Efficient Rendering of Synthetic Images

by

Armando Garcia

B.S., University of New Haven
(June 1979)

S.M., Massachusetts Institute of Technology
(February 1982)

E.E., Massachusetts Institute of Technology
(February 1985)

Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1986

© Armando Garcia, 1986

The author hereby grants to M.I.T. permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
February 1986

Certified by ______________________________
Donald E. Troxel, Thesis Supervisor

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 1 1 1986

LIBRARIES

Archives

Efficient Rendering of Synthetic Images

by

Armando Garcia

Submitted to the Department of
Electrical Engineenng and Computer Science
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy in
Electrical Engineering and Computer Science

Abstract

Ray tracing is considered to be the most elegant technique for rendering high quality computer generated images of three-dimensional environments. It combines the effects of hidden surfaces, shadows, reflection, and refraction, that are difficult or impossible to achieve by other rendering techniques. The main disadvantage with ray tracing is that it point-samples the environment with infinitesimal rays, which is computationally expensive and results in aliasing artifacts. The only solution to the aliasing problem in ray tracing is to oversample the space with additional rays, which further increases the computational requirements.

This thesis presents a novel image rendering algorithm that simulates the global illumination effects produced by ray tracing, but at considerable reduction is computational expense. Computational efficiency is achieved by considering only polygonal environments and using conventional hidden surface algorithms that can exploit the coherence properties of these environments. The visible surface algorithm used is based on an extension of the Newell, Newell, and Sancha list-priority algorithm and includes the simulation of planar reflection, and an approximation to planar refraction, using a recursive linear mapping technique. This recursive visible surface method is equivalent to tracing multiple rays in parallel through an environment to determine visible surfaces, reflections, and refractions. The new rendering algorithm also includes shadow generation, using a shadow projection approach, and simulates linear and non-linear transparency effects. In addition, four different scan conversion techniques are presented, which provide anti-aliased scan conversion of faceted, Gouraud, and Phong type polygons.

Experimental statistics for a variety of polygonal environments, ranging in complexity from 70 polygons to over 24,000 polygons, consistently indicate the efficiency improvement of this new rendering method over conventional ray tracing. Run-time statistics for these environments show the new rendering algorithm to be between 75 and 950 times faster than conventional (single ray per pixel) ray tracing, with the improvement proportional to scene complexity. Furthermore, real-time performance can be expected for moderately complex environments, provided that the new algorithm is supported with a suitable graphics engine.

Thesis Supervisor: Dr. Donald E. Troxel
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I would first like to thank my thesis supervisor, Professor Donald E. Troxel, for his guidance and encouragement throughout this research. I would also like to thank my readers, Professors Robert Halstead and David Zeltzer, for their numerous suggestions and comments. Professor Halstead's careful reading and editing of this document, and Professor Zeltzer's contribution of numerous polygonal objects is especially appreciated. During the course of this research, I have been a research staff member at the IBM Thomas J. Watson Research Center, and I am grateful for their financial support. I especially want to express my sincere appreciation to Dr. Richard Freitas, for his unlimited assistance throughout my research.

I also wish to acknowledge the assistance that I have received from fellow members of CIPG and DSPG, especially Mike Isnardi, for his assistance with text processing, Dennis Martinez, for his help with the color graphics recorder, and John Wang, for his contribution of the ray tracing program.

Finally, I would especially like to thank my wife, Lourdes, for her patience, warm encouragement, and assistance throughout this entire ordeal.

Dedicated

to my wife and parents

Table of Contents

Title Page 1

Abstract 2

Acknowledgements 3

Table of Contents 6

List of Figures 10

List of Tables 14

Chapter 1: Introduction 15

1.1. Realism in 3-D Computer Graphics 17

1.1.1. Object Modeling 17

1.1.2. Illumination Models 19

1.1.3. Polygon Shading 22

1.2. Conventional Image Rendering Techniques 25

1.3. Ray Tracing 28

1.4. Motivation for New Research 30

Chapter 2: Enhanced Ray Tracing Techniques 33

2.1. Introduction 33

2.2. The Bottleneck in Ray Tracing 33

2.3. Anti-Aliasing Difficulties 36

2.4. Distributed Ray Tracing 36
2.5. Ray Tracing with Cones 38
2.6. Beam Tracing Polygonal Environments 40
Chapter 3: New Techniques for Efficient Image Rendering 42
3.1. Introduction 42
3.2. A New Approach to Image Synthesis 43
3.3. Outline of The Algorithm 46
3.4. Image Generation Processor 49
3.4.1. Recursive Visible Surface Processor 50
3.4.2. Reflection and Refraction Mapping 54
3.4.3. Shadow Generation 56
3.4.4. Virtual Image Tree Rendering 56
3.5. Scan Conversion Processors 58
3.5.1. Scan Conversion Method 1: Area Coverage 60
3.5.2. Scan Conversion Method 2: Pixel Mask 61
3.5.3. Scan Conversion Method 3: Hybrid Area/Mask 63
3.5.4. Scan Conversion Method 4: Multi-Level Pixel Masks 64
3.6. Implementation and Results 69
3.7. Discussion 76
Chapter 4: Image Generation Processor 78
4.1. Introduction 78

4.2. Overview 78

4.3. Input Processing 80

4.4. Virtual Image Creation 84

4.5. General 3-D Clipping 89

4.6. Reflection/Refraction Mapping 93

4.7. Shadow Generation 101

4.8. Illumination Model and Shading Options 106

4.9. Non-Linear Transparency 111

4.10. Blending Options and Output Polygon Formats 113

Chapter 5: Scan-Conversion Processors 120

5.1. Introduction 120

5.2. Overview 120

5.3. Anti-Aliasing Techniques 123

5.4. Hidden-Surface Algorithm 126

5.5. Gouraud/Phong Polygon Tilers 127

5.6. Scan Conversion Method 1: Area Coverage 133

5.7. Scan Conversion Method 2: Pixel Mask 141

5.8. Scan Conversion Method 3: Hybrid Area/Mask 149

5.9. Scan Conversion Method 4: Multi-Level Pixel Masks 151

5.10. Virtual Frame Buffer 162

Chapter 6: Results, Conclusions, and Extensions 164

6.1. Implementation Overview 164

6.2. Comparison with Ray Tracing 165

6.3. Analysis of the New Rendering Method 185

6.4. Conclusions and Suggestions for Further Research 188

6.4.1. Hierarchical Visible Surface Processor 190

6.4.2. Improved Shadow Generation 192

6.4.2. Improved Modeling of Refraction 193

Appendix: Implementation Overview 194

A.1. Image Generation Processor 194

A.2. Scan Conversion Processors 198

A.3. User Manual Pages for all Programs 201

Bibliography 221

List of Figures

1.1 Polygonal Champagne Glass 18

1.2(a) Faceted Shaded Champagne Glass 23

1.2(b) Gouraud Shaded Champagne Glass 23

1.3 Geometry of Ray Tracing 29

2.1 Geometry of Amanatides' Cone 38

3.1(a) Example of New Algorithm 45

3.1(b) Example of Ray Tracing 45

3.2 Block Diagram of the New Image Rendering Process 47

3.3 Recursive Visible Surface Algorithm 51

3.4(a) Cube on a Mirror 53

3.4(b) Virtual Image Tree 53

3.5 Reflection Mapping Example 54

3.6 Summary of Scan-Conversion Techniques 59

3.7 Subpixel Polygon Fragment and its Pixel Mask 62

3.8 Pixel-Structure and Pixel-Fragment Definitions 65

3.9 Complex Pixel Example 68

3.10(a) Method 1: (512 × 512, VAX-11/785 Time = 14s + 63s) 70

3.10(b) Method 3: (512 × 512, VAX-11/785 Time = 14s + 155s) 71

3.11(a) Method 2: (512 × 512, VAX-11/785 Time = 10s + 100s) 72

3.11(b) Method 4: (512 × 512, VAX-11/785 Time = 10s + 108s) 73

3.12 Method 3: (512 × 512, VAX-11/785 Time = 12s + 155s) 74

3.13 Method 4: (512 × 512, VAX-11/785 Time = 3s + 55s) 75

4.1 Image Generation Processor Block Diagram 79

4.2(a) Scene Descriptor File Example 81

4.2(b) Resulting Image 81

4.3 Virtual Image Tree 85

4.4(a) Simple Depth Sorting Example 88

4.4(b) Newell, Newell, & Sancha Sorting Example 88

4.5 Truncated Viewing Pyramid in The Eye Coordinate System 89

4.6 General Clipping Volume 91

4.7 Geometry of Planar Reflection & Refraction 94

4.8(a) Refraction by a Plane Under an Orthographic Projection 97

4.8(b) Geometry of Refraction Under an Orthographic Projection 97

4.9 View of an Underwater Checkerboard From Air 99

4.10(a) Refracted Paraxial Rays by a Plane 100

4.10(b) Geometry of a Refracted Paraxial Ray 100

4.11 Shadow Volume Between Light Source and a Polygon 102

4.12 Geometry of Shadow Projection 104

4.13 Geometry of Reflection for Shading Calculations 107

4.14 Virtual Image Tree Polygon Formats 114

4.15(a) Cube on a Mirror 116

4.15(b) Virtual Image Tree 116

4.16 Parent Shading Interpolation Example 118

5.1 Scan Conversion Processor Block Diagram 122

5.2 Establishing Initial Left and Right Edges 128

5.3(a) Polygon Edge Structure Definition 129

5.3(b) Highlight Normal Structure Definition 129

5.4 Polygon Example Spanning Four Scan Lines 130

5.5 Examples of Typical Scan Line Segments 132

5.6 Two Typical Pixel Coverage Situations 134

5.7 Segment Processor I Block Diagram 135

5.8 Normalized Trapezoidal Scan Segment 136

5.9 Normalized Trapezoidal Scan Segment Examples 137

5.10 Complex Pixel Example (edge F over background B) 140

5.11 Subdividing a Pixel into n × m Subpixels 142

5.12 Scan Line Along Top Edge of a Polygon 144

5.13 Typical Polygon Scan Line Segment With Ht = 1 145

5.14(a) Cube on a Partly Diffuse Mirror 154

5.14(b) Virtual Image Tree 154

5.15 Pixel-Struct Definition 156

5.16 Pixel-Fragment Definition 156

5.17 Pixel-Status Definition 156

5.18 General Complex Pixel Fragment List 157

6.1(a) Mirror(NS) Scene : (VAX-11/785 Time = 1.4s + 45s) 168

6.1(b) Mirror(1S) Scene : (VAX-11/785 Time = 3.2s + 55s) 169

6.1(c) Mirror(1S) Scene : (Ray Traced; VAX-11/785 Time = 1.75h) 170

6.2(a) Gallery(NS) Scene : (VAX-11/785 Time = 3.2m + 3.6m) 171

6.2(b) Gallery(1S) Scene : (VAX-11/785 Time = 16m + 4.3m) 172

6.2(c) Gallery(4S) Scene : (VAX-11/785 Time = 55m + 6.8m) 173

6.3(a) Office1(NS) Scene : (VAX-11/785 Time = 3m + 3.4m) 174

6.3(b) Office1(1S) Scene : (VAX-11/785 Time = 49m + 5.1m) 175

6.3(c) Office1(2S) Scene : (VAX-11/785 Time = 88m + 6.7m) 176

6.4(a) Office2(NS) Scene : (VAX-11/785 Time = 23m + 5.2m) 177

6.4(b) Office2(1S) Scene : (VAX-11/785 Time = 5.3h + 8.6m) 178

6.4(c) Office2(2S) Scene : (VAX-11/785 Time = 10h + 11.5m) 179

A.1 Image Generation Processor Implementation 195

A.2 Scan Conversion Processor Implementation 199

List of Tables

3.1 VAX-11/785 Run-time Statistics for Figures 3.10-3.13 76

6.1(a) Environment Statistics for Figures 6.1-6.4 181

6.1(b) VAX-11/785 Run-time Statistics for Figures 6.1-6.4 181

6.2 Run-time Profile for the Image Generation Processor 186

Chapter 1

Introduction

One of the major goals of computer graphics continues to be the rapid generation of realistic images from three-dimensional environment descriptions. In some typical computer graphics applications, such as slide making, animation, and advertising, the primary objective is to generate aesthetically pleasing pictures that will attract the attention of the public, while the amount of time and computational power required to produce these images is of secondary importance. For interactive applications, such as computer aided design (CAD) and visual simulation, images have to be created fast enough (e.g., 30 frames per second) to provide the illusion of continuous motion when displayed on a video monitor. A change in some environment parameter, such as viewer position or lighting condition, should result in immediate visual feedback to the user so as to provide the level of interactiveness required for the given application.

Until recently, systems that could provide the computational power necessary for high quality interactive computer graphics have been very expensive due to the high cost of semiconductor memory and the amount of special purpose hardware required to achieve real-time performance. The advent of VLSI technology has made possible the implementation of many useful graphics functions, such as geometrical transformations, clipping, and perspective projection, directly in silicon [1] at substantial reduction in cost, area, and power consumption. In addition to these powerful graphics engines,

the availability of 16-bit/32-bit microprocessors and 256 K-byte memory chips has given rise to a number of low cost graphics workstations that are capable of supporting interactive color display of three dimensional environments [2, 3]. These workstations also implement hidden surface removal and surface shading using a combination of software, bit-slice microprocessors, and special-purpose hardware.

With the increased value and popularity of computer generated imagery, along with the availability of affordable graphics workstations, comes an increasing need to devise efficient algorithms that enhance the realism of the pictures, while maintaining a balance between the amount of computational power required and the time available to produce these images. The motivation for this research is to present new algorithms for realistic image synthesis, as applied to interactive computer graphics applications under limited computing power constraints. This chapter discusses some of the issues involved in producing realistic computer images and presents some early research on image rendering techniques. Chapter 2 discusses recent published work on enchanced ray tracing techniques. Chapter 3 then presents a new approach to efficient image rendering and gives an overview of the algorithms explored in this research. Chapter 4 describes the implementation details of a novel image generation processor, while Chapter 5 describes the implementation of various anti-aliasing scan conversion techniques. Finally, Chapter 6 discusses the results of this research and provides suggestions for further research on efficient image rendering techniques.

1.1. Realism in 3-D Computer Graphics

A number of factors contribute to the overall quality and realism of computer generated imagery. Realism here implies some measure of the subjective difference between an image produced by conventional photo-optical methods and an image generated from a three-dimensional model of the same scene using computer processing techniques. Object modeling, illumination models, and surface shading techniques are of primary importance in trying to synthesize realistic pictures. These models provide the means of describing the geometric and physical properties of a scene, and then simulating the distribution of light energy through the environment. In addition, the simulation of texture, shadows, transparency, reflection, and refraction also enhance the descriptive power of computer generated images, by adding visual effects that are evident in real-world scenes. The reduction of aliasing artifacts caused by the limited resolution of displays also has been shown to be of vital importance in producing superior quality images [4, 5].

1.1.1. Object Modeling

The simplest and most popular approach to object modeling is to approximate arbitrary surfaces using a collection of polygons. An example of a polygonal champagne glass is shown in Figure 1.1. A large body of shading and visible surface algorithms have been developed over the past two decades dealing solely with this surface primitive. However, because of the shortcomings in representing smooth surfaces with faceted clusters of polygons, research has extended to algorithms dealing with

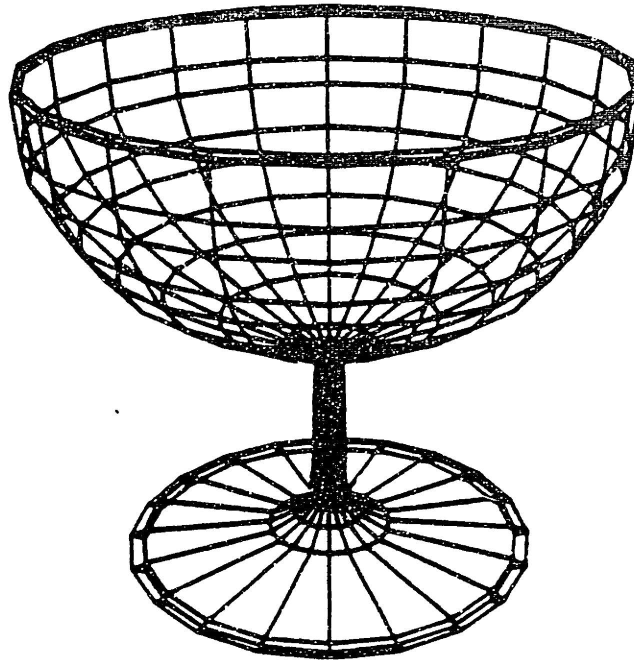


Figure 1.1 Polygonal Champagne Glass

parametric surface descriptions, such as bicubic patches, which allow greater flexibility in representing surface shapes with pleasing properties of continuity and smoothness. Problems in both surface shading and in rendering contour edges inherent in polygon-based algorithms are overcome by this technique, but unfortunately, the mathematics is no longer linear, making the rendering problem more complex.

Stochastic models, such as fractals [6] and particle systems [7], play an important role in modeling complex natural objects, such as terrain, trees, stones, and clouds. Hierarchical representations, which decompose an object space into successively simpler subspaces [8, 9] are also widely used to efficiently describe complex scenes at various levels of complexity and detail. In most applications a combination of these various

modeling techniques are needed to effectively describe an arbitrary scene. Each geometric model type presents certain tradeoffs between simplicity in modeling an object and later rendering it.

1.1.2. Illumination Models

Local illumination models typically consider only point light sources and surface orientation in computing the amount of light that is perceived by the viewer, while ignoring the overall environment in which the surface is placed. Simple lighting models are based on Lambert's cosine law, which states that the intensity perceived by an observer is independent of the observer's position and varies directly with the cosine of the angle between the light direction and the normal to the surface. In addition, some constant value is usually added to the intensity to account for the effects of ambient light on the surface. The net intensity function is given by

$$I = I_a + k_d \sum_{j=1}^{ls} (\mathbf{N} \cdot \mathbf{L}_j)$$

where

I = perceived intensity,
I_a = ambient light intensity,
k_d = diffuse coefficient,
$\mathbf{N}$ = unit surface normal ($|N| = 1$),
$\mathbf{L}_j$ = j^{th} light source direction vector ($|\mathbf{L}_j| = 1$),

A more realistic illumination model was introduced by Bui-Tuong Phong [10], which takes into consideration that for any real surface the light received by the eye is provided in part by the diffuse reflection and part by the specular reflection of the incident light. Thus, for example, if the surface is a perfect mirror, light will only

reach the eye if the surface normal, N, points halfway between the light source direction, L, and the eye direction E. This direction of the surface normal for maximum highlight is given by

$$\mathbf{H} = \frac{(\mathbf{L} + \mathbf{E})}{|\mathbf{L} + \mathbf{E}|},$$

For less than perfect mirrors, the amount of specular reflection detected by the eye decreases as the normal direction moves away from H. Phong measures this separation by using the cosine of the angle between N and H and simulates the degree of sharpness of the highlight by raising this cosine term to some power. The intensity from Phong's illumination model is given by

$$I = I_a + k_d \sum_{j=1}^{ls} (\mathbf{N} \cdot \mathbf{L}_j) + k_s \sum_{j=1}^{ls} (\mathbf{N} \cdot \mathbf{H}_j)^n$$

where

k_s = specular coefficient,
n = shininess of the surface.

Blinn [11] introduced yet a better model, based on theoretical and experimental work by Torrance and Sparrow [12, 13], which assumes that surfaces are composed of a collection of mirror like micro facets oriented in random directions all over the surface. The specular component of the reflected light is assumed to come from only those facets oriented in the maximum highlight direction, H. The diffuse component is assumed to come from multiple reflections between facets and from internal scattering. This model shows noticeable improvement over Phong's model, primarily for non metallic and edge-lit objects. In addition, Blinn presents an efficient computational method that results in approximately equal computational expense as if Phong's model

were used.

But in addition to diffused and specular reflection, the simulation of shadows, reflection, transparency, and refraction are some of the more desirable features in the generation of realistic computer images. To simulate these effects requires that the illumination model consider the overall environment in which a surface is placed, in addition to simply taking into account the light source direction and strength, viewer position, surface orientation, and surface properties. This, of course, introduces serious problems to most image rendering algorithms since the information needed by the illumination model is typically not available or difficult to compute.

The first image rendering algorithm capable of simulating the combined effects of hidden surface removal, shadows, reflection, and refraction was implemented by Whitted [14], which is now referred to as ray tracing. The information needed to compute the intensity at each pixel is stored in a tree of rays extending from the view point to the first surface encountered and from there to all light sources and recursively to other surfaces. A ray tracing visible surface processor creates this ray tree for each pixel of the display and then passes it to a shader for the final intensity calculation. Whitted's global illumination model is given by

$$I = I_a + k_d \sum_{j=1}^{ls} (\mathbf{N} \cdot \mathbf{L}_j) + k_r I_r + k_t I_t$$

where

k_r = reflection coefficient,
I_r = reflected light intensity,
k_t = transmission coefficient,
I_t = transmitted light intensity.

At each node in the ray tree, the above model is applied to combine the local intensity contributions from ambient and diffuse illumination, and recursively adds the contributions from the reflected and transmitted components.

1.1.3. Polygon Shading

Surface shading plays an important role in the visual appearance of synthetic objects. The simplest polygon shading technique calculates one intensity value for each polygon and then uses this value for all interior points (constant shading). This intensity value is usually determined using Lambert's law, which simulates ideal diffuse reflection and produces a reasonable approximation to a dull, matte surface.

An improved technique for shading curved objects approximated by collections of polygons was developed by Henri Gouraud [15]. This shading technique, referred to as smooth shading, is also based on Lambert's law but calculates a shading value at each polygon vertex. The shade at any point inside the polygon is determined by two successive linear interpolations of the shading values of its neighboring vertices. First, shading values along the left and right polygon edges intersected by a given scan-line are interpolated from the known shading values at the endpoints of the edges. Shading values for the interior points of the polygon along the current scan-line are then determined by linearly interpolating these two edge shading values. This very simple method of interpolating intensities gives a continuous gradation of shade over the entire surface, which in most cases restores the smooth appearance of the object. An example of faceted and Gouraud shading is shown in Figure 1.2.

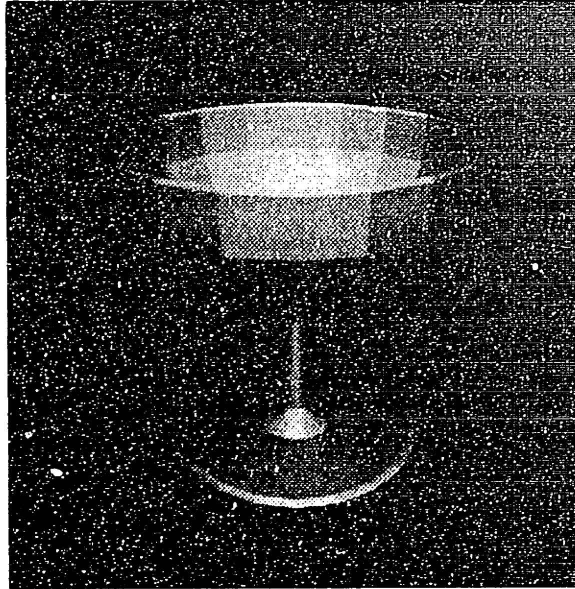


Figure 1.2(a) Faceted Shaded Champagne Glass

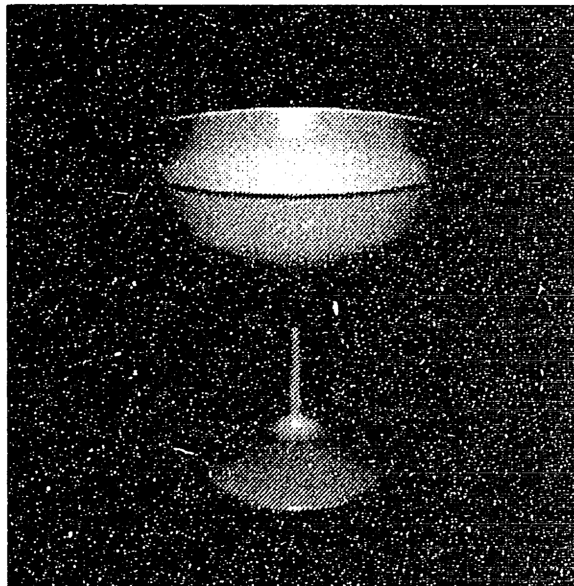


Figure 1.2(b) Gouraud Shaded Champagne Glass

The major drawback to Gouraud shading is that highlights cannot be adequately represented. For example, if an object should have a highlight at any point but a vertex, this shading technique will misplace the highlight, or miss it altogether, since the shading values at interior points are interpolated from neighboring vertex values. In addition, the Mach Band effect is visible where the slope of the shading function changes across an edge, as a result of our retina performing some kind of two-dimensional filtering operation on the shading function, which attenuates low spatial frequencies and amplifies high spatial frequencies. While Gouraud's linear interpolation technique produces a shading that is continuous in value across an edge, its derivative is not continuous.

Bui-Toung Phong [10] introduced a more sophisticated shading function model to better simulate specular reflection, as part of a technique for improving the appearance of curved glossy surfaces. Given a collection of polygons describing the surface to be rendered, surface normals are computed at each vertex. To calculate the shade at any point inside a polygon, the orientation of the surface at the given point is approximated by linearly interpolating the normals at the surrounding vertices. This method requires considerably more computation than Gouraud's shading technique, since the three components of the surface normal have to be interpolated and then normalized, but provides improved depiction of curved, glossy surfaces.

1.2. Conventional Image Rendering Techniques

Sutherland *et al.* [16] have written a highly informative survey of ten polygon-based visible surface algorithms. Two underlying principles shared by the algorithms is that they all sort the polygons that are potentially visible in a scene, and all take advantage of the coherence properties of the environment in order to speed up and simplify the rendering process. The term *coherence* is used to describe the extent to which the environment is locally constant. A classification of the various algorithms is performed according to the order in which they sort the polygons in a scene. Three major classes are formed; those that perform hidden-surface elimination in object-space; those that perform calculations in image-space; and those that work partly in each, the list-priority algorithms. Object-space algorithms perform their calculations at high precision and aim to compute "exactly" what the image should be. Image-space algorithms usually perform their calculation at the resolution of display screen.

One of the simplest methods used to create frame buffer images of three-dimensional scenes described by polygonal objects is the painter's algorithm [17]. Polygons to be displayed are assigned priorities, based on their image space distance from the screen, and then sorted. Rendering a scene is then accomplished by scan converting each polygon in order into a frame buffer, starting with the lowest priority one (i.e., the one furthest from the eye.) Polygons with higher priority will over-write those of lower priority, thereby performing hidden surface elimination. Translucency is simulated by only modifying the intensity values of lower priority polygons in the buffer rather than completely overwriting them. While there is clearly a considerable

overhead in writing pixel values into the buffer that may eventually be overwritten, most graphics workstations provide a polygon filling function in hardware that allows the painter's algorithm to work sufficiently fast for many interactive applications.

Another popular class of visible surface algorithms are referred to as scan-line algorithms, originally devised by Watkins [18], Bouknight [19,20], and Wylie et al. [21], for drawing polygonal objects. These three algorithms operate in image space and solve the visible surface problem one scan-line at a time. They all begin by performing a vertical sort of all polygon edges on the screen according to their uppermost vertices. Then for each scan-line, the various potentially visible polygon segments on that scan-line are sorted according to their horizontal displacements. Finally, a depth sort is performed at each polygon edge to determine the visible polygon segment for that horizontal span. The three approaches differ only in their use of various image-space coherence properties to make the calculations incremental rather than absolute. Of the three methods, Watkins' is the most economical because it uses scan-line coherence to optimize the horizontal sort and employs a logarithmic depth search. More recently, scan-line algorithms have been extended to render objects modeled by parametric surfaces [22].

Another simple image space rendering algorithm is the Z-buffer algorithm, originally developed by Catmull [23] to render objects modeled by bivariate parametric surface patches. This algorithm works by recursively subdividing each surface patch into smaller patches until its projection covers only one picture element (pixel) on the screen. At this stage, the intensity and visibility calculations are performed for the

corresponding picture element. The Z-buffer is a large random access memory, equal in size to the screen resolution, which holds the intensity of the image and the depth of the current visible surface at each picture element. Visibility at each pixel is simply determined by comparing the depth of the given patch fragment with that of the fragment currently occupying the corresponding pixel position. If greater, the new patch fragment is ignored, otherwise the picture element is updated with the new intensity and depth. A major problem with the Z-buffer algorithm is that anti-aliasing cannot be performed since image fragments arrive in arbitrary order. More recently, Carpenter [24] introduced the A-buffer hidden surface algorithm, which is basically a descendent of the Z-buffer, capable of producing good quality anti-aliased images at moderate cost.

Other types of image rendering algorithms include the area-subdivision algorithms developed originally by Warnock [25] and, more recently, by Weiler and Atherton [26] to solve the visible surface problem for polygonal objects. These algorithms employ the "divide-and-conquer" approach to determining the visible surface fragment in each area of the final image. Areas of the image are examined to decide which polygon or polygons are visible within the area of interest. If possible to determine, the appropriate polygons are displayed. Otherwise, the area is subdivided into smaller areas and the decision logic is recursively applied to each of the smaller areas. Subdivision terminates when the resolution of the display has been reached. Warnock's algorithm subdivides rectangular areas of the screen into four equal squares, while the Weiler-Atherton algorithm subdivides the screen area along polygon boundaries. This later

technique greatly reduces the total number of subdivision steps that have to be performed, but requires more work to perform each subdivision step.

1.3. Ray Tracing

Ray tracing is considered to be the most elegant technique for rendering high quality computer generated images of three-dimensional environments. As illustrated in Figure 1.3, the image is generated by back-tracking light rays from the view point through each pixel on the image plane and into the object space. Each ray traced through the scene is tested for possible intersection with every object in the scene. Whenever a ray intersects more than one object, the nearest point of intersection is the visible one. The intensity at this point is determined by tracing rays to all light sources, checking for possible shadowing by another object in the scene, and by recursively tracing the reflected and refracted rays.

Although ray tracing was first suggested by Appel [27] in 1967 and later used by MAGI [28] to solve the hidden-surface problem, it was not until recently that Kay [29] and Whitted [14] implemented this technique for general image rendering purposes. The ray tracing method combines the effects of hidden surfaces, shadows, reflection, and refraction, that are difficult or impossible to achieve by other techniques. In addition, ray tracing algorithms are relatively simple to program as compared to most other image rendering techniques. Conventional graphics operations such as clipping and perspective projection are automatically encoded into the geometry of the light rays. A variety of geometric primitives are easily handled since all that is required is a dif-

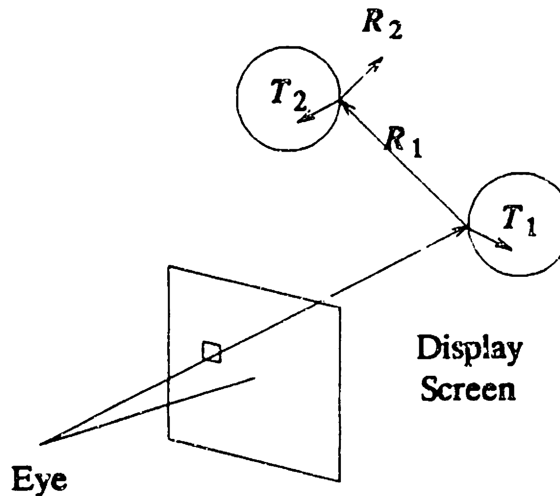


Figure 1.3 Geometry of Ray Tracing

ferent ray-surface intersection processor for each object type. To date, ray tracing has been applied to planar and quadric surfaces [28, 14, 30], bicubic surface patches [14, 30, 31], algebraic surfaces [32], procedurally defined objects [33], and volume densities [34].

Although ray tracing provides a powerful, yet simple method to create high quality pictures, the amount of computation time required to generate an image (in the order of hours) makes it unsuitable for interactive applications. Intersection calculations between a ray and a surface are very floating point intensive since, for each ray traced, all objects in the scene must be checked for possible intersection. It is difficult to take advantage of spatial coherence, as done in most other image rendering algorithms, because the shapes of reflections and refractions from curved surfaces are very complex. Of course, this same lack of coherence allows for unbounded (brute force)

parallelism.

Point sampling in classical ray tracing is also subject to aliasing problems, since rays are constrained to only sample the point in the center of the pixel. There is not enough information associated with a ray to calculate what else is visible in the region surrounding the sample point. The only way to anti-alias with standard ray tracing is to oversample the image space, but this substantially increases the number of rays that have to be traced.

1.4. Motivation for New Research

The underlying motivation for new research is the continuing demand to produce realistic computer generated images, while at the same time, reducing the amount of computing time needed generate them. Interactive computer aided applications, such as office design, interior design, and architectural design desire the ability to create an environment, specify lighting conditions, and then be able to "walk-through" this environment in "real-time" and see the world they have created. Of course, the degree of realism possible will depend mainly on the complexity of the scene, the sophistication of geometric and lighting models used, the rendering algorithm, and on the computational power available to perform the necessary tasks in the allotted time. This type of application is quite different from that of producing computer generated animated sequences, frame-by-frame in non real-time, using the most sophisticated image rendering techniques, and then playing it back at 30 frames per second.

Although computer graphics workstations are available today that can support real-time image rendering of moderately complex scenes, the pictures produced clearly look synthetic primarily because of the simple illumination models used for surface shading calculations. In most graphics systems, objects are represented with polygons and shaded either with a constant value or with Gouraud's interpolated shading technique, with intensity values computed using Lambert's law. The choice of polygonal objects permits using any one of the existing visible surface algorithms, which when supported by various geometric engines (e.g., 4 x 4 matrix multiplier, clipping unit, and scan-conversion processor) provide the necessary processing power for real-time performance. Unfortunately, many of the visual cues to the shape and appearance of objects, such as shadows, reflection, and refraction are not simulated because existing algorithms that are capable of simulating these desirable effects are too computationally expensive to be incorporated into these graphics workstations.

The purpose of this thesis is to present new image rendering algorithms that can support interactive computer graphics applications on small scale workstations. Considering the current state of VLSI technology, and the amount of computational power required to implement some of the available rendering algorithms, it is inconceivable to expect that one could generate real-time sequences of ray traced scenes on a desk-top graphics workstation in the very near future. The goal of this thesis is to produce realistic images that approximate the quality of images now synthesized by ray tracing and other high quality rendering algorithms, but at considerable reduction in computation expense. Then, coupled with a moderate cost supporting architecture, it is expected

that these new algorithms will provide real-time performance for a wide range of interactive applications.

In the following chapters, several new image rendering algorithms will be presented and results demonstrated by way of computer simulation. Results then will be analyzed and compared with existing ray tracing algorithms.

Chapter 2

Enhanced Ray Tracing Techniques

2.1. Introduction

This chapter discusses various techniques for improving the performance and capabilities of ray tracing algorithms and presents recent developments in extensions to the basic ray tracing method.

2.2. The Bottleneck in Ray Tracing

Since the implementation of ray tracing image rendering algorithms, statistical studies of these programs have shown that most of the execution time involves computation of ray-surface intersections. Whitted [14] reported this time to be between 75 percent for simple scenes and over 95 percent of the total time for more complicated scenes. Of course, this is to be expected since each generated ray must be checked for possible intersection with every object in the scene to determine the visible surface, and to check for shadowing effects between objects. In fact, Kajiya [35] has demonstrated that the number of intersection calculations is linear with respect to the product of the number of rays traced and the number of objects in the scene. For this reason, most attempts at improving the performance of ray tracing programs have concentrated on reducing the number of ray-surface intersection calculations that have to be per-

formed in rendering an image.

Using a hierarchical decomposing technique similar to one described by Clark [8], Whitted added bounding spheres to each object (polygon or bicubic path) in the scene. Obviously, the sphere was chosen because it is the simplest geometric shape to intersect with a ray. Intersection calculations begin by checking the bounding sphere and then proceeding to the enclosed object only if the ray intersected its bounding volume. For complex scenes, Rubin and Whitted [9] suggest a hierarchical decomposition of the entire object space into a tree of enclosing volumes, where each volume may contain other subvolumes or the actual displayable object. The bounding volumes are selected as parallelepipeds oriented to minimize their size. Ray-surface intersection calculations are performed by testing the outermost enclosing volume first and testing subvolumes only if the ray intersects the outer volume. More recently, Weghorst *et al.* [36] presented procedures for the selection of the bounding volume as either a sphere, a rectangular parallelepiped, or a cylinder. The selection process considers such factors as the cost of testing the bounding volume for intersection with a ray, and the volume of the bounding shape. As expected, the computational times for rendering ray traced images using any one of the methods mentioned above were significantly reduced. However, it should be emphasized that creating the hierarchical database is a non-trivial operation.

Another approach to reducing the number of ray-surface intersection calculations is to form a cellular decomposition of the entire object space, keeping track of which surfaces and light sources are within each cell [37, 38, 39]. As a ray is traced through

the environment, only those surfaces within a particular cell, which has been pierced by the ray, need be tested for intersections. If one or more ray-surface intersections occur within a cell, the closest one to the eye is chosen as the visible one. If no intersections are found the ray passes through that cell, enters its neighboring cell along the path of the ray, and the search for a visible surface continues. Thus, the ray-surface intersection problem is reduced from considering all objects in a scene to considering only those objects within cells along the path of each ray.

Yet another approach, applicable in ray casting rendering algorithms where rays are only traced to one level for hidden surface elimination, is to enclose each surface with a two-dimensional bounding box in image space [40]. As the image is rendered in scan line fashion, surfaces become active or inactive depending on whether the current pixel location is inside or outside the surface's bounding box. Only those surfaces which are active at a given raster location need be intersected by the ray to determine the visible one.

Although considerable computational improvements to ray tracing algorithms has resulted from these clever attempts to exploit object and image space coherence, the improvement has not been sufficient to permit real-time performance. In fact, the motivation behind investigating these various techniques was to reduce the execution time of current ray tracing algorithms, and to provide a manageable way to render highly complex scenes. Further exploitation of object and image coherence is needed to further improve the computational expense of ray tracing.

2.3. Anti-Aliasing Difficulties

A major problem with ray traced images is aliasing artifacts caused by its point sampling approach. The only way to anti-alias within standard ray tracing is to over-sample the image space with more rays, thereby increasing the rendering time by the same oversampling factor. Whitted proposed adaptive oversampling around areas where aliasing is most apparent to the viewer, this being along the silhouette of an object where abrupt changes in intensity occur, at locations where small objects disappear between sampling points, and during texture mapping onto a surface. Problems with this approach still exist for small objects that fall between sample points and for rays that are reflected or refracted by other objects. Alternate solutions to the aliasing problems of ray tracing recently have been investigated and are discussed in the following sections.

2.4. Distributed Ray Tracing

Cook, Porter, and Carpenter [41] recently presented a "distributed" ray tracing technique, which in addition to performing spatial anti-aliasing, simulates the effects of penumbras, motion blur, depth-of-field, and the entire shading function. The underlying principle used by the authors is that by distributing the directions of the rays of a super-sampled ray tracing procedure according to the analytic function they sample, (i.e., pixel area, lens area, reflectance and transmittance directions, light source area, and time) ray tracing can incorporate fuzzy phenomena, and need not be restricted to spatial sampling. The key to this technique is that no additional rays are needed

beyond those required to oversample the space, and provides correct and easy solutions to some previously difficult and unsolved problems. The types of fuzzy phenomena that can be simulated with this technique include blurred reflection and transparency, in addition to motion blur, depth of field, and penumbras. Some of the most impressive computer generated pictures to date, illustrating these simulated effects, have been produced by this distributed ray tracing technique.

More recently, Lee, Redner, and Uselton [42] formulated a relationship between the number of sample rays chosen to approximate the image function integral in distributed ray tracing, and the quality of this estimate. They show that the number of samples required do not depend directly on the number of dimensions being sampled, but only on the variance of the multi-dimensional image function. Thus, sampling an additional dimension does not imply an increase in the number of samples needed. A statistical technique is used to reduce sampling by selecting random samples until the variation between the first samples is below some threshold. Typically, eight samples per pixel are sufficient to provide good results, but this number may reach as high as 96 (where this was the maximum number of samples allowed in their experiments) in complicated shading areas. Excellent results are shown demonstrating the effects of variable degrees of surface smoothness, penumbras, and, of course, anti-aliasing. It is suggested that this technique also can be extended to additional dimensions to model other effects, such as motion blur, depth of field, wavelength sampling for improved color modeling, and wavelength dependent effects such as refraction.

2.5. Ray Tracing with Cones

The work by John Amanatides [43] provides a new approach to the ray tracing methodology, which attempts to solve some fundamental problems with conventional ray tracing. The definition of a ray is extended into a circular pyramid or "cone" by including the angle of spread and virtual origin of the ray, along with its conventional origin and direction. Figure 2.1 illustrates this new definition. Note that the virtual origin is defined as the distance from the apex of the cone to the origin, and is non zero for reflected or refracted cones.

The advantages of this approach includes a better method of anti-aliasing, a way of calculating fuzzy shadows and dull reflections, a method of calculating the correct level of detail in a procedural model and texture map, and finally, a procedure for reducing the number of intersection calculations. One disadvantage of this approach is that the intersection calculation between a cone and an object is rather complex. In fact, intersection calculations are only described for spheres, planes, and polygons.

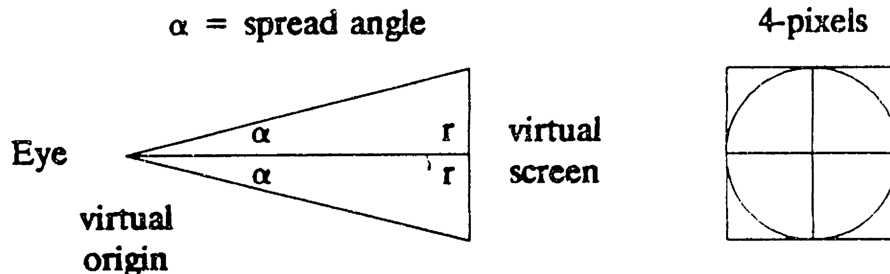


Figure 2.1 Geometry of Amanatides's Cone

Each calculation consists of a fast in/out test and a more complicated area intersection approximation.

Anti-aliasing is performed using a single cone per pixel and maintaining a sorted list of the eight closest objects intersected by the ray. The final pixel intensity is computed by blending the contributions from the various object fragments in the list. Fuzzy shadows are approximated by considering spherical light sources, instead of conventional point sources, sending a cone from each intersection point to the light source with radii equal to the size of the light source, and calculating how much of the light is blocked by intervening objects. Fuzzy reflections and translucency are produced by broadening the reflected and transmitted cones, which results in less detailed reflections and refractions.

Examples of various images generated using this improved technique are given, illustrating anti-aliasing, fuzzy shadows, are dull reflections. Each of the simple pictures shown took approximately 50 minutes to compute (at unknown resolution) on a VAX-11/780. A reduction in the number of intersection calculations required for ray casting is suggested by recursively firing cones of various sizes at the screen and performing a Warnock style culling process [25]. Extensions to this approach are necessary before it can be applied to ray tracing in general.

2.6. Beam Tracing Polygonal Environments

Heckbert and Hanrahan [44] recently presented an algorithm that traces "beams" of light through a scene, rather than individual rays as done in standard ray tracing. The underlying idea behind this technique is that for planar polygonal surfaces, reflections are linear transformations and refractions are often approximateiy so. Thus, in rendering a scene composed of polygonal objects, reflections and refractions can be computed in parallel for an entire surface using linear transformation techniques, rather than having to trace individual light rays. Exploiting this spatial coherence of polygonal environments reduces the number of intersection calculations, and permits using conventional incremental image rendering techniques to draw homogeneous regions of the image.

To render a scene, a recursive beam tracer is first used to find all visible polygons within an arbitrary two dimensional region, starting with the viewing pyramid and tracing each planar reflection and/or refraction. For each visible face on the screen, the beam tracer creates an object space data structure, called a "beam tree", similar to the ray tree in standard ray tracing. Each link of the beam tree represents a polygonal cone of light and tree nodes represent visible surface fragments intersected by these cones. But unlike a link in a ray tree, which always terminates on a single surface, each beam link may intersect many surfaces, as detected along the beam axis. Once the beam tree is created for a given screen space polygon, final intensities are computed by scan converting the polygon and all its associated reflected and retracted fragments. Both faceted and Gouraud shading techniques are supported.

Although the efficiency of beam tracing versus conventional ray tracing increases linearly with resolution, it is also more complicated and may not always be worth the extra expense. Intersection calculations between a beam and a polygon can become quite complex, since in general, the beam is concave and may contain holes. The expected improvement of beam tracing over ray tracing depends more on the intrinsic coherence of the imaged scene rather than resolution. The greater this coherence the more rays will be traced in parallel. Thus, beam tracing is most efficient for images having large homogeneous regions.

Chapter 3

New Techniques for Efficient Image Rendering

3.1. Introduction

As discussed briefly in Chapter 1, the motivation for this thesis was to explore new methods of producing realistic synthetic images under limited computing power constraints. A number of conventional image rendering techniques with different performance and image quality characteristics have been presented, including various extensions to the computational efficiency and capabilities of ray tracing. At one extreme we have the efficient object and image space rendering algorithms, which when supported with appropriate special purpose hardware, can produce moderate quality images for many different types of interactive applications. At another extreme we have the enhanced ray tracing algorithms capable of generating superior images that resemble the quality of pictures produced by conventional photographic techniques, but at considerable computational time and expense. A desirable characteristic for a new image rendering algorithm would be for it to compromise between these two extremes and generate realistic images at moderate expense.

The exploitation of both object and image space coherence is essential in the implementation of an efficient rendering algorithm. This fact is clearly evident by comparing, for example, the rendering times of a polygonal scene generated by a scan-

line algorithm and a ray tracing algorithm. The scan-line algorithm wins out by far because it exploits image space coherence properties of polygons to make all geometric and intensity calculations incremental, whereas a standard ray tracer computes each pixel intensity independently. On the other hand, the simulation of reflection and refraction effects is relatively straightforward in ray tracing, but typically not considered in conventional scan-line algorithms.

3.2. A New Approach to Image Synthesis

This thesis presents a new class of image rendering algorithms that combine the computational advantages of object and image space rendering algorithms with the benefits of the ray tracing methodology. Computational efficiency is achieved by considering only polygonal environments and using conventional visible surface algorithms that can exploit the coherence properties of these environments. Two such algorithms are the list-priority and scan-line algorithms, which are known to provide real-time performance with moderate hardware support. By considering only polygonal environments, linear transformation techniques can be used to simulate the effects of planar reflection and approximate the effects of planar refraction, inherent in conventional ray tracing algorithms. This linear mapping approach is equivalent to tracing multiple rays in parallel through the environment to determine reflections and refractions in planar surfaces. Shadows can be efficiently produced by the method of shadow projection, whereby entire shadow areas are computed by projecting objects between a light source and a given surface. Finally, efficient image space rendering techniques can be

used to scan convert polygonal regions of the image and perform spatial anti-aliasing.

The rendering algorithm explored in this research is based on an extension to the list-priority algorithm of Newell, Newell, and Sancha [17]. Their algorithm performs hidden surface removal and simulates linear transparency effects for scenes composed of polygonal faces. The new algorithm adds the simulation of non-linear transparency, reflection, refraction, and shadows, in addition to anti-aliased faceted, Gouraud, or Phong shaded polygon scan conversion. While this new approach simulates the desirable global illumination effects typically found only in ray tracing, it also can provide real-time performance with only moderate hardware support. In fact, an implementation of this new algorithm in the C programming language, running on an IBM-PC/XT provides a surprising level of interactive performance. For example, the image shown in Figure 3.1(a) took 2.4 minutes to generate at a resolution of 256 × 240 pixels (2 seconds to create a polygon display list for the image, with color and shadow information, and 2:20 to scan convert this list using an anti-aliasing Gouraud polygon tiler), while the second image (Figure 3.1(b)) took 74 minutes to generated with a conventional ray tracer. Furthermore, using a simple graphics display processor on the IBM-PC/XT to scan convert the generated polygon display list, the total rendering time for the new algorithm reduces to 2 seconds.

Real-time performance is expected for moderately complex environments, provided that the algorithm is supported with a suitable graphics engine, consisting mainly of a matrix multiplier, general polygon clipper, and polygon tiler. In addition, the underlying ideas behind this new rendering technique are easily extended to scan-line

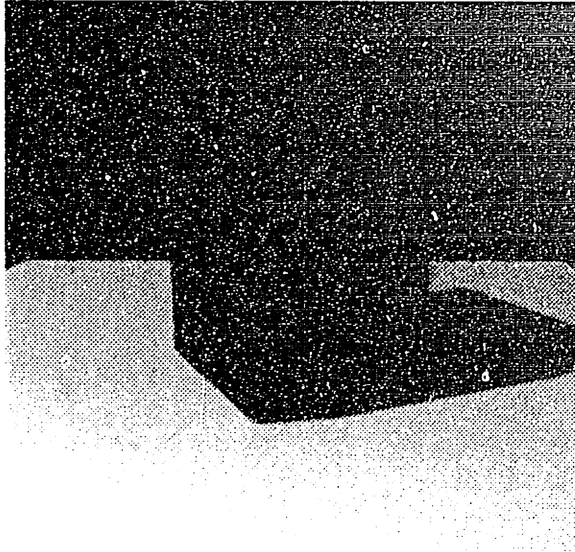


Figure 3.1(a) Example of New Algorithm

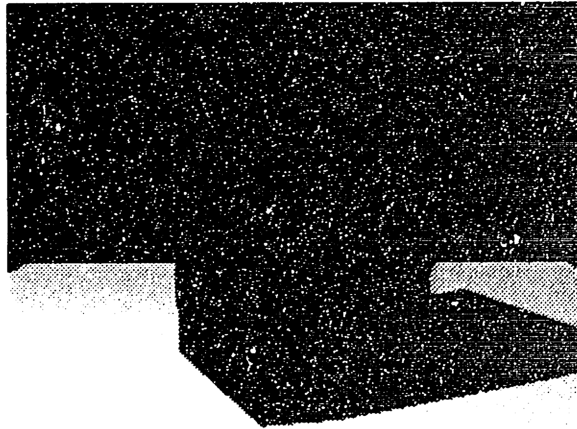


Figure 3.1(b) Example of Ray Tracing

algorithms.

3.3. Outline of The Algorithm

A block diagram of the extended list-priority algorithm is depicted in Figure 3.2, showing its two major components. The Image Generation Processor accepts a three-dimensional polygonal scene description and produces a depth sorted polygon list describing the two-dimensional projection of the image for the specified viewing conditions. Associated with each screen space polygon is a list of shadow polygons that have been cast onto its surface through shadow projections, and a tree of face fragments that have been mapped onto the polygon's surface through multiple reflections and refractions. Each polygon fragment generated is constrained to be convex, and therefore, can be scan converted using any simple polygon tiler.

Final hidden surface removal and polygon scan conversion is performed by the Scan Conversion Processor. The simplest rendering method uses a painter's algorithm, whereby polygons are individually scan converted into a frame buffer in back-to-front order, performing hidden surface elimination by its overwriting principle. Several anti-aliasing polygon tilers are also implemented, which scan convert either faceted, Gouraud, or Phong type polygons into a frame buffer in front-to-back order, performing hidden surface removal and intensity blending.

This new rendering algorithm is similar in principle to a technique presented by Heckbert and Hanrahan [44] for tracing "beams" of light through scenes described by planar polygonal objects. They noted that unlike the general case of a beam reflecting

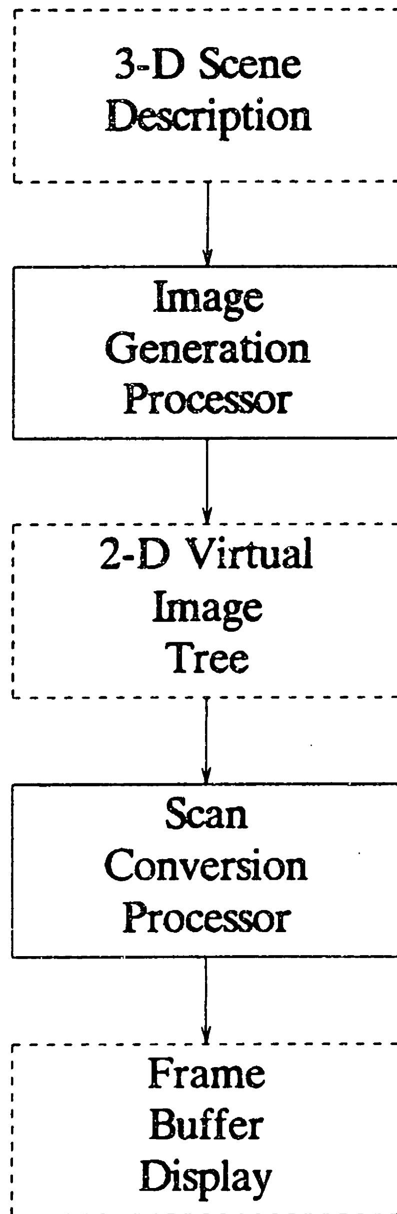


Figure 3.2 Block Diagram of the New Image Rendering Process

or refracting from a curved surface, beams formed at planar boundaries can be approximated by pyramidal cones. The main difference between beam tracing and the rendering algorithm presented in this thesis lies in the method used to perform hidden surface removal, and in the technique used to render the final image. In beam tracing, visible surfaces are determined by intersecting a beam with each surface in a front-to-back sorted polygon list, essentially performing the Weiler-Atherton [26] hidden surface removal algorithm for each beam. After every intersection, the polygon found is subtracted from the beam before proceeding through the depth ordered list to ensure that no other polygon in the list is visible within the area of the current visible polygon. The main problem with their technique is that the shape of the beams, and subsequent visible polygon fragments, become very irregular after only a few intersections. The two-dimensional set operators used in the intersection calculations must be able to handle concave beams and polygons containing holes. In addition, the polygon tiler used in rendering the final image also must be capable of supporting these complex shaped polygons, unless these polygons are further processed to eliminate concavity and holes.

These inherent problems with beam tracing are eliminated in the new algorithm by separating the task of *visible* surface calculations from that of *hidden* surface elimination. The Image Generation Processor computes all potentially visible surfaces within the field-of-view, including surface shadow polygons and reflected/refracted components, without checking for possible obstruction between surfaces. Hidden surface elimination is performed during the polygon scan conversion phase using any con-

ventional image space rendering algorithm. This of course means that in some cases the Image Generation Processor will compute surfaces that will end up being obscured in the final image. The main advantage of this two-stage approach is that visible surface calculations are relatively simple, even after a reflection or refraction mapping operation, since the shape of the polygonal beams are always convex and without holes. In addition, restricting output polygons to be well formed (i.e., convex and unfragmented) simplifies the final stage of polygon scan conversion and hidden surface elimination.

Other advantages of this new rendering algorithm over beam tracing include the addition of shadow generation and several efficient anti-aliasing scan conversion methods. The simulation of shadows and anti-aliasing were proposed by Heckbert and Hanrahan as possible extensions to beam tracing but never reported to have been implemented. These desirable features are incorporated in the extended list-priority rendering algorithm, along with the simulation of planar reflection and refraction. The relative simplicity of this new algorithm permits straight forward implementation on existing graphics workstations, and suggests future hardware implementation for real-time applications.

3.4. Image Generation Processor

The purpose of the image generation process is to create a two-dimensional virtual image description given a three-dimensional environment composed of polygonal objects and various light sources. This virtual image description consists of a sorted list

of screen space polygons representing all potentially visible polygons within the specified field of view. In addition, each screen space polygon may contain a list of surface detail polygons specifying shadow areas, and a tree of face fragments representing reflected and refracted components. This polygon list can be used directly as input to any graphics display system supporting convex polygon tiling, or further processed by a polygon scan conversion processor to produce high quality anti-aliased images.

3.4.1. Recursive Visible Surface Processor

The main component of the image generation process is a recursive visible surface processor responsible for finding all potentially visible polygons within an arbitrary three-dimensional clipping volume. An outline of this polygonal visible surface algorithm is shown in Figure 3.3. The procedure begins with the standard truncated viewing pyramid as the initial clipping volume. The visible surface processor transforms all objects in the scene to this eye coordinate system, clips all objects to the current clipping volume, eliminates back-facing polygons, and depth sorts the remaining polygon list. The resulting sorted polygon list describes all polygons within the specified field of view.

To find possible reflected or refracted intensity contributions on a given surface, an appropriate transformation matrix is computed to map the entire scene into the virtual reflected or refracted coordinate system of the given surface. This matrix is then combined with the current transformation matrix (CTM) to establish a new composite transformation matrix, and a new clipping volume is defined by translating the two-

```
poly_trace (polygon, depth, ...)
{
        Transform (all objects using the CTM);
        Clip (polygons in scene to CVV);
        Sort (scene_polygon_list);  /* BTF or FTB */

        For each polygon in scene_polygon_list
        {
                set_view_volume (scene_poly);
                If sorting_BTF
                        render (scene_poly); /* amb+dif+spec */
                        render_shadow_polygons (scene_poly);
                If recurse_further (scene_poly, depth)
                {
                        If reflective (scene_poly)
                                make_reflection_matrix (scene_poly);
                                poly_trace (scene_poly, depth+1, ...);

                        If refractive (scene_poly)
                        {
                                make_refraction_matrix (scene_poly);
                                poly_trace (scene_poly, depth+1, ...);
                }
                If sorting_FTB
                        render_shadow_polygons (scene_poly);
                        render (scene_poly); /* amb+dif+spec */
        }
}
```

Figure 3.3 Recursive Visible Surface Algorithm

dimensional projection of the given polygon along the z-axis. (The polygon itself forms the near clipping plane, while the far clipping plane is taken to be the one defined by the standard truncated viewing pyramid.) Then, the visible surface processor is again called to transform the entire scene to this new virtual coordinate system, clip all objects to the active clipping volume, and depth sort the remaining polygons. By

recursively invoking the visible surface processor for each reflective or refractive polygon, a *virtual image tree* is created describing the two-dimensional projection of a scene onto a viewing plane, along with any reflected, refracted, and shadow components. A simple example of a cube resting on a reflective surface is illustrated in Figure 3.4, along with its corresponding virtual image tree.

The virtual image tree associated with any given screen space polygon is somewhat analogous to multiple *ray trees* in standard ray tracing. Links in a ray tree correspond to single light rays traced through the environment, and nodes represent the closest ray-surface intersection. In a virtual image tree, links correspond to convex polygonal beams and nodes represent polygonal surface fragments intersected by these beams. But unlike links in a ray tree which terminate on a single node, the links in a virtual image tree may terminate on multiple nodes, representing all the potentially visible surfaces found within the volume swept out by the beam. Thus, in effect, tracing a single polygonal beam through the environment corresponds to tracing multiple rays in parallel. Rather than tracing a single ray and redirecting it after intersection with a reflective or refractive surface, the visible surface processor transforms the entire scene into the virtual reflected or refracted coordinate system of the surface, changes the cross section of the beam appropriately, and proceeds with visible surface calculations from the same view point. Given that the scene is composed solely of well formed convex polygons, the resulting polygonal beams and computed reflected, refracted, or shadow fragments will always be convex and without holes.

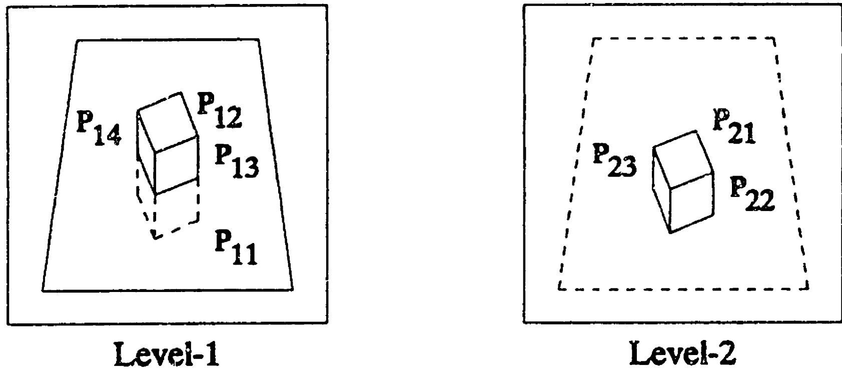


Figure 3.4(a) Cube on a Mirror

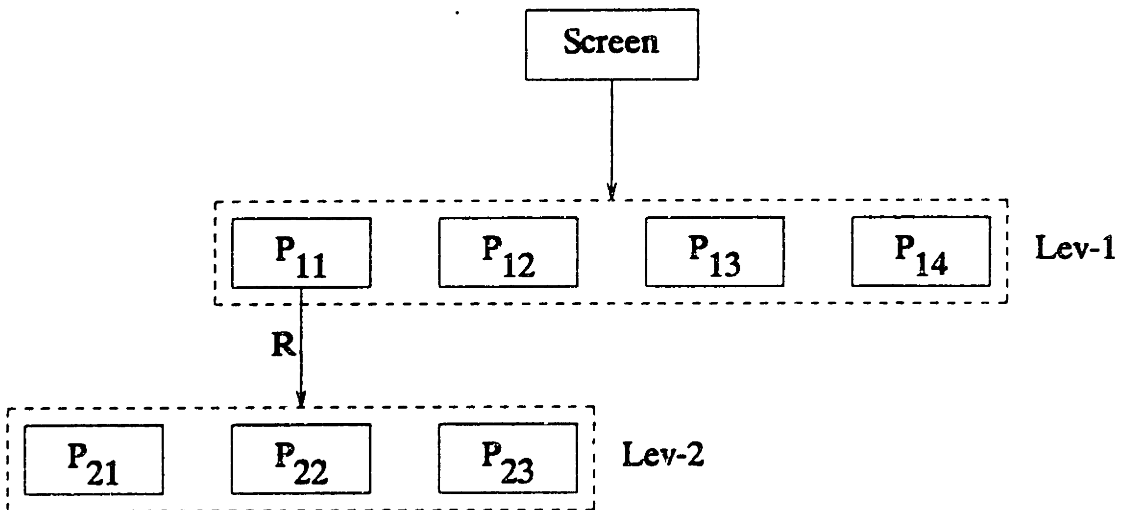


Figure 3.4(b) Virtual Image Tree

3.4.2. Reflection and Refraction Mapping

One of the key operations performed by the recursive visible surface processor involves calculating a transformation matrix for each reflective or refractive planar surface. Given this matrix, the entire scene is transformed into a new virtual coordinate system in which visible surface calculations continue from the original viewpoint to determine the reflected or refractive components for the given surface. The volume swept out as the surface is translated along the z-axis defines the region in this new coordinate system where transformed objects must lie to be potentially visible. Figure 3.5 illustrates this mapping operation, and corresponding bounding volume, for the mirror surface shown in Figure 3.4.

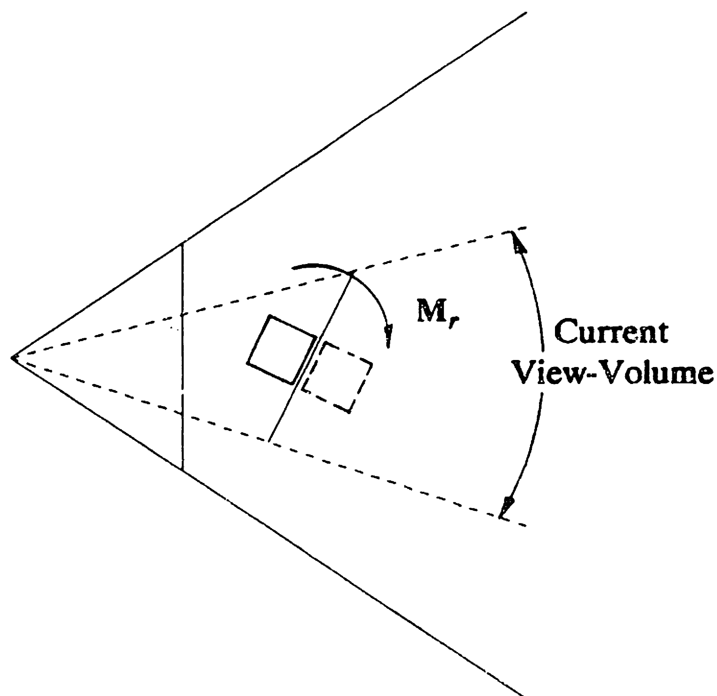


Figure 3.5 Reflection Mapping Example

Since reflection in a plane is equivalent to a linear mapping between each point and its mirror image, it can be represented by a 4×4 homogeneous matrix. Refraction by a plane, in general, is not a linear transformation since the angle of refraction is a non-linear function of the incident angle. However, there are two limiting situations under which planar refraction is linear [44]. The simplest case is under an orthographic projection, where the eye is located at infinity, resulting in all sight rays striking a plane at the same incident angle. Refraction in this case corresponds to a translation parallel to the plane and, therefore, can be expressed as a linear transformation. Another case where refraction by a plane can be expressed as a linear transformation is for rays at near perpendicular incidence, known as paraxial rays. For such rays, Snell's refraction law is approximately linear and planar refraction can be approximated by a scaling transformation perpendicular to the plane.

For reasons of implementation simplicity, planar refractions in perspective view are calculated using the paraxial approximation. In all cases, a linear transformation matrix is computed assuming that the angle subtended by a refractive surface is small, and that the incident angle of all sight rays impinging on the surface is also small. This, of course, implies that for most scenes, refractions will not be physically correct. However, for many applications, such as architectural design and interior design, the simulation of correct refraction is of secondary importance, as compared to the simulation of reflections and shadows. In addition, most people are unable to notice errors resulting from these linear approximations to refraction. A more general treatment of planar refraction is discussed at the end of Chapter 6.

3.4.3. Shadow Generation

In standard ray tracing, shadows are computed by tracing rays from each intersection point to every light source and checking for possible intersection with any surface in the scene. In the current algorithm, shadow areas are computed for a given target polygon by projecting all objects within the volume defined by a light source and the target surface. This approach is completely opposite from the ray tracing method and essentially performs shadow calculations once for a given surface, rather than once for each ray intersected with the surface. These computed surface shadow polygons are included along with the corresponding target polygon in the virtual image tree, and later used by the image rendering process to modify the intensity of the visible surface areas.

3.4.4. Virtual Image Tree Rendering

Final rendering of the virtual image tree can be accomplished in a variety of ways. Given a graphics display system with only a polygon tiling function and no hidden surface removal capability, a painter's algorithm can be used to render the image. For this case, the Image Generation Processor outputs the virtual image tree in back-to-front depth order, starting with the ambient, diffuse, and specular component polygon of each potentially visible surface, followed by its shadow polygons, and then its reflected and refracted components recursively. The shade of any sublevel (child) polygon includes the composite sum of all higher level parent polygon shades, and therefore, represents the total shade for the image area defined by the given polygon.

This pre-blending intensity operation, for either faceted or Gouraud shaded polygons, is performed by the Image Generation Processor while creating the virtual image tree.

Another rendering approach, which supports anti-aliasing, is to create the virtual image tree in front-to-back depth order and scan convert polygons into a frame buffer having coverage information at each image pixel location. For this case, the Image Generation Processor begins with the deepest reflection or refraction tree branch associated with the closest polygon and recursively works towards the active tree level before proceeding to the next polygon in the node list. Shadow polygons, and the ambient, diffuse, and specular component polygon are generated after all reflected and refracted components have been scan converted for the given surface. As each polygon is independently scan converted into the frame buffer, the resulting polygon pixel coverage (area) is computed and used to modify the polygon's pixel intensity accordingly (filtering). In addition, this pixel coverage information is saved at the corresponding frame buffer location and later used to blend several polygon fragments within the pixel and to prevent a hidden surface from modifying the pixel. Several rendering algorithms employing this scan conversion method are presented in the following section.

Another rendering option is to perform all intensity blending operations during the scan conversion phase (post-blending), rather than during the image generation phase (pre-blending). This rendering approach supports intensity blending between overlapped reflected and refracted components on a given polygon, and supports complex shadow situations involving overlapped shadow regions caused by multiple light

sources. A novel image rendering technique employing this scan conversion approach is presented in the following section.

3.5. Scan Conversion Processors

The function of the Scan Conversion Processor is to render an image described by a depth sorted list of two-dimensional polygons, such as the virtual image tree described earlier. Since polygons may overlap on the screen, the major task to be performed is hidden surface elimination. In addition, each screen space polygon may include a list of surface detail polygons defining shadow areas within its surface, and a multi-level tree of face fragments describing reflections and refractions, which have to be blended together. While most graphics workstations available today provide a polygon tiling function, very few support smooth (Gouraud) shading or perform anti-aliasing. In fact, most commercially available graphics systems that implement hidden surface removal use the Z-buffer technique which cannot support anti-aliasing.

Four different algorithms have been developed and implemented to perform the tasks of hidden surface elimination and anti-aliased scan conversion of faceted, Gouraud, or Phong type polygons. Each algorithm is tailored to a specific class of images and imposes different computational and memory requirements. In all cases, the possibility of future hardware implementation has been carefully considered in developing these algorithms. The basic requirement of all four methods is the availability of a frame buffer, not necessarily physical, which holds the R-G-B color components and coverage information at every pixel location. Polygons are scan converted

independently into this frame buffer in front-to-back fashion, while accumulating the intensity and coverage at each display pixel.

Figure 3.6 summarizes the characteristics of the four algorithms implemented. In all cases, an anti-aliasing filtering operation is performed at each pixel independently by scaling the given polygon fragment intensity by its calculated pixel coverage. This coverage is either determined by actually computing the fragment area over a given pixel (defined as having finite area), or by subdividing each pixel into subpixels and estimating the fragment coverage from the number of subpixels covered by the polygon fragment. By maintaining this coverage information at each pixel location, a front-to-back hidden surface removal and intensity blending operation is performed. As polygons are scan converted into the frame buffer, the area of a given surface fragment within each pixel is calculated and used to scale the fragment intensity accordingly. If the current pixel location is empty (coverage = 0), the area-weighted polygon intensity is simply written into the appropriate frame buffer position, along with the coverage information. Otherwise, if the pixel is not completely covered (coverage < 1), the

Method	Components	Bits/Pixel	Comments
1	r,g,b,a	32	a = area coverage. Simple anti-aliasing.
2	r,g,b,m	40	m = 4 × 4 pixel mask. Simple FTB clipping.
3	r,g,b,a,m	48	Combines methods 1 & 2.
4	pixel-struct	64	Multi-level pixel masks.

Figure 3.6 Summary of Scan-Conversion Techniques

current fragment intensity and existing pixel intensity are combined according to the blending formula

$$I_{new} = I_{old} + A_{cov} I_{cur}$$

$$A_{new} = min(A_{old} + A_{cur}, 1)$$

$$A_{cov} = \begin{cases} A_{cur}, & A_{cur} + A_{old} \leq 1 \\ 1 - A_{old}, & A_{cur} + A_{old} > 1 \end{cases}$$

where I_{old}, A_{old} is the existing pixel intensity and coverage, I_{cur}, A_{cur} is the current surface fragment intensity and potential pixel coverage, and I_{new}, A_{new} is the resulting total pixel intensity and coverage.

3.5.1. Scan Conversion Method 1: Area Coverage

For the first scan conversion method, an image pixel is defined as a square region having finite area with its center located at the corresponding image point. An 8-bit quantity (a) is maintained at each pixel location in the frame buffer to represent the pixel's accumulated coverage status. As polygons are scan converted into the frame buffer in front-to-back order, the "exact" polygon pixel coverage is calculated and then used to compute its contribution to the corresponding pixel. Filtering of polygon edges is accomplished by weighting the polygon's intensity by its calculated pixel coverage (area-weighting). Hidden surface elimination is performed simply by accumulating the coverage at each pixel and preventing modification at any pixel with full coverage.

The main advantage of this scan conversion method is that it produces good quality anti-aliased images, as compared to non-filtered images, at moderate computational

and memory expense. Pixel coverage calculations proceed incrementally, as does the Gouraud and Phong shading calculations, resulting in efficient operation. A simple extension of the pixel coverage idea provides a mechanism for simulating both linear and non-linear transparency, by allowing surfaces behind a previously scan converted transparent surface to partially modify the corresponding pixel locations. In addition, the relative simplicity of the algorithm permits easy extension into a hardware implementation for real-time performance. An example of an image rendered using this area sampling technique is shown in Figure 3.10, illustrating anti-aliasing, Gouraud shading, Phong Specular highlights, and non-linear transparency.

The main disadvantage with this simple method is that it cannot handle complex pixel situations correctly because it lacks subpixel geometry information. The pixel coverage byte only indicates the accumulated coverage resulting from one or more polygon edges falling inside the pixel area, but not the specific subpixel regions covered. Since polygons may overlap on the screen, subpixel hidden surface elimination may be incorrect, especially along silhouette edges. Thus, without a pre-process clipping step to eliminate hidden surfaces, this scan conversion method cannot correctly render virtual image trees containing shadow detail, reflections, or refractions.

3.5.2. Scan Conversion Method 2: Pixel Mask

A simple way of achieving an effective increase in display resolution, without actually rendering the image at this resolution, is to define each picture element as an array of subpixels. As a polygon is scan converted, its pixel coverage is determined by find-

ing which subpixels are covered by the polygon (Figure 3.7). This subpixel information then can be used to perform a filtering operation along polygon edges that do not completely cover a pixel. In addition, by saving the state of each subpixel in the array (pixel mask), subsequent polygon fragments falling within the same image pixel can be clipped out behind regions occupied by previous (visible) fragments. Clipping one polygon fragment against another corresponds to a simple boolean operation.

For this implementation, an array size of 4 × 4 subpixels was chosen to represent each image pixel. This increases the effective display resolution by a factor of 16 and results in a moderate increase in computation and storage requirement. When coupled with the pre-blending intensity option supported by the Image Generation Processor, this scan conversion algorithm can render virtual image trees containing shadow detail polygons from a single light source, and multiple levels of reflections or refractions.

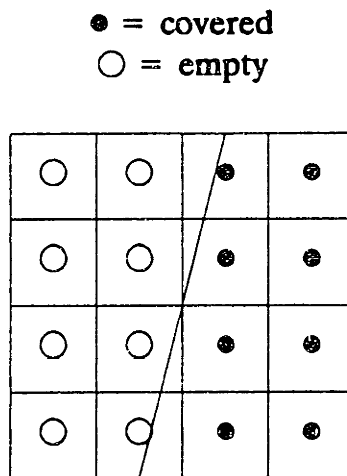


Figure 3.7 Subpixel Polygon Fragment and its Pixel Mask

Hidden surface elimination and polygon edge filtering is accomplished at the effective display resolution, and results in superior quality images, as compared to non-filtered images. An example of an image rendered using this pixel mask technique is shown in Figure 3.11, illustrating shadows from a single light source and multiple levels of reflection.

3.5.3. Scan Conversion Method 3: Hybrid Area/Mask

While the pixel mask rendering technique presented above works well at performing subpixel polygon clipping, the area sampling method actually works better at filtering edges. This is not surprising since the latter analytic method gives a better estimate of the actual subpixel area covered by a polygon edge. The pixel mask rendering technique still exhibits noticeable aliasing along silhouette edges because of the sharp intensity discontinuity that occurs between the edge and its background.

One simple way of enhancing the filtering operation along silhouette edges is to carry along the *actual* subpixel area of an edge fragment, as computed in scan conversion method 1, in addition to its pixel mask. As in the previous method, the pixel mask is used to perform subpixel polygon clipping when more than one polygon fragment falls within a pixel region. However, whenever a background polygon with full pixel coverage is to be blended with an existing foreground edge fragment, the actual saved fragment area is used, instead of the pixel mask bit count, to perform the area-weighted intensity blending operation.

Maintaining both the pixel mask and pixel coverage information also provides a mechanism to simulate linear and non-linear transparency, in addition to rendering shadows, reflections, and refractions. The pixel mask is used to perform subpixel fragment clipping, while the pixel coverage indicates the accumulated intensity coverage from previously rendered surfaces. Diffuse surfaces affect both the pixel mask and coverage information, while transparent surfaces affect only the pixel coverage. An example of an image rendered using this hybrid scheme is shown in Figure 3.12, depicting a transparent champagne glass and diffuse cube resting on a mirror with a checkerboard background.

3.5.4. Scan Conversion Method 4: Multi-Level Pixel Masks

This last scan conversion method presents a novel approach at rendering complex images described by virtual image trees. All three rendering techniques presented thus far simply perform hidden surface elimination and filtering at each image pixel, and assume that each polygon given for scan conversion contains the composite shading information for the corresponding region in the final image. These polygonal regions may correspond to simple diffuse surfaces, shadow detail, or composite regions representing the accumulation of multiple levels of reflection or refraction. Unfortunately, a surface with overlapped reflected and refracted components may not be rendered correctly since their corresponding image tree branches are traced independently, and therefore not blended together. A similar problem exists for reflected or refracted surfaces with shadowed areas, and for overlapped shadow areas caused by

multiple light sources.

The limitations inherent in the previous three rendering techniques are overcome by performing intensity blending operations during polygon scan conversion, rather than during the image generation phase. This rendering method handles both hidden surface elimination at each image tree level and performs inter-level intensity blending as the tree is traversed vertically. Hidden surface elimination and anti-aliasing at each tree level is performed using the pixel coverage and pixel mask ideas presented earlier, except that sufficient information is maintained at each pixel to save its coverage state between tree levels.

The algorithm works with two different data types (Figure 3.8): "pixel-structs" and "pixel-fragments". Pixel-structs occupy an array equal in size and shape to the final image, each occupying 8-bytes of storage, consisting of the accumulated pixel color (24-bits), a pixel-status byte (8-bits), and a possible pixel-fragment list pointer (32-

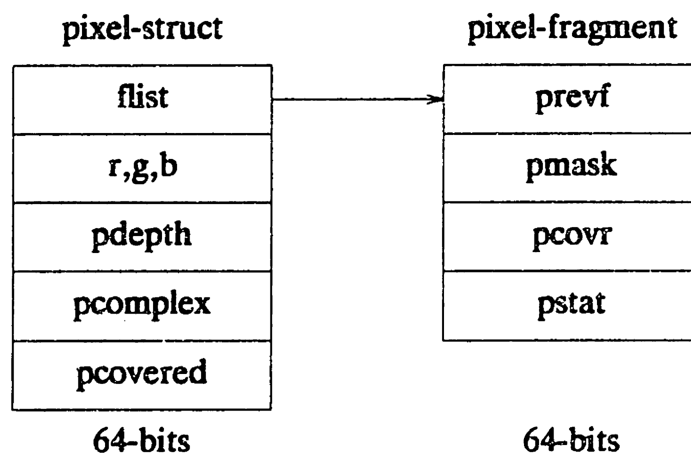


Figure 3.8 Pixel-Structure and Pixel-Fragment Definitions

bits). Pixel-fragments are dynamically allocated and deallocated as the image is rendered, and serve two purposes. They hold subpixel information when a pixel is partially covered by one or more polygon fragments at a given tree level, and are also used to save the state of a complex pixel at a higher tree level while polygons at sub-tree levels are blended with the same pixel.

Whenever an attempt is made to modify a complex pixel tagged with a tree depth lower than the current polygon tree depth (current-depth > pixel-depth), a new fragment structure is allocated to represent the pixel at the new level prior to performing any bit masking or intensity blending operations. Any previous higher level fragment structures are saved in a linked list. Similarly, whenever a polygon fragment is to be blended with a complex pixel tagged with a tree depth greater than the current depth (current-depth < pixel-depth), the pixel information is restored from the fragment list and the saved fragment structure is deallocated.

As an example, consider the complex pixel situation illustrated in Figure 3.9, corresponding to a pixel shared by two reflective polygon fragments, F_{10} and F_{20}, each with two reflected components, (F_{11}, F_{12}) and (F_{21}, F_{22}). Assuming a depth sorting order

$$F_{10} < F_{20}, F_{11} < F_{12}, F_{21} < F_{22}$$

the corresponding polygon rendering order would then be

$$(F_{11} + F_{12}) + F_{10} + (F_{21} + F_{22}) + F_{20}$$

Fragments F_{11} and F_{12} are blended together at tree level 2, after which the subpixel region associated with F_{10} is completely covered. To blend F_{10} with its reflected

fragments at the lower tree level, the existing pixel coverage information is first restored to its new tree level (in this case, the pixel coverage at level 1 is NULL). Fragment F_{10} is then blended with the existing pixel intensity and its corresponding pixel coverage (M_{11}, A_{11}) is made active. To blend F_{21} (followed by F_{22}) at tree level 2, the existing pixel coverage information at level 1 is first saved in the pixel's linked fragment list, and a new pixel structure is created to represent the pixel at level 2. Following this blending operation, the pixel at tree level 2 is completely covered, thus preventing any other fragment at or below this level from affecting the pixel. The final fragment, F_{20} at level 1, is then rendered by first restoring the saved pixel coverage information for level 1 (M_{11}, A_{11}) and then blending it with the existing pixel intensity.

Blending reflected and refracted intensity components for a given parent polygon involves rendering their respective image tree branches independently and then combining the results. The procedure used to perform this blending operation is to first render all reflected components, restore all parent polygon pixels to their original higher level state, and finally render the refracted components. The Image Generation Processor outputs a specially tagged copy of the parent polygon after tracing its reflection branch to invoke this restore operation. For these special polygons, all sub-level pixels associated with the given parent polygon are restored to their original state without affecting the existing pixel color or parent level pixel coverage information.

Generally, pixels are fully covered by a single object, and thus the active number of pixel-fragments at any one time is relatively small. However, as object surfaces are

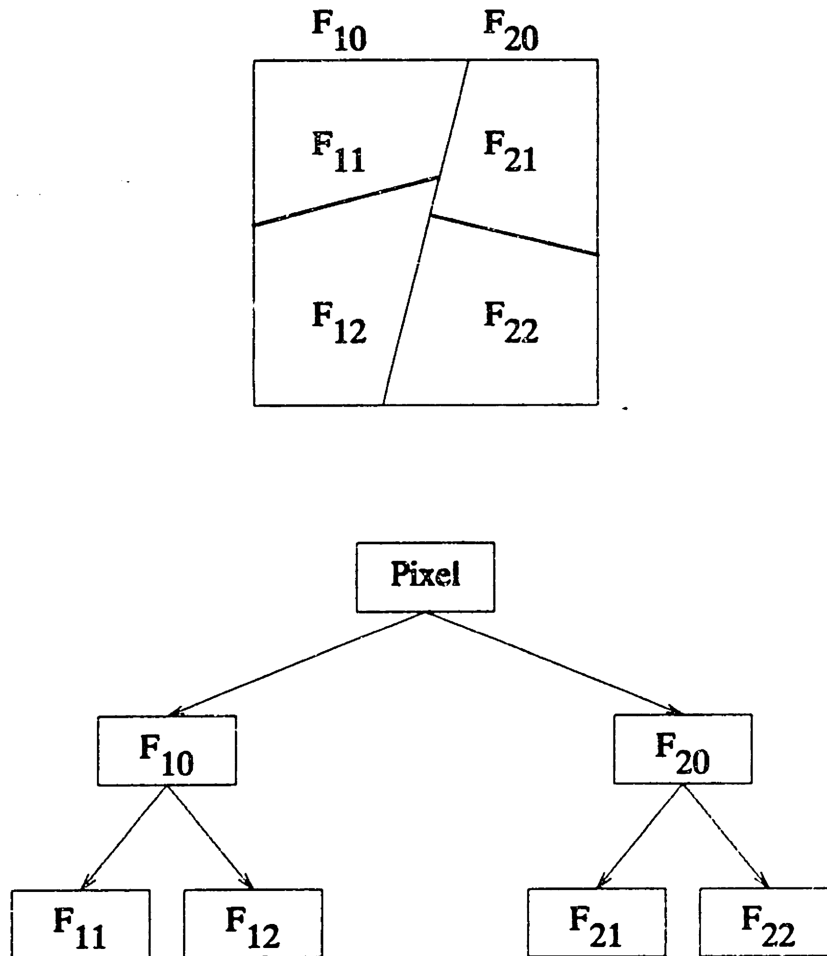


Figure 3.9 Complex Pixel Example

rendered into the frame buffer, most of the pixels along each polygon edge cause pixel-fragments to be created, while all pixels along internal object edges (those not along the silhouette of the object) will subsequently be completely covered when all polygons sharing the edge pixel are rendered. To save memory space, pixel fragments are deallocated, and the pixel is marked full, whenever a combined pixel mask indicates full coverage at any given level in the tree. For complex scenes, the total number of pixel-fragment requests are typically orders of magnitude greater than the maximum

number of active fragments at any given time.

3.6. Implementation and Results

The Image Generation Processor and Scan Conversion Processors described in this chapter have been coded in the C programming language to run on various machines. The current implementation runs either on the DEC-VAX series, DEC PDP-11 series, or IBM-PC series. In addition, the Image Generation Processor supports interactive display of virtual image trees on the Silicon Graphics IRIS 1200 Terminal and on an IBM-PC/XT with a custom graphics display system, both of which support 2D faceted polygon tiling. When using one of these graphics display terminals, the visible surface processor outputs the virtual image tree in back-to-front depth order with polygon shades at subtree levels pre-blended with their corresponding higher-level parent polygons. In this mode of operation, the display processor simply scan converts two-dimensional polygons into a frame buffer, using a painter's algorithm to eliminate hidden surfaces.

Figures 3.10 through 3.13 show images rendered by the four Scan Conversion Processors described in section 3.5. For each scene, a two dimensional virtual image description file was created by the Image Generation Processor and then processed by the corresponding Scan Conversion Processor to render the final image at 512×512 resolution, with 24-bits of color. The R-G-B color separations were then converted to luminance and output on an Autokon Model 8400 laser system, using a 65 dot/inch halftone screen. Table 3.1 summarizes the VAX-11/785 run-time statistics for each of

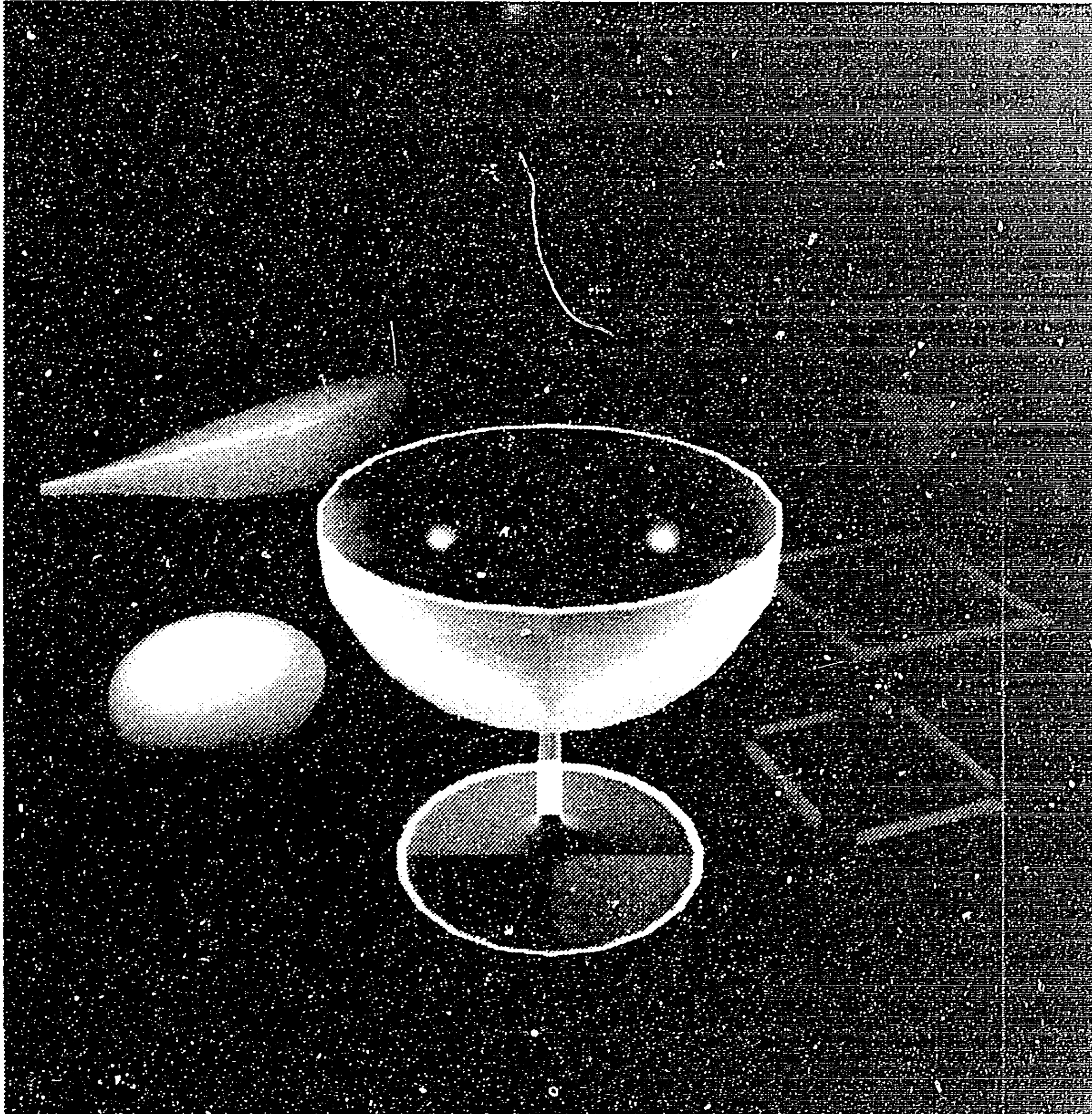


Figure 3.10(a) Method 1: (512 × 512, VAX-11/785 Time = 14s + 63s)

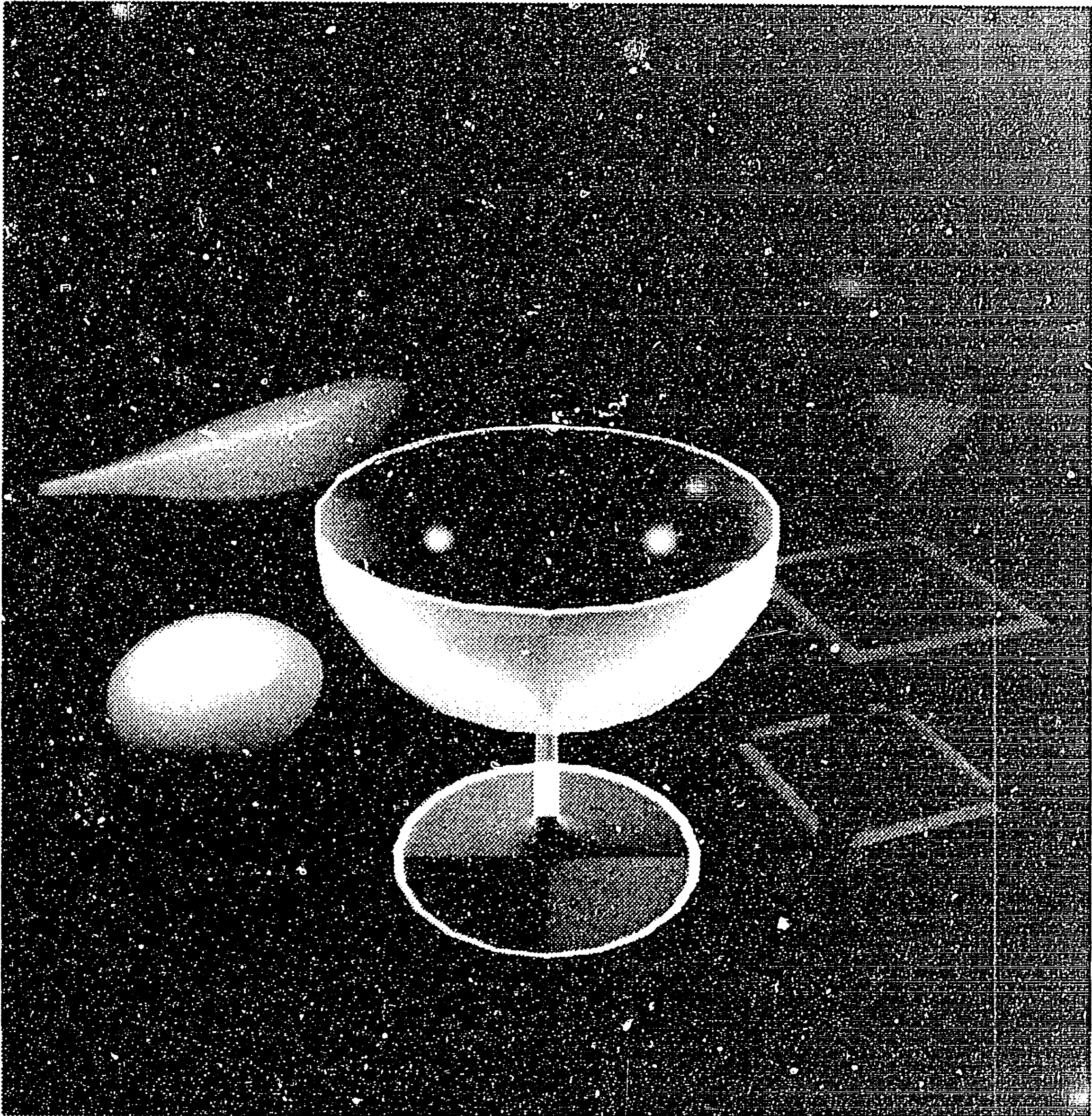


Figure 3.10(b) Method 3: (512 × 512, VAX-11/785 Time = 14s + 155s)

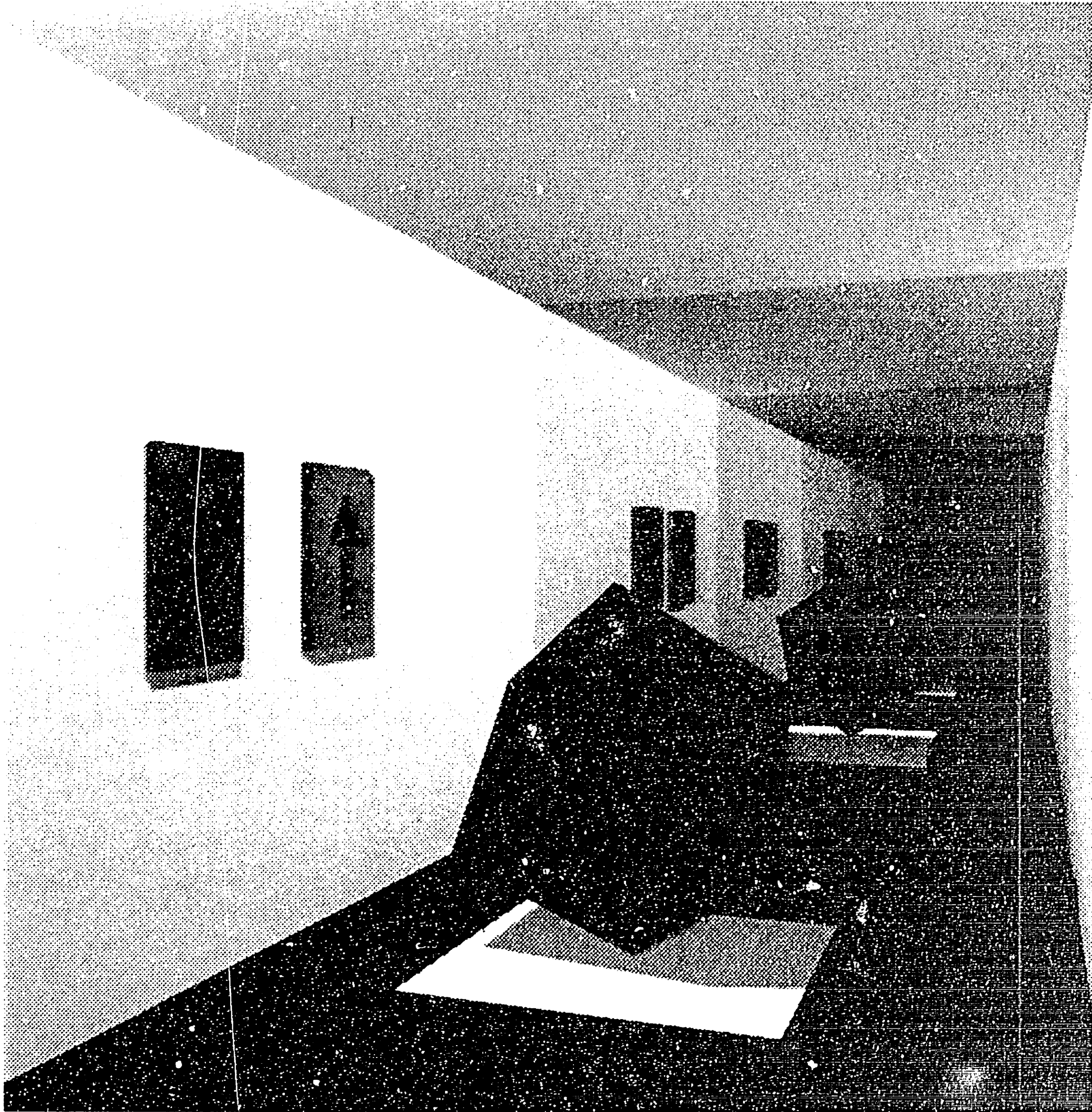


Figure 3.11(a) Method 2: (512 × 512, VAX-11/785 Time = 10s + 100s)

Figure 3.11(b) Method 4: (512×512, VAX-11/785 Time = 10s + 108s)

Figure 3.12 Method 3: (512 × 512, VAX-11/785 Time = 12s + 155s)

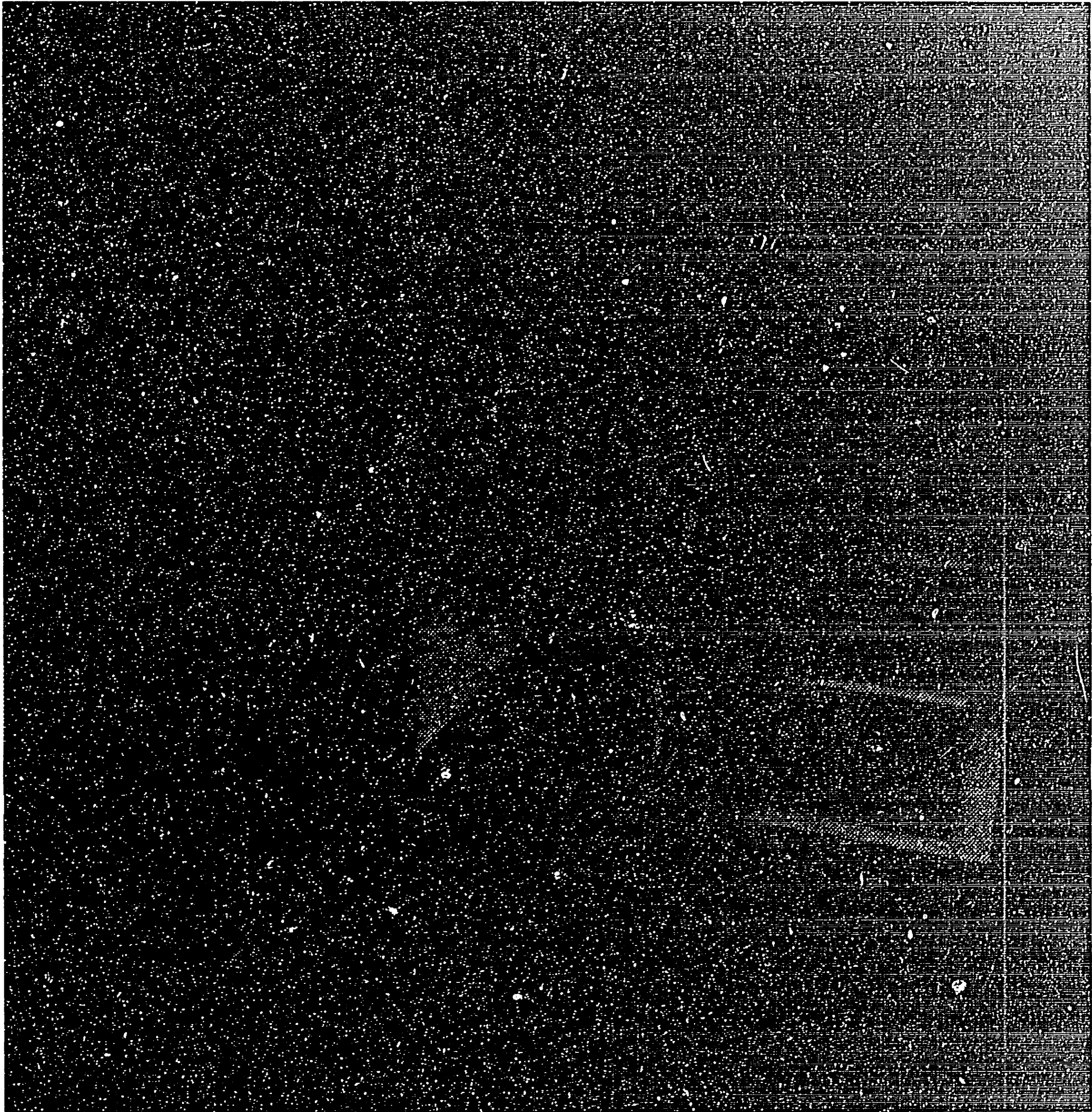


Figure 3.13 Method 4: (512 × 512, VAX-11/785 Time = 3s + 55s)

these images, including the total number of objects and polygons in the scene description, the number of polygons in the final image, the image generation time, and the scan conversion time. For comparison purposes, Figure 3.10 was rendered using scan conversion methods 1 and 3, and Figure 3.11 was rendered using scan conversion methods 2 and 4. Additional images are included in Chapter 6.

3.7. Discussion

The goal of this research was to present new image rendering techniques that simulate the global illumination effects of ray tracing, without the computational penalty associated with its algorithm. Since the intent is to incorporate these algorithms into existing graphics workstations, and perhaps influence the architecture of

Figure	Method No.	Total Objects	Total Polygons	Polygons In Image	Generation Time	Rendering Time
3.10a	1	7	1119	729	14s	63s
3.10b	3	7	1119	729	14s	155s
3.11a	2	9	42	525	10s	100s
3.11b	4	9	42	525	10s	108s
3.12	3	5	437	768	12s	155s
3.13	4	3	60	142	3s	55s

Table 3.1 VAX-11/785 Run-time Statistics for Figures 3.10-3.13

future worstations, consideration has been given to such factors as algorithm complexity and efficiency, along with the resulting image quality. In addition, consideration has been given to support a *progressive* image rendering technique, in which the quality and detail content of the final image improves with time.

Chapters 4 and 5 discuss the implementation details of the Image Generation Process and Scan Conversion Process, respectively. Chapter 6 discusses the implementation results of the various algorithms and presents possible enhancements and extensions to this new image rendering technique.

Chapter 4

Image Generation Processor

4.1. Introduction

This chapter discusses the issues involved in implementing the Image Generation Processor, including choice of illumination model, visible surface algorithm, reflection and refraction mapping, shadow generation, and output data formats.

4.2. Overview

The purpose of the image generation process is to create a virtual image description of a three-dimensional environment, given a geometric model of the scene, lighting conditions, and viewing specifications. As illustrated in Figure 4.1, input to the process consists of a scene description file, user commands, and an object data base containing geometric and color specifications for each available object. The scene description specifies the placement and surface properties of objects within a scene, lighting characteristics, and viewing parameters. Since the primary objective is to provide an interactive graphics environment, the image generation process is under full control of the user. Output from the process consists of a sorted list of screen space polygons, including surface shadow polygons and any face fragments that have been mapped onto a given polygon surface through multiple reflections and refractions.

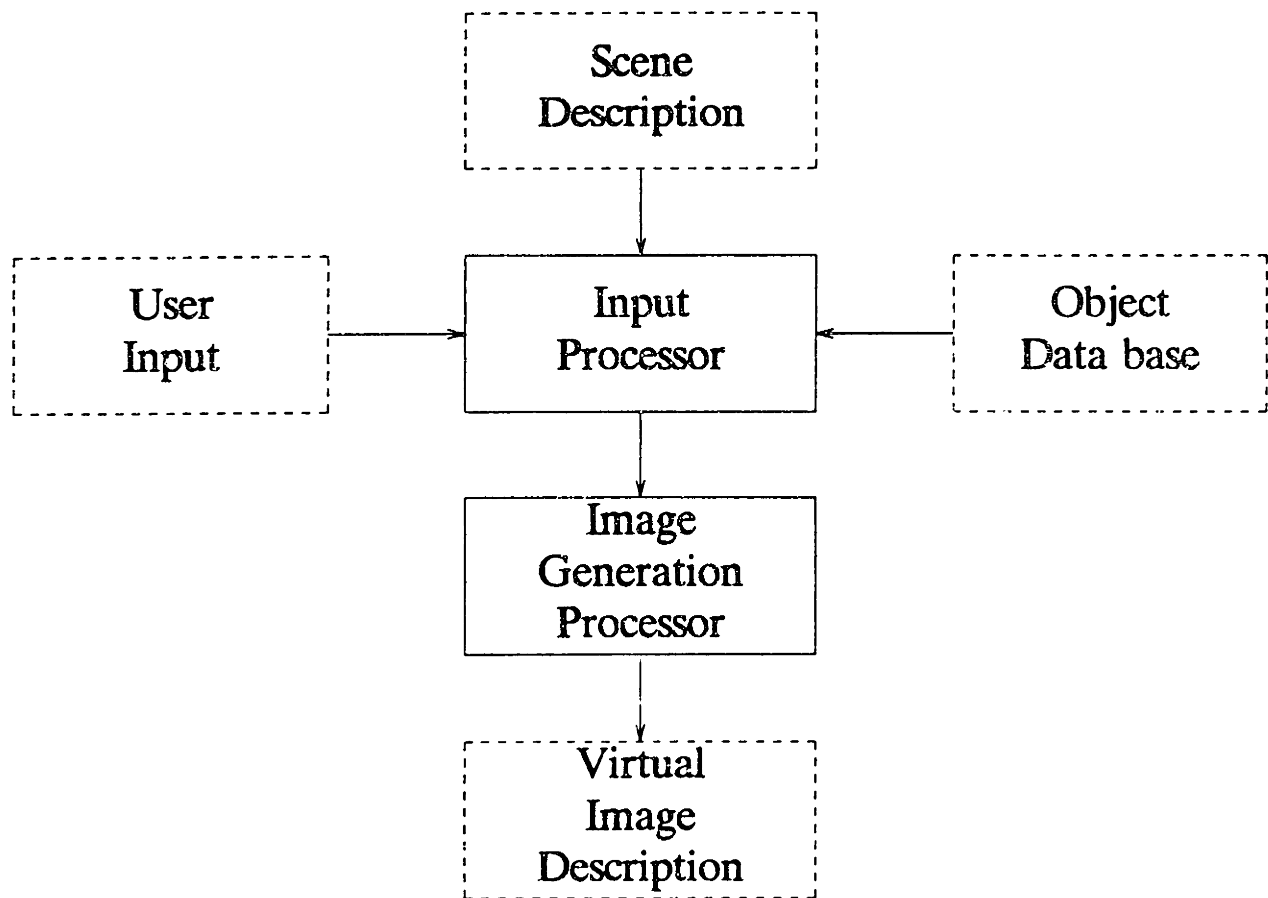


Figure 4.1 Image Generation Processor Block Diagram

This polygon list can be used directly as input to any graphics display system for immediate display, or can be further processed by a subsequent image rendering process for high quality anti-aliased scan conversion.

4.3. Input Processing

Three-dimensional environments are created through a series of commands that specify position information and characteristics of objects, light sources, and viewing conditions. These commands are given to the input processor either interactively by the user from the keyboard, from a pre-constructed scene descriptor file, or both. A useful repertoire of commands has been implemented to provide a simple and structured way of describing and positioning objects and light sources within a scene, and to specify convenient viewing and global scene parameters. Other commands permit control over various aspects of the image generation process, such as polygon sorting order, sorting complexity, and type of shading. A complete description of all available commands is provided in the Appendix.

An example of a scene descriptor file, along with its resulting image is illustrated in Figure 4.2. This simple scene depicts a diffuse cut-cube resting on table top, illuminated by a white point light source positioned away from the eye at (x,y,z) = (-4,-2,6) so as to cast a shadow onto the table top surface. A second light source, defined as diffuse, provides additional illumination but does not cast shadows. A right-handed coordinate system is assumed for both object definitions and scene descriptions, and both objects are defined with a center a (0,0,0) and a bounding-box

```
fname          test
resolution     512 512
perspective    45.0 1.0 0.01 1000.0
pushmatrix     /* save perspective transformation */
ident
polarview      7 0 75 0        /* dist,azim,inc,twist */
translate      0 0 -1          /* x,y,z */
light          -4.0 -2.0 6.0 0.8 0.8 0.8 100 Point
light           0.0 -10.0 10.0 0.5 0.5 0.5 50 Diffuse
#
defobj         cube
   color       1 1 1           /* r g b */
   scale       5 5 0.1         /* x y z */
   translate   0 0 -1          /* x y z */
endobj
defobj         cutcube
   color       0 1 0
   translate   0 0 1.05
   rotate      30 z
endobj
popmatrix
```

Figure 4.2 (a) Scene Descriptor File Example

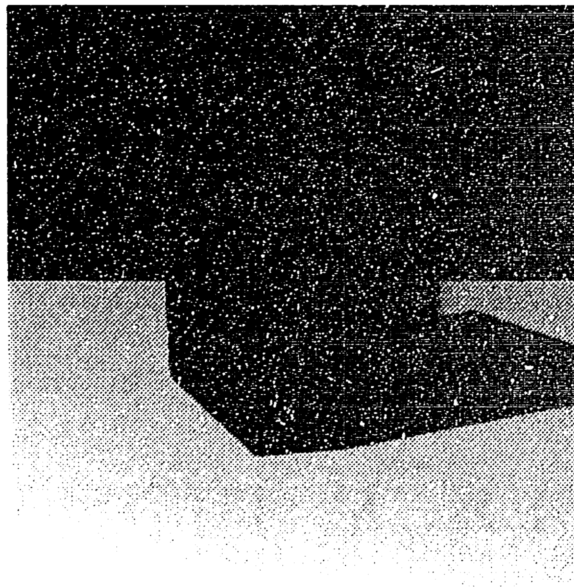


Figure 4.2 (b) Resulting Image

of -1 to +1 in all dimensions. The perspective command defines a field-of-view of 45 degrees, 1-to-1 aspect ratio, and near and far Z-clipping planes. Since the eye is assumed to be located at the origin looking down the negative Z-axis, the polarview and subsequent translate command specify the required viewing transformations needed to position the combined objects at the desired viewing angle and distance away from the eye. Commands within each object definition specify the object's color and desired geometric transformations.

Given a scene description, the first step in the image generation process is to construct an internal data structure representation of the entire scene. Associated with each specified object, a data structure is created containing the object's three-dimensional bounding-box description, an array of points specifying the coordinates of all its vertices, and a list of all polygons making up the object. In addition, various user specified object surface properties, such as color, shininess, and intensity coefficients are saved along with the object's geometric description for subsequent shading calculations. These surface parameters either can be specified as global for the entire object, as for example a white transparent champagne glass, or can be specified independently for each polygon or vertex making up the object, as for a multi-colored cube. Light source specifications include position, range, color, intensity, and type.

A 16-level deep transformation matrix stack is implemented to support hierarchical positioning of objects and light sources within a scene. Geometric operations, such as rotate, translate, and scale affect the current transformation matrix (CTM), which is applied to all light source coordinates and object points during the scene description

phase. Thus, to affect the placement of one or more objects in a scene, without affecting the global state of the environment, the CTM can be saved on the matrix stack, manipulated accordingly, and then restored after the desired object(s) have been defined. The transformation matrix stack also plays an important role during the virtual image creation phase in saving the state of the CTM prior to a reflection or refraction mapping operation.

A number of useful pre-processing operations are performed after each object is defined. First, the object's vertex points and bounding-box points are transformed using the CTM, which accounts for the composite set of geometric transformations specified to place the object into a world coordinate system, relative to all other objects in the scene. Next, polygon plane equations are computed and saved in the object's data structure, as they are later required to perform shading calculations and to compute reflection and refraction transformation matrices. In addition, if a polygonal object is defined as representing a curved object, vertex normals are also computed by averaging the plane equations of each polygon sharing the given vertex. These vertex normals are later needed to perform vertex shading calculations, for the case of Gouraud or Phong shading.

Prior to the actual image generation process, another useful pre-processing step is performed, which later improves the computation of shadows. This step involves checking to see which polygons are back-facing to any given light source and tagging them accordingly. By maintaining a simple 16-bit tag along with each polygon, up to sixteen light sources can be supported. This information is subsequently used to elim-

inate back-facing polygons of a closed object (i.e., objects entirely enclosed in polygons) from consideration in shadow polygon projection, since projecting those polygons facing the light source completely determines the shadow cast by a closed object onto another surface.

4.4. Virtual Image Creation

At the heart of the image generation process is a recursive visible surface processor responsible for finding all potentially visible polygons within an arbitrary three-dimensional clipping volume. The procedure begins with the standard truncated viewing pyramid as the initial clipping volume. All objects in the scene are transformed to this eye coordinate system by applying the specified viewing and projection transformations contained on the CTM. The visible surface processor clips all objects to the current clipping volume, eliminates back-facing polygons, and depth sorts the remaining polygon list. In the process, a *virtual image tree* is created containing all potentially visible polygons within the active clipping region. As illustrated in Figure 4.3, the first level of the virtual image tree contains all polygons within the specified field of view. Subsequent levels of the tree are created by recursively tracing polygonal reflections and/or refractions, using linear transformation techniques.

Several methods have been considered, and implemented, for creating the virtual image tree. The first, and simplest method involves using a two stage hierarchical approach to find all potentially visible surfaces at each level of the tree. The procedure begins by first transforming only the object's bounding box description to the current

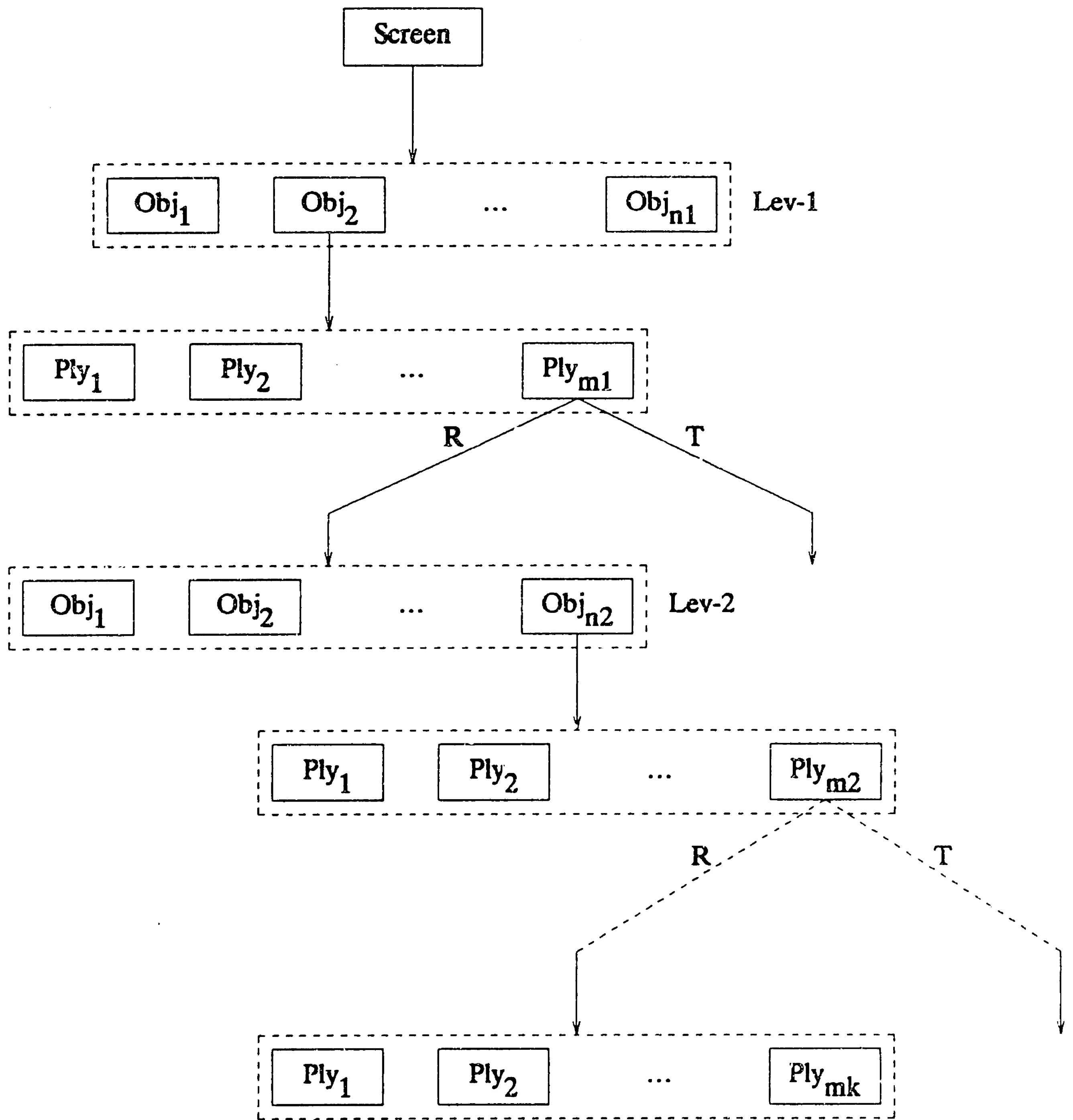


Figure 4.3 Virtual Image Tree

coordinate system, culling out objects completely outside the active clipping volume, and performing a simple depth sort of the remaining objects. Then, taking each object in depth order, polygons composing the active object are transformed by the CTM, clipped, and similarly depth sorted. For each reflective or refractive polygon in this sorted polygon list, a new clipping volume and composite transformation matrix are computed and made active, and the visible surface processor is recursively invoked to compute the reflected and refracted components for the given polygon.

The advantages of this approach are twofold. First, it employs a remarkably simple and fast sorting mechanism to establish a rendering order for the polygons composing a given object. Using an insertion sort for simple objects (less than 64 polygons), and a bucket sort for complicated objects, the sorting order is simply determined by either the minimum or maximum vertex Z-coordinate of the polygon, depending on the sorting order (i.e., front-to-back or back-to-front). The second advantage is that only a single object is active at each level of the virtual image, thus minimizing storage requirements for very complicated scenes. The only disadvantage is that correct object/polygon sorting order cannot be achieved in all cases by this simple sorting technique, especially for complex environments where objects within close depth proximity obscure each other. To partially solve this problem, a feature was implemented to expand all objects within each tree level into a composite sorted polygon list, while still using the simple sorting mechanism to order the polygons for rendering. While this feature eliminates a large percentage of sorting errors and works well for many environments, it does not completely solve the sorting problem.

The two simple methods described above were augmented by providing a more elaborate sorting mechanism to order polygons correctly and split those polygons that cause problems in the ordering. This sorting technique is essentially an implementation of the Newell, Newell, and Sancha (NN&S) sorting algorithm [17], with the addition that polygons are either sorted front-to-back or back-to-front, depending on the desired order. For the back-to-front case, each polygon, P, at the head of the sorted list is tested against each other polygon, Q, that could possibly be obscured by P because of incorrect ordering. Such polygons are those whose min-max vertex Z-coordinates overlap, which in general, involves only a small percentage of the total polygons in a scene. For these potentially troublesome polygons, a sequence of tests of increasing severity are performed to determine proper sorting order, which may result in moving Q to the top of the list or splitting either polygon P or Q and inserting the resulting fragments in their corresponding positions in the list. A simple example of two intersecting polyhedra is shown in Figure 4.4, illustrating the result of incorrect simple sorting and the more elaborate sorting technique. Notice that face splitting was required to render the objects correctly.

An extension to the NN&S sorting algorithm was considered, although not actually implemented, to improve its performance and reduce its storage requirements. As noted above, all potentially visible objects in the scene within each tree level must be expanded into a composite sorted polygon list, in order to perform the sorting step. A more efficient technique would be to use a hierarchical approach, whereby object bounding boxes at each level are sorted and tested for possible obstruction with each

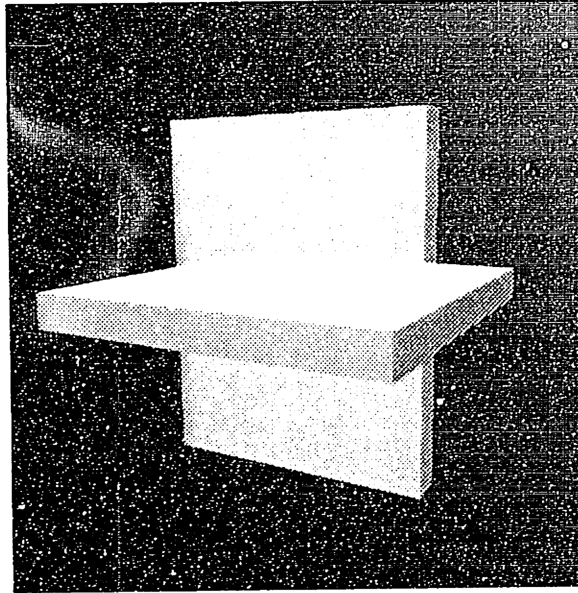


Figure 4.4 (a) Simple Depth Sorting Example

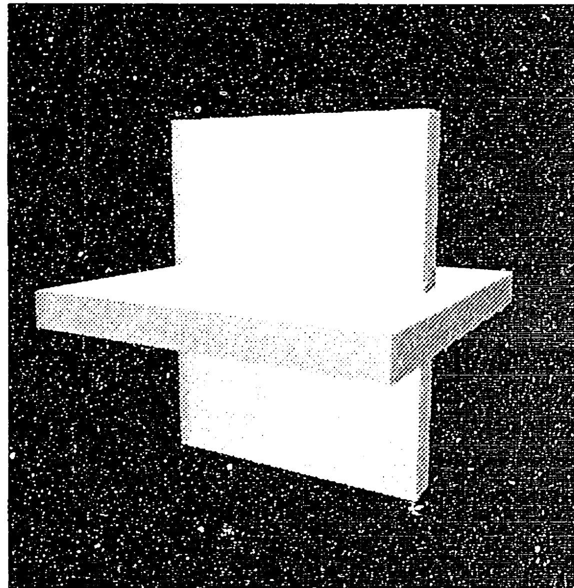


Figure 4.4 (b) Newell, Newell, & Sancha Sorting Example

other. Only those objects whose bounding boxes overlap in depth, and on the screen, need be expanded into a composite polygon list for more elaborate sorting by the above NN&S method. This simple extension to the NN&S algorithm would reduce the total number of polygons that have to be tested against each other, and minimize the total number of polygons active at any given time.

4.5. General 3-D Clipping

Most hidden surface elimination techniques involve some form of clipping operation to cull out objects outside the field of view, and clip those that cross the clipping volume [45]. Typically, a *truncated viewing pyramid*, like the one shown in Figure 4.5, is used to define the region in space where objects must lie to be visible on the screen. By applying a perspective transformation to all points in the scene, this

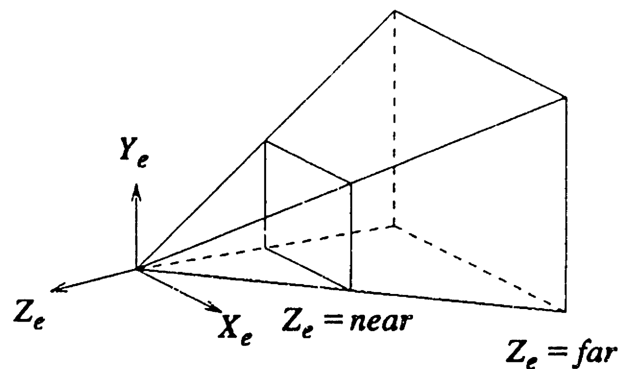


Figure 4.5 Truncated Viewing Pyramid in the Eye Coordinate System

truncated viewing pyramid is mapped into a *standard viewbox* in the screen coordinate system, characterized by values of x_s, y_s, and z_s that lie in the range -1 to +1. The clipping operation then simply becomes a test of the limits

$$-w \leq x \leq +w \ , \ -w \leq y \leq +w \ , \ \text{and} \ -w \leq z \leq +w .$$

where $[x, y, z, w]$ is the perspective homogeneous coordinate of each polygon vertex. The clipping algorithm enforces these limits by clipping each polygon edge to the six limiting planes defined above. After clipping, a perspective division by w yields the desired normalized screen coordinates of each vertex point in the range -1 to +1, which then can be mapped into physical screen coordinates by the following scaling operation:

$$X_{screen} = \frac{x}{w} V_{sx} + V_{cx}$$

$$Y_{screen} = \frac{y}{w} V_{sy} + V_{cy}$$

$$Z_{screen} = \frac{z}{w} V_{sz} + V_{cz}$$

where V_{cx}, V_{cy}, V_{cz} define the center of the viewport in the coordinate system of the output device, and V_{sx}, V_{sy}, V_{sz} define the respective distances between the center and the edges of the viewport.

A key operation in creating sublevels of the virtual image tree described earlier is to perform a general clipping function on the transformed image after every reflection and refraction transformation. Each screen space polygon, for which a reflection or refraction mapping function is performed, defines a general clipping volume as this two-dimensional polygon in the x-y plane is translated along the z-axis (Figure 4.6).

The face of the polygon defines the front clipping plane, and the translation of each polygon edge along the z-axis defines the sides of the clipping volume. For simplicity, the rear clipping plane is always chosen as the far clipping plane of the truncated viewing pyramid.

In order to improve the efficiency of the visible surface algorithm all clipping operations are performed in two hierarchical steps. First, the object's bounding box (8-points) is tested against the active clipping volume and tagged as either completely outside, completely inside, or partly in both. If the object is outside the active region, it is simply eliminated from further consideration. Otherwise, the object is added to the active object list for the current virtual tree level. Later, when an object is

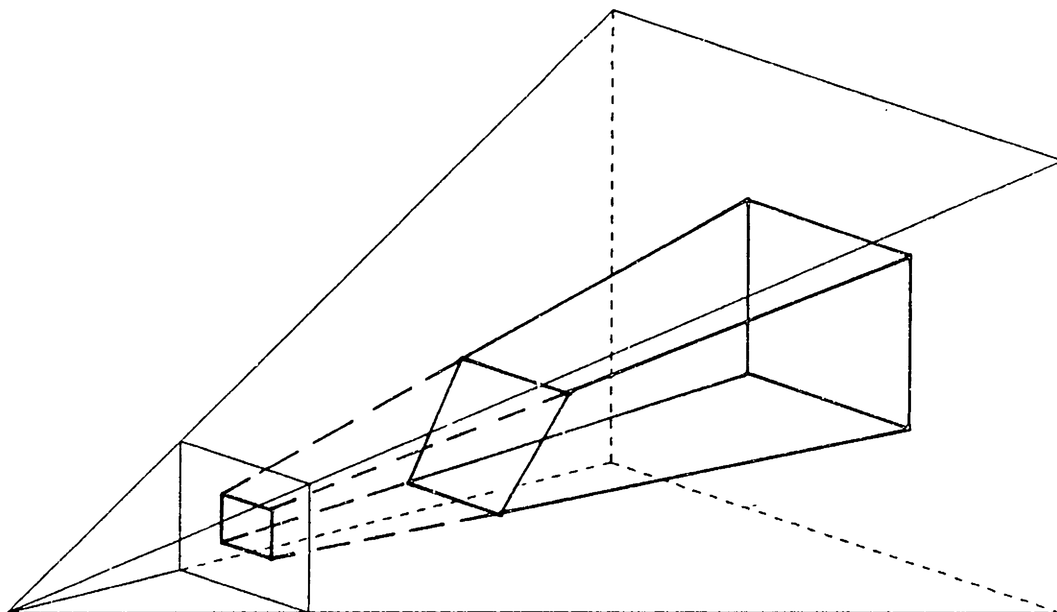


Figure 4.6 General Clipping Volume

activated for expansion into a polygon list, its saved clip-tag is examined to determine if clipping is required at the polygon level. If no clipping is required, the object's polygons are simply examined for possible back-face elimination and added to the active polygon list. Otherwise, a two-stage clipping operation is performed to accept, eliminate, or clip each polygon. First, every object vertex point is tested against each active clipping plane and a *clip − code* is computed and saved with each point, indicating on which side of each plane the point lies. As each polygon is considered, examining all the clip-codes associated with each of its vertices indicates whether the polygon is completely inside or outside the clipping volume, or whether it needs to be clipped to the boundaries.

The actual polygon clipping algorithm used is an extension of the reentrant clipping technique presented by Sutherland and Hodgman [46]. In order to support the recursive visible surface processor, a clipping volume stack was implemented to save the state of each active clipping region as branches of the virtual image tree are constructed. For each reflective or refractive polygon considered, the active clipping volume and current transformation matrix are pushed onto their corresponding stacks, and a new set of plane equations and composite transformation matrix are computed and made active. Then, the visible surface processor is recursively called to find all possible reflected and refracted fragments mapped onto the given polygon. The design of the clipping stack was motivated by the possibility of future hardware implementation of a general clipping engine, with the capabilities needed to support a multi-level visible surface processor.

4.6. Reflection/Refraction Mapping

Ray tracing algorithms typically perform all their calculations in the world coordinate system and redirect rays after each reflection or refraction. The virtual image processor performs all visible surface calculations in a transformed coordinate system, initially the viewing coordinate system, called the virtual coordinate system. After each polygonal reflection or refraction, the entire scene is transformed into a new virtual coordinate system where visible surface calculations continue for the active polygon. The transformations for reflection and refraction are specified by a 4 x 4 homogeneous matrix determined from the world space plane equation and surface properties of the given polygon. The current transformation matrix (CTM), and its associated matrix stack are used to recursively trace multiple reflections and refractions through a scene. Assuming that each world space homogeneous coordinate vector $P = [x, y, z, w]$ is transformed to a new virtual coordinate system by

$$\tilde{P} = P \cdot CTM_n, \quad n = 1,2,...$$

then, in general, the CTM at each level of the virtual image tree is given by

$$CTM_n = \mathbf{M}_{r(n-1)}\mathbf{M}_{r(n-2)} \cdots \mathbf{M}_{r(2)}\mathbf{M}_{r(1)}\mathbf{M}_v\mathbf{M}_p$$

where $\mathbf{M}_{r(k)}$ specifies the computed reflection or refraction transformation at level k in the tree, $\mathbf{M}_v$ specifies the composite viewing transformations, and $\mathbf{M}_p$ specifies the perspective transformation. $\mathbf{M}_v$ and $\mathbf{M}_p$ are established on the CTM at tree level 1 during the scene description phase, as specified by the user. The state of the CTM at each tree level is saved on the matrix stack and later restored after sub-levels of the tree have been determined.

Figure 4.7 shows the geometry of an incident ray V striking a plane with normal vector N and generating a reflected ray R and refracted ray T. All four vectors lie in the same plane and are assumed to be normalized. Since reflection in a plane is a linear transformation between each point $P_r = [x_r, y_r, z_r, w_r]$ and its mirror image $P = [x, y, z, w]$ it can be represented by a 4 x 4 homogeneous matrix, $\mathbf{M}_r$, and expressed as

$$P = P_r \mathbf{M}_r$$

This transformation can be found by noting that the perpendicular distance, d, between any point P_r and a plane defined by the plane equation $L = [A \ B \ C \ D]^T$ is given in vector form by $d = P_r L$. Thus, the virtual point, P, can be found by

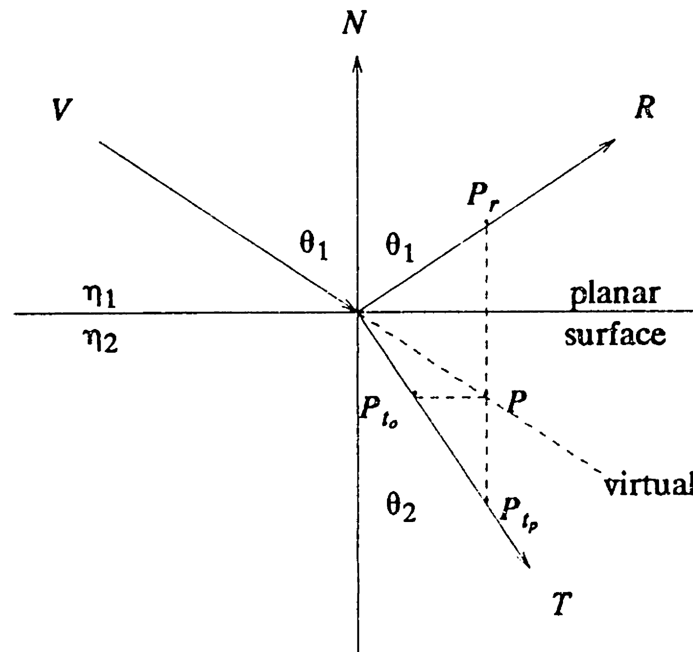


Figure 4.7 Geometry of Planar Reflection & Refraction

translating P_r a distance $-2d$ along the perpendicular direction to the plane, as specified by the plane's unit normal vector $N = [A \ B \ C \ 0]$, where $A^2 + B^2 + C^2 = 1$. In point formula form, this can be expressed as

$$P = P_r - 2(P_r L)N = P_r(\mathbf{I} - 2LN) = P_r \mathbf{M}_r$$

where $\mathbf{M}_r$ is the 4 x 4 homogeneous reflection transformation matrix given by

$$\mathbf{M}_r = \mathbf{I} - 2LN = \begin{bmatrix} 1-2A^2 & -2AB & -2AC & 0 \\ -2AB & 1-2B^2 & -2BC & 0 \\ -2AC & -2BC & 1-2C^2 & 0 \\ -2AD & -2BD & -2CD & 1 \end{bmatrix}$$

and $\mathbf{I}$ is a 4 x 4 identity matrix. Thus, given the plane equation coefficients, (A, B, C, D), of a reflective polygon in world space, a reflection transformation matrix, $\mathbf{M}_r$, is computed and then post-multiplied by the current transformation matrix to establish a new virtual coordinate system in which to perform visible surface calculations for the given polygon. The resulting matrix

$$CTM_{new} = \mathbf{M}_r \ CTM$$

contains the composite transformation which maps all objects in the scene to the virtual reflected coordinate system of the given polygon, along with any previous level reflection or refraction mapping operations.

Unlike planar reflection, refraction by a plane, in general, is not a linear transformation since the angle of refraction is a non-linear function of the angle of incidence. According to Snell's law, this relationship is given by

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$$

where θ_1 is the angle of incidence, θ_2 is the angle of refraction, and the index of refraction changes from η_1 to η_2 at the boundary. However, there are two situations

under which planar refraction is linear [44]. The simplest case is under an orthographic projection, where the eye is located at infinity, resulting in all incident rays striking a plane at the same angle θ_1. As illustrated in Figure 4.8, refraction by a plane is equivalent to a translation transformation parallel to the given plane and, therefore, can be expressed as linear transformation between each refracted point, P_t, and its virtual point, P, by

$$P = P_t \mathbf{M}_t .$$

This refraction transformation, $\mathbf{M}_t$, can be found by noting that $-(P_t L)$ gives the perpendicular distance of P_t from the plane and noting that

$$\tan\theta_1 = \frac{\alpha_1}{-(P_t L)}$$

$$\tan\theta_2 = \frac{\alpha_2}{-(P_t L)}$$

where α_1 and α_2 are the horizontal distances between the axis defined by the surface normal, N, and P, P_t, respectively (Figure 4.8(b)). The virtual point, P, can be found by translating P_t a distance $(\alpha_1 - \alpha_2)$ along the unit vector $\hat{S}$ parallel to the surface of the plane. In point formula form, this can be expressed as

$$P = P_t + (\alpha_1 - \alpha_2)\hat{S} = P_t - (\tan\theta_1 - \tan\theta_2)(P_t L)\hat{S}$$

where

$$\hat{S} = \frac{V - (N \cdot V)N}{|S|} = \frac{V - (N \cdot V)N}{\sin\theta_1}$$

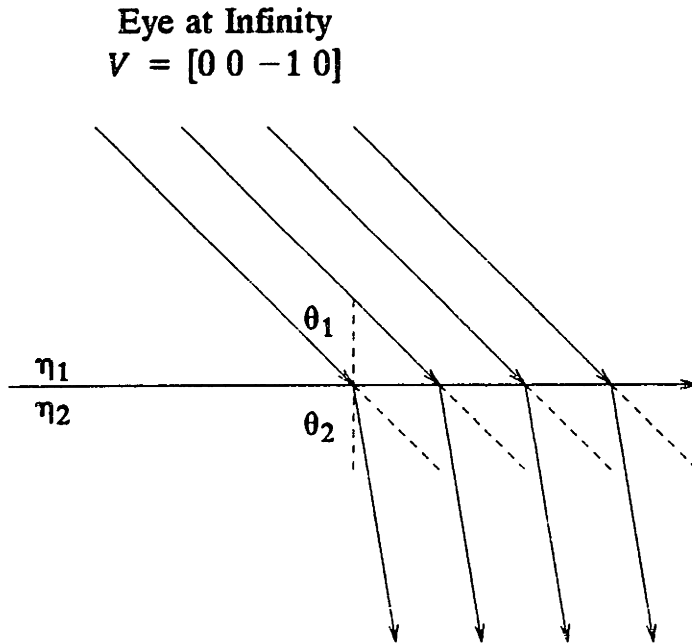


Figure 4.8(a) Refraction by a Plane Under an Orthographic Projection

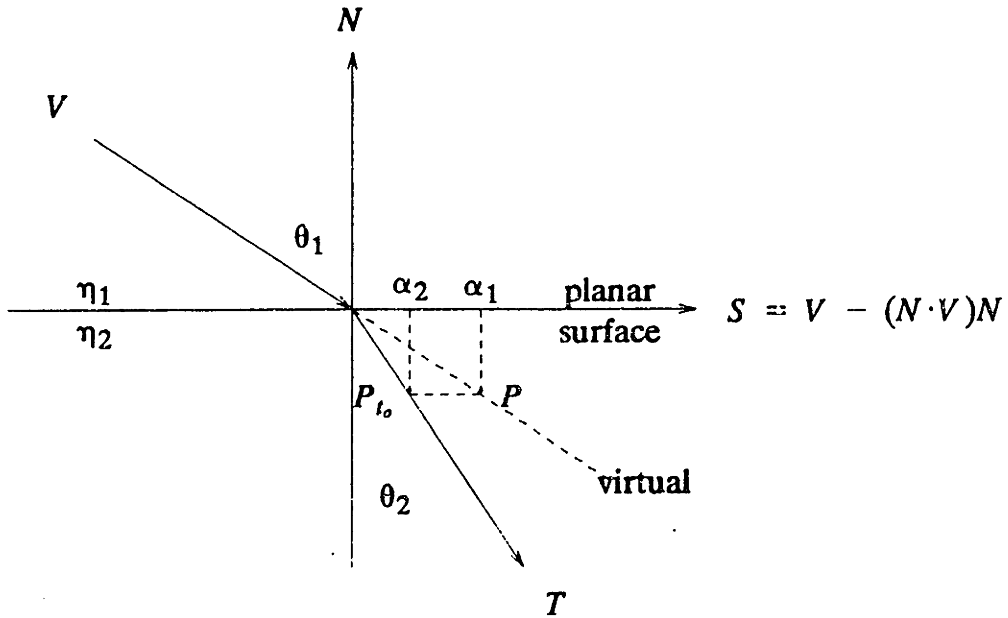


Figure 4.8(b) Geometry of Refraction Under an Orthographic Projection

Finally, substituting the normalized vector tangent, $\hat{S}$, we have

$$P = P_t - \alpha P_t L[V-(N \cdot V)N] = P_t(\mathbb{I} - \alpha L[V-(N \cdot V)N]) = P_t \mathbb{M}_t$$

where

$$\alpha = \frac{\tan\theta_1 - \tan\theta_2}{\sin\theta_1} = \frac{1}{\cos\theta_1} - \frac{\eta}{\cos\theta_2},$$

$$\cos\theta_1 = -(N \cdot V) = \beta, \quad \eta = (\eta_1/\eta_2), \text{ and}$$

$$\cos\theta_2 = \sqrt{1-\eta^2(1-\beta^2)}$$

When the eye is located at infinity, $V = [0\ 0\ -1\ 0]$, $\beta = -(N \cdot V) = C$, α is a constant, and the refraction transformation is given by

$$\mathbf{M}_{t_o} = \begin{bmatrix} 1-\alpha A^2C & -\alpha ABC & \alpha A(1-C^2) & 0 \\ -\alpha ABC & 1-\alpha B^2C & \alpha B(1-C^2) & 0 \\ -\alpha AC^2 & -\alpha BC^2 & 1-\alpha C(1-C^2) & 0 \\ -\alpha ACD & -\alpha BCD & \alpha D(1-C^2) & 1 \end{bmatrix}$$

Thus, for planar refraction under orthographic projections, an exact 4 x 4 linear transformation matrix is computed, based only on the coefficients of the plane equation, $L = [A\ B\ C\ D]$, and the relative index of refraction, η.

Another simple case where refraction by a plane can be expressed as a linear transformation is for rays at near perpendicular incidence, known as paraxial rays. This situation corresponds to the center region of Figure 4.9, which shows a view of an underwater checkerboard from the air [44], as generated by a standard ray tracer. For paraxial rays, refraction can be approximated by a linear transformation since $\sin\theta \sim \tan\theta \sim \theta$ and Snell's law becomes

$$\frac{\theta_1}{\theta_2} \sim \frac{\eta_2}{\eta_1} = constant$$

The effect of looking across a boundary with relative index of refraction η is that

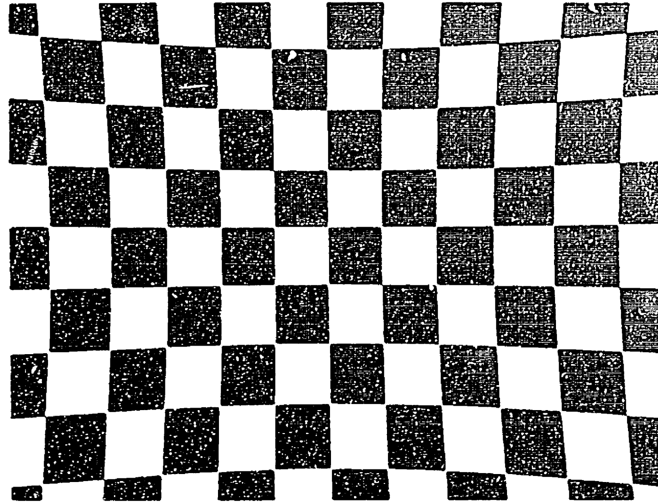


Figure 4.9 View of an Underwater Checkerboard From Air [44]

objects seem to appear at η times their actual distance. Thus, as illustrated in Figure 4.10, planar refraction within the paraxial approximation corresponds to a scaling transformation perpendicular to the plane, and can be expressed as

$$P = P_t + (\eta - 1)(P_t L)N = P_t(\mathbf{I} + \lambda LN) = P_t \mathbf{M}_{t_p}$$

$$\mathbf{M}_{t_p} = \mathbf{I} + \lambda LN = \begin{bmatrix} 1+\lambda A^2 & \lambda AB & \lambda AC & 0 \\ \lambda AB & 1+\lambda B^2 & \lambda BC & 0 \\ \lambda AC & \lambda BC & 1+\lambda C^2 & 0 \\ \lambda AD & \lambda BD & \lambda CD & 1 \end{bmatrix}$$

and $\lambda = \eta - 1$. Note that $\mathbf{M}_r = \mathbf{M}_t(\eta = -1)$, which results in a relatively simple method for computing either a reflection or refraction transformation, given the world space plane equation of a polygon and its surface properties.

It should be noted, however, that the above treatment of planar refraction assumes that the incident angle of all sight rays is small. Thus, for a perspective view,

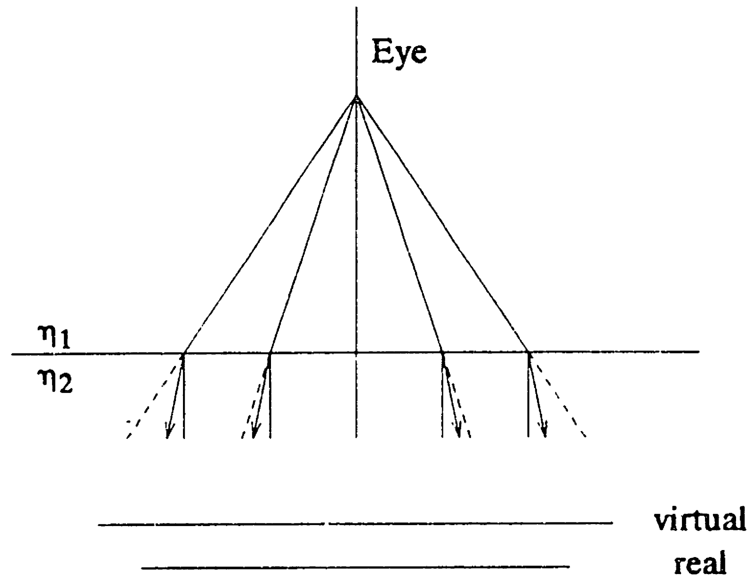


Figure 4.10(a) Refracted Paraxial Rays by a Plane ($\eta_2 > \eta_1$)

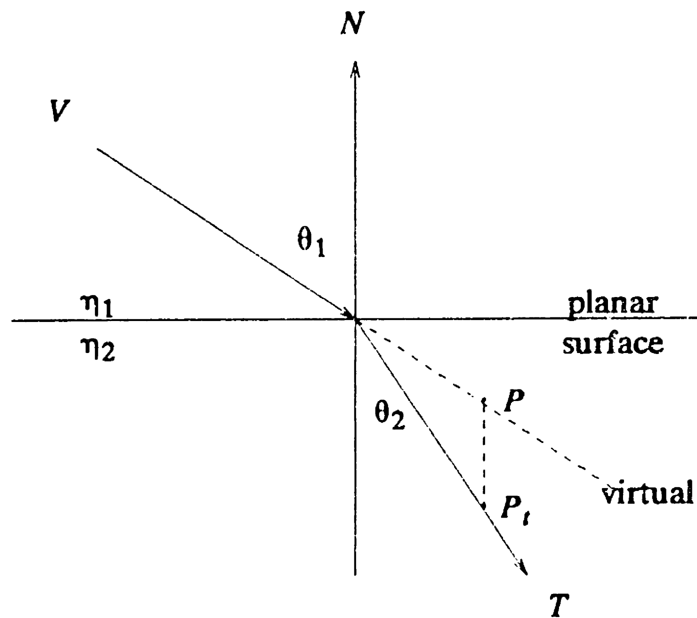


Figure 4.10(b) Geometry of a Refracted Paraxial Ray

this assumption is only valid for cases where a refractive surface, subtending a small angle, is nearly perpendicular to the line of sight. A more general treatment of planar refraction remains to be implemented, and is briefly discussed at the end of Chapter 6 as a possible extension to this rendering algorithm.

4.7. Shadow Generation

In standard ray tracing shadows are determined by sending rays from the visible point of intersection to each light source and checking for obstruction by other objects in the scene. In the present algorithm shadows are computed in reverse order by projecting polygons that lie between each light source and the active target polygon, causing cast shadows. This different approach aims at computing shadow areas during each shadow calculation instead of finding the modified intensity at a single point on the surface as done in ray tracing. Shadows are determined for each given polygon as the virtual image tree is created and then used by the image rendering process to modify the intensity of the visible surface areas.

An important consideration in shadow generation is not so much the actual projection of shadows (although this is the ultimate goal) as it is the elimination of unnecessary shadow projection calculations. This is to be expected since the number of possible shadows cast grows rapidly as the scene complexity increases. For each light source in the scene, the shadow generation process begins by creating a shadow volume, defined by the light source and each world space vertex of the target polygon. As illustrated in Figure 4.11, this volume defines the region in space where a polygon

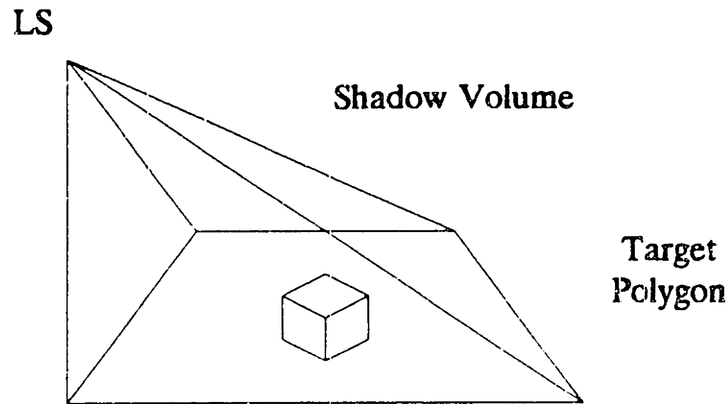


Figure 4.11 Shadow Volume Between Light Source and a Polygon

can cast a shadow on the surface of the target polygon. A two stage hierarchical clipping operation, similar to that described earlier in section 4.4, is then performed to cull out objects and polygons completely outside the shadow volume and to clip those polygons which cross its boundaries. In addition, polygons that have been tagged as back-facing to the light source (see section 4.2) are also eliminated from further consideration, since it suffices to project only front-facing polygons to completely define the shadow silhouette of a closed object.

Shadow polygons are computed by projecting the vertices of one polygon onto the plane of another. The parametric form of the line between the light source and each vertex to be projected is used to calculate this projection. As illustrated in Figure 4.12, given the two points $P_l = (x_l, y_l, z_l)$ and $P_v = (x_v, y_v, z_v)$, the set of points, $P = (x, y, z)$, that lie along the line joining P_l and P_v is given parametrically by

$$x(t) = x_l + (x_v - x_l)t$$

$$y(t) = y_l + (y_v - y_l)t$$

$$z(t) = z_l + (z_v - z_l)t$$

As the parameter t is varied between 0 and 1, a line is traced between the two end points P_l and P_v, respectively. To compute the projection of vertex P_v onto a polygon with plane equation $Ax + By + Cz + D = 0$, we simply intersect the parametric line with the plane and solve for the value of t at the common intersection point

$$A(x_l + x_d t) + B(y_l + y_d t) + C(z_l + z_d t) + D = 0$$

where $(x_d, y_d, z_d) = (x_v - x_l, y_v - y_l, z_v - z_l)$. Then, solving for t to find the projected point, P_s, we have

$$t = \frac{Ax_l + By_l + Cz_l + D}{Ax_d + By_d + Cz_d},$$

$$P_s = [x(t)\ y(t)\ z(t)]$$

Note that the numerator in the equation for t is constant throughout the shadow projection calculation between each light source and the target polygon, and thus need be computed only once. The denominator, on the other hand, varies with each vertex that is to be projected. Since each polygon considered for shadow casting has been clipped to the active shadow volume, there is no need to check if the computed shadow is actually correct or not, as required in other previous shadow algorithms [20].

Following shadow projection, each cast shadow polygon is transformed to the current virtual coordinate system by the CTM, and then possibly clipped to the active clipping volume. This second clipping operation is only required if the current target polygon is tagged as having been clipped by the visible surface processor. Note that no

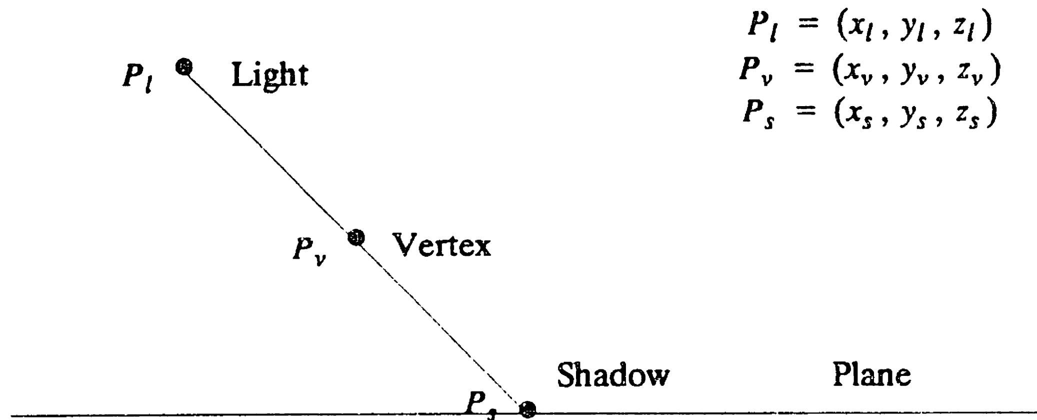


Figure 4.12 Geometry of Shadow Projection

checks are made to see whether a shadow polygon is actually visible in the final image, or whether two or more world space polygons cast shadows that overlap on a given polygon. Although this shadow overlap may result in unnecessary shadow calculations, the amount of computation needed to detect and eliminate these cases may in fact outweigh its rewards. In addition, the final image is unaffected by such cases.

Rendering shadow regions can be accomplished in a variety of ways, depending on the final scan conversion method used. In this implementation, shadow regions are rendered by scan converting all shadow polygons either before or after the corresponding target polygon, depending on the rendering order. For a back-to-front sorting order, shadows are rendered by first scan converting the target polygon and then overwriting it with all of its shadow polygons, while for a front-to-back sorting order, shadow polygons precede the target polygon. For efficiency, and also for implementation simplicity, all shadow polygons associated with a given light source carry the ident-

ical shade, as determined by computing the average target polygon shade without the given light source. In addition, since shadow polygons are independently scan converted, the shade of overlapped shadow regions caused by two or more light sources will not be exactly correct. Extensions to this simple shading scheme are discussed in Chapter 6.

An interesting variation to the above shadow generation technique would be to pre-compute all shadow polygons associated with each world space polygon and include them with the original polygon as surface detail. The advantages of this approach are twofold. First, for static scenes where light sources and objects are stationary and only the eye position is varied between successive frames, shadow polygons also remain stationary and only need be computed once for the entire sequence. Secondly, for scenes containing multiple reflection and refraction situations whereby a given object is visible many times throughout the image, pre-computation of shadows will also eliminate repeated computations. One disadvantage, however, is that the amount of storage consumed in describing a scene including shadow polygons may become extremely large, especially for very complex environments. Even if overlapped shadow polygons from a given light source are combined, the total number of resulting fragments may still be quite large.

4.8. Illumination Model and Shading Options

A number of techniques were considered and implemented for calculating the ambient, diffuse, and specular intensity contributions of each polygon and then blending these with possible reflected and refracted components. These various techniques result in different degrees of realism in the final image, and provide the user with control over the amount of computation spent in creating the image and its resulting quality. Depending on the output mode of display, either a single global shading value, or individual vertex shades are computed for each polygon. For example, if the image is to be immediately displayed on a graphics workstation having a solid polygon fill function, then only a single average polygon shade is determined. Otherwise, if the polygon list is to be further processed to produce smoothly shaded images, vertex shades are computed. In both cases, Phong's illumination model [10] is used to compute the ambient, diffuse, and specular intensity components of each polygon/vertex (Figure 4.13):

$$I = k_a I_a + k(j) \left(k_d \sum_{j=1}^{ls} (N \cdot L_j) + k_s \sum_{j=1}^{ls} (N \cdot H_j)^n \right)$$

where

I = perceived intensity,
I_a = ambient light intensity,
k_a = ambient coefficient,
k_d = diffuse coefficient,
k_s = specular coefficient,
l_s = number of light sources,
N = unit surface normal,
L_j = j^{th} light source unit direction vector,
H_j = unit vector halfway between L_j and the eye direction, E

n = measure of shininess of the surface.

The extra term, $k(j)$, was added to the diffuse and specular intensity calculation to account for the decrease in light energy received by the surface as a function of its specified range and actual distance traveled. This effect is simulated by

$$k(j) = \max\left[\left(1 - \frac{|P_j - P_v|}{Range_j}\right), 0\right]^2$$

where

P_j = j^{th} light source position,
P_v = vertex position,
$Range_j$ = maximum range of light source j.

Specular highlights are represented in a variety of ways. For constant or Gouraud shading, the specular component is determined by the corresponding term in Phong's intensity formula above and simply added to the polygon shade or vertex shade,

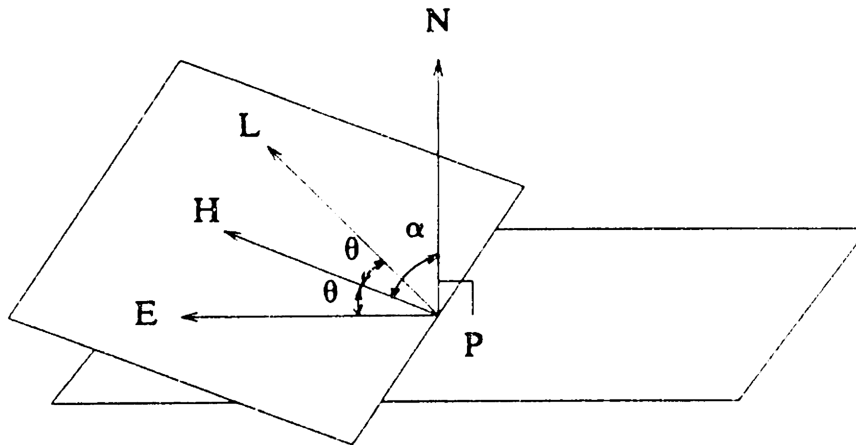


Figure 4.13 Geometry of Reflection for Shading Calculations

respectively. This results in a poor approximation to the actual specular highlight on a curved surface because the variation of the surface normal across the surface is not taken into account. A highlight anywhere within a polygon except at its vertices cannot be accurately depicted since intensity values inside the polygon are simply linearly interpolated from the values computed at the neighboring vertices. In order to better represent specular highlights in curved objects modeled by polygonal models, Phong's interpolated normal technique is used selectively only in those areas of the image where a highlight is present. An algorithm similar to the one presented by Phong and Crow [47] is used during shading calculations to determine which polygons contain specular highlights, thus requiring the more costly interpolated normal approach. This method allows using more efficient Gouraud type polygons for most of the image, and using Phong type polygons only in specific areas of the image.

Phong type polygons may include up to sixteen highlight normal vectors, and associated light source intensities, along with each vertex description. Final blending of the ambient, diffuse, and specular intensity components is performed during polygon scan conversion using the blending formula

$$I_{tot} = I_e + \sum_{i=1}^{nlts} I_{L_i} k_i$$

$$k_i = (N \cdot H_i)^n = \left(\frac{hltnml[i]_z}{|hltnml[i]|} \right)^n$$

where

n = shininess of surface,
$nlts$ = number of highlight vectors,
I_{tot} = total pixel intensity,
I_e = current pixel intensity,

$$I_{L_i} = i^{th} \text{ highlight source intensity,}$$
$$hltnml[i] = i^{th} \text{ highlight normal vector.}$$

The highlight normal vector, $hltnml[i]$, is computed by rotating the vertex normal, N, into a coordinate system having H_i aligned with the z-axis. In this space,

$H_i = [0\ 0\ 1]$, and

$$N \cdot H_i = \frac{hltnml[i]_z}{|hltnml[i]|}$$

During polygon scan conversion, these highlight normal vectors are linearly interpolated between the vertices defining the active left and right polygon edges, and then interpolated horizontally along each scan line to determine their approximate value at each display pixel position. This computation is certainly more costly than Gouraud's simple interpolated shading technique, but typically involves only a small percentage of the total number of polygons composing an image.

All intensity calculations have to be performed in world space, where objects and light sources are defined relative to the same coordinate system. However, polygon fragments generated by the recursive visible processor are in a virtual coordinate system defined by the composite application of any reflection or refraction transformations, in addition to viewing and perspective transformations. Therefore, for the case of Gouraud or Phong shading where vertex shades are required, polygons have to be transformed back to world space somehow prior to any shading calculations. To solve this shading problem efficiently, two different approaches are taken to obtain world space vertex coordinates, given the available screen space polygon information. The simplest, and most common case is when a given polygon has simply undergone coordi-

nate transformations, but not clipped to any volume. Since, for this case, there is a one-to-one correspondence between the vertex coordinates of the given polygon and its original replica in world space, the desired vertex coordinates are simply copied and then shading calculations performed. Otherwise, if the polygon has undergone any clipping operation its vertex coordinates are transformed back to world space by an appropriate inverse transformation matrix. For efficiency purposes, world space vertex normals are carried along with each polygon that is to be smooth shaded, and are not affected by coordinate or perspective transformations.

As with the determination of shadows for static environments, shading values also can be computed as a pre-processing step and saved along with each world space polygon. For faceted shading, this does not considerably increase storage requirements since only a single R-G-B triplet is needed per polygon. For Gouraud and Phong shading, however, the amount of information increases to an R-G-B triplet, and possibly a highlight normal vector and color for each light source, per vertex. Considering the potential explosion of memory requirements for complex environments, and the desire to handle such environments on small workstations with limited real and virtual memory, this shading pre-processing step was abandoned for the more efficient, compute-as-needed approach.

4.9. Non-Linear Transparency

Refraction by a plane in this implementation involves a linear transformation of all objects in the scene followed by a clipping and sorting operation. In many situations, it is desirable to simulate transparency through glassware modeled by polygonal meshes (e.g, the champagne glass shown in Figure 1.1) without the effects of refraction. Since for transparent material the relative index of refraction is unity, the refraction mapping approach will result in a transformation matrix, $\mathbf{M}_t$, equal to the identity matrix, $\mathbf{I}$. Rather than having to perform unnecessary linear mapping, clipping, and sorting operations for every transparent planar surface, a different approach is used for simulating the effects of transparency, as opposed to general refraction.

Transparency is a simple extension to the hidden surface algorithm used in this implementation. Since surfaces are rendered into a frame buffer independently in front-to-back or back-to-front order, all that is required to simulate transparent material is for the scan conversion processor to blend the intensity contribution of the current surface with that of any previously rendered surfaces in the frame buffer. Thus, for simple linear transparency, the intensity at each pixel is determined according to the blending formula

$$I_{new} = I_{old} k_t + I_{cur}$$

where

I_{new} = new pixel intensity value,
I_{old} = old pixel intensity value,
I_{cur} = current amb+dif+spec intensity value,
k_t = transparency coefficient.

This simple technique was first used by Newell, Newell, and Sancha [17] and later

improved upon by Kay [29] to simulate the effect of varying transparency through curved objects. Kay's non-linear transparency model tries to better simulate the effect of light passing through different amounts of material by decreasing the transparency factor near edges.

The technique used here to model non-linear transparency effects for curved polygonal objects is basically a variation of Kay's approach. During shading calculations, a transparency factor is computed at each polygon vertex taking into account both the eye direction, E, and the vertex surface normal, N, (Figure 4.13). The cosine of the angle between E and N gives a measure of the amount of light reaching the eye from behind a transparent object, as a function of the object's curvature relative to the eye position. This function gives a maximum when the eye direction vector is perpendicular to the surface, and decreases as it moves away from the surface normal direction. To simulate the transparency of different types of material, this cosine term is raised to some power, $tpwr$. This non-linear transparency function used is given by

$$k_t = k_{trp}(E \cdot N)^{tpwr}$$

where k_{trp} is the transparency coefficient specified for the given object.

Transparent surfaces are treated just like regular diffuse surfaces by the visible surface processor, except tagged accordingly, and carry along a transmission coefficient with each vertex. During the subsequent hidden surface removal and polygon tiling process, the scan conversion processor treats the vertex transmission coefficient as an additional color component and performs Gouraud type linear interpolation along

edges and scan lines to compute the R-G-B-T values at each display pixel location. The method used to blend final pixel intensities depends on the rendering order (front-to-back or back-to-front) and is discussed in the subsequent chapter dealing with the Scan Conversion Processor. An example of an image rendered using this non-linear transparency technique, including specular reflections, is demonstrated in Figure 3.10.

4.10. Blending Options and Output Polygon Formats

A variety of intensity blending options and output polygon formats are supported by the image generation process to allow different post-processing techniques on the sorted polygon list. For example, the polygon list can be sorted in front-to-back or back-to-front fashion, polygon shades may be specified by a single value (faceted shading) or with multiple values (Gouraud or Phong shading), and the blending of reflected and transmitted fragments may be encoded into the specified polygon shades or left to be performed by the post-process. A summary of the possible output polygon formats is provided in Figure 4.14.

Associated with each screen space polygon is a tree of fragments which have been mapped onto its surface by recursive reflection and refraction transformations. In addition, a list of shadow polygons also may accompany the polygon and each of its associated fragments. These various intensity components and shadow detail have to be blended together to form the final composite image for each area on the screen. In principle, the final intensity at each picture element is computed by recursively blend-

Type	Parameters
Special	<'0'> <level> <nvtx> <x,y vertex coords> /* per vertex */
Faceted	<'1'> <level> <nvtx> <r g b> <x,y vertex coords> /* per vertex */
Gouraud	<'2'> <level> <nvtx> <x,y vertex coords> <r g,b> /* per vertex */
Phong	<'3'> <level> <nvtx> <hilite> <nlts> <hltclrs> <x,y vertex coords> <r g b t> <hltnmls> /* per vertex */

where

Parameter	Data Type	Value
level	unsigned short	1 ... 31
nvtx	unsigned short	number of vertices
nlts	unsigned short	number of highlights
hilite	double	shininess of surface
x,y	floats	0 to 1
r,g,b,t	unsigned chars	0 to 255
hltclrs	unsigned chars	(lr,lg,lb)
hltnmls	floats	(nx,ny,nz)

Figure 4.14 Virtual Image Tree Polygon Formats

ing the various illumination components according to the formula

$$I = k_a I_a + k_d I_d + k_s I_s + k_r I_r + k_t I_t$$

which blends the ambient, diffuse, specular, reflected, and transmitted intensity contributions according to their respective coefficients. The actual method used to perform this blending operation depends entirely on the final mode of display and the tech-

nique used for performing hidden surface removal. Two techniques, with different computational requirements and resulting image quality, are now presented.

Typically, when a user is working with an image interactively on a graphics workstation having a fast polygon fill function but no hidden surface removal capability, polygons are shaded with a constant value and output to the screen as generated in back-to-front order. This rendering order simply performs hidden surface removal by overwriting pixels that are behind a visible surface. To render reflections or refractions, the image generation process recursively blends each parent polygon shade (i.e., its ambient, diffuse, and specular terms) with each of its children fragments at lower levels in the virtual image tree as the tree is constructed and traversed from top to bottom. Shadows are depicted simply by overwriting each parent polygon with all of its shadow polygons.

An example of this rendering procedure is illustrated in Figure 4.15, which depicts a cube resting on mirror along with its associated two-level virtual image tree. The first level contains all visible polygons within the initial viewing pyramid, consisting of the mirror surface, P_{11}, and the three front-facing polygons of the cube, P_{12}, P_{13}, P_{14}. The second level of the tree contains polygons mapped onto the mirror by reflection, consisting of P_{21}, P_{22}, P_{23}, which are visible surfaces of the cube's reflection. For a back-to-front rendering order, the image generation process outputs polygons in the following sequence:

$$P_{11},\ P_{21},\ P_{22},\ P_{23},\ P_{12},\ P_{13},\ P_{14}$$

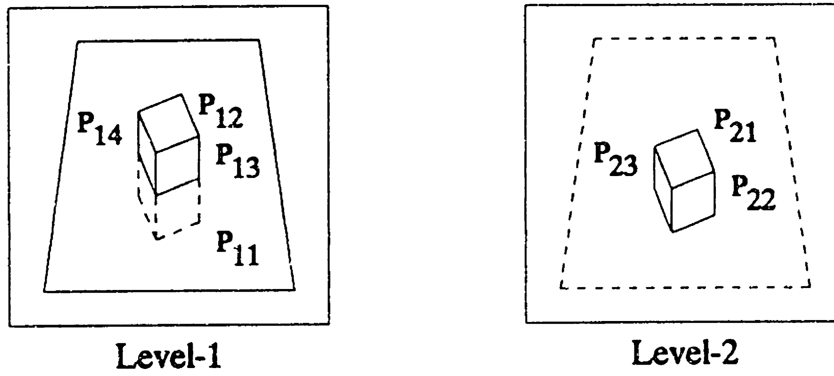


Figure 4.15(a) Cube on a Mirror

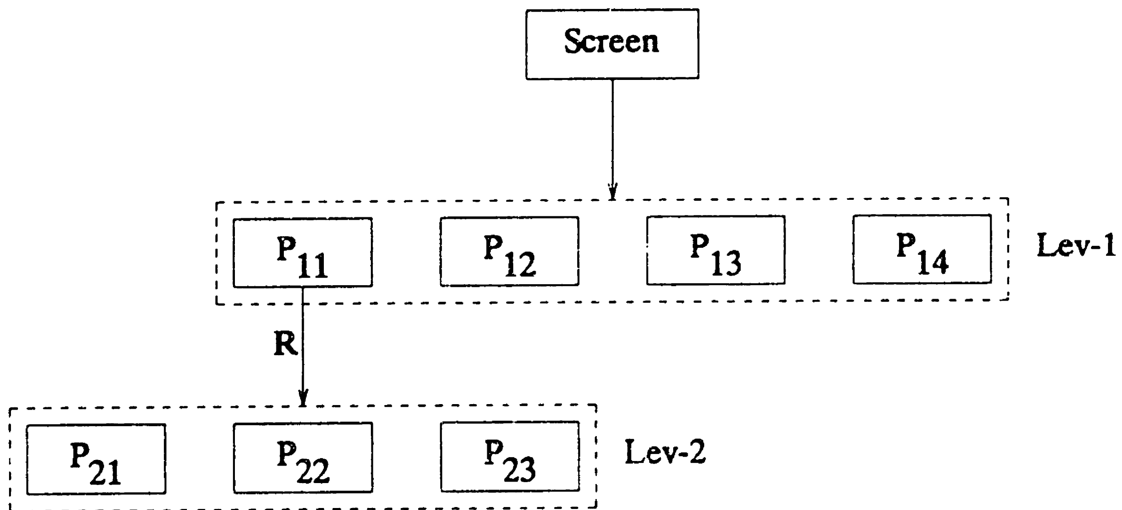


Figure 4.15(b) Virtual Image Tree

where the polygon shades of fragments P_{21}, P_{22}, and P_{23} are given by

$$I_{21} = k_r(k_a I_{a21} + k_d I_{d21} + k_s I_{s21}) + I_{11},$$

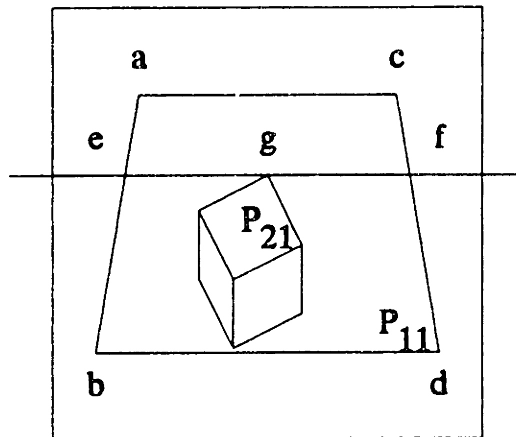
$$I_{22} = k_r(k_a I_{a22} + k_d I_{d22} + k_s I_{s22}) + I_{11},$$

$$I_{23} = k_r(k_a I_{a23} + k_d I_{d23} + k_s I_{s23}) + I_{11},$$

where

$$I_{11} = k_a I_{a11} + k_d I_{d11} + k_s I_{s11}.$$

This technique also has been extended to Gouraud shading by recursively blending vertex shades of each child polygon with the shade of the parent polygon evaluated at the appropriate points. These parent shades are determined by a double linear interpolation procedure, first along the two edges intersected by the given child vertex Y-coordinate (in screen space), and then horizontally to the desired X-coordinate. This procedure is illustrated in Figure 4.16 for fragment P_{21} of the example presented above.



e found by interpolating between a & b
f found by interpolating between c & d
g found by interpolating between e & f

Figure 4.16 Parent Shading Interpolation Example

Without additional processing, this simple image rendering technique has several limitations. First, it cannot correctly blend both reflected and refracted components on a given polygon since each branch of the image tree is processed independently and fragments in both the reflected and refracted branches may overlap on the screen. Second, it cannot combine shadows with either reflections or refractions, again because these fragments are treated independently and, therefore, not blended. Note however, that no problem arises for perfect mirrors or perfectly transparent objects, since shadows cannot be cast on these surfaces anyway. Lastly, complex shadow situations involving overlapped shadow regions caused by multiple light sources are not depicted correctly because of the overwriting nature of the algorithm. But, aside from these

special limitations, this simple rendering algorithm produces moderate quality images relatively quickly with shadows, reflections, and refractions.

A second technique for producing the final image, which solves the limitations of the simple method, involves a new rendering algorithm to be discussed in the following chapter. This new algorithm accepts a front-to-back sorted polygon list, along with any cast shadow polygons and multi-level reflected and refracted fragments associated with each polygon, and produces smooth shaded anti-aliased images with hidden surfaces removed. For this mode of operation, the image generation process simply outputs polygons in the desired front-to-back order, with vertex shades given by the computed sum of ambient, diffuse, and specular intensity components. Thus, for the example illustrated in Figure 4.15, the output sequence of polygon fragments is given by

$$P_{12},\ P_{13},\ P_{14},\ P_{21},\ P_{22},\ P_{23},\ P_{11}.$$

Final blending of these components with any reflected or refracted fragments is performed during polygon scan conversion by the subsequent image rendering process. Note that for the case of reflection (or refraction), fragments at sub-levels of the tree are output before their corresponding parent polygon. As discussed in the following chapter, this rendering order is required to allow correct blending of intensity fragments, while at the same time supporting hidden surface elimination with anti-aliasing.

Chapter 5

Scan Conversion Processors

5.1. Introduction

This chapter discusses the issues involved in implementing the Scan Conversion Processors, including hidden surface elimination, blending of diffuse, reflection, and refraction components, and various approaches to anti-aliasing. Four different implemented scan conversion techniques are presented.

5.2. Overview

The purpose of the Scan Conversion Processor is to render an image given a sorted list of two-dimensional convex polygons describing the scene. This image description consists of a polygon for each potentially visible surface on the screen, along with a list of surface detail polygons defining shadow areas within its surface. In addition, each screen space polygon may contain a tree of face fragments that have been mapped onto its surface by multiple reflections and refractions. The tasks to be performed by the scan conversion process include hidden surface elimination, polygon tiling, combining shadow detail polygons cast by various light sources, and blending of reflected and refracted components with the surface shading information. A number of different methods have been considered and implemented for performing these

tasks, which result in varying levels of computational expense and image quality. In all cases, the elimination of aliasing artifacts has been incorporated into the algorithms, and the possibility of future hardware implementation has been carefully considered.

A block diagram of the scan conversion process is shown in Figure 5.1. Input to the process is a screen-space virtual image description of a scene, such as the virtual image tree produced by the Image Generation Processor discussed in the previous chapter. This image description is assumed to be composed of two-dimensional convex polygons, which may overlap on the screen, with vertex coordinates specified in normalized screen-space (i.e., X,Y in the range -1 to +1). Thus, given the same image description, the final picture can be rendered at various display resolutions. A variety of data formats are supported to allow specification of faceted, Gouraud, or Phong polygon types. Faceted polygons are specified with a single R-G-B shading value, whereas Gouraud and Phong polygons include an R-G-B triplet for each vertex. In addition, Phong polygons may include up to sixteen highlight vectors, and associated light colors, along with each vertex specification to allow the simulation of well-defined specular highlights. Polygons are processed independently in front-to-back order by one of two separate anti-aliasing tilers, depending on their type. Each tiler slices a given polygon into horizontal strips in scan-line fashion and passes these scan segments to its corresponding segment processor, which performs hidden surface removal and anti-aliased scan conversion into a frame buffer.

Providing a dual path for processing mixed polygon types permits using simpler Gouraud type polygons for most of the image and only using expensive Phong polygons

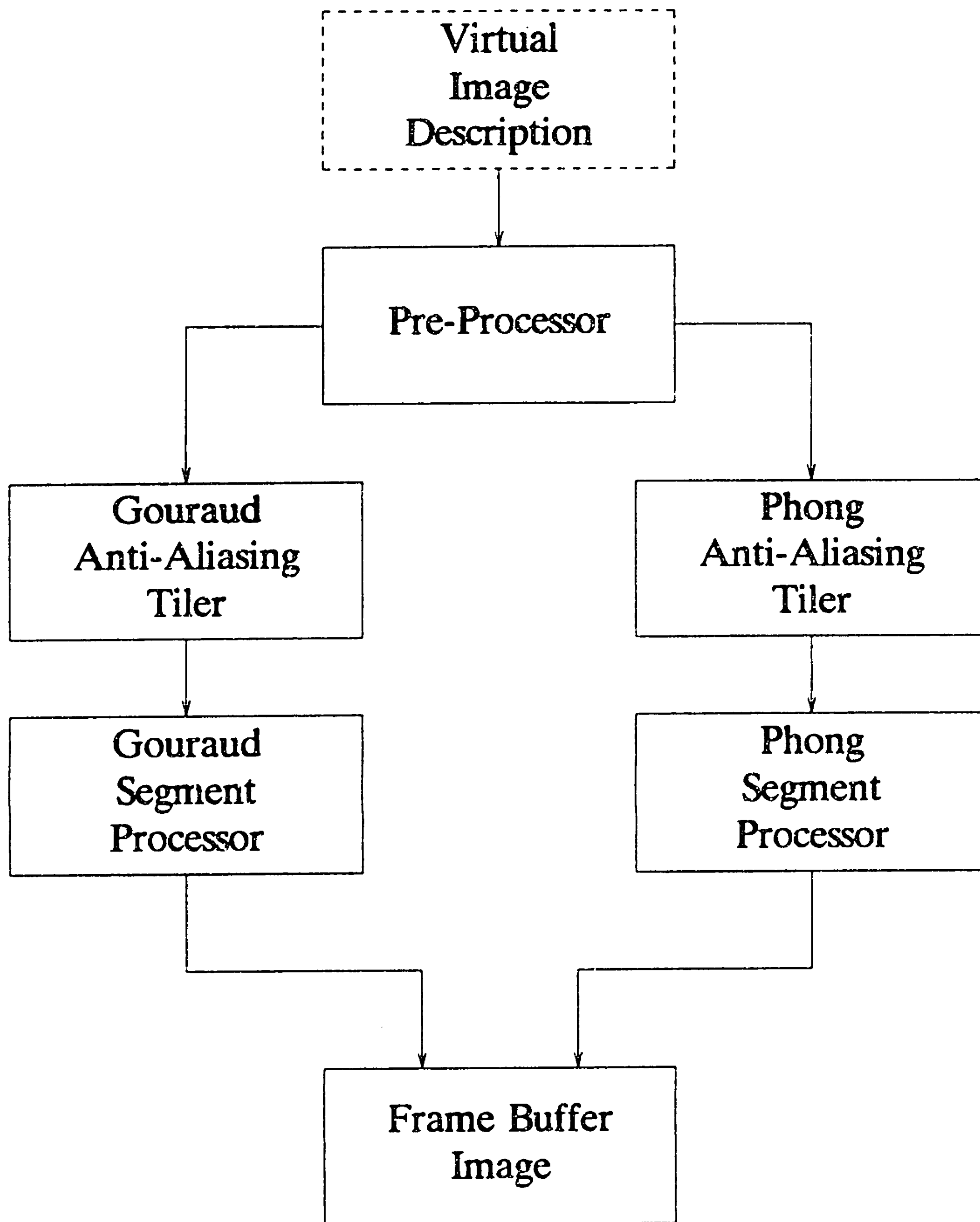


Figure 5.1 Scan Conversion Processor Block Diagram

to render specular highlights in specific regions of the image. As indicated above, Phong type polygons may include highlight vectors with each vertex, which have to be interpolated along edges and then along each scan line, and also normalized at each pixel position. To avoid unnecessary computation, Phong type polygons are only used when absolutely needed to represent a specular highlight in a specific area of the image, which typically corresponds to a small number of polygons.

Four different algorithms have been developed and implemented to perform the tasks of hidden surface elimination and anti-aliased blending of polygon fragments. Each algorithm is tailored to a specific class of images and imposes different computational and memory requirements. The basic requirement of all four methods is the availability of a frame buffer, not necessarily physical, which holds the R-G-B color information at every pixel location, along with an additional component to be used for hidden surface elimination and pixel blending operations. Depending on the algorithm used, this extra component imposes additional memory costs anywhere from one byte (8-bits) to five bytes per pixel.

5.3. Anti-Aliasing Techniques

Without doubt, the importance of anti-aliasing in the production of realistic computer generated imagery has been demonstrated over the past few years by the superior quality of images displayed. Furthermore, a survey of the computer graphics literature reveals that the same anti-aliasing technique probably has never been implemented more than once. Each new rendering algorithm presented seems to include a unique

approach to solving the aliasing problems inherent in synthetic images. Therefore, it should not be surprising to the reader to see yet another approach to anti-aliasing in the context of this thesis. In fact, several different methods were considered and implemented as part of the hidden surface algorithm.

Aliasing in computer generated images is caused by the limited resolution of displays, resulting in undersampling. The problem is most evident along edges of objects (as jagged edges), in areas of complicated detail (as Moire patterns), and in small objects, which appear and disappear between the sample points. As Crow points out [4,5] there are basically two techniques for solving these problems. The least painful method is to increase the effective resolution at which the image is generated and then filter prior to resampling at the display resolution. In practice, however, effective resolutions of four to eight times the display resolution are required to produce images without noticeable aliasing artifacts, thereby increasing the cost of producing these images. For most rendering algorithms, this cost is proportional to the square of the resolution. A different approach is to make each sample point represent a finite area on the display screen, instead of an infinitesimal spot, and essentially perform a filtering operation at each pixel during the scan-conversion phase. This area sampling technique has the effect of applying a convolution filter prior to sampling the image at the display resolution.

One of the earliest image rendering algorithms to perform anti-aliasing by area sampling was Catmull's scan line algorithm [48]. The image plane is divided into abutting sample squares of equal size, with centers at each image point. Catmull's algo-

rithm produces visible polygon segments along each scan line and includes a pixel clipping mechanism for finding visible portions of polygon fragments within each sample square. The intensity at each pixel is computed as a weighted sum of each visible surface fragment within the corresponding sample square

$$I_{pixel} = \sum_{i=1} I_{poly(i)} A_{poly(i)}$$

where $I_{poly(i)}$ is the intensity contribution of polygon i, and $A_{poly(i)}$ is the visible area of the polygon within the given sample square. While this algorithm correctly accounts for every visible polygon sliver at each pixel it is so computationally expensive that its primary use has been in simple two-dimensional animation with a small number of large polygons.

More recently, bit-masks have been used to approximate the visible coverage of each polygon fragment within a given sample square [49, 24], resulting in good quality images at considerable reduction in rendering time. Other recent algorithms, [50, 51, 52], have expanded the region under the filter kernel to include the area surrounding each sample square. This allows using different weighting functions for the filter kernel, instead of the unweighted filter approach used in area sampling, producing excellent results, but at considerable computational expense.

The approach taken here to perform anti-aliased scan conversion is essentially an extension of the area sampling technique discussed above. While other filtering methods are known to produce better results, they are considered too expensive for the desired application. Area sampling is found to be a good compromise between these enhanced filtering techniques and no filtering at all.

5.4. Hidden-Surface Algorithm

The method used to perform hidden surface elimination centers around a frame buffer containing color and coverage information for each picture element. Convex polygons are delivered to the scan conversion process, one-by-one, in front-to-back depth order by the Image Generation Process and scan-converted into a frame buffer. Since polygons may overlap on the screen, sufficient information is maintained at each pixel location to provide a mechanism for performing hidden surface removal. For the case of simple polygon tiling without anti-aliasing considerations, this simply amounts to a flag (e.g., one of the possible intensity values) indicating whether the given pixel had been covered by a previously rendered surface. Of course, a back-to-front rendering order performs essentially the same operation without the need for any coverage information. But, for the stated purpose of performing a filtering operation, additional information is needed at each pixel to indicate the extent of coverage caused by a previously rendered polygon. This coverage information is then used to prevent a hidden polygon fragment from affecting the pixel intensity of a previously rendered visible polygon, and to blend contributions from several visible polygon fragments within a given pixel.

Four different methods are considered, implemented, and compared for performing hidden surface removal, anti-aliasing, and blending of polygon intensity contributions from (ambient + diffuse + specular) shading information with its reflected and refracted components. Common to all methods is a pair of polygon tilers (one to handle Gouraud polygons and one to handle Phong polygons) that divide each polygon

considered into horizontal segments for processing by the appropriate scan conversion processor. A discussion of these polygon tilers is given below, followed by a description of each scan conversion method.

5.5. Gouraud/Phong Polygon Tilers

Each polygon delivered to the scan conversion process is tiled independently from top to bottom by either a Gouraud or Phong tiler, depending on its type. The function of these tilers is to divide each polygon into horizontal scan segments, determine the position and color at the end points of each segment, and pass them to the corresponding scan line segment processor, which computes the final intensity at each pixel along the given scan line. Since polygons are restricted to be convex and without holes, each scan line intercepts the polygon in at most two points, defining a single horizontal segment. For each polygon considered, the tiler first examines all the vertices making up the polygon to establish the topmost vertex and computes a screen space bounding box, which is used to eliminate tiny polygons. Given that polygon vertices are stored in clockwise order, two edges are defined between the top vertex and the preceding (left-edge) and succeeding (right-edge) vertices, as illustrated in Figure 5.2. Subsequent left and right edges are similarly defined by the simple vertex formula:

$$NewEdge = (OldEdge + NumVtx + VtxInc) \ mod \ Numvtx$$

where

$$NumVtx = \text{ number of vertices in polygon, and}$$

$VtxInc$ = -1 for left edge, +1 for right edge.

For each active left and right polygon edge, the tiler constructs a structure to hold attributes for the edge. As shown in Figure 5.3 (a), the edge attributes include the present X-position and color, along with their corresponding increments to establish the position and color at the next scan line. Thus, rather than recomputing the position and color information at each scan line, these increments are simply used to update the present state of the edge to determine its subsequent state. The edge structure also includes the vertex number associated with the terminating vertex of the current edge and a count specifying the number of scan lines remaining until the terminating vertex Y-position is reached. In addition, the edge structure contains an optional pointer to an array of edge-normal structures, which is used by the Phong tiler

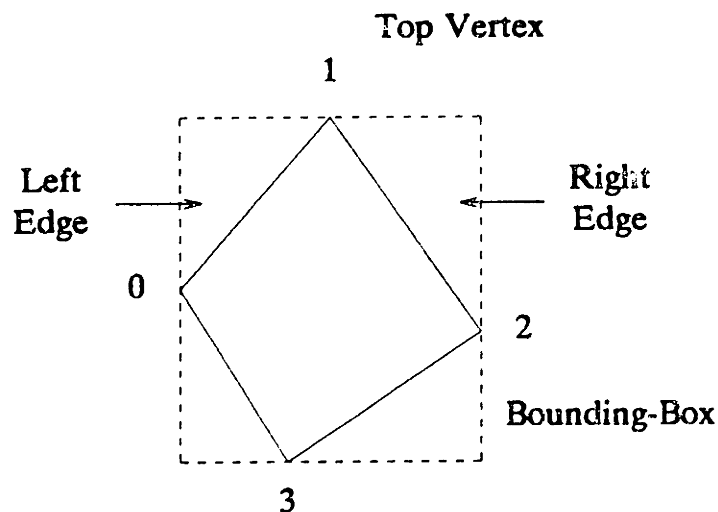


Figure 5.2 Establishing Initial Left and Right Edges

to store attributes for each specified highlight normal and its associated increments (Figure 5.3 (b)). When a terminating vertex is reached, the appropriate edge structure is recalculated to reflect the attributes for the new edge below the current one. The tiling process terminates when all polygon vertices have been exhausted.

Unlike simple polygon tilers, which do not perform anti-aliasing, the filtering tilers described above cannot ignore edges with vertical spans of less than one scan line, and must properly handle the transition from one edge to another. Figure 5.4 illustrates a simple example of a polygon spanning four scan lines, with left edges L_1, L_2, and L_3,

```
typedef struct
{
        double          x,dx;           /* Horiz position & increment */
        double          r,g,b;          /* Edge Color */
        double          dr,dg,db;       /* Color increments */
        double          t,dt;           /* Transparency & increment */
        double          ht;             /* Vertical height */
        int             vtxno;          /* Terminating vertex number */
        int             lnth;           /* Number of lines remaining */
        edge_normal     *nmlptr;        /* Optional highlight normals */
} edge_position;
```

Figure 5.3 (a) Polygon Edge Structure Definition

```
typedef struct
{
        double  xn,yn,zn;       /* Highlight normal vector */
        double  dxn,dyn,dzn;    /* Normal vector increments */
} edge_normal;
```

Figure 5.3 (b) Highlight Normal Structure Definition

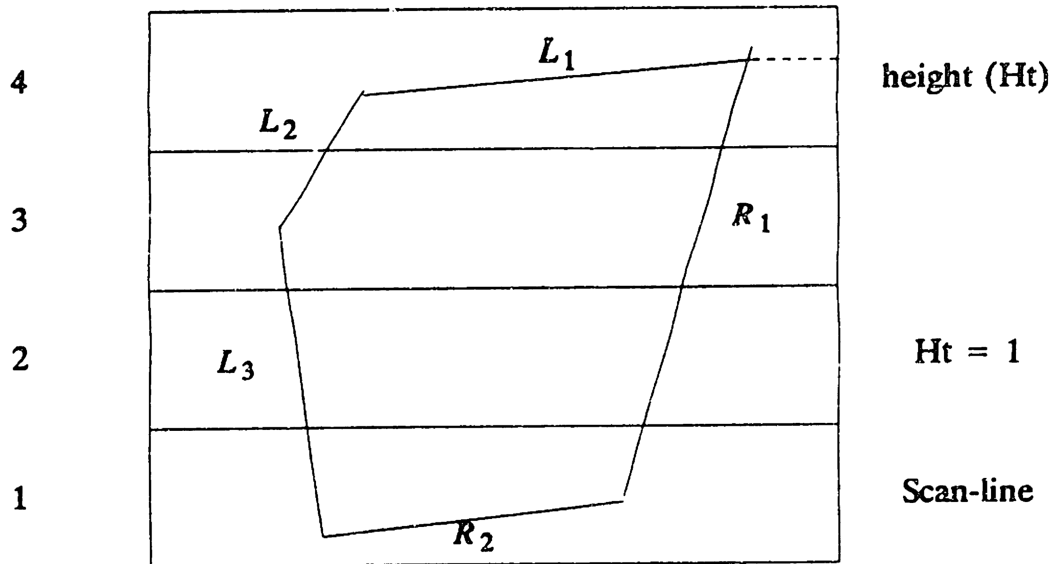


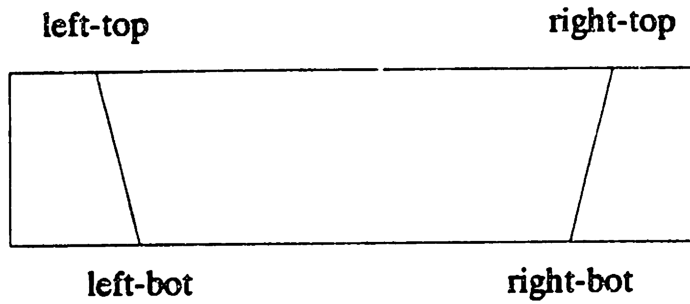
Figure 5.4 Polygon Example Spanning Four Scan Lines

and right edges R_1 and R_2. Along with each active edge structure is an attribute (ht) that specifies the vertical height of the current edge position from the bottom of the active scan line. Normally, the height associated with the active left and right edge positions is equal to one corresponding to the usual case of a polygon segment having full height coverage along a given scan line (e.g., scan line 2 in Figure 5.4). However, at the start or end of an edge (scan lines 4, 3, and 1), this height may be less than one and must be properly interpreted to determine the actual coverage of a polygon edge along the scan line, as needed by the anti-aliasing scan line segment processor. In addition, note that along scan lines containing a transition between two edges, there may be more than two edges active with potentially significant horizontal extent, which have to be properly accounted for. While the possible number of active left and right

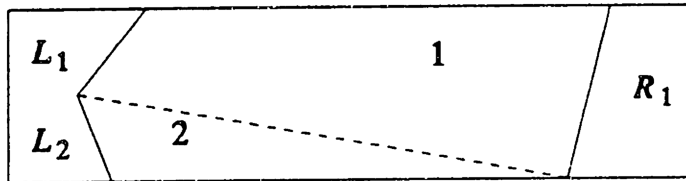
edges for one polygon may exceed a total of four along any given scan line, this number is rarely exceeded in most typical images.

In order to support proper filtering along the troublesome edges discussed above, each tiler maintains two edge structures per side containing the top and bottom attributes of each edge defining a scan line segment. Normally, as each polygon is tiled from top to bottom, the attributes specifying the current left and right bottom edge position and color are copied to the top edge structures, and the new bottom edge positions and color information are calculated by updating the current attributes with their corresponding increments. These four edge positions define a trapezoid with sloping sides and top and bottom edges aligned with the current scan line. This is illustrated in Figure 5.5 (a) for the case of scan line 2 in the example of Figure 5.4. However, at the two extremes of any polygon edge, the active edges within the scan line containing the given end vertex may define an arbitrary polygon segment with up to six sides (e.g., scan line 3 in Figure 5.4), assuming we limit the number of active edges within a given scan line to four. In order to simplify the subsequent process of polygon shading interpolation and filtering, the tiler always divides these polygons into two polygonal sections and invokes the segment processor twice for the given scan line. This is illustrated in Figure 5.5 (b) for the case of scan line 3 in the example of Figure 5.4.

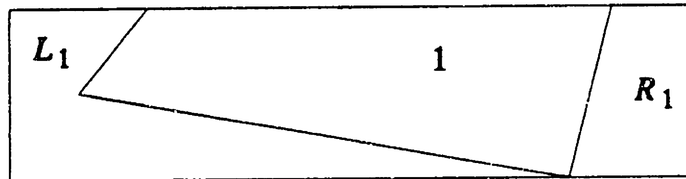
Given any convex polygon, the Gouraud or Phong tiler will generate polygonal scan line segments defined by four vertices (left-top, left-bot, right-top, right-bot), which are to be scan converted into a frame buffer. Each vertex is defined by its



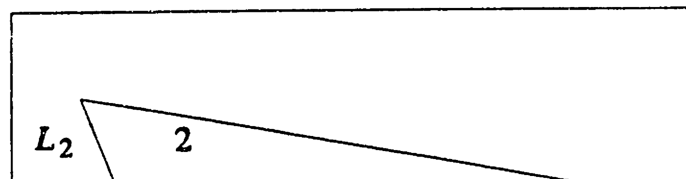
(a) Scan-Line Segment with Full Height Coverage



(b) Scan-line Segment at a Vertex



(c) Section 1 of Scan-line Segment (b)



(d) Section 2 of Scan-line Segment (b)

Figure 5.5 Examples of Typical Scan-line Segments

corresponding edge structure, which specifies its X-position, vertical height, and color information. In addition, Phong polygons also contain up to sixteen highlight normals with each vertex definition. The exact method used to scan convert these polygonal scan segments into a frame buffer, while performing anti-aliasing, hidden surface removal, and intensity blending depends on the choice of scan conversion processor. Four different implemented techniques for performing these functions are presented below.

5.6. Scan Conversion Method 1: Area Coverage

One of the simplest methods of performing anti-aliased scan conversion, while at the same time removing hidden surfaces, is to render surfaces in front-to-back order, keeping track of the accumulated area coverage at each pixel. Here, a pixel is defined as a square region having finite area with its center located at the corresponding image point. As surfaces are scan converted into a frame buffer, the area of a given surface fragment within each pixel is calculated and used to scale the fragment intensity accordingly. If the current pixel position is empty, then the fragment's area-weighted intensity is stored at the corresponding frame buffer location along with its area coverage. Whenever an attempt is made to modify a pixel location that has been previously written by a visible surface fragment, the existing pixel coverage information is used to blend together the new fragment intensity with the current pixel intensity. Thus, after all polygons have been scan converted into the frame buffer, the final intensity at each pixel is the weighted sum of the intensity contribution of the closest polygon fragments.

This rendering technique is somewhat analogous to Catmull's anti-aliasing scan-line algorithm [48], except that no front-to-back polygon clipping operation is performed at each pixel. The present technique takes advantage of the likely distribution of polygons in a scene to avoid this expensive clipping operation. For example, in typical scenes most of the edges are shared between polygons making up the same object. The pixels affected by such edges will be completely covered by these polygons, thereby preventing any other hidden surface behind these polygons from contributing to the pixel. Furthermore, along the silhouette edges of objects, the visible polygon fragment within a pixel behind the silhouette edge usually comes from a single surface. These two typical pixel situations are illustrated below in Figure 5.6.

The implementation of this scan conversion processor consists of a scan line segment pre-processor, followed by a shader-interpolator, and finally, a pixel integrator

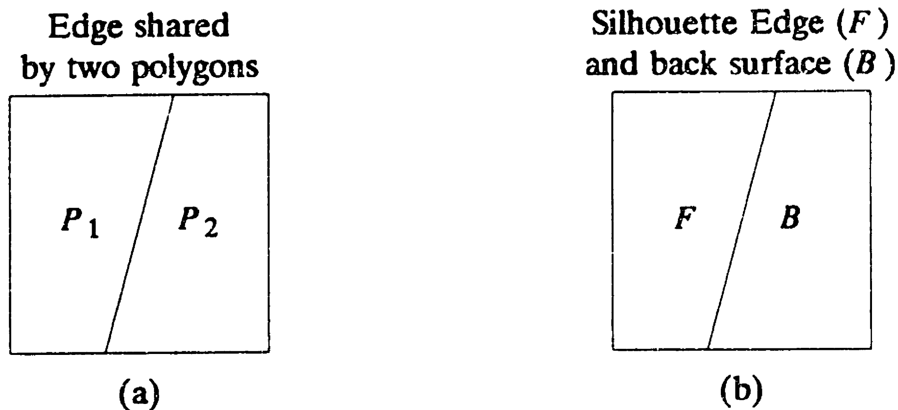


Figure 5.6 Two Typical Pixel Coverage Situations

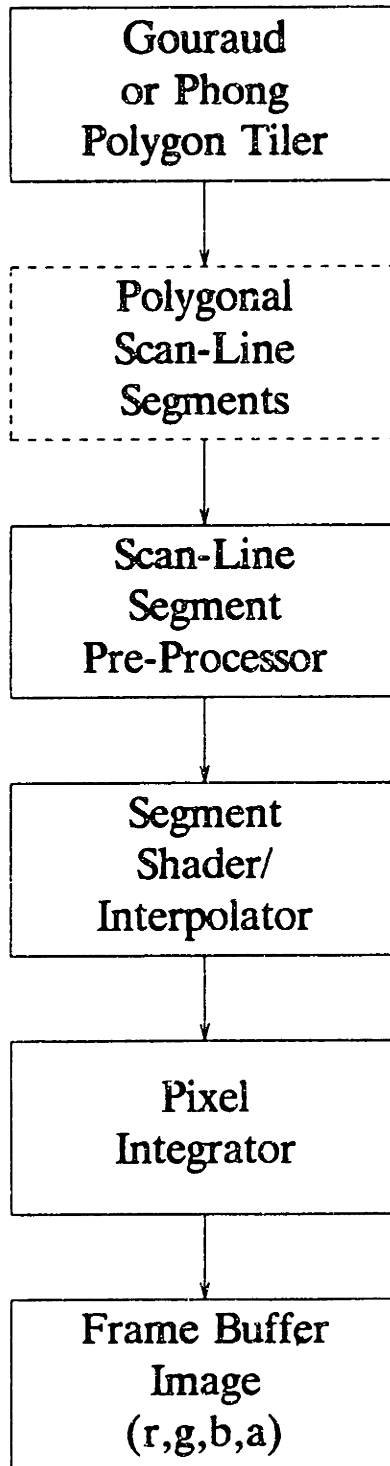


Figure 5.7 Segment Processor I Block Diagram

(Figure 5.7). Polygonal scan line segments are passed to the scan conversion processor by the preceding Gouraud (or Phong) polygon tiler. Since these segments can be of arbitrary shape, and since in this case, we are only concerned with the percentage of area coverage within each pixel affected by these segments, the scan line segment pre-processor takes each polygonal segment given and converts it into normalized trapezoidal spans. These spans are characterized by having their bottom edge aligned with the active scan line, possibly sloping top edge, and sides perpendicular to the bottom edge, as shown below in Figure 5.8. Given that each polygonal segment contains four vertices, the segment pre-processor produces at most three such trapezoidal spans for each scan line segment given. In most cases, these three regions take the form shown in Figure 5.9 (a), corresponding to the typical case of a horizontal polygon band generated by the tiler as it slices the polygon from top to bottom in scan line fashion. Another example of this pre-processing step is shown in Figure 5.9 (b), which corresponds to the polygonal segment depicted in Figure 5.5 (b).

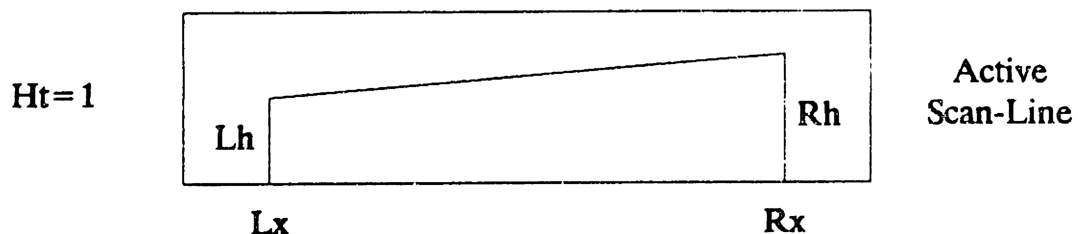
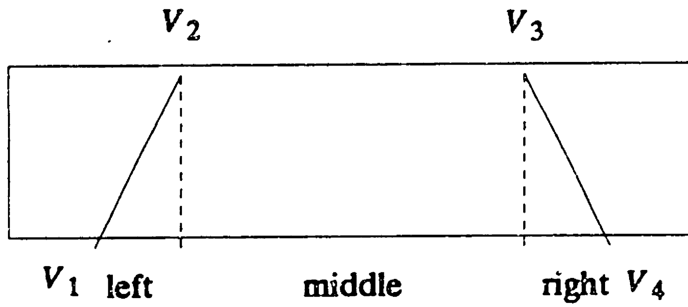
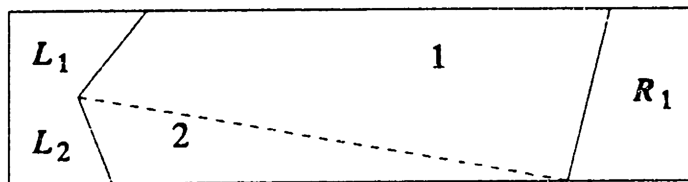


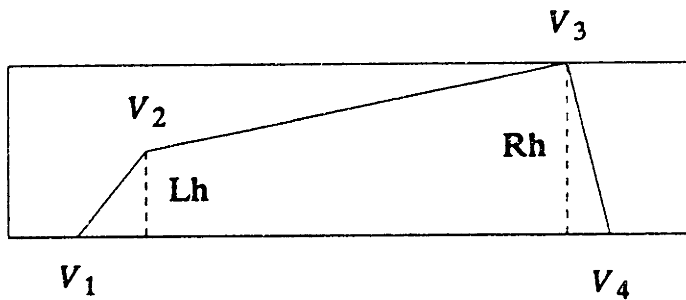
Figure 5.8 Normalized Trapezoidal Scan Segment



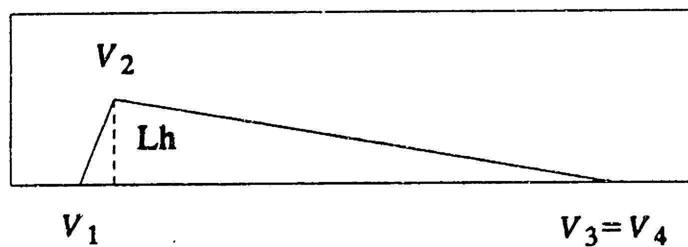
(a) Horizontal Segment Divided Into Three Spans



(b) Scan-line Segment at a Vertex



(c) Section 1 of Scan-line Segment (b)



(d) Section 2 of Scan-line Segment (b)

Figure 5.9 Normalized Trapezoidal Scan Segment Examples

Each trapezoidal region is then passed to the shader-interpolator, which further subdivides each trapezoidal region into horizontal sections to determine the coverage and shading of the polygon fragment along each pixel location. Each region is described by its end points, consisting of an X-position, vertical height (0 to 1) above the active scan line, color shading information, and optional highlight normal vectors. All shading and pixel coverage calculations are performed incrementally along the scan line by first computing the corresponding color, optional highlight normal, and height increments, given the values at the two ends of the span, and then using these to update the current values at each pixel as the scan line is traversed from left to right.

Final pixel blending is performed by the pixel integrator, which is given the interpolated polygon shade and optional highlight normal vectors, along with the actual area coverage of the polygon fragment for the given pixel. The pixel integrator first checks the current pixel coverage information to determine whether to perform a write or read-modify-write operation at the corresponding pixel location in the frame buffer. If the pixel is empty (coverage = 0), it simply scales the total polygon intensity by its area and writes the result into the appropriate frame buffer position, along with the coverage information. Otherwise, if the pixel is not completely covered (coverage < 1), it blends the current fragment intensity with the existing pixel intensity and calculates a new total pixel coverage according to the blending formula

$$I_{new} = I_{old} + A_{cov} I_{cur}$$

$$A_{new} = min(A_{old} + A_{cur}, 1)$$

$$A_{cov} = \begin{cases} A_{cur}, & A_{cur} + A_{old} \leq 1 \\ 1 - A_{old}, & A_{cur} + A_{old} > 1 \end{cases}$$

where I_{old}, A_{old} is the existing pixel intensity and coverage, I_{cur}, A_{cur} is the current surface fragment intensity and potential pixel coverage, and I_{new}, A_{new} is the resulting total pixel intensity and coverage.

This simple technique of blending polygon fragments at each pixel by accumulating their respective coverage areas performs the needed task of hidden surface removal, while at the same time, providing a mechanism for performing an anti-aliasing filtering operation. By the nature of the rendering order of polygons, a large percentage of pixels will be correctly rendered, since consecutive updates at a given pixel most likely come from adjoining polygons, which completely cover the pixel. Its main advantage is that it produces good quality images, as compared to non-filtered images, at moderate computational cost and memory requirements. The coverage information at each pixel is currently stored at 8-bits of precision, which is more than enough to support anti-aliased scan conversion, and actually could be reduced to fewer bits if necessary for a given implementation. The relative simplicity of the algorithm permits easy extension into a hardware implementation for real-time performance.

The main disadvantage with this simple area coverage technique is that it does not handle complex pixel situations correctly, due to the lack of geometry information available at each pixel. Figure 5.10 demonstrates a complex pixel consisting of a foreground silhouette edge, F, partly obscuring a background surface, B. Given that fragment F is rendered first, followed by fragment B, the final pixel intensity will be approximately shared by the two fragments (i.e, $I_{total} = 0.5 I_F + 0.5 I_B$) and the pixel will be marked fully covered, which of course is incorrect. To correctly render such a

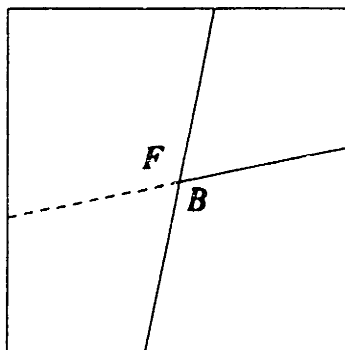


Figure 5.10 Complex Pixel Example (edge F over background B)

complex pixel requires additional information about the actual pixel region covered by a given polygon fragment, or a front-to-back clipping operation (as done by Catmull [48]) to determine the actual *visible* polygon fragments within each pixel.

As a result of this lack of sub-pixel geometric information, the simple area coverage technique is also unsuitable for rendering a virtual image tree containing sub-levels (i.e., reflection or refraction components.) The fundamental problem is that not enough information is available within each pixel to correctly eliminate hidden surfaces, while simultaneously blending multiple reflected and refracted fragments. However, even with its deficiencies, this rendering technique can correctly render many different images with simple reflections, refractions, and shadows at low computational expense and minimal memory requirements.

A simple extension of the pixel coverage idea provides a mechanism for simulating non-linear transparency, which allows the simulation of light scattering through translucent materials. During vertex shading calculations, the Image Generation Processor

computes a non-linear transparency factor, T, (see section 3.7) and includes it along with the vertex color information (R-G-B-T) passed to the scan conversion processor. The pixel integrator uses this transparency factor to modify the pixel coverage value, which allows a subsequent surface rendered behind the current transparent surface to contribute to the pixel's intensity. Thus, for transparent surfaces, the new pixel intensity and coverage information are computed by

$$I_{new} = I_{old} + A_{cov} I_{cur}$$

$$A_{new} = min(A_{old} + \bar{A}_{cur}, 1)$$

where

$$\bar{A}_{cur} = A_{cur}(1 - T)$$

$$A_{cov} = \begin{cases} \bar{A}_{cur}, & \bar{A}_{cur} + A_{old} \le 1 \\ 1 - A_{old}, & \bar{A}_{cur} + A_{old} > 1 \end{cases}$$

where T is the current surface transparency factor, I_{old}, A_{old} is the existing pixel intensity and coverage, I_{cur}, A_{cur} is the current surface fragment intensity and potential pixel coverage, and I_{new}, A_{new} is the resulting total pixel intensity and coverage. Thus, while a transparent surface may have full area coverage over a given pixel, its resulting pixel coverage may allow subsequent surfaces to modify the pixel information.

5.7. Scan Conversion Method 2: Pixel Mask

Increasing the effective resolution at which an image is rendered is known to be a simple technique for performing a filtering operation prior to resampling at the display resolution. A simple way of achieving this effective increase in resolution, without actually rendering the image at this resolution, is to change the definition of a picture

element from a single infinitesimal dot to an array of points. In effect, each display pixel is subdivided into an array of $n \times m$ pixels covering a finite area, as shown below in Figure 5.11. To compute the coverage of a given polygon fragment within a display pixel, all that is required is a mechanism for counting the number of subpixels covered by the fragment. This subpixel count then can be used to scale the intensity of the given polygon fragment to determine its approximate contribution to the display pixel. In addition, by saving the state of each subpixel in the array (pixel mask), subsequent polygon fragments falling within the same display pixel area can be clipped out behind regions occupied by previous fragments. Thus, clipping one polygon fragment against another becomes a simple boolean operation.

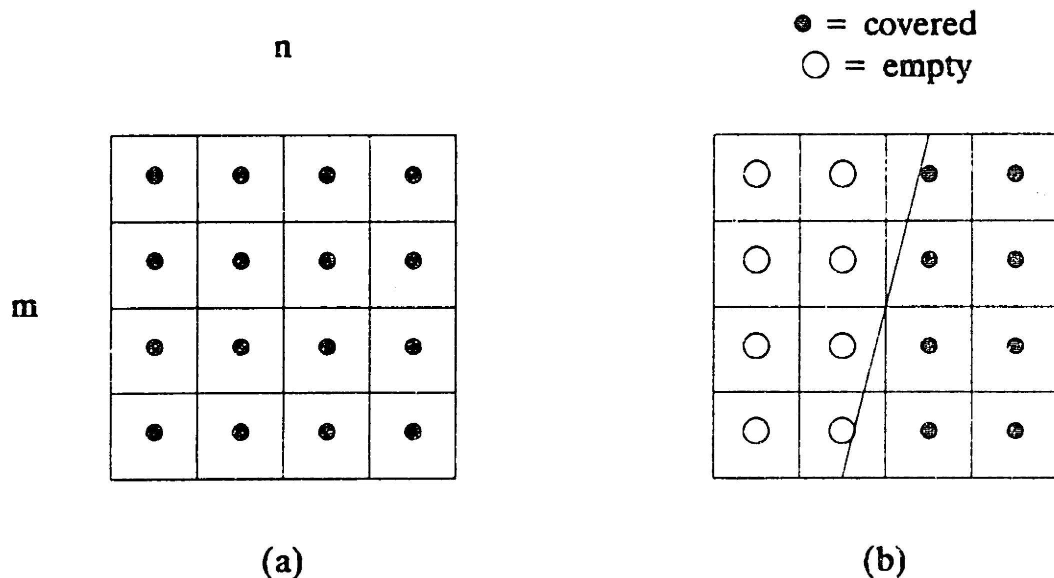


Figure 5.11 Subdividing a Pixel Into $n \times m$ Subpixels

The number of subpixels used in the array will significantly influence the quality of the final image, and the resulting storage requirements. It also may be desirable to have the size of the array be a power of 2 for efficient computation and storage reasons. Two array sizes previously used by other researchers have been 8×8 [49] and 8×4 [24]. The subpixel size chosen here is 4×4, considering that it increases the effective display resolution by a factor of 16 and results in a moderate increase in storage requirement at each pixel. Thus, assuming 24-bits of color, a total of 40-bits (or 5 bytes) are needed per pixel (as opposed to 88-bits or 56-bits), which corresponds to a 1.25 Mega-byte frame buffer memory requirement at 512×512 display resolution.

The implementation of this scan conversion processor is similar in principle to the one presented in the previous section and depicted in Figure 5.7. It consists of a scan line segment pre-processor, followed by a shader-interpolator, and finally, a pixel integrator. Polygonal scan line segments passed to the scan conversion processor by the preceding Gouraud (or Phong) tiler are characterized by four edges defined by their corresponding vertex X-coordinate and vertical height above the active scan line. Figure 5.12 shows a possible polygonal segment corresponding to the scan line along the top edge of a polygon. Since vectors, and since any given edge can span a considerable distance, the segment pre-processor subdivides each polygonal scan line segment into three (possibly zero-length) regions. Each sub-segment is then passed to the shader-interpolator, which further subdivides each segment span into pieces covering a single pixel region, and interpolates the the color/normals as it traverses the scan line. Final

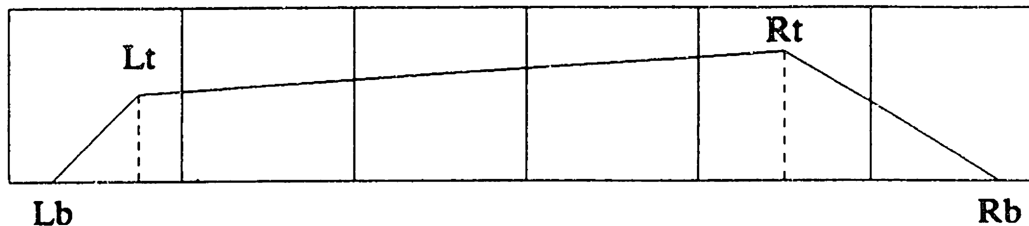


Figure 5.12 Scan Line Along Top Edge of a Polygon

pixel intensities are computed by the pixel integrator, which calculates the visible coverage of each polygon fragment within the pixel using pixel mask operations and blends the resulting weighted intensity with the current pixel intensity.

An underlying computation in this scan conversion method is finding the pixel mask corresponding to the subpixels covered by a given polygon fragment. This calculation is required at every display pixel location and, unlike the previous area-weighted method where the coverage area was computed incrementally along a scan line, cannot easily take advantage of scan line coherence, except in some special cases. Although various lookup table techniques have been previously implemented by others [49, 24] to compute this pixel mask, the approach taken here is a simple technique based on a two-dimensional clipping operation. Given the four vertices defining the polygonal scan line segment to be rendered, the segment pre-processor constructs a clipping region bounded by the four edges connecting the given vertices. This clipping region remains active for the entire scan line and used at each display pixel position to compute the pixel mask defining which subpixels are within the active segment boundary.

Since the majority of display pixels are completely covered by a single polygon fragment, the segment pre-processor attempts to establish a bounding box within each scan line in which pixel masks are trivially determined. Consider, for example, the polygon scan line fragment shown in Figure 5.13, which corresponds to a typical horizontal slice of a polygon with full height coverage. For cases like these (which are the most common), the segment pre-processor establishes the two scan line limits, *cp_xmin* and *cp_xmax*, within which the given polygon segment has full pixel coverage, corresponding to a pixel mask of all ones (M_{max}). While rendering, the pixel integrator simply tests the current display pixel position against these limits to check whether a pixel mask has to be computed or simply deduced. If the current display pixel position is outside these limits, then a pixel mask constructor is called which tests the center point of each subpixel region against the active clipping region. If inside, the subpixel bit is turned on, otherwise it is turned off.

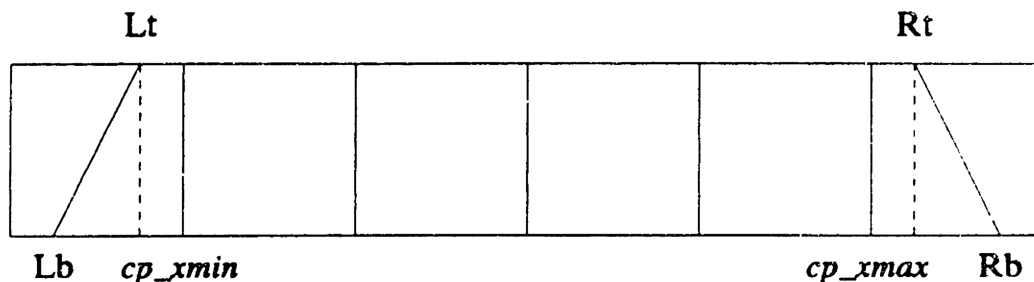


Figure 5.13 Typical Polygon Scan Line Segment With Ht = 1

Clipping polygon fragments behind a previously rendered fragment is a trivial boolean operation. At each display pixel, the pixel integrator checks the state of the existing pixel mask, M_{old}, to determine whether the current polygon fragment can possibly contribute to the intensity of the pixel. If the pixel is completely covered, as indicated by $M_{old} = M_{\max}$, then no attempt is made to compute a pixel mask for the current fragment or to modify the existing pixel location. Otherwise, a pixel mask is determined for the current polygon fragment, M_{cur}. To compute the visible portion of this fragment behind any previously rendered fragments in the pixel, a simple boolean AND operation is performed

$$M_{cov} = M_{cur} \cdot \bar{M}_{old}$$

where M_{cov} is the resulting visible mask for the given polygon fragment, and $\bar{M}_{old}$ represents the one's complement of the existing pixel mask. Assuming that M_{cov} is not null, a new pixel mask is calculated using a simple boolean OR operation

$$M_{new} = M_{cov} + M_{old},$$

which represents the composite coverage of all polygon fragments rendered at the pixel.

Given the pixel mask representing the visible polygon fragment within a pixel, the next step is to compute its corresponding area coverage. This amounts to a simple bit counting procedure to determine the number of subpixels affected. For efficiency reasons, the method used is to strip off four bits at a time and use this code as an index into a look-up table, which contains the actual bit count. The total bit sum is then converted into a normalized area coverage (0 to 1) and used to perform area-weighted

intensity blending at the pixel. The final pixel intensity is then found by the blending formula

$$I_{new} = I_{old} + f(M_{cov})I_{cur}$$

where $f(M_{cov})$ represents the visible coverage of the current polygon fragment, and I_{cur} is the fragments intensity.

This scan conversion method provides a relatively uncomplicated and fast anti-aliasing hidden surface mechanism. Within the accuracy provided by the bit mask size, it computes the exact contribution of all polygon fragments falling within each pixel, which allows computing a weighted sum of their intensity contributions. But above all, this scan conversion method provides the necessary support to render a wide range of virtual image trees with multiple levels of reflections and refractions, along with surface shadow polygons. The underlying motivation behind implementing this rendering technique was to provide a relatively fast and simple mechanism to render anti-aliased images described by virtual image trees. In addition, the goal was to make the rendering algorithm suitable for a moderate cost hardware implementation. Clearly, time-critical parts of the algorithm, such as the pixel integrator and pixel mask constructor are prime candidates for hardware implementation, resulting in considerable performance improvements.

The method used to render a virtual image tree using this scan conversion method is closely coupled with the fragment blending option supported by the Image Generation Processor discussed in the previous chapter. With this blending option, every sub-level polygon fragment in the virtual image tree, representing a reflected or

refracted component, is specified with all previous-level intensity components accumulated with its own intensity contribution. Thus, for example, a polygon fragment corresponding to a first level reflection would include its parent ambient, diffuse, and specular intensity components, $(I_a + I_d + I_s)_{parent}$, along with its own intensity contribution, $(I_a + I_d + I_s)_{poly}$, attenuated by its associated reflection coefficient, k_r. The total intensity specified with this polygon would then be

$$I_{tot} = (I_a + I_d + I_s)_{parent} + k_r(I_a + I_d + I_s)_{poly}$$

Surface shadow polygons at sub-levels of the image tree are similarly encoded and are specified immediately preceding their target polygon. Since the image generation process outputs polygon fragments in front-to-back depth order, starting with the deepest level in the tree, all that is needed to render the final image is for the scan conversion processor to tile polygons and perform hidden surface elimination. Pixel blending need only occur within complex pixels receiving contributions from multiple polygon fragments.

However, as mentioned in the previous chapter, this rendering technique does have its limitations without introducing additional pre-processing. First, it cannot correctly render overlapped regions in the image resulting from both reflected and refracted components on a single polygon. Since each branch of the tree is rendered independently, there is no way for the scan conversion processor to decide when to perform hidden surface elimination or when to blend. The decision is to always eliminate hidden surfaces, and thus, only the first branch of the tree rendered will be visible, except for those coincidental cases where the components between the two sub

branches do not overlap on the screen. Second, it cannot correctly combine shadow polygons with either a reflective or refractive surface. This would require that such a surface be divided into shadowed and non-shadowed regions and then traced separately. Lastly, complex shadow situations involving overlapped shadow regions caused by multiple light sources are not depicted correctly because of the overwriting nature of the algorithm.

It should be noted, however, that such limitations are not severe enough to consider implementing a relatively fast and simple image rendering technique that works well for most images. Several examples of images produced by this method are shown in the following chapter, demonstrating anti-aliasing, shadows, reflections, and refractions. Both faceted and Gouraud interpolated shading are supported, in addition to limited Phong specular highlights. This Phong limitation is simply due to the fact that only sixteen highlight vectors per vertex are currently supported. Since each sub-level polygon fragment in the virtual image tree must include all previous level shading information, there can be at most a total of sixteen highlights falling on a given polygon at any level of the tree. (I personally have not run up to this limitation!). A more elaborate image rendering technique is presented in section 5.8, which eliminates the blending limitations discussed earlier.

5.8. Scan Conversion Method 3: Hybrid Area/Pixel Mask

Although the pixel mask rendering technique presented above works well at performing subpixel polygon clipping and calculating approximate pixel coverage, the area

sampling technique discussed in section 5.5 actually works better at filtering edges. This is not surprising, since the latter analytic technique gives a better estimate to the amount of pixel area covered by a given polygon fragment. Images rendered by the pixel mask technique still exhibit substantial intensity quantization along the silhouette of objects, caused by the limited resolution of the pixel mask. The silhouette edges of objects are the most severely affected because of the sharp intensity discontinuity that occurs between the edge and its background. Increasing the subpixel array size would, of course, improve the rendition of edges, but also demands a substantial increase in computation and memory requirements.

A simple way of enhancing the filtering operation along silhouette edges is to carry along the *actual* area of a subpixel-sized edge fragment, as computed in method 1, in addition to its pixel mask. Then, whenever possible, the actual area is used instead of the bit count in the mask to compute the area-weighted intensity contribution to the pixel. For this implementation, this results in an additional 8-bits of storage per display pixel, raising the total storage requirement to 48-bits (or 6-bytes) per picture element.

Another advantage of saving both a pixel mask and pixel coverage value at each display pixel is to provide a mechanism to render surfaces behind transparent objects, as discussed in Scan Conversion Method 1. For this case, the pixel mask is used to perform efficient subpixel fragment clipping, while the pixel coverage indicates the accumulated intensity coverage from previously rendered surfaces. Thus, for example, a diffuse surface covering 50 percent of the pixel area would result in a pixel mask hav-

ing half of the subpixels covered and a pixel coverage of 0.5 (in the range 0 to 1). However, a transparent surface affects only the pixel coverage and intensity values and not the pixel mask.

5.9. Scan Conversion Method 4: Multi-Level Pixel Masks

All three rendering methods presented thus far simply perform hidden surface elimination and subpixel filtering operations to reduce aliasing artifacts in the final image. In all cases, it is assumed that each polygon given to be rendered contains the composite shading information for that region in the final image occupied by the polygon. For many scenes, it is relatively simple for a visible surface processor to produce polygonal patches representing homogeneous regions in the final image, which simply can be scan converted independently. These patches may correspond to simple visible diffuse surfaces, or composite regions representing the accumulation of multiple levels of reflection or refraction. Given that each surface patch contains a blend of all previous level intensity contributions, then all that is required to render the final image is to scan convert each patch and eliminate hidden surfaces.

As discussed earlier in this chapter, this rendering technique works well for many scenes but has several limitations. Since it performs no intensity blending operation between two different surface patches, except within a display pixel to blend subpixel fragments, it cannot add contributions from overlapped regions in the image corresponding to, for example, a reflection component and a refraction component visible at a given surface. To render such cases correctly requires a mechanism for decid-

ing when to perform hidden surface elimination and when to perform intensity blending operations. The only other way is to perform all hidden surface elimination calculations prior to the rendering phase, and then always blend overlapping polygons as they are scan converted into a frame buffer. Of course, this requires an extremely complicated recursive visible surface processor [44] that can trace fragmented polygonal beams of light through a scene with multiple reflective and refractive surfaces to determine *exactly* what surfaces are visible in the final image.

The decision made here was to develop a novel image rendering technique that could render complex environment situations specified by a virtual image tree. This rendering method would have to handle both hidden surface elimination at each level of the image tree, since polygons in the tree may overlap on the screen, and also perform intensity blending as the tree is traversed vertically. With this rendering concept, an image generation processor simply finds all potentially visible surfaces within any region in the image, while recursively tracing reflections and refractions, and builds a virtual image tree representation of the entire image. Each surface polygon in the image tree contains its own intensity contribution to the final image, along with any shadow polygons that have been projected onto its surface. To compute the final intensity within any given region in the image, the rendering process has to traverse the tree and add up all contributions from the multiple reflected and refracted components in sublevels of the tree. At the same time, it has to eliminate hidden surface fragments and perform anti-aliasing at each tree level.

The approach taken to implement this rendering technique was to extend the concepts developed in the previous three methods to support a multi-level, anti-aliasing, hidden surface algorithm. Scenes are assumed to be described by virtual image trees, with surfaces at each tree level sorted in front-to-back order. In addition, since polygons are rendered independently, it is further assumed that the image tree is traversed sequentially starting with the deepest branch associated with the closest polygon, and recursively working upwards towards the top level. This corresponds to the order in which the Image Generation Processor presented in the preceding chapter creates an image description. Each polygon in the tree is specified with its ambient, diffuse, and specular intensity contributions, possibly attenuated by any higher-level reflection or refraction coefficients. Shadow detail polygons are specified by including them prior to their target polygon specification. The example shown in Figure 5.14, illustrating a diffuse cube resting on a partly diffuse mirror, summarizes this image rendering procedure. For this case, the rendering order is

$$P_{12}, P_{13}, P_{14}, P_{21}, P_{22}, P_{23}, S_1, S_2, S_3, P_{11}$$

where S_1, S_2, S_3 correspond to shadow polygons cast onto the mirror by the diffuse cube.

The underlying idea behind this new rendering technique is to maintain sufficient information at every pixel to allow multi-level hidden surface elimination and anti-aliasing, in addition to inter-level intensity blending. The algorithm works with two different data types: "pixel-structs" (Figure 5.15) and "pixel-fragments" (Figure 5.16). Pixel-structs occupy an array equal in size and shape to the final image, whereas pixel-

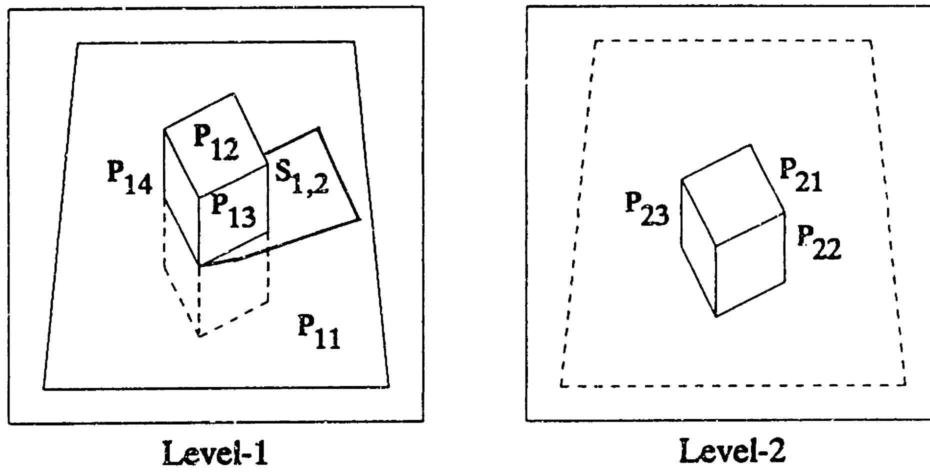


Figure 5.14(a) Cube on a Partly Diffuse Mirror

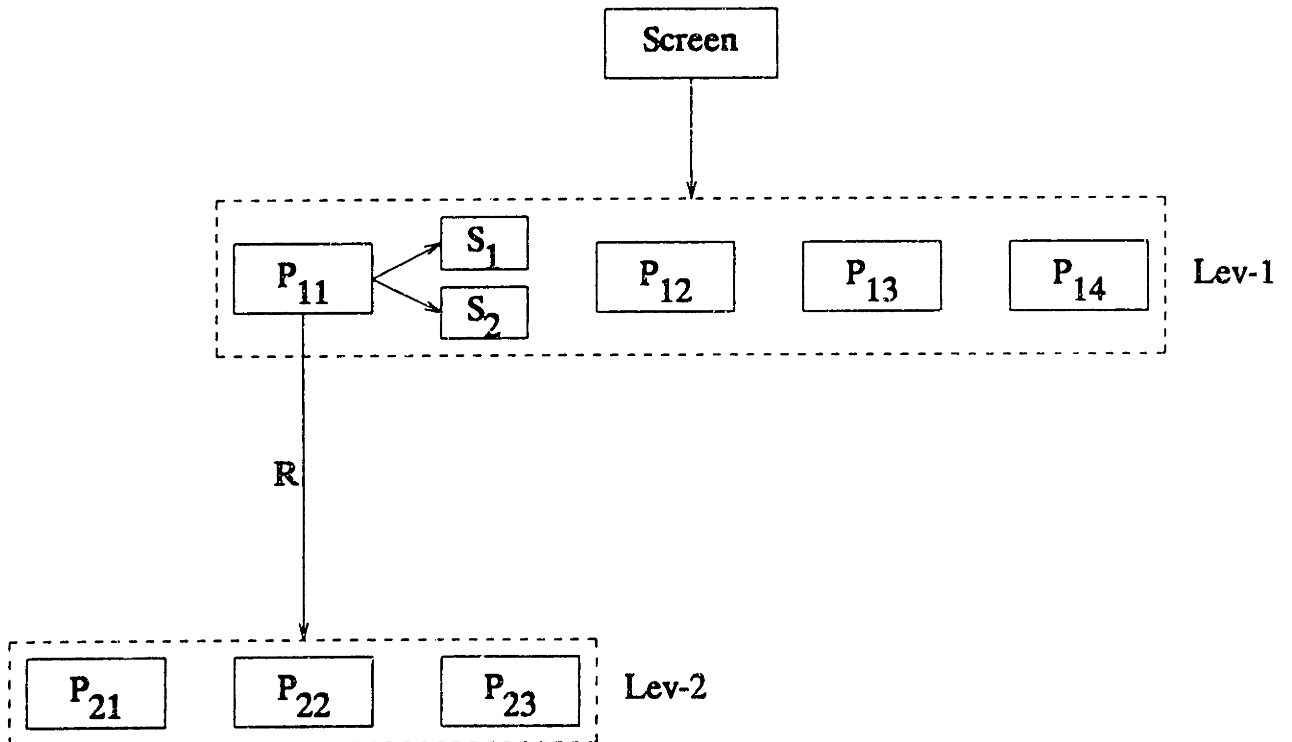


Figure 5.14(b) Virtual Image Tree

fragments are dynamic in nature and exist temporarily throughout the image as it is being rendered. Each pixel-struct occupies 8-bytes of storage, consisting of the accumulated pixel color (24-bits), a pixel-status byte (8-bits), and a possible fragment list pointer (32-bits). The pixel-status (Figure 5.17) contains a 5-bit field indicating the virtual image tree depth ($1 \cdots 31$) of the current polygon occupying the pixel, along with two flags specifying whether the pixel is fully covered or fragmented (i.e., complex). A current depth of zero indicates that the pixel is empty. A pixel is said to be "simple" if empty or completely covered, otherwise it is "complex". If a pixel is completely covered, the associated status flag is set and the current depth field (> 0) reflects the image tree depth at which the coverage occurred. Otherwise, if the pixel is complex at any tree level, the fragment list pointer (*flist*) points to a structure containing the accumulated pixel mask (*pmask*) and associated pixel coverage (*pcovr*).

Pixel-fragments are dynamically allocated and deallocated as needed, and serve two purposes. As indicated above, they are used to hold subpixel information when a given pixel is only partly covered by one or more polygon fragments. But in addition, these fragment structures are used to save the state of a complex pixel at a higher tree level while polygon fragments at lower tree levels are being rendered and blended with the same pixel. Whenever an attempt is made to modify a complex pixel tagged with a tree depth lower than the current polygon tree depth (cur-depth $>$ pix-depth), a new fragment structure is allocated to represent the pixel at the new depth, prior to performing any bit masking or intensity blending operations. Any previous lower depth fragment structures, corresponding to complex pixels active at higher levels in the vir-

```
typedef struct     /* 64-bits */
{
                   fragment     *flist;     /* fragment list ptr */
                   byte         r,g,b;      /* accum. pixel color */
                   pixelstat    pixstat;    /* pixel status */
} pixelstruct;
```

Figure 5.15 Pixel-Struct Definition

```
typedef struct     /* 64-bits */
{
                   fragment     *prevf;     /* previous level ptr */
                   pixelmask    pmask;      /* 4 x 4 bits */
                   pixelstat    pstat;      /* saved pixel status */
                   byte         pcovr;      /* pixel coverage */
} fragment;
```

Figure 5.16 Pixel-Fragment Definition

```
typedef struct     /* 8-bits */
{
                   unsigned     pdepth : 5;     /* pixel tree depth */
                   unsigned     covered : 1;    /* pixel covered flag */
                   unsigned     complex : 1;    /* pixel complex flag */
} pixelstat;
```

Figure 5.17 Pixel-Status Definition

tual image tree, are saved in a linked list. In general, the fragment list for a complex pixel at depth n in the tree is as illustrated in Figure 5.18, where any pixel-fragment at depth < n may be missing as a result of the image tree rendering order. Whenever a polygon fragment is to be blended with a complex pixel having a tree depth higher than the current depth (cur-depth < pix-depth), the active pixel fragment at the head

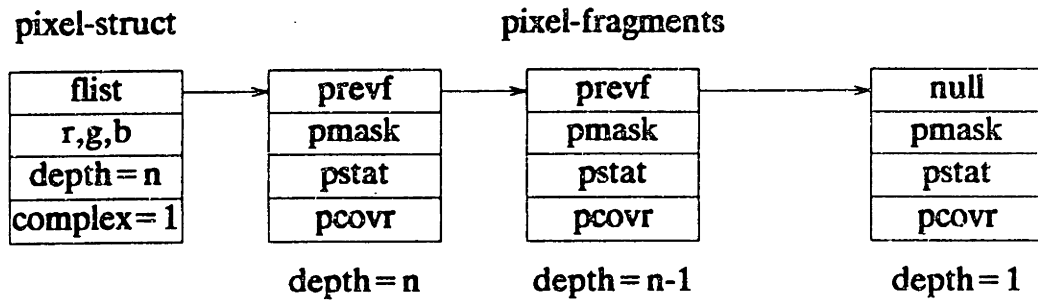


Figure 5.18 General Complex Pixel Fragment List

of the list is deallocated, and the pixel status information is retrieved from the saved fragment structure.

Generally, pixels are fully covered by a single object, and thus the active number of pixel-fragments at any one time is relatively small. However, as object surfaces are rendered into the frame buffer, most of the pixels along each polygon edge cause pixel-fragments to be created, while all pixels along internal object edges (those not along the silhouette of the object) will subsequently be completely covered when all polygons sharing the edge pixel are rendered. To save memory space, pixel fragments are deallocated, and the pixel is marked full, whenever a combined pixel mask indicates full coverage at any given level in the tree. For complex scenes, the total number of pixel-fragment requests are typically orders of magnitude greater than the maximum number of active fragments at any given time.

The implementation of this multi-level hidden surface algorithm follows directly from the pixel-mask rendering method outlined in section 5.6. In fact, the Gouraud and Phong polygons tilers, along with their associated scan line segment processors and

segment shader-interpolators are identical in both methods. The main difference involves the final pixel integrator algorithm, which implements the multi-level pixel fragment idea. The pixel integrator algorithm is powerful, yet relatively simple to implement. An outline of all possible pixel cases is given below. It is assumed that the frame buffer is initialized with all pixel-struct fields set to NULL:

Case 1: Pixel empty (pixel-depth = 0)

> This is one of the most common cases. If the given polygon fragment completely covers the pixel, the fragment color and tree depth are written at the corresponding pixel location, and the pixel is marked full. Otherwise, if the fragment's computed pixel mask is non-zero, a pixel-fragment structure is allocated to hold the new pixel mask (*pmask*) and coverage (*pcovr*), and the pixel is tagged complex. In addition, the area-weighted fragment intensity is computed and stored at the pixel along with its tree depth.

Case 2: Pixel full (current-depth ≥ pixel-depth)

> This is another common case. Once a pixel is marked fully covered at any depth n, then no polygon fragment at a tree depth $m \geq n$ is allowed to modify the pixel since it must be hidden. This assumes that polygons within any tree level are rendered in front-to-back order, and that sub-branches associated with any given polygon are traversed in reverse depth order (i.e., deepest branch first).

Case 3: Complex pixel (current-depth = pixel-depth)

> In this case, a polygon fragment is to be clipped and blended with a complex

pixel at the same tree depth. The clipping operation is performed by AND-ing together the bit mask computed for the given polygon fragment, M_{cur}, with the ONE's complement of the current pixel mask, M_{pix}.

$$M_{cov} = M_{cur} \cdot \bar{M}_{pix}$$

The resulting pixel mask, M_{cov}, indicates the amount of visible subpixel area covered by the given polygon fragment, which is used to blend together the new fragment color, I_{cur}, with the existing pixel color, I_{pix},

$$I_{tot} = I_{pix} + f(M_{cov}) I_{cur}$$

where $f(M_{cov})$ gives the resulting normalized fragment area (0 to 1). A new pixel mask is simply determined by a boolean OR operation

$$M_{tot} = M_{pix} + M_{cov}$$

Should this new pixel mask indicate that the pixel is fully covered ($M_{tot} = M_{\max}$), the current pixel-fragment structure is deallocated and the pixel is marked covered.

Case 4: Complex pixel (current-depth > pixel-depth)

This is the case when a sub-level polygon fragment is to be blended with a partly covered pixel at a higher tree level. The first step is to check if the fragment is visible within the pixel by performing a bit-mask clipping operation as described above. If the resulting coverage mask indicates non-zero coverage, the current pixel status byte, *pixstat*, is saved in the pixel-fragment structure and the current pixel depth is set to the new active tree depth. Then, the fragment's weighted color is blended with the current pixel color

and a new pixel coverage mask is determined. If this mask indicates full coverage, the pixel is simply tagged covered. Otherwise, a new pixel-fragment structure is allocated to hold the new pixel coverage information, and added to the head of the pixel-fragment list. This linked list contains the original higher level pixel information, along with any other higher level pixel-fragments.

Case 5: Blend at pixel (current-depth < pixel-depth)

This case provides the mechanism for blending a parent polygon at a higher level in the virtual image tree with all of its previously rendered children at lower levels. This involves first restoring the pixel information to the parent level before clipping and blending. If the current level pixel is complex, its fragment structure is deallocated and the pixel-structure is updated with the next fragment structure in the list (if any). The resulting restored pixel either can be empty, complex and at the same tree level with the parent, or complex and at a higher level that the current parent level. Depending on the state of the restored pixel, a procedure similar to that described above for case 1, 3, or 4 is followed to compute the fragment's coverage mask, M_{cov}, and new pixel mask M_{tot}, and possibly allocate a new fragment structure. In all cases, the current fragment color is attenuated by its visible coverage, $f(M_{cov})$, and blended with the existing pixel color.

Case 6: Restore pixel (current-depth < pixel-depth)

This last case provides the mechanism for blending both reflected and

refracted fragments associated with a given parent polygon (i.e., a polygon that has both reflection and refraction virtual image tree branches). The procedure used to perform this blending operation is to first render all reflected components, restore all parent polygon pixels to their original higher level state, and finally render the refracted components. The Image Generation Process outputs a specially tagged copy of the parent polygon between tracing its reflection and refraction branches to permit this restore operation. For these special polygons, a procedure similar to case 5 above is followed to restore all sub-level pixels associated with the given parent polygon to their original state without affecting the existing pixel color or coverage information.

Following the rendering of any virtual image tree, a number of first level pixel-fragments typically remain active as a result of any partially covered pixels along the silhouettes of objects. The current procedure is to perform a final pass through the entire image painting a background color. The final intensity at the remaining complex pixels is computed by blending the visible background color with the existing pixel color (using the pixel mask to determine the visible background coverage factor). In addition, the remaining pixel-fragment structure at each complex pixel is deallocated.

An alternative procedure would be to save the state of the final image in a file without painting the background color. This would allow rendering very complex scenes by subdiving the 3-D environment into manageable pieces [53] and creating a virtual image tree for each subsection independently. It also provides a mechanism for

rendering a scene composed of different object types (e.g., polygons, bicubic patches, fractals, etc..), where each object type is handled by its corresponding visible surface processor and then merged together by the same image rendering algorithm. The only requirement is that each visible surface processor produce image descriptions in the form of polygonal virtual image trees.

5.10. Virtual Frame Buffer

A common requirement of all rendering methods presented in this chapter is the availability of a frame buffer to hold the color and coverage information at each image point. Assuming that we allow 8-bits of resolution for each R-G-B color component, the amount of storage required per picture element ranges from four bytes (32-bits) for Method 1 to eight bytes for Method 4, corresponding to a frame buffer memory capacity 1 Mega byte and 2 Mega bytes, respectively, for a 512 × 512 image display resolution. While these memory requirements are not unreasonable to expect for either a dedicated frame buffer or as available user memory (real or virtual) on a moderate-sized computer, it was decided to implement a *virtual frame buffer* capable of supporting these demands, even on a small personal computer with limited memory.

Unlike scan line algorithms, which render the entire image in a single top-to-bottom pass through ALL polygons composing the scene, the rendering algorithms presented here treat each polygon independently and access sections of the image in arbitrary order. The frame buffer interface implemented provides the scan conversion process access to a single image scan line at a time, and maintains as many active scan

lines in physical memory as allowed by the particular host system. The entire frame buffer image is stored on disk and paged in and out of physical memory as needed. Thus, as far as the scan conversion processor is concerned, the entire frame buffer image is available for read/write access in arbitrary scan line order.

The virtual frame buffer design is flexible in tailoring different image size requirements on various host systems. It handles very large images (e.g., 2048 $\times$ 2048, or larger) on limited memory environments (e.g., DEC PDP-11's, or IBM-PC's), and minimizes the amount of virtual memory consumed on large main-frames. The maximum number of active scan lines held in physical memory is specified at run time and can range anywhere from one line to the number of lines in the final image.

Chapter 6

Results, Conclusions, and Extensions

6.1. Implementation Overview

The Image Generation Processor (*plytrace*) and Scan Conversion Processors (*rftb* 1, *rftb* 2, *rftb* 3, and *rftb* 4) described in this thesis have been coded in the *C* programming language to run on various machines. The current implementation runs either on the DEC VAX series, DEC PDP-11 series, or IBM-PC series, and can be easily ported to other systems that support *C*. For the most part, the implementation is device independent and requires no special purpose hardware to operate, although future hardware implementation has been considered in partitioning the rendering process. The Image Generation Processor does, however, provide interfacing to several graphics display terminals to allow real-time display of the virtual image tree creation process.

Throughout the implementation process, the goal has been to provide a user friendly and flexible environment in which to experiment with different aspects of the rendering process. Flexibility has been favored over optimization, and thus, one should expect considerable improvements to the run time statistics quoted in this thesis. The hierarchical scene description language provided to the user is powerful enough to describe complex scenes, and allows direct control over the entire scene

definition and image creation process. The user is given full control of the image rendering order, sorting algorithm, type of shading function (faceted, Gouraud, Phong), intensity blending option, shadow generation, and many other process functions. User manual pages for all programs implemented are included in the Appendix.

Examples of images created by the Image Generation Processor, and rendered by each of the Scan Conversion Processors described in the preceding chapter, are included at the end of Chapter 3. Considering that the difference in performance between the various scan conversion methods is relatively small (roughly a factor of 2 between methods 1 and 4), and since the total rendering time for complex images is dominated by the virtual image creation process, only Scan Conversion Processor 4 will we considered in analyzing the performance of the new rendering algorithm and in comparing its results with conventional ray tracing.

6.2. Comparison with Ray Tracing

Since the intent of this thesis was to present efficient rendenng algorithms that simulate the global illumination effects produced by ray tracing, a standard ray tracer was chosen to compare the run-time statistics, and resulting image quality, for various polygonal environments. The ray tracer used was initially written by a graduate student (John Wang) working in our group and then modified to accept the same scene description format used by the Image Generation Processor, and to use similar illumination models. For the most part, the ray tracer is patterned closely after Whitted's classical ray tracing algorithm [14], except that bounding boxes are used to enclose

complex objects, instead of spheres. In addition, the adaptive subdivision process discussed in Whitted's paper to reduce aliasing artifacts is not implemented; thus, only a single ray is traced per display pixel. Although various techniques could have been used to improve the efficiency of this ray tracer, such as the hierarchical or cellular decomposition techniques described in Chapter 2, these were not considered since similar enhancements also could be employed by the Image Generation Processor.

Figures 6.1 through 6.4 show images generated by the new rendering algorithm presented in this thesis, illustrating Gouraud shading, anti-aliasing, shadows, and planar reflections. The corresponding three-dimensional environments differ greatly in complexity, ranging from 68 polygons to over 24,000 polygons, and containing various object shapes and numbers of light sources. For each scene, a two-dimensional virtual image description file was created by the Image Generation Processor and then processed by Scan Conversion Processor 4 to render the final image at 512×512 display resolution, with 24-bits of color. For all images shown, the R-G-B separations were converted to luminance and output on an Autokon Model 8400 laser system, using a 65 dot/inch halftone screen. Because of the complexity of the scenes chosen, and the time needed to ray trace these environments using a classical ray tracing algorithm, only the mirror scene was generated at a resolution of 512×512 and is shown in Figure 6.1(c). All other test scenes were ray traced at an image resolution of 64×64 pixels and, therefore, are not illustrated.

Figure 6.1 depicts two polygonal objects resting on a partly diffused mirror surface, illuminated with a single point source from the right. The image illustrates reflec-

tion, shadows (Figure 6.1(b)), and overlapped shadowed-reflected regions. Figure 6.2 presents a gallery scene with mirrors at the front and rear walls, containing a reflective centerpiece, various diffuse objects, and four light sources. Reflections were traced to a maximum of 14 levels, and shadows were computed from the left rear light source (Figure 6.2(b)) and from all four light sources (Figure 6.2(c)). Note that the current implementation of the scan conversion processor does not correctly render overlapped shadow areas from multiple light sources, which results in uniform shading of all shadow regions on a given target surface. This is simply an implementation simplification and does not imply a theoretical limitation with the overall rendering approach.

Figures 6.3 and 6.4 depict an office scene with a mirror in the facing wall, including a bookcase with a mirrored back, two light sources, and an assortment of polygonal objects. The bookcase is composed of 98 objects and 2691 polygons, consisting of several dozen books, two soccer balls, a champagne glass, a large egg, a jet fighter, a butterfly, two plants, and several other objects. Figure 6.4 also includes a human skeleton (referred to as George by David Zeltzer), composed of 20 objects and 21045 polygons. For both images, reflections were traced to 3 levels, and shadows were computed from the foreground light source (Figure 6.3(b) and 6.4(b)) and from both light sources (Figure 6.3(c) and 6.4(c)).

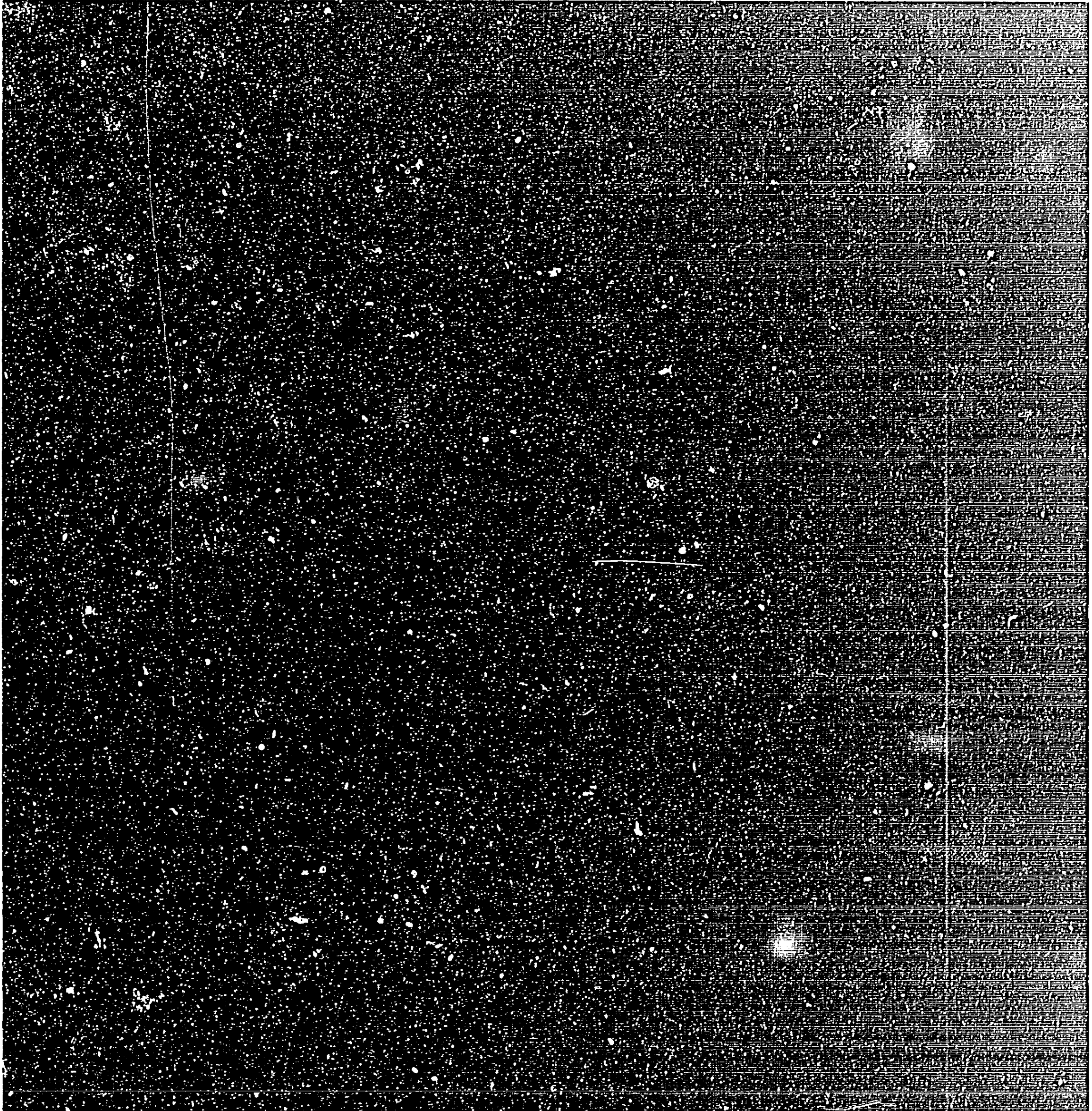


Figure 6.1(a) Mirror(NS) Scene : (VAX-11/785 Time = 1.4s + 45s)

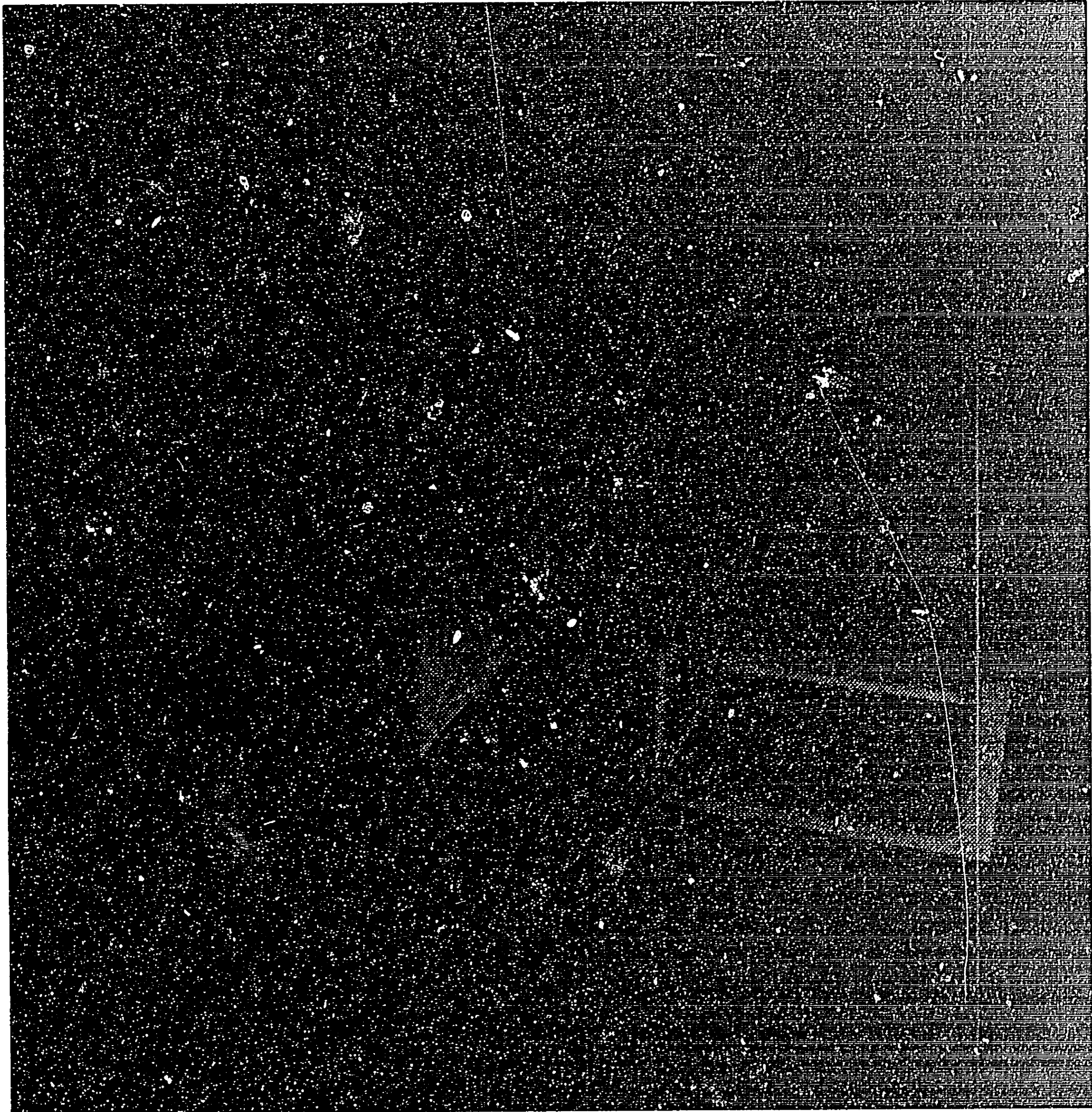


Figure 6.1(b) Mirror(1S) Scene : (VAX-11/785 Time = 3.2s + 55s)

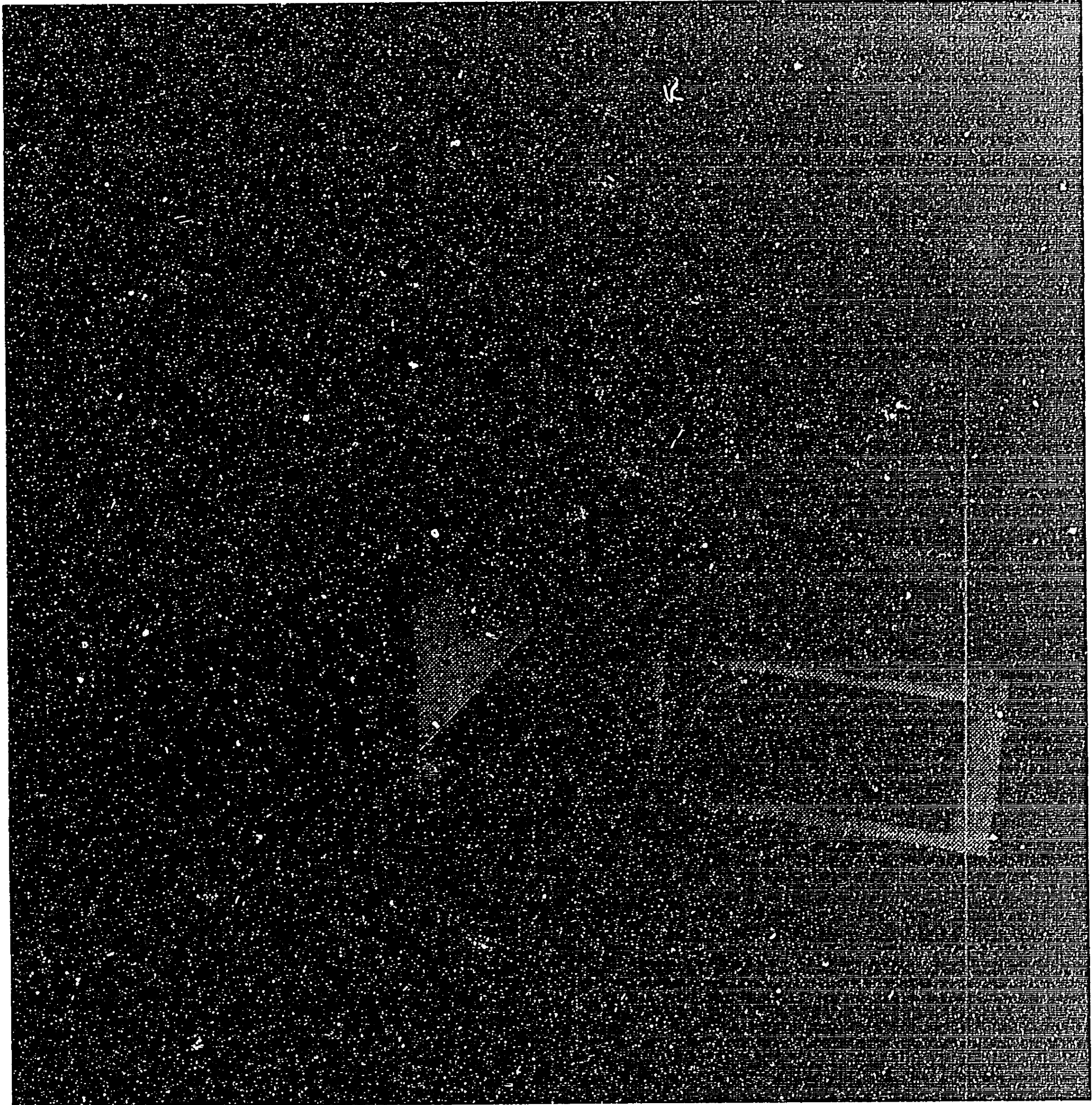


Figure 6.1(c) Mirror(1S) Scene : (Ray-Traced; VAX-11/785 Time = 1.75h)

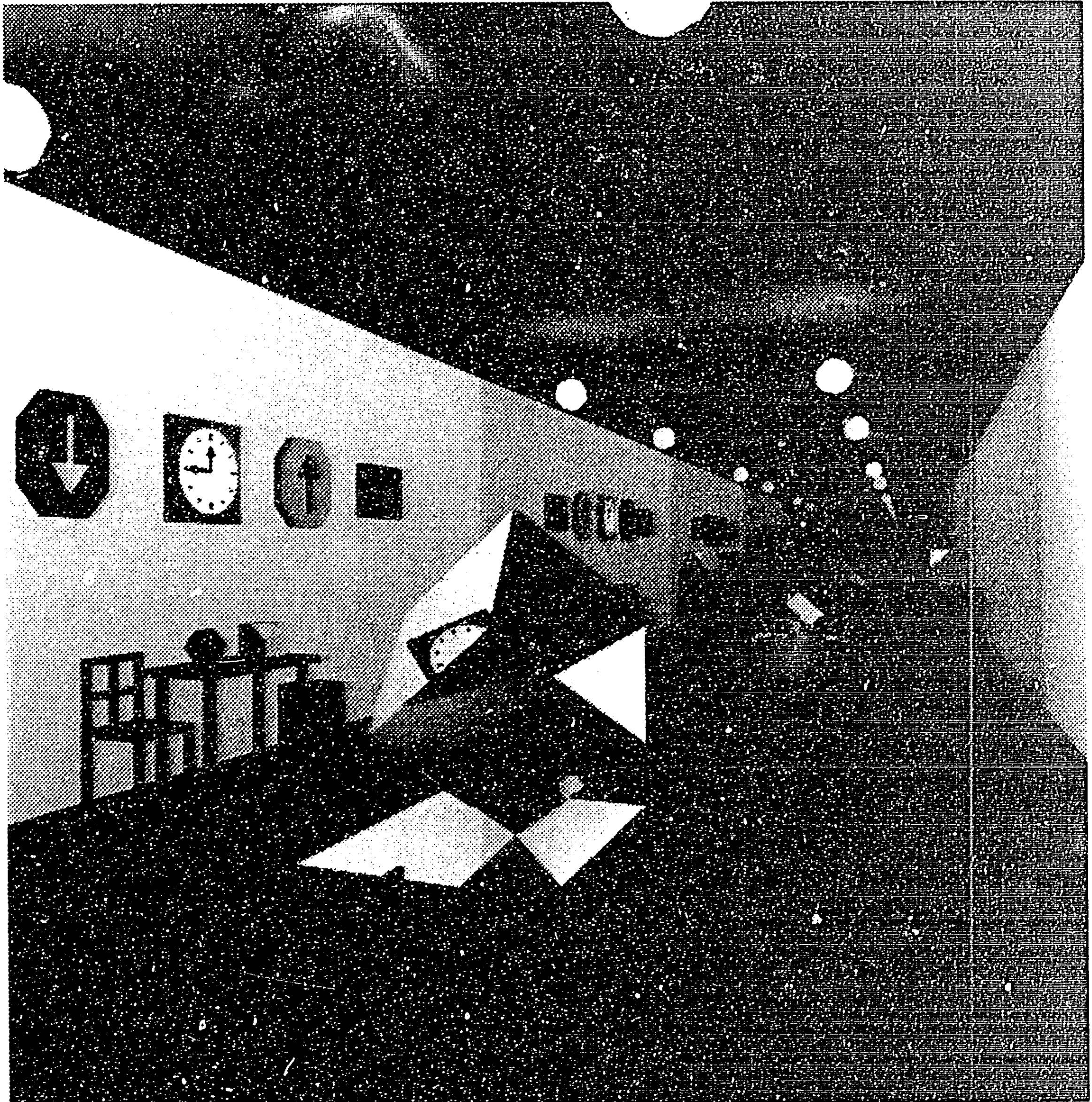


Figure 6.2(a) Gallery(NS) Scene : (VAX-11/785 Time = 3.2m + 3.6m)

Figure 6.2(b) Gallery(1S) Scene : (VAX-11/785 Time = 16m + 4.3m)

Figure 6.2(c) Gallery(4S) Scene : (VAX-11/785 Time = 55m + 6.8m)

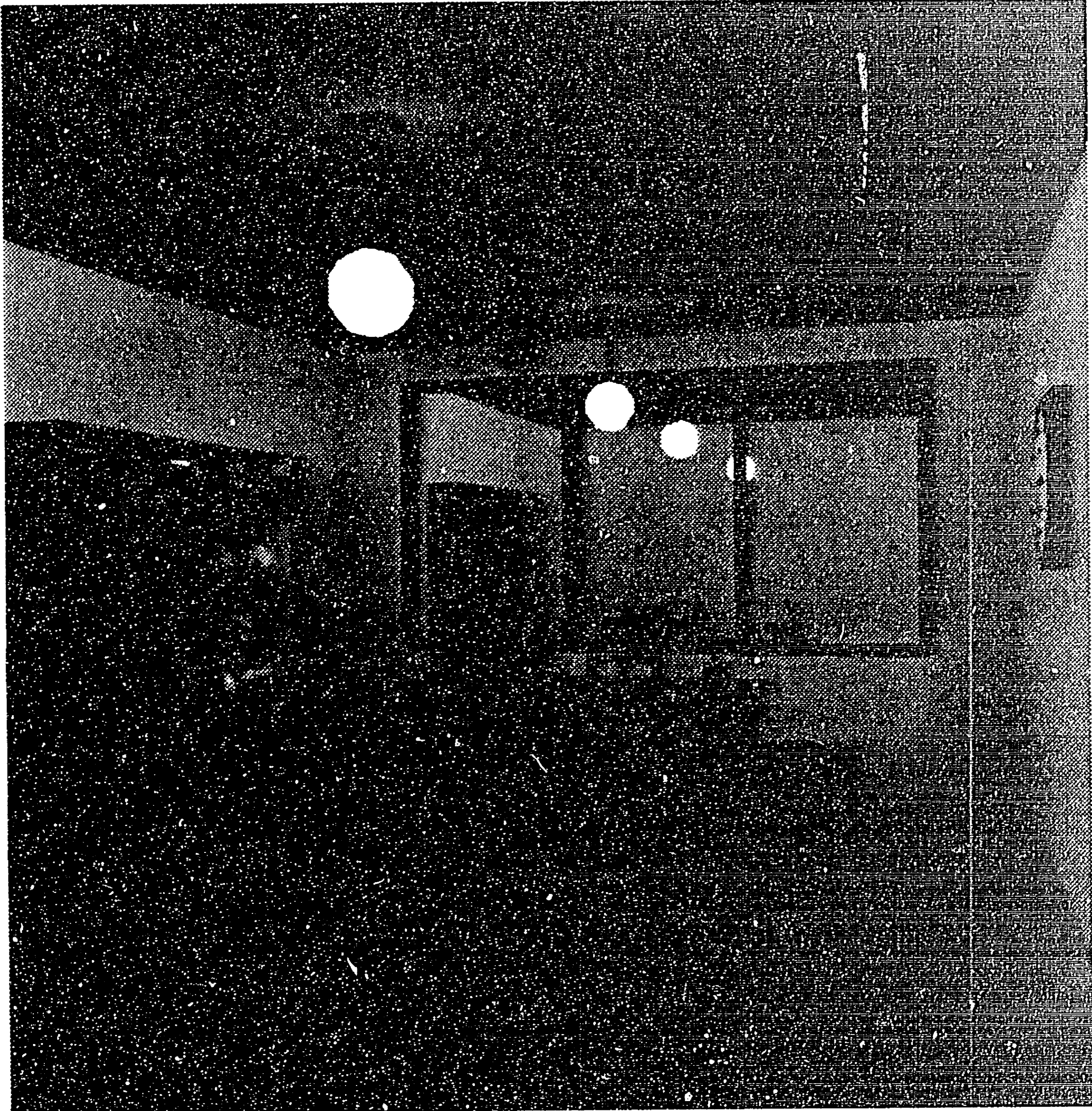


Figure 6.3(a) Office1(NS) Scene : (VAX-11/785 Time = 3m + 3.4m)

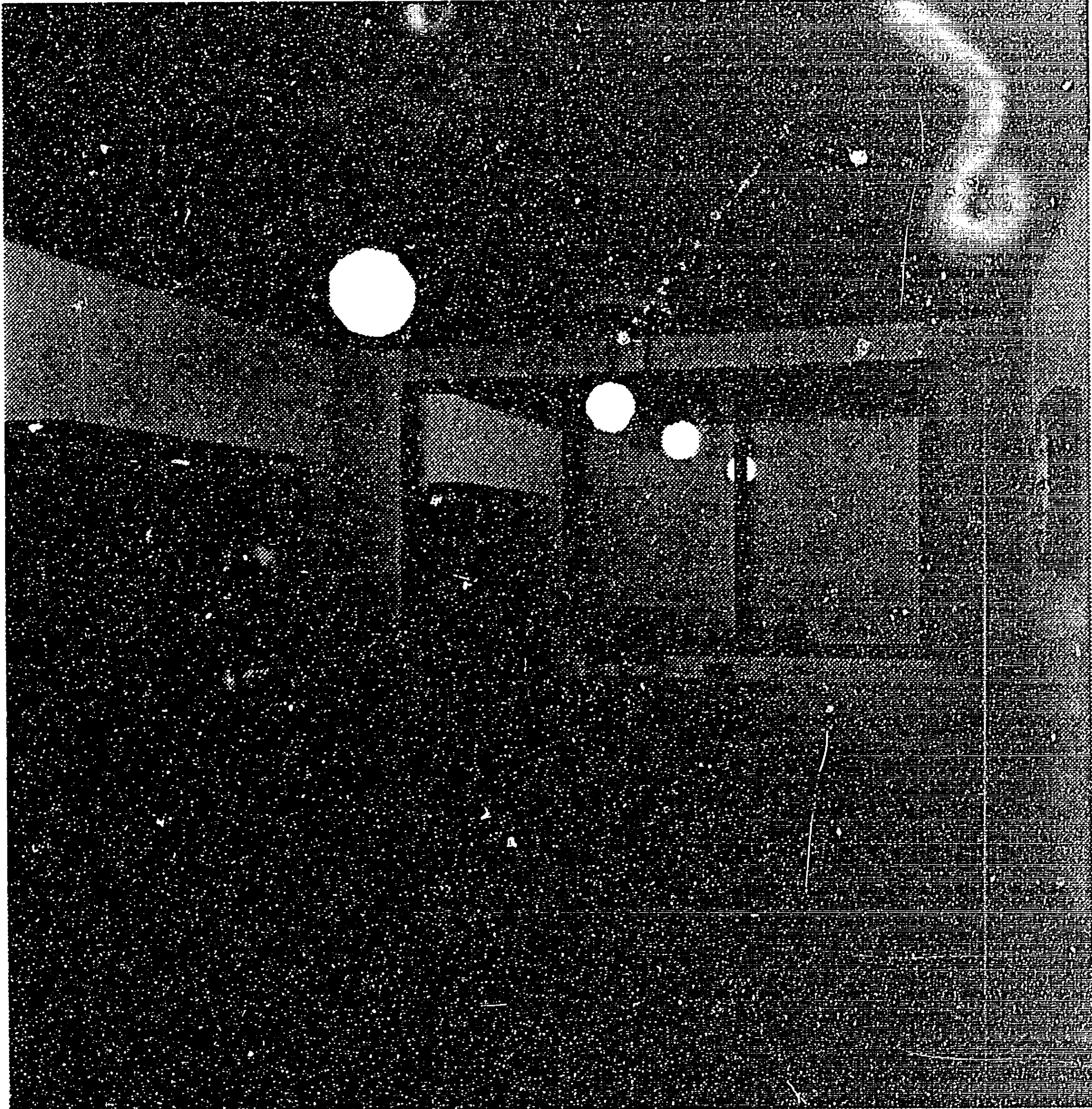


Figure 6.3(b) Office1(1S) Scene : (VAX-11/785 Time = 49m + 5.1m)

Figure 6.3(c) Office1(2S) Scene : (VAX-11/785 Time = 88m + 6.7m)

Figure 6.4(a) Office2(NS) Scene : (VAX-11/785 Time = 23m + 5.2m)

Figure 6.4(b) Office2(1S) Scene : (VAX-11/785 Time = 5.3h + 8.6m)

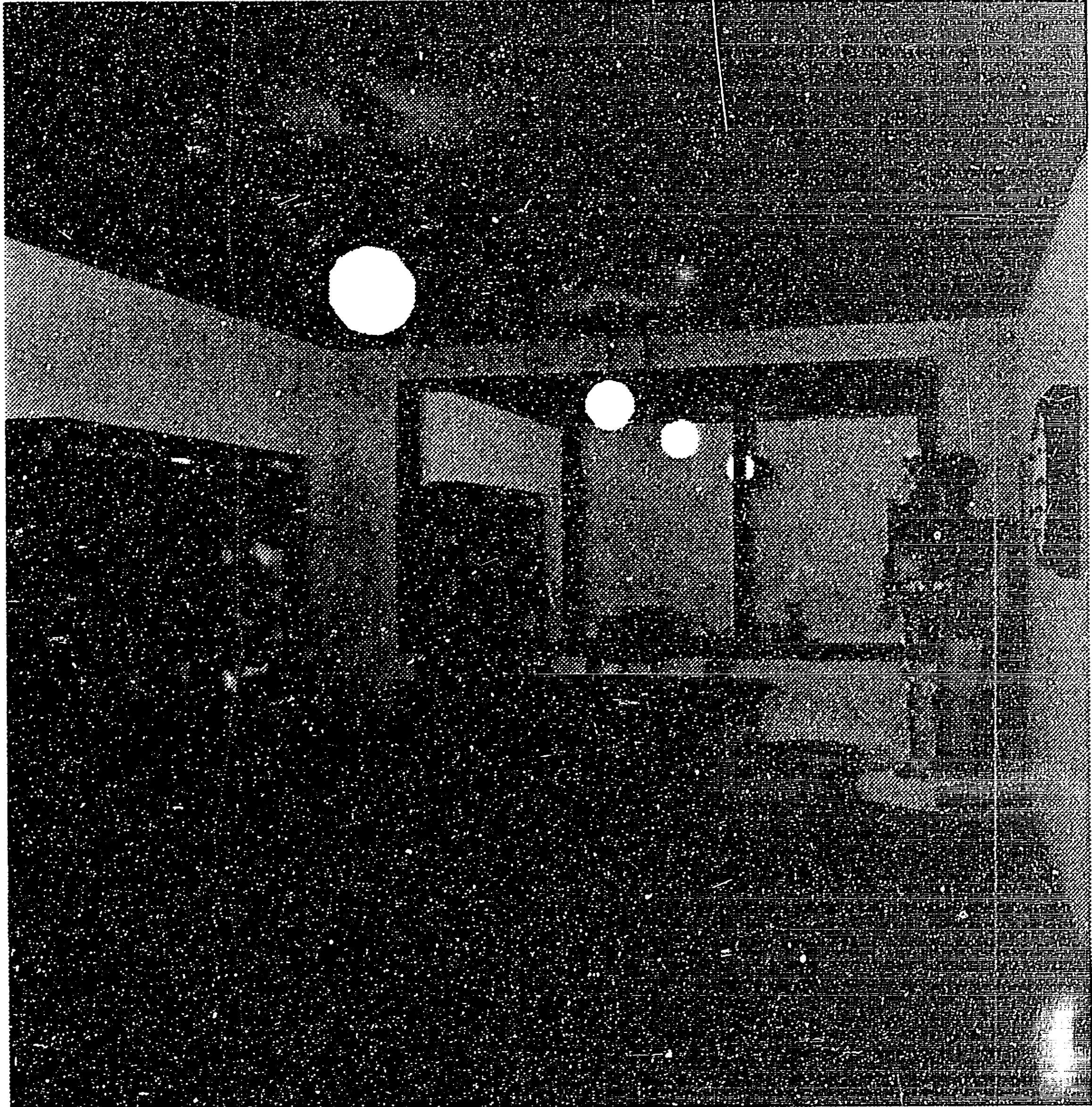


Figure 6.4(c) Office2(2S) Scene : (VAX-11/785 Time = 10h + 11.5m)

A summary of the DEC VAX-11/785 run-time statistics for all images is given in Table 6.1, including the number of objects and polygons in the scene description, the maximum recursion depth allowed, the resulting number of polygons in the virtual image description, the average ray tree depth, and the total processing times for the new algorithm and the standard ray tracer. For comparison purposes, the images were generated without shadows and with shadows from various light sources (NS indicates no shadows; 1S, 2S, 4S, indicates shadows computed from 1, 2, and 4 light sources, respectively). G-Time indicates the time taken by the Image Generation Processor to construct the two-dimensional virtual image description from the specified three-dimensional environment. R-time indicates the time taken by Scan Conversion Processor 4 to scan convert the virtual image description at 512 × 512 display resolution. The total image rendering time for the new algorithm is given by the sum of G-Time and R-Time. For run-time comparison purposes, the same environment descriptions for these images were processed by the standard ray tracer at a resolution of 64 × 64 pixels. The ray tracing time at this sub-resolution is indicated by RT-Time, and the corresponding rendering time for each image at 512 × 512 display resolution would be approximately 64 × RT-Time. Also included for each image is the average ray tree depth, ATRD.

For the polygonal environments tested, the new rendering algorithm is clearly superior to standard ray tracing in terms of both computational expense and resulting image quality. The relative run-time efficiency improvement of the new algorithm over ray tracing, as computed by

Figure	Scene	Total Objects	Total Polygons	Maximum Depth
6.1	Mirror	8	68	2
6.2	Gallery	68	629	14
6.3	Office1	194	3329	3
6.4	Office2	214	24374	3

Table 6.1(a) Environment Statistics for Figures 6.1-6.4

Scene	Polygons Tiled	G-Time	R-Time	ARTD	RT-Time	RI
Mirror(NS)	64	1.4s	45s	1.3	1.37h	106
Mirror(1S)	152	3.2s	55s	1.3	1.75h	108
Gallery(NS)	5627	3.2m	3.6m	1.4	17.3m × 64	163
Gallery(1S)	9426	16m	4.3m	1.4	32.1m × 64	101
Gallery(4S)	21179	55m	6.8m	1.4	74.1m × 64	77
Office1(NS)	6007	3m	3.4m	1.1	1.2h × 64	720
Office1(1S)	16513	49m	5.1m	1.1	2.2h × 64	156
Office1(2S)	26200	88m	6.7m	1.1	3.4h × 64	138
Office2(NS)	19500	23m	5.2m	1.1	7.0h × 64	953
Office2(1S)	44979	5.3h	8.6m	1.1	14.3h × 64	168
Office2(2S)	69720	10h	11.5m	1.1	20.5h × 64	127

G-Time = Image Generation Time, R-Time = Scan Conversion Time at 512 × 512 Image Resolution, and RT-Time = Ray Tracing Time.
NS = No shadows; 1S, 2S, 4S = Shadows from 1, 2, and 4 sources.
ARTD = Average ray tree depth, RI = Relative Improvement.

Table 6.1(b) VAX-11/785 Run-time Statistics for Figures 6.1-6.4

$$Relative\ Improvement = \frac{R_Time + G_Time}{RT_Time}$$

is summarized in Table 6.1 for all four images at 512 × 512 image resolution. As indicated, this relative improvement ranges from 75, for the Gallery(4S) image shown in Figure 6.2(c), to 950, for the Office2(NS) image shown in Figure 6.4(a), and increases with greater scene complexity. In addition, it should be noted that the ray tracing times quoted are for tracing a single ray through each display pixel, which leads to aliasing artifacts in the final image, as shown in Figure 6.1(c). To achieve the same effective image quality produced by the new rendering algorithm, the ray tracer would have to trace 16 rays per display pixel, corresponding to the 4 × 4 pixel mask used in Scan Conversion Processor 4, at least within complex pixels containing polygon edges. This would, of course, drastically increase the ray tracing times accordingly.

The overall efficiency improvement of the new rendering method over ray tracing depends on a number of factors, including image resolution, image coherence, scene complexity, and shadow generation. For example, the total time spent ray tracing an image is equal to the product of the time needed to compute the closest ray-surface intersection, the average ray tree size, and the image resolution. Thus, for example, doubling the horizontal and vertical image resolution increases the ray tracing time by a factor of 4. In contrast, the time taken by the Image Generation Processor to create a virtual image tree is resolution independent, which results in a linear increase in efficiency over ray tracing as a function of resolution. While the Scan Conversion Processor time is resolution dependent, its effect on the total image rendering time is minimal for complex scenes, as evident from the run-time statistics shown in Table 6.1. In

addition, the polygon tiling process can easily be offloaded to a special purpose processor, which would considerably reduce the total image rendering time for scenes of low to moderate complexity.

Image coherence and scene complexity, rather than resolution, determines the overall efficiency improvement of the new rendering method over ray tracing. For all test cases shown, the images contain large homogeneous regions which are effectively ray traced in parallel by the new rendering algorithm, as opposed to being sampled by many individual light rays using conventional ray tracing. Obviously, the effective number of rays traced in parallel increases with greater image coherence. One interesting observation, however, is that the relative improvement factor (RI in Table 6.1) for the scenes tested increased with greater scene complexity. Although, intuitively, one expects the resulting image coherence to decrease with increased scene complexity, it will generally be possible to find many homogeneous regions in the image that can be effectively ray traced in parallel. For the ray tracing case, increasing the number of polygons in the scene description implies that more ray-surface intersection calculations will have to be made, assuming that each ray traced is tested against all polygons in the environment. Thus, for example, the ray tracing times for the Gallery(NS) and Office1(NS) images increased linearly with the number of polygons in the scenes, taking into account the average ray tree depth in both cases. In contrast, for the new method, the rendering times and the total number of polygons tiled for both images remained about same.

Another important observation made from the run-time statistics is that the cost of calculating shadows is much higher for the new rendering algorithm than for ray tracing, although the overall efficiency improvement over ray tracing is still quite high. For the ray tracing case, the cost of computing shadows is proportional to the number of sight rays intersected by objects and the number of light sources considered in shadow testing. Thus, assuming all sight rays intersect objects, as was the case for the Gallery, Office1, and Office2 scenes, the computational time approximately doubles for each light source considered in shadow testing, since for each ray-surface intersection another ray is sent up to each light source to determine the total illumination at the intersection point. For the new method, however, the cost of computing shadows is proportional to the complexity of the scene, in addition to the number of light sources considered for shadow casting. As outlined in Chapter 4, shadow polygons are computed for each target polygon by casting all scene polygons that are contained in the shadow volume defined by the given light source and the target polygon. Clearly, the potential number of scene polygons found within any shadow volume increases with greater scene complexity. Thus, for example, the cost of computing shadows in the Gallery scene from each light source is approximately 5 times the image generation time without shadow generation. In contrast, this shadow cost factor is approximately 16 for the Office1 scene, and 13 for the Office2 scene. Note that, for both office scenes, most of the shadow calculation time is consumed in the bookcase region, where many surfaces obstruct each other.

6.3. Analysis of the New Rendering Method

In order to better understand where the computationally intensive operations are in the new rendering method, a run-time profile of the Image Generation Processor was taken for each image and summarized in Table 6.2. The entries in the table indicate the percentage of the total image generation time (G-Time) consumed by the corresponding function. The first category lists the major functions of the Image Generation Processor and includes the following: the percentage of time required to build the virtual image tree object list (*Addobj*); the percentage of time used to expand objects into a list of potentially visible polygons (*Expobj*); the percentage of time spent depth sorting the active polygon list (*Sortply*); the percentage of time required to compute shadow polygons (*Shadows*); the percentage of time used to calculate vertex shading according to the illumination model (*Shading*); and the total miscellaneous time, which includes reading the scene description and outputting the virtual image description to a file. The second category corresponds to computationally intensive operations used by higher level functions and include the following: the percentage of time required to check object bounding boxes against the active clipping volume or shadow volume (*ChkBBOX*); the percentage of time spent checking polygons against the active shadow volume (*ChkPLY*); the percentage of time spent actually clipping polygons to the active clipping volume or shadow volume (*Clipping*); and the percentage of time spent transforming object coordinates (*Transform*). *ChkBBOX* and *Clipping* are used by *Addobj* and *Shadows*, *ChkPLY* is used by *Shadows*, and *Transform* is used by *Addobj* and *Expobj*.

Scene	GNS	G1S	G4S	O1NS	O11S	O12S	O2NS	O21S	O22S
Addobj	39.5	5.2	1.4	2.5	0.1	0.0	0.5	0.0	0.0
Expobj	24.7	4.0	1.2	6.0	0.4	0.2	2.2	0.2	0.1
Sortply	6.5	1.2	0.4	55.5	3.0	1.8	83.3	7.5	4.9
Shadows	0	83.3	95.1	0.0	94.4	96.6	0.0	90.7	93.9
Shading	20.4	3.8	1.2	23.4	1.2	0.8	8.3	0.8	0.5
Misc	8.9	2.5	0.7	12.6	0.9	0.3	5.7	0.8	0.6
ChkBBOX	27.3	51.6	54.3	2.0	48.7	52.3	0.2	27.4	30.2
ChkPLY	0	21.5	25.4	0.0	33.8	32.2	0.0	50.5	46.3
Clipping	3.1	6.1	7.5	0.0	5.3	5.9	0.2	2.0	2.4
Transform	12.9	3.2	2.0	1.5	0.4	0.4	0.5	0.2	0.2

GNS = Gallery(NS), G1S = Gallery(1S), G4S = Gallery(4S).
O1NS = Office1(NS), O11S = Office1(1S), O12S = Office1(2S).
O2NS = Office2(NS), O21S = Office2(1S), O22S = Office2(2S).

Table 6.2 Run-time Profile for the Image Generation Processor

For the Gallery(NS) scene, about 40 percent of the total image generation time is taken up transforming object bounding boxes to the current virtual coordinate system (after each reflection), and checking these bounding boxes against the active clipping volume. Expanding the remaining objects into a polygon list, which includes transforming all object points and clipping polygons to the active clipping volume, consumes another 25 percent of the total time. The remaining time is taken up computing polygon vertex shades, sorting the active polygon list, and in other miscellaneous functions. For the Office1(NS) and Office2(NS) scenes, 55 to 85 percent of the total image generation time is required to depth sort the active polygon list, while the remaining time is mostly taken up by shading calculations. This to be expected for complex scenes, since the worst case Newell, Newell, and Sancha sorting time is

proportional to n^2, where n equals the number of polygons to be sorted. Note, however, that the ratio between the image generation times for Office1(NS) and Office2(NS) is equal to the ratio of the total polygon count in their respective scene descriptions (Table 6.1).

With shadow generation enabled, a large percentage of the total image generation time is consumed by the shadow processor, ranging from 83 to 97 percent. In addition, for each light source considered in shadow casting, the total image generation time increased by a factor of 5, 16, and 13, for the Gallery, Office1, and Office2 scenes, respectively. For the Gallery and Office1 scenes, 60 percent of the shadow calculation time is used to check object bounding boxes against the active shadow volume, to determine whether a given object can potentially cast a shadow onto the active target polygon. Another 25 percent is used to test polygons from the remaining objects against the shadow volume, and the remaining 15 percent is taken up clipping those polygons that cross the shadow volume. However, for the Office2 scene, most of the shadow calculation time is consumed by the polygon/shadow volume testing procedure (*ChkPLY*), as a result of George's (the skeleton) shadow at the far office corner.

In general, for scenes containing a large number of reflective or refracted surfaces, most of the image generation time will be spent transforming objects to the current virtual coordinate system and testing objects/polygons against the active clipping volume. The percentage of time spent sorting polygons will increase with greater environment complexity, especially when there is a high concentration of polygons within a given region in the scene. Such was the case in the Office1 and Office2 scenes, where most

of the polygon density was concentrated in the bookcase and in George. Finally, for most scenes, shadow generation will dominate the total image generation time, especially for multiple light source situations.

6.4. Conclusions and Suggestions for Further Research

This thesis has presented a novel rendering algorithm that simulates the global illumination effects produced by ray tracing, but at considerable reduction in computational expense. Computational efficiency is achieved by considering only polygonal environments and using conventional hidden surface techniques that can exploit the coherence properties of these environments. The visible surface processor used is based on an extension of the Newell, Newell, and Sancha list-priority algorithm [17] and includes the simulation of planar reflection, and an approximation to planar refraction, using a recursive linear transformation technique. This recursive visible surface method is equivalent to tracing multiple rays in parallel through an environment to determine visible surfaces, reflections, and refractions, and results in considerable efficiency improvement over conventional ray tracing for a wide class of environments. The new rendering algorithm also includes shadow generation, using a shadow projection approach, and simulates non-linear transparency effects. To support this new Image Generation Processor, four different Scan Conversion Processors were implemented, which provide anti-aliased scan conversion of faceted, Gouraud, and Phong type polygons.

The results presented in the previous section clearly demonstrate the overall efficiency improvement of this new rendering method over conventional ray tracing for a wide variety of polygonal environments. The advantages of this new rendering approach include a more efficient visible surface method, including reflections, refractions, and shadows, and a better method of anti-aliasing. Experimental results have shown that, even for complex environments, a great deal of coherence can be exploited to render the image more efficiently than by conventional ray tracing techniques. In addition, the relative simplicity of the visible surface algorithm, as compared to beam tracing [44], permits handling very complex environments without the overhead associated with the Weiler-Atherton hidden surface removal method [26]. Furthermore, real-time performance can be expected for moderately complex environments, provided that the algorithm is supported with a suitable graphics engine, consisting mainly of a matrix multiplier, general polygon clipper, and a polygon tiler.

There are, however, some limitations to the rendering method presented here. First, only polygonal objects are supported. Second, object space coherence is exploited by assuming that reflections and refractions form virtual images within a polygonal window and that these virtual images can be obtained using linear transformation techniques. While this assumption is valid for planar reflections, refractions in general will not be physically correct. Furthermore, this linear mapping approach is also not correct for curved surfaces that are modeled by polygonal meshes since, in reality, the surface orientation varies from point to point across each facet. Thus, while the faceted structure can be eliminated using Gouraud or Phong shading tech-

niques, reflections and refractions remain faceted. One way of overcoming these limitations would be to implement a hybrid scheme, combining the linear mapping technique with conventional ray tracing, to handle the non-linear situations. This would result in an efficient rendering method for most images and provide support for handling non-polygonal objects and simulating refractions in general using the ray tracing approach.

The remaining sections of this chapter discuss several possible enhancements and extensions to the rendering techniques presented in this thesis.

6.4.1. Hierarchical Visible Surface Processor

The current implementation of the recursive visible surface processor uses a two stage hierarchical approach to find all potentially visible polygons within each active clipping volume (Section 4.4). For each reflection or refraction mapping operation, the visible surface processor begins by checking each transformed object bounding box against the active clipping volume and builds a depth sorted list of all objects that are not completely culled out. Following this operation, all remaining objects are expanded into a linked list of polygons, which are possibly clipped to the active clipping volume, and depth sorted. The percentage of the total image generation time taken up by these three major steps are indicated in Table 6.2 for each test scene, and labeled as *Addobj*, *Expobj*, and *Sortply*, respectively.

For complex scenes composed of groups of many small objects, a great deal of time is consumed transforming individual object bounding boxes to the current virtual

coordinate system and then checking them against the active clipping volume. One way of improving the efficiency of the visible surface processor is to hierarchically decompose the entire scene into a tree of enclosing volumes (bounding boxes) [9], whereby groups of adjacent objects are bounded together by a single volume. This grouping process could be specified by the user in the environment description and used by the scene input processor to build the hierarchical data base. Visible surface calculations would then start at the outermost bounding volume, and proceed to subvolumes only if the outer volume is within the active clipping region. Furthermore, once an outer volume is found to lie completely within the clipping region, there is no need to check subvolumes, thus eliminating unnecessary computation.

Sorting complexity also can be reduced considerably with a hierarchical scene description. In the current implementation, all potentially visible objects at a given virtual image tree node are expanded into a composite polygon list prior to the sorting process. As evident from the run-time profiles for the Office1 and Office2 scenes (Table 6.2), the total time spent sorting can dominate the overall rendering time for complex environments. Using a procedure similar to that used in the Newell, Newell, and Sancha sorting algorithm, bounding volumes could be used to test which objects, or group of objects, need to be considered together in order to resolve the visibility problem for a given image region. For example, in the Office2 scene, there would be no need to check polygons in the bookcase against those composing George, and similarly for objects in different regions of the bookcase.

6.4.2. Improved Shadow Generation

Clearly, shadow generation dominates the overall rendering time for most scenes of moderate to high complexity. One of the major bottlenecks involves testing object bounding boxes against the shadow volume, defined by each light source and the active target polygon, to determine if any part of the object can cast a shadow onto the target polygon. For this case, the hierarchical scene decomposition technique discussed in the previous section can potentially reduce the object/shadow volume checking time by quickly eliminating groups of objects outside the shadow volume.

Another computationally expensive operation involves clipping all object polygons to the shadow volume, for those cases when the object bounding box crosses the shadow volume. Rather than checking each polygon individually, as currently done, a potentially better approach might be to compute a silhouette polygon of the entire object from the light's point of view. This single polygon, having fewer edges to test, would then be clipped to the active shadow volume and projected onto the target polygon. It should be pointed out, however, that computing a silhouette polygon may be more time consuming than the current fragmented approach.

For aesthetic purposes, it may be desirable to enhance the shading of shadow polygons and to correctly render overlapped shadow regions caused by multiple light sources. For reasons of implementation simplicity, all shadow polygons are treated just like regular object polygons, except shaded with a constant value. In fact, all shadow polygons for a given target surface receive the same shading value. In addition, since the scan conversion processors tile polygons independently, overlapped shadow

polygons from different light sources do not produce a darker shade, as expected. To simulate these effects correctly requires a more elaborate tiling processor, coupled with more accurate shading calculation.

6.4.3. Improved Modeling of Refraction

As previously stated in Chapter 4, the treatment of planar refraction is only valid for orthographic projections and when the incident rays are nearly perpendicular to the refractive surface. For some specific computer graphics applications this simplified approach to planar refraction may be inappropriate, since in general, the results will not be physically correct. Halstead [54] has suggested the possibility of improving the simulation of planar refraction by calculating a linear transformation matrix *exactly* for the ray impinging on the center of the surface, taking into account its incident angle and the relative index of refraction at the boundary. Then, assuming that the angle subtended by the surface is small, this matrix will produce a *better* approximation to the virtual refracted image formed on the surface. In effect, this refraction approach linearizes about a given incident angle to approximate Snell's law, rather than always assuming that the incident angle is small. In fact, the paraxial approximation used in this thesis is simply a special case of this general linearization approach.

Appendix A

Implementation Overview

A.1. Image Generation Processor

Figure A.1 illustrates the implementation structure of the Image Generation Processor (*plytrace*) discussed in Chapter 4, where each block corresponds to a C-module consisting of a set of functions implementing a specific section of the algorithm. The division of tasks was chosen to provide possible future extension of critical parts of the algorithm into hardware without major changes to the code. A brief discussion of each C-module block is given below:

pmain implements the command interpreter for describing three dimensional scenes and providing user control of the entire image generation process. This module is responsible for setting up all global parameters, initializing all process modules, and managing the active scene object list. A summary of the commands supported by the image generation process is provided in Appendix-B. These commands either can be specified interactively by the user, from a Scene Descriptor File invoked by the "get file-name" command, or both.

prepobj includes a number of useful functions for defining and configuring objects in a scene. For every object defined, a structure is allocated to hold its geometric description (bounding box, vertex coordinates, and polygon list), color description (object

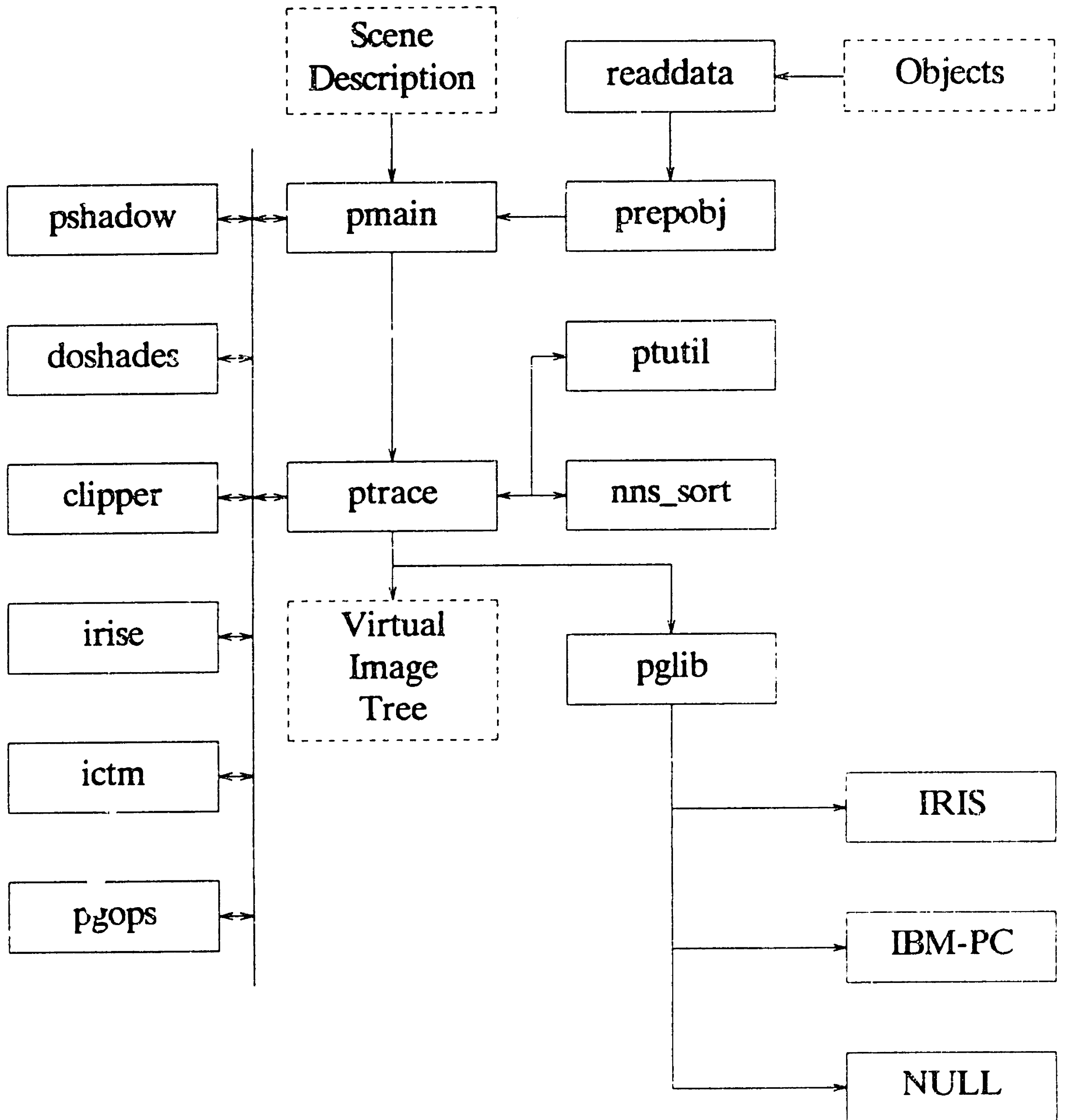


Figure A.1 Image Generation Processor Implementation

color, polygon color, or vertex colors), and various other object attributes, such as surface properties and object type. Object attributes may be defined global or local, and objects may be defined hierarchically.

readdata reads object geometric and color information from specified files, or from pre-defined object descriptions, and builds an internal data structure describing the object. Polygonal objects are described geometrically by a linear list of all x-y-z vertex coordinate pairs, followed by a list polygons, each defined by a list of indexes into this vertex list. Color and surface properties may be specified global for the entire object, or individually for each polygon or vertex.

ptrace implements the recursive visible surface algorithm described in section 4.3. Given a list of all objects in the scene, it finds all potentially visible surfaces in the specified viewing volume, along with surface shadow polygons, and recursively traces planar reflections and refractions. The final image description is provided as a *virtual image tree* , which can be immediately displayed on any one of the supported graphics displays and optionally output to a file for further processing.

ptutil provides useful utility functions for creating and maintaining a sorted list of all potentially visible objects/polygons at each level of the virtual image tree.

nns_sort implements the Newell, Newell, and Sancha sorting and face splitting algorithm, which is optionally called by the visible surface processor to handle complicated environments. This module is passed a depth sorted polygon list, in front-to-back or

back-to-front order, and returns the list with the top-most polygon (or fragment) ready for display.

pshadow implements the shadow generation process discussed in section 4.6. Given the active screen space polygon, it computes all surface shadow polygons by projecting objects between each light source and the target polygon. All shadow polygons are clipped to the boundary of the target polygon in screen space.

doshades computes polygon or vertex shades for the active polygon, and optionally determines highlight vectors for all scene light sources. For vertex shading, it assumes that each vertex contains a color and a normal vector.

clipper implements the general three-dimensional clipping unit described in section 4.4. Given the active screen space polygon, it establishes a new 3-D clipping volume to be used in reflection/refraction tracing, and saves the current active clipping volume in a multi-level clipper stack. The actual polygon clipping algorithm used is an extension of the reentrant clipping technique presented by Sutherland and Hodgman.

irise emulates a subset of the Silicon Graphics IRIS transformation matrix operations, and implements a multi-level Current Transformation Matrix (CTM) stack. In principle, it should be possible to re-compile the image generation process C-code on an IRIS Workstation without this emulation module and make use of the Geometric Engine inherent in the IRIS architecture.

ictm implements the Inverse Transformation Matrix, and associated matrix stack,

needed to transform screen space polygons back to world space for shading calculations.

pgops consists of various useful geometric functions needed throughout the image generation process.

pglib provides the device-dependent interface to various graphics displays that support polygon scan conversion. A *NULL* device is also supported to allow operating in stand-alone mode, where only the virtual image tree is created and output to a file for subsequent hidden surface removal and anti-aliased polygon scan conversion.

A.2. Scan Conversion Processors

Figure A.2 shows the implementation structure for all four scan conversion algorithms described in Chapter 5, where each block corresponds to a C-module consisting of a set of functions implementing a specific section of the algorithm. Any block whose name ends with an asterisk represents one of the four possible scan conversion methods presented in Chapter 5. A brief discussion of each block is given below:

rpmain accepts polygonal virtual image tree scene descriptions, and prepares each polygon for processing by either the Gouraud tiler or Phong tiler, depending on its type. This C-module is common to all scan conversion algorithms and handles all initialization and user interface.

*g_tiler** implements the Gouraud polygon tiler, including its associated scan line segment processor, segment shader-interpolator, and pixel integrator. It accepts one

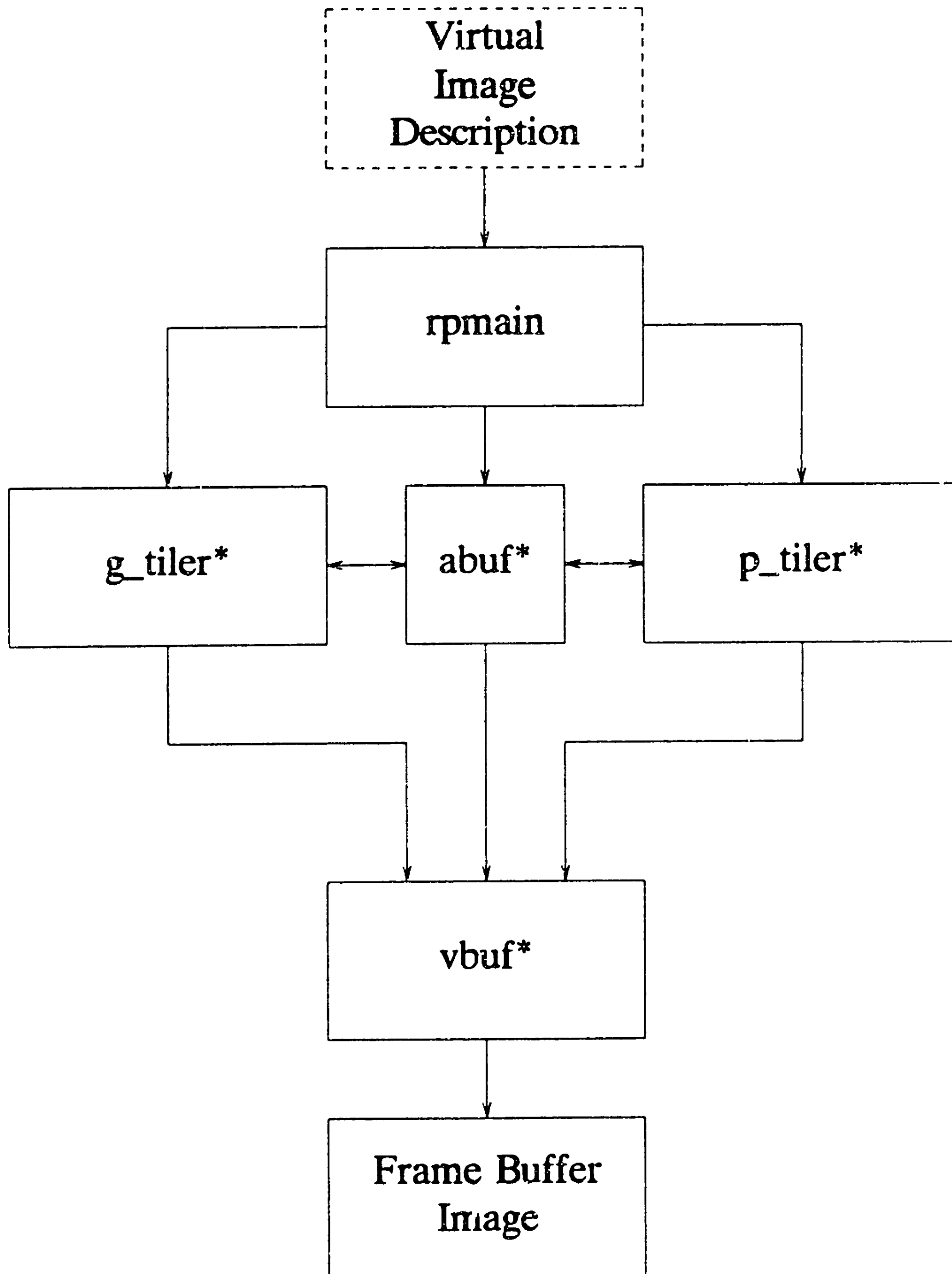


Figure A.2 Scan Conversion Processor Implementation

convex polygon description at a time with colors specified at each vertex and scan converts it into the frame buffer.

*p_tiler** implements the Phong polygon tiler, including its associated scan line segment processor, segment shader-interpolator, and pixel integrator. It accepts one convex polygon description at a time with colors and up to sixteen highlight vectors specified at each vertex and scan converts it into the frame buffer.

*abuf** implements the pixel mask constructor and area coverage bit counting routines for methods 2, 3, and 4. In addition, it supports the pixel-fragment pool for the multi-level pixel mask method (4), which provides dynamic allocation and deallocation of pixel-fragment structures.

*vbuf** implements the virtual frame buffer. For scan conversion methods 1, 2, and 3, the final image is specified as separate files for each R-G-B color component, pixel coverage, and pixel mask. However, for method 4, the final image is specified by a composite file containing pixel-structures, which must be passed through a filter program to extract its separate components (see user manual page *getrgb* in Appendix-B).

NAME
plytrace - image rendering processor for 3-D environments

SYNOPSIS
plytrace [-flags]

DESCRIPTION
Plytrace produces color shaded image descriptions of three-dimensional polygonal environments, including the simulation of shadows, linear and non-linear transparency, reflection, and refraction. Both faceted and Gouraud shading of polygons are supported, in addition to the simulation of Phong specular highlights. Output from the image generation processor is in the form of an ordered sequence of two-dimensional polygons, which can be immediately displayed on any graphics workstation having a polygon tiling function, and optionally saved in a file for further processing. The current implementation supports the Silicon Graphics IRIS Terminal and the IBM-PC Graphics Display System.

Scene descriptions and graphics commands either can be specified by the user interactively, from a scene description file, or both. Typically, a scene description file is created specifying the placement and characteristics of objects, light sources, and viewing position in an environment, and then invoked by the user using the *get* <*des-file*> command. Polygonal object definitions include surface color and surface properties, which can be specified global for the entire object, or individually for each polygon or vertex composing the object. Geometric and optional polygon/vertex color information for each object are specified through separate files (*.det, *.pcl, *.vcl), which are described in "man mkbin".

Output from the image generation processor consists of an ordered sequence of two-dimensional convex polygons that describe the resulting projection of the given three-dimensional scene description onto the specified viewing plane. This output image description takes the form of a tree, referred to as a *virtual image tree* , where polygons at the top level (Level-1) correspond to potentially visible surfaces in the specified field of view, and sub-level polygons correspond to surfaces visible through multiple planar reflections and refractions. In addition, each polygon in the virtual image tree may have a number of surface shadow polygons that have been projected onto its surface by shadow casting.

Output from the image generation processor is available in several different formats, as selected by the user. First, polygons may be generated in either front-to-back or back-to-front depth order, as required to perform the final

process of hidden surface elimination and polygon scan conversion. Second, polygons may be specified with a single shade (faceted shading) or with shades at each vertex (Gouraud shading), in addition to specular highlight vectors (up to 16) at each polygon vertex (Phong specular shading). When polygons are to be displayed immediately on a graphics workstation having only a polygon tiling function, the faceted shading option and a back-to-front rendering order is selected, which performs hidden surface elimination by its overwritting principle. If the output polygon list is to be further processed to produce high-quality, anti-aliased, shaded images (see, for example, "man rftb"), vertex shades and optional highlight vectors are included with each polygon and the list is generated in front-to-back order. In addition, the user can specify that reflected/refracted polygon intensity components be blended with their corresponding parent polygon intensity components during the image generation phase (pre-blending), or during the scan conversion phase (post-blending).

Plytrace is usually operated in interactive mode by simply typing "plytrace <cr>" and then issuing appropriate commands to its command line interface. A simple example of a scene description file is given below, along with a sample IBM-PC *plytrace* session to render the image. The command "get ccube.des" causes *plytrace* to read and interpret commands in the scene descriptor file, ccube.des, as if they had been typed in by the user directly. *Plytrace* responds with the total number of objects, vertex points, and polygons composing the scene, along with timing information. The next command typed by the user, "run y", causes *plytrace* to render and display the image on the graphics display, according to the viewing and projection transformations specified in the scene descriptor file. Argument "y" in the "run y" command indicates that shadow polygons are to be computed and displayed. At the completion of the rendering process, *plytrace* prints out statistics indicating the total number of polygons tiled, maximum recursion level, and the elapsed time. The final command, "quit", terminates *plytrace*.

Plytrace also can be operated in immediate mode, without user interaction, by including all appropriate flags on the invocation command line to invoke a scene description file. For the example above, the equivalent immediate-mode invocation of *plytrace* would be

plytrace -i -f ccube.des <cr>

which causes *plytrace* to read, interpret, and render the image described in the scene descriptor file, ccube.des, and then exit. The command line flags supported by *plytrace* are summarized below and are equivalent to the *setmode* command

parameters.

The following commands are recognized by *plytrace*. Note that all input characters are converted to lower case before processing and that commands are recognized by the first sequence of unique characters:

This is a comment
    Rest of line is a comment and ignored.

ambient <red, green, blue>
    Defines the ambient illumination color for the scene. Values should range between 0.0 and 1.0 (default is 1.0). The ambient coefficient, Ka, for a given polygon/vertex is determined as max(0, 1 - (Kd + Kr + Kt)), where Kd, Kr, Kt are the specified diffuse, reflection and transmission coefficients, respectively.

background <red, green, blue>
    Defines the background color for the scene. Values should range between 0.0 and 1.0 (default is 0.0).

call <file-name>
    Reads and interprets commands contained in the specified file, but unlike the *get* command, causes NO initialization operation. This command is typically used to hierarchically compose complex scenes, by defining subsets of the scene in various files and then combining them through multiple *call* commands.

callobj <obj-number>
    This command is used to copy geometric and color data information (polygon or vertex colors) from an existing object definition, instead of scanning their corresponding data files as normally done. If found inside an active object definition block, the specified source object's data is copied to the active destination object. Otherwise, if no active object exists, a new object is created, initialized with the active global parameters, and the specified source object data is copied into the new object's data structure. Thus, when invoked outside an active object definition block, it essentially corresponds to a *defobj*- *callobj*- *endobj* command sequence. *obj-number* should range between 0 and 255.

clear
    Clears the current viewport area to the background

color.

color <red, green, blue>
Specifies the global color or active object color. Values should range between 0.0 and 1.0. The default global object color is (1.0, 1.0, 1.0).

defobj <obj-name, [obj-number]>
This command begins an active object definition. It causes an object data structure to be allocated and initialized with the active global parameters. _Obj-name_ specifies the object's name, which when prefixed by the current detail directory path name (normally "det/" under UNIX and "det\" under PC-DOS) and appended with ".det", gives the full path name for the object's geometric data file. An optional unique object number between 0 and 255 may be specified, otherwise, the next available number is automatically chosen. The current transformation matrix (CTM) is automatically pushed on the CTM stack to prevent any subsequent modeling transformations from affecting the global state of the CTM. All subsequent modeling, color, and surface attribute commands only affect the active object.

delobj <obj-number-list>
Deletes one or more objects. The object number list should be separated by white space or commas and should reference previously defined objects.

depth <int>
Specifies the maximum recursion depth (1 to 14) for multiple reflection or refraction tracing. The default depth is 1.

det_dir <directory-path>
Specifies the detail directory path where an object's geometric data file (*.det), and optional polygon (*.pcl) or vertex (*.vcl) color file, is located. The default path is "det/" under UNIX and "det\" under PC-DOS, and can be changed as needed anywhere within the scene descriptor file.

diffuse <float>
Specifies the global or active object diffuse coefficient in the range 0.0 to 1.0 (default is 0.6).

disp <[(on), off]>
Display virtual image tree on attached device (when on). Default value is on.

endobj
Terminates the active object definition, transforms the

object's bounding-box and vertex points using the CTM, and restores the original CTM from the matrix stack. Unless a _callobj_ command was issued within the current active object definition, the object's geometric and color data are extracted from their corresponding files. The object's geometric data file (obj-name.det), and optional polygon/vertex color data files, must exist in the active detail directory (see the _det dir_ command).

fname <file-name>
Specifies the output file name (file-name.out) for saving the optional virtual image tree description.

get <file-name>
Causes a global initialize operation (see the _init_ command), then reads and interprets commands in the specified scene descriptor file. Note when issuing successive _get_ commands, the user should first clear the active viewport area (see the _clear_ command) before reading in a new scene descriptor file to prevent unwanted "debris" on the display screen.

getobj <obj-name>
This command is equivalent to a _defobj_ - _endobj_ command sequence, and is typically used to define objects whose color and surface properties are specified by the active global attributes.

ident
Sets the CTM to the identity matrix.

index <float>
Specifies the global or active object index of refraction (default value is 1.0).

init
Initializes all global parameters to their default values, deletes all object and light source definitions, and sets up a default viewport and perspective transformation matrix. The default viewport and perspective matrix depend on the display device being used. For the IBM-PC graphics display, viewport = <0, 639, 0, 479> and perspective = <45, 1.33, 0.01, 10000.0>; for the IRIS display, viewport = <0, 1023, 128, 767> and perspective = <45, 1.60, 0.01, 10000.0>; and for the NULL display, viewport = <0, 1023, 0, 1023> and perspective = <45, 1.00, 0.01, 10000.0>.

light <x, y, z, r, g, b, range, [(P),D]>
Defines a light source at the specified position (x,y,z), with color (r,g,b), and maximum range. The

optional attribute, P, indicates that the light source is a "Point" source, and therefore should cast shadows. Light attribute, D, indicates a "Diffuse" light source, which should not cast shadows. The current implementation supports eight light sources, which can be easily upgraded as needed (see MAXLTS in rdefs.h).

lookat(vx, vy, vz, px, py, pz, twist)
: Specifies a viewpoint (vx,vy,vz) and a reference point (px,py,pz) on the line of sight in world coordinates. *Twist* provides right-handed rotation about the z-axis in the eye coordinate system.

nolight
: Deletes all light source definitions.

ortho <left, right, bot, top, near, far>
: Loads a three-dimensional orthographic projection on the CTM. It defines a box-shaped 3D bounding volume in the eye coordinate system with (x,y,z) clipping planes defined by the specified parameters.

ortho2 <left, right, bot, top>
: Loads a two-dimensional orthographic projection on the CTM. It defines a box-shaped 2D bounding rectangle in the eye coordinate system with (x,y) clipping planes defined by the specified parameters. For 3-D coordinates, the z values are unaffected by this transformation.

outmode <mode>
: Specifies the output mode of display when using one of the supported graphics displays. The supported modes are given below, with mode *c* being the default.

pcolors <file-name>
: Specifies a file containing color data information describing the color, diffuse, reflection, and transmission coefficients of each polygon composing the active object. The file must exist in the current detail directory as file-name.pcl (see the *det dir* command).

perspective <fovy, aspect-ratio, near, far>
: Loads a perspective projection matrix on the CTM defining a truncated viewing pyramid in the eye coordinate system. The field-of-view angle in the y direction is given by *fovy* (in degrees), *aspect-ratio* determines the field-of-view in the x direction, and the locations of the near and far z-clipping planes are defined by *near* and *far*. The aspect ratio is defined as the ratio of x to y, and in general, should match with the aspect

ratio of the active viewport. Arguments *near* and *far* indicate distances from the eye position at (0,0,0) to the near and far clipping planes along the negative z-axis, and are always positive. A default perspective projection matrix is defined at the start of *plytrace* and after an *init* command.

polarview <z-dist, azim, inc, twist>
: Specifies a position and direction of view in polar coordinates. The origin of the eye coordinate system is placed at the point of view, with the negative z-axis aligned with the line of sight. *Z-dist* specifies the distance from the viewpoint to the world space origin, and essentially transforms all objects and light sources away from the viewpoint along the negative z-axis. *Azim* is the azimuthal angle in the x-y plane, and is equivalent to a right-handed rotation of -azim degrees about the positive z-axis. *Inc* is the incident angle in x-z plane, and is equivalent to a right-handed rotation of -inc degrees about the positive x-axis. *Twist* rotates the viewport around the negative z-axis using the right-hand rule. All angles are specified in degrees.

popmatrix
: Pops the CTM from the matrix stack.

pushmatrix
: Pushes the CTM on matrix stack, duplicating the current matrix.

quit
: Terminates the *plytrace* program.

reflectance <float>
: Specifies the global or active object reflection coefficient in the range 0.0 to 1.0 (default value is 0.0). This affects the intensity of all surfaces mapped by a reflection mapping operation for this object.

resolution <hres, vres>
: Specifies a viewport region centered on the screen to display the image, and loads a perspective transformation matrix with *fovy* = 45, *aspect-ratio* = hres/vres, and *near* and *far* z-clipping planes at -0.01 and -10000.0, respectively. The equivalent *viewport* command is given by *left* = (MAX_HRES - hres) / 2, *bot* = (MAX_VRES - vres) / 2, *right* = left + hres - 1, and *top* = bot + vres - 1, where MAX_HRES x MAX_VRES corresponds to the maximum viewport resolution of the given display device (for IBM-PC = 640 x 480, for IRIS = 1024 x 640, and for NULL = 1024 x 1024).

rotate <angle, axis>
Specifies a rotation matrix, Mrot, which pre-multiplies the CTM. The angle of rotation is given in degrees according to the right-hand rule, with the line of sight coincident with the negative z-axis. The axis of rotation is specified by its corresponding letter (x, y, or z).

run <[y]>
Renders the active scene, with the optional parameter, _y_, indicating that shadow polygons are to be computed and displayed (normally not computed). If the scene contains any visible reflective or refractive surfaces, _plytrace_ will recursively trace reflections and refractions through multiple levels, not exceeding the maximum recursion depth specified (see the _depth_ command above).

scale <sx, sy, sz>
Specifies a scale matrix, Msca, which pre-multiplies the CTM.

setmode <mode> [<on, off>]
Allows specification of various run-time parameters described below.

shininess <float>
Controls the spread of Phong specular highlights on an object. Typical values are in the range 100 to 500. Values smaller than 100 tend to spread out the specular highlight over a large area, whereas values large than 200 make small highlights.

sort <[(btf), ftb]>
Specifies the polygon depth sorting order. The default mode is to output polygons at each branch of the virtual image tree polygons in back-to-front (btf) order, starting with the deepest tree branch. For anti-aliased scan conversion, a virtual image tree description is generated with polygons sorted in front-to-back (ftb) order, also starting with the deepest tree branch.

specular <float>
Specifies the global or active object specular reflection coefficient in the range 0.0 to 1.0 (default value is 0.0).

threshold <float>
Sets the minimum intensity coefficient considered meaningful in tracing a planar reflection or reflection. Typical values are in the range 0.0 to 1.0 (default

value is 0.0).

translate <tx, ty, tz>
Specifies a translation matrix, Mtrn, which pre-multiplies the CTM.

transmittance <float>
Specifies the global or active object transmission coefficient in the range 0.0 to 1.0 (default value is 0.0). This affects the intensity of all surfaces mapped by a refraction mapping operation for this object.

transparency <transp, roll-off>
Specifies the global or active object transparency coefficient. The first number (between 0.0 and 1.0) indicates the fraction of incident light passing through the surface, where 0.0 is opaque and 1.0 is totally transparent. The second value governs how the transparency changes as a function of the amount of material the light must pass through to reach the eye. This makes the edges of a curved transparent object look less transparent, with larger values of _roll-off_ indicating thicker material. The default values for both parameters is 0.0.

type <polygon, [open, curved, concave, hidden, ltsrc]>
Specifies the object surface type and its attributes. Currently, only polygonal object types are supported. _Open_ is used to suppress the elimination of back-facing polygons when an object is rendered. _Curved_ indicates that a curved object is modeled as a polygonal mesh and that vertex normals are to be computed so as to perform smooth shading. _Concave_ indicates that the object is non-convex and may contain holes. Normally, polygons composing a convex object are ignored from shadow casting computations on a target polygon of the same object. For concave objects, however, inter-object shadowing is possible and must be considered. _Hidden_ suppresses display of the object. This attribute is typically used to define an object that is called multiple times by the _callobj_ command to build a scene having repeated instances of the same geometric object. _Ltsrc_ indicates that the object represents a light source. When displayed, its shade is taken to be the specified object color. The default value for all attributes is FALSE.

vcolors <file-name>
Specifies a file containing color data information describing the color, diffuse, reflection, and transmission coefficients of each vertex composing the

active object. The file must exist in the current detail directory as file-name.vcl (see the *det-dir* command).

viewport <left, right, bottom, top>
Specifies the region on the screen to display the image, and defines the mapping from world coordinates to screen coordinates. The display screen origin (0,0) is assumed to be located at the lower left corner, with all values ranging from 0 to the maximum allowed values for the given display device. A default viewport is defined at the start of *plytrace* and after an *init* command.

wait
Waits for <return> before continuing (great for demos).

window <left, right, bot, top, near, far>
Loads a projection transformation matrix on the CTM specifying the position and size of a rectangular viewing frustum in the eye coordinate system. The image will be projected with perspective onto the screen located at the *near* clipping plane position.

OUTMODE COMMANDS
The following are valid parameters for the *outmode* command:

a <bits/color>
Selects an adaptive video look-up table construction mode for which R-G-B color entries in the table are dynamically determined during the display process. Since the IBM-PC graphics display supports only 8-bits per picture element, and our current IRIS display terminal has only 12 bit-planes, the given polygon R-G-B color values are first quantized to the specified number of *bits/color* (between 1 and 8) before a table search, and possible entry definition is performed.

r,g,b
Specifies that only the Red, Green, or Blue color component is to be displayed at 8-bits of resolution.

y
Specifies that only the luminance component is to be displayed at 8-bits of resolution.

c
Specifies that polygon colors are to be displayed with R-G-B components encoded as [rrr-ggg-bb] for the IBM-PC graphics display and [rrrr-gggg-bbbb] for the IRIS display terminal. In this mode, the video look-up tables are pre-computed with R-G-B values quantized to

3-3-2 bits per component on the IBM-PC, and 4-4-4 bits per component on IRIS.

SETMODE COMMANDS

The following are valid parameters for the *setmode* command, and as command line flags to *plytrace*. For all parameters having an on/off condition, the default value is off:

digit
: Specifies a debugging level between 0 and 9 (default is 0).

b <on, off>
: If on, indicates that higher tree level parent polygon intensities are to be blended with each lower level children polygon for output to the virtual image file (pre-blending). If off (default), parent/child polygon intensity blending is performed by the subsequent scan conversion process (post-blending).

d <float>
: Sets the minimum screen detail size for ignoring tiny polygons (default value is 0.0002).

e <float>
: Sets value of epsilon used in clipping operations (default value is 0.0000001).

f <file-name>
: Specifies the scene descriptor file name when invoking *plytrace* in immediate mode.

h <on, off>
: Compute shadow polygon shades explicitly (on), or use an approximate intensity reduction method (off).

i
: Selects immediate mode of operation, whereby *plytrace* is invoked to render a single scene and then terminated. This mode must be specified as a command line flag to *plytrace*. The default mode of operation is interactive.

l <1-14>
: Specifies the maximum recursion level for multiple reflection or refraction tracing operations (default value is 1).

n <on, off>
: Selects the method used to sort polygons. When off, polygons are sorted based on their min/max vertex z-coordinates only. When on, Newell, Newell, & Sancha's

sorting and face splitting technique is used.

o <on, off>
If on, *plytrace* outputs a *virtual image tree* description to the specified output file.

p <on, off>
If on, *plytrace displays only object bounding boxes* field-of-view and performs no reflection/refraction tracing. Note, that while *b* is set ON during the object definition phase, only the object's bounding box is read into its data structure.

s <on, off>
On indicates that Phong specular highlights vectors are to be computed at each polygon vertex for subsequent normal interpolation within the polygon to depict well-defined specular highlights. Otherwise, specular highlights are only computed for each polygon/vertex and added to the ambient and diffuse intensity components.

v <on, off>
On indicates that polygon vertex shades, and optional highlight vectors are to be computed. Off indicates that a single average polygon shade is to be computed.

x <on, off>
If on, plytrace expands all potentially visible objects at each tree branch before sorting the resulting composite polygon list. Otherwise, only a single object is made active, expanded into a sorted polygon list, and rendered, before considering the next object.

EXAMPLES

Sample scene descriptor file:

```
fname           ccube
resolution      512 480
pushmatrix      /* save perspective transformation */
ident
polarview       7 0 75 0
translate       0 0 -1
light           -4.0 -2.0 6.0 0.8 0.8 0.8 100 Point
light           0.0 -10.0 10.0 0.5 0.5 0.5 50 Diffuse
#
defobj         cube
    color 1 1 1
    scale 5 5 0.1
    translate  0 0 -1
endobj
defobj         cutcube
    color 0 1 0
    translate  0 0 1.05
    rotate     30 z
endobj
popmatrix
```

Sample interactive session:

```
plytrace <cr>
plytrace - version 2.0 13-Jan-86

Command: get ccube.des
Scanning descriptor file ccube.des
2 objects, 18 points, 13 polygons
Time (HR:MIN:SEC) = 00:00:02
Command: run y
Building image.
5 Polygons tiled, Max depth = 1
Time (HR:MIN:SEC) = 00:00:01
Command: quit
```

FILES

fname.out, plytrace.log

SOURCE

/unips/src/graphics/ptrace/

SEE ALSO

rftb(1), mkbin(1)

AUTHOR

A. Garcia

NAME

rftb1 - anti-aliasing polygon tiler (pixel-coverage)
rftb2 - anti-aliasing polygon tiler (pixel-mask)
rftb3 - anti-aliasing polygon tiler (pixel-coverage/mask)
rftb4 - anti-aliasing polygon tiler (pixel-structs)

SYNOPSIS

rftb <input> [output] [hres] [vres] [act-lines] [-flags]

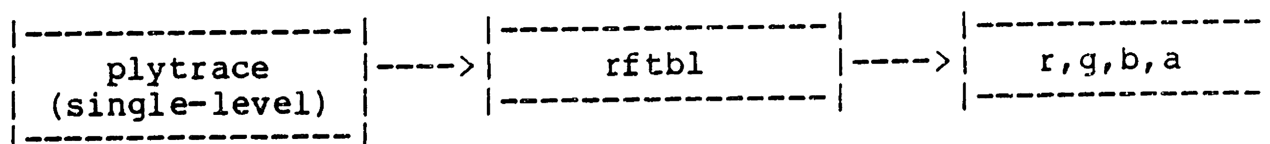
DESCRIPTION

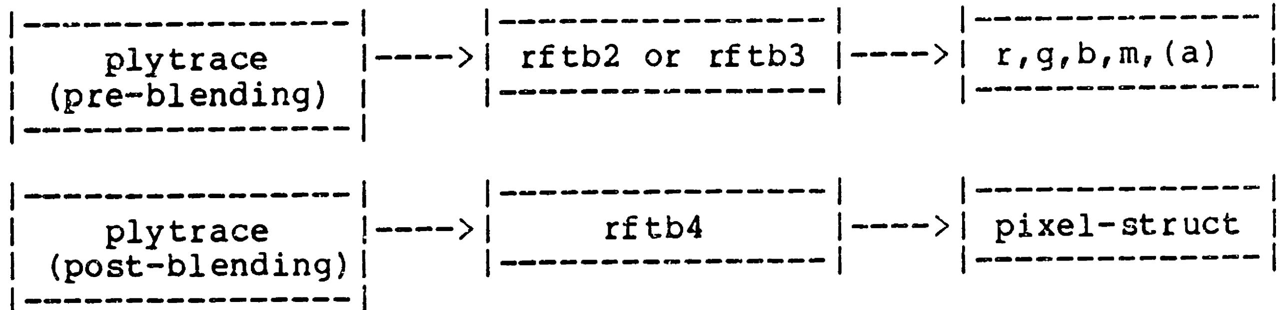
*Rftb** implements an anti-aliasing scan conversion processor for convex polygons. It accepts a sequence of polygons in front-to-back order, performs hidden surface elimination, and scan converts faceted, Gouraud, or Phong type polygons into a virtual frame buffer. Faceted polygons are specified with a single R-G-B shading value, while Gouraud and Phong type polygons are specified with an R-G-B shading value at each vertex. In addition, Phong type polygons may contain up to sixteen highlight vectors with each vertex definition, which allows better depiction of specular highlights on curved surfaces modeled by polygonal meshes. A two-step linear interpolation method of the shading values, and optional highlight vectors is performed to compute the total shade of any point within a polygon.

The input data source, <*input*>.*out*, is a binary file containing an ordered sequence of two-dimensional polygon descriptions, with format identical to the *virtual image tree* file created by *plytrace* (see "man plytrace"). [*Output*] specifies the optional output file name seed for generating appropriate virtual frame buffer component files (default seed is the input file name). The other optional parameters include the output image resolution, [*hres* x *vres*] (default is 256 x 256), the number of active frame buffer lines (*act-lines*) to be kept in physical memory during the rendering process, and various option flags, which are described below. The number of frame buffer lines kept active can range anywhere from a single line to the entire frame buffer (default is 4 lines).

Each scan conversion processor maintains a LOG file (rftb[1234].log) in the working directory, which contains a history of run-time statistics.

The normal mode of operation between *plytrace* and the four scan conversion processors (*rftb1*, *rftb2*, *rftb3*, *rftb4*) is as follows:





Rftb1 performs hidden surface elimination, anti-aliasing, and simulates linear and non-linear transparency using a *pixel-coverage* byte (8-bits) to represent each display screen sample. This polygon tiler is well suited for scenes composed of diffuse, specular, and transparent surfaces, but should not be used to render scenes containing shadows, reflections, or refractions. It produces four separate files (output.r, output.g, output.b, and output.a), containing the Red, Green, and Blue color separations of the final image, as well as the resulting pixel-coverage.

Rftb2 performs anti-aliased hidden surface elimination using a 4 x 4 *pixel-mask* (16-bits) to represent subpixel regions at each display screen sample. This polygon tiler can render virtual image trees describing scenes containing diffuse, reflected, and refracted surfaces, as well as shadow detail polygons. It is assumed that the multi-level virtual image tree was created by *plytrace* using the pre-blending mode of output, which means that the total shade of any sub-level polygon in the tree includes its higher-level parent polygon shade. *Rftb2* produces four separate files (output.r, output.g, output.b, and output.m), containing the Red, Green, and Blue color separations of the final image, as well as the resulting pixel-mask.

Rftb3 performs anti-aliased hidden surface elimination using both a *pixel-coverage* byte and a 4 x 4 *pixel-mask* to represent each display screen sample. It essentially combines the features of *rftb1* and *rftb2* to render virtual image trees describing scenes containing diffuse, transparent, reflected, and refracted surfaces, as well as shadow detail polygons. *Rftb3* produces five separate files (output.r, output.g, output.b, output.a, and output.m), containing the Red, Green, and Blue color separations of the final image, as well as the resulting pixel-coverage and pixel-mask.

Rftb4 performs anti-aliased hidden surface elimination and child/parent polygon intensity blending using a multi-level *pixel-coverage* byte and 4 x 4 *pixel-mask* at each image sample. This tiler can render complex 3-D environments, described by virtual image trees, containing multiple levels

of reflection and refraction, as well as shadow detail polygons. Since *Rftb4* performs child/parent intensity blending, it is assumed that the virtual image tree given was created by *plytrace* using the post-blending mode of operation. This means that reflected or refracted components of a given surface are blended with its corresponding parent surface during the scan conversion phase. *Rftb4* produces a single composite file containing color and coverage information at each display pixel location, which must be processed by *getrgb* (see "man getrgb") to extract the R-G-B color separations.

FLAGS

The following command line flags are supported:

-digit
Specifies a debugging level between 0 and 9.

-B
Indicates that a background polygon, with color specified in the input data file, is to be painted after the entire image has been rendered. Normally, the background is not painted.

The following flags only work with *rftb4* when running on an IBM-PC with the graphics display attachement. For the output display mode flags below (a,r,g,b,y,c), the default mode is *c*:

-Y
Indicates that the image is to be displayed on the graphics screen during the rendering process. Normally, polygons are not displayed.

-a <bits/color>
Selects an adaptive video look-up table construction mode for which R-G-B color entries in the table are dynamically determined during the display process. Since the IBM-PC graphics display supports only 8-bits per picture element, the given pixel R-G-B color values are first quantized to the specified number of *bits/color* (between 1 and 8) before a table search, and possible entry definition is performed.

-[r,g,b]
Specifies that only the Red, Green, or Blue color component is to be displayed at 8-bits of resolution.

-y
Specifies that only the luminance component is to be displayed at 8-bits of resolution.

-c

Specifies that pixel colors are to be displayed with R-G-B components encoded as [rrr-ggg-bb]. In this mode, the video look-up tables are pre-computed with R-G-B values quantized to 3-3-2 bits per component.

EXAMPLES

To render a virtual image tree file, test.out, using *rftb2* at 512 x 512 display resolution and with the background color painted, the following command is issued:

```
rftb2 test test2 512 512 -B
```

where the final virtual frame buffer image is contained in files test2.r, test2.g, test2.b, and test2.a on the current directory. Rendering statistics will also be concatenated to the tail of rftb2.log.

FILES

Rftb1 - output.[rgba], rftb1.log
Rftb2 - output.[rgbm], rftb2.log
Rftb3 - output.[rgbam], rftb3.log
Rftb4 - output, rftb4.log

SOURCE

/unips/src/graphics/aatiler/

SEE ALSO

plytrace(1), getrgb(1)

AUTHOR

A. Garcia

NAME

getrgb - extract r,g,b components from a pixel-struct file

SYNOPSIS

getrgb [fname]

DESCRIPTION

getrgb reads a pixel-struct file created by *rftb4* and outputs the Red, Green, and Blue color components to separate files. The output file name seed is taken as the specified input file name.

A pixel-struct file is assumed to contain a linear list of 64-bit data structures with the following format:

```
typedef struct
{
    fragment    *flist;         /* fragment list ptr */
    byte        r, g, b;        /* pixel color */
    byte        pixstat;        /* pixel status */
} pixelstruct;
```

where each *byte* field is 8-bits wide, and the fragment list pointer (flist) is 32-bits wide.

FILES

fname.r, fname.g, fname.b

SOURCE

/unips/src/graphics/util/

SEE ALSO

rftb(1)

AUTHOR

A. Garcia

NAME
mkdet - convert ASCII object geometric files to binary form
mkpcl - convert ASCII polygon color files to binary form
mkvcl - convert ASCII vertex color files to binary form

SYNOPSIS
mkdet <input_file> <output_file>
mkpcl <input_file> <output_file>
mkvcl <input_file> <output_file>

DESCRIPTION
Mkdet scans an ASCII object geometric description file containing vertex and polygon specifications and converts it to a binary detail shape file. The input file is scanned for the keyword "data", followed by a line specifying the number of vertices and polygons in the object. These two lines should be followed by the (x,y,z) coordinates of each vertex, and then a list of polygons making up the object. Each polygon entry should have the format:

<# of vertices in polygon> <vertex-index list>

where the vertex-index list specifies indices (1 to num_vtx) into the array of vertices given, listed in clock-wise order when viewed from outside the object (right-handed coordinate system assumed).

The format for the output geometric binary file is as follows:

```
char    "DET";                     /* File identification tag */
short   num_vtx, num_polys;        /* #-vertices, #-polygons */
float   xmin, ymin, zmin;          /* Bounding box min-coord */
float   xmax, ymax, zmax;          /* Bounding box max-coord */
float   vertex[num_vtx][3];        /* List of vertices */
short   nvtx, index[nvtx];         /* For each polygon */
```

Mkpcl, *Mkvcl* takes a color file in a format similar to the ASCII object geometric description and converts it to a binary polygon and vertex color file, respectively. However, a single number is expected after the "data" keyword, indicating the number of color entries to follow, and each line thereafter is expected to have the following format:

<red, green, blue, dif-coeff, ref-coeff, trn-coeff>

which specifies the red, green, and blue color components, along with the diffuse, reflection, and transmission coefficients for each polygon (or vertex). All values range from 0.0 to 1.0.

The format for the output color binary file is as follows:

```
char     "PCL" (or "VCL");       /* File identification tag */
short    num_clrs;               /* Number of colors in file */
byte     r,g,b,dif,ref,trn;      /* For each color (0 to 255) */
```

EXAMPLES

Typical input files for the above:

```
title     geometric data file for an x-y square
data
4   1
 0.  0.  0.
 0.  1.  0.
 1.  1.  0.
 1.  0.  0.
 4  1  2  3  4

title     polygon color file for a square
data 1
 .9 .8 .2 .0 .0 .0

title     vertex color file for a square
data 4
 .9 .8 .2 .0 .0 .0
 .8 .9 .2 .0 .0 .0
 .2 .8 .9 .0 .0 .0
 .9 .2 .8 .0 .0 .0
```

SOURCE

/unips/src/graphics/util/

SEE ALSO

plytrace(1)

Bibliography

1. J.H. Clark, "The Geometry Engine: A VLSI Geometry System for Graphics," *Computer Graphics*, **16**, no. 3, SIGGRAPH-82 (Jul. 1982), pp. 127-133.
2. J.H Clark and T. Davis, "Work station unites real-time graphics with UNIX, Ethernet," *Electronics* (Oct. 1983), pp. 113-119.
3. C. Panasuk, "Focus on Graphics Workstations," *Electronic Design* (Aug. 1985), pp. 157-164.
4. F.C. Crow, "The Aliasing Problem in Computer-Generated Shaded Images," *CACM*, **20**, no. 11 (Nov. 1977), pp. 799-805.
5. F.C. Crow, "A Comparison of Antialiasing Techniques," *IEEE Computer Graphics and Applications*, **1**, no. 11 (Jan. 1981), pp. 40-48.
6. B.B. Mandelbrot, *Fractals: Form, Chance and Dimension* (Freeman, 1977).
7. W.T. Reeves, "Particle Systems-A Technique for Modeling a Class of Fuzzy Objects," *Computer Graphics*, **17**, no. 3, SIGGRAPH-83 (Jul. 1983), pp. 359-376.
8. J.H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *CACM*, **19**, no. 10 (Oct. 1976), pp. 547-554.
9. S.M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Computer Graphics*, **14**, no. 3, SIGGRAPH-80 (Jul. 1980), pp. 110-116.
10. B.T. Phong, "Illumination for Computer Generated Pictures," *CACM*, **18**, no. 6 (Jun. 1975), pp. 311-317.
11. J.F. Blinn, "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics*, **11**, no. 2, SIGGRAPH-77 (1977), pp. 192-198.
12. K.E. Torrance and S.M. Sparrow, "Polarization, Directional Distribution, and Off-Specular Peak Phenomena in Light Reflected from Roughened Surfaces," *J. Opt. Soc. Am.*, **56**, no. 7 (Jul. 1966), pp. 916-925.
13. K.E. Torrance and S.M. Sparrow, "Theory for Off-Specular Reflection from Roughened Surfaces," *J. Opt. Soc. Am.*, **57**, no. 9 (Sep. 1967), pp. 1105-1114.
14. T. Whitted, "An Improved Illumination Model for Shaded Display," *CACM*, **23**, no. 6 (Jun. 1980), pp. 343-349.
15. H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, **C-20**, no. 6 (Jun. 1971), pp. 623-629.
16. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization Of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, **6**, no. 1 (Mar. 1974), pp. 1-55.

17. M.E. Newell, R.G. Newell, and T.L. Sancha, "A Solution to the Hidden Surface Problem," *Proc. ACM National Conference* (1972), pp. 443-450.

18. G. Romney, G. Watkins, and D. Evans, "Real Time Display of Computer Generated Half-Tone Perspective Pictures," *Proc. IFIP Congress* (1968), pp. 973-978.

19. W.J. Bouknight, "A Procedure for Generation of Three-Dimensional Half-Toned Computer Graphics Representations," *CACM*, **13**, no. 9 (Sep. 1970), pp. 527-536.

20. W.J. Bouknight and K.C. Kelley, "An Algorithm for Producing Half-Tone Computer Graphics Presentations with Shadows and Movable Light Sources," *SJCC, AFIPS*, **36** (1970), pp. 1-10.

21. C. Wylie, R.S. Romney, D.C. Evans, and A. Erdahl, "Half-Tone Perspective Drawings by Computer," *Proc. AFIPS FJCC*, **31** (1967), pp. 49-58.

22. J.M. Lane, L.C. Carpenter, T. Whitted, and J.F. Blinn, "Scan Line Methods for Displaying Parametrically Defined Surfaces," *CACM*, **23**, no. 1 (Jan. 1980), pp. 23-34.

23. E. Catmull, "Computer Display of Curved Surfaces," *IEEE Computer Graphics, Pattern Recognition, & Data Structures* (May. 1975), pp. 11-17.

24. L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 103-108.

25. J. Warnock, "A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures," *University of Utah Computer Science Department*, TR **4-15** (Jun. 1969).

26. K. Weiler and P. Atherton, "Hidden Surface Removal Using Polygon Area Sorting," *Computer Graphics*, **11**, SIGGRAPH-77 (1977), pp. 214-222.

27. A. Appel, "The Notion of Quantitative Invisibility and The Machine Rendering of Solids," *Proc. ACM National Conference*, **14** (1967), pp. 387-393.

28. E. Goldstein and R. Nagel, "3D Visual Simulation," *Simulation*, **16**, no. 1 (1971), pp. 25-31.

29. D.S. Kay and D. Greenberg, "Transparency for Computer Synthesized Images," *Computer Graphics*, **13**, no. 2, SIGGRAPH-79 (Aug. 1979), pp. 158-164.

30. M. Potmesil and I. Chakravarty, "A Lens and Aperture Camera Model for Synthetic Image Formation," *Computer Graphics*, **15**, no. 3, SIGGRAPH-81 (Aug. 1981), pp. 297-305.

31. J.T. Kajiya, "Ray Tracing Parametric Patches," *Computer Graphics*, **16**, no. 3, SIGGRAPH-82 (Jul. 1982), pp. 245-254.

32. P. Hanrahan, "Ray Tracing Algebraic Surfaces," *Computer Graphics*, **17**, no. 3, SIGGRAPH-83 (Jul. 1983), pp. 83-90.

33. J.T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Computer Graphics*, **17**, no. 3, SIGGRAPH-83 (Jul. 1983), pp. 91-102.

34. J.T. Kajiya and B.P. Von Herzen, "Ray Tracing Volume Densities," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 165-174.

35. J.T. Kajiya, "SIGGRAPH-83 Tutorial on Ray Tracing," *SIGGRAPH Course 10 Notes* (1983).

36. H. Weghorst, G. Hooper, and D. Greenberg, "Improved Computational Method for Ray Tracing," *ACM Transactions on Graphics*, **3**, no. 1 (Jan. 1984), pp. 52-69

37. C.B. Jones, "A New Approach to the Hidden Line Problem," *Computer Journal*, **14**, no. 3 (Aug. 1971), pp. 232-237.

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

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

40. S.D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, no. 18 (1982), pp. 109-144.

41. R.L. Cook, T. Porter, and L. Carpenter, "Distributed Ray Tracing," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 137-145.

42. M.E. Lee, R.A. Redner, and S.P. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics*, **19**, no. 3, SIGGRAPH-85 (Jul. 1985), pp. 61-68.

43. J. Amanatides, "Ray Tracing With Cones," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 129-135.

44. P.S. Heckbert and P. Hanrahan, "Beam Tracing Polygonal Objects," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 119-127.

45. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics* (McGraw-Hill, 1979).

46. I.E. Sutherland and G.W. Hodgman, "Reentrant Polygon Clipping," *CACM*, **17**, no. 1 (Jan. 1974), pp. 32-42.

47. F.C. Crow and B.T. Phong, "Improved Rendition of Polygonal Models for Curved Surfaces," *Proceedings Second USA-Japan Computer Conference* (Aug. 1975), pp. 475-480.

48. E. Catmull, "A Hidden-Surface Algorithm With Anti-Aliasing," *Computer Graphics*, **12**, no. 3, SIGGRAPH-78 (Aug. 1978), pp. 6-11.

49. E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Computer Graphics*, **17**, no. 3, SIGGRAPH-83 (Jul. 1983), pp. 141-149.

50. E. Feibush, M. Levoy, and R. Cook, "Synthetic Texturing Using Digital Filters," *Computer Graphics*, **14**, no. 3, SIGGRAPH-80 (Jul. 1980), pp. 294-301.

51. E. Catmull, "An Analytic Visible Surface Algorithm for Independent Pixel Processing," *Computer Graphics*, **18**, no. 3, SIGGRAPH-84 (Jul. 1984), pp. 109-115

52. G. Abram, L. Westover, and T. Whitted, "Efficient Alias-free Rendering using Bit-masks and Loop-up Tables," *Computer Graphics*, 19, no. 3, SIGGRAPH-85 (Jul. 1985), pp. 53-59.

53. F.C. Crow, "A More Flexible Image Generation Environment," *Computer Graphics*, 16, no. 3, SIGGRAPH-82 (Jul. 1982), pp. 9-18.

54. R.H. Halstead, *Private Communication* (Dec. 1985).