Real Time Holographic Image Rendering: Improvements in Lighting and Realism

by

Kathryn M. Nelson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1993

Author ..... Kathryn M. Nelson .....
Department of Electrical Engineering and Computer Science
May 17, 1993

Certified by ..... Stephen A. Benton .....
Stephen A. Benton
Allen Professor of Media Arts & Sciences
Thesis Supervisor

Accepted by .....
Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Real Time Holographic Image Rendering: Improvements in Lighting and Realism

by

Kathryn M. Nelson

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Electrical Science and Engineering

Abstract

The computer graphics front-end of the MIT Holographic Video Interactive Renderer (Holo-video) is improved and advanced through the implementation of rendering techniques and new lighting models. Specifically, software developed for this thesis provides for image overlap (or occlusion), multiple object rendering, and specular reflections. The algorithm for image occlusion from a finite number of viewing zone regions is presented with an example using five discrete viewing vectors, a number that provides sufficiently realistic overlap cues for the narrow (16-degree) viewing zone of Holo-video. The rendering of multiple object scenes is discussed, with emphasis on the implementation of occlusion. Two models for specular (mirror-like) lighting are discussed, and their implementation within the Holo-video system are described. Finally, a holographic video "game" is used to illustrate the operation and importance of the added features. The goal was to provide the additional elements of realism while still maintaining computation times that are sufficiently short to allow for user interaction.

Thesis Supervisor: Stephen A. Benton
Title: Allen Professor of Media Arts & Sciences

Acknowledgments

Much thanks to Mark Lucente and Prof. Benton. And to the others of the Spatial Imaging Group.

Much thanks to MIT for sponsoring my education.

Much thanks to my family–especially my parents Alan and Dianne Soderfelt, my sister Sherri, my brother Steve, and my sister Lisé.

Much thanks to Goro.....

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Holographic Video Display System (Holo-video) at MIT allows its user to manipulate the holographic image in real time. Given a 3-dimensional description of an object, a supercomputer computes its holographic fringe pattern. A beam of light then illuminates this fringe pattern to produce an image. The user adjusts buttons and dials interfaced to the computer to manipulate the image to produce a new image in a few seconds. The size, rotational orientation, position, and other factors can all be controlled interactively by the user. These images are currently a few centimeters in size. Six megabytes of data are required for each full-color image. Work is being done to increase both the view angle range and the image size [1].

Holo-video research involves many areas: setting up the optical hardware to illuminate the fringe patterns, coding for the actual computation of the fringe patterns, and coding for the computer graphics techniques that are applied to the image. This thesis focuses on the computer graphics side of Holo-video. It describes the code recently added to Holo-video to improve the realism of the image. This software is referred to hereafter as "Holo-realizer." Included are algorithms for occluding along a range of view angles, rendering multiple objects, and shading with a specular component. Also discussed is a video game that makes use of the occlusion and multiple object algorithms.

Chapter 2 describes the `occluded` function as used in the rendering of one object. This function allows for occlusion from five different view angles. Chapter 3 describes

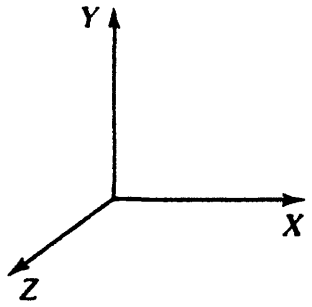

Figure 1-1: Holo-realizer's coordinate system [4, p. 2].

the rendering of two objects and summarizes the rendering of one object. It also describes the rendering of multiple objects. Chapter 4 describes specular reflection. Also discussed in this chapter is the implementation of specular shading. Chapter 5 describes the video game and the functions used to implement it. In Chapter 6, possible future expansions to the code are suggested.

The algorithms in this thesis are based on the coordinate system in Fig. 1-1, with the viewer located very close to the positive z-axis.

1.1 The Interactive Holographic Video Display System

1.1.1 Background

Computer-generated holograms have been discussed among holographers for the last three decades. However, the research has been limited by the display technology available. Frere and Leseberg have studied the computation of large pixel arrays for writing onto film. This process, however, takes many hours and therefore does not allow for images that can be manipulated by the user. To the best of our knowledge, the MIT holographic video system is the first to display holograms as quickly as several times per second. To achieve this, the amount of data and computation was reduced

Figure 1-2: A Volkswagen as displayed on the MIT holographic video system [3, p. 90].

as much as possible without compromising image quality [1, p. 2]. An example of the image Holo-video provides is in Fig. 1-2.

1.1.2 Description

Much information can be found on the MIT holographic video display. To understand this thesis, only a basic knowledge of Holo-video is required. Holo-video is based on a wide-window 3-channel acousto-optical modulator and a spinning polygonal mirror scanner to sweep out laser-illuminated lines that are 32 kilopixels across. A vertical deflection galvanometer then positions these to trace out 64 RGB horizontal lines. The entire sweep is completed 40 times per second, in synchronism with the output of a modified high-resolution frame buffer (2 megasamples in each of three layers, each feeding one view of about 16 degrees, and intended to be viewed at about 600 mm.) [1, p. 17].

Holo-realizer's rendering process is summed up in Fig. 1-3.

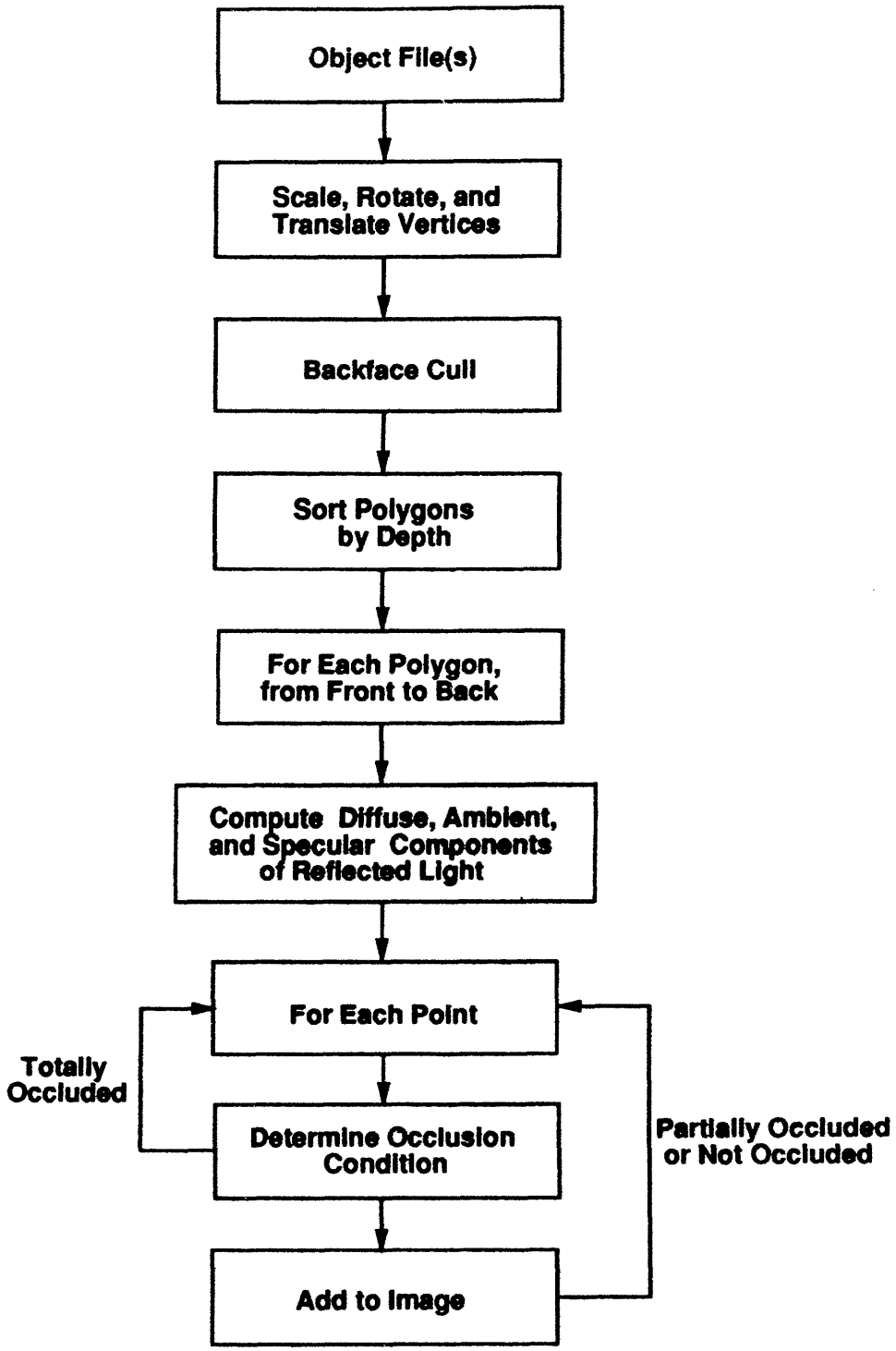

Figure 1-3: Flowchart of Holo-realizer's rendering process.

1.2 The Holo-Graphics Side

This thesis focuses on the graphical computing aspect of Holo-video. Because of this, it does not include fringe calculations or optical hardware descriptions.

Holo-graphics differs from computer graphics due to the 3-dimensionality of Holo-video. Computer graphics deals with the rendering of 3-dimensional models to a 2-dimensional display, whereas Holo-graphics deals with the rendering of 3-dimensional models to a 3-dimensional display. The main result of this difference is that Holo-video has continuously varying multiple view positions. Also, Holo-video fills a polygon with points on a 3-dimensional grid, instead of a 2-dimensional pixel array.

Additionally, the programs implemented are not intended to be perfect in the physical sense. The small size of the Holo-video image enables us to make many approximations while still maintaining image quality. Therefore, this thesis usually describes the theoretically-correct way to implement an algorithm and then describes the data-reduced quicker way, as it was implemented on Holo-video.

Chapter 2

Occlusion

Occlusion or overlap is the process by which nearer objects hide (occlude) farther objects. Due to Holo-video's angle of look-around, an occlusion algorithm must provide accurate occlusion throughout a range of view angles. Because overlap is a very powerful depth cue, the addition of occlusion to Holo-realizer makes the image more realistic.

Holo-realizer renders an object on a polygon-by-polygon basis. For each *hololine* (y value of the hologram), the x's from x_{min} (minimum x value) to x_{max} (maximum x value) are processed. For occlusion, the program ultimately must decide which (x, y, z) points to ignore (for example, if another point with a greater z value already resides at that (x, y) location) and which points to send to the framebuffer (if that (x, y) point has the greatest z value of all the points with the same (x, y) coordinates). This process of selective rendering will produce occlusion. In reality, occlusion is everywhere as we do not see an object which is behind another (opaque) object.

2.1 Preliminary Steps

Before passing the points of each polygon to the `occluded` function, steps are taken to make the actual occlusion process easier. Before an object is rendered, we have a polygon list containing the numbers of the polygons in the object. Our goal is to reduce the number of polygons and to put the polygons in some sort of helpful order.

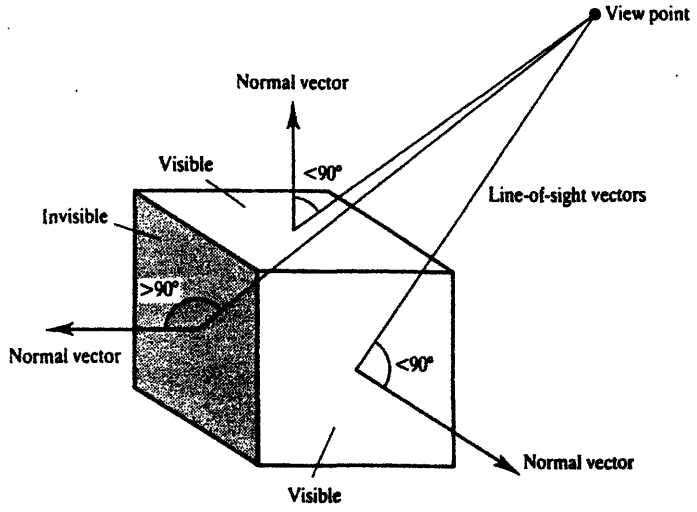

Figure 2-1: Backface culling: two visible polygons and one invisible polygon [4, p. 37].

This reduces both the number of polygons to be processed and the number of steps needed to process a polygon.

2.1.1 A Better Cull

One way to reduce the size of the polygon list to elimate ("cull") polygons that are completely occluded because their normals point toward the back, away from the viewer. Polygons face away from the viewer when the z-components of their normal vectors are negative. See Fig. 2-1.

However, due to the look-around that Holo-video provides, this culling process sometimes eliminates polygons that are occasionally needed (they should be visible when viewed at an angle other than straight-on). A modified backface cull was implemented to account for this.

The modified backface cull takes into account Holo-video's range of view angles. With this cull, a polygon that should be hidden in the straight-on case but visible in the side-view case is not culled out. The part that should be hidden is dealt with later in the `occluded` function.

After the cull, we deal with only those polygons which were not culled out.

2.1.2 Sorting the Polygons

Before rendering the object, a list containing the polygons that were not culled out is put into order from those closest (highest z value) to the viewer to those farthest (lowest z values) from the viewer. To do this, the average z value of each polygon was computed by averaging the z values of the vertices of that polygon. Using the average z values, the polygons are sorted in decreasing order using the Heapsort algorithm. The Heapsort algorithm was chosen for its minimum storage and high speed when dealing with sorts of this size (usually between 50 and 5000 polygons) [2, p. 242]. When the program runs, it uses this updated polygon list to process the polygons in order from closest to farthest. This sort is vital to the occlusion algorithm because if a point already exists at a particular (x, y) location, it probably should occlude any following points at that (x, y) location because its polygon is in front of any of the following polygons. Also, because our range of view angles is small (16 degrees), we are able to use one sort for all five view zones.

2.2 Occlusion

Occlusion from five different view zones is implemented using an array for each view zone. Each array represents the areas that are already filled with points. These areas are represented as lines. Each line is recorded using its minimum *step* value, maximum *step* value, and *hololine* value (where $hololine = y$). Except in the case of straight-on viewing, the *step* value represents the absolute value of the z-intercept of that particular view vector at that point. If the view vector never intersects the z-axis, the *step* value is equal to the x value (as in the case of the straight-on view). The distance between consecutive *step* values is the *stepspace*.

The minimum *step* values are always indexed by even indices, while the maximum *step* values are always indexed by odd indices. This is important in determining whether two consecutive indices represent a line (the lower index even, the upper index odd) or a space between lines (the lower index odd, the upper index even). And, for each row of lines, the lines read from smaller to larger *step* values. The

arrays are initialized to 2000, so any unfilled entries in an array have that value. The value 2000 was chosen because it is larger than the biggest *stepspace* and would thus be consistent with the ordering of higher to lower values in the arrays.

To determine the size of the space between steps (*stepspace*) is not easy. Except in the straight-on case (where the *stepspace* = 1), the calculation for *stepspace* is subject to round-off error. Due to this, the *stepspace* was determined experimentally. It was found that each view (other than the straight-on view) had two *stepspace*'s, differing only by 1. Therefore, the `occluded` function gets passed the smaller *stepspace*, then the function itself deals with the inconsistencies of a particular view zone's *stepspace*.

2.2.1 Explanation of an Array

For example, in the case of the straight-on view (so that $step = x$), if we have

$$Array = \begin{bmatrix} 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 \\ 1 & 5 & 7 & 8 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 \\ 0 & 0 & 2 & 2 & 4 & 4 & 6 & 6 & 8 & 8 \end{bmatrix},$$

this would mean:

1. At the 0^{th} *linerow* there is nothing in the framebuffer.
2. At the 1^{st} *linerow* there is something in the framebuffer from $x = 1$ to $x = 5$, and from $x = 7$ to $x = 8$.
3. At the 2^{nd} *linerow* there is something in the framebuffer from $x = 0$ to $x = 0$, from $x = 2$ to $x = 2$, from $x = 4$ to $x = 4$, from $x = 6$ to $x = 6$, and from $x = 8$ to $x = 8$. Note that these lines are actually points.

2.3 The `Occluded` Function

The `occluded` function starts by checking the specified *linerow* of a point to see if it is empty. A *linerow* is empty if $Array[linerow][0] = 2000$. Since the entries are in increasing order, 2000 must represent the minimum entry value. But, the arrays are initialized to 2000, so this *linerow* has not been touched since that time (it is empty). If so, this point is deemed not occluded.

Next, `occluded` checks if *step* comes before or is equal to the first entry in the array. If $step < Array[linerow][0]$, it either forms its own line or it becomes an extension of the other line. Either way, this point is not occluded. If $step = Array[linerow][0]$, the point is occluded.

If $step > Array[linerow][0]$, the program cycles through the array to determine which two entries the point falls between. (These two entries are the reference entries, the lower reference index $ref1$ and the upper reference index $ref2$.) Once this is done, whether or not the point falls in a line is determined by testing whether $ref1$ is even or odd. If it does fall in a line, it is occluded. If not, the next four cases are tested.

The function checks for the cases of the point: coming before the existing lines (not occluded), being at the end or start of a line (occluded), filling the gap between two lines to form a new line (not occluded), extending an existing line (not occluded), and making its own line (not occluded). These cases are described as follows:

1. Ends or starts a line: *step* comes at the end or start of a line if $step = Array[linerow][ref1]$ or $step = Array[linerow][ref2]$. If so, the array remains the same and the point is occluded.

2. Fills the gap between two lines: *step* fills the gap between two lines if $Array[linerow][ref1] + stepspace = step = Array[linerow][ref2] - stepspace$. If so, the array must change to account for the new bigger line–old minimum and maximum step values must be written over with new ones. The point is not occluded.

3. Extends an existing line: *step* extends an existing line if

$step = Array[linerow][ref1] + stepspace$ or

$step = Array[linerow][ref2] - stepspace$. If so, the appropriate reference entry must change to account for the new minimum or maximum. The point is not occluded.

4. Makes its own line: *step* makes its own line if it satisfies none of the above conditions. If so, part of the array must be be shifted to the right by two to make room for the new minimum and maximum (both of which are equal to *step*). The point is not occluded.

To reduce the amount of cycling required for each point, a *tracker* variable is kept for each view zone which tells the occluded function at which index to start cycling. This *tracker* variable is set to its correct value in the `occluded` function and is set to 0 for each new *linerow*.

Thus, the program finds in what view zones a point is occluded and then makes its amplitude of light 0 in those directions to account for the occlusion.

2.3.1 Example

For the straight-on view (so that $step = x$), we have the point $(10, 1, 20)$ and we have

$$Array = \begin{bmatrix} 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 \\ 1 & 5 & 7 & 8 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 \\ 0 & 0 & 2 & 2 & 4 & 4 & 6 & 6 & 8 & 8 \end{bmatrix}.$$

The program first checks if the linerow is empty. Since $Array[linerow][0] \neq 2000$, it is not. Next, the program checks if the $step < Array[linerow][0]$. In this case, since $10 \not< 1$, it continues.

The cycling begins. If this were not the first time we are dealing with this polygon's linerow, we would have our *tracker* variable set to within a cycle or two of where we would want to be. This, however, is not the case. Since this is the first time we

are dealing with this linerow, *tracker* will be set to 0. Starting with the 0^{th} and 1^{st} entries, we keep cycling until we find two entries our *step* comes between. This does not happen until we reach the 3^{rd} and 4^{th} entries. We now have the indices for our reference entries: $ref1 = 3$ and $ref2 = 4$.

The program checks if we are already in a line by determining if the $ref1$ is even or odd. Since 3 is odd, we are not in a line. The program continues by checking the next three cases.

1. Ends or starts a line: not true because $10 \neq 8$ or $10 \neq 2000$.
2. Fills the gap between two lines: not true because $10 + stepspace \neq 2000$ and $10 - stepspace \neq 8$.
3. Extends an already existing line: not true because $10 + stepspace \neq 2000$ or $10 - stepspace \neq 8$.

Finally, we must default to our last case–the point makes its own line. The *tracker* variable is set to 5 to reduce the amount of cycling needed for the next point on the *linerow*. The function returns a 0 (not occluded), and adjusts the array to accommodate for the new line. The new array is as follows:

$$Array = \begin{bmatrix} 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 & 2000 \\ 1 & 5 & 7 & 8 & 10 & 10 & 2000 & 2000 & 2000 & 2000 \\ 0 & 0 & 2 & 2 & 4 & 4 & 6 & 6 & 8 & 8 \end{bmatrix}.$$

Notice we did not consider the point's z value in the `occluded` function since we know the polygons are processed greatest z's first.

Chapter 3

Multiple Objects

Before any work on the video game was to be done, the task of displaying more than one object at a time had to be addressed. This raised a lot of problems. How will occlusion work with multiple objects? Will the objects be physically separate? If so, how will we deal with the objects staying out of the other objects' space?

To better understand how multiple objects are rendered, we must understand the basics of how one object is rendered.

3.1 Rendering One Object

One object is rendered using a single input–the object file. The object file contains the number of polygons, the number of vertices in the object, and the (x, y, z) coordinates for each vertex of each polygon in the object.

The program reads in *Numverts* (the number of vertices in the entire object) and *Numpolyg* (the number of polygons in the entire object). It then puts the coordinates of the vertices of each polygon into one of three arrays–Vtx for the x-component, Vty for the y-component, and Vtz for the z-component.

The program continues on a polygon-by-polygon basis. The polygons are sorted closest ones first. Then the polygon are processed one by one closest ones first. To process, the program runs through all the x's of all the y's of a particular polygon until it is done. For every (x, y, z) point what is written to the framebuffer depends

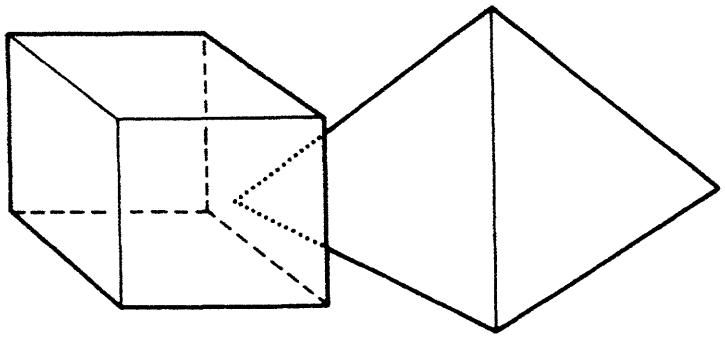

Figure 3-1: Occlusion and backface culling as seen on two objects [4, p. 36].

on the results of the five different calls to the `occluded` function. See Chapter 2 for more information on the `occluded` function.

3.2 Rendering Two Objects

Fortunately, because the `occluded` function operates on polygon lists and not object files, the rendering of two objects is very similar to the rendering of one object. Both the modifed backface cull and the occluded function is used. See Fig. 3-1.

The program will now expect two inputs instead of one–the files for both the objects. The files contain the number of polygons, the number of vertices in the whole object, and the (x, y, z) coordinates for each vertex of each polygon in the object.

The program reads in $Numverts1$, $Numpolyg1$, $Numverts2$, and $Numpolyg2$. It then sets up a new variable $Numpolyg3 = Numpolyg1 + Numpolyg2$. For object1, it puts the coordinates of the vertices of each polygon into one of three arrays–$Vtx1$ for the x-component, $Vty1$ for the y-component, and $Vtz1$ for the z-component. Similarly, for object2, it puts the coordinates the vertices of each polygon into one of three arrays–$Vtx2$ for the x-component, $Vty2$ for the y-component, and $Vtz2$ for the

z-component.

The program continues on a polygon-by-polygon basis. The polygons of both objects are concatenated into one polygon list of size $NumPolyg3$. To determine the coordinates of vertices of a polygon the polygon number (*polyn*) must be checked. If $polyn <= Numpolyg1$ (it belongs to the first object), the program maintains *polyn* and looks in object1's arrays for information. If $polyn > Numpolyg1$ (it belongs to the second object), a new *polyn* is used where $polyn = polyn - Numpolyg1$. The program will also look in object2's arrays to determine the coordinates of the vertices of *polyn*. The program continues in the same way as in the one object case.

For every new cycle, it is checked if the user (or program if not in interactive mode) indicates to move the first object or second object. Any object that is moved will have its list of vertices operated on. Then, the rendering of polygons continues as usual.

Since the `occluded` function operates on a list of polygons and not actual object files, no special precautions must be taken to have occlusion with two objects.

When the two objects occupy the same space, neither wins out. If the program were written to do so, much time and effort would be wasted by keeping track of the boundaries of the objects at all times so that they do not intrude on the other's space. Instead, if both objects occupy the same space, the two objects appear to be merged into one object. The inconsistencies regarding lighting parameters will be ignored. The only way to determine what object the a particular point will draw its characteristics from is (1) the average z value of its polygon's vertices and (2) whether it comes from object1 or object2. Higher average z values take precedence over lower ones, and if the average z values are equal, object1 will take precedence over object2.

3.3 Rendering Multiple Objects

There are two ways to extend the preceding ideas to rendering independent multiple objects, which one is better depends on the number of objects desired. To render N objects when N is small (< 5), the ideas in the two object case are followed, except

that now there are N times as many parameters as before. When N is large, a better strategy is to use linked lists. This way the program (instead of the programmer) will deal with the declaring of N times as many parameters.

Chapter 4

Specular Reflection

In reality, an illuminated object has three components of reflected light–diffuse, ambient, and specular. To make Holo-video more realistic, these different types of light must be added to the images. Diffuse and ambient light were already implemented. Specular light is view-angle dependent and requires a special function which allows a point to have a different brightness value for each of the five view zones.

4.1 Definition and Description

Because real-life objects do not perfectly diffuse light, they usually have some degree of glossiness (a mirror has infinite glossiness). Glossy surfaces differ from dull ones in many ways. First, light reflects off a glossy surface predominantly at an angle $-\theta$, where $+\theta$ is the angle between the incident beam and the surface normal. This is shown in Fig. 4-1. As a result, the amount of specular reflection is view-angle dependent. For example, a mirror upon which a single point source of light is shone will reflect all of the light along only one direction. It will only be seen when viewed along this direction. For all other view directions, that part of the surface will be dark. In real life, however, specular reflection is not so perfect. The viewer is able to see the light not only from the single direction, but also along other directions close to that direction. The area over which the light is seen is referred to as the highlight or glint. The color of the specular light is usually the color of the light source [4, pp.

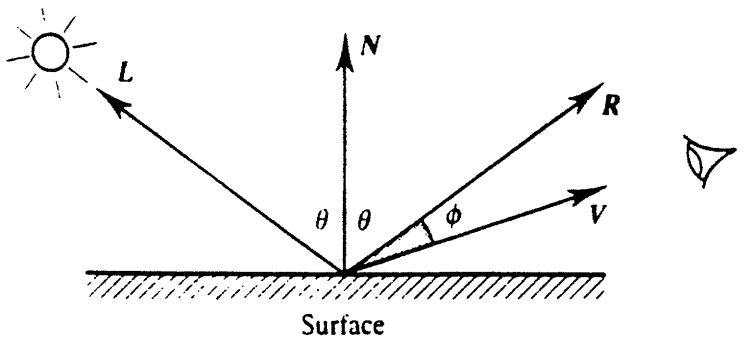

Figure 4-1: Vectors and angles used in the Phong shading model [4, p. 51].

50–51].

4.2 The Phong Shading Model

For a point on the surface of our object, we use two additional vectors to model specular reflection. This is shown in Fig. 4-1. The normal vector to the polygon's surface is N and the vector in the direction from the light source to the particular point in question is L. The two additional vectors–R and V–represent the direction reflection vector for specular light (R) and the direction vector for view (V). All vectors are normalized [4, p. 51].

Using this model, the intensity (I_s) of the specular light can be calculated using Equation (4.1).

$$I_s = I_i k_s (R \cdot V)^n / (r + k) \tag{4.1}$$

where we have approximated $I_i k_s/(r + k)$ to be 1 [4, p. 52].

The angle between the R and V vectors is ϕ. (In the case of infinite glossiness, this angle must be 0 to see any specular light.) The angle between the L and N vectors is θ, and is equal in magnitude to the angle between the N and R vectors. As

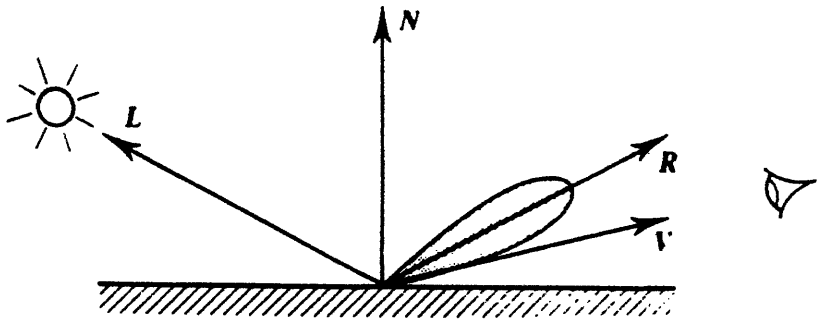

Figure 4-2: The effects of n–large n [4, p. 52].

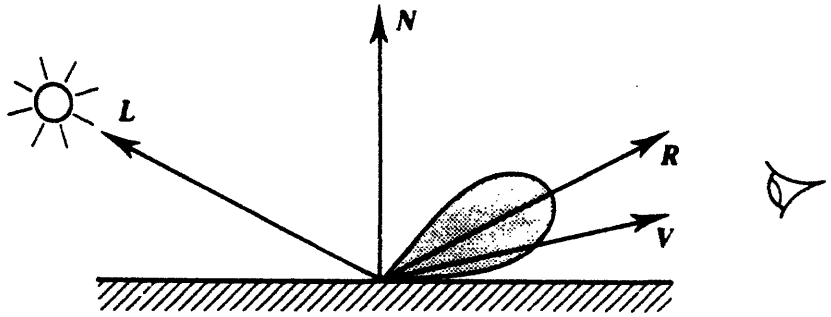

Figure 4-3: The effects of n–small n [4, p. 52].

stated previously, ϕ extends over a range which is dependent upon the glossiness of the object. The glossiness of the object depends on the exponent n. The higher the n the glossier the surface will appear. See Figs. 4-2 and 4-3. A mirror, for example, would have $n = \infty$ since it is a perfectly glossy surface. The glossier the surface, the smaller the area of the highlight. As the surface gets duller, the highlight grows in area but decreases in intensity [4, pp. 51–52].

Our model for specular reflection is only a local model. It does not consider light that comes from other specular reflections [4, pp. 52–53].

4.2.1 Summary

The Phong model can be summarized by the following:

- Ambient light is constant.
- The color of the highlight is that of the light source.
- The glossiness of an object is modelled empirically.
- Diffuse and specular components are modelled locally.
- Light sources are modelled as point sources.

[4, pp. 55]

4.3 Geometric Considerations

Because of the time spent in calculating R, it is better to use another related vector, H, to calculate the magnitude of the specular light. H is the normalized unit vector of a hypothetical surface situated in a direction halfway between L and V [4, p. 53]. See Fig. 4-4.

The vector H is desirable because:

- H is easy to calculate ($H = (L + V)/2$).
- $(R \cdot V)^n$ is not exact, and n is empirical, so $(N \cdot H)^n$ is a good substitute.

Using this model, the intensity (I_s) of the specular light can be calculated using Equation (4.2).

$$I_s = I_i k_s (N \cdot H)^n / (r + k) \tag{4.2}$$

where we have approximated $I_i k_s/(r + k)$ to be 1 [4, p. 54].

The differences between using $R \cdot V$ and $N \cdot H$ can be seen in Figs. 4-5 and 4-6. Notice how the use of $N \cdot H$ spreads the highlight out over a greater area. Adjusting n will compensate for the difference in the angle between R and V and the angle between N and H [4, p. 54].

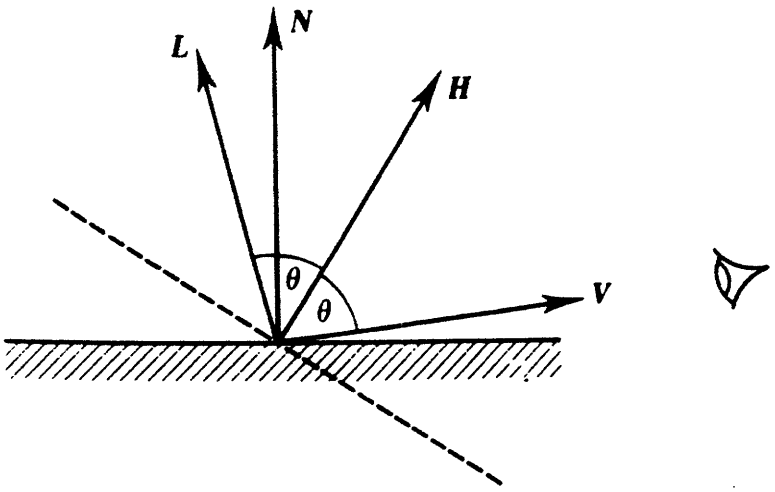

Figure 4-4: The H vector [4, p. 53].

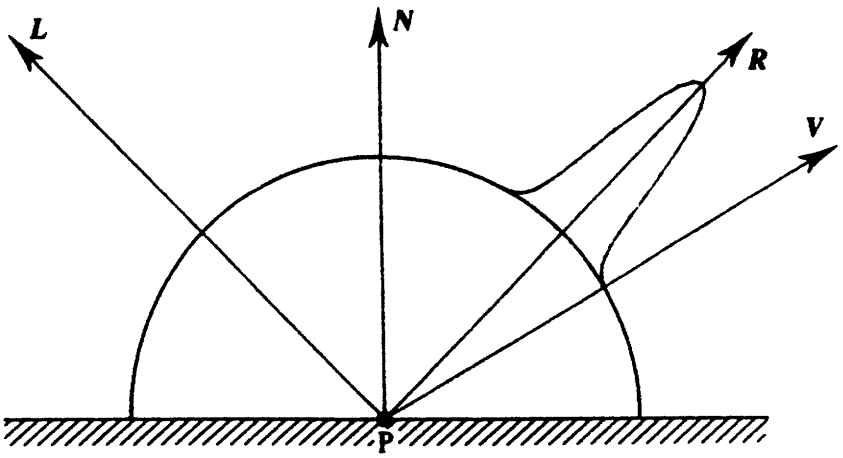

Figure 4-5: Using the $R \cdot V$ model [4, p. 54].

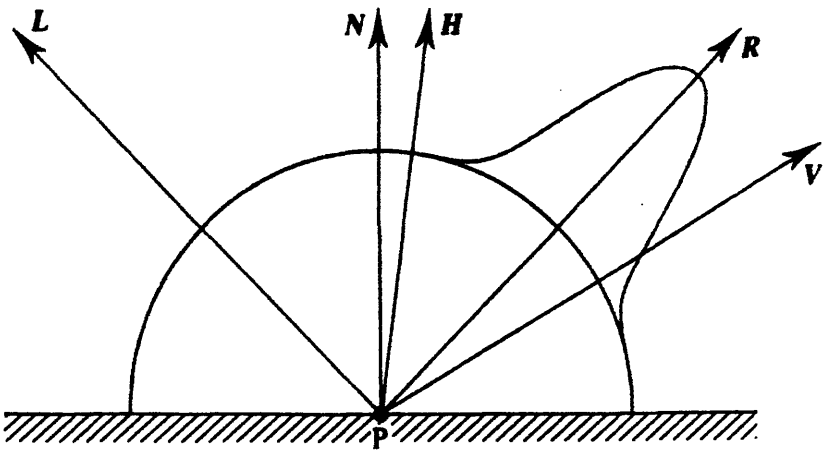

Figure 4-6: Using the $N \cdot H$ model [4, p. 54].

4.4 Implementation

To implement specular reflection, the V vectors must be specified and the H vectors must be calculated. The angle between consecutive view vectors is found by dividing the total angle of view into five equally spaced view zones.

The H vectors are calculated by adding the vectors L and V and dividing by 2 for each view zone. The other terms (except n) in equation are approximated as 1. Depending on how shiny we want the object to look, n varies from 1 to 5.

If done correctly, for every view zone of each point, the diffuse, ambient, and specular terms of light would be calculated and added together. Each point would have five of these sums which would then be passed to the function that would render the point with the five different brightnesses in the correct view zones (assuming there is no occlusion).

However, we did not calculate the specular light for each point. Because our image is small, we are able to calculate our specular light on a polygon-by-polygon basis (rather than a point-by-point basis) without losing any significant image quality in most cases. For a single polygon, N and L are calculated. Then, V's and H's are

calculated for this particular polygon. The resulting light intensity is then used for every point on this particular polygon. Of course, this works better if the number of polygons is large, and the individual polygons are physically small.

Chapter 5

Holographic Video Game

To exploit the interactive nature of Holo-realizer, some sort of procedure which persuades the user to interact had to be implemented. Since most people are familiar with video games, this seemed the optimal medium to use. If the users of Holo-realizer were playing a game instead of moving the objects when they chose to, they would be more motivated to interact with the system. Using the algorithms described in Chapters 2 and 3, a video game for Holo-realizer was developed.

The video game is a basic action game, requiring some hand-eye coordination, but little strategy. The user (the "player") fights to survive despite the existence of the enemies (the "attackers"); namely, the player tries to avoid colliding with the attackers in order to earn points and avoid death.

5.1 Preliminary Considerations

Because a video game must be fast to hold the player's interest, the normal rendering of images that Holo-video uses is modified to allow for more speed in the processing of polygons. Also, since a video game does not necessarily have to be realistic, the lighting is modified.

5.1.1 Modified Lighting

For the video game, the only lighting consideration is that the objects be bright and viewable. In order to speed up the processing of polygons, only ambient light is used. Neither the diffuse nor the specular terms are calculated.

5.1.2 Other Modifications

Because a video game is more fun with many simple objects on the screen than with a few complicated objects, care was taken in choosing what objects to use. No objects with greater than 2000 polygons were used.

5.2 Functions Needed for the Game

Before writing the overall control system for the game, a series of functions had to be created which would be needed for the game, but not used normally in interaction with Holo-video. Functions were needed to move an object in six different directions, to detect if the game was over, and to actually end the game.

5.2.1 Control Functions

To allow the player to move in a certain direction, the program simply shifts the appropriate origin coordinate of the object the correct amount. For example, the function `move_up` increments the z-coordinate of the origin of the player's object. Similar functions created were: `move_down` (decrements the y-coordinate), `move_right` (increments the x-coordinate), `move_left` (decrements the x-coordinate), `move_in` (decrements the z-coordinate), and `move_out` (increments the z-coordinate).

5.2.2 Game-Over Functions

Two additional functions were needed to implement the video game–one to detect when the game was over, and one to actually implement the game-over protocol.

The **over_yet** function checks after the rendering of each object whether the game is over. It determines the game is over when it finds that the origin of the player is close to the origin of an attacker. If **over_yet** deems the game to be over, **over_yet** makes use of the **game_over** function described below. If not, the game continues as usual.

The **game_over** function is called only after it is determined that the game is over. First, it sounds a beeper three times to indicate to the player impending doom. Second, it clears the screen and fills it with a truly spectacular display. And, lastly, it exits out of the program.

5.3 The Video Game Algorithm

Initially, the game takes as input two files–the player's file and the attackers' file. The player's file contains information about the player's object (which is the only object that can be interacted with). The attackers' file contains information about the objects of the attackers'. These objects are moved only as specified in the game program. The many objects in the attackers' file are treated as one object throughout the game.

The game runs by continually moving the attackers' objects and moving the player's object only when the player hits a certain button. After each object has been processed, **over_yet** is used to check if the game is over. If so, the actions carried out by **game_over** are started. If not, the buttons are checked to see if the player wishes to move. If so, the appropriate shifting is performed on the player's array of vertices. Whether the player moves or not, the attackers' array of vertices is changed so it will move. Then the program proceeds with the rendering, using the two object format described in Chapter 2.

Chapter 6

Conclusion

The purpose of this thesis was to add many features to Holo-video to improve the realism of the displayed image. We have shown that it is possible to add the realism cues of occlusion and specular reflection to an image while maintaining the interactivity of the system. This was accomplished by making substantial theoretical compromises which produced only small compromises in the quality of the image.

With the development of the video game for Holo-realizer, Holo-realizer has been brought into a more practical light. Users are now more inclined to interact with the system.

6.1 Future Work

6.1.1 Suggestions for Future Work

Holo-realizer can be developed further in two ways–either by making the existing programs more complex and theoretically correct (for example, making occlusion for a greater number of views), or by adding more data-reduced features to the system. Some examples of these features are:

- The addition of multiple user-controlled light sources.
- Light sources not at ∞.

- The animation of objects.
- The mutation of objects.
- Non-constant shading.
- Transparency.

However, as future versions of the display come to involve larger images, the accuracy of the models will become more important, and previous assumptions must be discarded. Some notable changes to the algorithms in this thesis would be:

- A modified backface cull that accounts for the larger view angle.
- Multiple sorts for the polygons.
- Dividing each degree of the view angle into a greater number of view zones.
- Calculating the specular light more than once for a given polygon.

Considering the pros and cons of each path, it would be better to expand Holo-realizer in the traditional sense–that is, using shortcuts for speed and not concentrating on theoretical correctness. In the future, the incorporation of additional features to Holo-realizer will provide important gains in realism.

REFERENCES

[1] Benton, Stephen A. and Lucente, Mark, "Interactive Computation of Display Holograms", *Proceedings of Computer Graphics International '92*: 1992.

[2] Flannery, Brian P.; Press, William H.; Teukolsky, Saul A.; and Vetterling, William T., *Numerical Recipes in C*, New York: Cambridge University Press, 1988.

[3] UnderKoffler, John Stephen, "Toward Accurate Computation of Optically Reconstructed Holograms", S.M. thesis, MIT, Cambridge, MA: 1991.

[4] Watt, Alan, *Fundamentals of Three-Dimensional Computer Graphics*, Reading, MA: Addison-Wesley Publishing Company, 1989.