Eric A., and Donald P. Greenberg, The Light Buffer: A Shadow-

Testing Accelerator. IEEE CG&A, vol. 6, n. 9, September 1986, pp 6-16.

[Kobayashi87]

Kobayashi, H., T. Nakamura and Y. Shigei, Parallel Processing of an object space for image synthesis using ray tracing. The Visual Computer, 1987(3), pp 13-22.

[Meagher82]

Meagher, Donald, Geometric Modeling Using Octree Encoding. Computer Graphics and Image Processing, vol. 19, 1982, pp 129-147.

[Rogers85]

Rogers, D. F., Procedural Elements for Computer Graphics. McGraw-Hill Book Company, New York. 1985, pg 296.

[Roth82]

Roth, S. D., Ray Casting for Modeling Solids. Computer Graphics and Image Processing, vol. 18, 1982, pp 109-144.

[SunXDR86]

Sun Microsystems, External Data Representation Protocol Specification. Sun 3.0 Documents, Revision B of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

[SunIPC86]

Sun Microsystems, InterProcess Communication Primer. Sun 3.0 Documents, Revision B of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

[SunNFS86]

Sun Microsystems, Network File System Protocol Specification. Sun 3.0 Documents, Revision B of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

[SunRPC86]

Sun Microsystems, Remote Procedure Call Programming Guide. Sun 3.0 Documents, Revision B of 17 February 1986. Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043.

[Ward86]

Ward, G., The RADIANCE Synthetic Imaging System. Lawrence Berkeley Laboratory Publication, 1986.

[Whitted80]

Whitted, T., An Improved Illumination Model for Shaded Display. Communication of the ACM, vol. 23, n. 6, June 1980, pp 343-349.

Appendix A: Program Listings

Notice of Copyright

```
/* Copyright (c) 1986 Regents of the University of California */

/*
 *  color.h - header for routines using pixel color values.
 *
 *      12/31/85
 *
 *  Two color representations are used, one for calculation and
 *  another for storage.  Calculation is done with three floats
 *  for speed.  Stored color values use 4 bytes which contain
 *  three single byte mantissas and a common exponent.
 */

#define  RED            0
#define  GRN            1
#define  BLU            2
#define  EXP            3
#define  COLXS          128        /* excess used for exponent */

typedef unsigned char  BYTE;       /* 8-bit unsigned integer */

typedef BYTE  COLR[4];             /* red, green, blue, exponent */

#define  copycolr(c1,c2)           (c1[0]=c2[0],c1[1]=c2[1], \
                                    c1[2]=c2[2],c1[3]=c2[3])

typedef float  COLOR[3];           /* red, green, blue */

#define  colval(col,pri)           ((col)[pri])

#define  setcolor(col,r,g,b)       ((col)[RED]=(r),(col)[GRN]=(g),(col)[BLU]=(b))

#define  copycolor(c1,c2)          ((c1)[0]=(c2)[0],(c1)[1]=(c2)[1],(c1)[2]=(c2)[2])

#define  scalecolor(col,sf)        ((col)[0]*=(sf),(col)[1]*=(sf),(col)[2]*=(sf))

#define  addcolor(c1,c2)           (c1[0]+=c2[0],c1[1]+=c2[1],c1[2]+=c2[2])

#define  multcolor(c1,c2)          (c1[0]*=c2[0],c1[1]*=c2[1],c1[2]*=c2[2])

#define  WHTCOLOR                  {1.0,1.0,1.0}
#define  BLKCOLOR                  {0.0,0.0,0.0}
#define  WHTCOLR                   {255,255,255,COLXS}
#define  BLKCOLR                   {0,0,0,0}

extern double  intens();
```

```
/*****************************************************************************/
/*
/* FILE          : distribute.h
/*
/* DATE          : 8/27/87
/*
/* AUTHOR        : Mydung Thi Tran
/*
/* DESCRIPTION : this include file contains :
/*                 - PROGRAM NUMBER, PROGRAM VERSION, PROCEDURE NUMBER
/*                   which are used in various Remote Procedure Calls.
/*                 - constants for exit codes,
/*                 - constants for resolutions, threshold etc....,
/*                 - structures for xdr routines.
/*
/*****************************************************************************/

#include <stdio.h>
#include <netdb.h>
#include <sys/time.h>
#include <signal.h>
#include <errno.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>
#include <sys/wait.h>
#include <sys/socket.h>
#include "color.h"

extern long random();                   /* forward declaration for random call */

                /* program numbers, versions, procedure numbers */

#define RAY_PROG          21000000 /* program number for SUN servers    */
#define RAY_VERS          2        /* program version for SUN servers   */
#define RAY_STAT_PROC     20       /* procedure for CPU status info     */
#define RAY_RTRACE_PROC   21       /* procedure for tracing rays        */
#define RAY_KILL_PROC     22       /* procedure for unregister service  */

#define COL_PROG          31000000 /* prog.# for collection server on VAX */
#define COL_VERS          3        /* version #                          */
#define COL_QUERY_PROC    30       /* procedure for completion list query */
#define COL_COLCT_PROC    31       /* procedure for collecting scanline result */
#define COL_KILL_PROC     32       /* procedure for unmap child server service */
#define COL_RECON_PROC    33       /* procedure for reconstructing raster image */

                /* contants for distributed processing */

#define MAXNUMOFHOST      2        /* maximum number of workstations */
#define XRES              512      /* x resolution */
#define YRES              400      /* y resolution */
#define BLOCKSIZE         20       /* number of scanlines in each block */
#define BLOCK             (XRES*BLOCKSIZE)
#define NUMBLOCK          (YRES/BLOCKSIZE)
#define THRESHOLD_PRCT    50       /* threshold value for percent idle time */
#define SCANLINE_LIMIT    40       /* time taken to trace each scanline */
#define TIME_LIMIT        (SCANLINE_LIMIT * BLOCKSIZE)


#define NOT_YET           0        /* scanline to be done */
#define DONE              1        /* scanline already done */
#define YES_QUERY         1        /* yes, query for progress */
#define NO_QUERY          0        /* no, don't query for progress */
```

```
#define REDO                1       /* some scanlines need to be redone */
#define IMAGE_COMPLETE      0       /* result for entire image collected */
#define IDLE                1       /* workstation is idle */
#define BUSY                0       /* workstation is busy */

                    /* constants for ray tracing routine */

#define FTINY               1e-7    /* small real number */
#define FHUGE               1e10    /* large real number */
#define dstrpix             0.67    /* square pixel distribution */
#define PI                  3.14159265358979323846
#define DOT(v1,v2)          (v1[0]*v2[0]+v1[1]*v2[1]+v1[2]*v2[2])
#define frandom()           (random() /2147483648.0)
#define pixjitter()         (dstrpix * (0.5 - frandom()))

                    /* constants for exit codes */

#define SUCCESS             1       /* congragulation, you have succeeded */
#define FAILURE             0       /* sorry */

typedef char        HOSTNAME[20];
typedef COLR        SCANBUF[XRES*BLOCKSIZE];
typedef double      FVECT[3];

/* the following structures are for xdr routines */

struct input_rt                     /* input to RTRACE */
    {
        int     curr_block;         /* current block number */
        char    *scene_file;        /* scene description filename */
        char    *view_file;         /* viewing parameter filename */
    };

struct output_rt                    /* output from RTRACE */
    {
        int     done_block;         /* finished scanline */
        int     block_length;       /* length of result in # of COLR items */
        SCANBUF result;             /* result for the scanline */
    };

struct block_info                   /* info for image reconstruction */
    {
        int     length;             /* number of pixel written to file */
        long    start_position;     /* offset from beginning of file */
    };
```

```
/*****************************************************************************/
/*
/* PROGRAM       : distribute_client.c
/*
/* AUTHOR        : Mydung Thi Tran
/*
/* DATE          : 8/13/87
/*
/* DESCRIPTION : this is part one of three parts of the work of my thesis:
/*                 - part 1: distribution client, run on the VAX,
/*                           which will be described in details ,
/*                 - part 2: ray tracing server, run on SUN workstations,
/*                           "raytracing_server.c",
/*                 - part 3: collection server, run on VAX,
/*                           in the file "collection_server.c".
/*
/*                 The goal is to apply distribute processing in ray tracing,
/*                 part 1 is the main control program, which distributes work
/*                 of generating a still image, to a group of workstations,
/*                 the number of workstations is four (4).  Work is given out
/*                 by blocks of scanlines.  Before work is distributed to the
/*                 workstation, the workstation's status will be checked by
/*                 the distribution client for availability.  When the ray
/*                 tracing server finish tracing block of scanlines, it will
/*                 contact the collection server to send result black.  Upon
/*                 finishing collecting result, collection server will update
/*                 progress report.  The distribution client, after finish
/*                 giving out all blocks will query the collection server for
/*                 report, if any block took too long to send result back,
/*                 i.e., trouble might occur on that workstations, that block
/*                 of scanlines will be redistributed to one of the available
/*                 server.
/*
/*                 The communication mechanism used is remote procedure call.
/*
/* INPUT         : should be called  with the following arguments:
/*                      + a filename for the collect, rtrace servers name list,
/*                      + a filename for the octree scene description (*.oct),
/*                      + a filename for the viewing parameter file (*.view).
/*
/* OUTPUT        : none.
/*
/*****************************************************************************/

#include "distribute.h"

bool_t          scan_list[NUMBLOCK];            /* list of blocks to distribued */
bool_t          progress[NUMBLOCK];             /* progress report */
CLIENT          *clients[MAXNUMOFHOST];         /* handles for rtrace servers */
CLIENT          *client_collect;                /* handle for collect server */
HOSTNAME        collect_server;                 /* collect server name */
HOSTNAME        wslist[MAXNUMOFHOST];           /* list of rtrace servers */
int             time_distributed[NUMBLOCK];     /* time start block of scanlines*/
struct          block_info reconstruct[NUMBLOCK]; /* image reconstruction info */
struct          timeval *timeptr;               /* argument for gettimeofday */
struct          timezone *tzoneptr;             /*                           */
struct          input_rt *rt_input;             /* argument for rtrace */

main(argc, argv)                                                    main
int argc;
char *argv[];
```

...main

```
{       /* declarations for main() */

            double atof();              /* convert ascii to double */
            FILE    *host_fp;           /* host list file pointer */
            int     i, j;               /* loop control variables */
            int     exitcode;           /* exit code */

            extern HOSTNAME wslist[];
            extern HOSTNAME collect_server;

        /* end of declarations for main() */

                        /*-----------------------------------------------*/

    /* boundary checks */
    if (argc < 4)     /* enough arguments specified? */
        {   fprintf(stderr, "usage: %s followed by: \n", argv[0]);
            fprintf(stderr, "host list, scene file, viewing para. file. \n");
            exit (FAILURE);
        }

    if (argc > 5)   /* too many arguments ? */
        {   fprintf(stderr, "%s: too many arguments specified \n", argv[0]);
            exit (FAILURE);
        }



    /* print starting time */
    fprintf(stdout, "Start distribute image at: \n");
    system("date");



    /* initialization */

    /* host names */
    if ((host_fp = fopen(argv[1], "r")) == NULL)
        {
            fprintf(stderr, "fopen: host list error in main\n");
            exit(FAILURE);
        }

    if (fscanf(host_fp, "%s", collect_server) != 1)
        {
            fprintf(stderr, "fscanf: error in main\n");
            exit(FAILURE);
        }

    for (i = 0; i < MAXNUMOFHOST; i++)
        if (fscanf(host_fp, "%s", wslist[i]) != 1)
            {
                fprintf(stderr, "fscanf: error in main\n");
                exit(FAILURE);
            }

    /* client handles */
    client_collect = NULL;
    for (i = 0; i < MAXNUMOFHOST; i++)
        clients[i] = NULL;

    /* list of scanline to be distributed */
    for (i = 0; i < NUMBLOCK; i++)
        scan_list[i] = NOT_YET;
```

```
    /* allocate memory space for pointers to structures */
    timeptr  = (struct timeval *)malloc(sizeof(struct timeval));
    tzoneptr = (struct timezone *)malloc(sizeof(struct timezone));
    rt_input = (struct input_rt *)malloc(sizeof(struct input_rt));

    if ((timeptr == NULL) || (tzoneptr == NULL) || (rt_input == NULL))
        {
            fprintf(stderr, "malloc: error in main\n");
            exit(FAILURE);
        }

    /* filename for scene description */
    rt_input->scene_file = argv[2];

    /* filename for viewing parameters */
    rt_input->view_file = argv[3];



    /*  trace rays  */
    exitcode = image();



    /*  clean up     */
    /* deallocate memory space */
    free(timeptr);
    free(tzoneptr);
    free(rt_input);

    /* done */
    if (exitcode = 0)
        {
            fprintf(stdout, "Sorry, you need to rerun the program.\n");
            exit(FAILURE);
        }

    else
        {
            fprintf(stdout, "Complete image at: \n");
            system("date");
            exit(SUCCESS);
        }

}   /* end of main() */

/*-------------------------------------------------------------------------

    ROUTINE       : image()

    DESCRIPTION   : For any block of scanlines that has not been done, an RPC
                    will be made to one of the group of SUN servers, to check
                    for availablitity.  If the server is busy, the next one in
                    the group will be checked until an idle server if found.
                    The idle server then is given the work of ray tracing the
                    block of scanlines.  Once all blocks for the entire image
                    are distributed, an RPC call to the collection server (see
                    main for details), will be issued to query for progress.
                    If complete result has not been collected, one of the two
                    actions will be taken:
                       1- if time linits for ray trace the block has not exceeded,
                          a waiting period is necessary,
                       2- if time ran out, redistribute the unfinished block(s).
                    When all results are collected, another RPC to the collection
```

```
*  server is sissued to reconstruct image.
*  If the final raster image is achieved, RPC calls are issued
*  to stop all servers before exiting sucessfully.
*
*-------------------------------------------------------------------------*/

image ()                                                        image

{   /* declarations for image() */

        bool_t    xdr_progress();          /* xdr routine for progress report */
        bool_t    redistribute;            /* redo scanline */
        bool_t    query;                   /* is this 1st query for progress */
        int       pid;                     /* child process id */
        int       i, m;                    /* loop control variable */
        int       currenttime;             /* for checking result overdue */
        int       time_elapsed;            /*              "                  */
        int       kill_rtrace;             /* call to unregister rtrace servers */
        int       kill_collect;            /* call to unregister collect server */
        int       queryerr;                /* query call for completion status */
        int       reconstruct_err;         /* call to reconstruct image */
        int       return_code = 1;         /* return code */

        extern bool_t progress[];
        extern bool_t scan_list[];
        extern HOSTNAME collect_server;
        extern HOSTNAME wslist[];
        extern int time_distributed[];
        extern struct timeval *timeptr;
        extern struct timezone *tzoneptr;

    /* end of declarations for image() */

                        /*-------------------------*/

    /* initialize the progress flag */
    query = YES_QUERY;
    redistribute = REDO;

    /* this is the distribution client */
    while (query == YES_QUERY)

       {   /* begin of while loop */

           /* distribute work to available wkstation, record starting time */
           if (redistribute == REDO)
               {
                   if (distribute () == 0)
                       {
                           fprintf(stderr, "distribute: error in image \n");
                           return_code = 0;
                       }

                   /* reset  redistribute flag */
                   redistribute = IMAGE_COMPLETE;
               }

           /* make rpc call for list of scanline done */
           if ((queryerr = callrpctcp(collect_server, COL_PROG, COL_QUERY_PROC,
                   COL_VERS, xdr_void, 0, xdr_progress, progress)) != 0)

               {
```

...image

```
                clnt_perrno(queryerr);          /* why failed */
                fprintf(stderr, "callrpctcp: query error in image\n");
                return_code = 0;
            }

        /* reset the query flag */
        query = NO_QUERY;

        /* check for overdue scanlines */
        for (m = 0; m < NUMBLOCK; m++)
            {   /* begin of for loop */
                scan_list[m] |= progress[m];
                if (scan_list[m] == NOT_YET)
                    {
                        /* get current time */
                        if (gettimeofday(timeptr, tzoneptr) != 0)
                            {
                                fprintf(stderr, "gettimeofday: error in image\n");
                                return_code = 0;
                            }
                        currenttime = (int)timeptr->tv_sec;
                        time_elapsed = currenttime - time_distributed[m];

                        /* check to see if it is overdue */
                        if (time_elapsed > TIME_LIMIT)
                            {
                                /* set progress flags */
                                redistribute = REDO;
                                query = YES_QUERY;

                            }   /* end of if TIME_LIMIT */
                        else
                            {   /* begin of else */
                                /* wait awhile */
                                sleep(2*BLOCKSIZE);

                                /* requery for progress */
                                if ((queryerr = callrpctcp(collect_server, COL_PROG,
                                    COL_QUERY_PROC, COL_VERS, xdr_void, 0,
                                    xdr_progress, progress)) != 0)
                                    {
                                        clnt_perrno(queryerr);      /* why fail */
                                        fprintf(stderr, "callrpctcp: query error in image\n");
                                        return_code = 0;
                                    }

                                /* recheck */
                                m--;
                            }   /* end of else */
                    }   /* end of if NOT_YET */

            }   /* end of for loop */

    }   /* end of while loop */

/* ask child server to reconstruct image */
if ((reconstruct_err = callrpctcp(collect_server, COL_PROG,
        COL_RECON_PROC, COL_VERS, xdr_void, 0, xdr_void, 0)) != 0)
    {
        clnt_perrno(reconstruct_err);   /*why failed */
        fprintf(stderr, "callrpctcp: reconstruct error in image\n");
        return_code = 0;
    }
```

...image

```
    /* send signal to kill servers */
    for (i = 0; i < MAXNUMOFHOST; i++)
        if ((kill_rtrace = callrpctcp(wslist[i], RAY_PROG, RAY_KILL_PROC,
                                      RAY_VERS, xdr_void, 0, xdr_void, 0)) != 0)
            {
                clnt_perrno(kill_rtrace);
                fprintf(stderr, "callrpctcp: unregister rtrace error in image\n");
                return_code = 0;
            }

    if ((kill_collect = callrpctcp(collect_server, COL_PROG, COL_KILL_PROC,
                        COL_VERS, xdr_void, 0, xdr_void, 0)) != 0)
        {

            clnt_perrno(kill_collect);
            fprintf(stderr, "callrpctcp: unregister collect error in image\n");
            return_code = 0;
        }

    /* done, exit */
    return return_code;


}   /* end of image() */
```

```
/*-----------------------------------------------------------------------------

    ROUTINE       : distribute()

    DESCRIPTION   : This routine is executed by the distribution client.
                    It checks the status of the remote machine for availability
                    by comparing the CPU idle time in the last 5 seconds with the
                    threshold percentage. The work of tracing a block is then
                    given to the remote machine if the status return indicates
                    availability. The block number and starting time are
                    maintained for synchronization.

-----------------------------------------------------------------------------*/
```

distribute() *distribute*

```
{   /* declarations for distribute() */

        int     j;                  /* loop control variable */

        /* RPC call for availability status check. */
        bool_t  avail;              /* available flag */
        int     statuserr;          /* status call */
        int     i;                  /* index into remote machine list */
        extern  HOSTNAME wslist[];
        extern  CLIENT *clients[];


        /* SUN RPC RTRACE variables */
        bool_t  xdr_rtinput();      /* xdr for arguments to RTRACE */
        int     rtraceerr;          /* return value of RPC RTRACE  */
        extern  bool_t scan_list[];
        extern  int time_distributed[];
        extern  struct input_rt *rt_input;

    /* end of declarations for distribute() */
```

```
/*-----------------------*/

/* distribute work to available remote machines */

for (j = 0; j < NUMBLOCK; )

{   /* begin of for loop */

    /* check for scanline to be distributed */
    if (scan_list[j] == NOT_YET)

    {   /* begin of if scanline NOT_YET distributed */

        /* check for boundary for hostlist index, reset if needed */
        if (i >= MAXNUMOFHOST)
            i = 0;

        /* make RPC call to check status of remote machine */
        if ((statuserr = callrpctcp(wslist[i], RAY_PROG, RAY_STAT_PROC,
                RAY_VERS, xdr_void, 0, xdr_bool, &avail)) != 0)
            {
                clients[i] = NULL;
                clnt_perrno(statuserr);        /* why failed */
                fprintf(stderr, "callrpctcp: status check error in distribute\n");
                goto next_server;
            }

        /* check status info */
        if (avail == BUSY)
            {
                fprintf (stdout," "%s is not available \n", wslist[i]);
                goto next_server;
            }
        else
            {   /* begin of else for server available */

                /* record time distributed */
                if (gettimeofday(timeptr, tzoneptr) != 0)
                    {
                        fprintf(stderr, "gettimeofday: error in distribute\n");
                        goto next_server;
                    }
                time_distributed[j] = (int)timeptr->tv_sec;
                rt_input->curr_block = j;

                /* make RPC call to trace rays for the current scanline */
                if ((rtraceerr = callrpctcp(wslist[i], RAY_PROG,RAY_RTRACE_PROC,
                        RAY_VERS, xdr_rtinput, rt_input, xdr_void, 0)) != 0)
                    {
                        clnt_perrno(rtraceerr);        /* why failed */
                        fprintf(stderr, "callrpctcp: rtrace error in distribute\n");
                        goto next_server;
                    }

                /* set up to do next block of scanlines */
                j++;

            }   /* end of else for server available */

        /* set up to go to next remote machine */
        next_server: i++;
```

```
        }   /* end of scanline is NOT_YET distributed */

        else
            j++;        /* go to next block of scanlines */

    }   /* end of for loop */

    return(1);

}   /* end of distribute() */

/*-------------------------------------------------------------------------------

    ROUTINE      : callrpctcp()

    DESCRIPTION : this routine is the intermediate RPC layer to the SUN
                  workstations to do the followings:
                      - to check workstations' status & determine availability,
                      - to distribute work, i.e. tracing one scanline.
                  The body of this routine contains the lowest level RPCs.

-------------------------------------------------------------------------------*/
```

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out) *callrpctcp*

```
char        *host;
char        *in, *out;
int         prognum, procnum, versnum;
xdrproc_t inproc, outproc;

{   /* declarations for callrpctcp() */

        enum    clnt_stat clnt_stat;                /* return value of call */
        int     i;                                  /* loop control variable */
        int     socket = RPC_ANYSOCK;               /* socket chosen by system */
        struct  hostent *hp;                        /* host entry */
        struct  sockaddr_in server_addr;            /* info on server address */
        struct  timeval total_timeout;              /* time out provision */
        extern  HOSTNAME wslist[];
        extern  HOSTNAME collect_server;
        extern  CLIENT *clients[], *client_collect;

    /* end of declarations for callrpctcp() */

                    /*-------------------------------*/

    /* test to appropriate client handle */
    /* is this a call to the collection server ? */
    if (strcmp(host, collect_server) == 0)
        {
            if (client_collect == NULL)
                {
                    /* get host entry */
                    if ((hp = gethostbyname(host)) == NULL)
                        {
                            fprintf(stderr, "gethostbyname: error in callrpctcp\n");
                            return(-1);
                        }
                    bcopy(hp->h_addr,(caddr_t)&server_addr.sin_addr,hp->h_length);
                    server_addr.sin_family = AF_INET;
                    server_addr.sin_port   = htons(0);
```

```
                /* create client socket, memory stream, and client handle */
                if ((client_collect = clnttcp_create(&server_addr, prognum,
                                versnum, &socket, BUFSIZ, BUFSIZ)) == NULL)

                    {
                        clnt_pcreateerror("clnttcp_create");
                        return(-1);
                    }

            }   /* end if client handle has not been created */
        /* make the call */
        total_timeout.tv_sec = 10*BLOCKSIZE;
        total_timeout.tv_usec = 0;
        clnt_stat = clnt_call(client_collect, procnum, inproc, in,
                                outproc, out, total_timeout);
        if (clnt_stat != RPC_SUCCESS)
            {
                clnt_perror(client_collect, "rpc");
                return(-1);
            }

        /* destroy client handle if needed */
        if (procnum == COL_KILL_PROC)
            clnt_destroy(client_collect);

        return((int)clnt_stat);

    }   /* end if this is the call to the collection server */

/* or if this is the call to ray tracing servers on the SUNs */
for (i = 0; i < MAXNUMOFHOST; i++ )
    {   /* begin of for loop */
        /* search for the appropriate SUN server */
        if (strcmp(host, wslist[i]) == 0)
            {
                /* check to see if client handle has been created ? */
                if (clients[i] == NULL)
                    {   /* not yet, create handle */
                        if ((hp = gethostbyname(host)) == NULL)
                            {
                                fprintf(stderr,"gethostbyname: error in calrpctcp\n");
                                return(-1);
                            }
                        bcopy(hp->h_addr,(caddr_t)&server_addr.sin_addr,
                                        hp->h_length);
                        server_addr.sin_family = AF_INET;
                        server_addr.sin_port   = htons(0);

                        /* create client socket, memory stream and client handle */
                        if ((clients[i] = clnttcp_create(&server_addr, prognum,
                            versnum, &socket, BUFSIZ, BUFSIZ)) == NULL)
                            {
                                perror("clnttcp_create");
                                return(-1);
                            }

                    }   /* end if handle has not been created */
                /* make the call */
                total_timeout.tv_sec = TIME_LIMIT;
                total_timeout.tv_usec = 0;
                clnt_stat = clnt_call(clients[i], procnum, inproc, in,
                                        outproc, out, total_timeout);
                if (clnt_stat != RPC_SUCCESS)
                    {
                        perror(clients[i], "rpc");
```

```
                    return(-1);
                }

            /* destroy client handle, if needed */
            if (procnum == RAY_KILL_PROC)
                clnt_destroy(clients[i]);

            return((int)clnt_stat);
        }  /* end if strcmp */

    }  /* end of for loop */

}  /* end of callrpctcp() */

/*-------------------------------------------------------------------------

    ROUTINE      : xdr_rtinput()

    DESCRIPTION : this is the eXternal Data Representation routine for the
                  arguments to SUN RPC RTRACE.
                  The input argument to RTRACE consists of:
                      - an integer for the number of block of scanlines,
                      - a filename for the scene description file,
                      - a filename for the viewing parameters.
                  For part one, executes by the distribution client, this XDR
                  will encode the argument before sending it over to the server.

-------------------------------------------------------------------------*/

bool_t
xdr_rtinput(xdrs, rt_input)                                    xdr_rtinput

XDR     *xdrs;
struct input_rt *rt_input;

{   /* declaration for xdr_rtinput() */

        int k;                          /* loop control variable */

    /* end of declaration for xdr_rtinput() */

                    /*-------------------------------*/


    if (!xdr_int(xdrs, &rt_input->curr_block))
        return(FALSE);

    if (!xdr_string(xdrs, &rt_input->scene_file, 20))
        return(FALSE);

    if (!xdr_string(xdrs, &rt_input->view_file, 20))
        return(FALSE);

    return(TRUE);

}   /* end of xdr_rtinput() */

/*-------------------------------------------------------------------------

  ROUTINE      : xdr_progress()

  DESCRIPTION : this is the eXternal Data Representation for the array of
                boolean type of NUMBLOCK elements, which contains information
                on whether or not the scanline result has been recorded in
                the rasterfile.  This routine is called to decode the
```

progress report structure received from the collection server.

```
/*-------------------------------------------------------------------------*/

bool_t
xdr_progress(xdrs, progress)

XDR        *xdrs;
bool_t   progress[NUMBLOCK];

{   /* declaration for xdr_progress() */

        int k;                  /* loop control variable */

    /* end of declaration for xdr_progress() */

                    /*-----------------------------------*/

    for (k = 0; k < NUMBLOCK; k++)
        {
            if (!xdr_bool(xdrs, &progress[k]))
                return(FALSE);
        }

    return(TRUE);

}    /* end of xdr_progress() */

/*-------------------------------------------------------------------------*/
```

xdr_progress

```
/*****************************************************************************/
/*
/* FILENAME      : raytracing_server.c
/*
/* AUTHOR        : Mydung Thi Tran
/*
/* DATE          : 8/10/87
/*
/* DESCRIPTION : this is part two of three parts of my thesis work.
/*                 Details of part one can be found in the file
/*                     "distribute_client.c"
/*                 Details of part three can be found in the file
/*                     "raytracing_server.c
/*                 Part two, executed by the SUN workstations, these servers
/*                     carry out the ray tracing work and sending result of a
/*                     block of scanlines back to its client, the distribution
/*                     server of part one.
/*
/* INPUT         : none
/*
/* OUTPUT        : none
/*
/*****************************************************************************/

#include "distribute.h"


bool_t    statistics();                 /* status check routine forward declaration*/
bool_t    xdr_rtinput();                /* XDR routine for argument to RTRACE */
bool_t    xdr_rtoutput();               /*                "                   */
char      collect_server[20];           /* host name for call back RPC */
char      inputfile[20];                /* rtrace input filename */
char      resultfile[20];               /* rtrace result filename */
char      viewfile[20];                 /* viewing parameter file */
char      octree[20];                   /* octree scene description file */
char      rt_com[512];                  /* ray tracing command */
int       dist_rt();                    /* dispatch routine forward declaration */
int       rtrace_work();                /* ray tracing routine forward declaration */
int       blocksize;                    /* number of scanlines per block */
long      header_length = -1;           /* result file header's length */
struct    input_rt *rt_input;           /* input argument to RTRACE */
struct    output_rt *rt_output;         /* output argument to RTRACE */
CLIENT    *client_callback = NULL;              /* client handle for sending result RPC */


main()                                                              main

{       /* declarations of main() */

            register SVCXPRT *transp;       /* service transport handle */

        /* end of declarations of main() */

                    /*******************************/

    /* allocate memory for arguments to RTRACE */
    rt_input = (struct input_rt *)malloc(sizeof(struct input_rt));
    rt_output = (struct output_rt *)malloc(sizeof(struct output_rt));

    /* initialize host name for call back RPC and the blocksize */
    strcpy(collect_server, "lb1-csam");
    blocksize = BLOCKSIZE;
```

...main

```
    /* create server socket, memory stream and service handle */
    if ((transp = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) == NULL)
        {
            fprintf(stderr, "svctcp_create: error in main \n");
            exit(1);
        }

    /* erase any trace, make sure this is the update version */
    pmap_unset(RAY_PROG, RAY_VERS);

    /* register service */
    if (!svc_register(transp, RAY_PROG, RAY_VERS, dist_rt, IPPROTO_TCP))
        {
             fprintf(stderr, "svc_register: error in main \n");
             exit(1);
        }

    /* never return */
    svc_run ();

    /* should not reach this point */
    fprintf(stderr, "svc_run returned: error in main \n");
    exit(-1);

}   /* end of main() */
```

```
/*-----------------------------------------------------------------------------

    ROUTINE       : dist_rt()

    DESCRIPTION. : this is the actual dispatch routine for distributed processing
                   in ray tracing. When this routine is called, one of following
                   will be taken care:
                       1- RAY_STAT: check for cpu availability,
                       2- RAY-RTRACE: decode input argument for ray tracing
                                routine, carry out the work, call the collection
                                server to send block of scanlines result back,
                       3- RAY_KILL: clean up, i.e. free memory, unmap service,
                       4- default: print "nomatch" error message.

-----------------------------------------------------------------------------*/
```

dist_rt(rqstp, transp) *dist_rt*

```
register struct svc_req *rqstp;          /* contains procedure number */
register SVCXPRT *transp;                /* service transport handle */

{       /* declarations for dist_rt() */

        bool_t    avail;                 /* CPU status check */
        char      ws_name[20];           /* name of host */
        int       gethost_err;
        int       maxlength = 20;        /* max # of elem in name array */
        int       callbackerr;           /* sending scanline result call */
        int       rtpid;                 /* child process id */

        extern char collect_server[];
        extern struct input_rt  *rt_input;
        extern struct output_rt *rt_output;
        extern bool_t statistics();
        extern bool_t xdr_rtinput();
        extern bool_t xdr_rtoutput();
```

```
        /* end of declarations for dist_rt() */

                        /*******************************/

switch ((int) rqstp->rq_proc)
{   /* begin of switch */

    case RAY_STAT_PROC:      /* check for CPU availability */

        {   /* begin of case RAY_STAT_PROC */

            /* get workstation name */
            if ((gethost_err = gethostname(ws_name, maxlength)) != 0 )
                {
                    fprintf(stderr, "gethostname: error in dist_rt \n");
                    return(0);
                }

            /* actual work, i.e. getting workstation status */
            avail = statistics(ws_name);

            /* send result back to client */
            if (!svc_sendreply(transp, xdr_bool, &avail))
                {
                    fprintf(stderr, "svc_sendreply: error in dist_rt \n");
                    return (0);
                }
            return(1);

        }   /* end of case RAY_STAT_PROC */

    case RAY_RTRACE_PROC:   /* trace rays for one scanline */

        {   /* begin of case RAY_RTRACE_PROC */

            /* get arguments */
            if (!svc_getargs(transp, xdr_rtinput, rt_input))
                {
                    fprintf(stderr, "svc_getargs: error in dist_rt \n");
                    svcerr_decode(transp);
                    return(0);
                }

            /* send void return back to control program */
            if (!svc_sendreply(transp, xdr_void, 0))
                {
                    fprintf(stderr, "svc_sendreply: error in dist_rt\n");
                    return(0);
                }

            /* carry out actual work, i.e. tracing rays */
            /* generate input and trace rays for scanline */
            if (rtrace_work(rt_input, rt_output)!= 0)
                {
                    fprintf(stderr, "rtrace_work: error in dist_rt\n");
                    return(0);
                }

            /* RPC call to send back results to collection server on VAX */
            if ((callbackerr = callrpctcp(collect_server, COL_PROG, COL_COLCT_PROC,
                        COL_VERS, xdr_rtoutput, rt_output, xdr_void, 0)) != 0)
                {
                    clnt_perror(callbackerr);
                    fprintf(stderr, "can't make rpc call send back result \n");
                    return(0);
```

```
            }

        return(1);

    }   /* end of case RAY_RTRACE_PROC */

case RAY_KILL_PROC:     /* kill service */

    {   /* begin of case RAY_KILL_PROC */

        if (!svc_sendreply(transp,xdr_void, 0))
            {
                fprintf(stderr, "svc_sendreply: error in dist_rt \n");
                return(0);
            }

        /* deallocate memory space used for pointers to structures */
        free(rt_input);
        free(rt_output);

        /* unmap service */
        svc_unregister(RAY_PROG, RAY_VERS);

        /* done, exit */
        exit(1);

    }   /* end of case RAY_KILL_PROC */

default:                    /* no procedure number matched */

    {   /* begin of case default */

        svcerr_noproc(transp);
        return(0);

    }   /* end of case default */

}   /* end of switch */

}   /* end of dist_rt() */

/*-----------------------------------------------------------------------------

    ROUTINE     : statistics()

    DESCRIPTION : remote machine status will be checked by calling the highest
                  level of RPC "rstat". "rstat" will be called twice, since all
                  values return by it is accumulative, the difference will be
                  calculated for CPU idle time and percentage of this idle time
                  will be compared to the THRESHOLD value. This function will
                  return BUSY if the percent of idletime is less than THRESHOLD,
                  IDLE, otherwise.

-----------------------------------------------------------------------------*/

bool_t
statistics(wsname)                                          statistics

char    **wsname;

{       /* declaration for statistics() */
```

```
        long    cpu1_idletime;      /* idle time in units of 20 milliseconds */
        long    cpu2_idletime;      /* idle time obtained by second call */
        long    cpu_idletime;       /* difference between two calls */

        long    current1_sec;       /* time in seconds at first call */
        long    current2_sec;       /* time at second call */
        long    sec_bet2call;       /* time elapsed between 2 calls */

        int     percent_idle;       /* percent of idletime */
        struct  statstime statptr;              /* return result */

      /* end of declarations for statistics() */

                /*-------------------------*/


    /* first call */
    rstat(wsname, &statptr);

    /* set up variables after first call */
    cpu1_idletime = statptr.cp_time[CP_IDLE];
    current1_sec  = statptr.curtime.tv_sec;

    /* wait awhile */
    sleep(5);

    /* second call */
    rstat(wsname, &statptr);

    /* set up variables after second call */
    cpu2_idletime = statptr.cp_time[CP_IDLE];
    current2_sec  = statptr.curtime.tv_sec;

    /* get the difference between the 2 calls */
    cpu_idletime = cpu2_idletime - cpu1_idletime;
    sec_bet2call = current2_sec  - current1_sec;

    /* convert to same units and calculate percentage */
    percent_idle = (int)((cpu_idletime * 2) / sec_bet2call);

    /* determine availibility */
    if (percent_idle > THRESHOLD_PRCT)
        {
            fprintf(stderr, "%s is available \n", wsname);
            return(IDLE);
        }
    else
        {
            fprintf(stderr, "%s is busy \n", wsname);
            return(BUSY);
        }

}   /* end of statistics() */

/*-----------------------------------------------------------------------------

    ROUTINE     : rtrace_work()

    DESCRIPTION : read viewing parameters from a file, generates input
                  (6 floating point numbers for viewpoint and direction
                  of rays for "rtrace", the actual ray trace routine.

-----------------------------------------------------------------------------*/
```

```
struct input_rt *in;
struct output_rt *out;

{               /* declaration for rtrace_work() */

        int     i;                      /* loop control variable */
        int     numread;                /* # bytes read from result */
        int     rtrace_pid;             /* process id for rtrace */
        int     wait_pid;               /* process id for child  */
        char    sbuf[128];              /* buffer for file header */
        FILE    *view_fp;               /* viewing parameter file */
        FILE    *input_fp;              /* input to rtrace */
        FILE    *result_fp;             /* temp result for scanline */

        extern  long header_length;
        extern  char octree[];
        extern  char viewfile[];
        extern  char inpufile[];
        extern  char resultfile[];
        extern  char rt_com[];

            /* end of declaration for rtrace_work() */

                        /*----------------------*/

    /* print scanline number for user's interest */
    fprintf(stdout, "start block number: %d \n", in->curr_block);

    /* set up the command for tracing rays */
    strcpy(octree, in->scene_file);
    setcommand();

    /* generate input for rtrace */
    strcpy(viewfile, in->view_file);
    if (gen_input() != 0)
        {
            fprintf(stderr, "gen_input: error in rtrace_work\n");
            return(-1);
        }

    /* spawn new process to retain a copy of executable image */
    rtrace_pid = fork();

    if (rtrace_pid == 0)            /* child: trace rays */
        {
            execl("/bin/sh", "sh", "-c", rt_com, 0);
            _exit(0);
        }

    if (rtrace_pid > 0)             /* parent got a copy to cont */
        {
            wait_pid = wait(0);     /* wait for child competion */
            if (wait_pid > 0)
                kill(wait_pid,SIGQUIT);

            /* store return result */
            out->done_block = in->curr_block;
            result_fp = fopen(resultfile, "r");

            /* discard header, store header length if needed */
            if (header_length = -1)
                {
                    while(fgets(sbuf, sizeof(sbuf), result_fp) != NULL &&
                            sbuf[0] != '\n')
```

```
                ;
            if (fgets(sbuf, sizeof(sbuf), result_fp) == NULL ||
                sscanf(sbuf, "-Y %d +X %d\n", &yres, &xres) != 2)
                {
                    fprintf(stderr, "fgets: discard header error \n");
                    return(-1);
                }
            /* get header length to store */
            header_length = ftell(result_fp);
        }
    else
        {
            /* get just beyond the header */
            fseek(result_fp, header_length, 0);
        }

    /* get result */
    numread = fread(out->result, sizeof(COLR), BLOCK, result_fp);
    out->block_length = numread;
    fclose(resultfile);

    /* finish actual work */
    return(0);
  }

} /* end of rtrace_work() */

/*-----------------------------------------------------------------------------

    ROUTINE     : setcommand()

    DESCRIPTION : the command for ray tracing a block of scanlines is set up
                  by calling this routine.  This command consists of:
                    - the name of the ray tracing routine, "rtrace",
                    - the input and output format option, "-ffc",
                      the first f stands for format, the second f stands for
                      float (input: floating point numbers), and the c stands
                      for COLR (output: special format, RED, BLUE, GREEN,
                      EXPONENT)
                    - the x and y resolution option, "-x XRES" & "-y YRES",
                      both of these numbers should be included so that the
                      result buffer will be flushed properly, and the header
                      of the output file wil be generated correctly,
                    - the octree scene dexcription filename,
                    - the input filename, this file contains the ray
                      origin and ray directions,
                    - the output filename, so that result could be copied
                      into a buffer to send back to the collection server.

-----------------------------------------------------------------------------*/

setcommand()                                                    setcommand

{   /* declarations for setcommand() */

        char    hostname[20];       /* name of ray trace server */
        FILE    *input_fp;          /* input to ray trace file pointer */

        extern  char octree[];
        extern  char viewfile[];
        extern  char inputfile[];
        extern  char resultfile[];
        extern  char rt_com[];
        extern  int  blocksize;
```

...setcommand

```
    /* end of declarations for setcommand() */

                    /*-------------------------------------*/

    /* initialise filenames */
    strcpy(inputfile, "for");
    strcpy(resultfile, "from");

    /* initialize command */
    switch (blocksize) {
        case  5:
                strcpy(rt_com, "rtrace -ffc -x 512 -y 5 ");
                break;
        case 10:
                strcpy(rt_com, "rtrace -ffc -x 512 -y 10 ");
                break;
        case 20:
                strcpy(rt_com, "rtrace -ffc -x 512 -y 20 ");
                break;

    }   /* end of switch statement */

    /* add octree scene filename */
    strcat(rt_com, octree);

    /* get hostname to distinguish input and result files */
    gethostname(hostname, 20);
    strcat(inputfile, hostname);
    strcat(resultfile, hostname);

    /* set up the rest of the command */
    strcat(rt_com, " < ");
    strcat(rt_com, inputfile);
    strcat(rt_com, " > ");
    strcat(rt_com, resultfile);

    /* finish */
    return(0);

}   /* end of setcommand() */

/*-------------------------------------------------------------------------------

    ROUTINE     : gen_input()

    DESCRIPTION : input to "rtrace", the ray tracing routine, is generated
                  and written into a file in this routine.  "rtrace" 's input
                  consists of 512 sets of data for 512 pixels in a scanline.
                  Each set consists of 3 floating points for the viewpoint,
                  and 3 floating points for the ray direction.

-------------------------------------------------------------------------------*/
```

gen_input() *gen_input*

```
{   /* declarations for gen_input() */

        double    anghor, angvert;  /* horizontal, vertical view angles */
        double    h, v;             /* horizontal, vertical offsets */
        FILE      *view_fp;         /* viewing parameter file pointer */
        FILE      *input_fp;        /* rtrace input file pointer */
```

```
        float       view[3];            /* view point coordinate */
        float       direction[3];       /* ray direction */
        FVECT       vp, vdir, vup;      /* view point, direction, up vectors */
        FVECT       vhinc, vvinc;       /* hor, vert increments on view plane */
        int         i;                  /* loop control variable */
        int         x, y;               /* pixel position */
        int         y0, y1;             /* initial, final scanline # of block */
        int         point, up, dir;     /* # floating point numbers read */
        int         angle;              /*          "                    */
        int         bview, bdir;        /* # floating point numbers written */

        extern      char    inputfile[];
        extern      char    viewfile[];
        extern      struct input_rt *rt_input;

    /* end of declarations for gen_input() */

                        /*-----------------------------------------------*/

    /* open files */
    view_fp = fopen(viewfile, "r");
    input_fp = fopen(inputfile, "w");
    if ((view_fp == NULL) || (input_fp == NULL))
        {
            fprintf(stderr, "fopen: error in gen_input\n");
            return(-1);
        }

    /* get viewing parameter from file */
    point = fscanf(view_fp, "%lf%lf%lf", &vp[0], &vp[1], &vp[2]);
    dir   = fscanf(view_fp, "%lf%lf%lf", &vdir[0], &vdir[1], &vdir[2]);
    up    = fscanf(view_fp, "%lf%lf%lf", &vup[0], &vup[1], &vup[2]);
    angle = fscanf(view_fp, "%lf%lf", &anghor, &angvert);

    if ((point != 3) || (dir != 3) || (up != 3) || (angle != 2))
        {
            fprintf(stderr, "fscanf: viewpara file error in gen_input\n");
            return(-1);
        }

    /* done with viewpara file */
    fclose(view_fp);

    /* set up view point */
    view[0] = (float)vp[0];
    view[1] = (float)vp[1];
    view[2] = (float)vp[2];

    /* calculate viewing direction increments */
    setview(view, vdir, vup, anghor, angvert, vhinc, vvinc);

    /* set up values for first and last scanlines of block */
    y0 = rt_input->curr_block * BLOCKSIZE;
    y1 = y0 + BLOCKSIZE;

    /* start genarating input */
    /* loop until the last scanline of block */
    for (y = y0; y < y1; y++)

        /* loop until last pixel of scanline */
        for (x = 0; x < XRES; x++)
```

...gen_input

```
            {   /* begin of for x loop */
                /* set up ray direction */
                h = x - XRES / 2.0;     /* horizontal offset */
                v = y - YRES / 2.0;     /* vertical offset */

                for (i = 0; i < 3; i++)
                    direction[i] = (float)(vdir[i] + h*vhinc[i] + v*vvinc[i]);

                /* write viewpoint and direction to rtrace's inputfile */
                bview = fwrite(view, sizeof(float), 3, input_fp);
                bdir  = fwrite(direction, sizeof(float), 3, input_fp);
                if ((bview != 3) || (bdir != 3))
                    {
                        fprintf(stderr, "fwrite: error in gen_input\n");
                        return(-1);
                    }

            }   /* end of for x loop */

    /* finish generate input */
    fclose(input_fp);
    return(0);

}   /* end of gen_input() */

/*-------------------------------------------------------------------------------

    ROUTINE     : setview()

    DESCRIPTION : calculate the vertical and horizontal increment
                  for input to rtrace command. This routine is extracted
                  from GREG's routine in the file called "image.c"
                  found in the directory "~/ray/rt".

-------------------------------------------------------------------------------*/
```

setview(vp, vdir, vup, anghor, angvert, vhinc, vvinc) *setview*

```
FVECT       vp, vdir, vup, vhinc, vvinc;
double      anghor, angvert;

{           /* begin of declarations of setview() */

            double    tan(), normalize();
            double    dt;

              /* end of declaration of setview() */

                    /*-----------------------------------*/

    /* normalize view direction vector */
    if (normalize(vdir) == 0.0)
        {
            fprintf(stderr, "zero view direction\n");
            return(-1);
        }

    /* compute horizontal direction */
    fcross(vhinc, vdir, vup);

    /* normalize view up vector */
    if (normalize(vhinc) == 0.0)
        {
            fprintf(stderr, "illegal view up vector\n");
            return(-1);
```

...setview

```
        }

    /* compute vertical direction */
    fcross(vvinc, vhinc, vdir);

    dt = 2.0 * tan(anghor*(PI/180.0/2.0));

    if (dt <= FTINY || dt >= FHUGE)
        {
            fprintf(stderr, "illegal horizontal view angle\n");
            return(-1);
        }

    dt /= XRES;
    vhinc[0] *= dt;
    vhinc[1] *= dt;
    vhinc[2] *= dt;

    dt = 2.0 * tan(angvert*(PI/180.0/2.0));

    if (dt <= FTINY || dt >= FHUGE)
        {
            fprintf(stderr, "illegal vertical view angle \n");
            return(-1);
        }

    dt /= YRES;
    vvinc[0] *= dt;
    vvinc[1] *= dt;
    vvinc[2] *= dt;

}   /* end of setview() */

/*-----------------------------------------------------------------------------

    ROUTINE     : normalize()

    DESCRIPTION : normalize a vector. This is an exact copy of the
                  routine "normalize" found in ~/ray/rt/fvect.c

-----------------------------------------------------------------------------*/

double
normalize(v)                                                    normalize
register FVECT(v);

{       /* declaration for normalize() */

          double    len;
          double    sqrt();

        /* end of declaration for normalize() */

    len = DOT(v,v);
    if (len <= FTINY*FTINY)
        return(0.0);

    if (len >= (1.0-FTINY)*(1.0-FTINY) &&
              len <= (1.0+FTINY)*(1.0+FTINY))
        return(1.0);

    len = sqrt(len);
    v[0] /= len;
    v[1] /= len;
    v[2] /= len;
```

...normalize

```
    return(len);

}   /* end of normalize() */

/*-------------------------------------------------------------------------

    ROUTINE.        : fcross()

    DESCRIPTION : calculate the cross product of two vectors v1 X v2

-------------------------------------------------------------------------*/
```

fcross

```
fcross(vres, v1, v2)
register FVECT vres, v1, v2;

{   /* begin of fcross() */

    vres[0] = v1[1]*v2[2] - v1[2]*v2[1];
    vres[1] = v1[2]*v2[0] - v1[0]*v2[2];
    vres[2] = v1[0]*v2[1] - v1[1]*v2[0];

}   /* end of fcross() */

/*-------------------------------------------------------------------------

    ROUTINE         : callrpctcp()

    DESCRIPTION : dispatch routine for sending result back to the child
                  process on the VAX side.

-------------------------------------------------------------------------*/
```

callrpctcp

```
callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)

char        *host;
char        *in, *out;
int         prognum, versnum, procnum;
xdrproc_t inproc, outproc;

{           /* declarations for callrpctcp() */

        enum    clnt_stat clnt_stat;
        int     socket = RPC_ANYSOCK;
        struct  sockaddr_in server_addr;
        struct  hostent *hp;                    /* host entry */
        struct  timeval total_timeout;          /* timeout provisions */

        extern  CLIENT *client_callback;

           /* end of declarations for callrpctcp() */

                        /*-------------------------------------------*/

    /* has socket and client handle been created ? */
    if (client_callback == NULL)
        {
            /* get host entry */
            if ((hp = gethostbyname(host)) == NULL)
                {
                    fprintf(stderr, "can't get address for host \n");
                    return(-1);
                }

            bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
            server_addr.sin_family = AF_INET;
```

...callrpctcp

```
            server_addr.sin_port    = htons(0);

            /* create client socket, memory stream, and client handle */
            if ((client_callback = clnttcp_create(&server_addr,
                  prognum, versnum, &socket, BUFSIZ, BUFSIZ)) == NULL)
                {
                    clnt_pcreateerror("clnttcp_create");
                    return(-1);
                }

        }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;

    /* call dispatch routine */
    clnt_stat = clnt_call(client_callback, procnum, inproc, in,
                          outproc, out, total_timeout);
    if (clnt_stat != RPC_SUCCESS)
        {
            clnt_perror(client_callback, "rpc");
            return(-1);
        }

    return((int)clnt_stat);

}   /* end of callrpctcp() */

/*-------------------------------------------------------------------------------

    ROUTINE       : xdr_rtinput()

    DESCRIPTION : eXternal Data Representation for input to "rtrace".
                  This routine is called to decode the input argument.
                  The input consists of the followings:
                      - an integer for the block number,
                      - a filename for the scene decription file,
                      - a filename for the viewing parameter file.

-------------------------------------------------------------------------------*/

bool_t
xdr_rtinput(xdrs, rt_input)                                    xdr_rtinput

XDR     *xdrs;
struct input_rt *rt_input;

{           /* declaration for xdr_rtinput() */

           int        k;          /* loop control variable */

            /* end of declaration for xdr_rtinput() */

                    /*-------------------------*/

    if (!xdr_int(xdrs, &rt_input->curr_block))
        return(FALSE);

    if (!xdr_string(xdrs, &rt_input->scene_file, 20))
        return(FALSE);

    if (!xdr_string(xdrs, &rt_input->view_file, 20))
        return(FALSE);

    return(TRUE);
```

```
} /* end of xdr_rtinput() */

/*-----------------------------------------------------------------------------

    ROUTINE       : xdr_rtoutput()

    DESCRIPTION   : eXternal Data Representation routine for the result of a
                    scanline sent from Sun workstations back to VAX side.
                    This routine is called to encode the result before
                    sending it to the collection server.
                    The result consists of the followings:
                        - an integer for the number of the block done,
                        - an integer for the number of "COLR" items store as
                              the result, this number is not necessary equal to
                              the number of pixels in the block (512 * ABLOCKSIZE),
                              due to run-length encoding,
                        - an array of pixel values. Each value is stored as
                              four unsigned chars: RED, BLUE, GREEN, EXPONENT.

-----------------------------------------------------------------------------*/

bool_t
xdr_rtoutput(xdrs, rt_out)                                    xdr_rtoutput

XDR     *xdrs;
struct output_rt *rt_out;

{
    /* declaration for xdr_rtoutput() */

        int         maxsize;

    /* end of declaration for xdr_rtoutput() */

    maxsize = XRES * 4 * BLOCKSIZE;

    if (!xdr_int(xdrs, &rt_out->done_block))
        return (FALSE);

    if (!xdr_int(xdrs, &rt_out->block_length))
        return (FALSE);

    if (!xdr_opaque(xdrs, rt_out->result, maxsize))
        return (FALSE);

    return (TRUE);

} /* end of xdr_rtoutput() */

/*-----------------------------------------------------------------------------*/
```

```
/*****************************************************************************/
/*
/* PROGRAM       : collect_server.c
/*
/* AUTHOR        : Mydung Thi Tran
/*
/* DATE          : 7/27/87
/*
/* DESCRIPTION : this is part three of 3 parts of my thesis work:
/*                    - part 1: the distribution client can be found in the
/*                              file "distribute_client.c".
/*                    - part 2: the ray tracing server, can be found in the
/*                              file "raytrace_server.c".
/*                  This routine is executed the result collection server.
/*                  It listens to call back from the ray tracing servers for
/*                  result.  The results, values for pixels of a scanline,
/*                  are written into the output file, specified by the external
/*                  file pointer "output_fp", these scanlines are written in
/*                  random order at first and when all scanlines are written,
/*                  image will then be reconstructed using information stored
/*                  in the data structure "result", which contained 3 fields:
/*                  scanline number, location in raster output file, length of
/*                  scanline (necessary due to run-length encoding).
/*                  For synchronization purposes, distribution client (part 1)
/*                  will invoke a remote procedure call to this server, to
/*                  request for list of scanline completed. Checking for image
/*                  completion is done in the distribution client, any overdue
/*                  scanline will be redistributed.
/*
/* INPUT         : should be called with:
/*                    - name for random outputfile,
/*                    - name for raster outputfile.
/*
/* OUTPUT        :  - random outputfile,
/*                    - raster outputfile.
/*
/*****************************************************************************/

#include "distribute.h"


bool_t     xdr_rtoutput();              /* xdr routine for output argument to rtrace */
bool_t     progress[NUMBLOCK];          /* progress report structure */
char       outputfile[20];              /* random output filename */
char       rasterfile[20];              /* raster output filename */
FILE       *output_fp;                  /* random file pointer */
FILE       *raster_fp;                  /* raster file pointer */
int        obtain_report();             /* dispatch routine forward declaration */
long       next_offset = 0;             /* offset from beginning of random file*/
struct     block_info reconstruct[NUMBLOCK];          /* image reconstruction info */
struct     output_rt *rt_output;        /* result sent back from rtrace servers */

main(argc, argv)                                                    main

int     argc;
char    *argv[];

{       /* declaration for main() */

            register SVCXPRT *transprt;   /* service transport handle */

        /* end of declaration for main() */
```

...main

```
                    /*-----------------------------------*/

    /* initialize filenames and pointer memory allocation */
    strcpy(outputfile, argv[1]);
    strcpy(rasterfile, argv[2]);
    rt_output = (struct output_rt *)malloc(sizeof(struct output_rt));
    if (rt_output == NULL)
        {
            fprintf(stderr, "malloc: error in main of collect\n");
            exit(-1);
        }

    /* create server socket, memory stream and service handle */
    if ((transprt = svctcp_create(RPC_ANYSOCK, BUFSIZ, BUFSIZ)) == NULL)
        {
            fprintf(stderr, "svctcp_create: error in main of collect \n");
            exit(1);
        }

    /* erase anytrace, make sure this is the update version */
    pmap_unset(COL_PROG, COL_VERS);

    /* register service, call back procedures from remote machines */
    if(!svc_register(transprt,COL_PROG,COL_VERS,obtain_report,IPPROTO_TCP))
        {
            fprintf(stderr, "svc_register: error in main of collect \n");
            exit(1);
        }

    /* run service routine */
    svc_run();

    /* should not reach this point */
    fprintf(stderr, "Error: svc_run returned \n");
    exit(-1);

}   /* end of main() */

/*-------------------------------------------------------------------------------

    ROUTINE      : obtain_report()

    DESCRIPTION : this is the collection service dispatch routine, one of the
                  following five cases will be taken care:
                      1- COL_COLCT: collect result from ray tracing servers in
                                    random order, write to a file, update progress
                                    report structure,
                      2- COL_KILL: unmap service, stop the server,
                      3- COL_QUERY: send progress report to distribution client,
                      4- COL_RECON: reconstruct image, rewrite the result in
                                    random file to raster order,
                      5- default: print "nomatch" error message.

-------------------------------------------------------------------------------*/

obtain_report(rqstp, transprt)                                  obtain_report

register struct  svc_req *rqstp;            /* contains procedure number */
register SVCXPRT *transprt;                 /* service traansport handle */

{       /* declarations for obtain_report() */
```

```
        int     i;                      /* loop control variables */
        int     length, howlong;        /* # of pixels for result scanline */
        int     px_2write;              /* # of pixels to write to rstrfile*/
        int     px_written;             /* # of pixels written to rstrfile */
        int     block_done;             /* # of done scanline */
        long    offset;                 /* starting position in randomfile */
        long    file_length;            /* starting position 4 nxt scanline*/
        bool_t  xdr_rtoutput();         /* xdr for output from rtrace */
        bool_t  xdr_progress();         /* xdr for progress report */
        COLR    buff[BLOCK];            /* temp storage 4 file transfer */
        void    svc_unregister();

        extern  char outputfile[];
        extern  char rasterfile[];
        extern  FILE *output_fp;
        extern  FILE *raster_fp;
        extern  bool_t progress[];
        extern  long next_offset;
        extern  struct output_rt *rt_output;
        extern  struct block_info reconstruct[];

        /* end of declarations for obtain_report() */

                        /*-------------------------*/

switch ((int)rqstp->rq_proc)
    {   /* begin of switch */

        case COL_COLCT_PROC:        /* collect result */

            {   /* begin of case COL_COLCT_PROC */

                /* get arguments, i.e. result from rtrace */
                if (!svc_getargs(transprt, xdr_rtoutput, rt_output))
                    {
                        svcerr_decode(transprt);
                        return(0);
                    }

                /* which scanline is done ? */
                block_done = rt_output->done_block;

                /* update progress report structure */
                progress[block_done] = DONE;

                /* write result to random file */
                if ((output_fp = fopen(outputfile, "a")) == NULL)
                    {
                        fprintf(stderr, "fopen: random file in obtain_report\n");
                        return(0);
                    }
                length = rt_output->block_length;
                fwrite(rt_output->result, sizeof(COLR), length, output_fp);
                fclose(output_fp);

                /* update structure for image reconstruction */
                reconstruct[block_done].length = length;

                /* record it as start position for image reconstruct */
                reconstruct[block_done].start_position = next_offset;

                /* set up offset for next scanline */
                next_offset = next_offset + (length * sizeof(COLR));
```

```
            /* send signal of completion */
            if (!svc_sendreply(transprt, xdr_void, 0))
                {
                    fprintf(stderr, "svc_sendreply: error in obtain_report\n");
                    return(0);
                }

            /* print message for user's interest */
            fprintf(stdout, "Block %d collected\n", block_done);

            /* finish, exit */
            return(1);

        }   /* end of case COL_COLCT_PROC */

    case COL_KILL_PROC:             /* unregister service */

        {   /* begin of case COL_KILL_PROC */

            /* send signal back to client */
            if (!svc_sendreply(transprt, xdr_void, 0))
                {
                    fprintf(stderr, "svc_sendreply: error in obtain_report \n");
                    return(0);
                }

            /* unmap service */
            svc_unregister(COL_PROG, COL_VERS);

            /* done, exit */
            exit(1);

        }   /* end of case COL_KILL_PROC */

    case COL_QUERY_PROC:            /* check progess report */

        {   /* begin of case COL_QUERY_PROC */

            /* send the boolean array marking scanline done back */
            if (!svc_sendreply(transprt, xdr_progress, progress))
                {
                    fprintf(stderr, "svc_sendreply: error in obtain_report \n");
                    return(0);
                }

            /* exit */
            return(1);

        }   /* end of case COL_QUERY_PROC */

    case COL_RECON_PROC:            /* reconstruct image */

        {   /* begin of case COL_RECON_PROC */

            /* open random and raster files */
            if ((output_fp = fopen(outputfile, "r")) == NULL)
                {
                    fprintf (stderr, "fopen: random file in obtain_report\n ");
                    return (0);
                }

            if ((raster_fp = fopen(rasterfile, "w")) == NULL)
                {
                    fprintf (stderr, "fopen: raster file in obtain_report\n");
                    return(0);
```

```
    }

/* do until the last block of scanlines */
for(i = 0; i < NUMBLOCK; i++)
    {   /* begin of for loop */

        /* find the starting point for scanline */
        offset =  reconstruct[i].start_position;

        /* get the scanline length */
        howlong = reconstruct[i].length;

        /* search for the right place in random file */
        if (fseek(output_fp, offset, 0) != 0)
            {
                fprintf(stderr,"fseek: random file in obt_reprt\n");
                return (0);
            }

        /* get info for the entire scanline */
        px_2write = fread(buff, sizeof(COLR), howlong, output_fp);
        if (px_2write != howlong)
            {
                fprintf(stderr, "fread: copy file in obt_report\n");
                return(0);
            }

        /* write into the final raster file */
        px_written = fwrite(buff, sizeof(COLR), px_2write,
                                raster_fp);
        if (px_written != px_2write)
            {
                fprintf(stderr, "fwrite: copy file in obt_report\n");
                return(0);
            }

        /* print message for user's interest */
        fprintf(stdout, "block %d written \n", i);

    }   /* end of for loop */

/* close all file pointers */
if (fclose(raster_fp) != 0)
    {
        fprintf (stderr, "fclose: raster file in obt_report\n");
        return(0);
    }
if (fclose(output_fp) != 0)
    {
        fprintf (stderr, "fclose: random file in obt_report\n");
        return(0);
    }

/* send signal for completion */
if (!svc_sendreply(transprt, xdr_void, 0))
    {
        fprintf(stderr, "svc_sendreply: error in obtain_report \n");
        return(0);
    }

/* exit */
return(1);
```

```
            }   /* end of case COL_RECON_PROC */

        default:
            {
                svcerr_noproc(transprt);
                return(0);
            }
        }   /* end of switch */

}   /* end of receive_report() */
```

/*----------

ROUTINE : xdr_rtoutput()

DESCRIPTION : eXternal Data Representation routine for the result of a scanline sent from Sun workstations back to VAX side. This routine is called to decode the result sent back. The result consists of the followings:
- *an integer for the number of the block done,*
- *an integer for the number of "COLR" items contained in block of results (this number could be different from (BLOCKSIZE * 512) due to run-length encoding,*
- *array of pixel valaues, each pixel is expressed as four unsigned chars: 1 RED, 1 BLUE, GREEN, 1 EXPONENT.*

----------*/

xdr_rtoutput

```
bool_t
xdr_rtoutput(xdrs,rt_out)

XDR     *xdrs;
struct output_rt *rt_out;

{       /* declaration for xdr_rtoutput() */

        u_int     maxsize;          /* maximum number of elements */

        /* end of declaration for xdr_rtoutput() */

                        /*----------*/

    maxsize         = XRES * 4 * BLOCKSIZE;

    if(!xdr_int(xdrs, &rt_out->done_block))
        return(FALSE);

    if (!xdr_int(xdrs, &rt_out->block_length))
        return(FALSE);

    if(!xdr_opaque(xdrs, rt_out->result,   maxsize))
        return(FALSE);

    return(TRUE);

}   /* end of xdr_output() */
```

/*----------

ROUTINE : xdr_progress()

DESCRIPTION : this is the eXternal Data Representation for the array of boolean type of YRES elements, which contains information on whether or not the scanline result has been recorded in the rasterfile. "DONE" or 1 means the result had been written

into the file. "NOT_YET" or 0 indicates that the result has not been received. This routine is called to encode the progress before sending it to the distribution client.

```
-------------------------------------------------------------------------*/

bool_t
xdr_progress(xdrs, progress)                                    xdr_progress

XDR     *xdrs;
bool_t progress[NUMBLOCK];

{           /* declarations for xdr_progress() */

            int k;          /* loop control variable */

            /* end of declarations for xdr_progress() */

                        /*-------------------------------*/

    for (k = 0; k < NUMBLOCK; k++)
        {   if (!xdr_bool(xdrs, &progress[k]))
                return(FALSE);
        }

    return(TRUE);

}   /* end of xdr_progress() */

/*-------------------------------------------------------------------------*/
```

Appendix B: Program Profiles

/* profile for "distribute_client.c"

%time	cumsecs	#call	ms/call	name
14.8	0.17	64	2.66	_write
12.2	0.31	72	1.94	_select
6.1	0.38	65	1.08	_read
5.2	0.44	10	6.00	_connect
5.2	0.50	871	0.07	_ntohl
3.5	0.54	62	0.65	_callrpctcp
3.5	0.58			_xdrrec_create
2.6	0.61			_clnttcp_create
2.6	0.64			_clntudp_bufcreate
2.6	0.67	1	30.00	_distribute
2.6	0.70	103	0.29	_malloc
2.6	0.73			_pmap_getport
2.6	0.76	15	2.00	_socket
2.6	0.79			_vfork
1.7	0.81	7	2.86	__doprnt
1.7	0.83	45	0.44	_gettimeofday
1.7	0.85	1	20.00	_main
1.7	0.87			5 4.00 _recv
1.7	0.89	5	4.00	_sendto
1.7	0.91			_xdr_bool
1.7	0.93			mcount
1.3	0.95			_xdr_long
1.3	0.96			_xdr_u_int
0.9	0.97			_authnone_create
0.9	0.98	2	5.00	_fprintf
0.9	0.99	6	1.67	_ioctl
0.9	1.00			_monstartup
0.9	1.01	17	0.59	_morecore
0.9	1.02	2	5.00	_open
0.9	1.03	5	2.00	_res_mkquery
0.9	1.04	19	0.53	_sbrk
0.9	1.05	5	2.00	_send
0.9	1.06	45	0.22	_setitimer
0.9	1.07	172	0.06	_strcmp
0.9	1.08			_xdr_enum
0.9	1.09			_xdr_opaque
0.9	1.10	16	0.63	_xdr_progress
0.9	1.11			_xdr_u_long
0.9	1.12			_xdr_void
0.9	1.13			_xdrmem_create
0.9	1.14			_xdrrec_endofrecord
0.9	1.15			_xdrrec_skiprecord
0.0	1.15	5	0.00	__doscan
0.0	1.15	3	0.00	__filbuf
0.0	1.15	2	0.00	__findiop
0.0	1.15	3	0.00	__flsbuf
0.0	1.15	344	0.00	_bcopy
0.0	1.15	5	0.00	_bzero
0.0	1.15	11	0.00	_close
0.0	1.15	5	0.00	_dn_comp
0.0	1.15	6	0.00	_dn_expand
0.0	1.15	10	0.00	_dn_find
0.0	1.15	5	0.00	_dn_skip
0.0	1.15	2	0.00	_execl
0.0	1.15	2	0.00	_execv
0.0	1.15	2	0.00	_execve
0.0	1.15	1	0.00	_fclose
0.0	1.15	1	0.00	_fflush
0.0	1.15	3	0.00	_fgets
0.0	1.15	2	0.00	_fopen
0.0	1.15	101	0.00	_free
0.0	1.15	5	0.00	_fscanf

0.0	1.15	3	0.00	_fstat
0.0	1.15	5	0.00	_gethostbyname
0.0	1.15	1	0.00	_getpagesize
0.0	1.15	10	0.00	_getpid
0.0	1.15	18	0.00	_getshort
0.0	1.15	264	0.00	_htonl
0.0	1.15	27	0.00	_htons
0.0	1.15	1	0.00	_image
0.0	1.15	6	0.00	_index
0.0	1.15	1	0.00	_inet_addr
0.0	1.15	1	0.00	_isatty
0.0	1.15	10	0.00	_ntohs
0.0	1.15	1	0.00	_profil
0.0	1.15	10	0.00	_putshort
0.0	1.15	5	0.00	_recvfrom
0.0	1.15	1	0.00	_res_init
0.0	1.15	5	0.00	_res_send
0.0	1.15	15	0.00	_sigblock
0.0	1.15	8	0.00	_signal
0.0	1.15	15	0.00	_sigpause
0.0	1.15	15	0.00	_sigsetmask
0.0	1.15	38	0.00	_sigvec
0.0	1.15	15	0.00	_sleep
0.0	1.15	5	0.00	_sprintf
0.0	1.15	45	0.00	_strlen
0.0	1.15	3	0.00	_strncmp
0.0	1.15	1	0.00	_strncpy
0.0	1.15	2	0.00	_system
0.0	1.15	5	0.00	_ungetc
0.0	1.15	2	0.00	_wait
0.0	1.15	20	0.00	_xdr_rtinput

/* profile for "raytracing_server.c" */

%time	cumsecs	#call	ms/call	name
18.9	14.27	307500	0.05	Fmuld
17.0	27.10	307350	0.04	Faddd
10.7	35.16	1	8059.88	_profil
8.8	41.78			mcount
6.3	46.51			d_exte
6.1	51.09	102400	0.04	_fwrite
5.8	55.45	5	871.99	_gen_input
5.4	59.51			d_usel
4.6	63.01	102400	0.03	Ffltd
4.2	66.15			d_rcp
3.4	68.75	153615	0.02	Fdtos
2.3	70.52	102522	0.02	_memcpy
1.7	71.84	376	3.51	_write
1.5	72.97	102470	0.01	Fsubd
1.3	73.95			d_norm
0.4	74.25	50	6.00	Fdivd
0.3	74.49	21	11.43	_open
0.2	74.67	52	3.46	_read
0.2	74.83	110	1.45	Fcmpd
0.1	74.93	311	0.32	_bcopy
0.1	75.01			d_pack
0.1	75.08			_rewind
0.1	75.14	12	5.00	_svc_getreqset
0.1	75.18	1	40.00	_clntudp_create
0.1	75.22	31	1.29	_close
0.1	75.26			ptwo
0.0	75.29			Frintd
0.0	75.32			Frints
0.0	75.35	20	1.50	Fscaleid
0.0	75.38	5	6.00	_ftell
0.0	75.40	4	5.00	_flock
0.0	75.42	5	4.00	_fork
0.0	75.44			_getenv
0.0	75.46	2	10.00	_gethostbyname
0.0	75.48	6	3.33	_lseek
0.0	75.50	18	1.11	_recvfrom
0.0	75.52	46	0.43	_select
0.0	75.54	18	1.11	_sendto
0.0	75.56	2	10.00	_xdrrec_create
0.0	75.58	16	1.25	_xdrrec_endofrecord
0.0	75.59			Fund
0.0	75.60	51	0.20	_bzero
0.0	75.60	11	0.00	__authenticate
0.0	75.60	25	0.00	__doscan
0.0	75.60	37	0.00	__filbuf
0.0	75.60	17	0.00	__findbuf
0.0	75.60	17	0.00	__findiop
0.0	75.60	45	0.00	__fp_normalize
0.0	75.60	45	0.00	__fp_rightshift
0.0	75.60	15	0.00	__fp_set_exception
0.0	75.60	75	0.00	__mul_65536
0.0	75.60	45	0.00	__pack_double
0.0	75.60	48	0.00	__rpc_dtablesize
0.0	75.60	23	0.00	__seterr_reply
0.0	75.60	11	0.00	__svcauth_null
0.0	75.60	7	0.00	__wrtchk
0.0	75.60	160	0.00	__xflsbuf
0.0	75.60	3	0.00	__yp_dobind
0.0	75.60	1	0.00	_accept
0.0	75.60	8	0.00	_authnone_create
0.0	75.60	12	0.00	_bind
0.0	75.60	9	0.00	_bindresvport

0.0	75.60	3	0.00	_calloc
0.0	75.60	10	0.00	_callrpc
0.0	75.60	5	0.00	_callrpctcp
0.0	75.60	1	0.00	_clnttcp_create
0.0	75.60	7	0.00	_clntudp_bufcreate
0.0	75.60	1	0.00	_connect
0.0	75.60	15	0.00	_decimal_to_binary_fraction
0.0	75.60	30	0.00	_decimal_to_binary_integer
0.0	75.60	55	0.00	_decimal_to_double
0.0	75.60	45	0.00	_decimal_to_unpacked
0.0	75.60	11	0.00	_dist_rt
0.0	75.60	2	0.00	_endhostent
0.0	75.60	1	0.00	_exit
0.0	75.60	17	0.00	_fclose
0.0	75.60	1	0.00	_fcntl
0.0	75.60	10	0.00	_fcross
0.0	75.60	12	0.00	_fflush
0.0	75.60	20	0.00	_fgets
0.0	75.60	55	0.00	_file_to_decimal
0.0	75.60	1	0.00	_finitfp_
0.0	75.60	17	0.00	_fopen
0.0	75.60	10	0.00	_fprintf
0.0	75.60	5	0.00	_fread
0.0	75.60	32	0.00	_free
0.0	75.60	20	0.00	_fscanf
0.0	75.60	15	0.00	_fstat
0.0	75.60	3	0.00	_get_myaddress
0.0	75.60	1	0.00	_getdomainname
0.0	75.60	1	0.00	_getdtablesize
0.0	75.60	10	0.00	_gethostname
0.0	75.60	1	0.00	_getpagesize
0.0	75.60	12	0.00	_getpid
0.0	75.60	4	0.00	_getsockname
0.0	75.60	10	0.00	_gettimeofday
0.0	75.60	2	0.00	_inet_addr
0.0	75.60	30	0.00	_ioctl
0.0	75.60	17	0.00	_isatty
0.0	75.60	5	0.00	_kill
0.0	75.60	1	0.00	_listen
0.0	75.60	1	0.00	_main
0.0	75.60	68	0.00	_malloc
0.0	75.60	20	0.00	_memccpy
0.0	75.60	10	0.00	_memchr
0.0	75.60	10	0.00	_normalize
0.0	75.60	1	0.00	_on_exit
0.0	75.60	2	0.00	_pmap_getport
0.0	75.60	1	0.00	_pmap_set
0.0	75.60	2	0.00	_pmap_unset
0.0	75.60	5	0.00	_rtrace_work
0.0	75.60	6	0.00	_sbrk
0.0	75.60	5	0.00	_setcommand
0.0	75.60	2	0.00	_sethostent
0.0	75.60	15	0.00	_setitimer
0.0	75.60	5	0.00	_setview
0.0	75.60	5	0.00	_sigblock
0.0	75.60	5	0.00	_sigpause
0.0	75.60	5	0.00	_sigsetmask
0.0	75.60	10	0.00	_sigvec
0.0	75.60	5	0.00	_sleep
0.0	75.60	13	0.00	_socket
0.0	75.60	3	0.00	_sprintf
0.0	75.60	5	0.00	_sscanf
0.0	75.60	5	0.00	_statistics
0.0	75.60	35	0.00	_strcat
0.0	75.60	11	0.00	_strcmp

0.0	75.60	33	0.00	_strcpy
0.0	75.60	26	0.00	_strlen
0.0	75.60	2	0.00	_strncpy
0.0	75.60	14	0.00	_strpbrk
0.0	75.60	1	0.00	_svc_register
0.0	75.60	1	0.00	_svc_run
0.0	75.60	11	0.00	_svc_sendreply
0.0	75.60	1	0.00	_svc_unregister
0.0	75.60	1	0.00	_svcfd_create
0.0	75.60	1	0.00	_svctcp_create
0.0	75.60	190	0.00	_ungetc
0.0	75.60	1	0.00	_usingypmap
0.0	75.60	5	0.00	_wait
0.0	75.60	5	0.00	_wait4
0.0	75.60	34	0.00	_xdr_accepted_reply
0.0	75.60	8	0.00	_xdr_bool
0.0	75.60	67	0.00	_xdr_bytes
0.0	75.60	8	0.00	_xdr_callhdr
0.0	75.60	11	0.00	_xdr_callmsg
0.0	75.60	9	0.00	_xdr_datum
0.0	75.60	179	0.00	_xdr_enum
0.0	75.60	205	0.00	_xdr_int
0.0	75.60	454	0.00	_xdr_long
0.0	75.60	39	0.00	_xdr_opaque
0.0	75.60	58	0.00	_xdr_opaque_auth
0.0	75.60	5	0.00	_xdr_pmap
0.0	75.60	34	0.00	_xdr_replymsg
0.0	75.60	5	0.00	_xdr_rtinput
0.0	75.60	5	0.00	_xdr_rtoutput
0.0	75.60	16	0.00	_xdr_string
0.0	75.60	83	0.00	_xdr_u_int
0.0	75.60	230	0.00	_xdr_u_long
0.0	75.60	2	0.00	_xdr_u_short
0.0	75.60	34	0.00	_xdr_union
0.0	75.60	26	0.00	_xdr_void
0.0	75.60	3	0.00	_xdr_ypdomain_wrap_string
0.0	75.60	3	0.00	_xdr_ypmap_wrap_string
0.0	75.60	3	0.00	_xdr_ypreq_key
0.0	75.60	6	0.00	_xdr_ypresp_val
0.0	75.60	27	0.00	_xdrmem_create
0.0	75.60	10	0.00	_xdrrec_eof
0.0	75.60	16	0.00	_xdrrec_skiprecord
0.0	75.60	2	0.00	_xprt_register
0.0	75.60	3	0.00	_yp_get_default_domain
0.0	75.60	3	0.00	_yp_match
0.0	75.60	1	0.00	_ypprot_err

*/*profile for "collect_server.c" */*

%time	cumsecs	#call	ms/call	name
35.3	4.66	1239	3.76	_read
27.2	8.25	385	9.33	_write
16.5	10.43	1129	1.93	_select
13.3	12.18	2985	0.59	_bcopy
1.3	12.35			_svcfd_create
1.1	12.50			_xdrrec_endofrecord
0.9	12.62			mcount
0.7	12.71	40	2.25	__doprnt
0.5	12.78	22	3.18	_open
0.4	12.83			_xdrrec_create
0.3	12.87	348	0.11	__flsbuf
0.2	12.90	1279	0.02	_ntohl
0.2	12.93			_xdr_callmsg
0.2	12.96	24	1.25	_sbrk
0.2	12.99	3	10.00	_sendto
0.2	13.01	28	0.71	_close
0.2	13.03	40	0.50	_fwrite
0.2	13.05	55	0.36	_lseek
0.2	13.07	58	0.34	_malloc
0.2	13.09			_monstartup
0.2	13.11			_xdr_replymsg
0.1	13.12	156	0.06	__filbuf
0.1	13.13	23	0.43	_fstat
0.1	13.14			_get_myaddress
0.1	13.15	1	10.00	_getsockname
0.1	13.16			_pmap_unset
0.1	13.17	3	3.33	_recvfrom
0.1	13.18	7	1.43	_socket
0.1	13.19			_svc_getreq
0.1	13.20			_xdr_opaque_auth
0.0	13.20	22	0.00	__findiop
0.0	13.20	5	0.00	_accept
0.0	13.20	1	0.00	_bind
0.0	13.20	22	0.00	_fclose
0.0	13.20	22	0.00	_fflush
0.0	13.20	22	0.00	_fopen
0.0	13.20	40	0.00	_fprintf
0.0	13.20	20	0.00	_fread
0.0	13.20	29	0.00	_free
0.0	13.20	20	0.00	_fseek
0.0	13.20	1	0.00	_getpagesize
0.0	13.20	3	0.00	_getpid
0.0	13.20	3	0.00	_gettimeofday
0.0	13.20	620	0.00	_htonl
0.0	13.20	3	0.00	_htons
0.0	13.20	10	0.00	_ioctl
0.0	13.20	1	0.00	_isatty
0.0	13.20	1	0.00	_listen
0.0	13.20	1	0.00	_main
0.0	13.20	22	0.00	_morecore
0.0	13.20	1	0.00	_ntohs
0.0	13.20	38	0.00	_obtain_report
0.0	13.20	1	0.00	_profil
0.0	13.20	2	0.00	_strcpy
0.0	13.20	16	0.00	_xdr_progress
0.0	13.20	20	0.00	_xdr_rtoutput

LAWRENCE BERKELEY LABORATORY
TECHNICAL INFORMATION DEPARTMENT
1 CYCLOTRON ROAD
BERKELEY, CALIFORNIA 94720