A Workbench for Realistic Image Synthesis

by

Brian D. Corrie

B.Sc., University of Victoria, British Columbia, 1988

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE 1990-04-25

We accept this thesis as conforming
to the required standard

[redacted]
Dr. H. A. Müller

[redacted]
Dr. D. M. Miller

[redacted]
Dr. P. Agathoklis

[redacted]
Dr. D. Szafron

ISBN 0-315-62393-4

Supervisor: Dr. H. A. Müller

Abstract

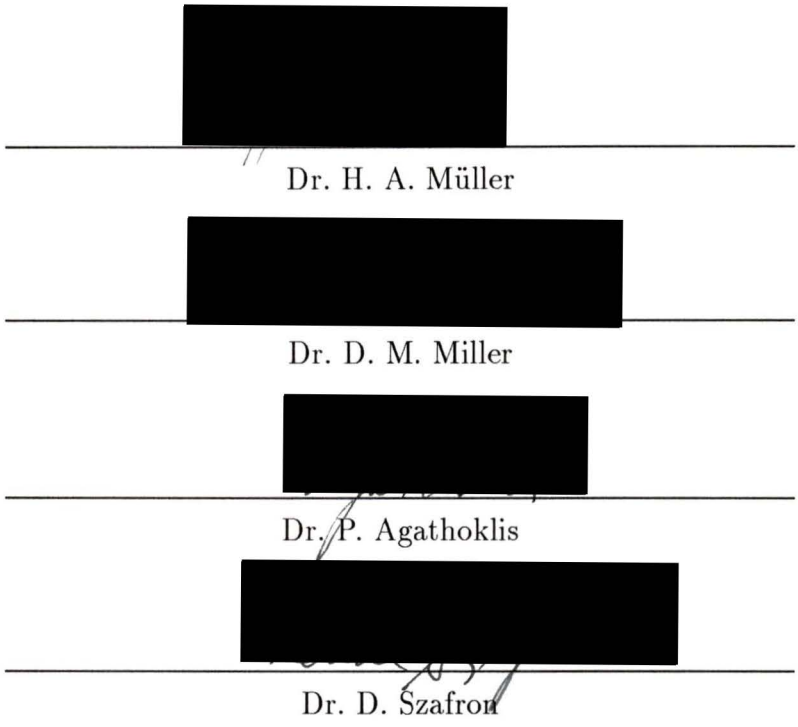
There are two primary tasks in the computer generation of photorealistic images, the modeling of a three-dimensional scene, and the rendering of that scene to create a two-dimensional image. An image synthesis workbench is needed to perform these tasks effectively and efficiently. The workbench provides a cohesive system that integrates the various stages of the image synthesis process. This thesis presents such a workbench.

The modeling component of the workbench, *Scene*, is a visual, three-dimensional, editor. It allows the user to interactively create, edit, and visualize the structure of a complex three-dimensional scene, significantly simplifying the modeling process.

The rendering component of the workbench, *SunRay*, is a ray tracing rendering system. Ray tracing is a powerful rendering technique that can simulate complex light interactions such as shadows, reflection, and refraction. The SunRay rendering system provides the photorealistic quality expected by the users of todays graphics systems.

To reduce the rendering time of the image, which on a single computer can be excessive, the SunRay renderer is designed to run in a distributed fashion on a loosely-coupled, heterogeneous network of computers. The system features an adaptive, load-balanced, image space ray tracing algorithm. The performance figures of the distributed ray tracer compare favorably with implementations reported in the literature. The result is a near linear reduction (with respect to the number of processors used) in rendering time.

Examiners:



Dr. H. A. Müller

Dr. D. M. Miller

Dr. P. Agathoklis

Dr. D. Szafron

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my thesis committee: Hausi Müller, Mike Miller, and Pan Agathoklis. Their support and suggestions were invaluable in the writing of this thesis. A special thanks to Hausi Müller, the eternal optimist, who always made it seem like there was light at the end of the tunnel.

Many thanks go to the people at the University of Victoria, especially those in the Computer Science Department. Pan Agathoklis introduced me to the world of Computer Graphics in a first year Engineering course. I have been obsessed ever since. Hausi Müller fueled the obsession with advanced courses and a trip to the ACM SIGGRAPH conference. Special thanks to both of these gentleman. Ali Shoja, Nigel Horspool, and Mike Miller provided interesting and diverse undergraduate courses that prepared me for graduate studies.

Thanks is sent to the Usenet community, especially those who participate in the comp.graphics discussions. Usenet was a valuable source of information for many of the ideas used in this work. A special note of thanks to Eric Haines, who managed to get a copy of a crucial paper to me in time for the results to be compared with this thesis.

My fellow students also provided much assistance. Thanks to Craig Sinclair, Brad Richards, and Jim Uhl for their proof reading and editorial comments, and to Rob Side, the local REM wizard. For providing the ideal mixture of distraction and support, I owe thanks to: Peter Walsh, Paul Strooper, Randal Tomczuk, Rod Byrne, Mike Whitney, Eric Davies, Katherine Franklin, Colin Woytowich, and the Sunday Volleyball Gang. Gary Duncan, Alan Idler, and Will Kastelic provided excellent system support, helping in both the writing and implementation of the ideas in this thesis.

The support received from family and friends has been crucial in the completion of this thesis. My family has been behind me one hundred percent in my work, and were always there to provide support and encouragement. Thanks to my good friends Bruce McGuire, Mitch Savage, and Sandi Peterson, who have listened to my ramblings about Computer Graphics for the last four years. A special thanks to Sherri Macdonald, who was there to encourage when I needed encouragement, and was in Ottawa when I needed to write. You probably wouldn't have seen much of me anyway.

Chapter 1

Introduction

It has long been a goal of computer graphics to produce photorealistic images, that is, images that cannot be discerned from an actual photograph of a three-dimensional scene. Realistic computer generated images are important in areas such as flight simulation, the entertainment industry, and mechanical design.

Attempting to create photorealistic images is an enormous undertaking. To be successful in this task, many skills and tools are required. It is necessary, in most cases, to build an image from scratch by:

- defining the three-dimensional objects in the scene;
- positioning the objects in three-dimensional space;
- describing the surface characteristics of the objects;
- describing the environment the scene exists in;
- creating the image, based on the characteristics defined in the previous steps.

Each of these tasks has its own problems and requires an efficient and practical solution to make realistic image synthesis feasible. An image synthesis workbench provides a cohesive, practical solution to these problems. By defining and adhering to standards, the components of the workbench provide a sophisticated and powerful foundation upon which complex, realistic images can be created.

The tasks listed above can be divided into two fundamental stages: the modeling or creation of a complex three-dimensional scene, and the rendering of the scene to create a two-dimensional image. As Steve Upstill describes the process in his book, *The RenderMan Companion* [UPSTILL 90]:

> "An astronomer recording images of distant galaxies with an hours-long time exposure. A fashion photographer snapping models in his cunningly lit studio. An Ansel Adams or Elliot Porter capturing the essence of natural grandeur on film. A proud father trying to get his whole squirming family together in the view finder of his camera. These people are all creating images: putting a camera in front of some scene, providing or waiting for appropriate lighting, then focusing that light on film.
>
> Computers mimic that process to make synthetic images."

Modeling is performed by both photographers and computer modelers. The photographer models a scene by positioning the subject of the photograph, placing props in aesthetically pleasing locations, and determining the backdrop and appropriate lighting for the scene. The computer modeler performs the same steps, but instead of physical objects, lights, and cameras, an internal representation of a non-existent scene is manipulated. Not only must the subject and props be placed by the modeler, but most of them must actually be built first. Thus, modeling is a creative process, allowing the modeler to create objects that do not exist in the real world. This is

one of the most attractive features of computer graphics, and allows users to visualize physical processes (such as air flow around an aircraft's wing), architectural structures, and mechanical parts that either do not exist, or cannot be seen by the human perceptual system.

Taking a photograph of a scene is a physical process, centering around the way light interacts with the objects in the environment. When an image is created, be it by exposing a piece of film to light, or by simulating the process through the use of a computer, the light interactions obey a strict set of laws.

In the real world, these laws are the laws of physics. In the simulated world, the laws are defined by sophisticated algorithms that are applied to the computer model. It is clear that modeling natural phenomena with the rendering process is difficult. There are an infinite number of subtly different phenomena that can cause a computer synthesized image to look unrealistic. To demonstrate the complex visual qualities of the natural world, inspect your surroundings. There are a multitude of surface properties and surface anomalies (scuffs, scrapes, chips, etc.) that give an image its realistic "feeling." To reproduce these characteristics in a computer generated image is very difficult, and requires a large amount of computation. Although the technology used today comes close to achieving photorealistic images, the techniques used model a very narrow spectrum of the phenomena observed in nature. When the model used does not encompass a feature essential to an image, the human mind is very adept at recognizing that something is missing. Usually, a feeling of "something is wrong with this picture," rather than an obvious error in the image, reveals a weakness in the rendering model.

Because the laws used to render an image are static, that is, they remain the same as different models are rendered, the rendering process is a stable one. Contrary to this, as new users require new models, the features demanded of a modeling system

change. Because modeling is a dynamic process, and rendering is relatively static, these components of the image synthesis system should be separated. To enable modeling and rendering software to work as a cohesive unit, an interface between these two vital components should be provided. If they adhere to this interface, any modeling system can use any rendering system, enabling various levels of realism to be used to render an image.

1.1 The problem

The two major problems in creating realistic computer generated images coincide with the two stages of the synthesis process. The first step is to create a three-dimensional model of the scene to be rendered. Because the model is different each time an image is rendered, three-dimensional modeling is a crucial aspect of image synthesis. Without modeling tools, complex scenes cannot be created. Modeling is a dynamic process, and it is therefore necessary to provide a flexible and extensible modeling tool. The definition of operations that can be applied to both scenes, and the primitive objects that exist in a scene, simplifies the creation and expansion of such a system. The building of complex scenes can be streamlined by visualizing the three dimensional model as it is being built.

There are currently a large number of rendering systems available, but only the powerful, and therefore expensive, rendering systems are accompanied by sophisticated modeling tools. Often, the model description produced by the modeler is proprietary, and only the accompanying rendering system can access the modeling information. A standard interface between modelers and renderers is needed. The interface allows the flexible coupling of powerful modeling tools (such as *Computer Aided Design* or *CAD* systems) and sophisticated rendering systems.

To create a realistic scene requires a very large number of modeled objects. When these objects are rendered with a complex shading model, enormous amounts of computing resources are required. A tradeoff currently exists between the time it takes to render an image, and the realism that can be produced in that time. The more complex and realistic the image, the longer the rendering will take. Research into realistic computer graphics proceeds in essentially two directions: increasing the realism of a rendered image, and increasing the speed of the rendering process. Realistic algorithms, such as *Ray Tracing* [GLASSNER 89A] and *Radiosity* [COHEN 85] have been proposed, but both techniques take an immense number of computing resources to render complex images. More efficient algorithms and parallel architectures have been suggested to reduce the rendering times for both of these techniques.

To create a system for realistic image synthesis two problems must be solved: the creation of complex three-dimensional models, and the expeditious rendering of those models to create realistic images. This thesis presents a flexible and powerful system that solves both of these problems.

1.2 The approach

To create an image synthesis workbench, distinct modeling and rendering components must be created. Each component must be able to communicate with the other components in the system. This is accomplished by creating a *scene description language* or *rendering interface*, that modeling and rendering components of the workbench use to describe three-dimensional models. The problem is now reduced to constructing the components of the image synthesis workbench.

The modeling component is provided by a *model editor*. The model editor makes use of three-dimensional projections, a bit mapped raster display, and a pointing de-

vice to provide a powerful modeling and visualization tool. Based on the definition of primitive objects, and the operations that can be performed on these objects, flexibility is maintained. Other modeling utilities that adhere to the rendering interface complete the modeling requirements of an image synthesis workbench.

Ray tracing is an elegant rendering technique, modeling the physics of a camera lens and the way it gathers light to produce realistic computer generated images. As a rendering process, it is well known for two reasons: (1) its elegant handling of shadows, reflection, and refraction, and (2) the immense computational requirements needed to render an image. Obviously, the first reason is a positive one, but the second one is a major drawback. This area has been the subject of much research in the past decade. Both algorithmic and architectural approaches have been used to reduce this problem, and as far back as 1980, in Whitted's landmark ray tracing paper [WHITTED 80], a multiprocessor approach was suggested.

In this thesis, a technique based on three-dimensional *Space Subdivision* is used to reduce the rendering time of the basic ray tracing algorithm [GLASSNER 84]. A *Local Area Network* (LAN) of independent or semi-independent processing elements is used to form a distributed processing environment, allowing the rendering process to be parallelized. This architectural configuration is becoming more and more prominent in the computing community, and provides a flexible and extensible processing core. To produce an efficient rendering system, processing power in the LAN must be utilized to as high a degree as possible. This is accomplished by developing and implementing a dynamic load-balancing strategy that adapts to the changing conditions that are predominant in a LAN. Workstations, which are commonly found in a LAN environment, are used to perform the three-dimensional, visual modeling.

1.3 Overview

The main thrust of this work is to provide a workbench for realistic image synthesis. To achieve this goal, both modeling and rendering systems were developed. A rendering interface, and other tools, are provided to give a complete, cohesive image synthesis system.

Chapter 2 provides background material on three-dimensional modeling. It also presents the modeling system and rendering interface developed for this research. Chapter 3 provides the background material needed to follow the development of the distributed ray tracing system. Chapter 4 describes the evolutionary progression of an exhaustive ray tracing system into a dynamic, load-balanced, parallel ray tracing system. Chapter 5 summarizes the results and contributions of this thesis, and suggests some directions for future research. The rendering interface and other image synthesis utilities are discussed further in the Appendices.

Chapter 2

Three-dimensional modeling

One of the most difficult tasks to perform when creating photorealistic computer generated images is the modeling of a three-dimensional scene. Consider the following scenario: your employer, an architectural firm, asks you to produce a realistic image of a hotel lobby that you are designing, to be shown to the firm's client and owner of the hotel. How do you approach the problem? The standard solution is to produce a freehand sketch of the lobby to show the client. Of course, living in the age of electronics, a computer may also be used to produce the image. The question is, which would be easier to produce, more useful in the long run, and more efficient in terms of time and money? While the standard solution is adequate, by using computer generated imagery, a more flexible and powerful visualization of the project can be produced.

By using a computer system to create and render the lobby, a view from any location, or even an animation of a simulated walk through the lobby [FUCHS 89], can be created. To build a three-dimensional *model* of the lobby, a powerful *modeling system* is essential. Without one, the computer system would never be considered as

an alternative to the more standard methods. With a powerful modeling system, the overhead of model creation is kept low, resulting in a powerful and exciting alternative to the standard techniques used by today's architects.

As one might expect, it is a very difficult task to create a model as complex as a hotel lobby, and yet it has already been accomplished with current technology [FUCHS 89]. As mentioned in Chapter 1, rendering technology is approaching the sophistication that is needed for this kind of a project, and once a rendering system is created, it can be used to render all the lobbies in all the hotels of the world. Unfortunately, a problem arises at this point: how do we describe these lobbies to the rendering system? Each lobby is unique, with a different architectural design, illumination, and visual properties. The rendering system needs to be created only once, but the computer model to simulate the three-dimensional scene must be created, usually from scratch, for every image required.

To ease the process of describing a model of a three-dimensional scene, modeling tools are used. These can range from sophisticated modeling and rendering packages, to simple textual utilities that perform some sort of preprocessing on modeling data that has been created manually. It is clear that an image synthesis platform is not complete without some tools of this nature. This chapter describes some of the modeling tools that are used in research and industry today. It also discusses the evolution of a set of modeling tools developed for this research, and the importance of three-dimensional modeling.

2.1 Requirements for modeling software

To perform three-dimensional modeling, it is essential that a distinction be made between the three-dimensional scene and the two-dimensional rendered image of that

scene [FIUME 86]. The scene is a result of the modeling stage of a graphics system, and is used as input to the rendering software. As stated by Fiume, the set of primitives required for scene specification is application dependent. In particular, a given application may require a special type of primitive. For this reason, the modeler must be flexible. A framework should be provided that allows for the creation of primitive definitions that can be modified or implemented by a system designer, as they are demanded by an application. This framework should consist of:

- *primitive graphic objects* (e.g., polygons or spheres);
- the semantics of operations that combine primitive graphic objects into complex structures called *composite graphic objects* (e.g., polygonal surfaces of revolution);
- the organization of composite graphic objects into scenes;
- the semantics of operations applied to composite objects and scenes to change their characteristics (e.g., viewing parameters or composite object surface properties).

Within this framework, we are able to add primitives to a modeler by specifying how each of the operations will affect the new primitive.

A three-dimensional scene is constructed of geometric objects called primitive graphic objects (PGOs). A modeler will support one or more different types of PGOs. Instances of these objects can then be combined into more complex composite graphic objects (CGOs) by operations supported by the modeler. These operations are used to create objects that have a logical relationship to each other, such as a bookshelf, lamp, or table. Composite objects are located in three-dimensional scene space, to form the geometric structure of the scene. A modeler should also be able to change

the surface properties of composite objects, specify the color and location of light sources in a scene, and specify the viewing location and direction that the renderer will use to create an image. Without these operations, the modeler will not satisfy the requirements of even the simplest of rendering systems.

The surface properties of primitives in a scene consist of three major components: the primitive's *color*, *surface type*, and *surface texture*. The color is usually specified by defining the amount of light that is reflected from the surface in each of the three primary colors (red, green, and blue). To specify a bright yellow primitive, the color triple $(1, 1, 0)$ is used. This defines a surface that reflects 100% of the incident red and green light, but absorbs all of the incident blue light. Black is specified as a surface that absorbs all incident light $(0, 0, 0)$ while white is specified as a surface that reflects all incident light $(1, 1, 1)$.

The surface type determines the reflective properties that a surface will have. A highly diffuse surface (such as cardboard) reflects little or none of the incident light from the objects around it. A highly specular surface (such as a mirror) reflects a large amount of the incident light. The diffuse and specular components can also be defined for light transmitting objects. By specifying these properties, objects made of transparent materials can be created. The surface texture of an object can be defined in various ways, ranging from mapping a photograph onto a primitive, to defining a complex procedural texture (such as a bumpy texture). Texture mapping is discussed in more detail in Section 4.2.

2.2 Modeling research

The desire to render a complex, artificial, three-dimensional scene led to the need for three-dimensional modeling software. This, in turn, led to the creation of early

modeling tools, *computer aided design* or CAD packages. Because engineers need to visualize their designs, powerful CAD tools were created. These systems usually lack the capability to apply operations to change object and scene characteristics, such as composite object surface properties and scene illumination. Therefore, they cannot provide complete modeling tools for photorealistic imaging. In spite of this fact, CAD packages are very powerful modeling tools, and are often used due to the lack of better solutions to the modeling problem.

In the computer graphics industry, graphics houses use a multitude of modeling software, ranging from Wavefront's modeler and Alias Research's Alias 2 modeling system, to Pixar's various products and in-house modeling tools [FICHERA 89]. All of these tools are very powerful, as can be seen by the impressively realistic images they have produced, but are usable only with high-end workstations and rendering software. A multitude of rendering systems currently exist, ranging in quality and price, from production quality renderers from the graphics houses, to sophisticated public domain or free rendering systems. All of these rendering systems need to model a three-dimensional scene, but only the high-end systems provide modeling tools.

2.2.1 The data-flow paradigm

Recent research has moved towards the separation of modeling and rendering, with the two components communicating with each other to produce an image. A pipeline or data-flow paradigm has been suggested by several researchers [POTMESIL 87] [HAEBERLI 88]. Modeling is performed by a set of small tasks, each performing a specific function, with the output of a given task "piped" to the next task in the pipeline for further processing. Both visual [HAEBERLI 88] and textual [POTMESIL 87] implementations have been completed. In both of these systems,

the division between the modeling and rendering stages is unclear, violating one of the guidelines set out by Fiume. Nevertheless, the concept of small tasks performing operations on the scene fits the rest of Fiume's framework for modeling software. This results in a pair of flexible image synthesis systems.

2.2.2 Three-dimensional virtual worlds

A very interesting and exciting approach to three-dimensional modeling is through the use of *three-dimensional glasses*, and an input and control device with many degrees of freedom, such as the VPL DataGlove [SIGGRAPH29 89]. The DataGlove records and transmits to a host processor, the position, orientation, and finger positions of the human hand, allowing complex grasping and positioning tasks to be performed in three-dimensional space. By viewing and interacting with a scene in three-dimensions, a very powerful and natural editing environment can be created. With this new technology, the problem of interacting with three-dimensional models in two-dimensions can be solved. Although current technology is cumbersome, the movement towards full three-dimensional interactive environments, such as Bolio and View [SIGGRAPH29 89], offers great promise for future development.

2.2.3 Rendering interfaces

Recently, Pixar has proposed a standard for a rendering interface called *RenderMan* [UPSTILL 90] [PIXAR 89]. A rendering interface, if adhered to by the graphics community, will allow any modeling software to work in harmony with any rendering system. The problem of standardization is common in computing science. In the area of computer graphics, it is being tackled by many companies and research institutes, both separately and in conjunction with Pixar. The RenderMan Interface is

> "... a method for describing the content of an image to a computer program in the kind of detail required to reach realistic levels of complexity and quality [UPSTILL 90]."

The RenderMan Interface provides a rich and complete specification for realistic image synthesis, much of which is supported by the rendering system presented in this thesis. Because the main focus of the research presented here is systems for realistic image synthesis, and not rendering interfaces, the philosophy of RenderMan is supported, but not the interface itself. Although RenderMan is very exciting work, its recent release (1989) and the complexity of the interface preclude its inclusion in this work.

2.3 YARI – yet another rendering interface

The rendering interface is an important component of an image synthesis system. It allows for the simple expansion of the system to adapt to both new modeling requirements and rendering technologies. *YARI*, the rendering interface designed for this research, was developed to facilitate the communications between the components of the graphics workbench. It is not as general an interface as RenderMan, but it does provide a clear, well-defined protocol for transferring modeling data. In particular, YARI lacks the control structures, the powerful shading language, and some of the modeling primitives (such as parametric surfaces) that RenderMan provides.

2.3.1 The language definition

This section explains the structure of a YARI scene description, but does not explain the meaning of all the parameters used in the modeling language. A discussion of this topic is beyond the scope of this thesis. For a more in-depth discussion of the

meaning of these parameters, refer to the shading section of a computer graphics text, such as Rogers' *Procedural Elements for Computer Graphics* [ROGERS 85].

To communicate a scene between two components of a graphics workbench, the *scene attributes* and *scene structure* must be described. The scene structure is defined by specifying scene primitives, their location, and their surface properties. The scene attributes describe how objects in the scene will be rendered. For example, light sources, the camera location and focal point, and background light intensities are specified by scene attributes.

YARI consists of two separate sections – a brief preamble followed by the main body. The preamble contains the definitions of the scene attributes, while the main body contains the definitions of the scene structure. The body of a YARI scene description is block structured, with nesting of blocks permissible. All blocks are enclosed in the *world block*, which defines the main body of the description.

Each block has a local geometric transformation that is applied to all objects that the block contains, with the world block's transformation being the identity matrix. As nested blocks are entered, the current graphics transformation is pushed onto the *graphics stack*, and an identical duplicate transformation is defined for the new scope. Any new transformations that are defined in this scope are combined with the current one (with a simple matrix multiplication), to create a new transformation. As primitives are defined, the new transformation is applied. When a block, and therefore the current scope, is left, the transformation on top of the graphics stack is popped, and made the current transformation.

Modeling primitives provided by the language are described in a verbose language, as are the transformations and viewing parameters. After one understands the meaning of the parameters, the language becomes quite easy to understand. Valuable aids, such as naming colors rather than using the primary red, green, and blue triples (RGB

triples), are provided to further simplify the "look" of the language. It is much simpler to understand a statement such as *color=purple* than *color=0.85,0.2,0.85*. The color naming is configurable at run time, and can be changed by the user, allowing dynamic color sets to be created. By providing both methods of specifying a color, ease and clarity of use is enhanced.

The scene attribute section of the model is simply a list of light sources, resolution specifications, camera attributes, and background light intensities. The number of light sources is unlimited by YARI, but most rendering algorithms will enforce a limit. For all other attribute declarations, the last one of a given type is the one that will be in effect for the image. Refer to Table 2.1 for a list of possible options, and how they are used.

The body of the model consists of a single *begin/end* pair, enclosing a list of transformations, primitive definitions, surface property definitions, or nested blocks. The transformation applied to a given object consists of all of the transformations defined in previous scopes, combined with the transformations defined before the primitive definition in the current scope. The primitive and transformation commands are listed in Table 2.2, while the surface properties are listed in Table 2.3.

A small example of a scene described by the scene description language is shown in Figure 2.1. The text defines a recursive tetrahedron image with sixteen primitives and one light source. This image and other similar, but more complex images are used for testing the rendering algorithms described in Chapter 4. The description is quite verbose, but note that it is not clear what the scene is by the textual content. This demonstrates the importance of visual three-dimensional modeling tools. A more complex and detailed scene description, and the BNF syntax for the scene description language, are given Appendix C.

- **resolution(x,y) ;**
  - Declares an image resolution of x pixels by y pixels.
- **background(color = r,g,b) ;**
  - Declares the background color of the image to be the color represented by the triple r,g,b.
- **background(color = name) ;**
  - Declares the background color of the image to be the color defined by the identifier *name*.
- **ambient(color = r,g,b) ;**
  - Declares the ambient light to be the color represented by the triple r,g,b.
- **ambient(color = name) ;**
  - Declares the ambient light to be the color represented by the identifier *name*.
- **light(position = x,y,z ; color = r,g,b) ;**
  - Defines a light source at location x,y,z of the color represented by the triple r,g,b.
- **light(position = x,y,z ; color = name) ;**
  - Defines a light source at location x,y,z of the color represented by the identifier *name*.
- **camera(position = x,y,z ; at = x,y,z ; up = x,y,z ; angle = deg) ;**
  - Defines the camera attributes to be used. The position of the camera is defined by the *position* triple, and the focal point of the camera is defined by the *at* triple. The *up* triple defines a direction vector that should be "up" in the image. *Angle* is the angle of view (in degrees) that the image will encompass.

Table 2.1: YARI commands for a scene description preamble

- **translate(x,y,z) ;**
  - Defines a translation of x,y,z in their respective dimensions and combines it with transformations in this and lower scopes.
- **scale(x,y,z) ;**
  - Defines a scale of x,y,z in their respective dimensions and combines it with transformations in this and lower scopes.
- **rotate(x,y,z) ;**
  - Defines a rotation of x,y,z about their respective axes and combines it with transformations in this and lower scopes. The rotations are specified in degrees.
- **sphere(center = x,y,z ; radius = r) ;**
  - Defines a sphere with a center at x,y,z with a radius of r.
- **polygon(vertices = n ; x1,y1,z1 ... xn,yn,zn) ;**
  - Defines a polygon with n vertices, with the i'th vertex defined by the xi,yi,zi triple.
- **attribute(attribute list) ;**
  - Defines the attributes of all the objects between this and the next attribute declaration or the end of the file. The attribute list is a list of zero or more definable surface attributes. Any or all of the attributes can be left out with a default object being a diffuse, solid, white surface.

Table 2.2: YARI commands for a scene description body

- **color = r,g,b ;**
  - Defines the color of the objects using the red, green, and blue triple *r,g,b.*
- **color = name ;**
  - Defines the color of the objects using the predefined color identifier *name.*
- **kdr = value ;**
  - Defines a *value* for the amount of diffuse reflection that emanates from an object. Values close to zero are highly reflective and values close to one are very diffuse.
- **kdt = value ;**
  - Defines a *value* for the amount of light that passes through an object. Values close to one are highly diffuse, and therefore do not transmit light, and values close to zero are highly reflective, and do transmit light.
- **phong = value ;**
  - Defines a *value* for determining the sharpness of high-lights on shiny objects. The higher the value the sharper the high-light.
- **refract = value ;**
  - The index of refraction of the object. This *value* determines how much the light is "bent" when it passes through an object. Values close to one refract very little, while values farther from one refract a lot.
- **texture = value value,value,value value,value,value,value;**
  - Defines the type of texture to use. These values are defined by the rendering system, and may be used in any manner desired.

Table 2.3: YARI attribute commands

```
/* A recursive tetrahedron with 16 primitives and one light source. */

camera(  position = 1.02285   ,-3.17715  ,-2.17451 ;
         at       = -0.004103 ,-0.004103 ,.216539  ;
         up       = -.816497  ,-.816497  ,.816497  ;
         angle    = 22.5);
resolution(512,512);           /* Image resolution of 512 x 512 */
background(color = UNC_Blue);  /* Color identifier UNC_Blue used */
light( position = 1.87607,-18.1239,-5.00042 ;
       color    = white);      /* White light source */

/*
 * The main body of the scene description. The triangles used to build the
 * recursive tetrahedron are red, course (low phong coefficient),
 * dull (high diffuse reflection) objects.
 */

begin
    attribute(color=red ; kdr=1 ; kdt=1 ; phong=1 ; refract=0);

    polygon(vertices=3 ;   -1 -1  1    -1  0  0    0 -1  0);
    polygon(vertices=3 ;    0  0  1     0 -1  0   -1  0  0);
    polygon(vertices=3 ;    0 -1  0     0  0  1   -1 -1  1);
    polygon(vertices=3 ;   -1  0  0    -1 -1  1    0  0  1);
    polygon(vertices=3 ;   -1  0  0    -1  1 -1    0  0 -1);
    polygon(vertices=3 ;    0  1  0     0  0 -1   -1  1 -1);
    polygon(vertices=3 ;    0  0 -1     0  1  0   -1  0  0);
    polygon(vertices=3 ;   -1  1 -1    -1  0  0    0  1  0);
    polygon(vertices=3 ;    0 -1  0     0  0 -1    1 -1 -1);
    polygon(vertices=3 ;    1  0  0     1 -1 -1    0  0 -1);
    polygon(vertices=3 ;    1 -1 -1     1  0  0    0 -1  0);
    polygon(vertices=3 ;    0  0 -1     0 -1  0    1  0  0);
    polygon(vertices=3 ;    0  0  1     0  1  0    1  0  0);
    polygon(vertices=3 ;    1  1  1     1  0  0    0  1  0);
    polygon(vertices=3 ;    1  0  0     1  1  1    0  0  1);
    polygon(vertices=3 ;    0  1  0     0  0  1    1  1  1);
end
```

Figure 2.1: A scene description of a recursive tetrahedron (16 objects)

2.4 Scene – a three-dimensional model editor

Scene is an interactive three-dimensional model editor, developed by the author, that allows the user to create, combine, and edit both primitive and composite objects, as well as perform operations to modify scene characteristics. It meets all of the requirements for modeling software, as set out by Fiume, and provides a very powerful tool for visualizing three-dimensional models.

The major problem in developing a three-dimensional modeler is the fact that an image of the model must be displayed in two dimensions, while editing operations need to be performed in three-dimensions. The solution used in this work is that of the standard CAD or architectural drawing tools, the orthographic and perspective projections [FOLEY 82].

An orthographic projection is one in which the direction of projection is normal to the viewing plane; there is no fore shortening or distortion when objects are projected into two-dimensional space. This is very useful for modeling, because measurements of distances and angles among parallel lines remains true. The most common orthographic projections are the front (elevation), top (plan), and side (elevation) projections. In each of these, the projection plane is perpendicular to a principal axis, and therefore the principal axis is equivalent to the direction of projection.

A perspective projection does not preserve measurements when projected to two dimensions, but it does provide crucial depth relationships to the observer. In perspective projections, objects that are farther away from the viewer seem smaller than those that are closer. The orthographic projections are used to create, and edit objects in the three-dimensional scene, while the perspective projection is used to visualize what the model looks like in a more realistic manner. The perspective projection of the model can be rendered in three ways, each of which has its own time/realism

tradeoff. The scene can be rendered very quickly as a simple wire frame, or more slowly as a flat shaded image with hidden surfaces removed [FOLEY 82].

2.4.1 Modeler primitives

A Scene editing session consists of the user creating and incrementally building a three-dimensional scene out of the primitive graphic objects provided by the modeler. The editor consists of four graphical displays, three of which provide a top, side, and front view of the scene, while the fourth provides a perspective view of the scene based on a user specifiable viewing location. A control panel allows the user to perform the editing and control operations needed by a modeling program.

Scenes are built by creating composite objects. Each composite object is either a single primitive graphic object, or a set of primitive graphic objects that have been combined by a *grouping* operation. Each primitive graphic object is created by building a *skeleton* of the object and then performing an operation to *instantiate* that primitive. The skeleton is built by creating points in the editing windows, connecting the points together to create line segments, and finally building the skeleton from these line segments. The primitive graphic objects supported by Scene are polygons, spheres, and quadric surfaces. Polygons are created by defining the two-dimensional outline of the polygon with an object skeleton. Polygons can be arbitrarily complex, and do not have to be planar, although it is undefined how the rendering component of the system will render non-planar polygons. Spheres are created by specifying a center and radius. This is done by creating an object skeleton with just two points, the first being the center, and the second a point on the circle. Because quadric surfaces cover a wide variety of surfaces (cones, ellipses, paraboloids, hyperboloids, etc.), it is necessary to enter the formulation of the quadric equation manually. To specify the

position and bounds of a quadric, a bounding box is created and the quadric is then clipped to the bounding planes of the volume. The operations to create composite objects are described in Section 2.4.2.

2.4.2 Grouping operations

Scene has various operations that can be performed, each of which falls into one of two categories: grouping or editing operations. Grouping operations are the fundamental operations that give modeling tools their extensive power. They allow the user to flexibly create arbitrary composite objects from the modeler's primitive graphic objects. The most powerful grouping operation is the *combine* operation, which makes it possible to group existing composite objects to form a more complex composite object. This leads to the creation of object groups, with objects that are logically related at the human perceptual level being related at the scene primitive level as well. To make this process complete, it is also necessary to *explode* a composite object into its various components. This makes it possible to split up and recombine the composite objects into new object groups.

Other powerful grouping operations, although not fundamental to the design of the modeler, provide very useful methods of constructing complex three-dimensional objects that would be very difficult to produce manually. Two operations are provided: one to create polygonal surfaces of revolution, the other to create three-dimensional extrusions of two-dimensional objects. A surface of revolution can be created by defining a two-dimensional Bézier curve in one of the editing windows, and sweeping it around one of the principal axes. Both the smoothness of the curve and the curve sweep can be controlled, making it possible to create both "blocky" and "smooth" surfaces of revolution.

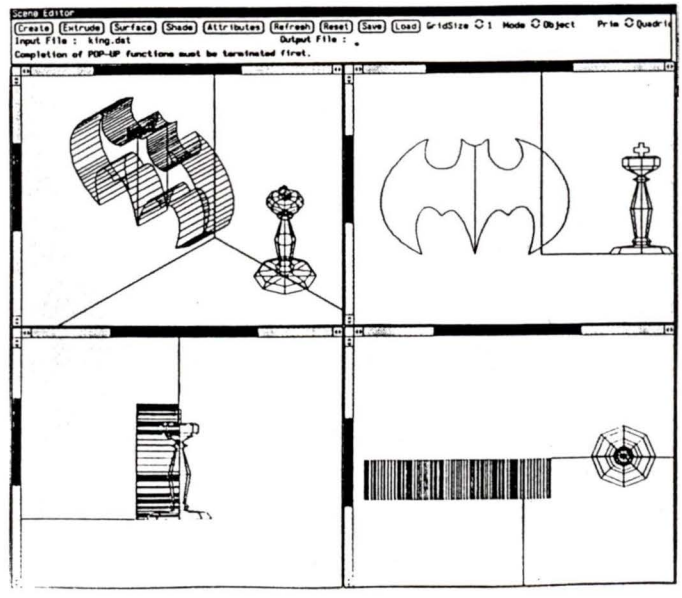


Figure 2.2: Examples of surfaces of revolution and extruded objects

Bézier curves. This skeleton is then extruded or pulled along one or more of the principal axes, creating a three-dimensional block. The top and bottom surfaces of the extruded object are the same shape as the skeleton, with each corresponding line segment of the top and bottom surfaces being joined by a polygon.

Figure 2.2 shows some examples of these objects. The "bat" object was created by defining a two-dimensional bat-like shape, and extruding that shape along the z-axis. This object was used in the rendering of Figure 2.3. The "king" chess piece of Figure 2.2 was created by defining a two-dimensional curve and sweeping it around the y-axis. An extruded cross was then added to the object to complete the king's crown. The rotational sweep was performed in forty-five degree increments. A complex chess scene that uses the *combine*, *sweep*, and *extrude* operations can be seen in Figure 2.4. The rendered scene is shown in Figure 2.5. This model took approximately six hours to construct. To perform such a task without modeling tools such as Scene would have been a very time-consuming and tedious task.

Figure 2.3: The bat model after rendering

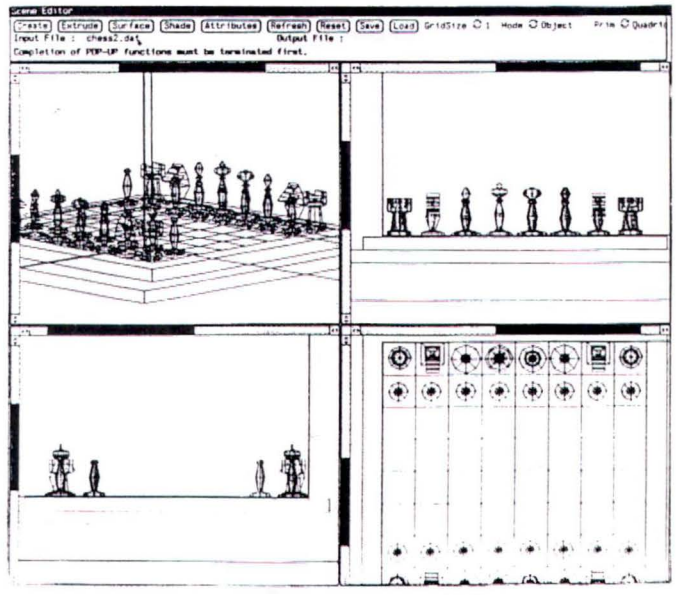


Figure 2.4: A chess board designed with Scene

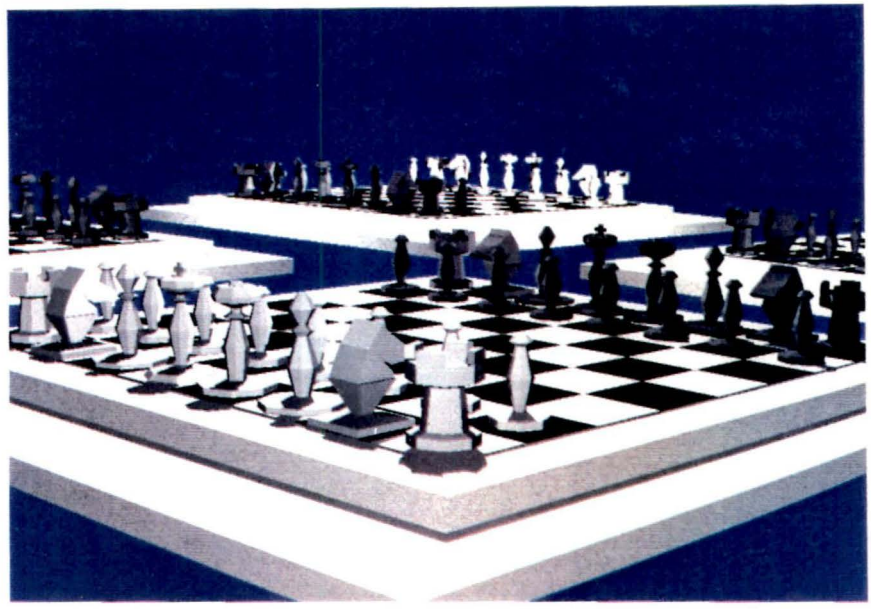


Figure 2.5: The chessboard model after rendering

degree increments. A complex chess scene that uses the *combine*, *sweep*, and *extrude* operations is shown in Figure 2.4. The rendered image is shown in Figure 2.5. This model took approximately six hours to construct. To perform such a task without modeling tools such as Scene would have been a very time-consuming, tedious, and error prone task.

2.4.3 Editing operations

Editing operations can be applied in many ways. Operations are provided for editing the Bézier curves and line segments that are used to create object skeletons. These operations allow the user to add and move the vertices of line segments, as well as moving the control points of Bézier curves. The *create* operation is used to instantiate a primitive graphic object from the current object skeleton. The type of primitive graphic object created depends on the type of object that has been selected for creation.

Operations on composite objects fall into two categories: geometric and surface operations. Geometric operations such as three-dimensional translation, rotation, and scaling can be applied to any composite object. This allows the user to change a composite object's size, shape, location, and orientation. Surface operations allow the user to edit an object's surface properties. These operations are what usually differentiate between a modeler that can be used for realistic image synthesis, and one that cannot. Composite object surface properties such as color, surface type, and surface texture can be changed.

Operations to change the global scene characteristics are also provided by the modeler. The *ambient* or *background* light intensity, the location and color of the light sources, and the viewpoint and viewing direction are all important scene attributes. Operations to edit these attributes are essential in providing a complete modeling system.

Because Scene is the modeling component of the graphics system implemented for this work, it adheres to the rendering interface defined by the YARI scene description language. Scene also provides a rich set of operations (for a brief summary, see Table 2.4), and several types of primitive objects, that enables a user to incrementally build a complex three-dimensional scene. Several rendering techniques, such as simple shading and hidden surface algorithms, provide a cursory means of visualizing the scene, before it is rendered by more powerful techniques. This provides an efficient feedback mechanism, assisting in the fast incremental creation of the scene. To add a new primitive graphic object to the modeler, the semantics of operations on that object must be described and implemented. By adhering to Fiume's modeler framework, a powerful and flexible modeling tool has been designed and implemented.

- Skeleton operations
  - Line Segments: Moving and adding vertices.
  - Bézier Curves: Moving control points and editing curve smoothness.
- Global object operations
  - Camera: Edit the camera location.
  - View Location: Edit the view location.
  - Light Source: Create a light, edit its location and color.
  - Background: Edit the background color and the ambient intensity.
- Composite graphic object operations
  - Transformations: Scale, rotate, and translate composite graphic objects.
  - Surface Properties: Edit the color, surface type, and surface textures of composite graphic objects.
- Grouping operations
  - Create: Create a primitive.
  - Combine: Add primitive graphic objects to a composite graphic object.
  - Explode: Decompose a composite graphic object into its components.
  - Revolve: Create a polygonal surface of revolution.
  - Extrude: Create an extruded polygonal object.

Table 2.4: A summary of modeler operations

2.5 Summary

The modeling tools and rendering interface discussed here provide a flexible and powerful means of creating, editing, and communicating a three-dimensional model to any rendering system that adheres to the rendering interface. It is clear that these are necessary components of a three-dimensional computer graphics system. Without an easy visual way to model a three-dimensional scene, it is nearly impossible to produce images of any significant complexity. Generally speaking, limited modeling support is one of the major bottlenecks in the creation of realistic computer images. As modeling systems move towards true three-dimensional editing (through the use of new technologies such as the DataGlove and three-dimensional glasses) and the environment we work in approaches reality, the modeling task will become much simpler. In many ways, computer graphics is as much an art as it is a science, and without adequate modeling tools, the vast potential of computer generated photorealistic images will remain unfulfilled.

Chapter 3

Ray tracing

Ray tracing is an elegant and powerful rendering technique. By simulating the optical properties of light, and how it is gathered by a camera, images with optical effects such as *reflection*, *refraction*, and *shadows* are easily produced. Other rendering techniques, such as the standard hidden-surface algorithms [SUTHERLAND 77] and *radiosity* [IMMEL 86], by themselves, do not provide these important image characteristics.

In the standard hidden-surface techniques, texture maps known as reflection maps are used to simulate reflection and refraction. That is, if a surface being rendered is reflective, an image of what the reflected scene should look like is mapped onto the surface, producing the illusion that the surface is reflective. This requires human intervention to both produce and place the reflection map, and is not inherent in the rendering model itself. This is a major drawback of these rendering techniques.

Radiosity, on the other hand, was introduced to solve one of the problems that other rendering techniques, including ray tracing, do not solve. *Diffuse interreflection* or *color bleeding* is the phenomenon observed when a colored object (not a light source) provides colored illumination to nearby objects. Consider a red cube in a

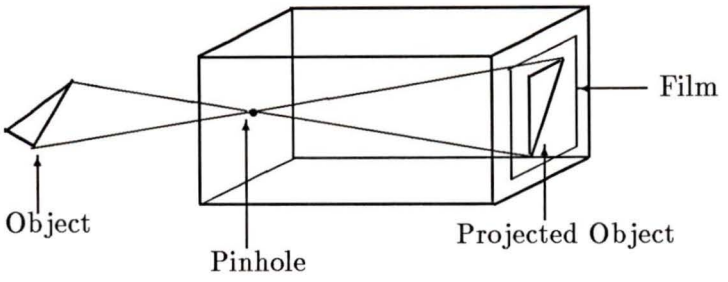


Figure 3.1: The pinhole camera

room with a white floor and white walls. In this example, the parts of the room that are near the cube are slightly affected by the color of the cube, and will have a slightly reddish hue. Ray tracing does not directly solve this problem, although extensions to the ray tracing model have been suggested to address it [WARD 88]. Radiosity models the principal of radiative heat transfer among surfaces, and solves the diffuse interreflection problem. Unfortunately, it does not handle reflection, refraction, or even shadows elegantly. The most promising solutions to this problem are the extended ray tracing model [WARD 88] and a hybrid radiosity and ray tracing algorithm [WALLACE 89].

3.1 The ray tracing model

Ray tracing, in its simplest form, simulates the light gathering of a simple pinhole camera, as illustrated in Figure 3.1. In computer graphics, a rendered image consists of a two-dimensional array of picture elements or *pixels* (see Figure 3.2). A pixel is the smallest component that is visible on a display, and each must be mapped to a single color to create an image. The image is broken up into *scanlines*, where a scan line is a single row of pixels. The *image plane* is the plane that the two-dimensional image

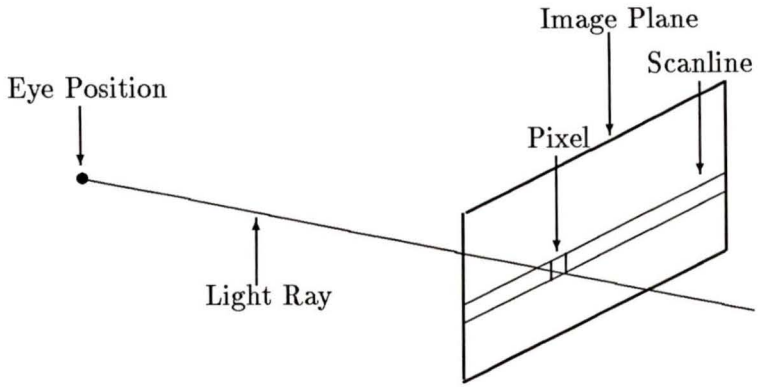


Figure 3.2: The modified pinhole camera

will be mapped onto, and corresponds to the film in the camera. Because a pixel is the smallest observable element of the image plane, the rendering process reduces to determining the amount and color of the light that passes through the pinhole and strikes each pixel. In computer graphics, the pinhole camera is usually modified so that the film is moved in front of the pinhole (see Figure 3.2). For the remainder of this work, the film will be referred to as the image plane or display, and the pinhole as the eye or camera position.

3.1.1 The Hall shading model

To determine the color and intensity of the light that reaches a pixel, the path of a light ray or *eye ray* (Ray E in Figure 3.3) is traced from the camera, through the pixel, into the three-dimensional scene. This ray is checked to determine if it intersects or hits any objects in the scene. If it does not intersect an object, a default background color is used to determine the color of the ray. If an intersection does occur, the object whose intersection is closest to the camera is used (Object O1 in

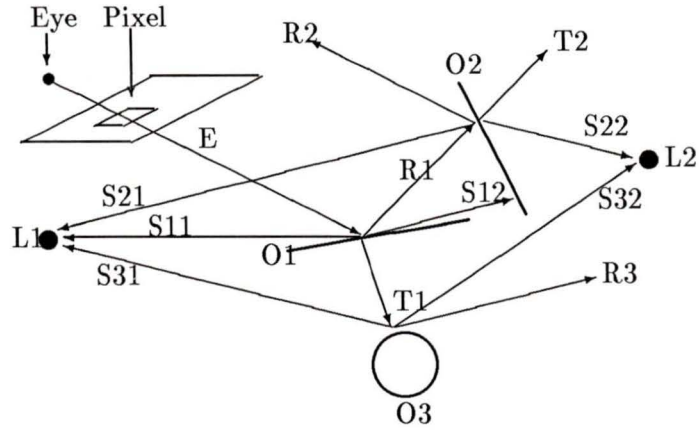


Figure 3.3: An example recursive shading of an eye ray

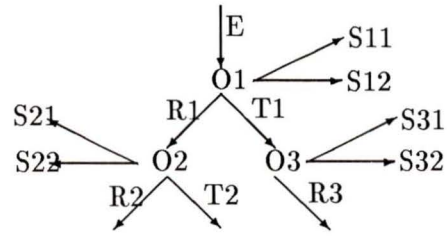


Figure 3.4: The corresponding ray tree for the shading example

Figure 3.3), since it will be the object that contributes to the color of the ray, and hence to the pixel.

Each object in a scene has several properties, such as color and surface texture, that determines how light acts when it hits that object. The amount of illumination that hits the intersection point of an object comes from two areas, light sources and other surfaces in the scene. Illumination from light sources (L1 and L2 in Figure 3.3) is computed by tracing what are commonly called *shadow rays*, from the point of intersection on the object to each light source in the scene (Rays S11 through S32 in Figure 3.3). If the shadow ray hits another object between the object and the light source (Ray S12 in Figure 3.3), then the object intersection point is in shadow from that light source. If no objects interfere with the shadow ray (Ray S11 in Figure 3.3) the light source illuminates the intersection point. The light source's contribution to the color of the surface is then computed, based on the surface properties of the object, the color of the light source, and the angle of incidence of the shadow ray.

The illumination that reaches the intersection point from other bodies is calculated by determining what light from other bodies is reflected in, or transmitted through, the object. As mentioned earlier, diffuse interreflection of other bodies is not covered by this model. To determine what objects contribute to the color of the ray, the directions, based on object geometry, of the reflected and transmitted rays are computed (Rays R1 and T1 in Figure 3.3). The shading model is then recursively applied to these *secondary rays*, creating a *ray tree* (see Figure 3.4). After the reflected and transmitted ray colors are computed, they are combined with the other color components to determine the final color of the ray. In Figure 3.3, for example, the reflection ray R1, that is spawned at the reflective object O1, intersects object O2. The color of object O2 contributes to the color of object O1, because its reflection can be seen in the surface of object O1. The color of object O2 is determined in the same recursive

k_{dr}, k_{dt}	the diffuse reflectance and transmittance coefficients.
k_{sr}, k_{st}	the specular reflectance and transmittance coefficients.
k_a	the ambient reflection coefficient.
$I_{sr}(\lambda), I_{st}(\lambda)$	the spectrum of the reflected and transmitted rays.
$I_a(\lambda)$	the spectrum of the ambient light.
$I_j(\lambda)$	the spectrum of light source j.
$F_{dr}(\lambda), F_{dt}(\lambda)$	the diffuse reflection and transmission spectrums at wavelength λ.
$F_{sr}(\lambda, \theta), F_{st}(\lambda, \theta)$	the specular reflection and transmission spectrums at wavelength λ and angle θ.
$F_a(\lambda)$	the ambient reflection spectrum at wavelength λ.
θ_r, θ_t	the angle between the normal vector and the reflected and transmitted rays.
T_r, T_t	transmissivity per unit length of the medium containing the reflected and transmitted rays.
$\Delta sr, \Delta st$	the distance traveled by the reflected and transmitted rays.
n_r, n_t	specular reflection and transmission highlight coefficients.
$\mathbf{N}, \mathbf{N}'$	the surface normal in the reflection and and transmission directions.
$\mathbf{L}_j$	the vector towards light source j.
$\mathbf{H}_j, \mathbf{H}'_j$	the perfect specular reflection and transmission vectors for light source j.
and	$k_{dr} + k_{sr} = 1, k_{dt} + k_{st} = 1,$ $0 \leq k_{dr}, k_{sr}, k_{dt}, k_{st} \leq 1, 0 \leq T_r, T_t \leq 1$

Table 3.1: Variable descriptions for the Hall shading model

	Light sources	Other bodies
Specular reflection	$k_{sr} \sum_j I_j(\lambda) F_{sr}(\lambda, \theta_r)(\mathbf{N} \cdot \mathbf{H}_j)^{n_r}$	$k_{sr} I_{sr}(\lambda) F_{sr}(\lambda, \theta_r) T_r^{\Delta sr}$
Specular transmission	$k_{st} \sum_j I_j(\lambda) F_{st}(\lambda, \theta_t)(\mathbf{N} \cdot \mathbf{H}'_j)^{n_t}$	$k_{st} I_{st}(\lambda) F_{st}(\lambda, \theta_t) T_t^{\Delta st}$
Diffuse reflection	$k_{dr} \sum_j I_j(\lambda) F_{dr}(\lambda)(\mathbf{N} \cdot \mathbf{L}_j)$	$k_a I_a(\lambda) F_a(\lambda)$
Diffuse transmission	$k_{dt} \sum_j I_j(\lambda) F_{dt}(\lambda)(\mathbf{N}' \cdot \mathbf{L}_j)$	

Table 3.2: The components of the Hall shading model

manner. Note that object O3 is not transparent, and a secondary transparency ray is not spawned at this point in the ray tree. Because this computation is recursive, a termination criterion must be determined. Usually, the recursion is stopped when a ray leaves the scene, a predetermined number of recursive steps have been taken, or the contribution of the ray becomes negligible. These components are the basis for the *Hall shading model* [HALL 89]. For a detailed formulation of the mathematical basis of the Hall shading model, refer to Table 3.2, Table 3.1, [HALL 89], and [GLASSNER 89B].

3.1.2 Rendering model improvements

Ray tracing, in its simplest form, models many important optical phenomena. Through the use of *stochastic* or *distributed ray tracing* [COOK 84] other important phenomena, such as motion blur, depth of field, penumbral shadows, and distributed light sources can be modeled. Stochastic ray tracing is the process of distributing randomly, over an area, a set of rays to trace. Since computer graphics is essentially a sampling process (a continuous image is being discretely sampled), *aliasing* can occur. Aliasing is caused by attempting to display frequencies that are too high for discrete samples (taken at each pixel) to reproduce. This results in disturbing visual artifacts, such as the "jaggies" on the edges of objects, strobing or "popping" of objects in

animation, and Moiré patterns in textures [HECKBERT 86].

For example, when an object is rendered, rays are intersected with that object. At the edge of the object, two rays will be traced through adjacent pixels, one that intersects the object, and one that does not. When this occurs for a distance, a staircase of pixels is created along the edge of the object. These "jaggies" detract significantly from the realism of the image. To alleviate this problem, several rays can be randomly distributed over the pixel area. When these rays are traced, and each provides a color (possibly different) for the pixel, they are filtered to produce a single color. This filtering can occur in many ways, and a full discussion is beyond the scope of this thesis. A simple filtering technique is to use a *box filter*, which averages the colors to produce a single result. Sampling randomly over an area in this manner reduces aliasing, and replaces it with noise, a less disturbing artifact to the human visual system.

Many of the physical processes simulated by ray tracing involve the evaluation of integrals. When a photograph is taken, the shutter is open for a fixed length of time, and the camera lens has a focal point. Ray tracing ignores these and other complex factors (such as blurry reflections and penumbral shadows) by assuming instantaneous shutter speeds, the pinhole lens model, and mirror reflections. By using stochastic sampling and Monte Carlo evaluation of the integrals, many of these complex phenomena can be modeled.

- **Motion blur**

  Distributing rays over time simulates a non-instantaneous shutter speed, causing fast moving objects to blur.

- **Depth of field**

Distributing ray origins over the area of the camera lens simulates depth of field.

- **Blurry reflection**

  Distributing ray directions over the reflection angle simulates non-perfect reflections (gloss or blurry reflections).

- **Translucency**

  Distributing ray directions over the transmission angle simulates non-perfect transmitting bodies (translucency or blurry transparency).

- **Penumbral shadows**

  Distributing shadow rays over the area of a distributed light source simulates penumbral shadows.

3.2 Efficiency considerations and algorithms

Ray tracing is one of the most realistic methods of image synthesis available to date. By applying a global illumination model and the laws of physics to a scene description, it produces realistic optical effects. These effects are difficult or impossible to produce using techniques based on a local illumination model. Radiosity is the only other method of rendering known to the author that is based on a global illumination model. Since image quality is one of the most important goals of a graphics system, this method offers vast potential for advancements in image rendering systems. Unfortunately, the implementation of the algorithm is computationally very expensive, causing it to be rejected by many installations outside the graphics research community in preference of less realistic, but faster rendering methods (such as the Z-Buffer

and the Warnock Algorithm [FOLEY 82]). Ray tracing is computationally expensive because of the large number of floating point calculations required to compute intersections between light rays and objects in the scene. In *exhaustive* ray tracing, every ray must be tested for intersection with every object in the scene, and it is not uncommon for complex scenes to take hours, or even days, to be rendered. It has been stated that from 75 to over 95 percent of the time involved in rendering a scene with an exhaustive ray tracer is spent calculating the intersection points of primitives and rays [WHITTED 80]. Obviously, this is an area where it is possible to make significant improvements, and is, in fact, where much of the current research in ray tracing is focused.

There are two basic ways to increase the efficiency of ray tracing with regards to the calculation of intersections: reducing their cost and number. The most common method of reducing the cost of performing an intersection with a complex object is to surround the object with a computationally simple *bounding volume*, such as a sphere or parallelepiped [RUBIN 80]. If a ray does not intersect the bounding volume used, then it does not intersect the object itself, and the complex calculation can be avoided. This technique can provide significant savings if the objects are very complex, such as fractal objects or parametrically defined surfaces, although it does not directly reduce the number of ray/object intersections that are performed.

3.2.1 Hierarchical bounding volumes

To reduce the number of ray/object intersection calculations, a hierarchy or tree of bounding volumes can be used [RUBIN 80] to obtain an O(log n) theoretical bound on the number of objects intersected for each ray, rather than the linear (O(n)) bound of exhaustive ray tracing. A hierarchy is constructed by grouping a set of bounding

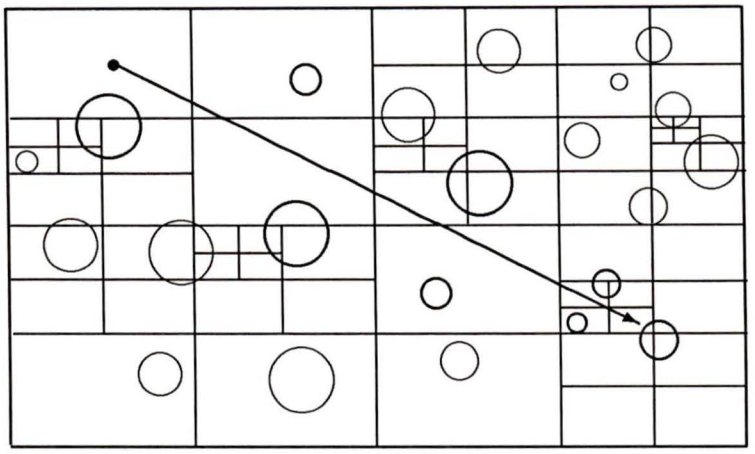


Figure 3.5: A two dimensional voxel subdivision example

volumes and enclosing them in a "parent" bounding volume. This is repeated recursively on the set of existing unenclosed bounding volumes until only a root or *world* volume, which encloses all objects in the scene, remains. How the volumes are grouped is an open research problem, although heuristics have been developed to generate hierarchies and determine their effectiveness [GOLDSMITH 87]. To test a ray against the hierarchy, the ray is first tested against the bounding volume of the root node. If an intersection occurs, each of its children are recursively checked for intersection. If no intersection occurs at a given node, none of that node's children can contain an intersection, and therefore all branches that originate at that node may be trimmed, and need not be considered further. This significantly reduces the number of objects that need to be tested for each ray.

3.2.2 Spatial subdivision

Instead of applying a divide-and-conquer approach to the object structure in three-dimensional space, as is done with hierarchical bounding volumes, spatial subdivision

applies the divide-and-conquer algorithm to three-dimensional space itself. The object space of the scene is subdivided into volume elements or *voxels*, with the subdivision providing information about which ray/object intersections to perform. The *world voxel* contains all objects in the scene, and is the base case for the subdivision. The world voxel is recursively subdivided, with all objects that intersect the volume of a voxel belonging to that voxel. The subdivision continues until there are less than some small number of objects in each voxel or some maximum subdivision level is reached.

Using this technique, rays are traced, starting in the voxel that the ray originates in, through those voxels that the ray intersects. As a ray intersects a voxel, all objects in that voxel are tested for intersection with the ray. If an intersection occurs, the object intersection that is the closest to the ray origin is used, and processing for the ray is halted. If no intersection occurs, the process is repeated with the next voxel along the ray's path, until the ray leaves the world voxel. This technique tests objects in roughly the order they appear along a ray (objects in a voxel are unordered), providing an automatic depth ordering on the objects for every ray. Because they only test objects that are in voxels that the ray intersects, spatial subdivision algorithms only test objects that are "close" to the ray. In Figure 3.5, a two-dimensional analog of three-dimensional spatial subdivision is shown, with an example ray being traced through several voxels. The objects that ray/object intersections must be performed for are shown in darker print. Notice how only those objects that are close to the ray are checked, and that the objects are intersected as the voxels are traversed from the source of the ray until an intersection is found.

Uniform space subdivision [FUJIMOTO 86] [CLEARY 88] uses voxels of uniform size, arranged in a three-dimensional grid. This provides a very fast incremental voxel traversal for following a ray's path, at a cost of having a grid that does not necessarily

fit the topology of the objects in the scene.

Non-uniform space subdivision [GLASSNER 84] adaptively subdivides space, with areas of the scene that have large numbers of objects being subdivided heavily, while other areas are not. At the cost of a more complex voxel traversal algorithm, fewer ray/object intersections are performed. The subdivision in Figure 3.5 is an adaptive spatial subdivision, with only one object remaining in each voxel.

3.2.3 Directional techniques

The most recent algorithmic acceleration technique to emerge is that of directional techniques [GLASSNER 89B]. In spatial subdivision, ray direction is used to select a set of voxels to intersect for each ray as it is being traced, eliminating most of the voxels from further consideration. Directional techniques store this directional information in their data structures, moving the direction computation to a pre-processing stage, where it is less costly. Unfortunately, these techniques require very large amounts of memory. Currently, three directional methods exist, the *Light Buffer* [HAINES 86], *Ray Coherence* [GLASSNER 89B], and *Ray Classification* [ARVO 87]. These algorithms are quite complex, and are beyond the scope of this thesis. The reader is referred to the excellent readings cited above for more information on these techniques.

All of the acceleration techniques discussed here result in significant reductions in rendering time, with speedups of an order of magnitude common for large scenes. No single algorithm is superior, and each will perform better on certain scenes than the other algorithms. This is due to the different algorithmic techniques that they are based upon [GLASSNER 89B]. In this work, a non-uniform spatial subdivision is used, and the other techniques are therefore not discussed further. For more reading

on these techniques, an excellent overview is provided in *An Introduction to Ray Tracing* [GLASSNER 89B].

3.3 Parallel ray tracing

Parallel algorithms for computer graphics are currently a very important area of research. There are regular conference sessions and papers presented on specialized and parallel architectures for computer graphics at the annual ACM SIGGRAPH conference. Ray tracing is no exception, and there have been several significant papers published in the recent past on this topic [SIGGRAPH16 89] [POTMESIL 89].

Ray tracing is inherently parallel in nature, and is well suited for implementation on a parallel architecture. Because each ray cast in a scene is independent of all other rays (light rays do not interact with each other), parallel architectures can be used to direct more computing power to solve the problem. Many researchers have taken advantage of this fact, and have designed algorithms for specialized architectures, such as the *Connection Machine* [HILLIS 87], the *LINKS–1* system [GLASSNER 89B] [SIGGRAPH16 89], and the *AT & T Pixel Machine* [POTMESIL 89]. The fact that ray tracing lends itself elegantly to parallelization does not remove the need for better and faster algorithms, but it does provide a means to drastically reduce the rendering time of a given image through brute-force techniques.

Parallel techniques for ray tracing can be partitioned into two basic classes: image and object space algorithms. *Image space* algorithms divide the two-dimensional image plane into sets of pixels, with a processor assigned to trace all rays associated with a given set. Each processor must have access to the entire database while rendering, implying that the database is either duplicated at each node or is available through some sort of distributed virtual memory. *Object space* algorithms divide

three-dimensional object space into subregions, and allocate processing power to each subregion. Each processor computes intersections with the objects that occur in the regions assigned to it, and therefore only a small part of the database is needed at each processing node. Information about the movement of rays is passed among processors, and thus among regions, by some efficient message passing technique. Both types of ray tracing algorithms have appeared in the literature. The early research in this area concentrated on the simulation of these algorithms on a serial architecture [CLEARY 83] [DIPPE 84] [GAUDET 88]. More recent research has resulted in the implementation of these algorithms on parallel architectures [POTMESIL 89] [GLASSNER 89B] [SIGGRAPH16 89].

Object and image space algorithms can be subdivided further, into adaptive and uniform algorithms. *Uniform algorithms* do not take the complexity of the image or object space into account, and therefore do not balance the load across the processors. *Adaptive algorithms* subdivide the image or object space with respect to the complexity of the image or the scene. In areas that are "complex," an adaptive algorithm will subdivide relatively heavily, while in areas that are "simple," the algorithm will perform little or no subdivision. The definition of "simple" and "complex" depends on the algorithm.

3.3.1 Object space algorithms

Cleary, Wyvill, Birtwistle, and Vatti describe and simulate a uniform object space algorithm that distributes the work to the processors in the system by mapping the subdivided regions of object space onto a two-dimensional or three-dimensional array of processors [CLEARY 83]. Because the object space is subdivided evenly, the actual primitives in any given region may vary from zero to some arbitrarily large number.

This can result in heavy loads on some processors, while others are lightly loaded, resulting in a poor overall system utilization. After an extensive theoretical analysis, the authors conclude that for small numbers of processors, a two-dimensional array of processors is more suitable than a three-dimensional array, producing a speedup factor between $n^{1/2}$ to n (where n is the number of processors being used).

An improvement on this algorithm is described by Dippe and Swensen [DIPPE 84]. They describe an object space that is adaptively subdivided into regions, where each processor is assigned one or more of these regions to process. Their algorithm differs significantly from the one described by Cleary *et al.* in that, as the processing proceeds, the subregions are dynamically resized to reduce the load on heavily loaded processors. This excess load is then distributed to neighboring processors that are (hopefully) lightly loaded.

One of the most important factors of a parallel system is the utilization of the processors. To utilize the processors to their maximum potential, it is necessary to balance the load across all processors. If the load cannot be balanced, it will lead to the under utilization of processors, and will nullify some of the advantages of using a parallel architecture.

Unfortunately, the sophistication and complexity of the load-balancing strategy can be a significant computational factor. This is not necessarily a bad property of the algorithm, as long as it is successful and the overhead is not prohibitive. In this algorithm, the cost of determining the regions to change, calculating their new size and shape (avoiding concave and oblong regions), moving the objects between regions, and the resulting communication overhead between processors, can be significant. Dippe and Swensen state that the processors in their system are well utilized, but it is unclear how much of the processing time is spent on load balancing, and how much is spent on rendering.

This algorithm also performs poorly in a scene that is uniformly complex. Very little is gained by the load balancing in this scenario, but a relatively large amount of overhead is required. This overhead is a result of the calculation of the loads of the processors, and the amount of load balancing that does occur. These factors decrease the actual processor time used for ray tracing, while increasing the overall system utilization. It is clear that this algorithm would perform better than one with no load balancing at all in most cases, but the overhead involved in the balancing of the load can also degrade the performance to below no load balancing levels on some scenes.

3.3.2 Image space algorithms

In their recent paper, Gaudet *et al.* [GAUDET 88] describe a uniform image space algorithm that utilizes a *pipelined engine for ray tracing* (PERT) as a basic building block for a parallel system. They consider two different PERT architectures, a three-processor pipeline and a single processor with three rendering tasks. A parallel machine is then simulated using these architectural building blocks. For each of the n PERT nodes used in the simulation, the $(i)th$ node, $i \in \{1 \ldots n\}$, is assigned to process scanline i, and then leapfrogs its way down the image, processing the $(i+n)th$, $(i+2n)th, \ldots$ scanlines. This algorithm also lacks a load-balancing strategy, and may result in the under utilization of some PERT nodes. The simulation is performed with different amounts of the database in local processor memory, with the parts of the database missing supplied by a broadcast processor over a communications bus. The authors found that in the three processor PERT, the utilization of processors was as low as 66 percent, due to the differing complexity of the three rendering tasks in each PERT and the resulting under utilization of the pipeline. In the one processor PERT configuration, they report good results up to the point that the communica-

tions between processors swamp the communication bus.

A relatively new parallel architecture used for ray tracing in industry is the *AT & T Pixel Machine* [POTMESIL 89]. The Pixel Machine consists of four basic building blocks: a pipeline of pipe nodes, an array of m by n parallel pixel nodes, a pixel funnel, and a video processor. The pipe and pixel processors can be added incrementally, with configurations of 16 to 64 pixel nodes possible. The pipe nodes typically execute distinct sequential algorithms, while the pixel nodes usually execute the same algorithm in parallel on different data sets. Each pixel node can address 16 Kbytes of local memory, with all other requirements being met by a parallel virtual memory provided by the display processor. If a pixel node does not have a desired page of memory, a parallel page fault is generated, and a request is sent to the display processor for that page. The processing for each pixel is distributed to a processor in an interleaved manner based on the pixel's location in the image. This distribution is very similar to the one used by Gaudet *et al.*, except that the distribution is on a pixel by pixel basis, rather than on a scanline basis. Otherwise, the use of the architecture is completely under software control. The Pixel Machine is therefore very flexible, and can implement many rendering algorithms, including ray tracing.

In the ray tracing algorithm implemented for the Pixel Machine, ray trees are distributed to the processors based on the initial pixel ray's location in the image, with the results written to a distributed frame buffer. This algorithm is a uniform image space algorithm, and performs no load balancing of the processors. Each processor, through the use of the parallel virtual memory, has all data needed to perform its calculations, although a significant communication cost may be involved. The pipeline is used to compute bounding volumes, and to perform transformations on the primitives before rendering begins. The parallel pixel nodes are used to perform the parallel ray/object intersections.

When the database size is small enough to fit into the local memory of each node, a linear speedup in the algorithm is observed as the number of processors increase. When the database size is increased, and the number of pixel nodes are kept constant, the performance degrades exponentially as paging begins (five thousand objects), but becomes linear again when the number of objects nears ten thousand. The speed of the algorithm is approximately five times slower with paging than it is without. This second test does not specify how many pixel nodes are used in the rendering, and it is therefore impossible to tell whether a communications bottleneck would be a problem as more nodes are added. If the timings were done using the maximum of 64 pixel nodes, then it is not a problem, since they were done with the maximum amount of communication overhead possible. Regardless of this small oversight, the Pixel Machine (and the ray tracing algorithm used) demonstrates a fast and efficient parallel system for realistic image synthesis.

Green and Paddon [GREEN 89A] [GREEN 89B] have recently described an adaptive image space algorithm that uses a data migration scheme among the processors to allow each access to the entire database. They introduce *data coherence*, a form of coherence or logical connection, that helps to determine which objects are heavily intersected by a given set of rays. By performing a miniature rendering of the scene, statistics can be gathered about object and data structure coherence. It can therefore be determined, to a fairly high degree of accuracy, what data should be paired with each pixel set, and therefore each processor. This leads to a minimal amount of data migration as the rendering occurs, and a near linear speedup in the number of processors used is produced. Using 22 processing nodes, speed up factors in the range of 19.15 to 21.16 were observed, with the results straying farther from perfect linearity as the number of processors increased.

Of the two types, image space algorithms have been receiving far more research

than their object space counterparts. An acceptable way of mapping object space onto an arbitrary configuration of processors has not yet been found. This leads to good special case simulation results, but seriously restricts the flexibility and configuration of the architecture being used. Until a satisfactory mapping of processors to work can be found, image space algorithms will continue to be the favored algorithm for parallel ray tracing.

3.4 Summary

Ray tracing is a very powerful image synthesis technique, providing sophisticated optical effects that are difficult or impossible to produce with other rendering techniques. These optical effects are a direct result of the realistic model used to simulate how light behaves as it interacts with a three-dimensional scene. Most other rendering techniques use subjective or analytical methods to produce images of similar quality, detracting from their ease of use, and often hampering the realism and quality of the images.

The global illumination model used by ray tracing leads to a large number of floating-point calculations, resulting in a very CPU intensive and slow rendering process. For this reason, both algorithmic and architectural techniques have been used to improve the efficiency and rendering speed of ray tracing. Divide-and-conquer algorithms, as well as directional techniques, have been applied to ray tracing, with no clear winner in speed and efficiency. Each ray tracing acceleration algorithm works better than its counterparts on some scenes. This is due to the fundamentally different techniques they use to attack the problem. It is not yet clear whether one will emerge as the superior algorithm. Many parallel architectures and algorithms have also been proposed and tested, both through simulation and implementation. It is

again unclear which method is superior. Image space algorithms are simpler than their object space counterparts, most noticeably when load balancing is performed. Unfortunately, due to the fact that the entire database must be available at each processing node while rendering takes place, the implementation of a complex and often slow distributed virtual memory may be necessary. Object space algorithms only need the local part of the database for rendering, but are often very difficult to load balance for arbitrary processor configurations.

To provide a fast and efficient ray tracing system, both algorithmic and architectural means should be used to reduce rendering times. Because there is no clearly superior algorithm in either of these areas, the system should remain very flexible, making it possible to incorporate advances in both new algorithms and faster architectures.

Chapter 4

A distributed rendering system

A serious problem to many individuals, both in and out of the graphics research community, is the fact that most of the previously discussed parallel systems require a high degree of specialization (usually a specific communication requirement) from their hardware. Most computer installations do not have access to these architectures, nor do they have the resources or the need to acquire one for ray tracing alone. Because of this fact, the parallel nature of ray tracing, and the accompanying rendering time reductions, cannot be utilized by many users. To bring the advantages of parallel ray tracing to more than just a few specialized installations, distributed computer systems can be used.

With the advances in technology that are constantly occurring today, distributed systems are becoming an attractive and promising alternative to large multiprogrammed uniprocessors. With the current decrease in cost and increase in power of VLSI chips, as well as the advances in communication technology, it is becoming easier and cheaper to invest in a network of relatively small, yet powerful machines. Because of these changes in technology, local area network based computer systems

are becoming more widespread, providing an efficient means of accessing a large number of computing resources at minimal cost.

Recently, both distributed systems and computer graphics have been researched extensively. As distributed systems become widely available, distributed windowing systems, and applications using them, will become more popular. It is clear that an amalgamation of the two technologies will offer some attractive prospects for the future, although there is much more research needed in both of these areas.

4.1 Distributed systems

Designing and implementing a complete distributed system is a non-trivial enterprise, requiring a significant investment of resources and time. Because this research is concentrating on the parallelization of ray tracing, and not distributed systems, two simplifying assumptions are made. The first and most important assumption is that only the ray tracing system makes use of the distributed nature of the local area network. In a general distributed system, it is desirable to have all processing done in a distributed manner, making use of all available resources at all times. As one might expect, designing a general distributed system is much more difficult than designing one for a specialized purpose. It is assumed that all processing being done in the network is executed on a machine by machine basis, with no global or distributed control over processing being performed. It is the responsibility of the distributed ray tracing system to seek out and discover idle processing nodes, and to make use of their resources. It is also assumed that each processing node is a UNIX machine, and therefore has a tested and reliable communication medium available (i.e., UNIX sockets). This simplifies the communications aspects of the implementation, but still allows for the heterogeneous nature of the processing nodes, since many machine ar-

chitectures run the UNIX operating system. With these two simplifying assumptions, it is feasible to implement a prototype distributed system for ray tracing in a short time frame.

4.1.1 Idle processing power in LANs

Ray tracing is CPU intensive, and it therefore takes considerable processing power to produce results in a reasonable amount of time. It has been shown that at any given time in a LAN computer system, a relatively large number of the connected processors are idle or lightly loaded. In larger systems, such as the ANDREW system [NICHOLS 87] at Carnegie-Mellon University, as many as 70 of the available 350 machines were found to be idle during peak operation hours, and over 100 were idle during non-peak hours.

At the University of Victoria, a very similar situation exists, but on a smaller scale. The computer science department's computing resources vary significantly in their accessibility and the processing power they possess. Machines range in size from an IBM 3090, with an attached vector processor, to low-end workstations. Because a decision was made to only use UNIX machines, the IBM 3090 was excluded from the processing environment for this work. Had IBM's version of UNIX (AIX) been available, it would have been possible to include this machine in the processing environment. Despite this restriction, a wide variety of architectures are still available for use, ranging from RISC machines, in the form of a Sun SPARCServer 370 file/compute server and Sun SPARCstation workstations, to low-end Sun and NeXT personal workstations. All of these machines are interconnected by a 10 Megabits/second Ethernet local area network, providing a reliable network environment for implementing a distributed system.

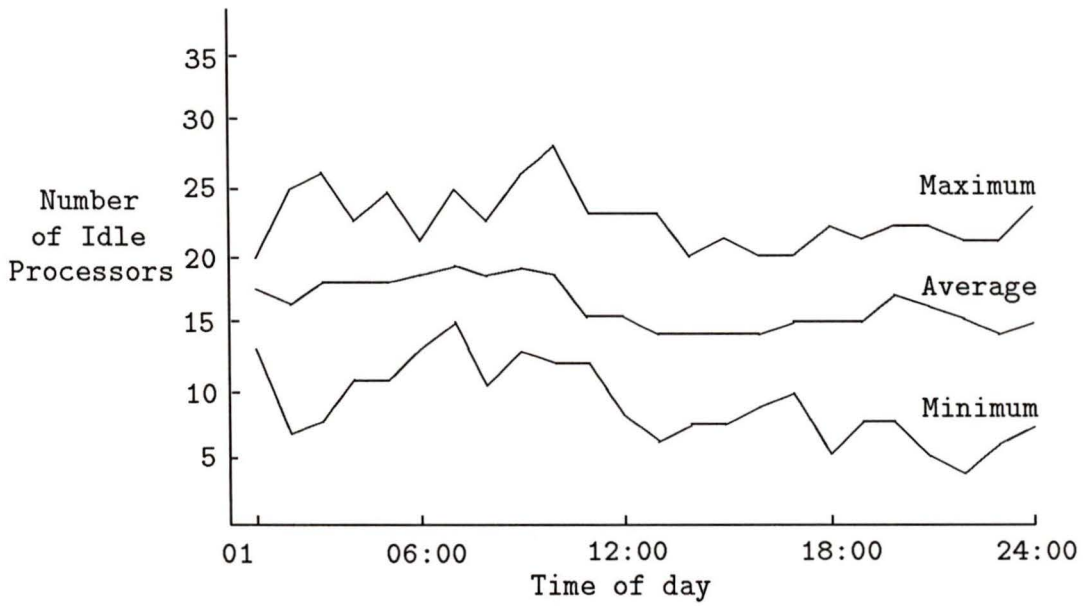


Figure 4.1: Idle workstations at the University of Victoria

In this environment, the workstations are used for various purposes: by faculty members and graduate students as personal workstations, by students in public laboratory areas, or as file/compute servers by the entire department. The student workstations are usually heavily loaded during the day, as are the compute and file servers, while the faculty machines are less so. Thus, many powerful machines are lightly loaded or completely idle for a significant part of the day. This results in a large computational resource that is constantly under utilized.

Figure 4.1 shows a graph displaying the maximum, minimum, and average number of idle processors at various times of the day (a total of 31 processors were tested). In a university environment, the peak processing times are from 08:30 – 17:30, Monday through Friday. This is the time period when university employees, faculty, and students are on campus, and therefore using the machines in the network. Non-peak processing is defined as the remainder of the work day (17:30 – 08:30), as well as the weekends. The measurements cited here were taken in mid-term, when courses were in full session, but not during the part of the term when processing was at its peak. Processing peaks near the end of each term, when students are working on final projects, and is at its lightest during the summer term, when fewer undergraduate students are in attendance at the University. The measurements were taken over a two-week time period, resulting in a reasonable estimate of the idle processing resources available in the local area network.

It should be pointed out that the numbers represented in the graph of Figure 4.1 may be slightly deceiving. One would expect that the processing during the day would be more significantly represented by a decrease in idle machines during this time. Although there is a decrease in the number of idle machines, it is slightly biased by the measurements taken during the weekends from 08:30 – 17:30. This affects the average number of processors that are idle during the peak processing times. For

example, during the 09:00 time slot the average number of idle processors during the week is 18 (58%), while on the weekend there are 21 (68%). At the extreme, as many as 90% of the processors were found to be idle during non-peak processing times, with up to 81% idle during the work day.

4.1.2 Load balancing

An important requirement of a distributed system is that it employ as much of the available processing power as possible. This is usually done by performing some sort of load balancing across all nodes in the system. Many of the low-level algorithms discussed in Chapter 3 ignore this problem all together. Load balancing is even more essential in a distributed system, because the processing nodes are likely to be of vastly varying speeds. This is due to either different machine architectures and clock speeds, or a different load factor produced by other users of the system. Thus, performing an even division of the amount of work will not result in an even distribution of the load to the processors. If a given processor is significantly slower than all other processors in the system, but is still given the same amount of work, it will take longer to finish its computation than the faster nodes. This aspect of distributed systems, and the fact that the loads on the processors change over time, demonstrates the need for a dynamic load-balancing strategy.

Considerable research has been performed in the area of load balancing distributed systems. Eager *et al.* [EAGER 86] show that simple load-balancing strategies perform well when compared with more complex and "smarter" strategies. As demonstrated by Dippe and Swensen's [DIPPE 84] load-balancing algorithm, a complex scheme can have detrimental effects on the system as a whole. A complex load-balancing strategy usually implies a large amount of communication overhead, enabling the various

processors to determine which other processors have a light load and to distribute the work to those lightly loaded nodes. In a specialized hardware system, this factor is important, but a high bandwidth bus can be used to reduce the cost. In a distributed system, this factor is crucial, because of the relatively slow (possibly very slow) communication media that are used. If a simple, yet efficient load-balancing strategy can be devised, a distributed system for ray tracing is a very attractive alternative to the low-level systems generally discussed in the computer graphics literature.

4.1.3 Heterogeneous environments

Another attractive feature of a distributed system is its ability to run on a wide variety of machine architectures. If the system is designed well, it will be able to configure itself dynamically with any number of processing nodes, and should not require processor homogeneity (i.e., they have the same architectural type). In a LAN configuration, this is a critical factor, since the network usually consists of a wide variety of architecturally different processors. All that is required is a reliable communication network among the processors, and the ability to perform ray tracing on each node. If the network does not guarantee an error free transmission, the implementation of the distributed system becomes much more complex.

4.1.4 Fault tolerance

In distributed systems, as well as on parallel architectures, it is desirable to have a high degree of robustness or fault tolerance. That is, if a node in the system becomes faulty, the system as a whole should be able to detect and recover from the fault without error, with the only degradation in performance being solely the loss of a

computational resource. Because the processors are so tightly coupled in parallel systems, many systems choose to ignore this problem.

One of the problems of using a specialized, tightly coupled parallel system, is that they do not adapt well to processing faults. Several of the systems described in Chapter 3 use a very structured communication system, with a given node providing communications to adjacent nodes in the structure [DIPPE 84] [CLEARY 83]. If a single node goes down in such a system, the communication that node provides is lost. In many systems, the processing can no longer continue. In other systems, work is bundled to individual nodes before rendering is started. If a node fails, the processing for that node will not be completed, and the rendering cannot be finished without special actions being taken [GAUDET 88] [POTMESIL 89]

These are undesirable properties in a distributed system, since processors are often loosely coupled. It is a relatively common occurrence for a distributed system's communication medium to fail, due to the long distances and uncontrollable conditions that prevail in these systems. It is also rare for a processor in a distributed system to be dedicated to one task, as they are in most parallel architectures. This leads to a multitude of external factors that can affect the system, and cause failures at any node at any time. Thus, it is important for a distributed system to be able to recover from faults without a resulting loss of information. With a robust distributed system, parallel computations are possible, even with unreliable communication and hardware components.

4.1.5 An abstract machine

A distributed systems is not dedicated to a specific task, so it is important that the system be fair to all of its users. Parallel architectures are typically used as dedicated

systems, with no requirements that other processing be considered at any time. In a distributed system, the goal is to distribute the processing to the computational nodes with the lightest loads, with any number of different jobs being performed at any one time. It is important that all jobs be treated fairly, and have access to equivalent processing resources. This is done by "hiding" the location of the processing being performed from the user, and having it completely under system control. This higher level of abstraction, away from the machine architectures, gives the impression of using a large machine with extensive resources. It does not burden the user with extra detail about which machines are actually performing the computation. The system can then control the resources better, distributing the work to the lightly loaded nodes, and adapting to load changes in the system.

4.1.6 System requirements

To implement a distributed system for ray tracing, a feasible method of breaking the process into atomic elements that are suitable for parallel processing is required. A dynamic load-balancing scheme that will make use of as much computing power as is available in the LAN at run time is also needed. Clearly, the overall system, and the load-balancing strategy in particular, should have some specific goals to attain a fair and evenly balanced distribution of work. Some of these goals are:

- The load of the system should be balanced evenly across all nodes.
- The system should not require node homogeneity, and should be able to run on any node capable of normal computational functions.
- The system should be able to detect and recover from system failures.
- The system should be transparent to the users of the system.

- The system should be fair to all other user tasks, allowing each a fair proportion of the computational resources.

The system presented in the remainder of this chapter allows a ray tracing job to be distributed across n machines, with the ray tracing tasks (processes) communicating through a tested and reliable Inter-Process Communication (IPC) protocol. If no other users are using the system, the rendering time should decrease linearly with respect to the number of processors. Because there will be other users in the system, it will perform slightly worse than the linear goal, depending on the load of the machines involved, and the overhead of the load-balancing strategy.

4.2 Components of the ray tracer

The ray tracing system developed for this research is based on the ray tracing programming guidelines laid out by Paul Heckbert [GLASSNER 89B]. Heckbert describes a well designed framework for exhaustive ray tracing that is elegantly expandable in several important areas. The ray tracer implemented here supports the standard ray tracing model, implementing reflection, refraction, and shadows, as well as the Hall shading model as described in Chapter 3 and in [GLASSNER 89B]. It uses a three color model (red, green, and blue) for light propagation because of its common use in the display technology of modern workstations. The three color model performs relatively poorly in comparison with a more complex spectral analysis, and may lead to spectral aliasing [GLASSNER 89A]. However, it is adequate for this work and the extension to a more sophisticated scheme is straightforward.

Three primitive types are supported by the ray tracer: *spheres*, *polygons*, and *quadric surfaces*. Quadric surfaces are very general modeling primitives, making it

possible to describe cones, spheres, ellipsoids, cylinders, hyperboloids, and paraboloids. It also has the capability to associate arbitrary *clipping planes* with any primitive. Although only three primitive types are supported, combined with the clipping planes, complex objects can be built. Primitive surface properties such as color, surface type, and surface texture can also be specified.

Several extensions to the standard ray tracing model have also been implemented for this work, including *colored light sources*, *stochastic* or *distributed ray tracing* [COOK 84], and *solid texturing* [PERLIN 85] [PEACHEY 85]. Stochastic ray tracing provides a means of rendering various "fuzzy" phenomena, such as motion blur, penumbral shadows, translucence, and blurred reflections. This topic is briefly explained in Chapter 3, and in depth by Cook and Porter [COOK 84].

Solid texturing is the process of mapping texture on objects being rendered by using a three-dimensional procedural texture. In contrast to standard two-dimensional texture mapping [HECKBERT 86], in which a two-dimensional texture is mapped onto a three-dimensional surface, solid texture is procedurally defined throughout three-dimensional space, with the texture for a given point in space being a function of its coordinates. To apply texture to a surface, the coordinates on the surface are computed, and a texture is procedurally defined based on those coordinates. Because two-dimensional texturing is a mapping process, that is, the texture is mapped onto a surface, significant texture distortions can occur, depending on the type of surface being mapped. If a rectangular texture, such as a digitized image, is being mapped onto a spherical surface, a large amount of distortion will occur at the poles of the sphere that are used to define the mapping. Two-dimensional texture mapping is also object specific. For each different surface type to be rendered, a mapping has to be defined to apply the texture to that surface. For complex surfaces, this task can be difficult, or even impossible to perform[PEACHEY 85].

Because solid texturing is defined continuously throughout three-dimensional space, it does not suffer from either of these problems. It therefore provides a powerful and realistic means of applying a complex texture to an arbitrarily complex surface, with no computational penalty above that of applying the same texture to a simple surface. Solid texturing is extremely useful for creating textured objects that appear as if they were carved out of a solid material, such as stone, wood, or clay. Other complex textures such as turbulent clouds, bubbles, and bumps are easily created with solid texturing.

4.2.1 Adaptive spatial subdivision

For any ray tracing system, the choice of acceleration algorithm used is important. The algorithm used here, called *Space Subdivision*, was introduced in Section 3.2.2, and is described in detail by Glassner [GLASSNER 84]. Space subdivision is a non-uniform spatial subdivision algorithm that adaptively subdivides the world voxel into subvoxels until a maximum subdivision level has been reached or there are less than some small number of objects in each leaf voxel. This algorithm reduces the number of ray/object intersections by testing only those objects that are close to the ray. It also tests objects in roughly the order they appear along a ray (objects in a voxel are unordered), providing an automatic depth ordering on the objects in the scene for every ray. Combined, these features significantly reduce the large number of time-consuming ray/object intersections that make exhaustive ray tracing so slow. This reduction is clearly shown in Table 4.1. The statistics for this table were gathered by rendering a recursively defined tetrahedron (Figure 4.2), a standard ray tracing test image [HAINES 87], in increasingly complex forms. This and other test images are used to gather statistics that can be compared to other research done in this area. Both Glassner [GLASSNER 84] and Green and Paddon [GREEN 89A] use the

	Tetra 2	Tetra 3	Tetra 4	Tetra 6
Resolution	512 by 512	512 by 512	512 by 512	512 by 512
Objects	16	64	256	4096
Exhaustive time [sec]	353.28	1224.01	4498.42	76313.1
Rays traced	328,870	321,870	316,929	308,405
Intersections	5,260,480	20,599,680	8,113,3824	1,263,226,880
Subdivided time [sec]	275.53	396.1	470.22	826.51
Rays traced	328,565	321,655	316,720	308,285
Intersections	2,194,819	3,103,569	3,397,565	3,673,856
Number of voxels	24	112	496	7320
Children/voxel	10	10	10	10
Speedup factor	1.28	3.09	9.57	92.33

Table 4.1: Statistics for recursive tetrahedrons of increasing complexity

recursive tetrahedron (Figure 4.2) as a test image for their rendering algorithms. Green and Paddon also use the mountain image depicted in Figure 4.3 for testing purposes.

When the recursive tetrahedron scene is rendered with a small number of objects (i.e., Tetra 2 image in Figure 4.1), space subdivision provides only a moderate speed increase. When the image is rendered with significantly more objects (i.e., Tetra 6 image in Figure 4.1), a vast rendering speed increase is obtained. The speed increase of slightly over 93 times that of exhaustive ray tracing is a direct result of the differing number of ray/object intersections that are performed by the two algorithms. The exhaustive ray tracer performs over 1×10^9 intersection calculations, while the space

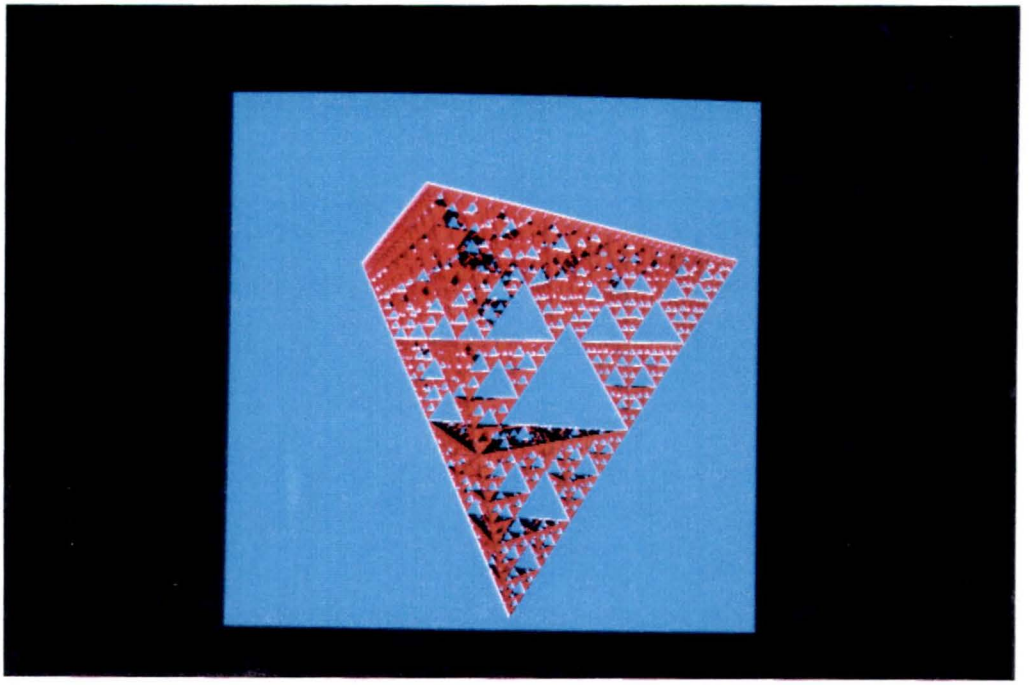


Figure 4.2: The recursive tetrahedron image with 4096 objects

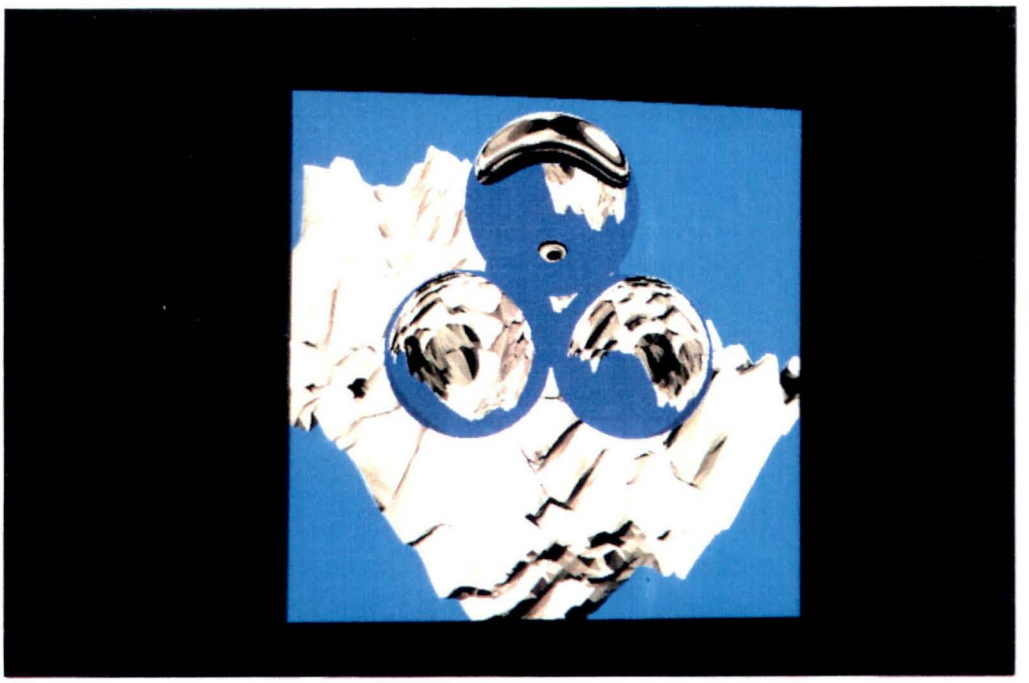


Figure 4.3: The mountain image with 8196 objects

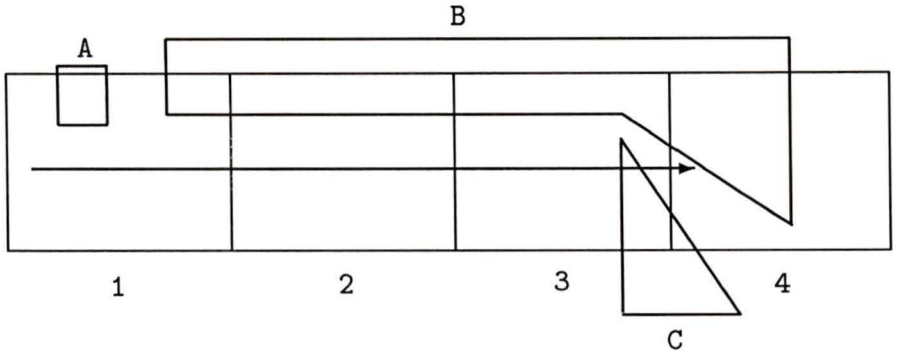


Figure 4.4: An example of multiple intersections

subdivision ray tracer performs just over 3.5×10^6. With the exhaustive ray tracer, the Tetra 6 image was rendered in just over 21 hours. This is considered a simple scene. To perform exhaustive ray tracing on a very complex scene would, literally, take days.

One of the drawbacks of space subdivision is that an object can be intersected more than once with the same ray, thereby increasing, rather than decreasing the number of ray/object intersections performed. As the voxels are traversed for a given ray, only those intersection points that are inside the current voxel are valid. This is necessary to preserve the depth ordering provided by voxel subdivision. If the ray intersects an object, but the intersection lies outside of the current voxel, the intersection is invalid and must be discarded. Later in the traversal, if the ray intersects a voxel that contains the same object, the intersection point needs to be recalculated. For example, in Figure 4.4, the ray is first intersected with object B in voxel one. Because the intersection is outside of this voxel, it is invalid, and the traversal moves to voxel two. If this intersection had been used, an incorrect result would have been obtained, since object C is the correct object for intersection with this ray. In voxel two, the

	Tetrahedron	Mountain	Bat	Furniture
Resolution	512 by 512	512 by 512	512 by 512	512 by 512
Objects	4096	2052	176	486
Rays traced	308,285	939,598	474,195	602,064
Non-buffered time [sec]	809.39	1754.09	837.84	1408.16
Intersections	3,673,856	11,096,642	3,173,581	9,071,577
Buffered time [sec]	826.51	1638.03	706.08	1266.98
Intersections	2,948,740	9,227,157	2,273,459	6,081,026
Buffer accesses	725,116	1,869,485	900,122	2,990,551
Buffer use (%)	24.59	20.26	39.59	49.18
Ratio of buffered to non-buffered times	1.02	0.93	0.85	0.90

Table 4.2: Statistics for multiple intersections of various images

intersection of the ray and object B is needed again. To prevent this recalculation, the intersection for each ray/object pair is buffered, and reused if the intersection is needed later in the voxel traversal.

The frequency of multiple intersections is shown in Table 4.2 for several test images. A decrease in rendering time of 15% is obtained for the bat image using this buffering technique. In both the bat and furniture images, the intersection buffer is used for a large percentage of the calculations (39.59% and 49.18%, respectively). These images result in the most significant rendering speed increases. The tetrahedron and mountain images use the buffer less, and therefore do not produce the speed increase of the other images. Note the slight increase in rendering time for the tetrahedron image. This is a result of the buffering overhead outweighing its advantages. This will only take place if the objects are small, and therefore do not cross voxel boundaries as frequently. The tetrahedron scene is designed to test how rendering algorithms handle an even distribution of small objects throughout the scene, and is therefore an unfavorable test case for buffering. It is used to demonstrate that the buffering technique does have a non-trivial overhead.

Space subdivision was chosen for this distributed system because it is relatively simple to implement, and does not use any form of ray coherence (i.e., no information about a given ray affects how another ray is traced). This causes the rays to be completely independent, and makes it possible to implement an image space algorithm for execution on a parallel architecture.

4.2.2 SunRay I

The initial prototype of the distributed ray tracer was implemented by splitting the image space into n equally sized, adjacent sets of scanlines, where n is the number

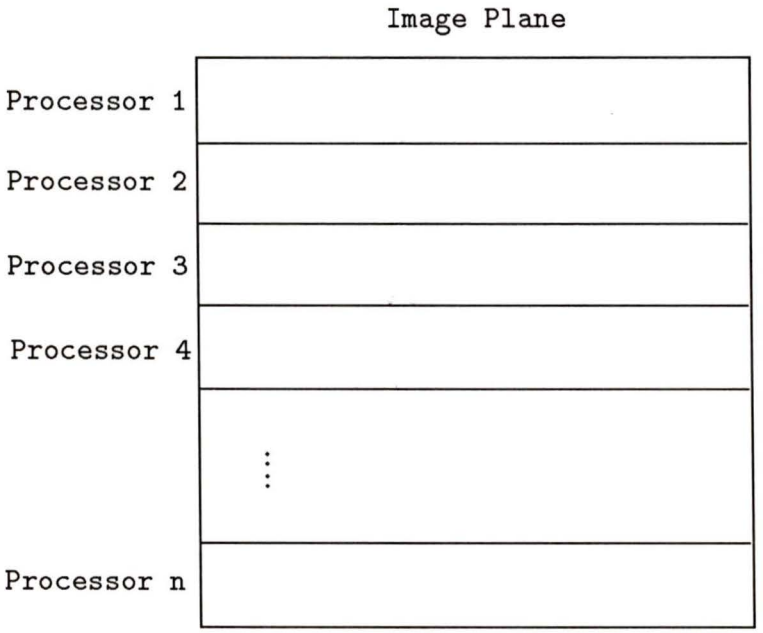


Figure 4.5: Scanline sets for SunRay I

of machines in the LAN available for ray tracing (see Figure 4.5). On each of these n machines, a single process is started, with each process being responsible for tracing one of the sets of scanlines and writing its portion of the image to a its own output file. On startup, each process must read the scene database to be rendered, and perform the voxel subdivision. The processes are started using the UNIX *remote shell* (rsh) command [SUN 88], and the output files are merged in a postprocessing step.

Space subdivision processes rays that pass through empty areas of the scene very quickly. This causes the complexity of rendering an image to vary across image space. A machine that processes a simple part of the image plane (one that contains little image detail) has few objects to check for each ray, and will complete its task in a short period of time. A machine that processes a complex part of the image plane has more objects to test for each ray, and therefore takes longer to complete its assigned processing. The differences can be significant if one area of the image has a large number of highly reflective or refractive objects. This causes more secondary rays to be fired, increasing the depth of the ray trees in that region. Because the image plane is divided into equal-sized scanline sets, each processor receives the same number of ray trees. Therefore, it is possible for one machine to finish much earlier than another. The difference in completion times can be as large as an order of magnitude.

In Table 4.3, the recursive tetrahedron is again used for gathering statistics. This image has a fairly even distribution of objects across the image plane, except near the top and bottom of the image. All machines used for this rendering are of approximately the same processing speed, and yet a significant difference in completion times is observed. Note the smaller number of intersections performed (and the different rendering times) for node 1 and node 8. These nodes processed the top and bottom of the image plane, where there is very little image detail. Also note that because the tetrahedron's projection onto the image plane is triangular, the image gets less

Processing node	Processor load	Secondary rays	Total rays	Intersections	System time [sec]
1	light	0	33,792	5,584	531.32
2	light	4,893	38,685	437,466	929.6
3	light	17,878	51,670	1,329,875	1934.14
4	light	9,692	43,484	751,068	1717.68
5	light	5,226	39,018	451,347	1375.5
6	light	5,476	39,268	375,424	1117.06
7	light	2,376	36,168	266,244	1077.36
8	light	496	26,096	60,055	702.22

Table 4.3: SunRay I statistics for the recursive tetrahedron (4096 objects)

Processing node	Processor load	Secondary rays	Total rays	Intersections	System time [sec]
1	light	23,574	57,366	987,151	2833.4
2	light	41,562	75,354	1,221,909	3281.36
3	light	65,454	99,246	1,296,033	3314.44
4	light	158,465	192,257	1,963,650	4371.76
5	light	154,545	188,337	1,785,771	4110.26
6	light	85,187	118,979	1,032,524	2710.08
7	light	105,308	139,100	1,268,757	3332.26
8	light	44,805	70,405	486,135	2093.94

Table 4.4: SunRay I statistics for the mountain (8196 objects).

complex as it progresses toward the bottom. This is reflected in the decreasing number of intersections performed by the nodes that process the bottom portions of the image plane. The secondary rays that are produced by this image are shadow rays, and are most noticeable in the more complex regions of the image. This increases the processing that is performed by the nodes that render this part of the image (i.e., Nodes 3 and 4 in Table 4.3).

In Table 4.4, the mountain test image is used to show the important effect that secondary rays have on load balancing. The area of the image plane that contains the transparent spheres will generate many more secondary rays than the other areas of the image. Therefore, the processing of that area will take much longer. In Table 4.4, Nodes 4 and 5 have many more secondary rays to process than the other nodes. This significantly increases the rendering time required by these nodes, and decreases the effectiveness of the load balancing. Combined, Nodes 4 and 5 performed 37.34% of the intersection calculations, and processed 40.44% of the rays that were traced. For an evenly balanced load, these numbers should be close to 25% (Nodes 4 and 5, combined, are 25% of the processing power used for this image). Clearly, the load-balancing policy used here is inadequate for most images.

4.2.3 SunRay II

To avoid the sometimes substantial differences in complexity of the different scanline sets, a new scanline distribution policy was implemented. Each set of scanlines given to a processor is determined by an arbitrary processor numbering scheme. The scanline set for processor i is computed by choosing an interleaved set starting at scanline i, and adding scanline $i + jn$, while $i + jn < s$ ($j = 1, \ldots \infty$, $n =$ the number of processors, $s =$ the number of scanlines) (See Figure 4.6). In this manner, each pro-

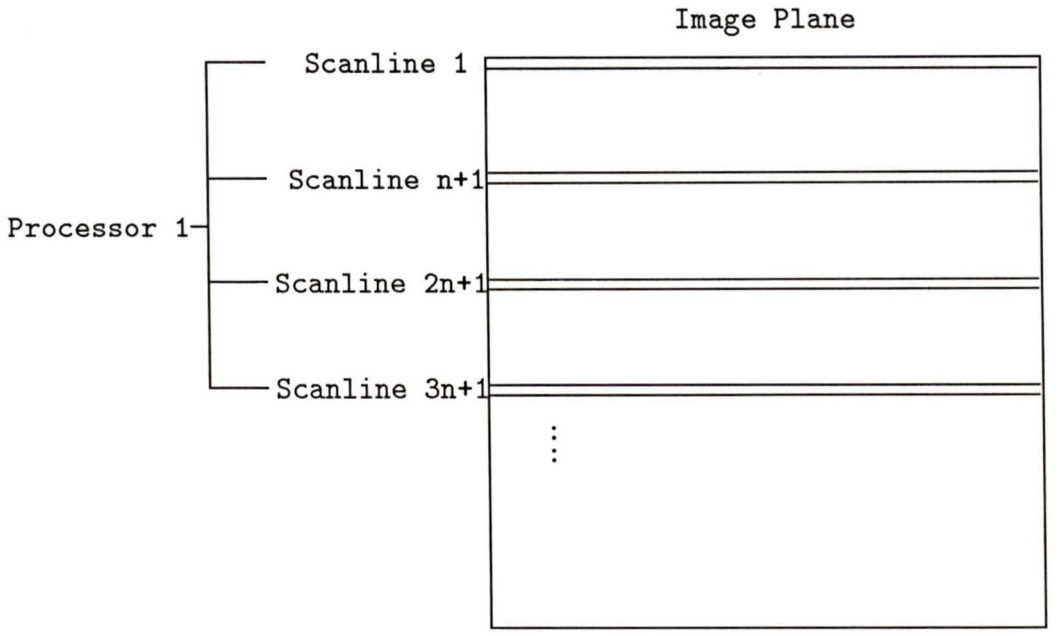


Figure 4.6: Scanline sets for SunRay II

Processing node	Processor load	Secondary rays	Total rays	Intersections	System time [sec]	Elapsed time [sec]
1	light	84,820	117,588	1,230,719	3059.1	3180
2	light	84,690	117,458	1,237,124	3058.48	3180
3	light	84,535	117,303	1,237,678	3057.96	3180
4	light	84,827	117,595	1,231,643	3445.18	3540
5	light	84,983	117,751	1,232,936	3425.74	3540
6	light	85,053	117,821	1,234,475	3051.8	3180
7	heavy	85,017	117,785	1,242,680	3606.82	4320
8	light	84,950	117,718	1,239,771	3063.22	3180

Table 4.5: SunRay II statistics for the mountain (homogeneous processors)

Processing node	Processor type	Processor load	Rays traced	Intersections	System time [sec]	Elapsed time [sec]
1	SUN 3/110	Light	117,588	1,230,719	3054.64	3240
2	SUN 3/110	Light	117,458	1,237,124	3058.34	3180
3	SUN 3/110	Light	117,303	1,237,678	3059.06	3180
4	SUN 3/75	Light	117,595	1,231,643	3452.6	3720
5	SUN 3/75	Heavy	117,751	1,232,936	3469.5	4860
6	SUN 3/75	Light	117,821	1,234,475	3048.72	3540
7	SUN 3/75	Light	117,785	1,242,680	3059.66	3180
8	SUN 4/370	Light	115,478	1,231,717	418.73	442

Table 4.6: SunRay II statistics for the mountain (heterogeneous processors)

cessor is assigned an approximately equal amount of work, since no spatial coherence is maintained in the scanline sets, and no processor receives a set of adjacent scanlines that is "easy" to process. This algorithm performs far better than SunRay I, since the amount of work bundled for each machine is more evenly distributed. However, this algorithm still assigns the same amount of work to each processor (each scanline set is the same size) regardless of the processing power and machine load.

In Table 4.5, the mountain image is used as a test image. The new scanline distribution policy now distributes the work to the processors in an interleaved manner. The secondary rays are distributed to all nodes, resulting in a more even distribution of work. This is demonstrated by the approximately equal number of intersections performed by each processing node. Note the significant increase in the elapsed time of processor seven. The elapsed time is the actual time that has passed since the process started, while the system time is the amount of CPU time used by the process.

In this case, the system time is close to the other nodes (i.e., within 220 seconds of the closest node), while the elapsed time is almost 800 seconds longer. This is due to the high external load on processor seven, and demonstrates a significant problem with the even distribution policy used by this method.

Up to this point, only similar types of processing nodes have been used, each one carefully chosen to have little or no computational load external to the ray tracing system. When this distribution policy is used in a distributed environment (as described in section 4.1.6) it no longer performs well. To demonstrate this fact, several different machine architectures are used, each with an external load from other users of the system. The results are shown in Table 4.6. To show how much of an effect external load and differing architectures can have on the load-balancing strategy, a "worst case" scenario is contrived. This is done by creating a heavy load on one of the slower processing nodes (Node 5, a Sun 3/50) used for this test, while leaving the fastest processing node (Node 8, a Sun SPARCServer 370) with little or no external load (relative to its processing power). In Table 4.6, each node is still processing approximately the same number of rays, but a significant difference in rendering times exists among them.

The rendering time differences among these processing nodes, even with an approximately equal amount of work, is significant. Node 8 completes its processing in an elapsed time of 442 seconds, while the other nodes require between 3180 and 4860 seconds to complete their work. Note that Node 5 takes over 1100 seconds longer (elapsed time) than the next slowest processing node. Again, this is due to the external load applied to that processor. This demonstrates the primary problem of a static load-balancing scheme in a dynamic, distributed environment. Even if the relative processing power of the nodes is known before rendering (i.e., the scanline set sizes can be adjusted to balance the amount of work more evenly) this distribution policy

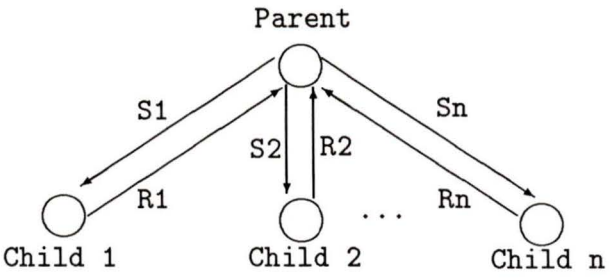


Figure 4.7: Communications for SunRay III

cannot adapt to the dynamic loads that are common in distributed systems. Thus, this policy does not approach the linear decrease in rendering time that is hoped for from a parallel system.

4.2.4 SunRay III – a dynamic load-balancing strategy

The implementation of a dynamic, distributed ray tracing system comes close to achieving the theoretical goal of a linear reduction in rendering time. A controller process is used to keep track of the current scanline being processed. Initially, the controller process creates n children, each on a different machine in the network. The parent process then informs each of the children as to which scanline to process next (the arcs labeled Si in Figure 4.7). After a child completes a scanline, it outputs the image data to an output file for that child, informs the parent of its completion (the arcs labeled Ri in Figure 4.7), and waits for the next scanline to process. Only scanline identifiers, and not the scanlines themselves, are passed between processors. When there are no more scanlines to process, the children are informed, and each one exits when it completes its last scanline. The output files produced by the children are merged in a postprocessing step.

If a given child is assigned to process a simple area of the image plane, it will complete each scanline quickly and immediately ask for more work. In this manner, even completion times for all children are guaranteed. The worst-case completion time difference is t_{max}, where t_{max} is the time it takes to render the most complex scanline.

Let processor p be the first to complete, with no more scanlines to be processed. Clearly, all other machines must finish before time t_{max} passes, since t_{max} is the maximum time it takes to render a scanline. At most, it will take time t_{max} for all other machines to finish. This will only happen if all other machines finish just before processor p, and all of them have one more scanline to process that is of maximum complexity. In all other cases, the differences in completion times will be less than t_{max}. As long as the overhead of the communications is not too large, this algorithm will produce a near linear reduction in rendering time.

4.2.4.1 REM – a remote execution manager

REM is a remote execution manager developed by the Real-Time Systems Group at the University of Victoria. It was developed to study the remote execution of distributed tasks in a local area network environment [SHOJA 88A] [SHOJA 88B]. It provides a means of remotely executing tasks on various machines, in a manner transparent to the invoking process. It also provides a number of utilities, such as process checkpointing, process duplication, performance monitoring, distributed debugging, process communication utilities, and a measure of load balancing. When a task is started through REM, it is automatically started on the machine that has the lightest load. This provides an automatic distribution of the load to the least loaded machines in the LAN. For example, if fourteen machines are in the LAN, and ten are to be used by the ray tracer, REM will start the ray tracer on the ten most

lightly loaded machines, balancing the load across those ten as evenly as possible. After starting the ray tracing processes, it is the responsibility of the ray tracing system to perform further load balancing.

REM also provides a layer on top of the standard UNIX communication protocols, making it simpler for remote processes to communicate with each other. Using REM as a means of implementing the distributed portion of the ray tracer removes the need to use the lower-level communication protocols, and simplifies the implementation of the load-balancing strategy. The extra layer causes some inefficiencies, but communication is minimal, and it is believed that the gain in evenly balanced load will offset the overhead in communication costs. If it were required to use pixels, rather than scanlines, as the atomic element for load balancing, then the communication costs would be much more important, since communication would take place more frequently. This situation may arise as communication and processor speeds change, making it necessary to adapt the granularity of the parallel processing.

4.2.4.2 A distributed ray tracer

Through the use of REM and the load-balancing strategy described above, a *dynamic, load-balanced, distributed* ray tracing system was implemented. To run the ray tracing system, REM must be running on every machine that is eligible for use by the system. The REM system processes simply wait for the application processes to connect with REM, and once connected, handle all inter-process communication. To start the ray tracing processes, a single UNIX command is entered, telling the system how many machines and which scene description input file to use. Since REM is running on all machines, and the users of SunRay III have control over how many machines they want to use, it is simple to use only idle machines on the network. By decreasing the number of machines to be used by the ray tracer, the n machines with the lowest load

will be used.

The SunRay III system is both dynamic in its handling of the load of the system, and robust in its recovery from system failure. Because each scanline is assigned to a process dynamically, the system adjusts to the changing load in the system. If a machine becomes loaded because of use from other users of the system, the processing that cannot be handled by the loaded machine is transferred to other machines automatically, since the loaded machine will ask for scanlines less often.

If a machine was to fail (excluding the machine with the controller process) for any reason, fault detection and recovery is simple. When a machine fails, it no longer asks for scanlines to process, and the rest of the processing is simply distributed to the other machines in the system. Upon completion of the image, for each failed machine at most one scanline is corrupted. This is easily detected by the centralized processor, as it is informed when each scanline is finished, and therefore knows when a node has not completed its designated processing. It takes little time to render this individual scanline and to merge it back into the image. The simplicity of the centralized control in the load-balancing strategy outweighs the occasional cost of having to restart the rendering system when the machine with the controller process fails. As of this writing, only one fault has occurred that has caused a restart of a rendering job (excluding those that were contrived for testing purposes).

The major drawback of the SunRay III system is that all machines used in the computation of a given image are going to be heavily loaded during the duration of the computation. This is clearly unfair to the other users of the system. Although REM will start the ray tracing processes on the least loaded machines, it is not always possible to have as many idle machines available as desired. In this case, it is necessary to start a SunRay III job on a machine that already has a fairly high load, and as a result, these machines will have slow response times. The slow response times

are noticeable to jobs that are interactive, such as window management processes and other user applications. One solution to this problem is to use the UNIX *nice* command [SUN 88] to lower the priority of the SunRay III processes. This allows all normal interactive user processes to maintain a reasonable response time, while leaving no CPU cycles unused. In a normal operating environment, where most jobs are short, this option behaves quite well. There is, of course, a slow down of all processes on any given machine, but this is to be expected when a CPU intensive task such as ray tracing is performed.

If the system SunRay III is being run on has a large number of CPU bound processes, as might be found in a research environment, this may be an undesirable result, since the SunRay III processes would get only a small percentage of the processor in these cases. Of course, if all machines are idle when SunRay III is run, it uses 100% of the processor time available, regardless of whether the *nice* option was used or not. The use of the *nice* command is selected by the user of the SunRay III system by a UNIX command line flag.

4.2.4.3 Results

The dynamic load-balancing policy of SunRay III was tested on both the mountain and recursive tetrahedron test images. In Table 4.7, the recovery from faults at processing nodes is demonstrated. The recursive tetrahedron is rendered using SunRay III, with processor faults being induced by terminating the REM process on nodes three and seven. An approximately even set of completion times is achieved for all active nodes, with the active nodes completing the processing that was lost by the faulty ones. Only the last scanline rendered by each faulty processor is lost when they are killed (processors three and seven completing 56 and 30 scanlines respectively). This scanline is later rendered by one of the other processing nodes before termina-

Processing node	Secondary rays	Total rays	Intersections	System time [sec]	Elapsed time [sec]
1	6,316	38,572	510,204	1292	1336
2	7,229	48,189	577,923	1270	1317
3	FAULT	NA	NA	NA	NA
4	7,406	46,830	587,757	1270	1324
5	7,180	45,068	569,066	1246	1301
6	7,185	43,537	564,284	1228	1297
7	FAULT	NA	NA	NA	NA
8	6,433	39,713	509,603	1169	1291

Table 4.7: SunRay III statistics for the tetrahedron (with processor faults)

tion. This recovery is performed automatically by the SunRay III system, with no user intervention required. A rendering speed increase of 4.77 times that of a single processor is achieved in this case.

In the mountain example, conditions similar to those used to demonstrate the flaws in SunRay II's load-balancing policy are created. For this test, several processing nodes are heavily loaded, and the more powerful Sun 3/280 (Node 4) is lightly loaded. The Sun SPARCServer 370 used to test SunRay II was not used for SunRay III because REM is not yet available for RISC architectures. Table 4.8 shows the even completion times of the processing nodes (within 18 seconds of each other), regardless of the contrived "worst case" scenario of unbalanced external loads. Note the significant increase in the number of intersections performed by the Sun 3/280 machine (Node 4) over that of the heavily loaded Sun 3/50s and SUN 3/60s (Nodes 1, 6, and 8). A rendering speedup of 4.28 times that of the single processor case is achieved (elapsed time for one processor is 16795 seconds).

Processing node	Processor type	Processor load	Rays traced	Intersections	System time [sec]	Elapsed time [sec]
1	SUN 3/75	heavy	87,687	993,054	2710.64	3927
2	SUN 3/75	light	123,798	1,249,288	3503	3925
3	SUN 3/75	light	119,820	1,201,729	3435.84	3926
4	SUN 3/280	light	205,417	2,254,532	3333.72	3917
5	SUN 3/75	light	134,510	1,404,030	3334.94	3915
6	SUN 3/110	heavy	64,190	582,634	2175.76	3917
7	SUN 3/75	light	129,879	1,372,810	3303.02	3912
8	SUN 3/75	heavy	75,568	822,030	2703.12	3909

Table 4.8: SunRay III statistics for the mountain (heterogeneous processors)

The non-linearity can be attributed to three factors: the overhead of the communication required by the load-balancing policy, the external loads placed on the system, and the duplicate preprocessing stage that takes place on each processing node to create the voxel hierarchy. If the preprocessing times are excluded from the rendering time, SunRay III comes close to a linear speedup in the number of processors used. In Table 4.8, the node with the longest rendering time (actual time spent rendering) is Node 4, at 2856 seconds. On one processor, the time spent rendering is 15,274 seconds. This is a rendering time reduction of 5.35, much closer to linear than the previous result. Because several processors are heavily loaded, a linear increase is not expected in this case.

In Table 4.9, the recursive tetrahedron is rendered with eight processors, no external loads, and homogeneous processors. The results are then compared to the tetrahedron rendered on one processor. On one processor, the tetrahedron was rendered in

Processing node	Total rays	Intersections	Elapsed preprocessing time [sec]	Total system time [sec]	Total elapsed time [sec]	Elapsed render time [sec]
1	33,440	416,490	541	1115	1154	613
2	38,478	459,866	499	1116	1150	651
3	38,081	460,948	497	1112	1146	649
4	29,907	378,241	551	1110	1144	593
5	37,038	456,412	505	1096	1143	638
6	36,871	453,575	502	1097	1138	636
7	36,242	454,351	503	1095	1137	634
8	61,196	596,731	390	1102	1135	745

Table 4.9: SunRay III statistics for the tetrahedron (homogeneous processors)

6369 seconds, with 5816 seconds of rendering and 553 seconds of preprocessing time. The SunRay III rendering was performed in an elapsed time of 1202 seconds. This gives a rendering time reduction of 5.3 times that of the time it took one processor. If only the rendering times are compared, a much more favorable result is obtained. Using the longest elapsed rendering time of the SunRay III job (745 seconds) a rendering time reduction of 7.81 times over that of the single processor is obtained. This reduction is very close to the linear goal, as well as to the results obtained by Green and Paddon [GREEN 89A]. Although Green and Paddon do not use eight processing nodes, by performing a linear interpolation on their results for six and ten processing nodes, a rendering time decrease of 7.83 times is estimated for the tetrahedron scene with 4096 primitives.

In SunRay III, each processing node must read the entire database and perform

the voxel subdivision individually. For the tetrahedron scene, the file read takes approximately 45 seconds to complete. The voxel subdivision takes 453 seconds on average. The overhead of this processing causes the relatively poor performance of this method when the total elapsed time is considered. Determining a method to reduce this overhead is an interesting area of research, and would increase the overall efficiency of the system a significant amount.

One of the major problems of SunRay III, or any image space algorithm, is the fact that each processing node must have access to the entire scene database. This is achieved for SunRay III by requiring that each processing node be able to perform the rendering individually. Large scene databases can be rendered because of the virtual memory that is used by UNIX systems. If the entire scene database does not fit into the processing nodes' memory, parts of it are paged out to disk by the operating system. A problem arises when processing nodes are diskless, and are served by the same file server. If the file server is swapping pages of memory for several processing nodes, it becomes a processing bottleneck. This bottleneck does not decrease the effectiveness of the load-balancing scheme, but it does decrease the effectiveness of the parallel system as a whole. Even completion times are still achieved, but the result moves further away from linearity as more and more paging occurs. Green and Paddon's system [GREEN 89A] does not seem to suffer from this problem, although the scene databases they use for testing are relatively small, containing anywhere from 4096 to 8401 objects. The AT & T Pixel Machine [POTMESIL 89] does suffer from this problem, and a rendering speed decrease of a factor of five is reported when the database gets too large for local processor memory (5000 objects).

4.3 Summary

This chapter outlined the stages of the step-wise refinement of a distributed ray tracing algorithm. It culminated with a description of a robust, dynamic, load-balanced, distributed system that is able to adapt to changes over time in the status of the system. It meets all of the criteria for a distributed system set out earlier in the chapter, and approaches the theoretical limit of a linear-time reduction with the number of processors used in the system.

Chapter 5

Conclusions

This thesis presents a workbench for creating realistic computer generated images. The workbench is a cohesive system for creating and manipulating three-dimensional models and two-dimensional images. A three-dimensional visual model editor and a sophisticated rendering system were developed by the author as the key components in the system. The workbench provides an extensible *image synthesis environment* that is capable of producing both *complex three-dimensional models* and *realistic computer generated images.*

5.1 Summary of results

The workbench is implemented in a LAN environment, which provides two essential features to the system. Workstations, a common component of a LAN, are used to provide sophisticated three-dimensional *visual modeling tools.* The processing power available in the LAN is utilized by *parallelizing* the rendering process, providing a fast, efficient, and powerful rendering system. A cohesive workbench is maintained

through the use of a *rendering interface* that is adhered to by all components of the system.

To provide the rendering component of the workbench, a *ray tracing* system was implemented. To increase the performance of the system, a technique known as *space subdivision* was used, resulting in rendering time decreases of an order of magnitude. Several LAN parallel implementations of the ray tracing system were tested. Results that approach a linear decrease in rendering time, with respect to the number of processors, were achieved.

As the system developed, and rendering times decreased to a practical level, another important problem in image synthesis was recognized: the creation of realistic three-dimensional models is a very difficult task. The model editor developed for this work provides a set of powerful *modeler operations* that are used to create complex three-dimensional scenes. These operations allow model primitives to be *grouped* to form complex objects that are logically related at the human perceptual level. The model editor is an interactive tool, providing a fast and efficient means of prototyping three-dimensional models for rendering.

To facilitate the efficient transfer of three-dimensional modeling data between workbench components, a rendering interface was developed. This interface provides an effective way to expand the workbench, integrating new modeling and rendering components as needed. By adhering to the rendering interface, these components can communicate with all other components of the system.

5.2 Future research

Image synthesis, as a science and as an art, is still in its infancy. Much research is required before flexible photorealistic image synthesis will be possible. The number

of phenomena that must be modeled are simply too extensive for current techniques. The areas of parallel rendering, three-dimensional modeling, and rendering interfaces all require more extensive research. Areas of future research are:

1. More advanced modeling systems are always needed. Research into the vital operations that are essential for modeling systems in general is required. By basing a modeling system on these essential operations, specialized modeling tools can be built quickly and easily.

2. Modeling systems based on three-dimensional virtual worlds offer exciting and powerful alternatives to standard modeling techniques. This area is expected to receive a lot of attention in the near future.

3. Rendering interfaces are a relatively new research area in computer graphics, and the long battle for "the" standard interface has just begun. It is not yet fully clear what is required of a rendering interface.

4. The concept of a shading language offers extensive control over the shading of objects. The area of procedural shading controlled by the user is one of the most exciting areas of research in computer graphics today. Research into powerful and flexible shading models could propel computer graphics to the goal of photorealistic image synthesis.

5. Acceleration techniques for ray tracing offer many avenues for research. Investigations into coherence among scene primitives, data structures, and rays need to be continued to further advance the rendering speed of ray tracing.

6. Parallel algorithms for ray tracing are usually specialized systems. As Green and Paddon state [GREEN 89A], a flexible system that can incorporate new rendering techniques is essential. This may prove to be too much of a contradiction,

as different techniques often require very different approaches. A general parallel approach that will work on all architectures for all scene descriptions is an elusive, yet desirable goal.

7. The file server bottleneck that is predominant in LAN systems prevents very large scene descriptions from being rendered at high speeds. An interesting approach would be to adapt Green and Paddon's [GREEN 89A] data migration scheme to a LAN configuration. Determining whether this approach is feasible in a LAN may lead to some interesting results.

8. Determining a way of distributing the voxel subdivision computation would allow SunRay III to produce similar results to that given by Green and Paddon [GREEN 89A]. This would improve the efficiency of the SunRay III rendering scheme to very close to achieving a linear rendering time decrease with respect to the number of processors used.

9. A method of determining the optimum scanline or pixel sets for parallel processing, given a configuration of processors, may prove to be vital as hardware and communications technologies continue to evolve. Because of the varying loads that exist in distributed systems, this will be difficult to solve exactly, but an approximate solution may be adequate in this area.

Most of these areas of research have emerged in the last decade. This work provides some promising results for using LAN systems for realistic image synthesis. It also provides a framework on which further investigations into these exciting areas can be based.

Bibliography

[ARVO 87] J. Arvo and D. Kirk, "Fast Ray Tracing by Ray Classification," *Computer Graphics*, Volume 21, Number 4, July 1987.

[AKELEY 88] K. Akeley and T. Jermoluk, "High-Performance Polygon Rendering," *Computer Graphics*, Volume 22, Number 4, August 1988.

[CLEARY 83] J. Cleary, B. Wyvill, G. Birtwistle, and R. Vatti, "Multiprocessor Ray Tracing," 83/128/17, Department of Computer Science, University of Calgary, Alberta, October 1983.

[CLEARY 88] J. Cleary and G. Wyvill, "Analysis of an Algorithm for Fast Ray Tracing Using Uniform Space Subdivision," *The Visual Computer*, Volume 4, 1988.

[COHEN 85] M. Cohen and D. Greenberg, "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics*, Volume 19, Number 3, July 1985.

[COHEN 88] M. Cohen, S. Chen, J. Wallace, and D. Greenberg, "A Progressive Refinement Approach to Fast Radiosity Image Gen-

eration," *Computer Graphics*, Volume 22, Number 4, August 1988.

[COOK 84] R. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Computer Graphics*, Volume 18, Number 3, July 1984.

[CROW 87] F. Crow, "The Origins of the Teapot," *IEEE Computer Graphics and Applications*, Volume 7, Number 1, January 1987.

[DIPPE 84] M. Dippe and J. Swensen, "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis," *Computer Graphics*, Volume 18, Number 3, July 1984.

[EAGER 86] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Volume 12, Number 5, May 1986.

[FICHERA 89] R. Fichera, "Visualization: Nuts, Bolts, and Algorithms," *Supercomputing Review*, Volume 2, Number 9, September 1989.

[FIUME 86] E. Fiume, *The Mathematical Structure of Raster Graphics*, Academic Press, Toronto, 1986.

[FOLEY 82] J. Foley and A. VanDam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1982.

[FUCHS 89] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor Enhanced Memories," *Computer Graphics*, Volume 23, Number 3, July 1989.

[FUJIMOTO 86] A. Fujimoto, T. Tanaka, and K. Iwata, "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, Volume 6, Number 4, April 1986.

[GAUDET 88] S. Gaudet, R. Hobson, P. Chilka, and T. Calvert, "Multiprocessor Experiments for High-Speed Ray Tracing," *ACM Transactions on Graphics*, Volume 7, Number 3, July 1988.

[GLASSNER 84] A. Glassner, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, Volume 4, Number 10, October 1984.

[GLASSNER 89A] A. Glassner, "How to Derive a Spectrum from an RGB Triplet," *IEEE Computer Graphics and Applications*, Volume 9, Number 4, July 1989.

[GLASSNER 89B] J. Arvo, R. Cook, A. Glassner, E. Haines, P. Hanrahan, P. Heckbert, D. Kirk, *An Introduction to Ray Tracing*, Editor A. Glassner, Academic Press, Toronto, 1989.

[GOLDSMITH 87] J. Goldsmith and J. Salmon, "Automatic Generation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, Volume 7, Number 5, May 1987.

[GREEN 89A] S. Green and D. Paddon, "A Highly Flexible Multiprocessor Solution for Ray Tracing," Technical Report TR-89-02, University of Bristol, Computer Science Department, March 1989.

[GREEN 89B] S. Green and D. Paddon, "Exploiting Coherence for Multiprocessor Ray Tracing," *IEEE Computer Graphics and Applications*, Volume 9, Number 6, November 1989.

[HAEBERLI 88] P. Haeberli, "ConMan: A Visual Programming Language for Interactive Graphics," *Computer Graphics*, Volume 22, Number 4, August 1988.

[HAINES 86] E. Haines and D. Greenberg, "The Light Buffer: A Shadow Testing Accelerator," *IEEE Computer Graphics and Applications*, Volume 6, Number 9, September 1986.

[HAINES 87] E. Haines, "A Proposal for Standard Graphics Environments," *IEEE Computer Graphics and Applications*, Volume 7, Number 11, November 1987.

[HALL 89] R. Hall, *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, 1989.

[HECKBERT 86] P. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications*, Volume 6, Number 11, November 1986.

[HILLIS 87] W. D. Hillis, "The Connection Machine," *Scientific American*, Volume 256, Number 6, June 1987.

[IMMEL 86] D. Immel, M. Cohen, and D. Greenberg, "A Radiosity Method for Non-Diffuse Environments," *Computer Graphics*, Volume 20, Number 4, August 1986.

[KAY 86] T. Kay and J. Kajiya, "Ray Tracing Complex Scenes," *Computer Graphics*, Volume 20, Number 4, August 1986.

[NICHOLS 87] D. Nichols, "Using Idle Workstations in a Shared Computing Environment," *ACM Operating Systems Review*, Volume 22, Number 3, 1987.

[PEACHEY 85] D. Peachey, "Solid Texturing of Complex Surfaces," *Computer Graphics*, Volume 19, Number 3, July 1985.

[PERLIN 85] K. Perlin, "An Image Synthesizer," *Computer Graphics*, Volume 19, Number 3, July 1985.

[PIXAR 89] Pixar, *The RenderMan Interface*, September 1989.

[POTMESIL 87] Michael Potmesil and Eric M. Hoffert, "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," *Computer Graphics*, Volume 21, Number 4, July 1987.

[POTMESIL 89] Michael Potmesil and Eric M. Hoffert, "The Pixel Machine: A Parallel Image Computer," *Computer Graphics*, Volume 23, Number 3, July 1989.

[ROGERS 85] D. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.

[RUBIN 80] S. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics*, Volume 14, Number 3, July 1980.

[SHOJA 88A] G. Shoja, G. Clarke, and T. Taylor, "REM: A Distributed Facility For Utilizing Idle Processing Power of Workstations," *Distributed Processing*, M. Barton, E. Dagless, and G. Reijns, Editors, North-Holland, 1988, pgs. 205–218.

[SHOJA 88B] G. Shoja, "A Distributed Facility for Load Sharing and Parallel Processing Among Workstations," Internal Report Number DCS–95–IR, Department of Computer Science, University of Victoria, B.C., September 1988.

[SIGGRAPH13 87] R. Cook, A. Glassner, E. Haines, P. Hanrahan, P. Heckbert, and L. R. Speer, "SIGGRAPH '87 — An Introduction to Ray Tracing," SIGGRAPH 1987 Course Notes, Course 13, 1987.

[SIGGRAPH16 89] S. Whitman, F. Crow, H. Fuchs, and N. Gharachorloo, "SIGGRAPH '89 — Parallel Processing and Advanced Architectures in Computer Graphics," SIGGRAPH 1989 Course Notes, Course 16, 1989.

[SIGGRAPH29 89] D. Zeltzer, F. Brooks, R. Deyo, S. Fisher and D. Sturman, "SIGGRAPH '89 — Implementing and Interacting with Real-time Microworlds," SIGGRAPH 1989 Course Notes, Course 29, 1989.

[SUN 88] Sun Microsystems, *SunOS Reference Manual*, May 1988.

[SUTHERLAND 77] I. Sutherland, R. Sproull, and R. Schumaker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, Volume 6, Number 1, March 1977.

[UPSTILL 90] S. Upstill, *The RenderMan Companion*, Addison-Wesley, Don Mills, 1990.

[WALLACE 87] J. Wallace, M. Cohen, and D. Greenberg, "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and

Radiosity Methods," *Computer Graphics*, Volume 21, Number 4, July 1987.

[WALLACE 89] J. Wallace, K. Elmquist, and E. Haines, "A Ray Tracing Algorithm for Progressive Radiosity," *Computer Graphics*, Volume 23, Number 3, July 1989.

[WARD 88] G. Ward, F. Rubinstein, and R. Clear, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics*, Volume 22, Number 4, August 1988.

[WHITTED 80] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, Volume 23, Number 6, June 1980.

[ZHOU 88] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Transactions on Software Engineering*, Volume 14, Number 9, September 1988.

Appendix A

An art gallery

This appendix displays some of the images created by the image synthesis workbench. Figure A.1 demonstrates the most complex primitive provided by the SunRay rendering system, a quadric surface. The quadric shown here is a hyperboloid of one sheet. Notice the distorted reflections of the spheres on the surface of the hyperboloid. This is possible because the ray intersection calculation is being performed with the mathematical formulation of the quadric, and not a polygonal approximation to the surface.

Figure A.2 is a typical ray tracing image of several spheres floating above a polygon. Most ray tracing images are of this form because of the difficulty in producing a complex three dimensional model without modeling tools. It is included here because it demonstrates several advanced features of the rendering system. The image contains three spheres, and a single polygon. The silver sphere embedded in the polygon is a mirror-like sphere, reflecting its environment with great detail. The large sphere is a transparent sphere with a slight pinkish tinge. Notice the distortion of the white sphere near the bottom of the transparent sphere. This is caused by the refraction of

the light as it passes through the transparent sphere. Also note the subtle reflection of the polygon and the silver sphere on the inside surface of the top of the transparent sphere. Another important feature of this image is the penumbral shadows produced by the spheres in the image. This is a result of using stochastic ray tracing to simulate an area light source rather than a point light source.

Figure A.3 is another typical ray tracing image, and is again included to demonstrate the powerful features of the rendering system. Notice the marble texture contained in the pillar itself. This is provided by the procedural textures discussed in Section 4.2. Notice the reflection of the detailed texture in the three spheres.

Figure A.4 is a standard computer graphics image. No work in computer graphics is complete without an image of the teapot [CROW 87]. This image is of a crystal teapot on a marble table top. Notice the complex reflections and refractions of the light incident on the complex areas of the teapot, such as the spout.

Figure A.5 is the most complex image created to date with the image synthesis workbench. This image would not be possible without the powerful modeling tools provided by the workbench. The model for this image was based on an image produced by Hank Weghorst and Gary Hooper of Cornell University's Program of Computer Graphics [GLASSNER 89B]. The goal was to recreate the image to a high degree of accuracy. This was done by building the three-dimensional model from scratch (using Scene), and renering that model with the SunRay rendering system. Although the image displayed in the color plates of [GLASSNER 89B] is superior in quality to the one displayed here, the image rendered by SunRay is of sufficient quality to demonstrate the power of this image synthesis workbench.

Figure A.6 demonstrates the power of three dimensional procedural textures. This image was rendered using only three modeling primitives. The realism of the ocean waves are a result of a "bumpy" procedural texture produced by perturbing the

normal vector of the polygon in a regular manner. The turbulent clouds of the moons rising over the ocean are produced by a very turbulent marble texture. Although this is one of the most visually complex images displayed in the art gallery, it is by far the simplest three dimensional model. This results in a very fast rendering time, even for visually complex images.

All of the images produced for this thesis were obtained by taking photographs of the color monitors of the SUN 3/60 workstations available at the University of Victoria. Color prints were then placed in the thesis and the final pages were reproduced by a color photocopier.

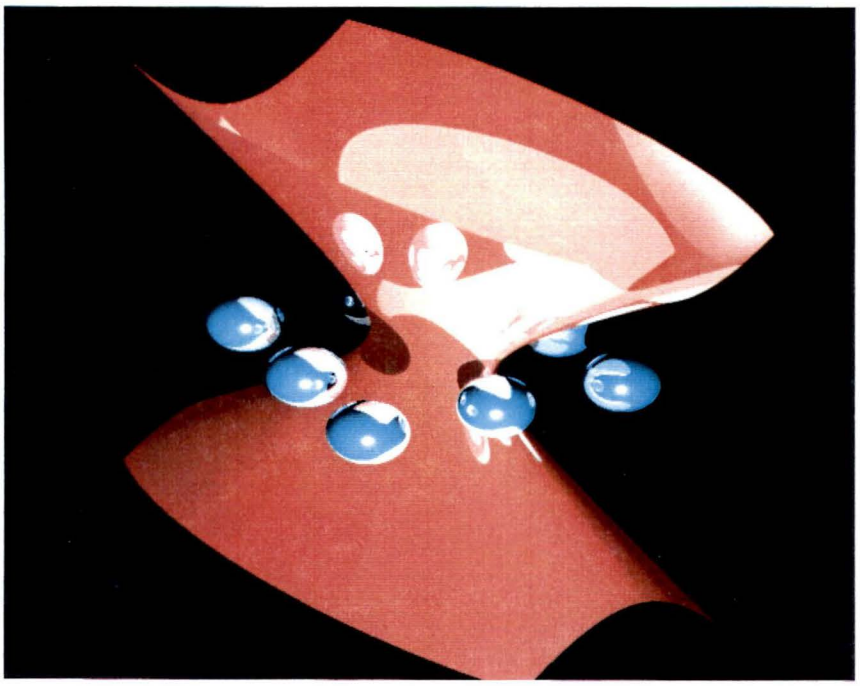


Figure A.1: A hyperboloid of one sheet

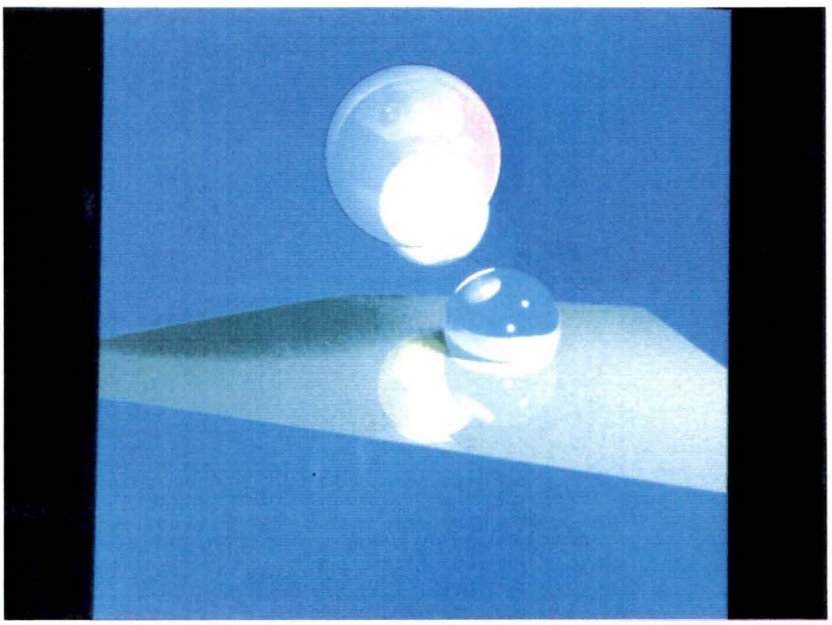


Figure A.2: Refraction, reflection, and penumbral shadows

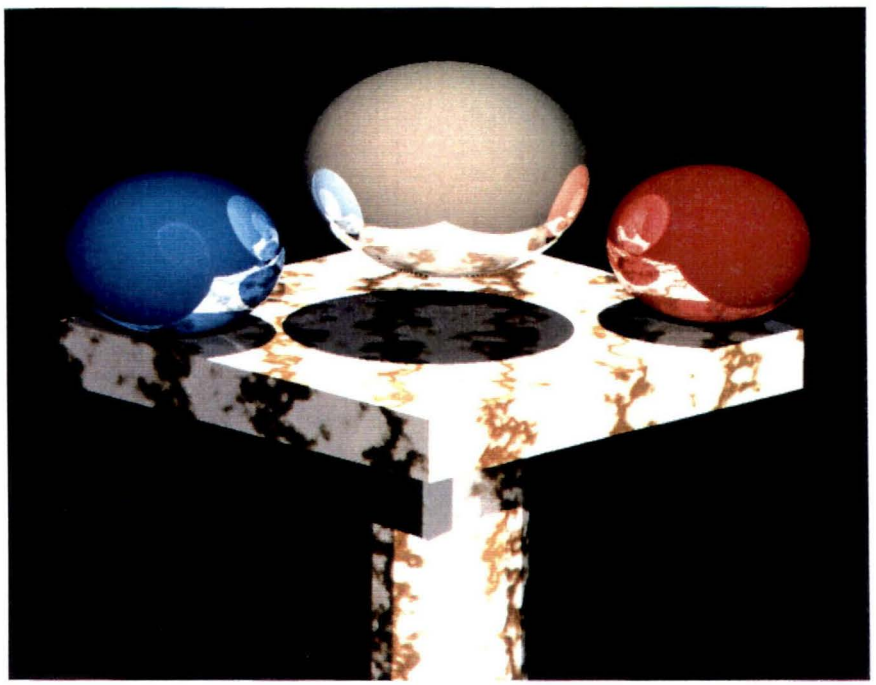


Figure A.3: Marble pillar

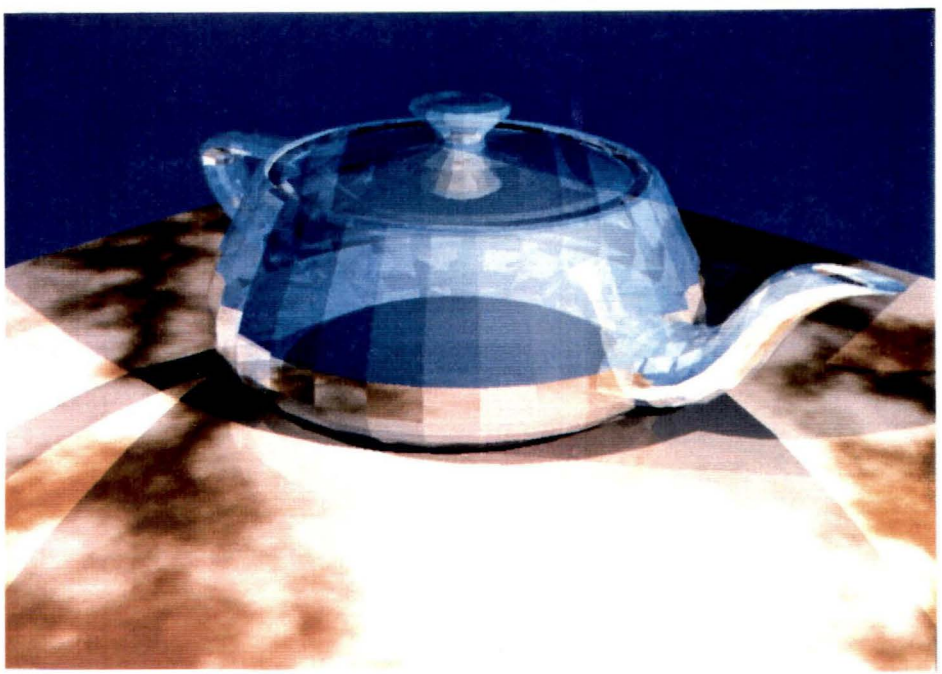


Figure A.4: Crystal teapot

Figure A.5: An evening at the office

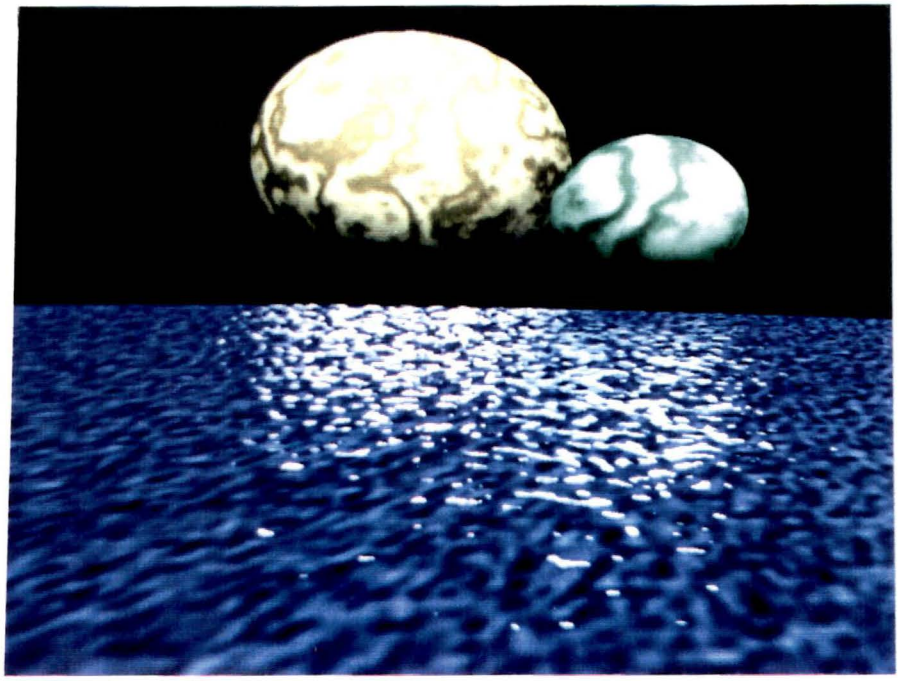


Figure A.6: Seascape

Appendix B

Workbench utilities

This appendix briefly describes the small, yet important utilities used to complete the graphics workbench. The utilities described here come from several sources, many of which originate from the world of public domain, or free software. Where applicable, the authors are credited with the work they have done. Special mention is given to Michael Mauldin and other contributors to the public domain "Fuzzy Bitmap" image processing utilities. This is a large set of utilities that provide image manipulation, image format conversion, and image compression tools. A graphics workbench is not complete without tools like these in its repertoire. One of the most valuable sources of information for the graphics workbench was the Usenet graphics community. Through participating in the discussion of computer graphics, many of the valuable utilities listed in this section were obtained.

B.1 Image manipulation utilities

This section lists the image processing utilities available in the graphics workbench.

fbm The *Fuzzy Bitmap Manipulation* utilities. These tools provide many methods of manipulating images that have been created by the other components of the graphics workbench. A comprehensive list of the tools provided and a brief description of each is given at the end of this appendix. A round of thanks goes out to the authors of this image processing package.

median A powerful image dithering utility. Dithering is the process of converting an n bits/pixel image to an m bits/pixel image. The rendering system used in the workbench produces 24 bits/pixel output files, while much of the display hardware available today can display only eight bits/pixel. Thus, a dithering algorithm is essential. This tool was produced by Ed Falk, at Sun Microsystems of Mountain View, California, and is also public domain software.

show A modified display program for displaying Sun rasterfiles on a Sun workstation. The base program is the standard Sun Microsystems image display program, modified to suit the needs of the graphics workbench.

la An image editor, written by Jim Cole, an undergraduate student at the University of Victoria. It allows images to be edited in much the same manner as the MacPaint program for the Apple Macintosh.

pic2amiga An image conversion utility to convert 24 bits/pixel Sun rasterfiles to Commodore Amiga Interleaved Bit Map (ILBM) files. This type of conversion is possible by the *fbm* tools, but they do not convert directly from 24 bits/pixel to ILBM files, and thereby lose information in the process.

B.2 Scene description utilities

This section lists the scene description manipulation utilities available in the graphics workbench.

display A polygonal wire frame display program, allowing the user to translate, rotate, and scale objects.

yc A YARI compiler that acts as a front end processor for the rendering system. It takes a scene description and converts it to a model the rendering system can process directly.

nff2gfx A scene description file format converter that transforms Eric Haine's Neutral File Format (NFF) scene description format [HAINES 87] to the YARI format.

off2gfx A scene description file format converter that transforms Object File Format (OFF) objects to the YARI format. This format originates from Digital Equipment Corporation's Workstation Systems Engineering division.

check Generates checkerboards out of polygons in user specifyable orientations and colors.

B.3 UNIX shell scripts

This section lists some short UNIX shell scripts, developed by the author, that use the above tools to perform complex image manipulations that are performed frequently.

```
#!/bin/csh
#
# Convert a Sun rasterfile into a PostScript file capable of being
# printed on a PostScript based laser printer. For each file,
# uncompress the rasterfile, convert it to an FBM file, change it
# from a color file to a gray scale file, and finally convert it
# to postscript.
#

foreach i ($*)
      echo "Converting file" $i "to a postscript file"
      sun2sun -s $i | fbcat -F | clr2gray | fbps > $i.ps
end
```

```
#!/bin/csh
#
# Convert a 24 bits/pixel renderer output file to an 8 bits/pixel Sun
# rasterfile. First convert from the renderer image format to a 24
# bits/pixel Sun rasterfile, then dither the image to 8 bits/pixel,
# compress the file using Run Length Encoding, and store the final
# image as a Sun rasterfile.
#
foreach i ($*)
    echo "Converting..."
    ray2sun $i.ray /tmp/$i.sun.tmp
    echo "Quantizing..."
```

```
    median /tmp/$i.sun.tmp /tmp/$i.pic.tmp -fsd
    /bin/rm /tmp/$i.sun.tmp
    echo "Compressing..."
    sun2sun /tmp/$i.pic.tmp $i.pic
    /bin/rm /tmp/$i.pic.tmp
    echo "Conversion complete"
end
```

B.4 Fuzzy bitmap tools

This section briefly describes each of the FBM image processing utilities. This text was taken directly from the README file provided with the distribution of the Fuzzy Bitmap software, with little or no modification.

clr2gray Convert color to grayscale

fbcat Copy image (used for format conversion)

fbclean Flip isolated pixels (clean image)

fbedge Perform edge detection

fbext Extract region, resize, change aspect ratio

fbhalf Halftone grayscale image (Blue noise, Floyd-Steinberg, etc.)

fbhist Compute histogram

fbinfo Dump image header

fbm2pod Convert grayscale image to Diablo graphics

fbmask Set region to gray value

fbnorm Normalize image intensity / increase contrast

fbps Convert to PostScript

fbquant Color quantization (24 bit to 8..256 colors) Modified Heckbert

fbrot Rotate 90, 180, or 270 degrees

fbsample Sample a 1 bit file to produce an 8 bit file

fbsharp Sharpen (edge enhancement) by digital Laplacian

gray2clr Add a "gray" colormap to a grayscale image

idiff (and udiff) convert raw byte stream into byte difference

pbm2ps Convert PBM file to PostScript

pbmtitle Add a title to a PBM file

raw2fbm Convert raw file to FBM format (e.g., Amiga Digiview files)

Appendix C

Workbench integration

This appendix describes how the rendering interface integrates the components of the graphics workbench into a single, cohesive system. First, several ways of creating scene description files are presented. The manner in which a rendering system uses a scene description file is then explained. Some of the weaknesses of the interface are discussed, followed by a simple scene description and the BNF description of the YARI syntax.

C.1 Creating a scene description

There are many ways to create a scene description file. The naive way is to use a pencil and paper to design a scene, and then generate the YARI code that describes that scene by hand. At the other end of the spectrum, *Scene*, the most powerful modeling tool in the workbench, can be used to create scene description files. Various other modeling utilities also exist (see Appendix B) that assist a computer modeler in the creation of scene descriptions. Tools to convert from other scene or object description

formats, such as the Neutral File Format [HAINES 87] and the Object File Format format, allow modelers to use objects and scenes from other rendering systems. In computer graphics, there are several objects that are commonly used to test rendering algorithms. The most famous of these is the Utah teapot [CROW 87]. Tools to create objects, such as the teapot, checkerboards, and the standard procedural database objects [HAINES 87], also exist.

C.2 Rendering a scene description

Once a scene description has been created, it exists in the domain of the rendering interface. This implies that any modeling restrictions inherent in the interface will be inherited by the rendering system. Because different rendering systems have different capabilities, some may be restricted by the interface, while others may not be capable of correctly rendering the scene as described. The second scenario is preferred, and is the reason why the RenderMan rendering interface is so flexible and complex. It is the responsibility of the rendering system to process a scene description, and transform it into a model that it is capable of rendering. Features of the rendering interface that the renderer cannot handle are either ignored (the user should be informed of the restriction), or converted to an acceptable form. For example, if it cannot render quadric surfaces, it could polygonize the surface, and render it as a set of polygons. For the graphics workbench, the transformation from scene description to renderer model is performed by a YARI compiler. This compiler parses the scene description file, applies the transformations to the objects, and outputs a renderer specific scene description. All renderers must have this capability if they are to use the rendering interface.

C.3 Scene description limitations

Although the rendering interface used here is an important part of the graphics workbench, it has some limitations that should be addressed. The language lacks three important features of a structured programming language: variable definitions, function calls, and control flow. With the addition of this functionality, objects can be parameterized (through the use of function calls), assigned to variables, and instantiated, significantly reducing the amount of duplicate code used in the scene descriptions. With the addition of control flow (*for* and *while* loops), complex structures can be built in a more concise manner. An example of where instantiation would be useful is demonstrated in the scene description listed in Section C.5. This description contains several identical polygons that have different transformations applied to each one of them. If this polygon could have been instantiated, the repeated definition would not be necessary.

Another limitation of YARI is the lack of a flexible shading model. In most rendering systems, the shading model is a part of the renderer, restricting its flexibility. Because the shading model provides much of the realism and texture in computer graphics, a more flexible approach is required. Ideally, the scene description language should contain a shading language, that allows the modeler to define complex textures and to apply them to objects as needed. By expanding YARI in these areas, its functionality would approach that of the RenderMan interface.

C.4 The YARI extended BNF syntax

This section contains the Backus-Naur Form (BNF) description of the syntax of the YARI scene description language.

```
<world>      ::= <view> <body>

<view>       ::= <viewelem> ; | <view> ; <viewelem>
<viewelem>   ::= background ( color = <vector> ) |
                 background ( color = <ident> ) |
                 ambient ( color = <vector> ) |
                 ambient ( color = <ident> ) |
                 resolution( <value>,<value> ) |
                 light( position = <vector> ; color = <vector> ) |
                 light( position = <vector> ; color = <ident> ) |
                 camera( position = <vector> ; at = <vector> ;
                         up = <vector> ; angle = <value> )

<body>       ::= begin <blist> end
<blist>      ::= <belem> |
                 <blist> <belem>
<belem>      ::= <attribute> ; |
                 <primitive> ; |
                 <transformation> ; |
                 <body>

<attribute>  ::= attribute ( <alist> )
<alist>      ::= <aelem>  |
                 <alist> ; <aelem>
<aelem>      ::= color = <vector> |
                 color = <ident> |
                 kdr = <value> |
                 kdt = <value> |
                 ka = <value> |
                 phong = <value> |
                 refract = <value> |
                 texture = <value  <value> , <value> , <value>
                                   <value> , <value> , <value>


<prim>       ::= sphere ( center = <vector> ; radius = <value> ) |
                 polygon ( vertices = <value> ; <vertlist> ) |
                 quadric ( <matrix> )

<vertlist>   ::= <vertlist> <vector> | <vector>
<matrix>     ::= <value> <value> <value> <value> <value>
                 <value> <value> <value> <value> <value>
```

```
<vector>    ::= <value> , <value> , <value>
<value>     ::= <number>
<ident>     ::= <identifier> ;
```

C.5 A scene description example

This section contains a simple scene description, using one of the powerful features of YARI, the graphics stack. The scene was created by hand, in approximately one hour, and demonstrates how difficult this form of modeling can be.

```
/*
 * Viewer location = 100,100,500
 * Object location = 0,0,0
 * View up vector = 0,-1,0   angle of view = 20
 */

camera(position=100,100,500 ; at=0,0,0 ; up=0,-1,0 ; angle=20);
resolution( 512,512 ) ;
background( color = 0.1,0.1,0.1 ) ;

/*
 * A small amount of ambient light and two light sources provide
 * the illumination for this scene.
 *
 */

ambient( color = 0.3,0.3,0.3 ) ;
light( position = 200,400,500 ; color = 1,1,1 );
light( position = 0,500,0 ; color = 1,1,1 );

begin

/* Create a polygon at y = 0, and make it reflective. The texture */
/* used contains woodlike veins of dark and light brown.    */

    attribute( color = .6039,.4196,0 ; kdr = .4 ;
               texture = 1 .5176,.4118,.2784 1,1,1 ) ;
```

```
    polygon( vertices=4;
        -160,0,-324
        160,0,-324
        160,0,324
        -160,0,324);

/* Create a big reflective sphere at the back of the scene */

    begin
        attribute( color = 1,.9,.7 ; kdr = .5 ; phong = 160 ) ;
        sphere( center = -160,100,-324 ; radius = 100 ) ;
    end

/* Create a blob of spheres. */

    begin
        attribute( color = 1,0,.8 ) ;
        translate( 0,50,-200 ) ;

        sphere( center = 0,10,0 ; radius = 10 ) ;
        sphere( center = 0,-10,0 ; radius = 10 ) ;
        sphere( center = 10,0,0 ; radius = 10 ) ;
        sphere( center = -10,0,0 ; radius = 10 ) ;
        sphere( center = 0,0,10 ; radius = 10 ) ;
        sphere( center = 0,0,-10 ; radius = 10 ) ;
    end

/*
 * Translate the cube so it is centered at the origin. This transformation
 * is global to all other transformations (ie all cubes will be centered
 * at the origin).
 */

    translate( 0,-15,0 ) ;

/*
 * Create a few cubes with different rotations, scales, and
 * translations. Note that the global transformation centers
 * the cube at the origin, so that the scale and rotations do
 * not deform the cube, but only change its size and orientation.
 */
```

```
begin
    rotate( 45,45,0 ) ;
    scale( 1.5,1.5,1.5 ) ;
    translate( -30,60,0 ) ;

    attribute( color = 0,.7,.8 ; kdr = 0.5 ) ;

    polygon( vertices=4;
        0,0,-32    32,0,-32   32,0,0      0,0,0);
    polygon( vertices=4;
        0,30,-32   32,30,-32  32,30,0     0,30,0);
    polygon( vertices=4;
        0,0,-32    32,0,-32   32,30,-32   0,30,-32);
    polygon( vertices=4;
        32,0,-32   32,0,0   32,30,0     32,30,-32);
    polygon( vertices=4;
        32,0,0     0,0,0      0,30,0      32,30,0);
    polygon( vertices=4;
        0,0,0      0,0,-32    0,30,-32    0,30,0);
end

begin
    rotate( 30,-30,0 ) ;
    scale( 1.5,1.5,1.5 ) ;
    translate( 40,60,-80 ) ;

    attribute( color = 0,1,0 ) ;

    polygon( vertices=4;
        0,0,-32     32,0,-32   32,0,0     0,0,0);
    polygon( vertices=4;
        0,30,-32    32,30,-32  32,30,0    0,30,0);
    polygon( vertices=4;
        0,0,-32     32,0,-32   32,30,-32  0,30,-32);
    polygon( vertices=4;
        32,0,-32    32,0,0     32,30,0    32,30,-32);
    polygon( vertices=4;
        32,0,0      0,0,0      0,30,0     32,30,0);
    polygon( vertices=4;
        0,0,0       0,0,-32    0,30,-32   0,30,0);
end
```

```
    begin
        rotate( 0,-10,45 ) ;
        scale( 2,2,2 ) ;
        translate( -150,60,40 ) ;

        attribute( color = 1,0,0 ) ;

        polygon( vertices=4;
            0,0,-32     32,0,-32    32,0,0     0,0,0);
        polygon( vertices=4;
            0,30,-32    32,30,-32   32,30,0    0,30,0);
        polygon( vertices=4;
            0,0,-32     32,0,-32    32,30,-32  0,30,-32);
        polygon( vertices=4;
            32,0,-32    32,0,0     32,30,0    32,30,-32);
        polygon( vertices=4;
            32,0,0      0,0,0       0,30,0     32,30,0);
        polygon( vertices=4;
            0,0,0       0,0,-32     0,30,-32   0,30,0);
    end

end
```

VITA

Surname: **Corrie** Given Names: **Brian Douglas**
Place of Birth: **Kimberly, British Columbia** Date of Birth: **August 10, 1964**

Educational Institutions Attended:

University of Victoria	1984 to 1988
University of Victoria	1988 to 1990

Degrees Awarded:

B.Sc.	1988	University of Victoria, British Columbia

Partial Copyright License

Title of Thesis:

A Workbench for Realistic Image Synthesis

Author: [redacted]
Brian D. Corrie
April 19, 1990