Rendermatic: An Implementation of the Three Dimensional Computer Graphics Rendering Pipeline

by

Brian M. Croll

B.S., Mathematical Sciences
Stanford University
Stanford, California
1984

SUBMITTED TO THE DEPARTMENT OF ARCHITECTURE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
OF

MASTER OF SCIENCE

AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1986

©Massachusetts Institute of Technology

Signature of the Author

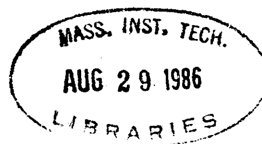
....
Brian M. Croll
Department of Architecture
August 15, 1986

Certified by

....
David Zeltzer
Assistant Professor of Computer Graphics
Thesis Supervisor

Accepted by

....
Nicholas Negroponte
Chairman
Departmental Committee on Graduate Students


MASS. INST. TECH.
AUG 29 1986
LIBRARIES

Rotch

Rendermatic: An Implementation of the Rendering Pipeline

by

Brian M. Croll

Submitted to the Department of Architecture on August 15, 1986 in partial fulfillment of the requirements of the degree of Master of Science.

Abstract

Computer graphics rendering of three dimensional objects is viewed as a three step process: sampling, mapping, and reconstruction. Each component of the rendering pipeline is described in this light. Problems which occur with the implementation of scan conversion and the a-buffer are discussed. A modular program architecture suitable for parallel processing is proposed. Rendermatic, a C library including routines for Gouraud and Phong shading, z-buffer hidden surface elimination, and a-buffer anti-aliasing is introduced.

Thesis Supervisor: David Zeltzer
Title: Assistant Professor of Computer Graphics

This work was supported in part by NHK (Japan Broadcasting Corp.) and Defence Advance Research Projects Agency (DARPA) contract #MOA903-85-K-0351.

Acknowledgements

Special thanks are extended to:

My advisor, David Zeltzer, for giving me the opportunity to study computer graphics.

David Chen for always being cheerfully available to answer questions, to provide stimulation, and to write code that needed writing. Special credit goes to David for being a partner in the ideas and hard work that went into Rendermatic.

Masa Inakage and Jonathon Linowes for introducing me to ray tracing and interactive graphics.

My roommates and honorary roommate who provided comic relief, support, and one hell of a good time.

Julia for the fun times and memories.

All the crew in the Garden and VLW for their encouragement, advice, and friendship.

And, finally, to my family for a quarter century of love and nourishment.

Contents

1 Introduction **7**

2 The Development of Computer Graphics Rendering **10**

3 An Overview of the Rendering Pipeline: Sampling, Mapping, and Reconstructing **20**

4 Sampling **24**
- 4.1 Sampling an Object for Geometric Modeling 24
- 4.2 Sampling The Scene for Rendering 26
- 4.3 A Comparison of Phong and Gouraud Shading 27
- 4.4 Summary 29

5 Mapping the Samples **31**

6 Reconstructing the Surface **33**
- 6.1 Scan Converting the Surface 33
  - 6.1.1 Spatial Coherence 34
  - 6.1.2 Exploiting Coherence 34
  - 6.1.3 The Edge Structure 36
  - 6.1.4 The Edge Table 37
  - 6.1.5 The AET 38
  - 6.1.6 Summary of The Algorithm 38
  - 6.1.7 Problems Encountered in Scan Conversion 39
  - 6.1.8 Summary 40
- 6.2 Interpolating Values 40
- 6.3 The A-buffer: Anti-aliasing 43
  - 6.3.1 The A-Buffer Algorithm 45

6.3.2 Modifying Scan Conversion 48
6.3.3 A Summary of the New Scan Conversion 51
6.3.4 Creating a Pixelstruct 53
6.3.5 Inserting Fragments 55
6.3.6 The Fragment Packer 55
6.4 Summary 60

7 Conclusion **61**
7.1 A Look Back 61
7.1.1 A Quick Review 62
7.1.2 Limitations 63
7.2 A Look Ahead 64
7.2.1 An Overview of the Proposed Pipeline 65
7.2.2 Data Structures 67
7.2.3 Functions 69
7.2.4 Summary 71
7.3 Conclusion 74

A Rendermatic **75**

B The Data Structures **77**
B.1 The Object Structures 81
B.1.1 Object Information 81
B.1.2 A Polyhedron For Each Space 87
B.1.3 Functions that apply to the whole object 89
B.1.4 Transforming the Object 89
B.2 The Polyhedron Structure 91
B.2.1 Allocating and De-allocating Polyhedra 95
B.2.2 Transforming the Polyhedra 96
B.2.3 General Polyhedron Functions 96
B.2.4 Accessing Polyhedra from the Object Level 97
B.3 The Polygon Structure 98
B.3.1 Allocating and De-allocating Polygons 100
B.3.2 Polygon Extent 100
B.4 The Polygon Information Structure 101
B.4.1 Allocating and De-allocating Poly-Info 104
B.5 The Point Structure 105

B.5.1 Allocating and De-allocating POINTS 106
B.6 The Point Information Information Structure 107
B.6.1 Allocating and De-allocating PT_INFO 109
B.7 Optical 110
B.7.1 Allocating and De-Allocating Optical Structures . . 112

C Rendermatic Rendering 114
C..2 Wireframe Rendering 115
C..3 Rendering Shaded Images 116
C.1 The A-Buffer 116
C.1.1 The Data Structures 116
C.1.2 Functions 118

D The Viewing Transformations 122
D.1 Definitions 122
D.2 An Overview 123
D.2.1 World Space to Eye Space 124
D.2.2 Eye Space to Clipping Space 126
D.2.3 Concatenating the matrices 127
D.2.4 Clipping Space to Image Space 128

Chapter 1

Introduction

A painter stares into a blank canvass. Facing the painter is a fundamental problem: how can the painter arrange pigment on the canvass to create a meaningful pattern?

A computer graphics system starts with a blank framebuffer. The question posed to the system is how can it load the framebuffer with an array of numbers which creates a meaningful image?

One approach is to mathematically calculate the numbers. The numbers filling the array have no significance beyond being the result of an abstract function; the image generated is a visual representation of a mathematical entity. Images generated by the Mandelbrot set and by cellular automata are popular examples of this type of graphics.

Another approach to generating the pattern is to hand place the numbers in the array. This can be done with the aid of an interactive painting system. By "painting" with the system, one gets the illusion of placing pigment on a surface while creating the underlying grid of numbers.

The final approach, the topic of this thesis, is to simulate photography.

The user creates a model of three dimensional objects; the computer system takes the picture.

In order to "take" a convincing picture, the system must process an enormous volume of information. Traditionally, a computer graphics system has been viewed primarily as an assortment of programming techniques used to condense and manipulate this information. Accordingly, computer graphics rendering has been presented from this point of view.

This thesis notes that these programming techniques are the result of a deeper problem inherent in computer graphics. No matter how much memory, how much processing power, and how many programming techniques one employs, there will be an overabundance of information to model and process; the visual world is extraordinarily complex, and our eyes are remarkably sensitive. To overcome these obstacles, each computer graphics system must sample the three dimensional model and reconstruct a two dimensional image from those samples. This thesis presents the rendering pipeline as a process of sampling three dimensional objects, mapping the samples to the screen, and finally, constructing the final image from the samples.

Chapter 2 reviews the history of computer graphics rendering. Chapter 3 is an overview of the rendering pipeline from the sampling perspective. Chapter 4 discusses the process of sampling objects. Gouraud and Phong shading are viewed as two separate methods of sampling an object. Phong shading's superior performance is explained from this point of view. Chapter 5 briefly discusses mapping the objects from three dimensions onto the screen. Chapter 6 reviews the techniques used to reconstruct the image

from the samples. Scan conversion and interpolation are discussed. A detailed explanation of the a-buffer[5] is included. The ***mask buffer***, a new data structure, is introduced in order to assist in creating the correct pixel mask at the corners of a polygon. Finally, in Chapter 7, the conclusion, the sampling and reconstruction process is applied to rendering in a parallel processing environment. A modular, flexible, and easy to maintain rendering pipeline suitable for a parallel machine is proposed.

In the appendices, an implementation of the serial rendering pipeline, *Rendermatic*, is presented. Rendermatic is a C library of graphics routines used for wireframe and smooth shaded images. The library includes functions for Gouraud and Phong shading, multiple light sources, and the a-buffer. The major data structures, their accompanying functions, and the rendering functions are described.

In the appendices, the data structures and functions of an implementation of the serial pipeline are presented. The implementation, Rendermatic, is a C library of graphics routines for wireframe and smooth-shaded rendering. The library includes functions for Gouraud and Phong shading, multiple light sources, and anti-aliasing with the a-buffer. The final appendix, included primarily as a reference, is an explanation of the viewing transformation.

Chapter 2

The Development of Computer Graphics Rendering

Computer graphics has been around nearly as long as computers. As far back as the Whirlwind computer in 1953[23] , output from computers has been displayed graphically. However, the potential of computer graphics was not recognized until 1963 when Ivan Sutherland's doctoral thesis, Sketchpad [33], broke the ground and laid the foundation for the discipline of computer graphics. Sketchpad introduced computers as an instrument to create easily manipulated and easily modified two dimensional drawings. Sketchpad explicitly demonstrated the utility of computer graphics, from circuit design to computer animation. While Sketchpad introduced computer graphics, a follow up project, Sketchpad III by Timothy Johnson[20] introduced three dimensional (3-D) computer graphics [1] in 1963. Sketchpad

[1] The name 3-D computer graphics is perhaps a misnomer. Most computer graphic images are two dimensional; they are displayed on two dimensional surfaces. Computer generated holography and computer generated sculpture may rightly lay claim to the term 3-D computer graphics. Perhaps "synthetic photography" is a better name for what is

III applied the homogeneous coordinate system[30] with its accompanying rotation, scaling, translation, and perspective transformations to position three dimensional objects and project them onto a viewing plane. All the objects created by Sketchpad III were polygonal in nature, and only the outlines of the objects were drawn. Sketchpad III did not handle curved surfaces. Furthermore, lines that were obstructed by surfaces closer to the viewer were not removed. By the early sixties, then, the promise of 3-D computer graphics was established; so were the technical problems. The type of objects rendered had to be extended to include objects with curved surfaces, the hidden edges and surfaces had to be removed, and the outlines of the objects had to be filled in.

The most straightforward approach to rendering curved surfaces was to apply the three dimensional techniques explored in Sketchpad III to a curved surface[36][25]. Unfortunately, the time needed to calculate which curves lay behind others was exorbitant. To speed up the hidden surface calculations, another approach was taken. Instead of rendering curved surfaces directly, the surfaces were approximated by polygons. The hidden surface problem for polygons was less formidable than for curved surfaces; Roberts had already found a mathematically correct, though computationally inefficient, solution. Thus, from the start, there were two distinct approaches to rendering curved surfaces. One approach side-stepped the complexity of curved surfaces by approximating them with simple polygons. The other attacked the problem by performing hidden surface calculations

currently called 3-D computer graphics

on the curved surfaces.

As would be expected, approximating curved surfaces with polygons diminished the quality and accuracy of the images produced. The surfaces clearly showed the edges of the polygons of the constituent polygons. Additionally, the silhouettes of the supposedly smooth curved surfaces appeared faceted. Since this method simplified the hidden surface problem substantially, the degradation of the quality of the resultant image was tolerable. This simple polygonal approach stimulated a number of successful algorithms used to draw approximate representations of curved surfaces.[2] At the heart of each of these rendering systems was a new hidden surface algorithm. By 1972 there was a wide enough range of hidden surface algorithms for polygons to warrant the writing of the classic paper, "A Characterization of Ten Hidden Surface Algorithms[34]" classifying and comparing ten distinct hidden surface algorithms.

To a large degree, the hidden surface problem for polygons was satisfactorily solved by the early 1970's. The task of smoothly shading the visible polygons, however, was not. Early on it was recognized that the diffuse shading of a point in a polygon depended on the angle between the surface normal, the vector to the light source, and the distance from the viewer to the surface. In order to fill the interior points of a polygon, a simple illumination model was used to find the color at the vertices of the polygon. The normal at each vertex was assumed to be the polygon normal. Since it is possible to have abrupt changes in the normal between adjacent poly-

[2]See [34] for an excellent survey of these techniques.

gons, it is not surprising that the creases between the polygons were clearly visible. Because of Mach banding even boundaries between polygons with only a subtle change in color were pronounced.

In 1972, Gouraud[19] introduced a new interpolation scheme to smooth the kinks in the polygonal surfaces. Gouraud's algorithm was similar to previous algorithms except that the normal used in the illumination calculation at each vertex was not the polygon normal. Instead, the normal assigned to each vertex was either an average of the surface normals of constituent polygons, or the actual surface normal taken from the surface that the polygon was approximating. Gouraud's interpolation method guaranteed that the change in color over the edge of one polygon onto the next would be constant. The resulting images looked smooth, but as Gouraud pointed out, because of the sensitivity of the visual system to changes in color, even linear interpolation of the color over the edge of two polygons did not entirely eliminate Mach bands.

Phong[4] introduced an improved illumination model and shading algorithm in 1975 to produce smoother and more convincing images. Phong recognized that the illumination of an object depended on the specular properties, or the gloss, of an object as well as its diffuse properties. To model the specular reflection, Phong developed a specular function dependent on the the angle between the vector from the object to the viewer and the direction of the light reflecting off the object. Phong's illumination model was a linear combination of the diffuse and specular components of light emitted from the object. The model did not work well with Gouraud's shading method. Specular highlights are very sensitive to the

shape, or more precisely, the surface normal, of the object. With Gouraud shading, there is information about the surface normal only at the vertices of the polygons approximating the surface; unless there is a huge number of polygons, there is not enough information to render the specular highlights correctly. To provide more information about the shape of the surface, Phong interpolated the vertex normals over the face of the polygon so that there would be a normal associated with each point in the polygon. Because an illumination calculation has to be calculated at every point within the polygon, Phong's algorithm is substantially slower than Gouraud's. However, the resulting images are smoother and more realistic than the images produced by Gouraud's algorithm. Thus with Phong and Gouraud shading, there were two efficient algorithms to smooth out the kinks in the polygonal surfaces.

With polygonal approximations the silhouettes of curved objects were still approximated by line segments. Because of this and the need to render exact data, work continued in displaying curved surfaces without first approximating them as polygons. In 1975, Catmull[6] introduced the z-buffer as a method to eliminate the hidden surfaces from surface patches. Although the z-buffer solved the problem of removing the hidden surfaces of curved surfaces, its speed, simplicity, and generality, coupled with the decreasing cost of memory, cemented the use of polygons to approximate curved surfaces. Ironically, the z-buffer persisted not as a hidden surface method for curved surfaces, but as a simple, cheap, and fast method for removing the hidden surfaces of polygons.

The basic elements of an efficient rendering pipeline for polygonal ob-

jects were intact by 1976. The algorithms to manipulate objects in three dimensions and then project them onto the viewing plane had been known for a long time; eliminating hidden surfaces could be done simply and quickly by using a z-buffer; and reconstructing the appearance of the curved surface, complete with simulated illumination, had been solved satisfactorily by Gouraud and Phong. With the basics in hand, the next challenge was to increase complexity and quality of the images generated by polygonally based renderers. The task of merely rendering objects was complete.

Now, the problem of rendering realistic objects had to be confronted. In order to create realistic computer generated images, the computational model had to more closely approximate nature. That meant that the illumination model had to be expanded to include more complicated lighting effects. With the more natural lighting, the shape of the objects had to become more detailed. In order to improve the illumination model, the rich models found in physics were tapped. Blinn[3] was the first to extend the illumination model by implementing the Torrence-Sparrow model for specular reflection. Cook[11] argued that computer graphics images looked like plastic because the standard specular reflection calculation was wrong. Cook pointed out that the color of a specular reflection off of a surface depends heavily on the properties of the surface; each type of material reflects different wavelengths in a unique manner. Warn[35] looked to theatrical lighting to create a more complex lighting model, complete with barn doors and directional lighting. More recently, Greenberg *et. al.* [18][8] have modelled the second order effects of light bouncing from one object onto another. Transparencies were first accomplished by Newell, Newell

and Sancha[29] in 1972 but improved by Kay[21] in 1979.

There are two ways to create more detail on the surface of an object. The most obvious way is to create very detailed objects. With the application of fractals[26]to computer graphics much progress has been made in this area. Another way to create detail is to "paint" it onto the surface. Catmull[6] introduced this method, texture mapping, in 1975. Although effective in producing objects with patterns painted on them, this method did not successfully simulate natural textures. Instead of looking like part of the object, the natural textures looked as if they too were painted onto the object. As Blinn[2] pointed out, the texture existed independent of the lights. When the lights changed position, the texture remained unchanged. To solve this problem, Blinn mapped the surface normals of a textured surface onto the object instead of colors. This way, when the lights changed, so did the illumination of the texture. Unfortunately, since only the normals were perturbed, no matter how bumpy the surface was, the silhouette of the object remained smooth.

With more complex objects, more accurate illumination models, and finer surface detail, much progress was made in creating realistic images using polygons. However, two problems loomed large. When using the z-buffer for hidden surface elimination, it was difficult to do transparencies and anti-aliasing correctly. In order to do anti-aliasing correctly, one needs to know the color of neighboring pixels in the final image. With the z-buffer algorithm, polygons are written to the framebuffer in arbitrary order. Therefore, when the color of a pixel is stored in the framebuffer, it is not known what color its neighboring pixels will be, thereby making anti-

aliasing impossible. Similarly, to calculate transparency, one must know the color of the object directly in back of the transparent object. When an object is being written to the framebuffer, this information is not available. The problem, then, with the z-buffer is that pixel values are being calculated before all the relevant data has been gathered.

If one put off calculating a pixel's value until all the polygons have been processed, it would then be possible to calculate the correct value for the pixel. The a-buffer implemented by Carpenter[5] does just that. At every pixel, all relevant information is kept in a list. Then, after all the polygons have been processed, the final color value for each pixel is calculated. Like the z-buffer before it, the a-buffer is a memory intensive algorithm. However, in the spirit of polygonal rendering techniques, the a-buffer produces reasonable quality images quickly.

With the success of the polygonal approach, interest in displaying exact curved surfaces waned in the late 1970's only to be revived by Turner Whitted[37] in 1980. Whitted resurrected an old rendering algorithm for quadric surfaces first pioneered by MAGI[24][17]in 1968 – ray tracing. To solve the hidden surface problem for quadric surfaces, MAGI "shot" rays from the viewing plane to see which object the ray intersected first. Whitted extended this algorithm by tracing the ray even further; if the object was transparent or reflective, Whitted recursively traced the transparent and reflective rays. Furthermore, Whitted included shadows by checking to see if a ray from a point on a surface to a light source intersected any objects. The pictures Whitted produced were stunning. Because of the superb image quality, interest in rendering curved surfaces was once again piqued.

Unfortunately, ray tracing objects was slow; anti-aliasing the image was *very* slow. In order to calculate which objects a ray intersected, the ray had to be tested with each object. Thus, the way to speed up ray tracing substantially is to minimize the number of intersection calculations. The most successful approach to this problem so far has been to partition the space in order to limit the number of objects a ray could possibly intersect.[16] [27] Beyond the improvements in the basic ray tracing algorithm, another cause for optimism is the ease with which ray tracing can be implemented on a parallel machine. Since each pixel is calculated independently, the color at all the pixels could be calculated concurrently. Numerous schemes for the implementation of ray tracing on a parallel machine have been proposed [7][28], and some have been implemented with excellent results[31].

Since the ray tracing paradigm closely followed the physical model of light, extending the algorithm to include phenomenon such as translucency, lens effects, accurate shadows, and motion blur was straightforward. Furthermore, at its heart and at the surface, ray tracing is a sampling process. Therefore, the traditional methods used to solve anti-aliasing could be applied directly to ray tracing[13][9][22]. In one fell swoop, Cook, Porter, and Carpenter's distributed ray tracing algorithm[10] provided a general model to include complex lighting phenomenon and the method to solve the anti-aliasing problem. In the same way the a-buffer provided a general model to solve the major problems of polygonal rendering, distributed ray tracing is the general model for rendering arbitrary curved surfaces with high precision, complexity, and, in the end, quality.

In 1984, both of the approaches to rendering set in motion in the early

1960's came to fruitful conclusions. The a-buffer allowed for fast rendering and anti-aliasing of polygonal objects. Distributed ray tracing successfully solved the problem of rendering high quality images of curved surfaces.

Chapter 3

An Overview of the Rendering Pipeline: Sampling, Mapping, and Reconstructing

On one side, there is a mathematical world filled with information. On the other side lies a grid of zeros in a framebuffer. Three dimensional computer graphics links the two. It determines which points from three dimensions should be mapped to the screen, how they should be mapped, and finally, how the image should be constructed from the points.3

To move points on an object from three dimensions to the two dimensional screen, one must create a projection map tying a point on an object to a point on the screen. The two main types of rendering, ray tracing and polygonal rendering, both address this problem, but from different directions. In ray tracing, the map [1] starts in two dimensions and goes to three.

[1] This map corresponds only to the first level of rays. This first level of rays determines the map from three space to the viewing plane. Subsequent rays define the reflectance and refraction maps from one surface to another.

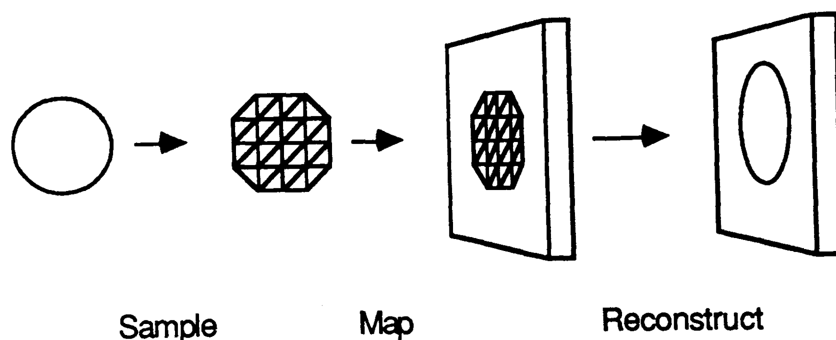


Figure 3.1: *Sampling, mapping, and reconstructing a surface.*

One chooses a pixel in two dimensions and asks the question: Which point in three dimensional space will be assigned here? The map for polygonal rendering is the inverse. One chooses a point on a three dimensional surface and asks: To which pixel on the screen will this point be assigned? Though by different methods, both ray tracing and polygonal rendering explicitly create a map linking three dimensional surfaces to the screen.

Even if one knows the map between a surface and the screen, it is clear that one cannot map all the points of the surface; the surfaces are continuous. To solve this problem, a finite number of discrete samples are taken from the surface to approximate it. Both ray tracers and polygonal renderers sample the surfaces in order to project the surface into two dimensions.

The method of sampling the objects differs slightly in ray tracing and polygonal rendering. Since the projection map for ray tracing is from two dimensions to three, one can use the screen as a guide to taking the three dimensional samples. One finds a pixel on the screen and uses the projection

map to find which point on a surface is mapped to it; a ray is cast from the pixel to find which object it intersects. One is guaranteed that every sample taken will be seen.

In polygonal rendering, sampling transforms the continuous curvaceous surface into a networks of flat polygons. Each vertex in this net represents a sample point from the surface. Since one samples the surface irrespective of the screen, many of the samples will be extraneous. Hidden surface elimination and clipping sift the samples, leaving only those destined for the screen.

After the samples have been mapped from three dimensions to two, the final image must be constructed from these samples. In ray tracing, the samples' final destination on the screen is carefully chosen. [2] Therefore, to construct the final image, the samples must only be filtered to minimize the artifacts of sampling. With polygonal rendering, the samples are spread sparsely on the screen.[3] Since the number of samples is less than the resolution of the screen, the image must be reconstructed on the viewing plane.

In short, both ray tracing and polygonal rendering share the three basic steps of rendering: sampling the objects, mapping the samples from three dimensions to two, and constructing the final image in two dimensions from those samples. The next chapters will examine the polygonal rendering

[2]The destination of the sample might have been chosen by a stochastic method, for instance *jittering* or a poisson distribution, to facilitate anti-aliasing.

[3]It is assumed that the original surface was sparsely sampled; a viewpoint can be chosen arbitrarily close to the object to guarantee that the number of samples mapped to the screen will be less than the number of pixels.

pipeline more carefully.

Chapter 4

Sampling

Sampling surfaces for rendering is a two step process. In the first step, each surface is sampled in its own local frame of reference, object space. The resulting set of samples form a general polygonal representation of the surface. This representation can be stored and used over again from scene to scene. The second round of sampling occurs in eyespace when all the surfaces have been assembled into one scene. These samples are used to create the final image. Since these new samples are intimately connected to the specific scene, they must be re-calculated for every new scene(fig. 4).

4.1 Sampling an Object for Geometric Modeling

One wishes to describe a three dimensional, continuous surface with a set of samples. This continuous surface may only be in one's imagination, serving as a guide to constructing the samples directly, as is the case with most

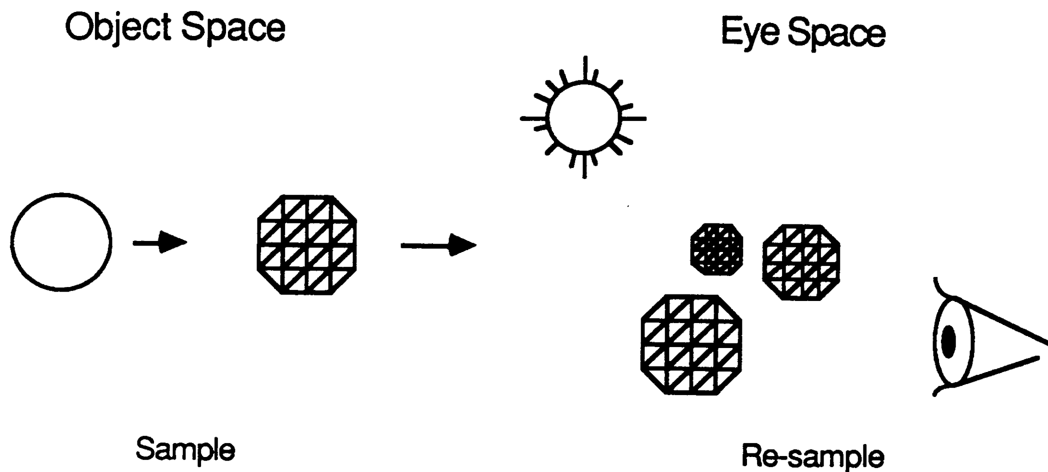


Figure 4.1: *The objects are sampled in object space, then again in eyespace.*

object generation programs. Or, the continuous surface may be parametrically defined, with its control points either taken from a real object, or synthesized on a computer. In both of these cases, the samples have to provide enough information to generate a facsimile of the original surface.

What information is enough to generate the surface? The amount and type of information taken at each sample point depends on how one plans to re-create the image. However, there are two broad classes of information that a sample might contain. The first class describes the shape of the surface. The other describes how the surface interacts with light.

Shape Information

The most rudimentary information a sample point must contain is its location and how it connects to other points. For its location, three coordinates, (x, y, z) are kept. There are a variety of ways to specify the connectivity of

a point with other points. This information can be expressed explicitly in a list of edges, or implicitly by the ordering of the points. In either case, both the location and connectivity of the point must be attached to the sample. If a shaded image is desired, the normal of the surface at the sample point is kept. In short, the location of each sample point, the connectivity between sample points, and the surface normal at each of sample point are the major components describing the shape of a surface.

Surface Qualities

To create a shaded image of an illuminated object, it is important to keep information describing how the object interacts with light. Depending on the lighting model being used, this information varies in type and quantity. However, at the very least, each sample should contain information regarding the diffuse and specular properties of the object. For a slightly more sophisticated illumination model, the transparency, the reflectivity, and the index of refraction of the surface at the sample point are kept.

4.2 Sampling The Scene for Rendering

When a continuous surface is sampled, it is sampled in isolation without regard to other objects. The resulting samples' locations and surface normals are defined in the surface's local frame of reference. In a lighted scene with numerous surfaces, the surface's position in relation to the lights, the eyepoint, and other surfaces determines its final illumination. Therefore, to calculate the illumination of a surface properly, information regarding the

relationship of the surfaces to each other must be included in the samples. To get this information, all the lights, all the surfaces, and the eyepoint are transformed into the same frame of reference.

The samples are then re-calculated in the new frame of reference. The content of these outgoing samples depends on the type of shading being used. In Gouraud shading, the illumination for each sample is calculated. The outgoing sample contains a color value and a location. In contrast to Gouraud shading, the illumination calculations for Phong shading are put off until the two dimensional surface is being reconstructed. Therefore, with Phong shading, the samples contain the list of parameters necessary for the illumination calculations.

4.3 A Comparison of Phong and Gouraud Shading

In Gouraud shading, one illumination calculation is performed for each vertex of a polygon. In Phong shading, an illumination calculation is performed for these vertices *and* each point inside of the polygon. Therefore, Phong shading is slower than Gouraud shading. Furthermore, since the illumination calculations are put off until the very end, Phong shading must store the illumination parameters for each polygon [1] Thus, in addition to being slower than Gouraud shading, Phong shading also requires more

[1]This memory load can be relieved if it assumed that the illumination parameters are constant over a surface. Then the parameters are stored for each object.

memory.

Although more efficiently produced, the image created by Gouraud shading is substantially inferior to the image created by Phong shading. To understand why, one must look at how the illumination calculation affects the reconstruction of a object from its samples. When the geometric model of the object's surface is created, enough samples are generated to adequately describe the surface. The sampling rate for the surface is tailored to the spatial frequency inherent to the continuous object's shape. As a result, the shape of the object can be re-constructed accurately with the given sampling rate.

Due to lighting effects such as specular reflection, the spatial frequencies of the colors in an object tend to be higher than the frequencies of its shape. Therefore, the sampling rate used to determine the shape of the object is not adequate for the color. If the color is sampled at this rate, aliasing can occur.

Since the color of an object depends on the lights, the eyepoint, and surrounding objects, the information regarding the object's color is re-sampled in the common frame of reference. Ideally one would hope to adjust the sampling rate to the spatial frequency of an object's color. Unfortunately, the sampling rate is already set; there are as many samples as there are vertices in the polygonal approximation of the surface. Gouraud shading goes ahead and samples the color at this point, thereby risking the possibility of aliasing in the reconstructed image.

Phong shading takes a more conservative approach. Instead of sampling the color, Phong shading retains enough information to reconstruct

the shape of the surface. Since, at the outset, the sampling rate was tailored to the shape of the object, the shape can be reconstructed faithfully. Only after the shape of the surface has been reconstructed does Phong shading illuminate the surface and calculate its color. This way, the color never has to cross the barrier of being sampled and then reconstructed. The final image produced by Phong shading is superior to the image produced by Gouraus shading because Phong shading avoids sampling and reconstructing the high spatial frequencies of the surface's color.

4.4 Summary

The first step in rendering a continuous surface is to distill into a manageable form the overwhelming amount of information describing a continuous surface. Taking discrete samples of the surface solves this problem.

Each sample taken from the surface contains data regarding the surface's shape and optical properties. The shape of the surface is defined in a local coordinate system without regard to any other surface. In order to render a scene with many objects, the samples have to be moved into a common frame of reference. Each sample is re-calculated in this general frame of reference. In Gouraud shading the samples are condensed into a location and an r, g, b value. In Phong shading the location and surface normal of each point are converted to the new frame of reference and retained. Since the illumination value introduces high frequency components due to specular reflection, Gouraud shading introduces high frequency components to the samples. These frequencies are not faithfully reproduced when the

image is reconstructed. Phong shading avoids this problem by introducing the high frequencies after the image has been sampled and reconstructed.

During the sampling process described in this chapter, the samples were moved from their local space to a common frame of reference. The next chapter examines this transformation and the subsequent mapping of the points to the screen.

Chapter 5

Mapping the Samples

As the last chapter alluded, there are two sections in a sample's journey from its local coordinates to its final place on the screen. The first goes from the local coordinates, object space, to the common frame of reference, eye space. [1] The second goes from eye space to the screen.

On the first leg, all the information is carried along. The lights, the surface normals, and the sample locations are all transformed by the modeling matrices into a common frame of reference. Only the locations of the samples are transformed further to the screen; the surface normals, the lights, and the eye point stay put in eye space.

On the second leg of the journey, only the locations of the sample points change. In effect, this creates a link between the information necessary for illumination in eye space and its position on the screen. The viewing transformations change the location of each sample point from eye space to the

[1] World space can also be used as a common frame of reference. However, eye space is more convenient because the eye point is located at the origin. Therefore, the vector from the eye to the surface is the coordinates of the point on the surface.

screen. After the viewing transformation is complete, each sample contains illumination information in eye space, and an accompanying location on the screen. The illumination calculations can then be performed properly in eye space, with the resulting color located at its correct spot on the screen.

Thus, the first transforms, the modeling matrices, move all the lights and sample points into a common frame of reference. The modeling matrices are the familiar scaling, translation, and rotation matrices. After the objects have been maneuvered into their position in eyespace, the locations of these samples are transformed to the display by the viewing transformations. The viewing transforms are described in detail in appendix D.

Chapter 6

Reconstructing the Surface

There are three main steps to reconstructing the surface: scan conversion, interpolation, and, for anti-aliasing, fragment packing in the a-buffer. Scan conversion fills each polygon with new sample points. Interpolation attaches illumination values to these constructed samples. And, the a-buffer and fragment packer convert the samples collected from all the polygons into a final image.

6.1 Scan Converting the Surface

Given the points and edges of a polygon that have been projected onto a display device, it is now necessary to generate a grid of points which fill the polygon. Scan conversion creates that grid. The basic idea behind scan converting a polygon is to intersect the polygon with a series of evenly spaced lines corresponding to scanlines on the output device. The resulting intersections define the edges of the polygon on each scanline. The regions between the polygon edges on each scanline are then filled. To speed up this

process, it is desirable to take advantage of the continuity of the polygon. Each point on a polygon is similar to its neighbors. Therefore, to get information about a point, one needs only to modify the information already known about its neighbor. Scan conversion is the algorithm which exploits the continuity of a polygon in order to fill in the polygon's outlines with a regular grid of points.

6.1.1 Spatial Coherence

Scan conversion uses two types of spatial coherence. The first is coherence from point to point within the polygon. Since a polygon is continuous over its surface, one needs to know only the polygon's edges to fill its interior. The second type of coherence is scanline to scanline coherence. One can calculate the edge intersection on a scanline given the intersection of that edge with the previous scanline. Thus, point to point coherence allows one to fill a region on a scanline by defining only the region's endpoints. Scanline to scanline coherence allows one to incrementally calculate the region's endpoints from one scanline to the next.

6.1.2 Exploiting Coherence

Exploiting the point to point coherence is trivial knowing only the boundaries of a scanline region; one simply fills the region. Taking advantage of scanline to scanline coherence is slightly more involved. The first step

is to scan the polygon in order, from bottom to top.[1] As one goes from one scanline to the next, the edge intersections can be calculated incrementally. To do this, one needs to know the initial intersection point and a rule for finding subsequent intersections. The initial intersection point comes for free; it is the minimum endpoint of the edge. The rule for finding subsequent intersection is the following:

$$x_{i+1} = x_i + k \qquad (6.1)$$
$$y_{i+1} = y_i + 1$$

where $k = 1/m$ and m is the slope of the edge.

Thus, by exploiting the polygon's coherence, after the initial overhead, calculating an edge intersections involves only one addition. Filling the region between two edges requires one addition for each point.

The question now is, how does one know which of the edges of the polygon the scanline intersects? The answer is found by sorting the polygons according to their range in y. If the scanline falls within the range of an edge, then the scanline intersects that edge. And, subsequent scanlines will continue to intersect that edge until a scanline exceeds the edge's maximum y value. So, the strategy is to sort the edges according to their minimum y value. The edges are kept sorted in the edge table (ET). As soon as the scanline intersects an edge, the edge becomes "active" and is put onto a list of active edges (the AET). When the scanline exceeds the range of an

[1] It is assumed that positive y points up; positive x points to the right. The "minimum endpoint" of an edge is the endpoint with the smallest y value. The "maximum endpoint" is the endpoint with the larges y value.

field	*description*
ymax	The maximum y value of the edge.
x	The boundary of the scanline region associated with this edge.
dx	The change in x for each increment in the scanline.

Figure 6.1: *The representation of an edge during scan conversion.*

edge, the edge is removed from the AET and is retired.

6.1.3 The Edge Structure

Each edge has the value of x for the current scanline, the maximum y value for the edge, and the increment dx/dy. When the edge is in the ET, the x value represents the x at the minimum endpoint. Then, as the scanline rises, the increment dx/dy is added to x. As a result, in the AET, x represents the boundary for a scanline region.

The minimum representation of an edge for scan conversion is shown in Figure 6.1.

If memory limitations permit, it is also useful to keep additional information. Explicitly storing both endpoints in addition to the x value at the current scanline oftentimes eases expansions in the basic scan conversion algorithm. Also, a flag to indicate whether the edge is part of a local maximum, or the edge is a silhouette edge is useful to maintain. Therefore, a slightly more complete representation is shown in figure 6.2 [2]:

[2] When values associated with the vertices are interpolated in conjunction with scan conversion, the edge representation grows larger

field	*description*
ymax	The maximum y value of the edge.
x	The boundary of the scanline region.
dx	The change in x for each increment in the scanline.
maxx	The x value at the maximum endpoint.
miny	The y value at the minimum endpoint.
flag	For general use.

Figure 6.2: *A more complete edge representation.*

6.1.4 The Edge Table

The edge table (ET) is a one dimensional array with one slot for each scanline. If the image is going to be scanned with a vertical resolution of one thousand scanlines, the array will have one thousand elements. In each of these elements there is a list of the edges which begin [3] at the corresponding scanline. If there are no edges which start at a given scanline, the corresponding slot has a null value. [4] All of the edges in the ET are in holding, waiting to be intersected by the scanline moving up the polygon. An edge is "activated" when the scanline reaches its slot in the ET. When the edge is activated, it is removed from the ET and sent to the AET. By sorting the edges and by storing them in order, the ET simplifies the search

[3] When placing an edge in the ET, the fractional part of the beginning point of the edge is ignored, ie. the edge is indexed into the array by the truncated value of its minimum y value.

[4] Horizontal edges are not loaded into the ET. Since the edge is parallel to the scanline, ignoring the edge will not affect the shape of the polygon.

for newly intersected edges.

6.1.5 The AET

The AET is a sorted list containing all the edges intersect the current scanline. All of these edges come in pairs; one edge defines the beginning of a scanline region, with its complement edge defining the end. To find which edges bracket a scanline region, the edges on the AET are sorted by their minimum x value. Stepping through the sorted list, then, the first two edges define the first scanline region, the second two define the second scanline region, and so forth.[5] After these regions have been filled, the scanline intersections for the next scanline are calculated. To find these new intersections, one increments the existing ones by the appropriate increment in equation 6.1.

6.1.6 Summary of The Algorithm

The following is an outline of the basic scan conversion algorithm:

1. Load the ET with the edges of a polygon.
   (a) If an edge is horizontal, ignore it; do not load it into the edge table.
2. For each scanline:
   (a) Add edges to the AET

[5]Notice that the AET will always have an even number of edges in it.

(b) Remove inactive edges from the AET.

(c) Use the AET as a guide to fill each scanline regions.

(d) Increment the edge intersections.

3. Stop when there are no more edges on the AET.

6.1.7 Problems Encountered in Scan Conversion

Order of The AET

When an edge is inserted into the AET, it is imperative to ensure that the AET is sorted correctly. If it is not, the beginning and end edges of a scanline region can be reversed. This occurs when the endpoints of two incoming edges form a local minimum of the polygon. In this case, the two edges have the same x value. Since the edges are sorted according to their x values when they are inserted into the AET, there is no guarantee that the beginning edge will fall on the correct side of the end edge. To protect against the edges' order being inverted, it is necessary to examine the dx value (1/slope) of the edges if two edges share the same x value. The edge with the smaller dx value lies to the left of the other edge.

Sub-Pixel Polygons and Slivers

If a polygon's extent in y is less than one, it will be scanned incorrectly with the current algorithm. The algorithm fails because the resolution of the ET is not fine enough to sort the sub-pixel polygon's edges in y. Therefore, the beginning and end of the scan region for this polygon cannot be determined.

To obtain values for a these problem polygons, they must be treated as special cases.

The following is a simple solution to filling the in sub-pixel polygons and slivers. First detect these polygons by testing each polygon's extent in y. Then, find the edge with the minimum x value, and the edge with the maximum x value. Set the edge with the minimum x value as the beginning edge of a scanline region and the maximum x value as the end of a scanline region. Fill this region.

6.1.8 Summary

Filling polygons is a central step in the rendering pipeline. Because testing each pixel serially to check if it lies within a polygon would be too slow, the polygon scan conversion algorithm described above was developed. This method of scan conversion can be easily modified to act as an interpolator of values over the polygon. Interpolating values associated with the polygon vertices is the next step in the pipeline.

6.2 Interpolating Values

Scan conversion easily can be modified to interpolate values associated with the vertices over the entire polygon. Usually, the values represent the color, transparency, reflection, refraction, and other illumination parameters. Gouraud[19] used it to interpolate color values. Phong[4] used it to interpolate surface normals. For z-buffer hidden surface, the z values

field	*description*
ymax	The maximum value of the edge.
x	The boundary of the scanline region.
dx	The change in x for each increment in the scanline.
value1	A value to be interpolated over the polygon.
Δ value1	The increment for value1 with respect to y.
value2	A value to be interpolated over the polygon.
Δ value2	The increment for value1 with respect to y.
value3	A value to be interpolated over the polygon.
Δ value3	The increment for value1 with respect to y.
etc.	Other interpolated values and their increments.

Figure 6.3: *An edge modified for interpolation.*

at the vertex are also interpolated over the polygon. Interpolating these values involves only minor additions to the scan conversion of polygons.

Similar to simple scan conversion, interpolating values over the polygon exploits scanline to scanline and point to point coherence. The method is to incrementally interpolate the values at the vertex both horizontally and vertically, so that as each new point is filled in, an interpolated value can be given to it. The first step is to interpolate the values along the edges of the polygon. Then, the values are interpolated vertically between the polygon's edges. Usually this interpolation occurs in conjunction with scan conversion.

The New Edge Structure Since this interpolation occurs along with scan conversion, it is convenient to extend the edge structure used in scan conversion. The resulting edge structure is shown in figure 6.3, where $\Delta value = (V_{max} - V_{min})/(y_{max} - y_{min})$; V_{max} is the interpolated value at the maximum endpoint of the edge; V_{min} is the interpolated value at the minimum endpoint of the edge; and $(y_{max} - y_{min})$ is the rise of the edge.

Interpolating Over a Scan Region These values serve as the bounds of a second interpolation along the horizontal points on the scanline. For each region on a scanline, the increment is the following:

$$(V_{end} - V_{begin})/(x_{begin} - x_{end})$$

where V_{begin} is the value associated with the begin edge, V_{end} is the value associated with the end edge, and $(x_2 - x_1)$ is the length of the scan region. As a region is filled in, V_{begin} is incremented by ΔV_x. For instance, the nth point from the beginning edge will be assigned the value, $V_{x_{begin}} + n\Delta V_x$.

A Summary of the Modified Algorithm

The following is an outline of the modified scan conversion algorithm.

1. Convert the polygon into edges suitable for scan conversion.
2. Load the ET with the edges of a polygon
3. For each scanline:
   (a) Add edges to the AET
   (b) Remove inactive edges from the AET

(c) For each scanline region

   i. Calculate increments for interpolated values over scanline region.

   ii. For each new pixel

      - Use the interpolated value.
      - Draw the pixel
      - Increment the interpolated value.

(d) Increment the interpolated values for new scanline.

(e) Increment the edge intersections.

4. Stop when there are no edges left in the AET

6.3 The A-buffer: Anti-aliasing

When trying to include anti-aliasing in a renderer which uses a z-buffer, fundamental problems with the z-buffer arise. One of the strengths of the z-buffer is its simplicity. This simplicity, however, complicates anti-aliasing.

First, A Flawed Anti-Aliasing Algorithm

If one were trying to modify the z-buffer algorithm to include anti-aliasing, one might try the following. As each new polygon is scan converted, each pixel lying on the polygon is given a coverage value. [6] Then as the color

[6] If a pixel is thought of as a box, the coverage is the percentage of the box that is filled by a polygon. Pixels completely inside the polygon have a coverage of 1.0; pixels on the edge have a coverage between 0.0 and 1.0; pixels outside of the polygon have a coverage

values for each pixel are written ¡to the frame buffer, the new color value is mixed with the color already in the frame buffer to arrive at the final color value. The following formula determines the mix:

$$color = (newcolor)(coverage) + (oldcolor)(1.0 - coverage),$$

where coverage is a number from 0.0 to 1.0, with 1.0 meaning that the pixel is completely covered by the polygon.

To see the problem with this algorithm, imagine that the frame buffer is initialized to black. A white object is being written to the frame buffer. At the edges of the polygon, the coverage is less than one, so the resulting polygon in the framebuffer is white with a grey border. Later, an adjacent white polygon is written to the framebuffer. At its edges, the coverage, too, is less than one. Therefore, when it is laid down beside the polygon already in the framebuffer, the color at its edges are mixed with the grey edge of the first polygon. Grey is mixed with grey, resulting in grey. Unfortunately, the edge between the two polygons should be solid white.

An alternate solution is to anti-alias only edges which form the silhouette of an object. Although this solves the problem for adjacent polygons, a line will still appear between adjacent objects. The fundamental problem with both of these approaches is that the final color value for the pixels on either side of an edge must be known before that edge can be anti-aliased correctly. The manner in which the polygons are written to the screen in a z-buffer algorithm precludes knowing this information until all the polygons have been processed.

---

of 1.0.

The a-buffer introduced by Loren Carpenter[5] solves this problem by retaining all relevant information until all the polygons have been processed. Each polygon is is split into fragments along the boundaries of the pixels it covers. The fragments from every polygon are stored in the a-buffer. Only when all of the polygons have been processed are the final colors for each pixel calculated. The a-buffer puts off anti-aliasing until the color values on either side of all the edges are known.

6.3.1 The A-Buffer Algorithm

Data Structures

The a-buffer is a two-dimensional array the same size as the framebuffer, except that in addition to storing a minimum z for each pixel, the a-buffer stores a list of polygon fragments. The structure containing the list of fragments is a *pixelstruct*. There is one pixelstruct in the a-buffer for each pixel on the screen.

Why store a list of fragments at a pixel instead of just the closest one? When a fragment is the closest to the viewer, opaque, and fully covers the pixel, a list of fragments is not necessary; only the color of the polygon and the z value at the pixel are stored in the pixelstruct. This type of pixelstruct is termed *simple*. However, if a polygon fragment does not fill the entire polygon, or if the polygon is transparent, a list of fragments must be maintained. This list is necessary to calculate the pixel's final value. This type of pixelstruct is called a *fragment list*. In short, depending on the nature of the covering polygons, a pixelstruct will either be simple or a

field	*description*
z	The minimum z for the pixel.
r,g,b	The color of the pixel.
a	The coverage of the pixel.

Figure 6.4: *The simple pixelstruct.*

field	*description*
z	The minimum z for the pixel.
fragment list	This is the head of the fragment list.

Figure 6.5: *A fragment list.*

field	*description*
next	The next fragment on the list.
r,g,b	The color of the fragment.
opacity	The opacity of the fragment.
area	The exact area of the pixel that the fragment fills.
object	The object from which the fragment came.
m	The bit mask representing the shape of the fragment.
zmax	The maximum z of the fragment.
zmin	The minimum z of the fragment.

Figure 6.6: *A pixel fragment.*

fragment list. The structure in figure 6.4 represents a simple pixelstruct, the structure in figure 6.5 represents the head of a fragment list, and figure 6.6 represents a fragment on that list.

The Z Value No matter if the pixelstruct is simple or a fragment list, the pixelstruct has single z associated with it. Beyond merely storing the minimum z value at the pixels, the z also serves as a flag specifying whether the pixelstruct is simple, a fragment list or empty. If the pixelstruct is simple, z is positive; if the pixelstruct is a fragment list, z is negative; and, if the z is set to a small number (minus the machine size), the fragment is empty.

PixelMasks The percentage that an edge covers a pixel is not enough information to specify fully the nature of the intersection between an edge and a pixel. For example, suppose that an edge vertically splits a pixel in half. Suppose also that another edge farther away from the viewer horizontally splits the same pixel in half (see figure 6.7).

The coverage for both these pixels is 0.5. Therefore, both will contribute 50% to the final color of the picture. However, only half of the back fragment is seen by the viewer; the other half is blocked by the edge in front. As a result, the front fragment should contribute 50% to the final color, the back edge should contribute 25%, and the remaining 25% should come from the background. From this example, it is clear that in addition to the coverage at pixel, the shape of the region inside of a polygon must be stored.

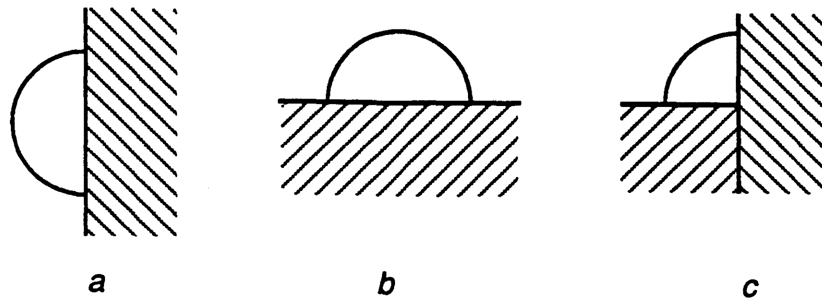


Figure 6.7: *The final coverage value for a pixel depends on the shape and location of the edges.*

To solve this problem, pixel masks are introduced. A pixel mask is a grid of bits representing a pixel at a higher than screen resolution. If a bit lies within a polygon, its value is 1; if it is on the outside, its value is 0. The pixel mask, thus defines the shape of the pixel region covered by a polygon. By knowing the shape of pixel fragments, the correct coverages for a pixel can be calculated.

6.3.2 Modifying Scan Conversion

Scan conversion proceeds as usual in the a-buffer algorithm until it is time to write a pixel to the framebuffer. Instead of loading a color into the framebuffer as usual, a bitmask must be created, a coverage has to be calculated, and the minimum z and the maximum z of the fragment covering the pixel have to be calculated. Given the intersection points of an edge and a pixel, the coverage and the bitmasks can be found by table look-up, or by analytic calculations. But first, one must find these intersections.

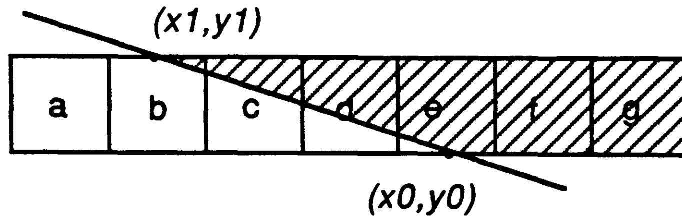


Figure 6.8: *An edge intersecting a scan line of pixels at points* (x_0, y_0) *and* (x_1, y_1).

During scan conversion, it is assumed that the scanlines occur on integer values, 0.0, 1.0, 2.0,... Thus, the initial intersection between the scanline and the edge occurs at the bottom of the pixel. (see figure 6.8.)

If the edge is the start of a new scan region, x_0 will be the boundary of the region. Pixel *e* is the first pixel in the region. Pixels *b* and *c* will not be scanned. As a result of this case, it is important to find and scan the region of partially covered pixels on an edge explicitly. Given the starting point, (x_0, y_0), and the slope of the edge, these intersections can be found by starting at (x_0, y_0) and using the following formula to find subsequent intersections:

$$\begin{aligned} x_{i+1} &= x_i + increment \\ y_{i+1} &= y_i + 1/dx \end{aligned} \quad (6.2)$$

where increment is -1 if the slope of the line is negative, and $+1$ if the slope is positive.[7] Incrementing stops when the current y value is greater

[7]Note that the first x may not be on an integer value. Therefore, the first increment

field	*description*
mask	The bitmask describing the covered region.
area	The area of the covered region.
zmin	The minimum z value for the fragment.
zmax	The maximum z value for the fragment.

Figure 6.9: *An element in the mask buffer.*

than or equal to $y_0 + 1$, or is greater than or equal to the maximum y of the edge. [8]

If the maximum y occurs within a pixel, the process described above encounters a problem: there is not enough information to fill the remainder of the pixel. A simple solution to this problem is to scan the edges into a temporary buffer before the scan regions are filled. The following is an explanation of this algorithm.

The Mask Buffer The mask buffer is a one dimensional array the length of one scanline. It serves as a holding cell for all the pixel masks, areas, $z$min's, and $z$max's created by the intersection of scanline with a polygon. Each element in the array corresponds to an pixel in the scanline. The elements contain the information shown in 6.9

For each new scanline, the masks in the buffer are initialized. These masks are assumed to be fully covered; their bits are all set to 1. After the masks have been initialized, the new edges (if any) in the edge table (ET)

---

in x may be a fraction of the normal unit increment.

[8]The code for this function appears in the appendix.

are loaded into the active edge table (AET). Before the inactive edges are purged from the AET, every edge on the AET is scanned into the mask buffer. For each edge, the bitmasks, areas, and z values are created as described previously. When the bitmasks are loaded into the mask buffer, they are "and'ed" to masks currently in the buffer.

Once all the edges in the AET have been scanned, the inactive edges are removed from the AET, and scan conversion continues as usual. When it is time to create a fragment, the mask buffer is indexed to find the mask, the area, and the z values in mask buffer element corresponding to the current pixel being written.

An example will show how this procedure solves the maximum y problem described earlier. Suppose one is scanning the polygon vertex shown in figure 6.10. Since the edges are being loaded into the mask buffer after the new edges have been added to the AET yet before the inactive edges have been flushed, both edge a and edge b are still in the AET. After edge a has been scanned into the mask buffer, the resulting bitmask is shown in figure 6.10(ii). Then, after b is scanned in and "and'ed" into the mask buffer, the final bitmask correctly defines the corner of the polygon (figure 6.10(iv)). Note that this process will also create the correct mask for sub-pixel polygons.

6.3.3 A Summary of the New Scan Conversion

The following is an outline of the basic scan conversion modified to accommodate the a-buffer.

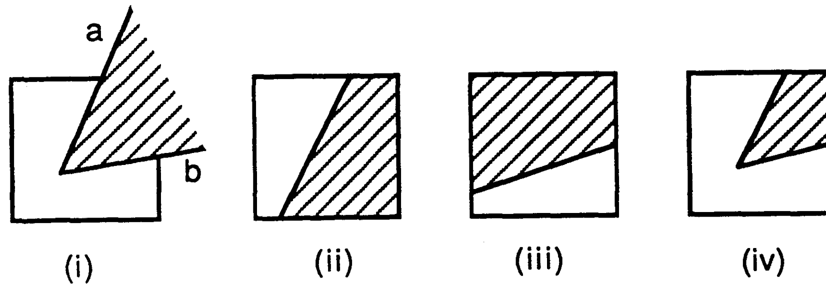


Figure 6.10: *(i) The polygon corner defined by edges a and b. (ii) The pixel mask resulting from edge a. (iii) The pixel mask resulting from edge b. (iv) The result of "and"ing the two masks. This is the final pixel mask in the mask buffer.*

1. Load the ET with the edges of a polygon.
   (a) If an edge is horizontal, ignore it; do not load it into the edge table.
2. For each scanline:
   (a) Add newly activated edges to the AET
   (b) Load the mask buffer with *all* the edges in the AET. All the pixelmasks, areas, zmin's and zmax's for a scan line are computed.
   (c) Remove the inactive edges from the AET.
   (d) For each pixel in each scan region:
      i. Calculate the color value
      ii. Find the bitmask, area, zmin and zmax for the current pixel in the mask buffer.

iii. Create a fragment.

iv. Insert it into the *a*-buffer.

(e) Increment the edge intersections.

3. Stop when there are no more edges on the AET.

4. After all the polygons have been processed, pack the *a*-buffer.

6.3.4 Creating a Pixelstruct

How a pixelstruct is created depends on both the incoming data and the data already existing in the pixel's location in the a-buffer. There are two types of incoming polygon fragments: partially covered and completely covered. These cases are handled differently depending on the state of the pixelstruct receiving the new fragment. The pixelstruct can be simple, it can be a fragment list, or it can empty. Thus, there are six possible cases. The following are how each these cases is handled:

Incoming Pixel is Completely Covered, Existing PixelStruct is Empty This requires no extra computations past transferring the data into the simple pixelstruct.

Incoming Pixel is Completely Covered, Existing Pixelstruct is Simple

- Now that there are two fragments for this pixelstruct, a fragment list must be created.

- The existing simple pixelstruct is converted into a fragment and made head of the fragment list.
- A new fragment is allocated for the incoming data.
- The new fragment is inserted into the newly formed list.
- To indicate that the pixelstruct is a list, the sign of the pixelstruct's z is switched from positive to negative.

Incoming Pixel is Completely Covered, Existing Pixelstruct is a Fragment List The incoming fragment is allocated and inserted into the fragment list.

Incoming Pixel is Partially Covered, Existing Pixelstruct is Empty

- The incoming fragment is allocated and made the head of the fragment list.
- To indicate that the pixelstruct is a list, the sign of the pixelstruct's z is switched from positive to negative.

Incoming Pixel is Partially Covered, Existing Pixelstruct is Simple

- The existing simple pixelstruct is converted into a fragment and made head of the fragment list.
- The new fragment is inserted into the newly formed list.
- To indicate that the pixelstruct is a fragment list, the sign of the pixelstruct's z is switched from positive to negative.

Pixel is Partially Covered, Existing Pixelstruct is a Fragment List The incoming fragment allocated and is inserted the fragment list.

6.3.5 Inserting Fragments

Inserting a fragment into a fragment list involves three functions, sorting the fragments in z, combining fragments, and throwing away fragments to save space. The fragments are sorted in z to facilitate packing them later on. Combining fragments occurs when fragments on the fragment list have the same color, and their pixelmasks combine to make a fully covered pixel mask. This happens at the border of polygons from the same object. Another method to save space is to throw away all the fragments behind fully covered, opaque fragments. Note that after pixelmasks are combined, they may form fully covered, opaque fragments, in which case all the fragments behind can thrown out. Thus, after combining fragments, the resulting fragment should be checked to see if it is opaque and fully covered. If the opaque and fully covered fragment is the first on the fragment list, the list can be condensed into a simple pixelstruct.

6.3.6 The Fragment Packer

The fragment packer is a recursive routine to convert the data stored in the a-buffer into an array of r, g, b, and coverage values. The packer is applied to each pixelstruct in the a-buffer. Simple pixelstructs need no processing. Fragments lists, on the other hand, need to be collapsed into the final color and coverage values.

Packing a Fragment List

To understand how the fragment packer works, it is useful to think of each fragment as a painted pane of glass. The painted region corresponds to the part of the fragment which is covered by a polygon. The clear region corresponds to the part of the fragment which is not covered. Simply put, the fragment packer finds the average color of a fragment. The following equation finds this average:

$$C_{final} = C_i A_i + C_{i-1} A_{i-1} \tag{6.3}$$

where

C_{final} is the average color;

C_i is the color of the painted portion of the fragment;

C_{i-1} is the color which shines through the clear region of the fragment;

A_i is the region of the fragment which the pain covers;

and A_{i-1} is the area of the clear region;

Note that the area A_{i-1} can also be thought of as the area of the visible region of the fragment behind, fragment N in figure 6.11.

For the first fragment on the list, fragment M (figure 6.11), the areas A_i and A_{i-1} are known: A_i is given and the visible region of the second fragment, M, A_{i-1}, is equal to 1.0 - A_i. The color C_{i-1}, however, is not yet known. Therefore, before calculating the color of M, one must first calculate the color, C_{i-1}, of fragment N.

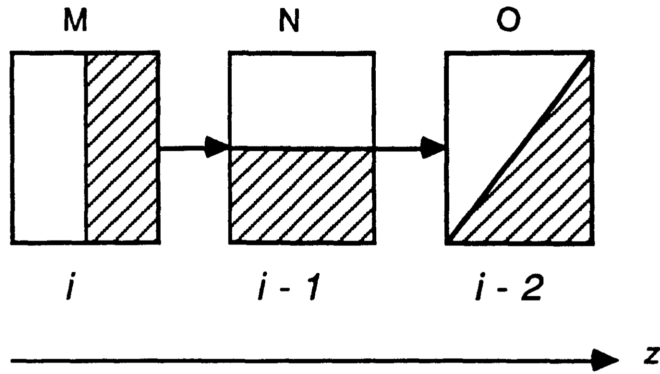


Figure 6.11: *A list of fragments in a pixelstruct.*

The color C_{i-1} is the average color of the visible region of fragment N in figure 6.11. This region consists of a clear and a painted region. Finding the average color of the visible region of fragment N is a similar problem to finding the average color of fragment M. Thus equation 6.3 again can be used to find C_{i-1}.

Restating equation 6.3, the following recursive equation is used to find the average color of fragments from opaque polygons:

$$
\begin{aligned}
C_{average} &= P_i A_i + T_i Q_i \\
T_i &= P_{i-1} A_{i-1} + T_{i-1} Q_{i-1} \\
Q_i &= Q_{i+1} - A_i \\
T_0 &= C_{background}
\end{aligned}
\tag{6.4}
$$

where

$C_{average}$ is the final value for the fragment;

P_i is the color of the covered region of the current fragment;

P_{i-1} is the color of the covered region of the next fragment on the list;

A_i is the area of the covered and visible region of the current fragment;

A_{i-1} is the area of the covered and visible region of the next fragment on the list.;

T_i is the color of the region of the current fragment which is not covered;

T_{i-1} is the color of the uncovered region of the next fragment on the list.

Q_{i+1} is the area of the uncovered and visible region of the previous fragment on the list.

Q_i is the area of the uncovered and visible region of the current fragment;

and Q_{i-1} is the area of the uncovered and visible region of the next fragment on the list.

So far, packing fragments from only opaque polygons has been addressed. If a fragment is transparent, both the covered and uncovered regions of the fragment are influenced by fragments behind. Therefore, the color of both regions has to be found by the following recursive equation:

$$C_{average} = P_{total}A_i + T_iQ_i \qquad (6.5)$$

where

$$P_{total_i} = P_i(1.0 - transparency) + P_{i-1}(transparency)$$

$$P_{total_0} = P_0(1.0 - transparency) + P_{background}(transparency)$$

$$T_i = P_{total_{i-1}} A_{i-1} + T_{i-1} Q_{i-1}$$
$$T_0 = P_{background}$$

$$Q_i = Q_{i+1} - A_i$$

P_{total} is the final color of the current fragment;

$P_{total_{i-1}}$ is the final color of the next fragment on the list;

$P_{total_{total}}$ is the final color of the last fragment on the list;

Q_{i+1} is the area of the visible and uncovered region of the fragment on the list.

and

Q is the area of the visible region of the current fragment.

Notice that Q must be passed down from one level of recursion to the next and deeper level.

Implementation of the Fragment Packer

When the fragment packer is implemented, both the covered and uncovered regions are represented by bitmasks. The area of a mask can be found by simply counting the percentage of "on" bits.

In the algorithm presented above, two areas, A and Q, must be calculated. Let M_A be the bitmask defining the region A and M_Q be the bitmask define the region Q. The following relationships are used to find these bitmasks:

$$\begin{aligned} M_{A_i} &= M_{Q_{i+1}} \cap M_i \\ M_{Q_i} &= M_{Q_{i+1}} \cap \wedge M_i \end{aligned} \tag{6.6}$$

$$M_{init} = I$$

where I is a full mask.

6.4 Summary

After the ponts have been sampled and transformed into image space, a facsimile of the surface is reconstructed on the viewing plane. The surface is reconstructed one polygon at a time. Each polygon is filled by scan conversion. By modifying the scan conversion algorithm, its vertex values can be interpolated over the interior of the polygon. The polygon is then broken into pixel-sized fragments. These fragments are stored in the a-buffer. After all the polygons have been processed, the fragments in the a-buffer are packed to create the final image.

Chapter 7

Conclusion

The three dimensional rendering pipeline is well established, so are its difficulties. Although the pipeline is conceptually simple, implementations of it are notoriously inelegant, and the resulting code undecipherable. Given the limitations of a serial processor, these labyrinthine programs are inescapable. However, freed of these limitations in a parallel processing environment, one can construct a simple, modular, and easy to maintain rendering pipeline. In the following conclusion, the serial pipeline is reviewed and its major weaknesses noted. With these limitations in mind, a proposal for a modular parallel rendering pipeline is presented.

7.1 A Look Back

In the preceding chapters, the algorithms used to create a shaded image from a three dimensional polygonal database have been described. This pipeline of algorithms was divided into three phases: sampling, mapping,

and reconstruction.

7.1.1 A Quick Review

Sampling The initial sampling process converts each continuous surface into a manageable network of polygons. Each vertex in the network represents a sample point from the original surface. A second round of sampling occurs after the objects have been assembled into a coherent scene in eye space. With the surface normals in eye space, the vectors from the vertices to the lights in eye space, and the vectors from the eyepoint to the vertices in eye space, all the relevant information is in place to illuminate the surface. Having prepared the illumination data, the next step is to link the points in eye space to a position on the screen.

Mapping The mapping from eye space to the screen is performed by applying the viewing transformations to each vertex in eye space. After these transformations, each vertex contains its illumination parameters in eye space and a location in image space on the screen.[1]

Reconstruction The last phase of the pipeline reconstructs the transformed surface. To reconstruct the surface, the scan converter fills each polygon with illumination data interpolated from the vertices. Given this data, the illumination function calculates a color for each point inside the polygon. The resulting pixel-sized regions of the polygon are converted into

[1] If Gouraud shading is being used, the illumination calculation is performed at this point; the transformed point contains a color instead of the illuminated parameters.

pixel fragments and inserted into the a-buffer. Finally, after all the polygons have been processed, the fragment packer calculates the color value for each pixel on the screen. The image is then displayed.

7.1.2 Limitations

Over the years that the algorithms in the pipeline were developed, the algorithms were molded by the available hardware. In the beginning, the algorithms were designed for machines which, by today's standards, lacked both processing power and memory. As processing power and memory became cheaper, the antiquated algorithms gave way to calculationally complex algorithms such as ray tracing, and memory intensive algorithms such as the a-buffer. At the present, the constraints limiting the rendering algorithms come not so much from lack of memory or processing speed, but from the sequential nature of the von Neumann computer.

The effects of these limitations are especially apparent in the reconstruction phase of the rendering pipeline. When an image is reconstructed, the pixels are processed one at a time. For a screen resolution of 1,000 $\times$ 1,000 pixels, up to a million pixel calculations must be performed. So that each pixel does not have to be calculated from scratch, the reconstruction algorithms take advantage of the spatial coherence of the image to carry information from one pixel calculation to the next. Thus, at a fundamental level, the limitations of a sequential computer have have shaped the reconstruction algorithms.

Consequently, the algorithms are efficient, but the programs they pro-

duce are difficult to modify and to comprehend. Since information from one pixel is used as a starting point for the next pixel calculation, it is desirable to calculate as much information regarding one pixel as possible before moving on. Thus, scan conversion, interpolation, illumination, and anti-aliasing are rolled into one large function, the *tiler*. The tiler performs each of these functions on one pixel before moving to the next.

The constituents of the tiler are heavily interdependent. For instance, if one wishes to alter the interpolation function, one would also have to modify the scan conversion algorithm, and possibly the anti-aliasing function. The tiler is hard to understand because it is large; its constituent functions are not easily divided into manageable chunks of code. In summary, the drive to make the serial reconstruction of an image efficient resulted in programs that are both difficult to understand and difficult to modify.

With memory cheap and parallel machines already here, it is useful to reconsider the present rendering pipeline. The following is a proposal for a more modular rendering pipeline.

7.2 A Look Ahead

In order to overcome the limitations in the current pipeline, a new pipeline should fulfill two criteria: it should be easy to modify, and it should be adaptable to parallel processing. For the pipeline to be easily modified, each function in the rendering pipeline should exist as a distinct module. Each of these modules should handle a variety of data types; the same modules should be used by Gouruad and Phong shading, with or without

the a-buffer. If the pipeline is decomposed into modules which closely mirror the functions found in the pipeline, the resulting program will be easy to understand.

To make the pipeline adaptable to parallel processing, each function in the pipeline should apply to one of the three pervasive data structures: points, polygons, or pixels. Thus, the same function could be applied to a number of chunks of data simultaneously. An added benefit of a modular pipeline is that the pipeline can be converted easily from serial to parallel processing by replacing each serial module with an equivalent parallel module.

7.2.1 An Overview of the Proposed Pipeline

The proposed pipeline is divided into three steps: mapping, reconstruction, and picture generation. (It is assumed that the data coming into the pipeline has already been sampled in object space; it is already a database of polygons.) The mapping step is performed by the modeling matrices and the viewing transformations. Reconstruction corresponds to the interpolation and scan conversion steps of the traditional pipeline. And, image generation is similar to the a-buffer's fragment packer. Each of these steps can also be divided according to the type of data it processes. The mapping module transforms points; the reconstruction module fills and interpolates polygons; and the image generation module processes pixels.

Mapping Points The mapping module is the same as in the traditional rendering pipeline. Since the same pre-calculated matrix can be applied

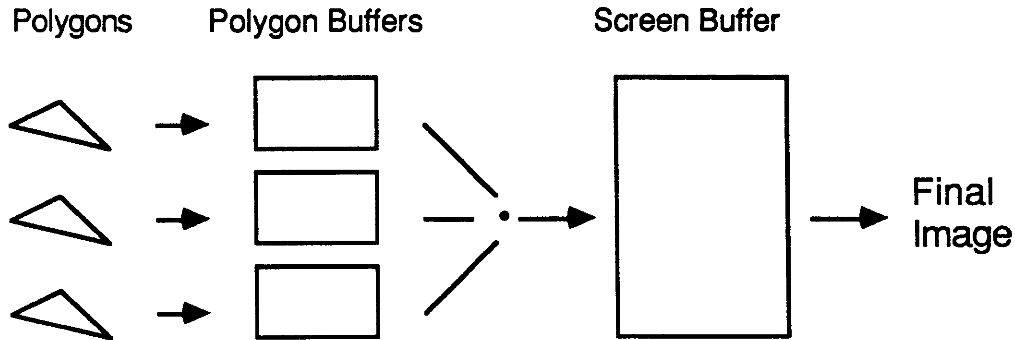


Figure 7.1: *An overview of the data structures.*

to all the points simultaneously, this module easily can be converted to a parallel process.

Reconstructing Polygons The reconstruction module is broken into three sub-modules. The first, the scan converter fills a polygon defined by a list of vertices with a grid of points. The next sub-module, the interpolator, attaches data interpolated from the vertices to each point inside of the polygon. The final module, the fragment creator, generates the appropriate pixel fragment for each pixel in the polygon.

Generating Pixel Colors The image generator converts the data stored at each pixel into an r, g, b value. For Gouraud shading, the image generator merely returns the existing r, g, b values. For Phong shading, the image generator performs the illumination calculation. And, for an a-buffer algorithm, the image generator packs each fragment list.

7.2.2 Data Structures

An overview of how the data structures relate to the pipeline can be found in figure 7.1.

Mapping Points

For mapping points from eyespace to display space, the data structures will depend on the details of the system. Conceptually, the point data structures used in the traditional pipeline will suffice.

Reconstructing Polygons: The Polygon Buffer

To reconstruct the polygons, a new data structure is introduced – the polygon buffer. The polygon buffer is a two dimensional array the size of the polygon's bounding box. Each element in the array corresponds to a pixel. The purpose of the polygon buffer initially is to store the vertices of a polygon. After the polygon is scan converted, the polygon buffer contains the filled polygon.

Each element in the buffer contains a z value used for hidden surface removal. The other data filling the polygon will vary depending on the rendering model being used. For Gouraud shading, each element in the buffer will contain a color; for Phong shading, each element will contain two vectors,[2] and the information necessary for illumination; and, for an

[2] The location of the lights is stored separately. Since the eye to surface vector was calculated in eye space, the vector is also the location of the sample in eye space. Therefore, the light vectors can be derived by subtracting the eye to surface vector from each light vector.

a-buffer algorithm, each element will contain a fragment.[3] Therefore, the elements in the buffer must be flexible enough to accommodate a number of different structures efficiently. The implementation of this feature varies according to the limitations of the programming language. A flag is included to indicate what type of data is stored.

Image Generation: The Screen Buffer

The screen buffer is similar to the a-buffer; it serves as a buffer to store data describing an image until that data is converted into r, g, b values. The screen buffer is the same size as the final image; each element in the buffer corresponds to a pixel. At each element, a z value is stored. As in the z-buffer, this z represents the distance to the closest surface from that pixel. The data loaded into the screenbuffer differs according to which rendering model is being used. If Gouraud shading is used, the buffer will contain the color value for the final image. For Phong shading, the buffer will contain the information necessary for an illumination calculation, that is, a surface normal, a vector from the eyepoint to the surface, the color of the surface, and the other illumination parameters. If the a-buffer algorithm is being used, each element contains a pixelstruct similar to the pixel struct described previously.

[3] The fragment will be a slightly expanded version of the fragment used in the a-buffer. The vectors used in Phong shading will be included in the fragment.

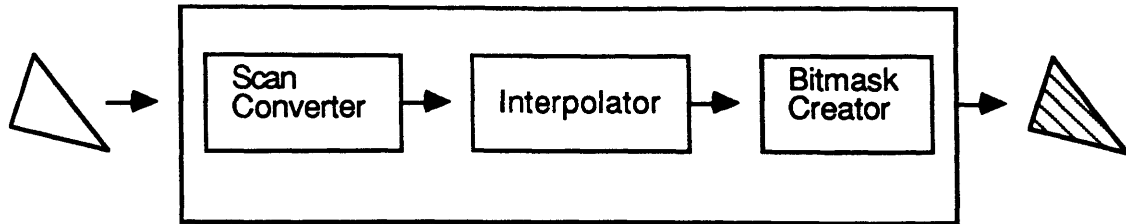


Figure 7.2: *The functions in the reconstructer.*

7.2.3 Functions

For each of the data structures described in the previous section, there is an associated function. These functions pass over the data in the buffers, modifying the buffers contents as it goes. The following section describes these functions.

Mapping Points

The modeling matrices and the viewing transformations are applied to each point.

Reconstructing Points: The Reconstructer

The reconstructer passes over a polygon buffer containing the vertices of a polygon and the associated illumination information and returns a buffer filled with interpolated values and pixel masks. Three separate modules are composited to form the reconstructer. (See fig. 7.2.) Depending on the type of data within the buffer, the reconstructer behaves differently. The

first step, then, is to query the flag in the buffer find what type of data is in it. With the data types known, the interpolator can proceed, using the correct functions for the data.

The first module, *the filler*, determines which points are inside the polyon, which are outside, and which lie on the edge. With a serial machine, this can be accomplished with the standard scan conversion algorithm. With a parallel machine, each pixel can be tested against the edges concurrently. Thus, a simple algorithm to decide if the pixel is in, out, or on the edge of a polygon can be used. Once the state of the each pixel is determined, each pixel is given an "in", "out," or "on the edge" value.

The next module, *the interpolator*, interpolates the values found at the vertices of the polygon. Only pixels that are either in the polygon or on the edge are given values. Once again, for a serial machine, the standard scan conversion algorithms used to interpolate the values can be used. In a parallel environment, an analytic function can be used to find interpolated values simultaneously for each pixel in the polygon. (See [14] for and example.)

Finally, if the image is to be anti-aliased, the final function, *the mask creator*, is applied to the polygon buffer. Using techniques described in by Fiume[15], the creator makes a pixel mask for each pixel in the buffer.

Generating an Image: The Image Function

As each polygon passes through the reconstructer, it is loaded into the screenbuffer. There the data is kept until all of the polygons have been loaded. Then, the image function is passed over the loaded screen buffer.

The image function transforms the data stored at each element in the screenbuffer into r, g, b values. Like the reconstructer, the image function conforms to the data in the buffer; it behaves differently depending on the contents of an element. Before it acts on each element in the buffer, the image function checks the element's flag to determine the data type found in the element. It then acts upon the data accordingly.

For Gouraud shading, the image function does not act; it leaves the existing r, g, b values in the buffer. For Phong shading, the image function performs an illumination calculation using the values stored in the buffer and a list of lights found outside of the buffer as parameters. Note that by changing the list of lights while keeping the contents of the buffer constant, it is possible to test different light positions and colors quickly. If the *a*-buffer algorithm is being used, the image function acts like the fragment packer described earlier. Since each pixel is processed independently, the image function can be applied to each pixel concurrently.

7.2.4 Summary

The pipeline (see fig7.3) proposed above transforms a polygonal object in object space into a shaded image on the screen. The first step of the pipeline is to transform all the vertices into image space. Once all of the vertices have been transformed, each polygon is loaded into its own polygon buffer. In the polygon buffer, the filler function passes over the buffer, changing the wireframe polygons into solid surfaces. After the filler is done, the interpolator assigns values interpolated from the vertices of the polygon.

The Pipeline

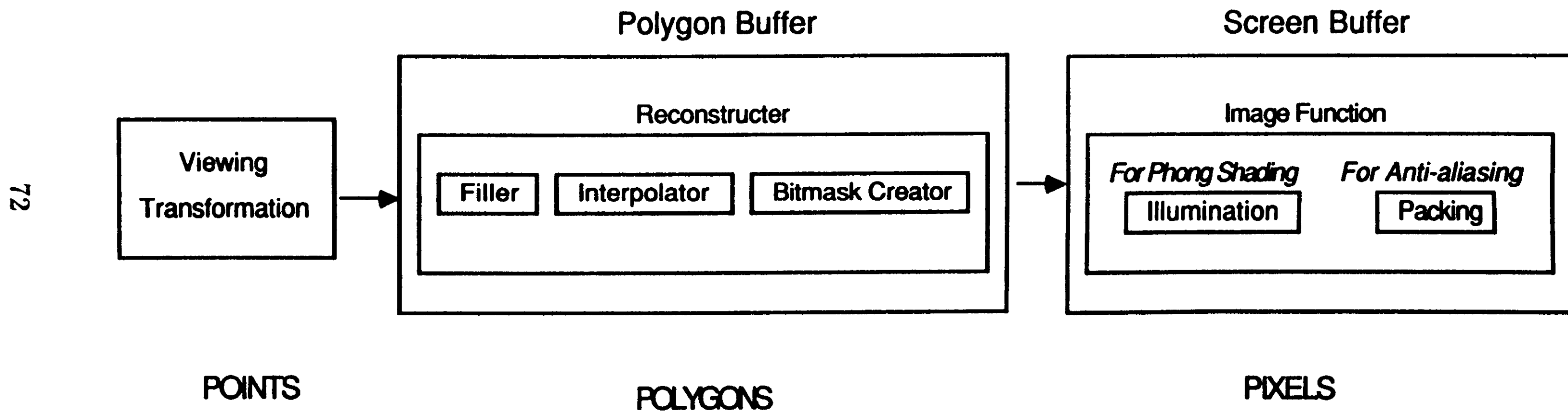


Figure 7.3: *The proposed rendering pipeline.*

The last step in the polygon buffer is to convert each pixel-sized region within the polygon into a fragment. Passing over the polygon buffer one at a time, each of the above modules can process the points in the buffer concurrently. Furthermore, all the polygons of a scene can be processed simultaneously (barring memory and processor limitations.)

After the interpolator is through with the polygon, the polygon is loaded into the screenbuffer. The points in a polygon can be loaded concurrently, but because of the sequential nature of the z comparisons needed for hidden surface elimination, the polygons must be loaded one at a time. However, as Crow[12] points out, a certain degree of parallelism can be achieved in this step by subdividing the screen into areas that can be rendered independently. Once all the polygons are loaded into the screenbuffer, the image function packs the information stored at each element into an r, g, b value. Since each pixel is independent from the next, the image function can work simultaneously on all of the pixels.

The pipeline described above is arranged so that one module at a time acts on one type of data. The points are processed in one batch by the viewing transformations; the polygons are all processed by the reconstructer; and finally, all the pixels are processed together by the image function. Since each module contains only one function, the pipeline is easy to modify; to change a function, one has to change only one black box. And finally, since each function acts on a large amount of similar data, the pipeline is well suited for parallel processing.

7.3 Conclusion

Having had its foundations laid in the 1960's and early 1970's, computer graphics has been steadily building up from its origins. The fundamental techniques underlying both polygonal rendering and ray tracing were established early on in the history of computer graphics. In 1984, with the introduction of distributed ray tracing and the *a*-buffer, it appeared that the details were also in place.

Parallel processing, however, has been rocking the foundations. No longer are the tried and true methods sufficient; as is exemplified by the scan conversion algorithm, much of the effort in the past has been defined by the limitations of a serial computer. In the future, many of the fundamental algorithms will have to be re-thought. As the golden age of serial computer graphics is coming to a close, a new era dominated by parallel processing is beginning.

Appendix A

Rendermatic

Rendermatic is a set of library routines written in C which render three-dimensional polygonal objects. At its core, the package supports wireframe rendering, Gouraud and Phong shading, multiple lights sources, z-buffer hidden surface elimination, and an a-buffer for anti-aliasing, transparency, and picture compositing. These routines are not meant to be a single program complete with a user-interface and transparent functionality. Instead, these routines form a package of subroutines called from a C program. Thus, the package serves as the basis for a wide variety of applications.

Although the system was developed on a Digital Equipment Corp. VAX 785 accompanied by a Ramtek 9300 framebuffer, the code was written in a device independent manor. At present, while running on the VAX, Rendermatic supports a Rastertek framebuffer. Additionally, Rendermatic runs on the Hewlett-Packard Model 3000 workstation.

After Rendermatic reads the object data in from a file, the data is converted to Rendermatic's own internal data structures. These data struc-

tures contain a list of attributes describing how their data should be rendered. These data structures are sent through the rendering pipeline and rendered according to these attributes. If no attributes have been set, Rendermatic provides default values. One need only specify an attribute if the default is insufficient.

The following section describes the data-structures and their associated functions. Afterwards, the rendering functions are presented and discussed.

Appendix B

The Data Structures

Rendermatic organizes the data describing a surface into four major levels of detail: points, polygons, polyhedra, and objects. Points represent samples taken from the ideal, continuous surface at the outset of the rendering pipeline. These samples, coupled with information regarding their connectivity, pull together to form a polygon. One polygon describes a region defined by a set of connected sample points. When all the polygons of a surface are grouped together, they form a polyhedron. A polyhedron represents a surface in one of the succession of spaces visitied during the viewing transformations. There is one polyhedron for each of these spaces. Each of the polyhedra describe the same surface. An object serves as the umbrella covering each of the polyhedra descendant from one surface.

At each of these levels of detail, Rendermatic divides the data into two classes: information regarding the basic shape of surface, and information describing the quirks of a specific object. This division allows the data describing numerous instances of the same class of object to be stored effi-

ciently. Each instance of an object can share a large amount of data with other objects from that class. For example, each maple leaf in a forest has roughly the same shape. This shape can be generated by modifying a generic maple leaf template. Thus, the data points describing the template need to be stored only once. While the general shape of a surface is common to each instance, the color, luster, and other characteristics caused by illumination tend to vary from one instance to another. Additionally, the scale, orientation, and location of each instance can differ for each object. These data describing the quirks of an individual surface are stored separately for each instance. By separating the information describing the generic shape of the surface from rest of the surface data, Rendermatic efficiently shares the shape information, yet allows each object to retain its own characteristics.

Once an object starts through the viewing transformations, the information describing its particular quirks does not change, only the points describing its shape do. Therefore, as the object moves from world space to image space, only the points have to be copied; the other information can be shared throughout the viewing transformations. During the viewing transformations, the split between the location information and the object's attributes allows the object to exist simultaneously in the different space encountered during the viewing transformations while sharing the same attribute information.

The data structures are shown in block diagram form in figure B. The OBJECT, POLYHEDRON, POLYGON, and POINT structures reference the generic shape data shared by each instance of the object. The

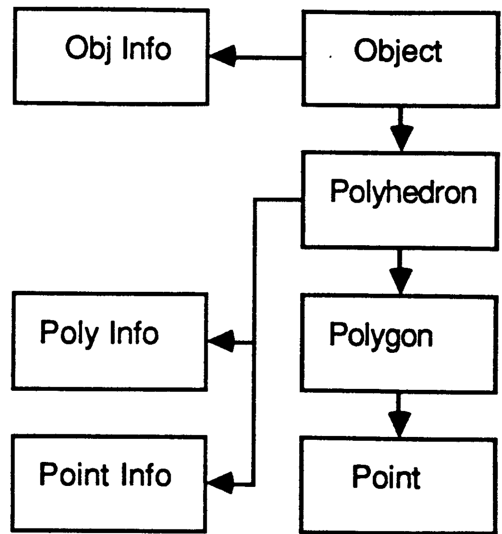


Figure B.1: *The object data structures.*

OBJ_INFO, POLY_INFO, and PT_INFO structures contain the information specific to one instance of the object.

B.1 The Object Structures

The structure for an object is the following:

```
typedef struct
{       POLYHEDRON *wph
        POLYHEDRON *eph
        POLYHEDRON *cph
        POLYHEDRON *dph
        POLYHEDRON *iph
        OBJ_INFO     *info
}OBJECT;
```

The object is the broadest representation of a polygonal surface. Each object represents one instance of a surface. The major structures within an object are one polyhedron for each space visited during the viewing transformations and a structure, the OBJ_INFO structure, containing qualities specific to this instance of the surface. Unless otherwise specified, the data in the OBJ_INFO structure is inherited to the more detailed structures constituting the object.

B.1.1 Object Information

The OBJ_INFO structure looks like the following:

```
ypedef struct
{       Matrix      M
        Color       *color
```

```
        OPTICAL         *optical
        Light           *spots
        char            *texture_map
        char            *detail
        int             flag
} OBJ_INFO;
```

The following is a description of each of the fields in the info structure, and the functions which manipulate them.

M M is the modelling matrix which defines the position, orientation, and scale of the surface in worldspace.

```
    RotateObject(name, rotation_matrix)
        string *name
        Matrix rotation_matrix
    ScaleObject(name, scale_matrix)
        string *name
        Matrix scale_matrix
    TranslateObject(name, translation_matrix)
        string *name
        Matrix translation_matrix
    obj_scale(obj, sx,sy,sz)
        OBJECT *obj
        WorldType sx,sy,sz
    obj_rotate(obj, rx,ry,rz)
        OBJECT *obj
```

```
        WorldType rx,ry,rz
obj_translate(obj, dx,dy,dz)
        OBJECT *obj
        WorldType dx,dy,dz
obj_reorigin(obj)
        This function sets the modelling matrix to the identity matrix.
        OBJECT *obj
Matrix *obj_get_get_matrix(obj)
        OBJECT *obj
```

Color Color is a pointer to a color structure containing the r,g,b values for the entire object. Unless the color of a polygon, or a color of a point is explicitly set, every polygon and every point of a surface will have this color.

```
obj_set_color(obj,r,g,b)
        OBJECT *obj
        ColorType r,g,b
obj_get_color(obj,r,g,b)
        OBJECT *obj
        Colortype *r,*g,*b
```

Optical Optical is a pointer to the optical structure containing the light information for the entire object. Like the color structure, the optical will be inheritied by each polygon and point unless the optical information for

a point or a polygon is set explicitly.

obj_set_optical(obj, optical)

OBJECT *obj

OPTICAL *optical

obj_get_optical(obj, optical)

OBJECT *obj

OPTICAL *optical

obj_set_shading_params(obj, diffuse, specular, exponent, ambient)

OBJECT *obj

WorldType diffuse, specular, exponent, ambient

obj_get_shading_params(obj, diffuse, specular, exponent, ambient)

OBJECT *obj

WorldType *diffuse, *specular, *exponent, *ambient

obj_set_fudge(obj, fudge)

Fudge is the fudge factor used in most simple illumination models.

OBJECT *obj

WorldType fudge

obj_get_fudge(obj, fudge)

OBJECT *obj

WorldType *obj

obj_set_transparency(obj, transparency)

OBJECT *obj

WorldType transparency

obj_get_transparency(obj, transparency)

OBJECT *obj

WorldType *transparency

Spotlight Spotlight is the list of lights which illuminate this object. This list can be a subset of all the lights illuminating a scene.

```
obj_add_light(obj, light)
        OBJECT *obj
        Light *light
obj_del_light(obj, light)
        OBJECT *obj
        Light *light
```

Texture Map Texture map is a slot reserved to point to the file containing a texture map for the object.

Detail Detail is a file pointer pointing to the ".det" file describing object data in Ohio State format.

Flag Flag contains bits used to indicate which object attributes are on or off.

```
Smooth or Faceted ?
        obj_set_faceted(obj)
                OBJECT *obj
        int obj_is_faceted(obj)
                Returns "True" if the object is faceted.
                OBJECT *obj
```

obj_set_smooth(obj)

OBJECT *obj

int obj_is_smooth(obj)

OBJECT *obj

Gouraud, or Phong Shaded ?

obj_set_gouraud(obj)

OBJECT *obj

obj_set_phong(obj)

OBJECT *obj

int obj_get_shading_model(obj)

OBJECT *obj

Should the object be backface culled ?

obj_set_backfacecull(obj)

OBJECT *obj

obj_unset_backfacecull(obj)

OBJECT *obj

int obj_is_cullable(obj)

OBJECT *obj

Should the Object be posted on the list to be rendered?

obj_post(obj)

OBJECT *obj

obj_unpost(obj)

OBJECT *obj

int obj_isposted(obj)

OBJECT *obj

Return the flag

int obj_get_flag(obj)

OBJECT *obj

Name Name is a string containing the name of the object.

obj_set_name(obj, name)

OBJECT *obg

string *name

obj_get_name(obj)

OBJECT *obj

OBJECT *obj_name2ptr(name)

string *name

Next Next is an object pointer pointing to the next object on the list. These functions are used to cycle through lists of objects.

int obj_is_tail(obj)

Returns "True" if the object is at the end of the list.

OBJECT *obj

*OBJECT obj_get_head()

Returns returns the head of the object list.

B.1.2 A Polyhedron For Each Space

Each of the polyhedra within the object represents the polyhedron in each of the spaces visited during the viewing transformations.

wph Wph slot is reserved for a polyhedron in world space.

```
obj_get_wph(obj)
        OBJECT *obj
```

eph Eph is a slot reserved for a polyhedron in eyespace.

```
obj_get_eph(obj)
        OBJECT *obj
```

cph Cph is slot reserved for a polyhedron in clipping space.

```
obj_get_cph(obj)
        OBJECT *obj
```

dph Dph is a slot reserved for a polyhedron in display space.

```
obj_get_dph(obj)
        OBJECT *obj
```

iph Iph is a slot reserved for a polyhedron in image space.

```
obj_get_iph(obj)
        OBJECT *obj
```

B.1.3 Functions that apply to the whole object

OBJECT *obj_alloc()

obj_free(obj)

OBJECT *obj

obj_init(obj)

Initializes the object with all the default values.

obj_cull(obj)

Backface culls an object.

OBJECT *obj

B.1.4 Transforming the Object

obj_wsp2esp_wireframe(obj)

Transforms an object from world space to eyespace.

This routine is used if the object will be a wireframe drawing.

OBJECT *obj

obj_wsp2esp_zb(obj)

OBJECT *obj

Transforms an object from world space to eyespace.

This routine is used if smooth shading is being used.

obj_esp2csp(obj)

OBJECT *obj

Transforms an object from eyespace to clipping space

(the canonical viewing frustrum.)

obj_csp2dsp(obj)

OBJECT *obj

Transforms the object to floating point display space.

obj_dsp2isp(obj)

OBJECT *obj

Transformas the object to integer image space.

B.2 The Polyhedron Structure

A polyhedron describes an entire surface in one of the spaces encountered during the viewing transformation. The information in the polyhedron is divided into shape information and illumination information. The shape of the polygon is described by an array of polygons, an array points, and an array of edges. For the polygon array and the point array there is a corresponding array describing the characteristics of the surface when it is illuminated. This information is specified by an array of point information structures, and an array of polygon structures.

The elements within the point and point information arrays correspond exactly. Similarly, the elements within the polygon and polygon information arrays correspond exactly. For example, the information in the fifth cell in the polygon array describes the same polygon as the information in the fifth cell of the poly_info array. Given the index of a point, one can gather all the information relevant to the point by using this index to access the array of points and the array of point information. Similarly, all the information relevent to a polygon can be found by accessing the polygon array and the polygon information array with the polygon's index.

This correspondence holds true not only for the arrays within one polyhedron, but also for the point, polygon, and edge arrays within *all* polyhedra found in an object. The ith point refers to the same point in the world space polyhedron, the eyspace polyhedron, the clipped space polyhedron, the display polyhedron, and the image space polyhedron. Thus, one index can access a specific point in all of the polyhedra within an object. The

same holds true for the pt_info arrays, the polygon arrays, the poly_info arrays, and the edge arrays.

The following is the polyhedron structure:

```
typedef struct
{         short           Nopts
          short           Nopolys
          short           Noedges
          POINT           *pts
          PT_INFO         *pt_info
          POLYGON         *polys
          POLY_INFO       *poly_info
          EDGE            *edges
          Vector          bbox[2]
}POLYHEDRON
```

The following is a description of the fields in the POLYHEDRON data structure, and their associated funtions.

Nopts Nopts contains the number of points in the polyhedron.

ph_set_nopts(ph)

POLYHEDRON *ph

int ph_get_nopts(ph)

Returns the number of points in a polyhedron.

POLYHEDRON *ph

Nopolys Nopolys contains the number of polygons in the polyhedron.

int ph_get_nopolys(ph)

Returns the number of polygons in a polyhedron.

POLYHEDRON *ph

ph_set_nopolys(ph)

POLYHEDRON *ph

Noedges Noedges contains the number of edges in the polyhedron.

int ph_get_noedges(ph)

Returns the number of edges in the polyhedron.

POLYHEDRON *ph

ph_set_noedges(ph)

POLYHEDRON *ph

pts Point is a dynamically allocated array of all the POINT structure for the polyhedron. The index used to a specify a point in the pts array will specify the same point in the pt_info array.

POINT *ph_get_pts(ph)

Returns a pointer to the array of points in the polyhedron.

POLYHEDRON *ph

pt_info Pt_info is a dynamically allocated array containing all the PT_INFO structures for the polyhedron. The index used to specify a point in the pt_info array will specify the same point in the pts array.

```
PT_INFO *ph_get_pt_info(ph)
        POLYHEDRON *ph
        Returns a pointer to the array of point information
        structures in the polyhedron.
```

polys Polys is a dynamically allocated array containing all the POLYGON structures for the polyhedron. The index used to specify a polygon in the polys array will specify the same polygon in the poly_info array.

```
POLYGON *ph_get_polys(ph)
        POLYHEDRON *ph
        Returns a pointer to the array of polygon structures in
        the polyhedron.
```

poly_info Poly_info is a dynamically allocated array containing all the POLY_INFO structures for the polyhedron. The index to used specify a polygon in the poly_info array will specify the same polygon in the polys array.

```
POLY_INFO *ph_get_poly_info(ph)
        POLYHEDRON *ph
        Returns a pointer to the array of polygon information structures in
        the polyhedron.
```

edges Edges is a dynamically allocated array containing all the EDGE structures for a polyhedron.

EDGE *ph_get_edges(ph)

Returns a pointer to the array of edges int the polyhedron.

POLYHEDRON *ph

bbox Bbox is an array of two coordinate triplets (x,y,z). The first coordinate contains the minimum x,y, and z value of the polyhedron. The second coordinate contains the maximum x,y, and z value of the polyhedron.

ph_set_bbox(ph, xmin,ymin,zmin,zmax,ymax,zmax)

POLYHEDRON *ph

WorldType xmin, ymin, zmin, zmax, ymax,zmax

ph_get_bbox(ph,xmin,ymin,zmin,xmax,ymax,zmax)

POLYHEDRON *ph

WorldType *xmin,*ymin,*zmin,*xmax,*ymax,*zmax

B.2.1 Allocating and De-allocating Polyhedra

POLYHEDRON *ph_alloc()

ph_free(ph)

POLYHEDRON *ph

B.2.2 Transforming the Polyhedra

ph_wsp2esp(ph, vm, mm, flag)

POLYHEDRON *ph

Matrix vm (The view matrix)

Matrix mm (The modelling matrix)

int flag (The type of perpective,ORTHOGONAL or PERSPECTIVE.)

ph_esp2csp(ph,nr, perspective)

POLYHEDRON *ph

Matrix nr (The right normalization matrix.)

ph_csp2dsp(ph, dm)

POLYHEDRON *ph

Matrix dm (The display matrix.)

ph_dsp2isp(ph)

POLYHEDRON *ph

B.2.3 General Polyhedron Functions

ph_cull(ph)

POLYHEDRON *ph

ph_get_centroid(ph)

POLYHEDRON *ph

ph_findsilhouette(ph)

Marks the edges on the silhouette of the polyhedron.

POLYHEDRON *ph

B.2.4 Accessing Polyhedra from the Object Level

```
POLYHEDRON *obj_get_wph(obj)
        OBJECT *obj
POLYHEDRON *obj_get_eph(obj)
        OBJECT *obj
POLYHEDRON *obj_get_cph(obj)
        OBJECT *obj
POLYHEDRON *obj_get_iph(obj)
        OBJECT *obj
```

B.3 The Polygon Structure

The polygon structure defines the topology of a polygon. The information regarding the illumination and rendering attributes of a polygon can be found in the polygon information structures. The actual points and edges are **not** contained in the polygon structure. Instead, the indices into the point and edge lists are stored.

The POLYGON data structure:

```
typedef struct
{       short       Noindices
        short       *pt_indices
        short       *edge_indices
        Vector      normal
}POLYGON;
```

The following is a description of the fields found in the POLYGON structure:

Noindices Noindices is the number of vertices and edges in the polygon.

```
poly_set_noindices(poly, numb)
        POLYGON *poly
        int numb
int poly_get_noindices(poly)
        POLYGON *poly
```

Pt_indices Pt_indices (point indices) is a dynamically allocated array filled with the indices of the constituent points of the polygon. An index from the pt_indices array can be used to access pts array and pt_info array found in a polyhedron to obtain information about a specific point.

poly_set_index(poly, which, index)

POLYGON *poly

int which (This is which vertex is being set)

int index (This is the point index that the vertex is being set to.)

int poly_get_pt_index(poly, which)

If "which" is 3, then this routine would return the point index of the third vertex.

POLYGON *poly

int which

Edge_Indices Edge_indices (edge indices) is a dynamically allocated array filled with the indices of the edges which form the polygon. Information about the edge can be found by using an index from the edge_indices to access the edge in the edge array found in a polyhedron.

poly_get_edge_index(poly, which)

POLYGON *poly

int which

Normal Normal is a vector representing the surface normal of the polygon.

poly_set_normal(poly, x, y, z)

POLYGON *poly

WorldType x.y,z

poly_get_normal(poly, x, y, z)

POLYGON *poly

WorldType *x, *y, *z

B.3.1 Allocating and De-allocating Polygons

*POLYGON poly_alloc(numb)

Returns the address of the top of an array of polygons.

int numb

poly_free(polys, numb)

Frees a block of polygons.

POLYGON *polys

int numb

B.3.2 Polygon Extent

poly_find_extent(poly, pts, xmin, ymin, zmin, xmax, ymax, zmax)

POLYGON *poly

POINT *pts (This is the point list into which the polygon vertices index)

B.4 The Polygon Information Structure

The polygon information structure (POLY_INFO) contains all the information describing how a region within a polygon interacts with light and how it should be rendered. If any of the elements in the POLY_INFO structure are empty, when queried, the values returned will be inherited from the OBJ_INFO data structure.

The POLY_INFO structure:

```
typedef struct
{         Color          *color
          OPTICAL        *optical
          Spotlight      *spots
}POLY_INFO;
```

The following is a description of the fields within the POLY_INFO data structure:

Color Color is a pointer to a color structure of r,g,b values describing the color of the region within the polygon. If the color pointer is NIL, then the color stored in OBJ_INFO will be returned as the color of the polygon.

```
poly_info_set_color(r,g,b)
        ColorType r, g, b
poly_info_get_color(r,g,b)
        ColorType *r,*g,*b
```

Optical Optical is a pointer to the optical properties of the polygonal surface. If the optical pointer is NIL, then the optical structure is inherited from the OBJ_INFO structure.

poly_info_set_optical(info, opt)
    POLY_INFO *info
    OPTICAL *opt
OPTICAL *poly_info_get_optical(info)
    POLY_INFO *info
poly_info_set_shading_params(info,diffuse,specular,exponent, ambient)
    POLY_INFO
    WorldType diffuse, specular, exponent, ambient

Spotlights Spotlights is a list of lights that illuminate the polygon. This list can be a subset of the list of lights that are illuminating the whole scene.

poly_info_add_light(info, light)
    POLY_INFO *info
    Light *light
poly_info_rm_light(info, light)
    POLY_INFO *info
    Light *light

Flag Flag is an integer describing rendering attributes of the polygon.

poly_info_set_flag(info, flag)

POLY_INFO *info

int flag

int poly_info_get_flag(info)

POLY_INFO *info

Faceted polygon, or smooth ?

poly_info_set_faceted(info)

POLY_INFO *info

int poly_info_is_faceted(info)

Returns "True" if the polygon is faceted.

POLY_INFO *info

poly_info_set_smooth(info)

POLY_INFO *info

poly_info_is_smooth(info)

Returns "True" if the polygon represents a curved surface.

POLY_INFO *info

Gouraud or Phong shading ?

poly_info_set_gouraud(info)

POLY_INFO *info

poly_info_set_phong(info)

POLY_INFO *info

int poly_info_get_shading_model(info)

POLY_INFO *info

Backface Culling the polygon.

poly_info_cull(poly, info, ph)

"De-activates" a polygon if it is facing backwards.

POLYGON *poly (The specific polygon)

POLY_INFO *info (The corresponding

POLYHEDRON *ph (The parent polyhedron.)

int poly_info_isculled(info)

Returns "True" if the the polygon has been culled.

POLY_INFO *info

B.4.1 Allocating and De-allocating Poly-Info

POLY_INFO *poly_info_alloc()

poly_info_free(info)

POLY_INFO *info

B.5 The Point Structure

A point represents one sample point of a continuous surface. The POINT data structure contains the shape information at one point on the surface. The information describing the other attributes of the surface are found in the PT_INFO structure.

The information found in the POINT structure:

```
typedef struct
{       Vector      point
        Vector      normal
}POINT;
```

The following is a description of each of the fields within the POINT structure:

Point Point is a triplet (x,y,z) containing the coordinate of the sample point.

```
    pt_set_point(pt, x, y, z)
            POINT *pt
            WorldType x,y,z
    pt_get_point(pt, x, y, z)
            POINT *pt
            WorldType *x, *y, *z
```

Normal Normal is a vector containing the direction of the surface normal at the sample point. If the parent polygon is faceted, then the normal the point returns is the surface normal of the polygon.

```
pt_set_normal(pt, x, y, z)
        POINT *pt
        WorldType x, y, z
pt_get_normal(pt, x, y, z)
        POINT *pt
        WorldType *x, *y, *z
```

B.5.1 Allocating and De-allocating POINTS

```
POINT *pt_alloc(numb)
        int numb
pt_free(pt)
        POINT *pt
```

B.6 The Point Information Information Structure

The PT_INFO structure describes how one sample point on a surface interacts with light. Information detailing the shape of the object at this point can be found in the POINT structure.

The following information is found in the PT_INFO structure:

```
typedef struct
{       Color          *color
        OPTICAL        *optical
        SpotLight      *spots
        int            flag
}PT_INFO;
```

The following is a description of the fields within the PT_INFO structure:

Color Color is a pointer to a structure containing the r,g.b colors of the surface at one point. If the color pointer is NIL, then the color value returned is inherited from the parent POLY_INFO structure. (Note that if the color pointer in the POLY_INFO structure is NIL, then POLY_INFO returns the color from the parent OBJ_INFO structure.)

```
pt_info_set_color(info, r, g, b)
        PT_INFO *info
        WorldType r, g, b
```

pt_info_get_color(info, r, g, b)

PT_INFO *info

WorldType *r, *g, *b

Optical Optical is a pointer to a structure containing the optical properties of the surface at this point. If the optical pointer is NIL, then the optical structure returned is inherited from the parent POLY_INFO structure. (Note that if the optical pointer in the POLY_INFO structure is NIL, then POLY_INFO returns the optical structure from the OBJ_INFO.)

pt_info_set_optical(info, optical)

PT_INFO *info

OPTICAL *optical

*OPTICAL pt_info_get_optical(info)

PT_INFO *info

Flag

pt_info_local_max(info)

Marks a point as a local maximum of a polygon.

PT_INFO *info

int pt_info_is_local_max(info)

Returns "True" if the point is a local maximum.

PT_INFO *info

B.6.1 Allocating and De-allocating PT_INFO

```
PT_INFO *pt_info_alloc(numb)
        int numb
pt_info_free(numb)
        int numb
```

B.7 Optical

The optical structure contains the information describing how light interacts with a surface. The optical structure is found at the point, polygon, and object level of a surface. The optical properties are inherited from the object to the polygon, then from the polygon to the point.

The OPTICAL structure:

```
typedef struct
{       ColorType     diffuse
        ColorType     specular
        ColorType     exponent
        ColorType     ambient
        ColorType     transparency
        ColorType     fudge
}OPTICAL;
```

The follow is description of the functions and quantities found in the OPTICAL structure:

Diffuse This real number is the diffuse component used in the lighting calculation.

```
opt_set_diffuse(opt)
        OPTICAL *opt
WorldType opt_get_diffuse(opt)
        OPTICAL *opt
```

Specular This real number is the specular coefficient used in the lighting calculation.

```
opt_set_specular(opt)
        OPTICAL *opt
WorldType opt_get_specular(opt)
        OPTICAL *opt
```

Exponent This real number is the exponent used in Phong illumination to define the spread of the specular highlight.

```
opt_set_exponent(opt)
        OPTICAL *opt
WorldType opt_get_exponent(opt)
        OPTICAL *opt
```

Ambient This real number is the ambient coefficient used in the illumination calculations.

```
opt_set_ambient(opt)
        OPTICAL *opt
WorldType opt_get_ambient(opt)
        OPTICAL *opt
```

Transparency This real number between 0.0 and 1.0 determines the level of transparency of the object. If the object is transparent, the value is 1.0; if the object is opaque, the value is 0.0.

```
opt_set_transparency(opt)
        OPTICAL *opt
WorldType opt_get_transparency(opt)
        OPTICAL *opt
```

Fudge This real number is the fudge factor used to attenuate the drop off in illumination intensity.

```
opt_set_fudge(opt)
        OPTICAL *opt
WorldType opt_get_fudge(opt)
        OPTICAL *opt
```

B.7.1 Allocating and De-Allocating Optical Structures

```
OPTICAL *opt_alloc()
opt_free(opt)
        OPTICAL *opt
```

```
opt_inherit(pt_info, poly_info, obj)
```

This routine attempts to return the optical at the point level first. If the structure is NULL, then the routine

looks to the polygon level. If the polygon has not been assigned an optical structure, the routine returns the object's optical structure.

PT_INFO *pt_info
POLY_INFO *poly_info
OBJECT *obj

Appendix C

Rendermatic Rendering

The first step in the rendering pipeline is to load the Rendermatic data structures with objects. Objects are loaded into the Rendermatic data structures by *InstanceObject*. All the surface normal calculations, vertex normal calculations, and bounding box calculations are done in *InstanceObject*. After the data structure has been loaded, *InstanceObject* inserts the new object into the object list.

Rendering an object in Rendermatic is simple; it involves one of two commands, either *Wireframe()* or *Rendermatic()*. *Wireframe* renders a wireframe representation of the objects; *Rendermatic* renders a smooth-shaded representation. The modelling transformations, viewing transformations, and image reconstruction all occur within each of these commands. The two rendering commands send one object at a time through the pipeline. When an object reaches image space, it is reconstructed and written to either a file or a framebuffer.

The viewing transformations are performed by the functions *obj_wsp2esp*,

obj_esp2csp, *obj_csp2dsp*, and finally, *obj_dsp2isp*. After and object makes it to image space, *zbuff_object* takes over. *Zbuff_object* splits the object into polygons, sending one polygon at a time to be rendered by *zbuff_poly*. Inside of *zbuff_poly* the polygon is loaded into the edge table (ET) by *ET_load_poly*. The resulting ET is scan converted, interpolated, and rendered by *zbuff_ET*. For Gouraud shading, illumination is performed in *ET_load_poly*; for Phong shading, illumination is performed in *zbuff_ET*.

The special rendering options are specified by the attributes of the objects, not by the rendering functions. In short, the objects are loaded into the data structures; then one of the rendering commands, either *WireFrame* or *Rendermatic*, sets the rendering process in motion.

C..2 Wireframe Rendering

WireFrame() Wireframe draws a all of the objects posted on the object list in a wireframe representaton.

The following functions set of wireframe attributes for the scene.

WireFrameQuickOn() When WireFrameQuick is set, only the bounding boxes of each object are drawn.

WireFrameSilhouetteOn(),WireframeSilhouetteOff() When WireframeSilhoueette is on, only the silhouettes of the objects are drawn. WireframeSilhouetteOn()

C..3 Rendering Shaded Images

Rendermatic() Rendermatic renders all the posted objects on the object list. Depending on the state of the object, it will render each object with Gouraud shading or Phong shading.

PostObject(obj), UnPostObject(obj) If an object is posted, then the object will be rendered when *Rendermatic* or *WireFrame* is called.

Render2File(image,z) This function sets *Rendermatic()* to render the image to a file. Image is the name of the image file, and z is the name of the file to store the z-buffer.

RenderPrintf() This is similar to a "verbose" switch. This command causes *Rendermatic* to write a message when it finishes rendering an object.

C.1 The A-Buffer

The following as a listing of the important data structures and functions used for the a-buffer. They follow closely to the structures listed in the paper [5] by Carpenter.

C.1.1 The Data Structures

Pixel Masks A pixel mask is two 32 bit words.

typedef struct byte pixelmask[8]

Fragment The fragment structure is similar to the fragment in Carpenter's paper.

```
typedef struct _fragmnt
{       struct _fragmnt  *next
        Color            color
        WorldType        opacity
        WorldType        area
        OBJECT           *object_ptr
        pixelmask        m
        WorldType        zmax, zmin
}fragment;
```

SimplePixel The SimplePixel structure is similar to Carpenter's structure.

```
typedef struct _fragmnt
{   Color               color
    WorldType           coverage
}simplepixel;
```

PixelStruct

```
typedef struct _fragmnt
{
    WorldType           z
    union
```

```
    {
                    simplepixel simple
                    fragment *fragptr
    }pixel;
}pixelstruct;
```

PixelMaskInfo The PixelMaskInfo stucture is unique to Rendermatic. This is the buffer used to store pixel masks and their related information during scan conversion. The buffer is explained more fully in Chapter 5.

```
typedef struct
{
    WorldType       zmin,zmax
    WorldType       area
    pixelmask m
}PixelMaskInfo;
```

C.1.2 Functions

Loading the a-buffer

prepare_abuff_scanline(screen_y, **AET, pmask_line)**

ScreenType	screen_y (This is the current scan line.)
ET_Element	*AET (This is the current active edge list.)

PixelMaskInfo pmask_line[] (This is the mask buffer.)

Prepare_abuff_scanline calculates the area, pixel mask, zmin, and zmax for each pixel on a scanline. This information is calculated before the scanline is filled, and accessed when the scanline regions are being filled.

edge_to_pixel_masks(cur_x, cur_y, cur_z, max_y, dxdy, dzdy, edge_kind, mask_line)

ScreenType	cur_x, cur_y
WorldType	cur_z
ScreenType	max_y (This is the maximum value of the edge).
int	edge_kind (Indicates whether the edge is beginning or end of a scanline region).
WorldType	dxdy,dzdy (Incrments with respect to y)
PixelMaskInfo	pmask_line[] (This is the mask buffer.)

Edge_to_pixel_masks loads the information from one edge into the mask buffer. This routine is called by prepare_abuff_scanline.

create_edge_mask(x,y,edge_y_max, incr, mask)

ScreenType	x,y (Where the scanline enters the pixel.)
ScreenType	edge_y_max (The maximum y value of the current edge.)
ScreenType	incr (The increment in x with respect to y.)
pixelmask	mask (The mask returned from this function.)

This procedure converts the covered region of a pixel into a pixelmask.

calc_pixel_area(cur_x,cur_y,dxdy)

ScreenType	cur_x, cur_y
ScreenType	dxdy

Given the point where an edge enters a pixel and $1/slope$ of the edge, this routine returns the area of the pixel under the edge.

create_frag(zmin,zmax,mask, area,opacity,r,g,b,A_buff)

Screentype	zmin,zmax
pixelmask	mask

WorldType area,opacity

ColorType r,g,b

pixelstruct **A_buff

This routine creates a fragment for one pixel

fragment_insert(p_struct, frag_in)

pixelstruct	*p_struct
fragment	*frag_in

This function inserts one fragment into an element from the a-buffer.

Packing the *a*-buffer

pack(A_buff)

pixelstruct **A_buff

Pack takes an *a*-buffer filled with picture information and returns the r,g,b values for the final image.

Appendix D

The Viewing Transformations

The following sections are based on Alvy Ray Smith's SIGGRAPH tutorial[32]. The material in Smith's article is reiterated with special attention paid to the transformations that affect the illumination calculations. Furthermore, the terminology used differs slightly than Smith's. This appendix serves well as a reference for the preceding chapters.

D.1 Definitions

Object Space: The local coordinate system of an object. Usually, the origin of the local coordinate system lies at the center of the object.

World Space: A common frame of reference for the lights, the eytepoint, and the objects. Illumination can be performed here, but eye space is more convenient.

Eye Space: The standard frame of reference. This space is similar to world space except the eyepoint is always located at the origin and is looking down the z-axis.

Clipping Space: The canonical viewing frustum used for clipping.

Display Space: The coordinate system of the display. Coordinates in display space are floating point numbers.

Image Space: The discrete coordinate system of the display device. Image Space coordinates are integer values.

D.2 An Overview

The viewing transformations find the map needed to guide a sample point from the three dimensional world space to the two dimensional image space. One way to perform these transformations would be to find a single transformation that takes objects directly from world space to image space. However, to make illumination calculations and clipping calculations manageable, two stops are made on the road from world space to eye space. The first one is in eye space to perform the illumination calculations; the second is in clipping space in order to simplify the clipping calculations. The viewing transformation takes the following path: world space to eye space, eye space to clipping space, and finally clipping space to image space.

D.2.1 World Space to Eye Space

The modeling matrices transform the objects from object space to world space. In world space, all the objects, all the lights, and the viewpoint are positioned relative to each other in the same coordinate system. The frame of reference is neutral. In eye space, however, all the objects are defined in the eyepoint's frame of reference. The eyepoint occupies the origin, the eyepoint's view normal is aligned with the z-axis, the eyepoint's view-up vector is aligned with the y-axis, and the remaining eyepoint vector u is aligned with the x-axis. Since both of these coordinates systems are orthonormal and of the same scale, the angles between objects are preserved in the transition from world space to eye space. Furthermore, because the eyepoint's position is at the origin, the vector from the eyepoint to a point on the surface is just the coordinate of the object. The illumination calculation depends on the angles between the objects, the viewer, and the light source. Therefore, eye space is ideal for the illumination calculations.

The world space to eye space transform is the matrix that will transform the axes defining world space coordinates into the coordinates defining the eyepoint local coordinate system. In order to understand this transformation, it is helpful to break this transformation into two distinct operations.

The first matrix, A, translates the world space origin to the location of the eyepoint. This matrix is the following:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -V_x & -V_y & -V_z & 1 \end{bmatrix}$$

which can be thought of as the partitioned matrix,

$$A = \begin{bmatrix} I_3 & 0^T \\ -V & 1 \end{bmatrix}$$

Now the two coordinate systems share a common origin. The second matrix, B, aligns their axes. This matrix is the following:

$$B = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $\vec{u}$, $\vec{v}$, and $\vec{n}$ are the vectors defining the axes of the local coordinate system at the eyepoint.

Note that the world space coordinate system is right-handed [1]; the eyepoint's local coordinate system is left-handed[2].

[1]the x-axis points to the right, the y-axis points up, and the z-axis points into the distance.

[2]the x axis points to the right, the y axis points to the distance, and the z axis points up

D.2.2 Eye Space to Clipping Space

The next transformation coerces one pyramid, the field of view in eye space, into another pyramid, the canonical view volume[3] in clipping space. The field of view in eye space is the volume bounded by the four edges which run from the origin, pass through the corners of the viewing window, and then end at the far plane. The canonical view volume is bounded by the four edges which run from the origin to the points $(-1,1,1,)$, $(1,1,1)$, $(1,-1,1)$ and $(-1,-1,1)$. Note that the transformation from eye space to clipping space deforms the original space; the angles between points are not preserved in clipping space. Therefore, the illumination calculations cannot be done in clipping space. [4]

To understand this transformation, it is useful to view it as the concatenation of four transformations. The first matrix, E, deforms the space so that the view window becomes square:

$$E = \begin{bmatrix} 1/s_u & 0 & 0 & 0 \\ 0 & 1/s_v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where s_u is the window halfsize in the horizontal direction, and the s_v is the window halfsize vertical direction.

[3]The canonical view volume is the similar to the canonical viewing frustum except the volume between the near clipping plane and the origin is included

[4]It is interesting to note that the viewing plane and the objects lie close to the viewpoint in most cases; therefore, the coordinates of the objects in clipping space are diminutive.

The next matrix, D, shears the space so that the center of the viewplane window lies on the z-axis:

$$D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -c_u & -c_v & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where (c_u, c_v) is the center on the viewplane window.

And the final matrix scales the space so that the far view plane is at $z = 1$:

$$C = \begin{bmatrix} d/f & 0 & 0 & 0 \\ 0 & d/f & 0 & 0 \\ 0 & 0 & 1/f & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where d is the distance from the eye to the viewing plane, and f is the distance from the eye to the far plane.

D.2.3 Concatenating the matrices

The matrices which bring an object from world space to clipping space can be concatenated into one matrix and then factored into two matrices:

$$N_L N_R = ABCD$$

N_L depends solely on the orientation and direction of the view, and the other matrix, N_R,depends on the lens attributes of the camera at the eyepoint. Thus, to change the position and orientation of the eyepoint, one need only to change one of the matrices.

The matrix which moves the viewing position is N_L:

$$N_L = \begin{bmatrix} u_x & v_x & n_x & 0 \\ u_y & v_y & n_y & 0 \\ u_z & v_z & n_z & 0 \\ -u \cdot V & -v \cdot V & -n \cdot V & 1 \end{bmatrix}$$

where $\vec{u}$, $\vec{v}$, and $\vec{n}$ are the vectors defining the axes of the local coordinate system at the eyepoint; where V is the vector from the origin to the eye position, and where $u \cdot V$,$u \cdot V$, $u \cdot V$ are the dot products of the coordinate vectors and the eyepoint vector.

The matrix which relies only on the lens attributes of the camera is N_R:

$$N_R = \begin{bmatrix} 1/s_u & 0 & 0 & 0 \\ 0 & 1/s_v & 0 & 0 \\ 0 & 0 & 1/(f-n) & 0 \\ -c_u/s_u & -c_v/s_v & -n/(f-n) & 1 \end{bmatrix}$$

where s_u and s_v are the window half sizes, c_u and c_v define the center of the window on the viewplane, f is the far clipping plane, n is the near clipping plane, and d is the distance from the eye to the viewplane.

D.2.4 Clipping Space to Image Space

The trip from clipping space to image space is done in two stages. The first is device independent. It transforms the three dimensional points into normalized device coordinates (NDC)[5]. The second stage maps the NDC

[5]Normalized device coordinates is a coordinate system where the x,y, and z values fall between 0.0 and 1.0

points onto a display device. This second stage depends entirely upon the coordinate system of the display device.

Stage One: Clipping space to NDC

There are two transformations which bring the coordinates from the clipping space into NDC. First, the viewing pyramid in clipping space is transformed into a cube shaped space. This space, the perspective space, is bounded by the planes $x = -1.0$, $x = 1.0$, $y = -1.0$, $y = 1.0$, $z = 0.0$, and $z = 1.0$. Notice that the objects are distorted when they are transformed into perspective space. In this new distorted space, the orthogonal projection of the objects onto the viewing plane is equivalent to a perspective projection of the points in clipping space. Since this transformation brings the points to a space where the perspective projection is trivial, this transformation is called the perspective transformation. The perspective transformation is performed by the following matrix, F:

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & f/(f-n) & 1 \\ 0 & 0 & -n/(f-n) & 1 \end{bmatrix}$$

Next, the coordinates of the objects in the perspective cube are transformed to NDC. The matrix G performs this transformation:

$$G = \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix}$$

Stage Two: NDC to ScreenSpace

In this final stage, the samples are constructed into a final image on the display. There are two distinct operations that occur in this stage. During the first, the image is constructed by scan conversion. This constructed image is a continuous image residing in display space. The transformation which takes the image from NDC to display space depends solely on the coordinates of the display device. The following matrix, H, performs this transformation:

$$H = \begin{bmatrix} X_{max} - X_{min} & 0 & 0 & 0 \\ 0 & Y_{max} - Y_{min} & 0 & 0 \\ 0 & 0 & Z_{max} - Z_{min} & 0 \\ X_{min} & Y_{min} & Z_{min} & 1 \end{bmatrix}$$

In order for the resulting continuous image to be viewed on a device, it must be broken into discrete pixels on the screen. Aliasing occurs during this last round of sampling. Therefore, any filters used for anti-aliasing are applied during this last step to the screen.

Bibliography

[1] Appel, Arthur, "The Notion of Quantitative Invisibility and the Machine Rendering Solids," *Proceedings, ACM National Conference (Oct.)*, ACM, New York, 1967, pp. 387-393.

[2] Blinn, James F., "Simulation of Wrinkled Surfaces," *Computer Graphics*, 12(2), Proc. SIGGRAPH 78, 1978, pp. 286-292.

[3] Blinn, James F., "Models of Light Reflection For Computer Synthesized Pictures," *Computer Graphics*, 11(2), Summer 1977, pp. 192-198.

[4] Bui-Tuong, Phong, "Illumination for Computer Generated Pictures," *Communications of the ACM*, 18(6), June 1975, pp. 311-317.

[5] Carpenter, Loren, "The A-buffer, an Antialiased Hidden Surface Method," *Computer Graphics*, 18(3), Proc. SIGGRAPH 84, 1984, pp. 103-108.

[6] Catmull, Edwin, "Computer Display of Curved Surfaces," Tutorial and Selected Readings in Interactive Computer Graphics, *H. Freeman (ed.), IEEE*, 1980, pp. 309-315.

[7] Cleary, John G., Brian Wyvill, Grahamn M. Birtwistle, and Reddy

Vatti, "Multiprocessor Ray Tracing," Department of Computer Science, The University of Calgary, Research Report No. 83/128/17, October 1973.

[8] Cohen, Michael F., "The Hemi-Cube: A Radiosity Solution for Complex Environments," *Computer Graphics*, 19(3), Proc. SIGGRAPH 85, 1985, pp. 31-40.

[9] Cook, Rob, "Antialiasing by Stochastic Sampling," *Course Notes: State of the Art in Image Synthesis* ACM SIGGRAPH '85, July 22, 1985

[10] Cook, Robert L., Thomas Porter, and Loren Carpenter, "Distributed Ray Tracing," *Computer Graphics*, 18(3), Proc. ACM Siggraph 84, July 1984, pp 137-145.

[11] Cook, Robert, and Kenneth E. Torrence, "A Reflectance Model for Computer Graphics," *ACM Trans. on Graphics*, Vol. 1, No. 1, 1982, pp. 7-24.

[12] Crow, F.C., "A More Flexible Image Generation Environment," *Computer Graphics*, 16(3), Proc. SIGGRAPH 82, 1982, pp. 9-18.

[13] Dippe, Mark Z.and Erling Henry Wold, "Antialiasing Through Stochastic Sampling," *Computer Graphics*, 19(3), Proc. SIGGRAPH 85, July 1985, pp. 69-78.

[14] Duff, Tom, "Smoothly Shaded Renderings of Polyhedral Objects on Raster Displays," *Computer Graphics*, 13(2), Proc. SIGGRAPH 79, August, 1979, pp. 270-275..

[15] Fiume, Eugene, and Alain Fournier, "A Parallel Scan Conversion Algorithm With Anit-Aliasing for a General-Purpose Ultracomputer," *Computer Graphics*, 17(3), Proc. SIGGRAPH 83, July, 1983, pp 141-149.

[16] Glassner, Andrew, "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics & Aplications*, 4(10), October, 1984, pp. 15-22,

[17] Goldstein, R.A., and Nagel, R., "3-D Visual Simulation," *Simulation*, January 1971, pp. 25-31,

[18] Goral, Cindy M., Kenneth E. Torrence, Donald P. Greenberg, and Bennet Battaile, "Modeling the Interaction of Light Between Diffuse Surfaces," *Computer Graphics*, 18(3), Proc. SIGGRAPH 84, 1984, pp. 213-222.

[19] Gouraud, Henri, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, C-20(6), June 1971, pp. 623-628.

[20] Johnson, T.T., "Sketchpad III, A Computer for Drawing in Three Dimensions," *Proceedings of the Spring Joint Computer Conference*, Detroit, Michigan, May21-23, 1963.

[21] Kay, Douglass Scott, and Donald Greenberg, "Transparency for Computer Synthesized Images," *Computer Graphics*, 13(2), Proc. SIGGRAPH 79, August, 1979, pp. 158-164.

[22] Lee, Mark E., Richard A. Redner, and Samuel P. Uselton, "Statistically Optimized Sampling for Distributed Ray Tracing," *Computer Graphics*, 19(3), Proc. ACM Siggraph 85, July 1985, pp. 61-65.

[23] Machover, Carl, "A Brief, Personal History of Computer Graphics," *Computer*, November 1978.

[24] MAGI, "3-D Simulated Graphics Offered by Service Bureau," *Simulation*, p. 69, February 1968.

[25] Mahl, Robert, "Visible Surface Algorithms for Quadric Patches", *IEEE Transactions on Computers*, C-21(1), January 1972, pp. 1-5.

[26] Mandelbrot, B., *Fractals: Form, Chance, and Dimension*, W.H. Freeman, San Francisco, 1977.

[27] Naylor, Bruce F., and William C. Thibault, "Application of BSP Trees to Ray-Tracing and CSG Evaluation," School of Information and Computer Science Tech. Rep., GIT-ICS 86/03, February 4, 1984.

[28] Nemoto, Keiji, and Takao Omachi, "An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing," *Graphics Interface*, May 1986, pp. 43-48.

[29] Newell, M.E., R.G. Newell, and T.L. Sancha, "A Solution to the Hidden Surface Problem," Proc. ACM National Conference, 1972.

[30] Riesenfeld, R.F., "Homogeneous Coordinates and Projective Planes in Computer Graphics," *IEEE Computer Graphics & Applications*, January, 1981, pp. 50-55.

[31] Simms, Karl, personal communication, Thinking Machines Corp. & The Media Laboratory, M.I.T.

[32] Smith, Alvy Ray, "The Viewing Transformation," Computer Graphics Project, Computer Division, Lucasfilm, Ltd. Technical Memo. No. 84,

June, 1983.

[33] Sutherland, Ivan E., "Sketchpad, A Man-Machine Graphical Communication System," *Proceedings of the Spring Joint Computer Conference*, Detroit, Michigan, May21-23, 1963.

[34] Sutherland, Ivan E., R.F. Sproull and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys*, 6(1), March 1974, pp. 1-55.

[35] Warn, David R., "Lighting Controls for Synthetic Images," *Computer Graphics*, 17(3), Proc. ACM Siggraph 83, 1983, pp. 13-21E

[36] Weiss, Ruth A., "BE VISION, A Package of IBM 7090 FORTRAN Programs to Draw Orthographic Views of Combinations of Plane and Quadric Surfaces," *Journal of the Association for Computing Machinery*, 13(2), April 1966, pp. 194-204.

[37] Whitted, Turner, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, 23(6), June 1980, pp. 343-349.