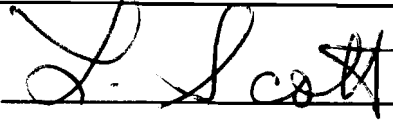


AN ABSTRACT OF THE THESIS OF

John Dennis Albers for the Master of Science
in Mathematics presented on July 27, 1988

Title: HIDDEN LINE REMOVAL - MORE THAN MEETS THE EYE

Abstract approved: 

The purpose of this thesis is to investigate a method in which three-dimensional objects can be geometrically modeled and realistically displayed on a two-dimensional view screen. When a computer generates an image, without special programming instructions, all parts of the object including the hidden parts are displayed. The identification and removal of the hidden parts of an object plays a major role in the production of realistic images. Along with a development of the basic concepts involved with three-dimensional graphics, this thesis presents three hidden line removal algorithms. These algorithms will correctly remove all hidden lines from any object that can be modeled as a polyhedron.

HIDDEN LINE REMOVAL - MORE THAN MEETS THE EYE

A Thesis

Presented to

the Division of Mathematics

EMPORIA STATE UNIVERSITY

In Partial Fulfillment

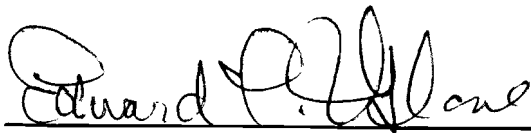
of the Requirements for the Degree

Master of Science

By

John Dennis Albers

July 1988



Approved by the Graduate Council



Approved by the Major Department

464058

DEC 1 1933

ACKNOWLEDGEMENTS

I would like to thank Debra for her support while I was preparing this thesis.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
A Brief History	1
Applications	3
Overview Of The Thesis	3
II. THREE-DIMENSIONAL REPRESENTATIONS	5
Object Representation Tables	6
Data Verification	8
III. BASICS OF THREE-DIMENSIONAL GRAPHICS	10
The Viewpoint	10
The Eye Coordinate System	11
Projection	14
The Viewing Parameters	18
IV. HIDDEN LINE REMOVAL	19
Algorithm Number One (Back Face Removal)	19
Algorithm Number Two (Clipping Edges Against Surfaces)	23
Algorithm Number Three	35
Line Processing	38
V. CONCLUSION	44
Summary Of The Thesis	44
Conclusions	45
Recomendations For Future Study	47
BIBLIOGRAPHY	48
APPENDIX A	49
APPENDIX B	51
APPENDIX C	54
APPENDIX D	56
APPENDIX E	61
APPENDIX F	69
APPENDIX G	79
APPENDIX H	89

LIST OF TABLES

Table		Page
I.	Surface Table	7
II.	Vertex Table	7
III.	Edge Table	7
IV.	Surface Orientation Table	21
V.	Edge Table With Erasure Information	38
VI.	Edge Two (Zero Erasures)	41
VII.	Edge Two (One Erasure)	41
VIII.	Edge Two (Two Disjoint Erasures)	42
IX.	Edge Two (Three Overlapping Erasures Resolved Into Two Disjoint Erasures)	42

LIST OF FIGURES

Figure	Page
1. Regular Tetrahedron	7
2. The Eye Coordinate Axes	12
3. Translate Origin to (Theta,Phi,Rho)	12
4. Rotate (90-Theta) clockwise about Z' Axes ...	12
5. Rotate (180-Phi) clockwise about X' Axes	12
6. Convert to Left-Hand System	12
7. The Projection of an Edge	15
8. Finding Screen Coordinates	16
9. Octahedron	22
10. Concave Polyhedron	24
11. Transparent Object	24
12. Algorithm II, Case 1	27
13. Algorithm II, Case 2	28
14. Algorithm II, Case 3	29
15. Algorithm II, Case 4a	32
16. Algorithm II, Case 4b	33
17. Algorithm II, Case 4b	34
18. Algorithm II, Case 4c	36
19. Point Located Interior To The Polygon	52
20. Point Located Exterior To The Polygon	53

LIST OF PROCEDURES AND FUNCTIONS

Procedure Or Function	Pages
AddEdge	73,84
AddErasure	97
BehindPlane	107
BuildEdgeTable	73,84
CalcNormalVector	83
CalcNormals	64
CalcR2Intersection	104
CalcR2Line	104
CalcR2Vertex	95
CalcR3EyeVertex	109
CalcR3Plane	104
CalcViewVector	95
Clip	92
CramersRule2x2	103
Dec	92
DecodeFunctionKey	94
Deg	89
Det2x2	103
Det3x3	103
DrawAxes	94
DrawBorder	91
DrawBox	95
DrawEdge	72,83
DrawObject	65
DrawSurface	64

Procedure Or Function	Pages
DrawSurfaces	65
EatTestEdge	100
EraseStats	89
ExitProgram	90
EyeXYZ	91
FindEdgeNumber	98
FindIntersectionPoints	106
FindPreImage	108
FindSlope	96
Find_t	96
GetKeySequence	94
Inbetween	102
Inc	92
InfrontPlane	107
Initialize	66,76
InsidePoly	105
LoadSurface	90
LoadVertex	89
MarkSurfaceEdges	98
Max	97
Min	96
Nothing	102
OutsideBox	106
ProcessObject	75,87
ProcessOverlap	99

Procedure Or Function	Pages
ProcessTestEdge	108
R2Angle	103
R2Diff	105
R2Magnitude	103
R2VectorsEqual	97
R2MidPoint	102
R3Diff	91
R3DotProduct	94
R3MidPoint	102
RectangularBoundary	106
RemoveErasure	100
RemoveHiddenLines	74,85
ScreenXY	92
Set_Backedge_Flags	86
Sgn	105
ShowStats	89
SortErasure	98
SwitchIntegers	97
SwitchReals	97
UnmarkSurfaceEdges	99
WrapAroundDec	92
WrapAroundInc	92
ZapScreen	91

CHAPTER I

INTRODUCTION

When looking at an object, the viewer may think that he is seeing it in its entirety, but actually he is only seeing about one-half of the object. It is the opaqueness of the closer surfaces that prevent portions of the surfaces located further away from being seen. When a computer generates an image, no such automatic elimination of the hidden parts takes place. Instead all parts of the object, including the parts that should be hidden from view, are displayed. Hidden line removal refers to the task of identifying and removing the hidden parts of an object.

A Brief History

Towards the end of the era of the second generation computer, interactive computer graphics made its first appearance. While working on his PhD at the Massachusetts Institute of Technology, Ivan Sutherland introduced the concept of using a keyboard and a hand-held light pen for selecting, pointing, and drawing [4, pg. 12]. Most significantly, he developed a data structure based on the topology of the object rather than on the picture. In 1963, Sutherland introduced a computer program called sketchpad. People were excited because this program was able to display a three-dimensional object with the hidden lines removed. A documentary film about the techniques used in this program

was sent to nearly every computer center in the United States.

General Motors was the first user of an elaborate graphics system developed by IBM [4, pg. 15]. The system was called DAC-1 (design augmented by computer). This system was eventually made public at the 1964 Fall Joint Computer Conference. The DAC-1 was the birth of computer aided design (CAD) systems.

The first major reasearch center for computer graphics was established at the University of Utah [4, pg. 15]. In 1972 several breakthroughs were made. Ed Catmull directed reasearch towards finding ways to generate images of curved surfaces. The solution consisted of dividing each surface into very small patches whose relationships to one another could be defined mathematically. Dr. James Blinn developed methods for effective surface modeling. Starting with a wireframe drawing composed of lines, color and texture are then added to the surfaces to give them a realistic appearance.

By 1984, computer graphics technology had advanced so much that it enabled a skilled user to match photographic reality [4, pg. 18]. A picture of an article could be so accuratly simulated on a computer that it was almost impossible to distinguish between the computer generated image and the actual photograph. By the turn of the century it is very probable that computer graphics will begin to replace conventional photographic

technology.

Applications

Many real world applications that involve computer graphics require the display of three-dimensional images of objects and scenes. For example, flight simulation is an application which requires the rapid and continuous display of realistic images relative to the pilot and aircraft [6, pg. 23]. A few of the many applications requiring the generation of realistic images include molecular modeling, animation, and computer-aided design (CAD) systems [10, pg. 294]. Because of the continuing need and desire for more and more realism in computer generated images, hidden line removal becomes a very important aspect in the design of computer graphics software.

Overview Of The Thesis

The purpose of this thesis is to investigate a method in which three-dimensional objects can be geometrically modeled and displayed realistically on a two-dimensional graphics display screen. Chapter I is the introduction which introduces the idea of hidden line removal, gives a short account of the history of computer graphics, lists applications, and finally gives an overview of the thesis. A method for organizing and checking a polyhedral model's vertices, surfaces, and edges is given in Chapter II. In the third chapter, the processes involved in displaying a three-dimensional object are discussed. Chapter IV presents several elementary hidden

line removal algorithms. The fifth and final chapter includes a summary of the thesis, conclusions, and recommendations for future study. Following this chapter, is the bibliography, and then eight appendices. Appendices A,B, and C contain supplementary information pertaining to the second and third hidden line removal algorithms. Appendix D contains instructions for using the programs located on the program disk. Finally appendices E,F,G, and H contain the source code for each of the hidden line removal algorithms.

CHAPTER II

THREE-DIMENSIONAL REPRESENTATIONS

Any three-dimensional object can be modeled by a polyhedron [10, pg. 309]. Objects that have curved surfaces (cylinders, cones, spheres, ...) can be partitioned into a number of flat polygon surfaces. If the object can be partitioned into an adequate number of these flat polygon surfaces, usually an acceptable modeling of the actual object can be achieved. The main components (vertices, edges, and polygon surfaces) of this polyhedral model must be organized in such a way that the computer can use them.

Definition 1. A **polygonal path** is determined by a number of points, $P_1, P_2, \dots, P_{n-1}, P_n$ called the **vertices**, given in a definite order. The path is the set of points on the segments $P_1P_2, P_2P_3, \dots, P_{n-1}P_n$ [2, pg. 144].

Definition 2. A **polygon** is a polygonal path whose beginning and end coincide. If the vertices are all different and no two sides have a point in common (other than a vertex of two adjacent sides), then the polygon is called **simple** [2, pg. 145].

Definition 3. A **polygonal cell** is the set of points on, or interior to, a simple plane polygon $P_1, P_2, \dots, P_{n-1}, P_n$. The **edges** of the cell are

the sides P_1P_2 , $P_2P_3 \dots$ of the polygon [2, pg. 244].

Definition 4. A polyhedron is the set of points on a finite number of polygonal cells joined together in the following way:

- 1.) Any two cells have either no points in common, exactly one vertex in common, or exactly one edge in common.
- 2.) Every edge is on precisely two cells.

[2, pg. 244].

Definition 5. A polyhedron is called **convex** if the segment joining any two points of the polyhedron lies either on the polyhedron or in its interior. A polyhedron is **regular** if it is convex and all faces are congruent regular polygons [2, pg. 244].

Object Representation Tables

One way to organize the main components of a polyhedron, is to create three lists: a **vertex table**, an **edge table**, and a **polygon surface table** [6, pg. 190]. Tables I, II, and III organize the components of the regular tetrahedron in figure 1. First of all, each polygon surface is defined in the polygon surface table as a list of edges. Second, the R^3 standard coordinates for each vertex of the object are stored in the vertex table. Lastly, the edge table lists the endpoint vertices

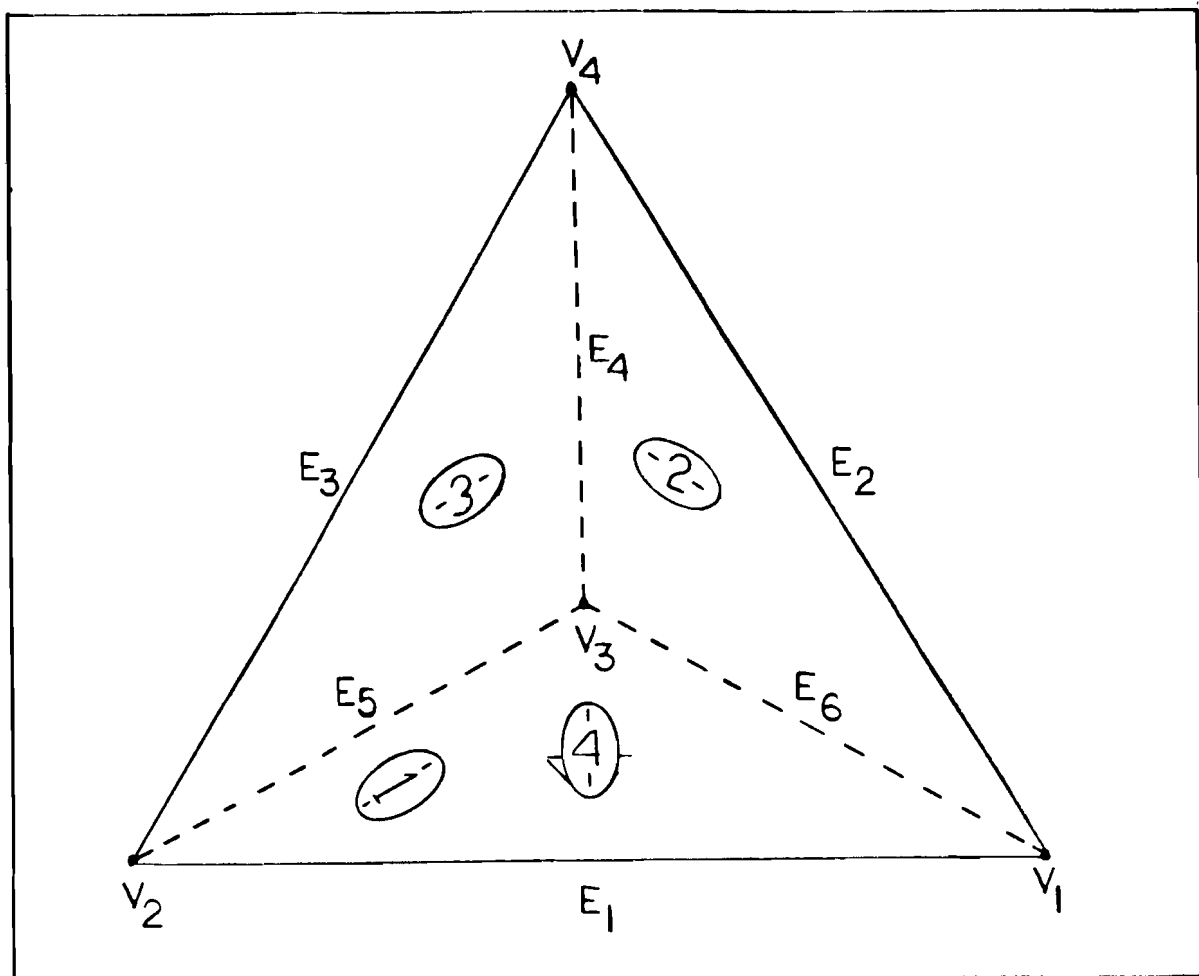


Figure 1.

Table I.

Surfaces
$S_1 : E_1, E_5, E_6$
$S_2 : E_2, E_4, E_6$
$S_3 : E_3, E_4, E_5$
$S_4 : E_1, E_2, E_3$

Table II.

Vertices
$v_1 : (x_1, y_1, z_1)$
$v_2 : (x_2, y_2, z_2)$
$v_3 : (x_3, y_3, z_3)$
$v_4 : (x_4, y_4, z_4)$

Table III.

Edges
$E_1 : (v_1, v_2)$
$E_2 : (v_1, v_4)$
$E_3 : (v_2, v_4)$
$E_4 : (v_3, v_4)$
$E_5 : (v_2, v_3)$
$E_6 : (v_1, v_3)$

defining each edge.

All information about the polyhedral model can be derived from the polygon surface and vertex tables. However without the edge table, the model would have to be processed using the polygon surface table causing some edges to be processed twice. Listing the data in three tables provides an efficient way for the computer to access the major components (vertices, edges, and polygon surfaces) of a polyhedron. The edge table could also include pointers into the polygon surface table so that common edges between surfaces could be found more rapidly [6, pg. 191].

Data Verification

As the complexity of the polyhedral model increases, so does the possibility for the distortion of the model due to errors in the representation tables. The following five tests can be used to help check for the consistency and completeness of the data in the representation tables [6, pg. 192].

- 1.) Make sure that every vertex is listed as an endpoint for at least two edges.
- 2.) Verify that every edge is part of at least one polygon.
- 3.) Check to see that every polygonal surface is closed.
- 4.) Verify that each polygonal surface has

at least one shared edge.

- 5.) If the edge table contains pointers to the polygon surface table, make sure that the polygon surfaces actually share those common edges.

The author also suggests an additional way to help check for the consistency of the data in the representation tables:

- 6.) If the polyhedron is convex, then the the number of vertices: v , number of edges: e , and number of faces: f , must satisfy the **Euler Descartes** formula: $v - e + f = 2$.

CHAPTER III

BASICS OF THREE-DIMENSIONAL GRAPHICS

The focus of this chapter is to discuss and explain the process in which different views of a three-dimensional object modeled by a polyhedron can be displayed on a two-dimensional view screen. The process is the same for all polyhedra and involves projecting the edges onto a flat surface called the projection plane. Before this projection process takes place, the eye coordinates (coordinates relative to the viewer's eye) for each vertex of the polyhedron must be calculated.

The Viewpoint

The location of the viewer's eye relative to the object is commonly referred to as the viewpoint. The viewpoint will be identified by spherical coordinates (**Theta**, **Phi**, **Rho**). Imagine that a line is drawn from the origin to the viewpoint. The third parameter **Rho** represents the distance along this line. The first parameter **Theta** represents the angle that the plane formed by the line and the Z axis makes with the plane formed by the X and Z axes. The second parameter **Phi** represents the angle that the line makes with the Z axis. Increasing **Rho** will have the effect of moving the viewer farther away from the object, while decreasing **Rho** will have the opposite effect. The direction from which the viewer will see the object can easily be changed by altering the values of **Theta** and or **Phi**.

The Eye Coordinate System

It will prove to be more convenient for projection and hidden line removal, to think of an object's vertices in terms of coordinates relative to the eye instead of coordinates relative to the standard coordinate axes. For this reason the eye coordinates of an object's vertices need to be calculated.

The eye coordinate system, $[X_e:Y_e:Z_e]$ consists of three mutually perpendicular axes intersecting at the viewpoint and is shown in figure 2. The eye coordinate system is always orientated so that the positive Z_e axis points towards the origin of the standard coordinate system [8, pg. 139]. The positive X_e axis points to the viewer's right, and the positive Y_e axis points upward. Also note that the eye coordinate system is a left-handed system. The vertices of a polyhedral object can be represented by coordinates relative to the standard coordinate system, or by coordinates relative to the eye coordinate axes.

Transferring coordinates relative to the standard axes system to coordinates relative to the eye coordinate system is accomplished by a sequence of four transformations [8, pg. 141]. The intermediate axes systems are each referred to as $[X':Y':Z']$. Note that for scaling and rotation a (3 x 3) transformation matrix is all that is actually required. However to make matrix multiplication compatible between the (4 x 4) translation matrix **A** and the (3 x 3)

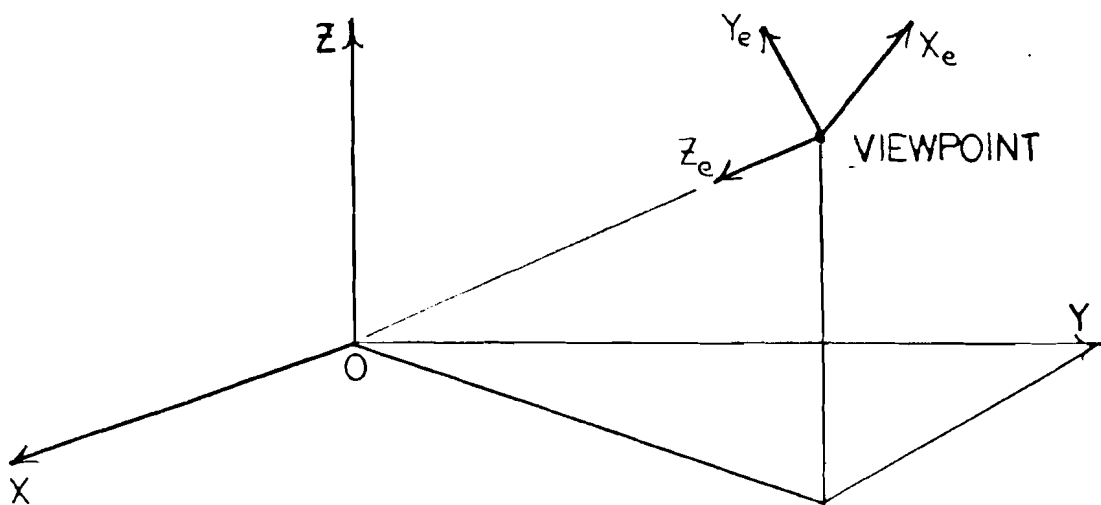


Figure 2.

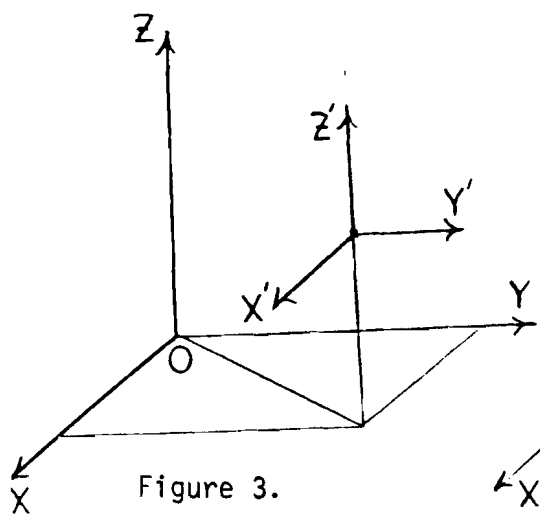


Figure 3.

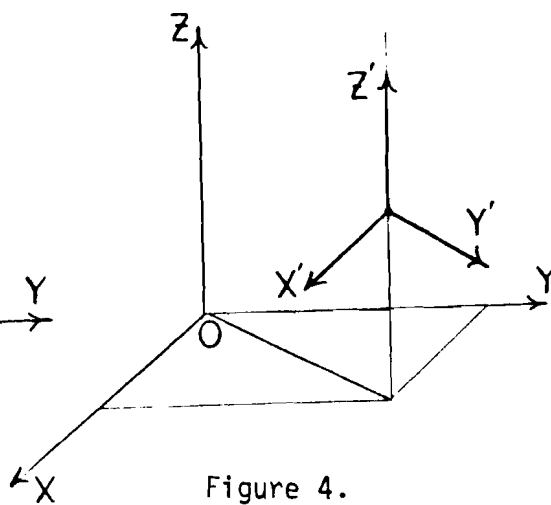


Figure 4.

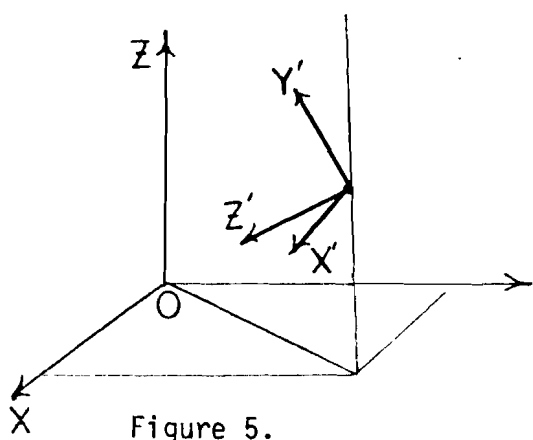


Figure 5.

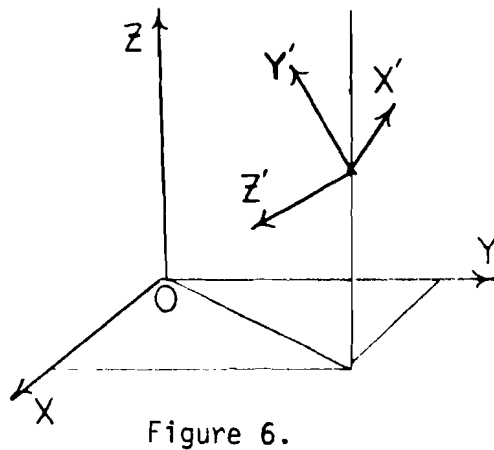


Figure 6.

rotation and scaling matrices **B**, **C**, and **D**, an extra row and column have been added.

1.) Translate origin to (**Theta**, **Phi**, **Rho**):

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\text{Rho} & -\text{Rho} & -\text{Rho} & 1 \\ \cos(\text{Theta}) & \sin(\text{Theta}) & \cos(\text{Phi}) & \\ \sin(\text{Phi}) & \sin(\text{Phi}) & & \end{bmatrix}$$

Refer to figure 3.

2.) Rotate through ($90 - \text{Theta}$) clockwise about the Z' axis:

$$\mathbf{B} = \begin{bmatrix} \sin(\text{Theta}) & \cos(\text{Theta}) & 0 & 0 \\ -\cos(\text{Theta}) & \sin(\text{Theta}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Refer to figure 4.

3.) Rotate through ($180 - \text{Phi}$) clockwise about the X' axis:

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -\cos(\text{Phi}) & -\sin(\text{Phi}) & 0 \\ 0 & \sin(\text{Phi}) & -\cos(\text{Phi}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Refer to figure 5.

4.) Convert to left-hand system:

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

Refer to figure 6.

The matrix product ABCD is given below:

$$\begin{bmatrix} -\sin(\Theta) & -\cos(\Theta) \cos(\Phi) & -\cos(\Theta) \sin(\Phi) & 0 \\ \cos(\Theta) & -\sin(\Theta) \cos(\Phi) & -\sin(\Theta) \sin(\Phi) & 0 \\ 0 & \sin(\Phi) & -\cos(\Phi) & 0 \\ 0 & 0 & \text{Rho} & 1 \end{bmatrix}$$

If the standard coordinates (x,y,z) of a vertex are known, the eye coordinates of that vertex may be obtained through the matrix product $(x,y,z,1) ABCD$. Note that since the matrix product ABCD is a (4×4) matrix, a dummy fourth coordinate must be attached.

Projection

Up until now all that has been accomplished is to convert the coordinates of the object's vertices relative to the standard axes system into coordinates relative to the eye axes system. The final step of getting a representation of the object that can be shown on a two-dimensional computer screen is to actually project the edges of the object onto the projection plane. Referring to figure 7, the points on the edge AB when projected form the line segment A'B' on the projection plane. This type of projection is known as a perspective. Such a projection is popular because it is very similar to the way that images are formed by the human eye and by lenses on photographic film [10, pg. 295]. Perspective projection conveys more depth information than other types of projection. This is because distant objects will appear smaller than the nearer ones under

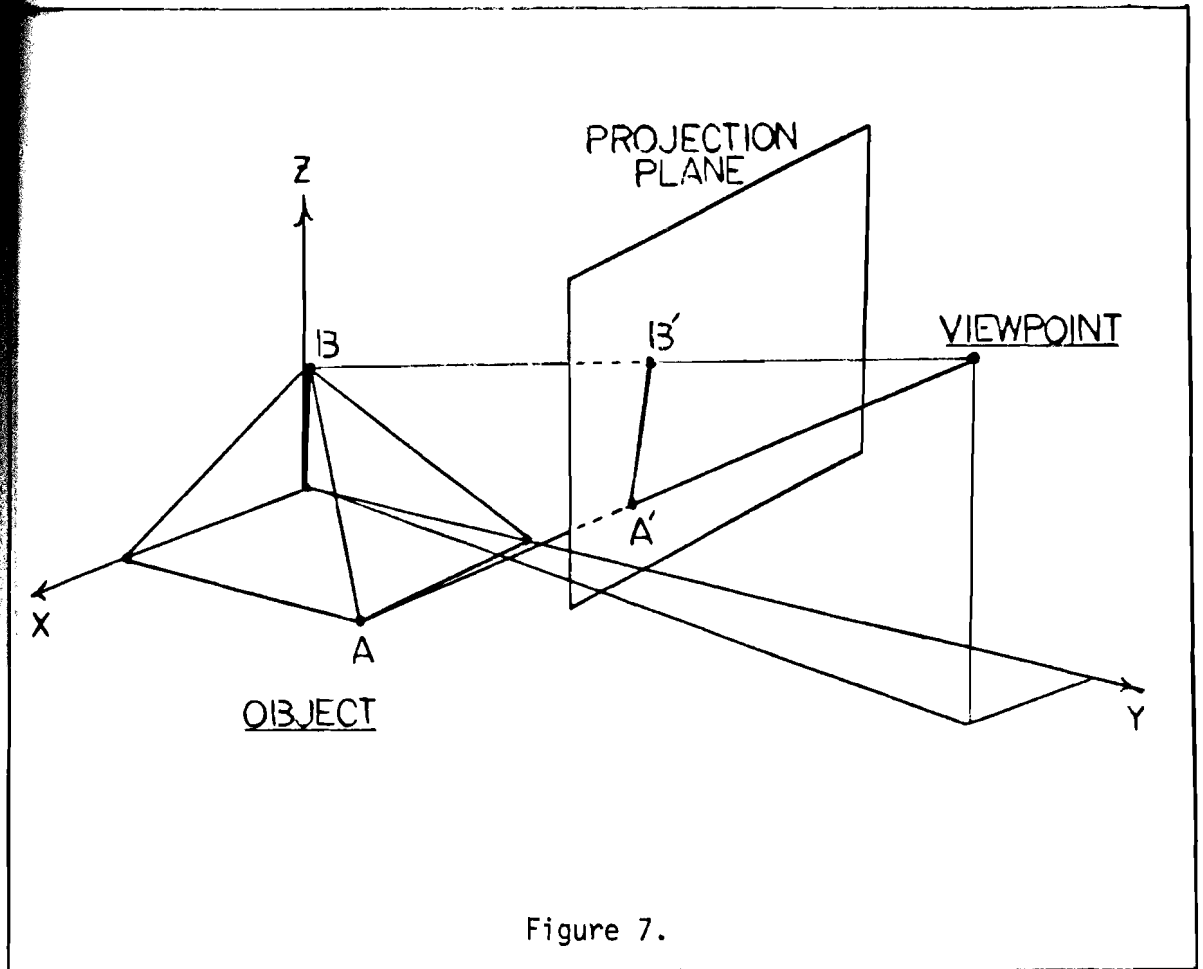


Figure 7.

this type of projection.

Refer to figure 8, point P which could represent a vertex on the object projects to P', a point on the projection plane. The X_e and Y_e coordinates of P' are called the screen coordinates of the point P. Given any point (x, y, z) relative to the eye coordinate axes, the screen coordinates (S_x, S_y) can be calculated [8, pg. 137]. Again referring to figure 8, right triangles OBA and ODC are similar triangles. Notice also that right triangles OBF and ODE are similar.

Therefore:

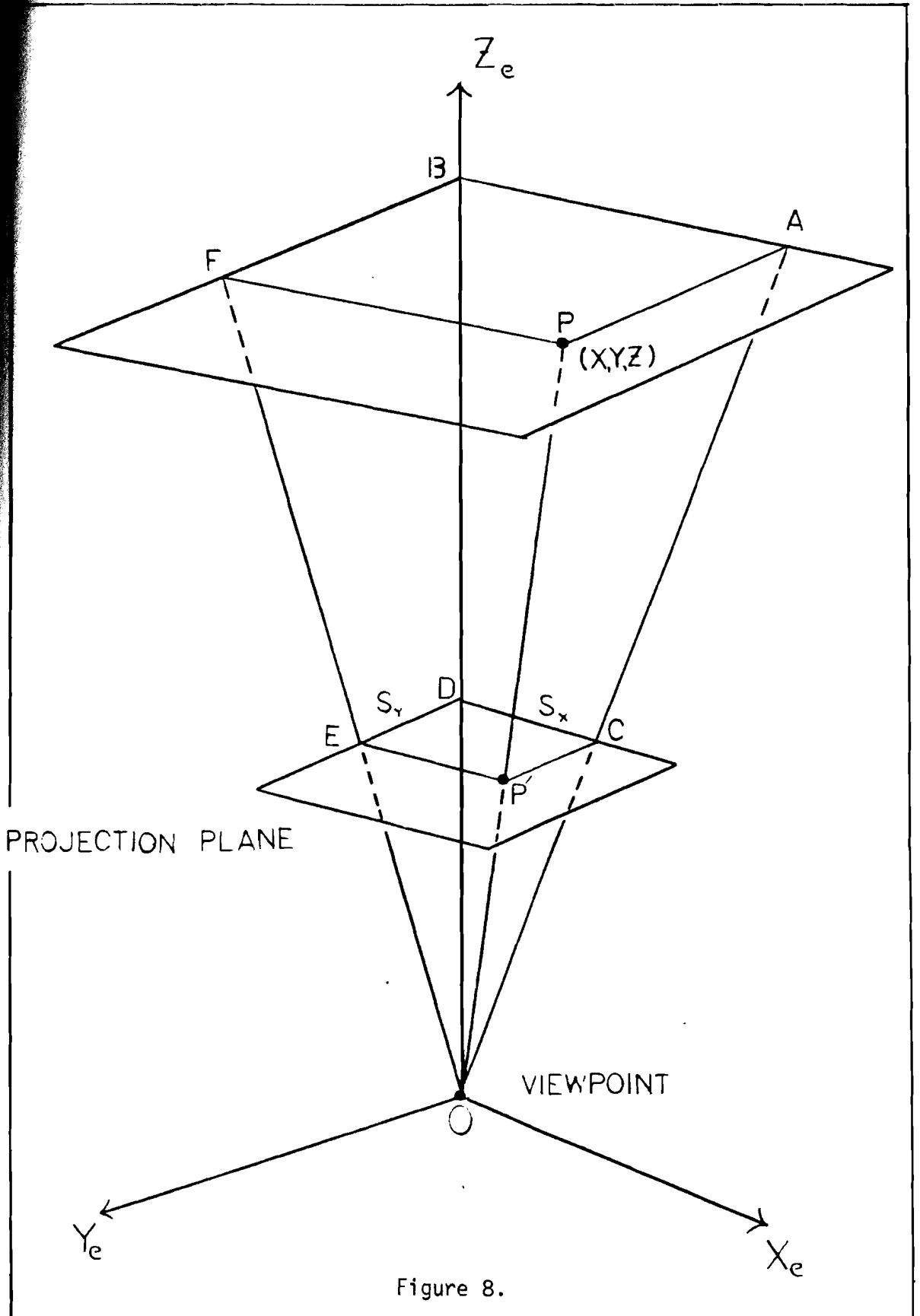


Figure 8.

$$DC/OD = BA/OB \quad \text{and} \quad DE/OD = BF/OB.$$

By substitution:

$$S_x/D = X_e/Z_e \quad \text{and} \quad S_y/D = Y_e/Z_e.$$

Solving for S_x and S_y :

$$S_x = D(X_e/Z_e) \quad \text{and} \quad S_y = D(Y_e/Z_e).$$

From the matrix product $(x,y,z,1)$ ABCD:

$$X_e = -x \sin(\Theta) + y \cos(\Theta)$$

$$Y_e = -x \cos(\Theta)\cos(\Phi) - y \sin(\Theta)\cos(\Phi) + z \sin(\Phi)$$

$$Z_e = -x \cos(\Theta)\sin(\Phi) - y \sin(\Theta)\sin(\Phi) - z \cos(\Phi) + Rho$$

Once the screen coordinates have been determined, the polyhedral model can be displayed on the graphics display device. However, the programmer should be aware of the coordinate system used by the particular display device he intends to use. First of all, the resolutions of the X and Y axes are typically not the same. If this fact is neglected, distorted images can occur. For example squares will appear as rectangles, or circles can appear as ellipses. To correct the problem, one of the coordinates of the points to be plotted is multiplied by a scaling factor (known as an aspect ratio) to compensate for the differences in the resolutions. Second, the origin is not located in the center, but typically is located in the upper left hand corner of the viewing area. Third, the Y

axis usually increases from top to bottom. Finally, the X and Y axes can only represent discrete integer quantities.

The Viewing Parameters

It is worthwhile to consider how changing the viewpoint parameters (Θ , Φ , ρ) and D (the distance of the projection plane from the viewpoint) affect the image generated on the graphics display device [8, pg. 146]. First of all by changing Θ and or Φ , views of the object from different angles can be generated. Second, the image size of the object can be controlled by changing ρ (the distance from the viewpoint to the $[X:Y:Z]$ origin). Increasing ρ will make the image appear smaller, while decreasing ρ will make the image appear larger. Changing the value of D is a second way of changing the image size of the object. Increasing D will enlarge the image size, while decreasing D will reduce the image size.

It is necessary to have two parameters ρ , and D to control the image size [8, pg. 146]. Increasing ρ will decrease the effect of perspective, but the object's image size will appear smaller. To compensate for the smaller image size, increase the value of D . Decreasing ρ will increase the effects of perspective, but the object's image size will become larger. To compensate for larger image size, decrease the value of D .

CHAPTER IV

HIDDEN LINE REMOVAL

It is relatively easy to display a three-dimensional object on a two-dimensional graphics screen. However, in order to generate a truly realistic image, the line segments and surfaces which are not visible to the viewer must be identified and removed. Many algorithms for hidden line removal exist, some simple and some very sophisticated. This thesis will discuss three elementary algorithms that will correctly remove all hidden lines from any object that can be modeled as a polyhedron.

Algorithm Number One (Back Face Removal)

The following algorithm removes the hidden lines from an object by eliminating the back surfaces [8, pg. 156]. Appendix E contains a complete source listing written in Turbo Pascal for this algorithm. Refer to Appendix D for instructions on how to use the programs contained in this thesis.

Assumptions

The back face removal algorithm operates under the following assumptions about the object:

- 1.) The object being processed is modeled by a **convex** polyhedron.
- 2.) The polyhedron is constructed in such a way that the viewer cannot see the interior of the object from any viewpoint.

- 3.) There are no obstructions in the line of sight from the viewpoint to the object.

Theory Of Operation

As one looks at a convex polyhedron, the visible surfaces are the ones facing the viewer. This is because light traveling from these surfaces has an unobstructed path to the viewer's eyes. The other surfaces, the ones that are not visible, are called back surfaces. These back surfaces are facing away from the viewer, and the light from these surfaces is blocked from reaching the viewer's eyes by other surfaces.

Two normal vectors can be associated with each surface of a polyhedron. One normal vector points outward away from the polyhedron; the other points inward. The outward normal vector will be used as the orientation vector for each surface, and is denoted by \mathbf{N} . The outer normal vector \mathbf{N} is given by: $\mathbf{N} = (v_2 - v_1) \times (v_3 - v_1)$ where $v_1, v_2,$ and v_3 are any three properly ordered vertices belonging to the surface. The proper ordering of the vertices of a surface is very important and will be discussed later. A second vector \mathbf{W} the line of sight vector will be associated with each surface. This vector \mathbf{W} is directed from a vertex on the surface to the viewpoint. For each surface, β represents the angle between \mathbf{W} (the line of sight vector) and \mathbf{N} (the surface orientation vector). The visibility of a given surface is determined by the following:

- 1.) If $0^\circ \leq \text{beta} \leq 90^\circ$ (ie. $\mathbf{W} \cdot \mathbf{N} > 0$), the surface is facing the viewer and should be displayed.
- 2.) If $90^\circ \leq \text{beta} \leq 180^\circ$ (ie. $\mathbf{W} \cdot \mathbf{N} \leq 0$), the surface is facing away from the viewer and should not be displayed.

Care must be taken in how the vertices of each surface are labeled in order for the outer normal orientation vectors to be calculated correctly [8, pg. 159]. Referring to the octahedron in figure 9, the identification of the first vertex of each surface is completely arbitrary. However once this identification has been made, it is vital that the listing of the remaining vertices continue in a counterclockwise direction as viewed from the outside of the object (in this case the octahedron). Table IV gives one example of how the surfaces of the octahedron could be oriented.

Table IV.

Surface Orientation	
$S_1 : v_{01}, v_{02}, v_{03}, v_{04}$	$N_1 = (v_{02} - v_{01}) \times (v_{03} - v_{01})$
$S_2 : v_{04}, v_{03}, v_{06}, v_{05}$	$N_2 = (v_{03} - v_{04}) \times (v_{06} - v_{04})$
$S_3 : v_{07}, v_{08}, v_{05}, v_{06}$	$N_3 = (v_{08} - v_{07}) \times (v_{05} - v_{07})$
$S_4 : v_{10}, v_{09}, v_{08}, v_{07}$	$N_4 = (v_{09} - v_{10}) \times (v_{08} - v_{10})$
$S_5 : v_{11}, v_{12}, v_{09}, v_{10}$	$N_5 = (v_{12} - v_{11}) \times (v_{09} - v_{11})$
$S_6 : v_{11}, v_{02}, v_{01}, v_{12}$	$N_6 = (v_{02} - v_{11}) \times (v_{01} - v_{11})$
$S_7 : v_{02}, v_{04}, v_{05}, v_{08}, v_{09}, v_{12}$	$N_7 = (v_{04} - v_{02}) \times (v_{05} - v_{02})$
$S_8 : v_{02}, v_{11}, v_{10}, v_{07}, v_{06}, v_{03}$	$N_8 = (v_{11} - v_{02}) \times (v_{10} - v_{02})$

If, instead, the vertices are given in a clockwise direction as viewed from outside the object, then the

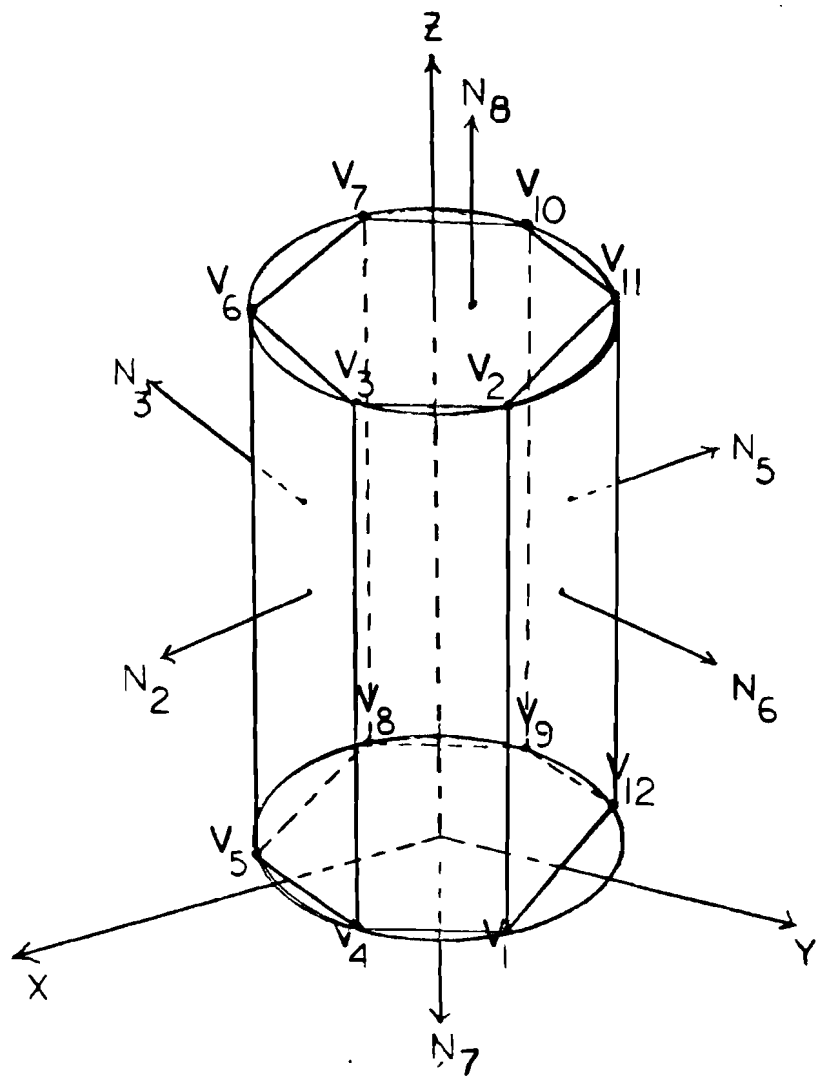


Figure 9.

orientation vector N will be pointing inward, and undesired results will occur when using this hidden line removal algorithm.

Figure 10 gives an example of a concave polyhedron in which algorithm number one will not correctly remove all of the hidden lines. All of the back surfaces will be correctly identified and removed. The problem with concave polyhedra is that the front surfaces are not always completely visible. For example, the surface PQR is facing the viewer but is not completely visible.

Figure 11 gives an example of a convex polyhedron in which algorithm number one will also not correctly remove all of the hidden lines. The hexahedron is constructed in such a way that the viewer can see the interior of the object from certain viewpoints. Consider the surfaces $ABCD, BCGH, ABFG, EFGH, ADEF$ to all consist of opaque materials while the surface $CDEH$ consists of a transparent material. Algorithm number one will incorrectly identify surfaces $EFGH$ and $ADEF$ as being back surfaces (surfaces that are completely hidden from view).

Algorithm Number Two (Clipping Lines Edges Surfaces)

The following algorithm removes hidden lines by comparing all edges of the object with each surface [5, pg. 230]. Appendix F contains a complete source listing written in Turbo Pascal for this algorithm.

Assumptions

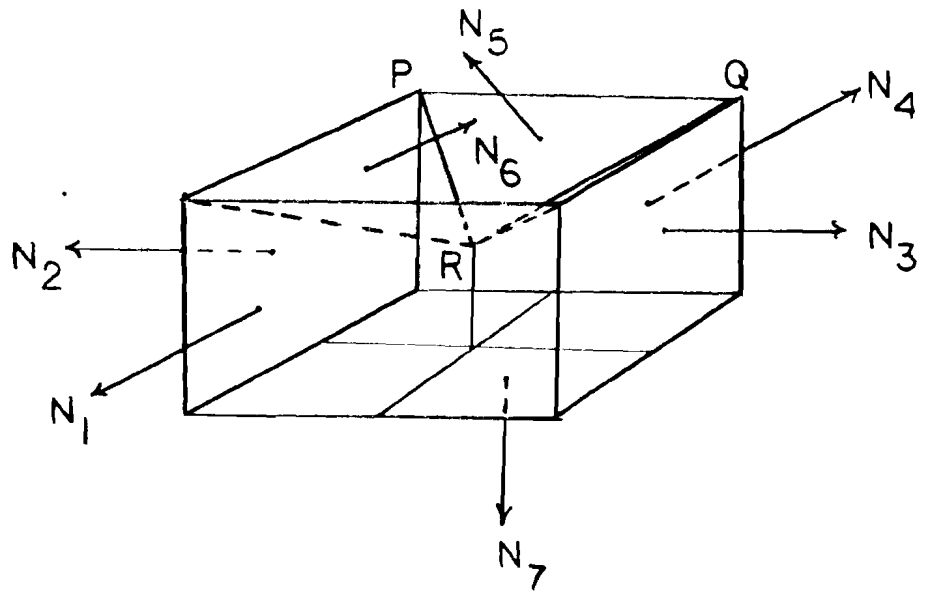


Figure 10.

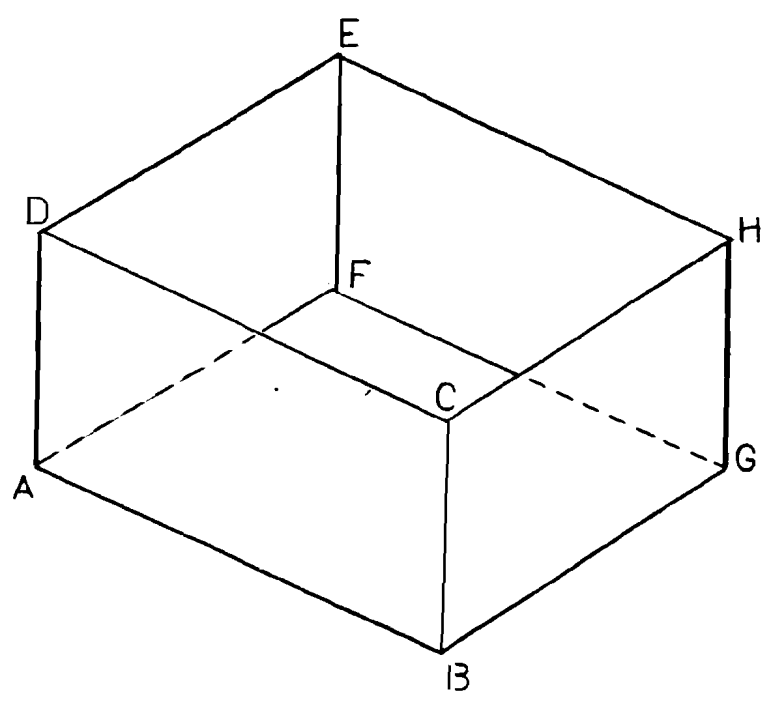


Figure 11.

The edge clipping algorithm operates under the following assumptions about the object:

- 1.) The object being processed is modeled by a concave or convex polyhedron.
- 2.) The surfaces that make up the polyhedron are all convex polygons.
- 3.) The polyhedron is constructed in such a way that the viewer can possibly see the interior of the object from certain viewpoints.
- 4.) There are no obstructions in the line of sight from the viewpoint to the object.

Theory Of Operation

Before the algorithm begins, all edges of the polyhedron are initialized or marked as being visible to the viewer. Each surface is then taken one at a time, and all edges defining the other surfaces are compared one by one to this surface to determine if the visibility of the edges are blocked. Any portion of the edges that are blocked are marked as being erased. After all the edges have been processed with respect to each surface, all visible portions of the object are then drawn.

Given an edge PQ and a polygonal surface $A_1A_2 \dots A_n$ the following four steps will determine if the visibility of the edge is blocked by that particular surface, and if so, exactly how much of the edge PQ will have to be erased.

- 1.) If both endpoints P and Q of the edge are in front of the actual surface $A_1A_2 \dots A_n$, then the edge is completely visible relative to the surface. No further testing of the edge is necessary with respect to this surface. Refer to figure 12.

- 2.) If the projected edge $P'Q'$ is completely exterior to the projected surface $A'_1A'_2 \dots A'_n$ then the edge PQ is completely visible relative to the surface. No further testing of the edge is necessary with respect to this surface. Refer to figure 13.

- 3.) If the projected edge coincides with any of the projected surface edges $A'_1A'_2, A'_2A'_3, \dots, A'_{n-1}A'_n$, then the edge should not be erased. No further testing of the edge is necessary with respect to this surface. Refer to figure 14.

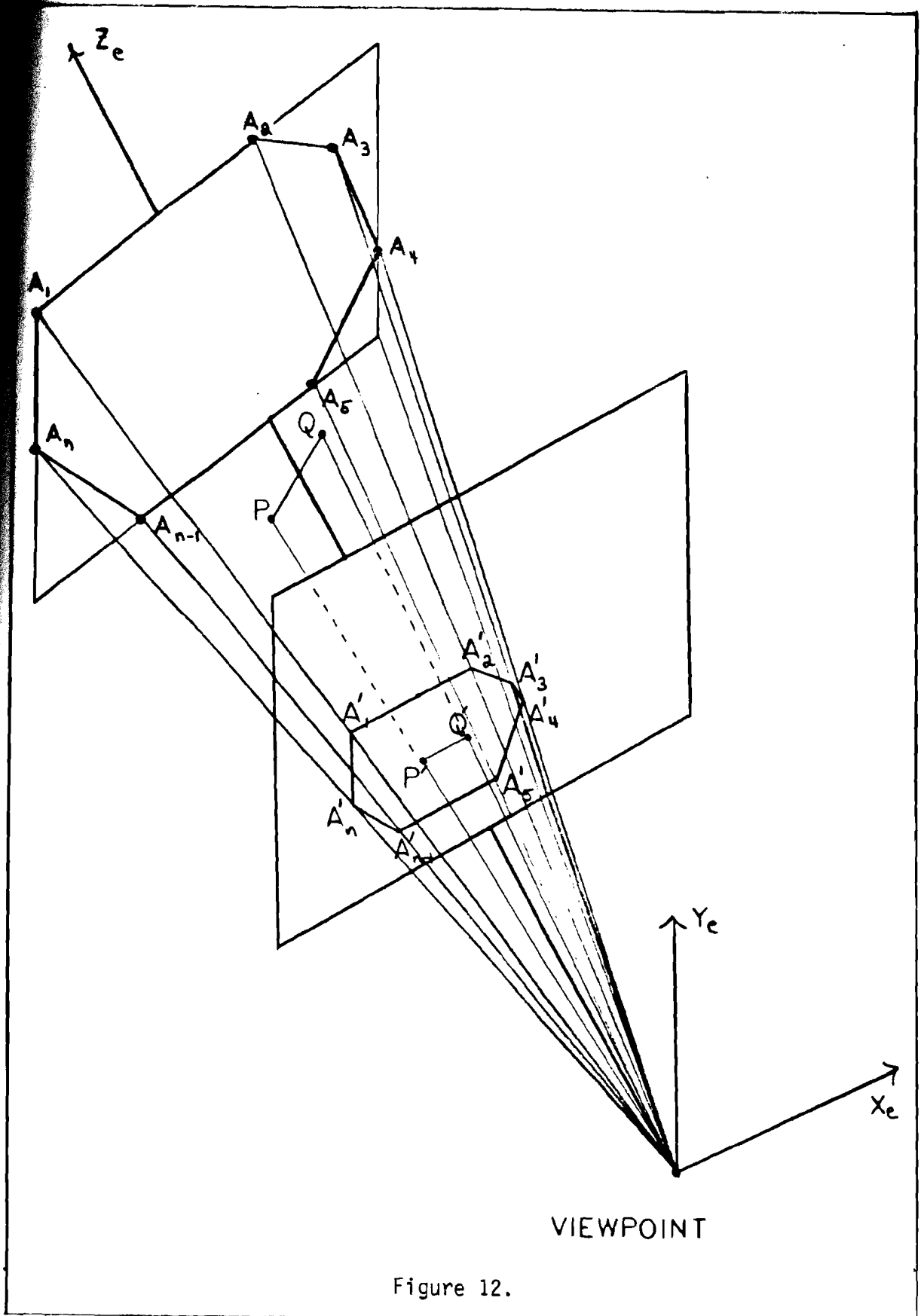


Figure 12.

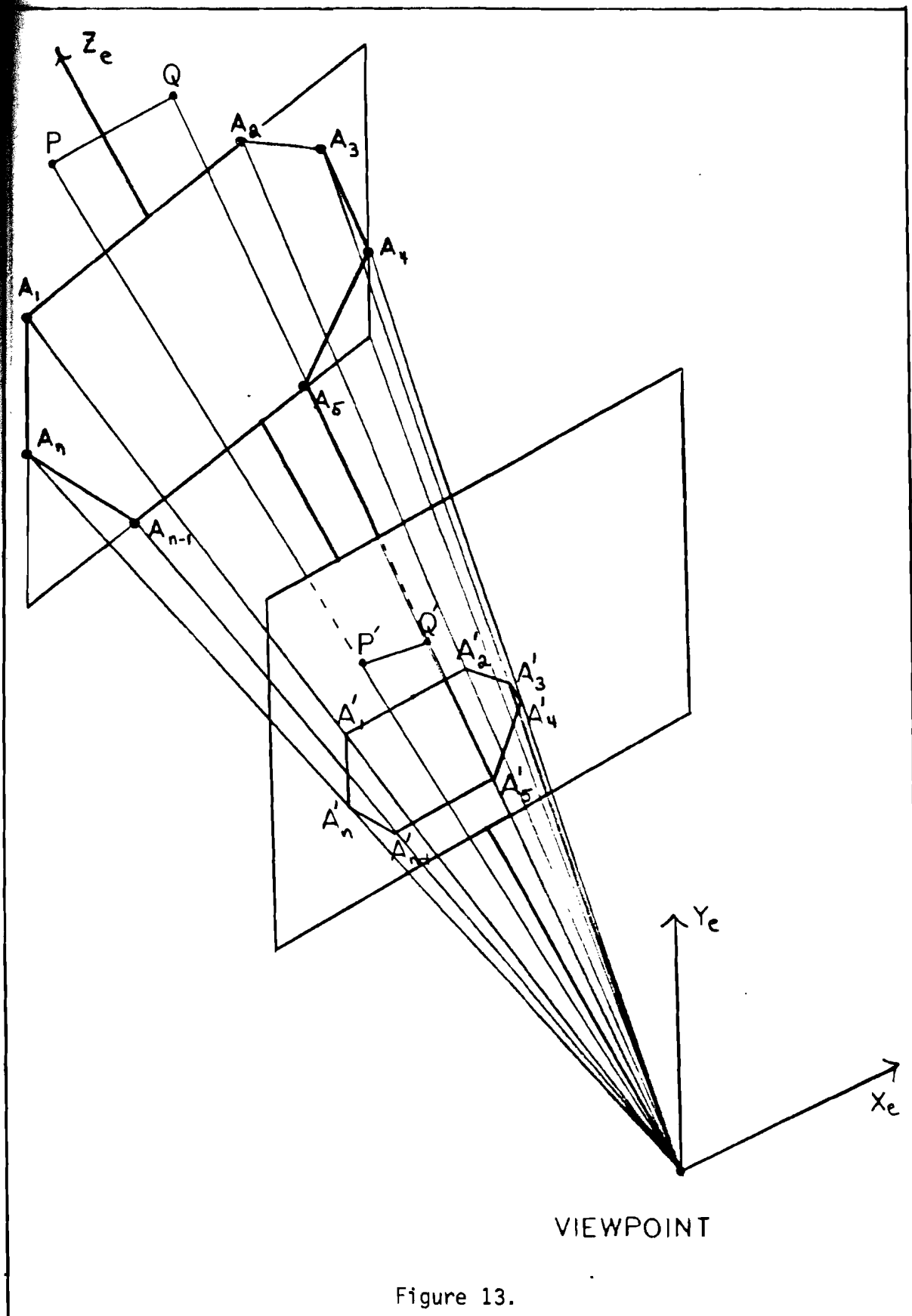
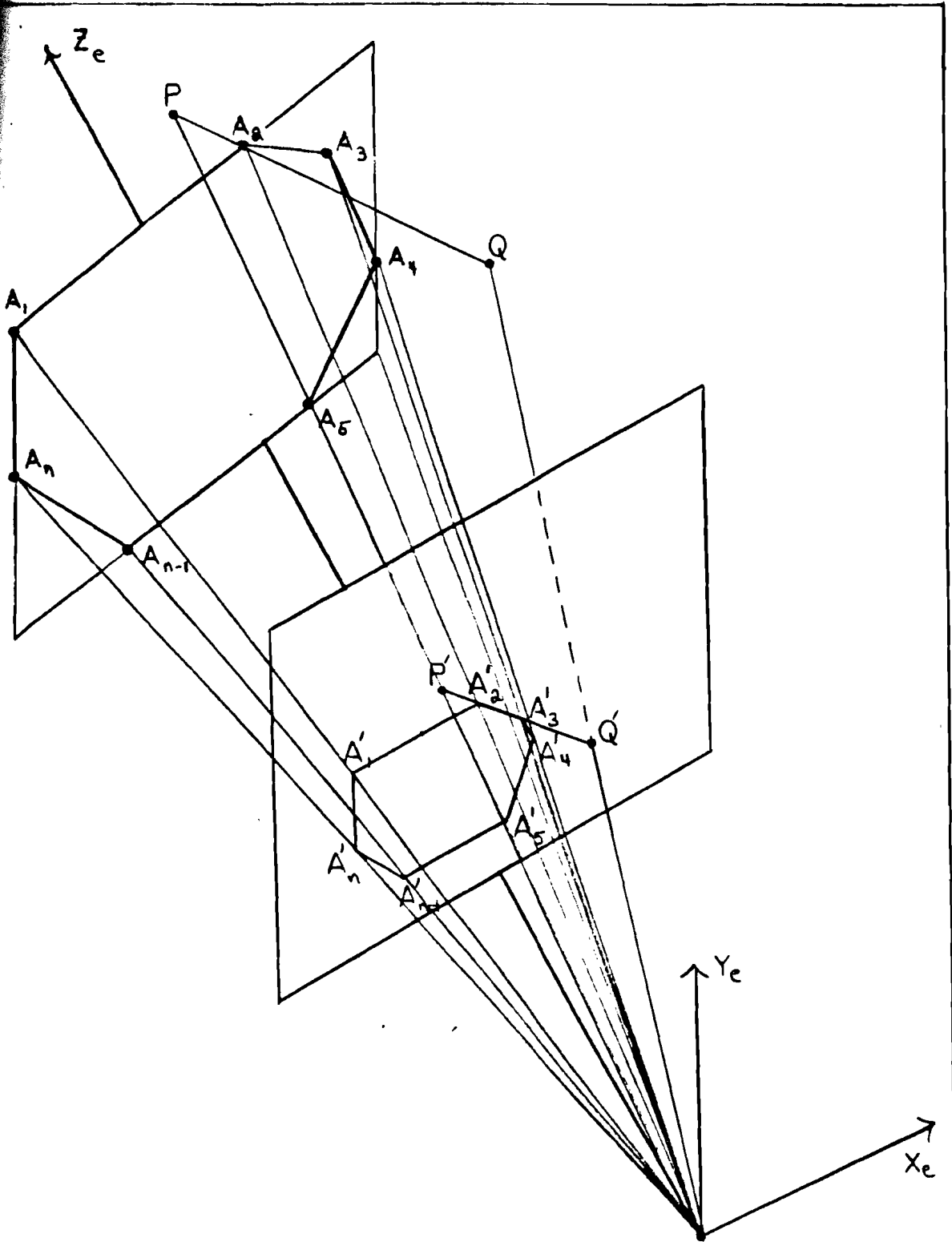


Figure 13.



VIEWPOINT

Figure 14.

4.) Find any intersection points where the projected edge intersects the projected surface edges. Remember that the intersection points of the line segments defined by the edges is desired rather than the intersection points of the infinite lines defined by the edges. Refer to appendix A for an explanation of how to find these intersection points. Since by assumption all surfaces are convex polygons, there will be at most two intersection points.

a.) Zero intersection points.

If the midpoint M' of the projected edge is exterior to the projected surface then the projected edge is completely exterior to the projected surface and therefore the actual edge is completely visible with respect to the surface. If M' is interior to the projected surface then the projected edge lies completely interior to the projected surface. Since at least one endpoint of the actual edge is behind the actual surface, the actual edge

lies completely behind the surface and therefore should be erased. Refer to figure 15.

b.) One intersection point.

The pre-image of a point R located on the projected edge $P'Q'$ is defined to be the point located on the actual edge which projects onto R . Determine which endpoint P' or Q' of the projected edge is interior to the projected surface. If the pre-image of this endpoint is behind the actual surface then erase the portion of the projected edge from the intersection point I'_1 to the projected endpoint located interior to the projected surface. Refer to figure 16. If neither endpoint P' or Q' is interior to the projected surface then the projected edge lies completely exterior to the projected surface and should not be erased. Refer to figure 17.

c.) Two intersection points.

If both endpoints P and Q of

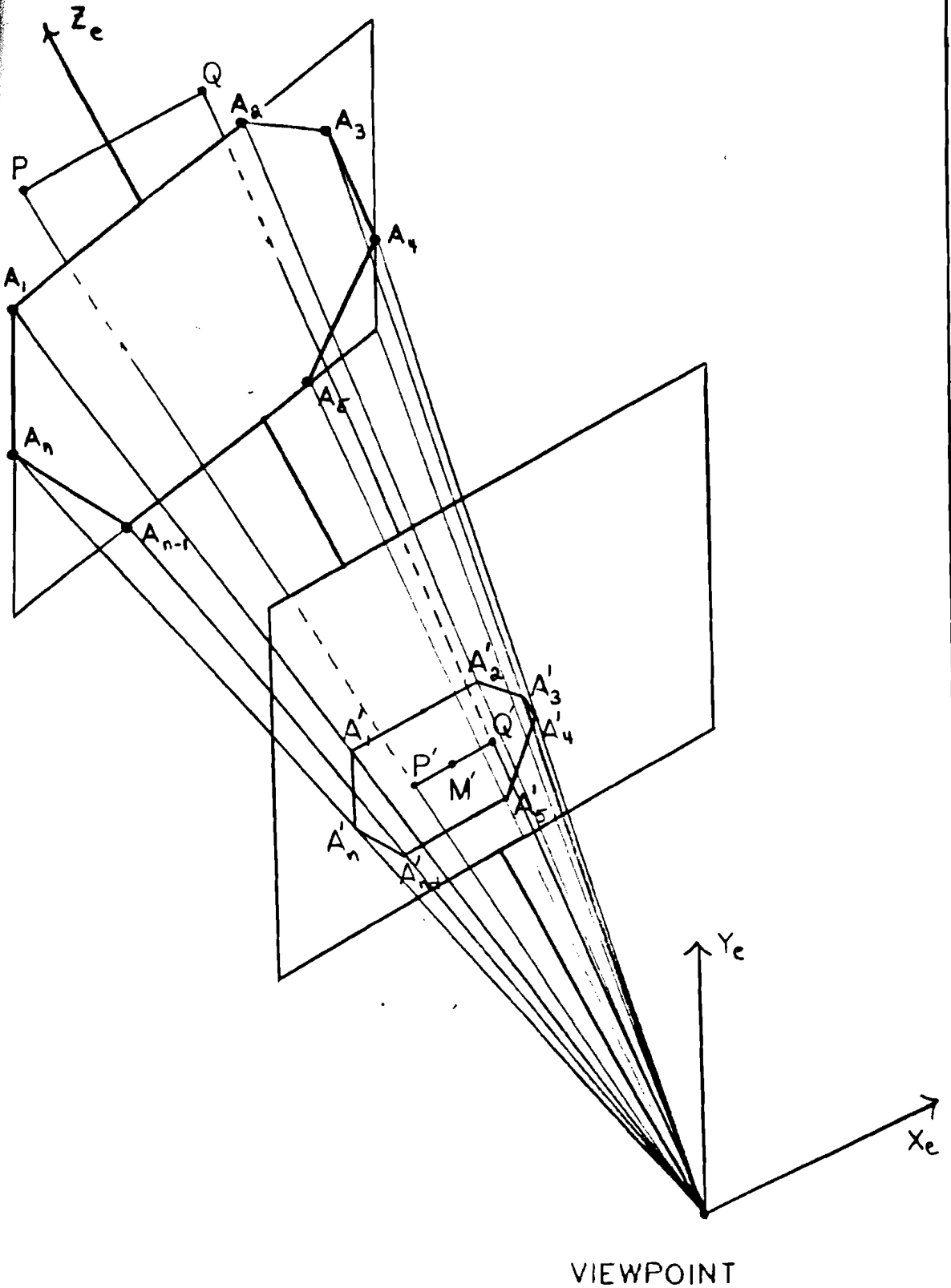


Figure 15.

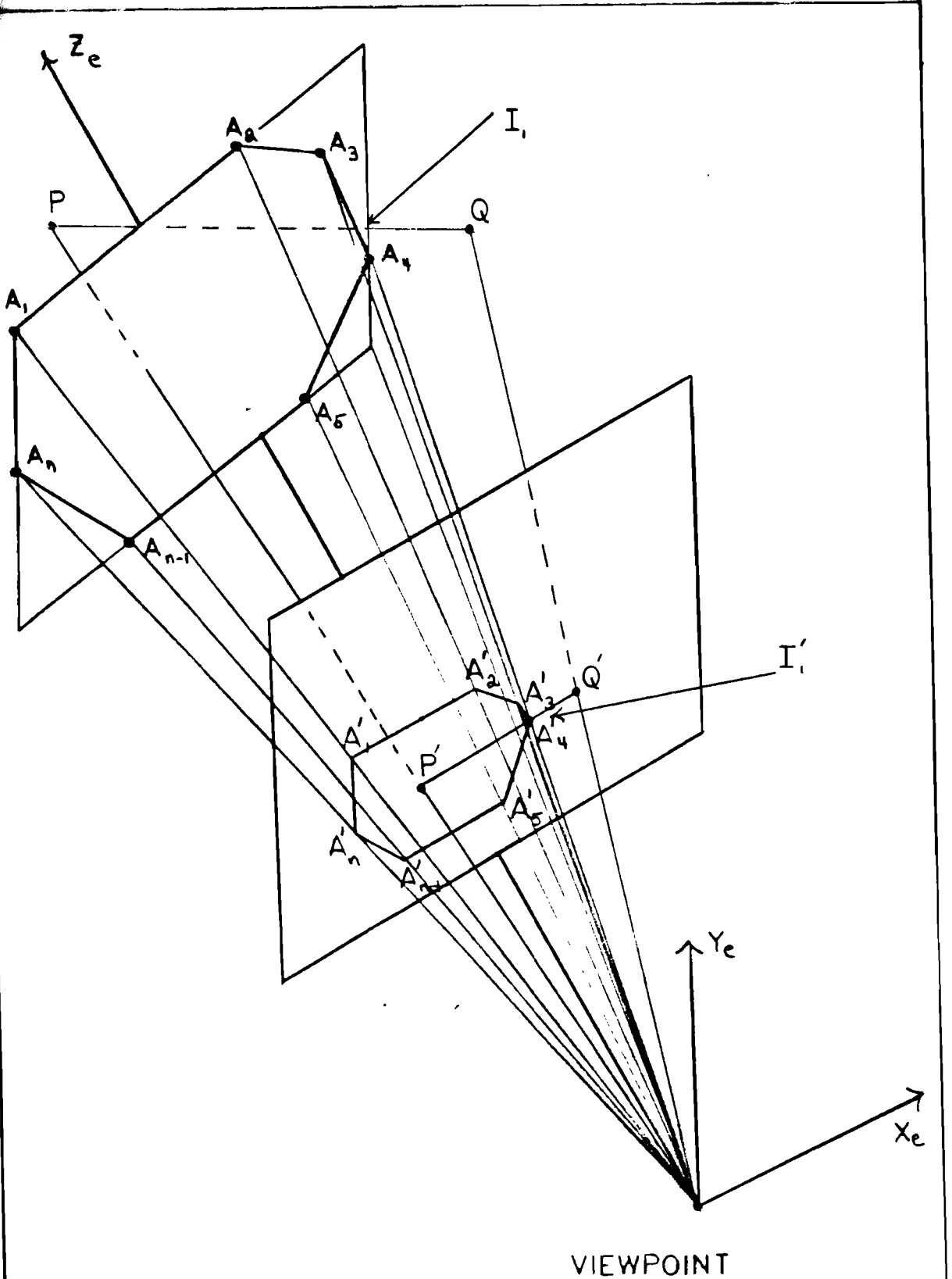


Figure 16.

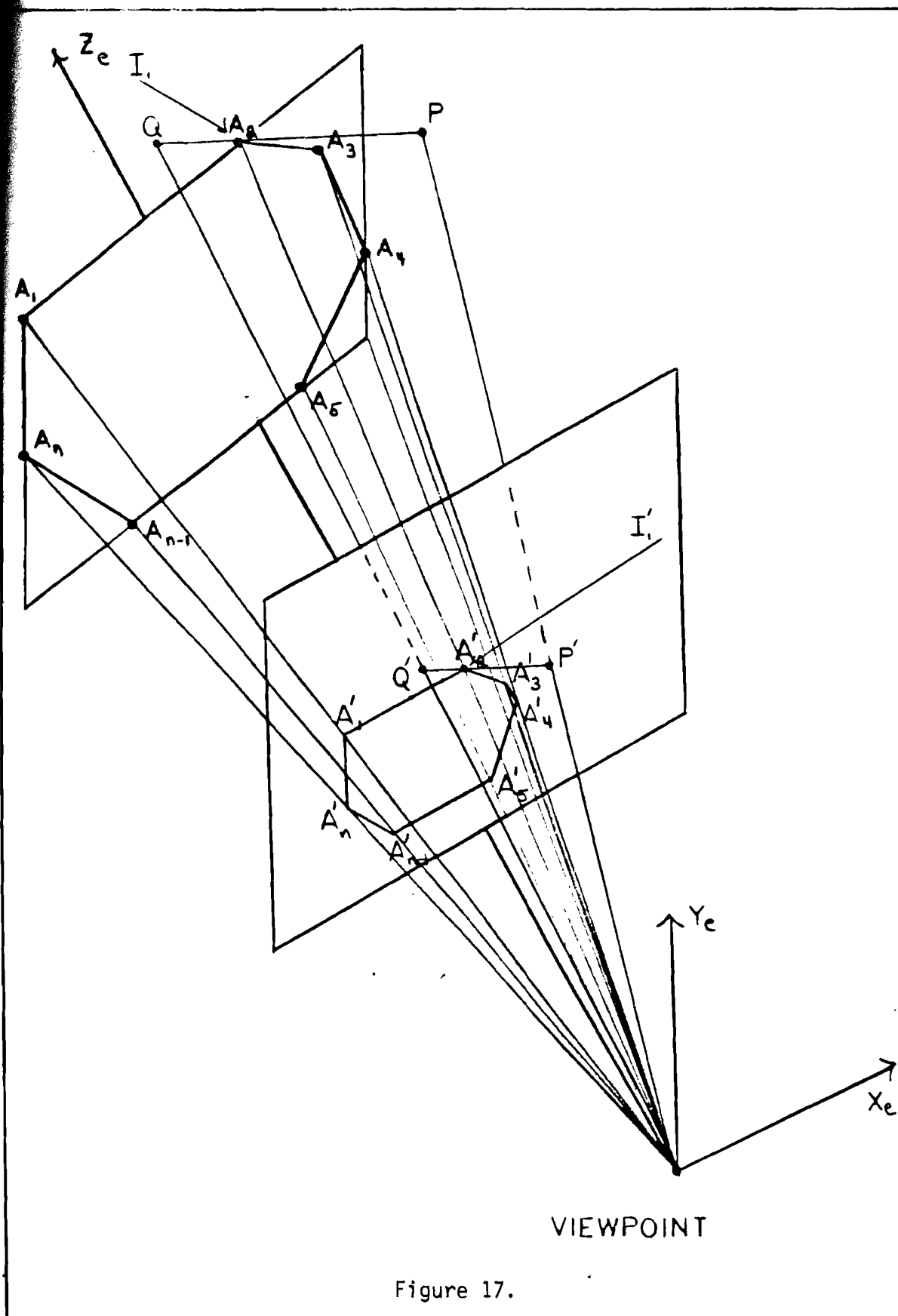


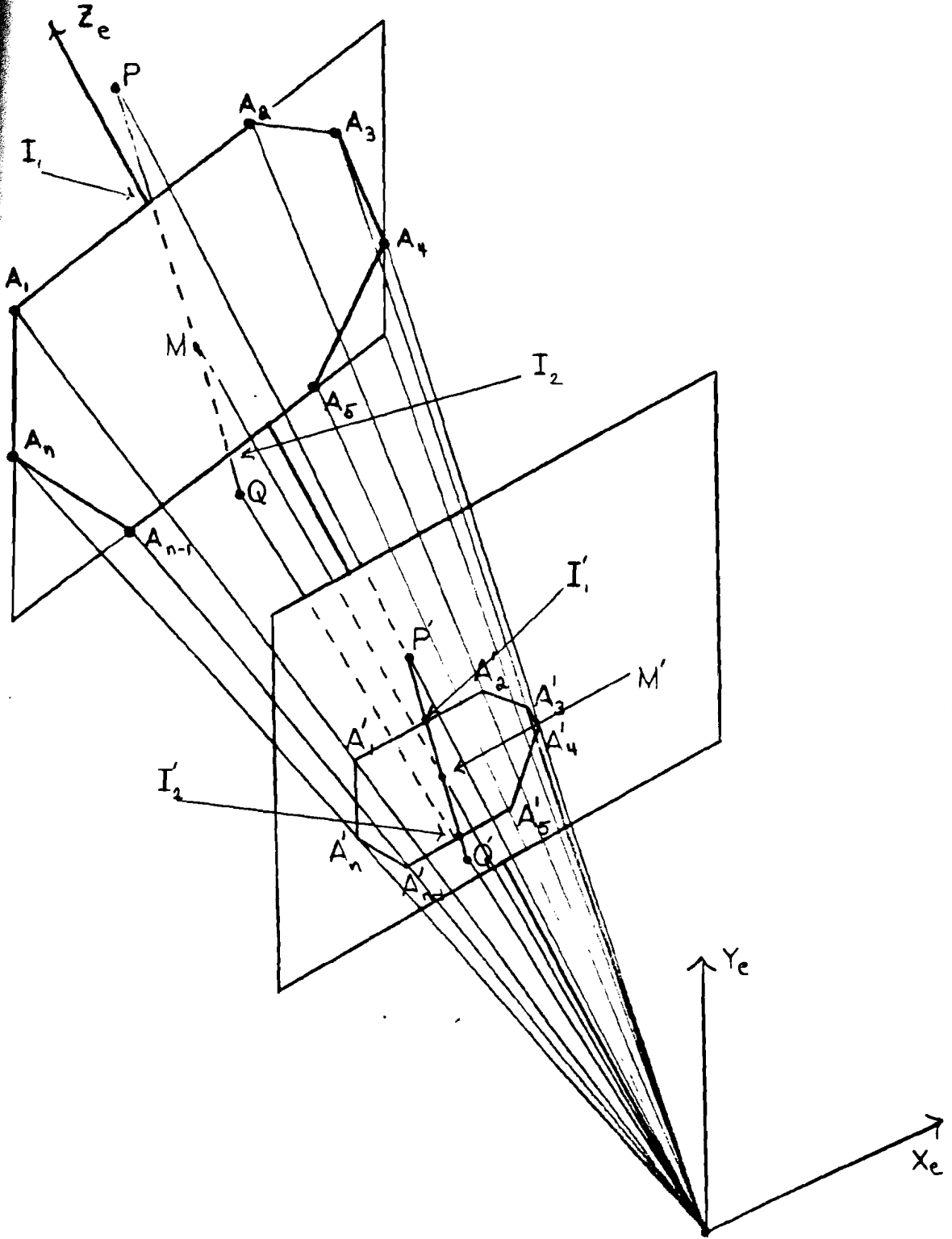
Figure 17.

the actual edge are behind the actual surface then erase the portion of the projected edge from I'_1 to I'_2 . If endpoints P and Q are not both behind the actual surface then let M' be the midpoint of the projected edge $P'Q'$. If M is behind the actual surface then erase the portion of the projected edge located between the intersection points I'_1 and I'_2 . Refer to figure 18. If M is in front of the actual surface, then the actual edge is completely visible with respect to the surface and should not be erased.

Refer to appendices B and C for information on how to determine whether or not a given point P is located in front or behind a surface and if the point P is located in the interior of a plane polygon.

Algorithm Number Three

The following hidden line removal algorithm is a combination of algorithms one and two. Appendix G contains a complete source listing written in Turbo Pascal for this algorithm.



VIEWPOINT

Figure 18.

Assumptions

Algorithm number three operates under the following assumptions about the object:

- 1.) The object being processed is modeled by a polyhedron.
- 2.) The polyhedron is constructed in such a way that the viewer cannot see the interior of the object from any viewpoint.
- 3.) The surfaces that make up the polyhedron are all convex polygons.
- 4.) There are no obstructions in the line of sight from the viewpoint to the object.

Theory Of Operation

All lines on the object are first marked as being hidden. Algorithm number one is then used to determine the surfaces that are facing the viewer. All edges on these front surfaces are marked as visible. Each of the front surfaces are taken one at a time, and all edges belonging to the other front surfaces are compared one by one (using the four steps outlined in algorithm number two) to this particular front surface to determine if the visibility of the edges are blocked by this surface. Any portion of the edges that are blocked by this surface are marked as erased. After all the edges have been processed with respect to each surface, all visible portions of the object are then drawn.

Line Processing

When implementing algorithms number two and three, in theory the entire object could have been drawn on the graphics display device including the hidden lines. Then the necessary line segments could be erased to produce the final image. Because of round-off errors in the calculation of intersection points and screen coordinates, lines are often not entirely erased. Thus leaving an unwanted trail of stray points in the final output of the object. Another disadvantage of this technique could not be used in connection with those graphics output devices which are unable to erase lines, such as plotters and printers. This section explains and outlines a technique for storing and processing the object in computer memory before displaying the final version of the object with the hidden lines removed.

To effect the task of storing and processing the object in computer memory, the edge table needs to be expanded to hold the necessary erasures to be performed on each particular edge. Refer to table V.

Table V.

Edge Table											
Edge No.	Vertex		Number Erasures	Erasur ^e 1		Erasur ^e 2		Erasur ^e 3		Erasur ^e 4	
	v_i	v_j		t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
0001	01	03	000000	--	--	--	--	--	--	--	--
0002	03	04	000003	.12	.13	.97	.99	.10	.11	--	--
0003	07	10	000002	.01	.50	.52	.87	--	--	--	--
0004	11	15	000001	.11	.87	--	--	--	--	--	--

The edge table is constantly being updated while hidden line removal algorithms two or three are executing. Every time a hidden line removal algorithm calls for a portion of a projected edge to be erased, the edge table is updated with this information. Finally after the hidden line algorithm has completed executing, the visible portions of the object are drawn using the information from the edge table.

The equation for each projected edge (v_i, v_j) of an object can be represented in the parametric form:

$$R(t) = (x_1, y_1) + t(x_2 - x_1, y_2 - y_1) , \quad 0 \leq t \leq 1$$

where (x_1, y_1) and (x_2, y_2) are the endpoints of the projected edge (v_i, v_j) . If a portion of the projected edge (v_i, v_j) needs to be erased, then the endpoints of the line segment to be erased, t_1 and t_2 are stored in the edge table. Note that t_1 and t_2 are the parameters associated with the parametric equation $R(t)$ representing the projected edge (v_i, v_j) .

As an example, let edge number two, namely (v_3, v_4) have endpoints (x_1, y_1) and (x_2, y_2) . Suppose that the portion of projected edge (v_3, v_4) from point P having coordinates (p_1, p_2) to point Q having coordinates (q_1, q_2) needs to be erased. Rather than store the four coordinates (p_1, p_2) and (q_1, q_2) of this segment, the parameters t_1 and t_2 that correspond to points P and Q respectively are stored in the edge table. Later on (p_1, p_2) and

(q_1, q_2) can be decoded from t_1 and t_2 by the following:

$$(P_1, P_2) = (x_1, y_1) + t_1(x_2 - x_1, y_2 - y_1)$$

$$(q_1, q_2) = (x_1, y_1) + t_2(x_2 - x_1, y_2 - y_1).$$

Remember that there is often round off error in the calculation of the endpoints P and Q to be erased, therefore P and Q are often very close to but not exactly located on the projected edge (v_3, v_4) . Approximate values for t_1 and t_2 can be obtained in the following way:

Let m denote the slope of the projected edge (v_3, v_4) .

1.) If $\text{Abs}(m) = 1$ then

$$k_1 = ((p_1 - x_1)/(x_2 - x_1) + (p_2 - y_1)/(y_2 - y_1))/2$$

$$k_2 = ((q_1 - x_1)/(x_2 - x_1) + (q_2 - y_1)/(y_2 - y_1))/2$$

$$t_1 = \text{Min} \{k_1, k_2\} \quad \text{and} \quad t_2 = \text{Max} \{k_1, k_2\}$$

2.) If $\text{Abs}(m) < 1$ then

$$k_1 = (p_1 - x_1)/(y_2 - y_1)$$

$$k_2 = (q_1 - x_1)/(y_2 - y_1)$$

$$t_1 = \text{Min} \{k_1, k_2\} \quad \text{and} \quad t_2 = \text{Max} \{k_1, k_2\}$$

3.) If $\text{Abs}(m) > 1$ then

$$k_1 = (p_2 - y_1)/(x_2 - x_1)$$

$$k_2 = (q_2 - y_1)/(x_2 - x_1)$$

$$t_1 = \text{Min} \{k_1, k_2\} \quad \text{and} \quad t_2 = \text{Max} \{k_1, k_2\}$$

The last problem that must be solved is the potential for overlap in the erasures on a particular edge. This

problem can be best illustrated with an example. Suppose that edge number two, (v_3, v_4) has projected endpoints $(122, 130)$ and $(200, 200)$ and no portions of this edge are to be erased yet. Refer to table VI.

Table VI.

Edge Table											
Edge No.	Vertex		Number Erasures	Erasure1		Erasure2		Erasure3		Erasure4	
	v_i	v_j		t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
0002	03	04	000000	--	--	--	--	--	--	--	--

Now suppose that a hidden line removal algorithm calls for the portion of the projected edge from $(137.6, 144)$ to $(153.2, 158)$ to be erased. The parameters corresponding to the endpoints of this erasure are stored in table VII.

Table VII.

Edge Table											
Edge No.	Vertex		Number Erasures	Erasure1		Erasure2		Erasure3		Erasure4	
	v_i	v_j		t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
0002	03	04	000001	.20	.40	--	--	--	--	--	--

Next suppose that the portion of the projected edge from $(184.4, 188.3)$ to $(188.3, 189.5)$ needs to be erased. The parameters corresponding to the endpoints of this erasure are stored in table VIII.

Table VIII.

Edge Table											
Edge No.	Vertex		Number Erasures	Erasure1		Erasure2		Erasure3		Erasure4	
	v_i	v_j		t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
0002	03	04	000002	.20	.40	.80	.85	--	--	--	--

Lastly, suppose that the portion of the projected edge from (145.4,151.0) to (161,165) needs to be erased. Finally a problem results, which is that this erasure (.30,.50) overlaps the first erasure already stored in the table. To solve the problem, simply remove the first erasure (.20,.40) from the edge table and substitute (.20,.50) in its place. Refer to table IX.

Table IX.

Edge Table											
Edge No.	Vertex		Number Erasures	Erasure1		Erasure2		Erasure3		Erasure4	
	v_i	v_j		t_1	t_2	t_1	t_2	t_1	t_2	t_1	t_2
0002	03	04	000002	.20	.50	.80	.85	--	--	--	--

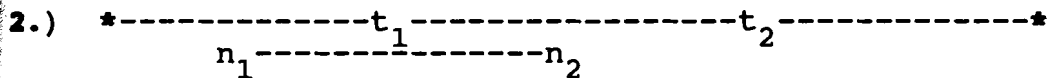
Let t_1 and t_2 be the endpoints of an existing erasure on an edge. Suppose that n_1 and n_2 are the endpoints of a new erasure. There are basically four types of overlap that can occur:

1.) $* \text{-----} t_1 \text{-----} t_2 \text{-----} *$
 $n_1 \text{-----} n_2$

$$t_1 \leq n_1 \leq t_2.$$

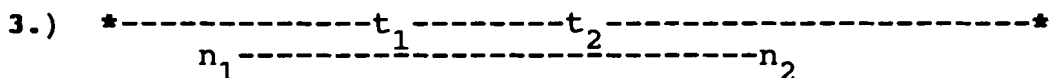
If this type of overlap happens, then remove (t_1, t_2)

from the edge table and add (t_1, n_2) to the table.



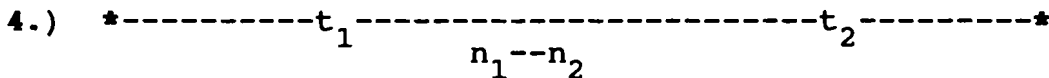
$$n_1 \leq t_1 \leq n_2.$$

If this type of overlap happens, then remove (t_1, t_2) from edge table and add (n_1, t_2) to the table.



$$n_1 \leq t_1 \quad \text{and} \quad t_2 \leq n_2.$$

If this type of overlap happens, then remove (t_1, t_2) from edge table and add (n_1, n_2) to the table.



$$n_1 \geq t_1 \quad \text{and} \quad n_2 \leq t_2.$$

If this type of overlap happens, then ignore (n_1, n_2) and do not update the edge table.

CHAPTER V

CONCLUSION

Summary Of The Thesis

The purpose of this thesis was to explore how three-dimensional objects can be modeled and realistically displayed on a two-dimensional view screen. All man made images are either moving or static. These images are either obtained through construction or by a recording device. For example, photography deals with the recording of static images, while cinematography is concerned with the recording of moving images. However cinematography and photography are not well suited for the construction of three-dimensional images, the reason being that an object must first exist before a picture can be taken. One alternative is the traditional method of drawing and painting to produce images of real world objects. The other more viable alternative is to use a computer for image generation. Hidden line removal is an important aspect in the generation of realistic three-dimensional computer images.

The first chapter was an introduction to the thesis. In it was discussed the idea of hidden line removal and its applications. Also given was an overview of the thesis along with a short account of the history of computer graphics. Chapter two presented the notion of using polyhedra to model three-dimensional objects. Also contained in this chapter was a method for organizing the

edges, vertices, and polygonal surfaces of a polyhedral model. The third chapter dealt with the concept of projecting the edges of the polyhedral model onto the view screen thus producing an image of the object. A major step in this process was the conversion of the coordinates of the object's vertices relative to the standard coordinate system into coordinates relative to the eye coordinate system. Chapter three concluded with a discussion of the viewing parameters which control the size of the object and the direction from which the object will be viewed. Three elementary hidden line removal algorithms were presented in chapter four. The first algorithm was for use with convex polyhedra. The second and third algorithms could be used to remove the hidden lines from both concave and convex polyhedra. Following chapter five are a bibliography and eight appendices A,B,C,D,E,F,G, and H. Appendices A,B, and C contain supplementary information pertaining to the second and third hidden line removal algorithms. Appendix D contains instructions for using the programs located on the program disk. Finally appendices E,F,G, and H contain the source code for the hidden line removal algorithms.

Conclusions

Throughout the course of implementing the hidden line removal algorithms on the computer, several conclusions became evident to the author.

- 1.) Although theoretically an object can be approximated to an arbitrary fine

precision by a plane faced polyhedron, this method of surface modeling is not always practical. For example, a polyhedral approximation of a coffee cup could contain many surfaces and would be difficult to generate and to modify. A simple alteration of any kind would result in having to recalculate many of the coordinate values.

- 2.) To test the hidden line removal algorithms contained in this thesis, several polyhedral models were constructed. A considerable amount of time (much more than was expected) was spent generating the coordinate values and surface orientations for these very simple models.
- 3.) It is very important to understand how the viewing parameters (Θ , Φ , ρ , and D) affect the computer generated images. It took some practice in adjusting the viewing parameters in order to get a particular view of the object.
- 4.) Hidden line removal is very important in the generation of realistic images. Before the hidden line removal algorithms were tested, objects were displayed on

the view screen in their entirety. Almost without exception the computer generated images were very confusing and difficult to interpret. Most of the ambiguities were resolved when the hidden line removal algorithms were applied.

- 5.) Due to the nature of convex polyhedra, the removal of hidden lines is easily accomplished. The surfaces of a convex polyhedron are either completely visible or completely hidden from view. Because of this fact, the first hidden line removal algorithm executes very rapidly as compared to the significantly slower execution speeds of the second and third hidden line removal algorithms designed primarily for concave polyhedra.

Recomendations For Future Study

It is the opinion of the author that a subsequent study involving parallel processing could help increase the characteristically slow execution speeds of hidden line removal algorithms in general. Future studies might also include enhancing an object's realism further through the use of special effects such as shading, transparency, shadows, and textures.

the view screen in their entirety. Almost without exception the computer generated images were very confusing and difficult to interpret. Most of the ambiguities were resolved when the hidden line removal algorithms were applied.

- 5.) Due to the nature of convex polyhedra, the removal of hidden lines is easily accomplished. The surfaces of a convex polyhedron are either completely visible or completely hidden from view. Because of this fact, the first hidden line removal algorithm executes very rapidly as compared to the significantly slower execution speeds of the second and third hidden line removal algorithms designed primarily for concave polyhedra.

Recomendations For Future Study

It is the opinion of the author that a subsequent study involving parallel processing could help increase the characteristically slow execution speeds of hidden line removal algorithms in general. Future studies might also include enhancing an object's realism further through the use of special effects such as shading, transparency, shadows, and textures.

BIBLIOGRAPHY

- [1] Angell, I. O. A Practical Introduction to Computer Graphics, HALSTEAD PRESS, New York, NY, 1982 NY, 1982.
- [2] Brumfiel, C. F. Eicholz, R. E., and Shanks, M. E. Geometry, Addison-Wesley Publishing Co., Inc., Reading, MA, 1962.
- [3] Demel, J. T. and Miller M. J. Introduction to Computer Graphics, Brooks/Cole Engineering Division, Monterey, CA, 1984.
- [4] Hearn, D. and Baker, M.P. Computer Graphics, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
- [5] Hearn, D. and Baker, M.P. Computer Graphics for the IBM Personal Computer, Prentice-Hall, Inc. Englewood Cliffs, NJ, 1983.
- [6] Lewell, J. Computer Graphics, Van Nostrand Reinhold Co., New York, 1985.
- [7] McGreagor, J. and Watt, A. The Art of Graphics for the IBM PC, Addison-Wesley Publishing Co., Reading, MA, 1986.
- [8] Mufti, A. A. Elementary Computer Graphics, Reston Publishing Co., Inc., Reston, VA, 1983.
- [9] Myers, R.E. Microcomputer Graphics, Addison-Wesley Publishing Co., Reading, MA, 1982.
- [10] Newman, W. M. and Sproull, R. F. Principals of Interactive Computer Graphics, McGraw-Hill Book Company, New York, NY, 1979.

APPENDIX A

Obviously in order to calculate the intersection point of two lines, the equations of both lines must be known. The standard form for the equation of a straight line in R^2 is $A x + B y + C = 0$ where $A, B,$ and C are constants. Given two distinct points (x_1, y_1) and (x_2, y_2) the constants A, B and C can be calculated by the following [9, pg. 45].

$$A = y_2 - y_1$$

$$B = x_1 - x_2$$

$$C = x_2 y_1 - x_1 y_2$$

One way to find the intersection point of two lines is to solve the following system:

$$A_1 x + B_1 y = -C_1 \quad (\text{line one}).$$

$$A_2 x + B_2 y = -C_2 \quad (\text{line two}).$$

By Cramer's rule:

$$x = \frac{\text{Det} \begin{bmatrix} -C_1 & B_1 \\ -C_2 & B_2 \end{bmatrix}}{\text{Det} \begin{bmatrix} A_1 & B_1 \\ A_2 & B_2 \end{bmatrix}} \quad y = \frac{\text{Det} \begin{bmatrix} A_1 & -C_1 \\ A_2 & -C_2 \end{bmatrix}}{\text{Det} \begin{bmatrix} A_1 & B_1 \\ A_2 & B_2 \end{bmatrix}}$$

Remember that the intersection point of two finite line segments is desired and not the intersection point of the infinite lines defined by the two line segments. Assuming that the two line segments are not dependent or

inconsistent, the following three possibilities exist [9, pg. 48].

- 1.) The intersection point lies between the endpoints of both line segments. In this case the two line segments actually intersect.
- 2.) The intersection point lies between the endpoints of one line segment and not the other. In this case the two line segments do not actually intersect.
- 3.) The intersection point does not lie between the endpoints of either line segment. Also in this case the two line segments do not actually intersect.

APPENDIX B

The following algorithm can be used to determine if a given point P is located interior or exterior to of a plane polygon [8, pg. 143]. The algorithm is best explained by using figures 19 and 20. Consider the following angles where the v_i 's represent the vertices of the polygon.

$$A_1 = \sphericalangle v_1 P v_2, A_2 = \sphericalangle v_2 P v_3, \dots, A_n = \sphericalangle v_n P v_1$$

If the rotation about the point P from v_i to v_{i+1} is clockwise then A_i is given a negative value; otherwise A_i is given a positive value.

- 1.) If P is interior to the polygon, then the sum of the A_i 's will equal plus or minus 360° . Referring to figure 19, since P is inside the polygon the angles $A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7$ add up to a sum of -360° .
- 2.) If P is exterior to the polygon, then the sum of the A_i 's will equal 0° . Referring to figure 20, since P is outside the polygon the angles $A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7$ add up to a sum of 0° .

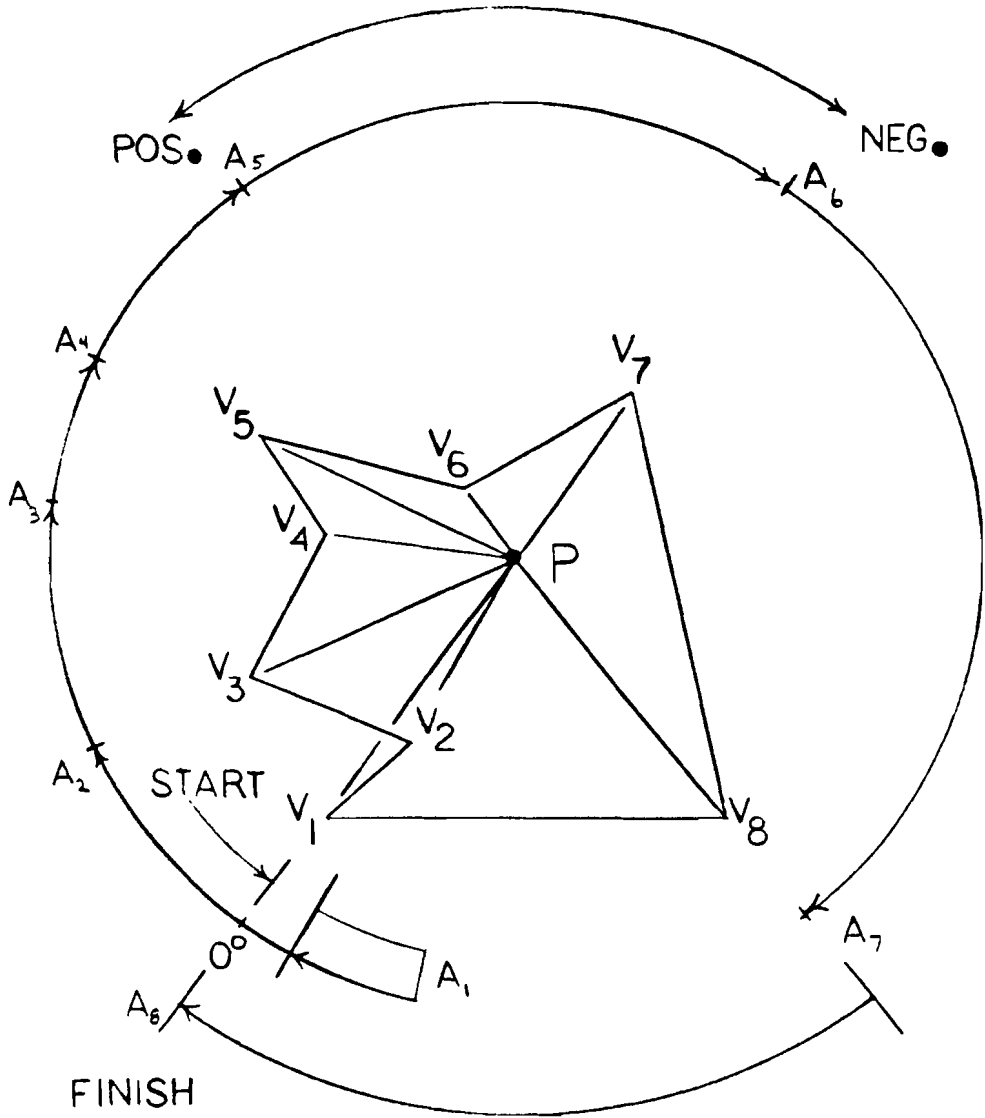


Figure 19.

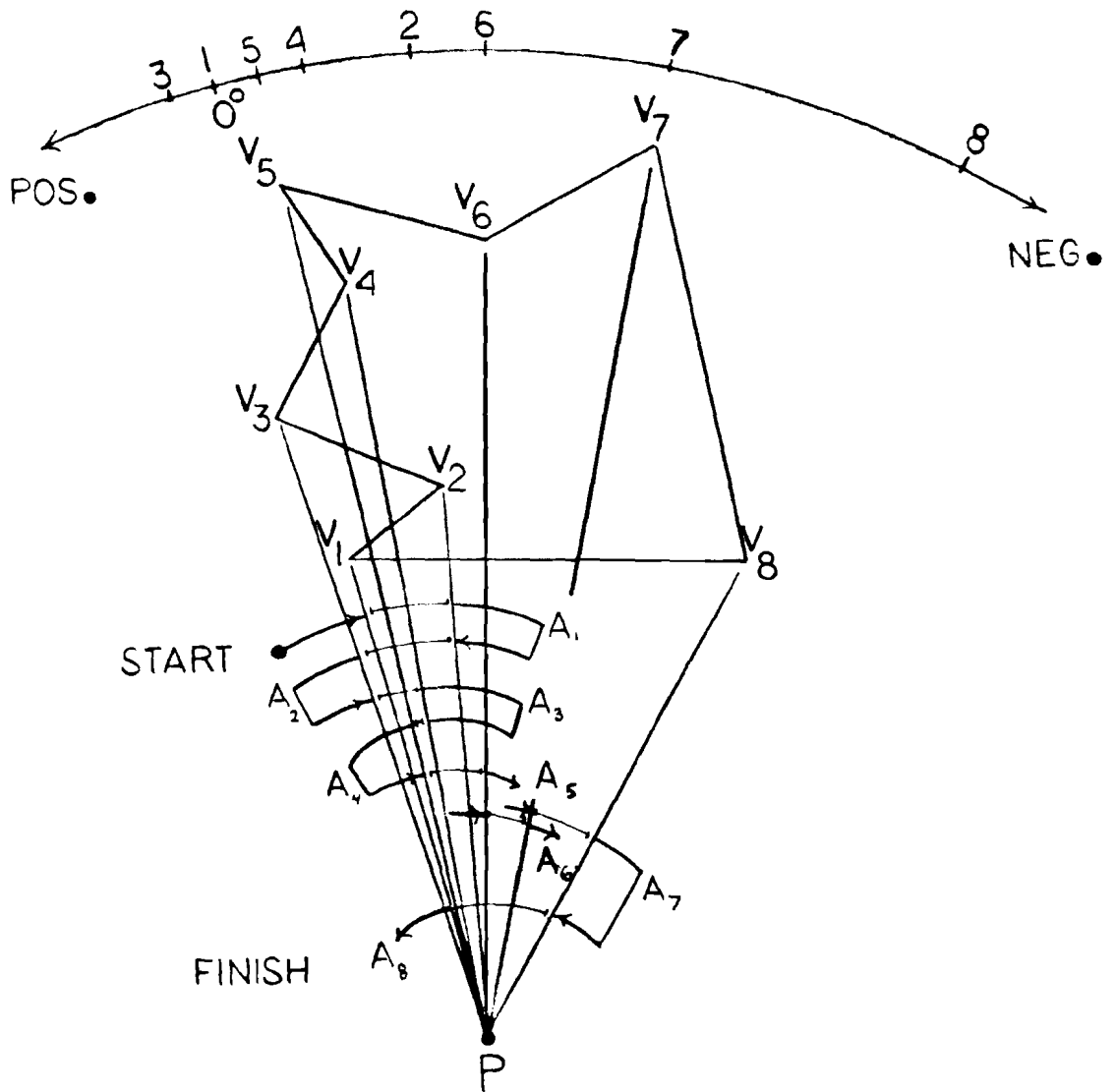


Figure 20.

APPENDIX C

Considering the viewpoint to be the origin, a point P is defined to be in front of a plane if and only if the point P is on the same side of the plane as the origin or is contained within the plane. In any other case P is defined to be behind the plane.

The standard form for the equation of a plane is $A x + B y + C z + D = 0$ where $A, B, C,$ and D are constants. Given three non collinear points $(x_1, y_1, z_1), (x_2, y_2, z_2),$ and (x_3, y_3, z_3) the constants $A, B, C,$ and D can be determined by the following [6, pg. 195].

$$A = Y_1(z_2 - z_3) + Y_2(z_3 - z_1) + Y_3(z_1 - z_2)$$

$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$

$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$

$$D = -x_1(y_2 z_3 - Y_3 z_2) - x_2(y_3 z_1 - Y_1 z_3) - x_3(y_1 z_2 - Y_2 z_1)$$

Consider a plane that does not contain the origin and having the equation: $A x + B y + C z + D = 0$ where $D > 0$. This plane partitions R^3 into three disjoint sets of points [1, pg. 54].

- 1.) The set of points lying on the same side of the plane as the origin. Any point (x, y, z) in this set will satisfy:

$$A x + B y + C z + D > 0.$$

- 2.) The set of points contained within the plane itself. Obviously any point (x, y, z)

in this set will satisfy:

$$A x + B y + C z + D = 0.$$

3.) The set of points lying on the side opposite the side of the plane containing the origin. Any point (x,y,z) in this set will satisfy:

$$A x + B y + C z + D < 0.$$

Therefore the point P having coordinates (i,j,k) is defined to be behind the plane having equation: $A x + B y + C z + D$ where $D > 0$ if and only if $A i + B j + C k + D < 0$. On the other hand, point P is defined to be in front of the plane if and only if $A i + B j + C k + D \geq 0$.

APPENDIX D

What follows are some guidelines and helpful information on how to use the programs on the program disk.

Vertex And Surface Definition Files

In order to display the image of any polyhedral model two files must exist. The first file is the **vertex definition** file. The vertex definition file contains a listing of the standard three-dimensional coordinates for each vertex. This sequential file consists of an ordered sequence of records, one record per line, and each record having three fields. The three fields contain the X,Y, and Z standard coordinates respectively for each consecutive vertex. Examples of vertex definition files are **HOUSE.VER**, **DODEC.VER**, **SIXTY.VER**, **SPHERE.VER**, and **TETRA.VER** which are all located on the program disk. To obtain a listing of the contents of any vertex definition file simply type the following at the DOS prompt:

```
A> copy filename.ext con
```

For example, if the user wanted to look at the contents of the vertex definition file **TETRA.VER** he would simply type

```
A> copy TETRA.VER con
```

and the following output would be produced:

```
5.77350269 10.00000000 0.000000000
5.77350269 -10.00000000 0.000000000
-11.54700538 0.00000000 0.000000000
0.00000000 0.00000000 16.329931620
```

The above file **TETRA.VER** is a discription of the vertices of a regular tetrahedron. In the above file, records 1,2,3

and 4 correspond to the three-dimensional coordinates of v_1, v_2, v_3 , and v_4 (vertices 1-4) respectively.

The second file, the **surface definition** file contains a listing of the edges that form each surface. This sequential file consists of an ordered sequence of records, one record per line, and each record not necessarily having the same number of fields. The first field always lists the number of edges that form each surface, and the remaining fields list the vertices defining each of the edges that make up that particular surface. Examples of surface definition files are **HOUSE.SUR**, **DODEC.SUR**, **SPHERE.SUR**, **SIXTY.SUR** and **TETRA.SUR** which are all located on the program disk. For example, if the user wanted to view the contents of the surface definition file **TETRA.SUR** he would simply type

```
A> copy TETRA.SUR con
```

and the following output would be produced:

```
03 01 04 02 01
03 01 03 04 01
03 03 02 04 03
03 01 02 03 01
```

The above file **TERTTA.SUR** is a discription of the surfaces of a regular tetrahedron. Records (1-4) correspond to surfaces (1-4) of the object respectively. The first field indicates the number of edges that form the surface. The remaining fields list the vertices that define each of the edges that make up the surface. For example, surface number one is formed by three edges. The first, second, and third edges are v_1v_4, v_4v_2 , and v_2v_1

respectively. Note that when using hidden line removal algorithms number one and three, the vertices of each surface must be ordered in such a way that the normal vector associated with each surface points in an outward direction away from the center of the object. Refer to chapter four for more information on the proper ordering of the vertices of a particular surface.

The names of the vertex and surface definition files corresponding to a particular object must be the same (preferably the name of the object that they describe) and the extensions must be **.VER** and **.SUR** respectively.

Function Keys

The three programs **HLR01.PAS**, **HLR02.PAS**, and **HLR03.PAS** on the program disk are controlled exclusively by the function keys (**F1 - F10**). The following is an explanation of what each function key does.

Function Key	Operation
F1	Decrease Theta .
Shift F1	Decrease Theta rapidly.
Ctrl F1	Turns status information on or off.
F2	Increase Theta .
Shift F2	Increase Theta rapidly.
Ctrl F2	Turns coordinate axes on or off.
F3	Decrease Phi .

Function Key	Operation
Shift F3	Decrease Phi rapidly.
Ctrl F3	Turns the hidden line removal algorithm on or off.
F4	Increase Phi.
Shift F4	Increase Phi rapidly.
F5	Decrease Rho.
Shift F5	Decrease Rho rapidly.
F6	Increase Rho.
Shift F6	Increase Rho rapidly.
F7	Decrease D.
Shift F7	Decrease D rapidly.
F8	Increase D.
Shift F8	Increase D rapidly.
F9	Draws object. *
F10	Exits the program and returns control to the Turbo or DOS command prompt.
<p>* The object will be drawn according to whether or not hidden line removal is turned on or off. It may take several minutes for an image of the object to appear on view screen.</p>	

Executing The Programs

To run the programs **HLR01.PAS**, **HLR02.PAS**, or **HLR03.PAS** do the following:

1. Insert the program disk into drive A.
2. Turn the computer on. If the computer is

already on then restart the computer by pressing the three (3) keys **Ctrl,Alt,** and **Del** simultaneously.

3. Depending on which program the user wants to run, (**HLR01,HLR02,** or **HLR03**), type one of the following at the **DOS** prompt:

A> **HLR01** objectname

A> **HLR02** objectname

A> **HLR03** objectname

For example, if the user wanted to remove the hidden lines from a regular dodecahedron by using the first hidden line removal algorithm, he would simply type the following:

A> **HLR01 DODEC**

and the program **HLR01** would begin executing and would load vertex and surface definition files **DODEC.VER** and **DODEC.SUR** respectively.

APPENDIX E

The following is the source code written in Turbo Pascal for algorithm number one. Refer to appendix D for instructions on how to run the program.

Program Hidden_Line_Removal_01 (Input,Output) ;

Language/Compiler : TURBO Pascal 3.01A
Computer : IBM PC or compatibles

Copyright(C) : Permission is granted by the author to use, reproduce, modify all or part of the computer programs contained herein for the readers own personal use. This permission is not to be construed as a license to distribute or sell all or any part of the programs to others in any shape or form.

Disclaimer : The author has spent a great deal of time and effort in preparing this thesis and the programs contained herein. This includes many hours of research, development and testing of the programs to determine their effectiveness. The author shall not be liable in any event for incidental or consequential damages in connection with or arising out of the furnishing, performance or use of all or any portion of these programs.

)

Const

{ Palette 0 color constants. }

Black = 0 ;
Green = 1 ;
Red = 2 ;
Brown = 3 ;

{ Display window boundary constants. }

Xmin = 1 ;
Xmax = 310 ;
Ymin = 20 ;
Ymax = 199 ;

{ Roughly the center of the display window. }

CenterX = 155 ;


```
CenterY      =      89 ;
AspectRatio  =      0.93 ; { the approximate aspect ratio for the IBM PC
                             graphics screen. }
```

{ These constants determine how much the viewing parameters Theta, Phi, Rho, and D will increase or decrease when the respective function keys are pressed that control each of these parameters. }

```
DelTheta    = 0.0175 ; { roughly one degree. }
DelPhi      = 0.0175 ; { roughly one degree. }
DelRho      =      5 ;
DelD        =      5 ;

DMin        =      5 ; { The minimum value for D. }
RhoMin      =      5 ; { The minimum value for Rho. }
VertexMax   =     300 ; { The maximum number of vertices that an
                          object can have. }
SurfaceMax  =     300 ; { The maximum number of surfaces an object
                          can have. }
MaxSurfVertices =     20 ; { The maximum number of vertices that can
                              form a surface. }
BorderColor =      Red ; { The color of the display window border. }
ObjectColor =     Green ; { The edges that form the object will be
                              displayed in this color. }
```

Type

```
R3Vector     = Record
                x,y,z : Real ;
                End ;

R2Vector     = Record
                x,y : Real ;
                End ;

SurfaceRecord = Record
                Vertices   : Array[1..MaxSurfVertices] Of Integer ;
                NumVertices : Integer ;
                End ;

R3VertexType = Array[1..VertexMax] Of R3Vector ;
NormalType   = Array[1..SurfaceMax] Of R3Vector ;
SurfaceType  = Array[1..SurfaceMax] Of SurfaceRecord ;
R2VertexType = Array[1..VertexMax] Of R2Vector ;

FileNameType = String[8] ;

StatsRecord  = Record
                StatsOn,Axis,Hidden : Boolean ;
                Th,Ph,Rh,Dist : Real ;
                ObjName : FileNameType ;
                End ;
```

```

R3_SINRD_VERTEX          : R3VertexType ;
{ R3_SINRD_VERTEX[i] contains the standard three-dimensional coordinates for
  the i-th vertex. }

R2Vertex                 : R2VertexType ;
{ R2Vertex[i] contains the two-dimensional screen coordinates for the i-th
  vertex. }

Surface                  : SurfaceType ;
{ Surface[i].NumVertices contains the number of vertices that form the i-th
  surface.

  Surface[i].Vertices[j] contains the j-th vertex in the formulation of
  the i-th surface. }

Normal                   : NormalType ;
{ Normal[i] contains the normal vector associated with the i-th surface. }

VertexCount              : Integer ;
{ The actual number of vertices that form the object. }

SurfaceCount             : Integer ;
{ The actual number of surfaces that form the object. }

FunctKey                 : Integer ;
{ Contains a number associated with the function key that was pressed. }

Theta,Phi,Rho,D         : Real ;
{ The viewing parameters. }

A,B                      : Char ;
{ Holds the sequence of characters generated when a function key is
  pressed. }

Stats                    : StatsRecord ;
{ Contains the status information which is displayed on the first two lines
  of the display screen. }

ObjectName               : FileNameType ;
{ Contains the name of the object being displayed. }

```

```
($I GRAPH1.TPU )
```

```
Procedure CalcNormals(R3_SINRD_VERTEX:R3VertexType ; Surface:SurfaceType ;  
                          SurfaceCount:Integer ; Var Normal:NormalType) ;
```

```
Var a,b : R3Vector ; n : Integer ;
```

```
Begin
```

```
For n := 1 To SurfaceCount Do
```

```
  Begin
```

```
    R3Diff(R3_SINRD_VERTEX[Surface[n].Vertices[2]],  
          R3_SINRD_VERTEX[Surface[n].Vertices[1]],a) ;
```

```
    R3Diff(R3_SINRD_VERTEX[Surface[n].Vertices[3]],  
          R3_SINRD_VERTEX[Surface[n].Vertices[1]],b) ;
```

```
    Normal[n].x := a.y * b.z - b.y * a.z ;
```

```
    Normal[n].y := b.x * a.z - a.x * b.z ;
```

```
    Normal[n].z := a.x * b.y - b.x * a.y ;
```

```
  End ;
```

```
End ;
```

```
Procedure CalcR2Vertex(R3_VERTEX:R3VertexType ; VertexCount:Integer ;  
                          Var R2Vertex:R2VertexType ; Theta,Phi,Rho,D:Real) ;
```

```
Var k : Integer ; R3Eye : R3Vector ;
```

```
Begin
```

```
For k := 1 To VertexCount Do
```

```
  Begin
```

```
    EyeXYZ(Theta,Phi,Rho,D,R3_SINRD_VERTEX[k],R3Eye) ;
```

```
    ScreenXY(Theta,Phi,Rho,D,R3Eye,R2Vertex[k]) ;
```

```
  End ;
```

```
End ;
```

```
Procedure DrawSurface(Surface:SurfaceRecord ; R2Vertex:R2VertexType ;  
                          Color:Integer) ;
```

```
Var k,vn1,vn2 : Integer ;
```

```
Begin
```

```

For k := 1 to Surface.NumVertices Do
Begin
  vn1 := Surface.Vertices[k] ;
  vn2 := Surface.Vertices[k+1] ;

  Clip(R2Vertex[vn1].x,R2Vertex[vn1].y,R2Vertex[vn2].x,R2Vertex[vn2].y,
    Color,Xmin,Xmax,Ymin,Ymax) ;
End ;
End ;

Procedure DrawSurfaces(Surface:SurfaceType ; SurfaceCount,Color:Integer ;
  Normal:NormalType ; R3_SINRD_VERTEX:R3VertexType ;
  R2Vertex:R2VertexType ; Theta,Phi,Rho,D : Real ;
  Stats:StatsRecord) ;

Var ViewVector : R3Vector ; n : Integer ;

Begin
  For n := 1 To SurfaceCount Do
  Begin
    CalcViewVector(Surface[n],R3_SINRD_VERTEX,Theta,Phi,Rho,ViewVector) ;

    If Stats.Hidden Then
    Begin
      If R3DotProduct(ViewVector,Normal[n]) > 0 Then
        DrawSurface(Surface[n],R2Vertex,Color) ;
      End
    Else
      DrawSurface(Surface[n],R2Vertex,Color) ;
    End ;
  End ;
End ;

Procedure DrawObject(R3_SINRD_VERTEX:R3VertexType ; VertexCount:Integer ;
  Surface:SurfaceType ; SurfaceCount:Integer ;
  Normal:NormalType ; Color:Integer ;
  Theta,Phi,Rho,D:Real ; Stats:StatsRecord) ;

Begin
  CalcR2Vertex(R3_SINRD_VERTEX,VertexCount,R2Vertex,Theta,Phi,Rho,D) ;

  Write(Chr(7)) ;
  ZapScreen ;
  DrawSurfaces(Surface,SurfaceCount,Color,Normal,R3_SINRD_VERTEX,
    R2Vertex,Theta,Phi,Rho,D,Stats) ;

  If Stats.Axis Then DrawAxes(Theta,Phi,Rho,D) ;

```

```

Procedure Initialize(Var R3_STINRD_VERTEX:R3VertexType ;
                    Var Surface:SurfaceType ;
                    Var Normal:NormalType ;
                    ObjectName:FileNameType ;
                    Var Theta,Phi,Rho,D:Real ;
                    Var VertexCount,SurfaceCount:Integer ;
                    Var Stats:StatsRecord) ;

```

```

  Var Infile : Text ;

```

```

Begin
  Assign(Infile,ObjectName + '.Ver') ;
  Reset(Infile) ;

  LoadVertex(Infile,R3_STINRD_VERTEX,VertexCount) ;
  Close(Infile) ;

  Assign(Infile,ObjectName + '.Sur') ;
  Reset(Infile) ;

  LoadSurface(Infile,Surface,SurfaceCount) ;
  Close(Infile) ;

  CalcNormals(R3_STINRD_VERTEX,Surface,SurfaceCount,Normal) ;

  Theta := 0 ;
  Phi := Pi/2 ;
  Rho := 425 ;
  D := 1549 ;

```

```

With Stats Do

```

```

  Begin
    StatsOn := True ;
    ObjName := ObjectName ;
    Axis := False ;
    Hidden := True ;
    Th := Theta ;
    Ph := Phi ;
    Rh := Rho ;
    Dist := D ;
  End ;

```

```

  ZapScreen ;

```

```

End ;

```

```

Begin

```

```

  ClrScr ;

```

```
TextMode(C80) ;
```

```
If ParamCount = 0 Then
```

```
Begin
```

```
Write('Object : ' ) ;
```

```
ReadLn(ObjectName) ;
```

```
End
```

```
Else
```

```
ObjectName := ParamStr(1) ;
```

```
Initialize(R3_SINRD_VERTEX, Surface, Normal, ObjectName, Theta, Phi, Rho, D,
VertexCount, SurfaceCount, Stats) ;
```

```
Repeat
```

```
Stats.Th := Theta ; Stats.Rh := Rho ;
```

```
Stats.Ph := Phi ; Stats.Dist := D ;
```

```
If Stats.StatsOn Then ShowStats(Stats) Else EraseStats(Stats) ;
```

```
GetKeySequence(A,B) ;
```

```
If Ord(A) = 27 Then
```

```
FuncKey := DecodeFunctionKey(B) ;
```

```
If Ord(A) = 27 Then
```

```
Begin
```

```
Case FuncKey Of
```

```
01 : WrapAroundDec(Theta, DelTheta, 0, 2*Pi) ;
```

```
02 : WrapAroundInc(Theta, DelTheta, 0, 2*Pi) ;
```

```
03 : WrapAroundDec(Phi, DelPhi, 0, Pi) ;
```

```
04 : WrapAroundInc(Phi, DelPhi, 0, Pi) ;
```

```
05 : Dec(Rho, DelRho) ;
```

```
06 : Inc(Rho, DelRho) ;
```

```
07 : Dec(D, DelD) ;
```

```
08 : Inc(D, DelD) ;
```

```
09 : DrawObject(R3_SINRD_VERTEX, VertexCount, Surface, SurfaceCount,
Normal, ObjectColor, Theta, Phi, Rho, D, Stats) ;
```

```
11 : WrapAroundDec(Theta, DelTheta*10, 0, 2*Pi) ;
```

```
12 : WrapAroundInc(Theta, DelTheta*10, 0, 2*Pi) ;
```

```
13 : WrapAroundDec(Phi, DelPhi*10, 0, Pi) ;
```

```
14 : WrapAroundInc(Phi, DelPhi*10, 0, Pi) ;
```

```
15 : Dec(Rho, DelRho*10) ;
```

```
16 : Inc(Rho, DelRho*10) ;
```

```
17 : Dec(D, DelD*10) ;
```

```
18 : Inc(D, DelD*10) ;
```

```
21 : Stats.StatsOn := Not Stats.StatsOn ;
```

```
22 : Stats.Axis := Not Stats.Axis ;
```

```
23 : Stats.Hidden := Not Stats.Hidden ;
```

End ;

End ;

Until (FunctKey = 10) And (Ord(A) = 27) ;

ExitProgram ;

end.

APPENDIX F

The following is the source code written in Turbo Pascal for algorithm number two. Refer to appendix D for instructions on how to run the program.

```
Program Hidden_Line_Removal_02 (Input,Output) ;
```

```
{ Language/Compiler   : TURBO Pascal 3.01A  
  Computer           : IBM PC or compatibles
```

```
Copyright(c)       : Permission is granted by the author to use, reproduce,  
                    : modify all or part of the computer programs contained  
                    : herein for the readers own personal use. This  
                    : permission is not to be construed as a license to  
                    : distribute or sell all or any part of the programs to  
                    : others in any shape or form.
```

```
Disclaimer         : The author has spent a great deal of time and effort  
                    : in preparing this thesis and the programs contained  
                    : herein. This includes many hours of research,  
                    : development and testing of the programs to determine  
                    : their effectiveness. The author shall not be liable  
                    : in any event for incidental or consequential damages  
                    : in connection with or arising out of the furnishing,  
                    : performance or use of all or any portion of these  
                    : programs.
```

```
}
```

Const

```
{ Palette 0 color constants. }
```

```
Black      = 0 ;  
Green      = 1 ;  
Red        = 2 ;  
Brown     = 3 ;
```

```
{ Display window boundary constants. }
```

```
Xmin       = 1 ;  
Xmax       = 310 ;  
Ymin       = 20 ;  
Ymax       = 199 ;
```

```
{ Roughly the center of the display window. }
```

```
CenterX    = 155 ;  
CenterY    = 89 ;
```


AspectRatio = 0.93 ; { the approximate aspect ratio for the IBM PC graphics screen. }

{ These constants determine how much the viewing parameters Theta, Phi, Rho, and D will increase or decrease when the respective function keys are pressed that control each of these parameters. }

DelTheta = 0.0175 ;
 DelPhi = 0.0175 ;
 DelRho = 5 ;
 DelD = 5 ;
 DMin = 5 ; { The minimum value for D. }
 RhoMin = 5 ; { The minimum value for Rho. }
 VertexMax = 100 ; { The maximum number of vertices that an object can have. }
 SurfaceMax = 100 ; { The maximum number of surfaces an object can have. }
 MaxSurfVertices = 8 ; { The maximum number of vertices that can form a surface. }
 BorderColor = Red ; { The color of the display window border. }
 ObjectColor = Green ; { The edges that form the object will be displayed in this color. }
 ErasureMax = 10 ; { The maximum number of disjoint erasures that an edge can have. }
 EdgesMax = 100 ; { The maximum number of edges that an object can have. }

Type

ErasureRecord = Record
 t1,t2 : Real ;
 End ;

ErasureType = Array[1..ErasureMax] Of ErasureRecord ;

EdgesRecord = Record
 nv1,nv2 : Integer ;
 Erasure : ErasureType ;
 NumErasures : Integer ;
 Flag : Boolean ;
 End ;

EdgesType = Array[1..EdgesMax] Of EdgesRecord ;

R3Vector = Record
 x,y,z : Real ;
 End ;

R2Vector = Record
 x,y : Real ;
 End ;

LineType = Record
 end1,end2 : R2Vector ;
 End ;

```

VerticesType      = Array[1..MaxSurfVertices] Of R2Vector ;
SurfaceRecord     = Record
                   Vertices      : Array[1..MaxSurfVertices] of Integer ;
                   NumVertices   : Integer ;
                   End ;
R3VertexType      = Array[1..VertexMax] Of R3Vector ;
R2VertexType      = Array[1..VertexMax] Of R2Vector ;
SurfaceType       = Array[1..SurfaceMax] Of SurfaceRecord ;
FileNameType      = String[8] ;
StatsRecord       = Record
                   StatsOn,Axis,Hidden : Boolean ;
                   Th,Ph,Rh,Dist : Real ;
                   ObjName : FileNameType ;
                   End ;

```

Var

```

R3_STNRD_VERTEX           : R3VertexType ;
{ R3_STNRD_VERTEX[i] contains the standard three-dimensional coordinates for
  the i-th vertex. }

R3_EYE_VERTEX            : R3VertexType ;
{ R3_EYE_VERTEX[i] contains the eye three-dimensional coordinates for the
  i-th vertex. }

R2Vertex                 : R2VertexType ;
{ R2Vertex[i] contains the two-dimensional screen coordinates for the i-th
  vertex. }

Surface                   : SurfaceType ;
{ Surface[i].NumVertices contains the j-th vertex in the formulation of the
  i-th surface. }

Edges                     : EdgesType ;
{ Edges[i].nv1 contains the number of the first vertex defining the
  i-th edge.

  Edges[i].nv2 contains the number of the second vertex defining the
  i-th edge.

  Edges[i].Erasure[j].t1 contains the the first endpoint defining the j-th
  erasure of the i-th edge.

```

Edges[i].Erasure[j].t2 contains the second endpoint defining the j-th erasure of the i-th edge.

Edges[i].NumErasures contains the number of erasures currently on the i-th edge.

Edges[i].Flag is a process flag. If Edges[i].Flag = TRUE then the i-th edge will not be processed (ie. tested for visibility with respect to the surfaces of the object).)

VertexCount : Integer ;

{ The actual number of vertices that form the object. }

SurfaceCount : Integer ;

{ The actual number of surfaces that form the object. }

EdgesCount : Integer ;

{ The actual number of edges that form the surface. }

Theta,Phi,Rho,D : Real ;

{ The viewing parameters. }

FunctKey : Integer ;

{ Contains the number associated with the function key that was pressed. }

A,B : Char ;

{ Hold the sequence of characters generated when a function key is pressed. }

Stats : StatsRecord ;

{ Contains the status information which is displayed on the first two lines of the display screen. }

ObjectName : FileNameType ;

{ Contains the name of the object being displayed. }

{ \$I GRAPH1.TPU }

{ \$I GRAPH2.TPU }

Procedure DrawEdge(Edge:EdgesRecord ; R2Vertex:R2VertexType ; Color:Integer) ;

Var x1,y1,x2,y2,q1,q2 : Real ; k : Integer ;

Begin

x1 := R2Vertex[Edge.nv1].x ; y1 := R2Vertex[Edge.nv1].y ;

```
x2 := R2Vertex[Edge.nv2].x ; y2 := R2Vertex[Edge.nv2].y ;
```

```
If Edge.NumErasures = 0 Then
```

```
  Clip(x1,y1,x2,y2,Color,Xmin,Xmax,Ymin,Ymax)
```

```
Else
```

```
Begin
```

```
  SortErasure(Edge.Erasure,Edge.NumErasures) ;
```

```
  q1 := Edge.Erasure[1].t1 ;
```

```
  Clip(x1,y1,x1+q1*(x2-x1),y1+q1*(y2-y1),Color,Xmin,Xmax,Ymin,Ymax) ;
```

```
  k := 1 ;
```

```
  While k <= Edge.NumErasures - 1 Do
```

```
    Begin
```

```
      q1 := Edge.Erasure[k].t2 ;
```

```
      q2 := Edge.Erasure[k].t1 ;
```

```
      Clip(x1+q1*(x2-x1),y1+q1*(y2-y1),x1+q2*(x2-x1),y1+q2*(y2-y1),Color,  
          Xmin,Xmax,Ymin,Ymax) ;
```

```
      k := k+1 ;
```

```
    End ;
```

```
  q1 := Edge.Erasure[Edge.NumErasures].t2 ;
```

```
  Clip(x1+q1*(x2-x1),y1+q1*(y2-y1),x2,y2,Color,Xmin,Xmax,Ymin,Ymax) ;
```

```
End ;
```

```
End ;
```

```
Procedure AddEdge(Var Edges:EdgesType ; Var EdgesCount:Integer ;  
                  nv1,nv2,NumErasures:Integer ; Flag:Boolean ) ;
```

```
Begin
```

```
If EdgesCount < EdgesMax Then
```

```
Begin
```

```
  EdgesCount := EdgesCount + 1 ;
```

```
  If nv1 > nv2 Then SwitchIntegers(nv1,nv2) ;
```

```
  Edges[EdgesCount].nv1      := nv1 ;
```

```
  Edges[EdgesCount].nv2      := nv2 ;
```

```
  Edges[EdgesCount].Flag      := Flag ;
```

```
  Edges[EdgesCount].NumErasures := NumErasures ;
```

```
End ;
```

```
End ;
```

```
Procedure BuildEdgeTable(Var Edges:EdgesType ; Var EdgesCount:Integer ;  
                          Surface:SurfaceType ; SurfaceCount:Integer) ;
```

```
Var j,k,nv1,nv2,EdgeNumber : Integer ; Found : Boolean ;
```

```
Begin
```

```
EdgesCount := 0 ;
```

```
For j := 1 To SurfaceCount Do
```

```
Begin
```

```
For k := 1 To Surface[j].NumVertices Do
```

```
Begin
```

```
nv1 := Surface[j].Vertices[k] ;
```

```
nv2 := Surface[j].Vertices[k+1] ;
```

```
FindEdgeNumber(Edges,EdgesCount,nv1,nv2,EdgeNumber,Found) ;
```

```
If Not Found Then AddEdge(Edges,EdgesCount,nv1,nv2,0,False) ;
```

```
End ;
```

```
End ;
```

```
End ;
```

```
Procedure DrawObject(Edges:EdgesType ; EdgesCount:Integer ;  
R2Vertex:R2VertexType) ;
```

```
Var k : Integer ;
```

```
Begin
```

```
For k := 1 To EdgesCount Do DrawEdge(Edges[k],R2Vertex,Green) ;
```

```
End ;
```

```
Procedure RemoveHiddenLines(R3_STNRD_VERTEX,R3_EYE_VERTEX:R3VertexType ;  
R2Vertex:R2VertexType ; Surface:SurfaceType ;  
Var Edges:EdgesType ; EdgesCount,VertexCount,  
SurfaceCount : Integer ; Theta,Phi,Rho,D:Real) ;
```

```
Var Surf,K,NumInt,ntv1,ntv2,NumSurfVertices,EdgeNumber : Integer ;
```

```
e,f,g,h,minpolyx,maxpolyx,minpolyy,maxpolyy : Real ;
```

```
TestEdge,PolyEdge : LineType ;
```

```
p1,p2 : R2Vector ;
```

```
PolyPoints : VerticesType ;
```

```
h1,h2,h3 : R3Vector ;
```

```
Dependent,Found,Behind1,Behind2,Insidel,Inside2 : Boolean ;
```

```
Begin
```

```
For Surf := 1 To SurfaceCount Do
```

```
Begin
```

```
NumSurfVertices := Surface[Surf].NumVertices ;
```

```
For K := 1 To NumSurfVertices+1 Do
```

```
PolyPoints[K] := R2Vertex[Surface[Surf].Vertices[K]] ;
```

```

MarkSurfaceEdges(Edges,EdgesCount,Surface[Surf]) ;

RectangularBoundry(NumSurfVertices,PolyPoints,minpolyx,maxpolyx,
                    minpolyy,maxpolyy) ;

h1 := R3_EYE_VERTEX[Surface[Surf].Vertices[1]] ;
h2 := R3_EYE_VERTEX[Surface[Surf].Vertices[2]] ;
h3 := R3_EYE_VERTEX[Surface[Surf].Vertices[3]] ;

CalcR3Plane(h1,h2,h3,e,f,g,h) ;

For EdgeNumber := 1 To EdgesCount Do
Begin

    ntv1 := Edges[EdgeNumber].nv1 ;
    ntv2 := Edges[EdgeNumber].nv2 ;

    If Edges[EdgeNumber].Flag Then
    Nothing
    Else
    Begin
        TestEdge.end1 := R2Vertex[ntv1] ;
        TestEdge.end2 := R2Vertex[ntv2] ;

        Behind1 := BehindPlane(R3_EYE_VERTEX[ntv1],e,f,g,h) ;
        Behind2 := BehindPlane(R3_EYE_VERTEX[ntv2],e,f,g,h) ;

        If ((Not Behind1) And (Not Behind2)) Or
            (OutsideBox(TestEdge,minpolyx,maxpolyx,minpolyy,maxpolyy)) Then
        Nothing
        Else
        Begin
            FindIntersectionPoints(TestEdge,NumSurfVertices,PolyPoints,
                                   p1,p2,NumInt,Dependent) ;

            If Not Dependent Then
            Begin
                Insidel := InsidePoly(NumSurfVertices,PolyPoints,TestEdge.end1) ;
                Inside2 := InsidePoly(NumSurfVertices,PolyPoints,TestEdge.end2) ;

                ProcessTestEdge(TestEdge,EdgeNumber,
                                R3_EYE_VERTEX[ntv1],R3_EYE_VERTEX[ntv2],
                                R2Vertex,p1,p2,NumInt,NumSurfVertices,
                                PolyPoints,e,f,g,h,Theta,Phi,Rho,D,Edges,
                                Behind1,Behind2,Insidel,Inside2) ;

                End ;
            End ;
        End ;
    End ;
UnMarkSurfaceEdges(Edges,EdgesCount,Surface[Surf]) ;
End ;
End ;

```

```

Procedure ProcessObject(R3_SINRD_VERTEX:R3VertexType ;

```

```

R3_EYE_VERTEX:R3VertexType ;
R2Vertex:R2VertexType ;
VertexCount:Integer ; Surface:SurfaceType ;
SurfaceCount:Integer ;
Edges:EdgesType ; EdgesCount:Integer ;
Theta,Phi,Rho,D:Real ; Stats:StatsRecord) ;

```

```

Var K : Integer ;

```

```

Begin

```

```

CalcR3EyeVertex(R3_SINRD_VERTEX,VertexCount,R3_EYE_VERTEX,
Theta,Phi,Rho,D) ;

```

```

CalcR2Vertex(R3_EYE_VERTEX,VertexCount,R2Vertex,Theta,Phi,Rho,D) ;

```

```

If Stats.Hidden Then

```

```

Begin

```

```

For K := 1 To EdgesCount Do

```

```

Begin

```

```

Edges[K].NumErasures := 0 ;

```

```

Edges[K].Flag := False ;

```

```

End ;

```

```

Write(Chr(7)) ;

```

```

RemoveHiddenLines(R3_SINRD_VERTEX,R3_EYE_VERTEX,R2Vertex,Surface,Edges,
EdgesCount,VertexCount,SurfaceCount,Theta,Phi,Rho,D) ;

```

```

End ;

```

```

Write(Chr(7)) ;

```

```

ZapScreen ;

```

```

DrawObject(Edges,EdgesCount,R2Vertex) ;

```

```

If Stats.Axis Then DrawAxes(Theta,Phi,Rho,D) ;

```

```

End ;

```

```

Procedure Initialize(Var R3_SINRD_VERTEX:R3VertexType ;

```

```

Var R3_EYE_VERTEX:R3VertexType ;

```

```

Var Surface:SurfaceType ;

```

```

Var Edges:EdgesType ;

```

```

ObjectName:FileNameType ;

```

```

Var Theta,Phi,Rho,D:Real ;

```

```

Var EdgeCount,SurfaceCount,VertexCount:Integer ;

```

```

Var Stats:StatsRecord) ;

```

```

Var Infile : Text ; k:Integer ;

```

```

Begin

```

```

Assign(Infile,ObjectName + '.Ver') ;

```

```

Reset(Infile) ;

LoadVertex(Infile,R3_SINRD_VERTEX,VertexCount) ;
Close(Infile) ;

Assign(Infile,ObjectName + '.Sur') ;
Reset(Infile) ;

LoadSurface(Infile,Surface,SurfaceCount) ;
Close(Infile) ;

Theta := Pi/4 ;
Phi   := Pi/2 ;
Rho   := 425 ;
D     := 1549 ;

With Stats Do
Begin
    StatsOn    := True ;
    Axis       := False ;
    Hidden     := True ;
    ObjName    := ObjectName ;
    Th         := Theta ;
    Ph         := Phi ;
    Rh         := Rho ;
    Dist       := D ;

End ;

BuildEdgeTable(Edges,EdgesCount,Surface,SurfaceCount) ;

End ;

Begin

If ParamCount = 0 Then
Begin
    TextMode(C80) ;
    Write('Object : ') ;
    ReadLn(ObjectName) ;
End
Else
ObjectName := ParamStr(1) ;

Initialize(R3_SINRD_VERTEX,R3_EYE_VERTEX,Surface,Edges,ObjectName,
    Theta,Phi,Rho,D,EdgesCount,SurfaceCount,VertexCount,
    Stats) ;

ZapScreen ;

Repeat

    Stats.Th := Theta ; Stats.Ph := Phi ;

```



```

Stats.Rh := Rho ; Stats.Dist := D ;

If Stats.StatsOn Then ShowStats(Stats) Else EraseStats(Stats) ;

GetKeySequence(A,B) ;

If Ord(A) = 27 Then
  FunctKey := DecodeFunctionKey(B) ;

If Ord(A) = 27 Then
Begin
  Case FunctKey Of

    01 : WrapAroundDec(Theta,DelTheta,0,2*Pi) ;
    02 : WrapAroundInc(Theta,DelTheta,0,2*Pi) ;
    03 : WrapAroundDec(Phi,DelPhi,0,Pi) ;
    04 : WrapAroundInc(Phi,DelPhi,0,Pi) ;
    05 : Dec(Rho,DelRho) ;
    06 : Inc(Rho,DelRho) ;
    07 : Dec(D,DelD) ;
    08 : Inc(D,DelD) ;

    09 : ProcessObject(R3_SINRD_VERTEX,R3_EYE_VERTEX,R2Vertex,
      VertexCount,Surface,SurfaceCount,
      Edges,EdgesCount,Theta,Phi,Rho,D,Stats) ;

    11 : WrapAroundDec(Theta,DelTheta*10,0,2*Pi) ;
    12 : WrapAroundInc(Theta,DelTheta*10,0,2*Pi) ;
    13 : WrapAroundDec(Phi,DelPhi*10,0,Pi) ;
    14 : WrapAroundInc(Phi,DelPhi*10,0,Pi) ;
    15 : Dec(Rho,DelRho*10) ;
    16 : Inc(Rho,DelRho*10) ;
    17 : Dec(D,DelD*10) ;
    18 : Inc(D,DelD*10) ;
    21 : Stats.StatsOn := Not Stats.StatsOn ;
    22 : Stats.Axis := Not Stats.Axis ;
    23 : Stats.Hidden := Not Stats.Hidden ;

  End ;
End ;

Until (FunctKey = 10) And (Ord(A) = 27) ;

ExitProgram ;

End.

```

APPENDIX G

The following is the source code written in Turbo Pascal for algorithm number three. Refer to appendix D for instructions on how to run the program.

Program Hidden_Line_Removal_03 (Input,Output) ;

{ Language/Compiler : TURBO Pascal 3.01A
Computer : IBM PC or compatibles

Copyright(c) : Permission is granted by the author to use, reproduce, modify all or part of the computer programs contained herein for the readers own personal use. This permission is not to be construed as a license to distribute or sell all or any part of the programs to others in any shape or form.

Disclaimer : The author has spent a great deal of time and effort in preparing this thesis and the programs contained herein. This includes many hours of research, development and testing of the programs to determine their effectiveness. The author shall not be liable in any event for incidental or consequential damages in connection with or arising out of the furnishing, performance or use of all or any portion of these programs.

)

Const

{ Palette 0 color constants. }

Black = 0 ;
Green = 1 ;
Red = 2 ;
Brown = 3 ;

{ Display window boundary constants. }

Xmin = 1 ;
Xmax = 310 ;
Ymin = 20 ;
Ymax = 199 ;

{ Roughly the center of the display window. }

CenterX = 155 ;
CenterY = 89 ;


```

LineType      = Record
                end1,end2 : R2Vector ;
                End ;

VerticesType  = Array[1..MaxSurfVertices] Of R2Vector ;

SurfaceRecord = Record
                Vertices   : Array[1..MaxSurfVertices] of Integer ;
                NumVertices : Integer ;
                End ;

R3VertexType  = Array[1..VertexMax] Of R3Vector ;
R2VertexType  = Array[1..VertexMax] Of R2Vector ;
SurfaceType   = Array[1..SurfaceMax] Of SurfaceRecord ;

FileNameType  = String[8] ;

StatsRecord   = Record
                StatsOn,Axis,Hidden : Boolean ;
                Th,Ph,Rh,Dist : Real ;
                ObjName : FileNameType ;
                End ;

```

Var

```

R3_SINRD_VERTEX      : R3VertexType ;
{ R3_SINRD_VERTEX[i] contains the standard three-dimensional coordinates for
  the i-th vertex. }

R3_EYE_VERTEX        : R3VertexType ;
{ R3_EYE_VERTEX[i] contains the eye three-dimensional coordinates for the
  i-th vertex. }

R2Vertex             : R2VertexType ;
{ R2Vertex[i] contains the two-dimensional screen coordinates for the i-th
  vertex. }

Surface              : SurfaceType ;
{ Surface[i].NumVertices contains the j-th vertice in the formulation of the
  i-th surface. }

Edges                : EdgesType ;
{ Edges[i].nv1 contains the number of the first vertex defining the
  i-th edge.

  Edges[i].nv2 contains the number of the second vertex defining the
  i-th edge.

```

Edges[i].Erasure[j].t1 contains the the first endpoint defining the j-th erasure of the i-th edge.

Edges[i].Erasure[j].t2 contains the second endpoint defining the j-th erasure of the i-th edge.

Edges[i].NumErasures contains the number of erasures currently on the i-th edge.

Edges[i].BackEdge is a back edge flag for the i-th edge.

If Edges[i].BackEdge = TRUE then the i-th edge is a back edge and therefore should not be displayed.

Edges[i].Flag is a process flag. If Edges[i].Flag = TRUE then the i-th edge will not be processed (ie. tested for visibility with respect to the surfaces of the object).)

VertexCount : Integer ;

{ The actual number of vertices that form the object. }

SurfaceCount : Integer ;

{ The actual number of surfaces that form the object. }

EdgesCount : Integer ;

{ The actual number of edges that form the surface. }

Theta,Phi,Rho,D : Real ;

{ The viewing parameters. }

FunctKey : Integer ;

{ Contains the number associated with the function key that was pressed. }

A,B : Char ;

{ Holds the sequence of characters generated when a function key is pressed. }

Stats : StatsRecord ;

{ Contains the status information which is displayed on the first two lines of the display screen. }

ObjectName : FileNameType ;

{ Contains the name of the object being displayed. }

(\$I GRAPH1.TPU)

(\$I GRAPH2.TPU)

```
Procedure DrawEdge(Edge:EdgesRecord ; R2Vertex:R2VertexType ; Color:Integer) ;
```

```
Var x1,y1,x2,y2,q1,q2 : Real ; k : Integer ;
```

```
Begin
```

```
  If Not Edge.BackEdge Then
```

```
    Begin
```

```
      x1 := R2Vertex[Edge.nv1].x ; y1 := R2Vertex[Edge.nv1].y ;
```

```
      x2 := R2Vertex[Edge.nv2].x ; y2 := R2Vertex[Edge.nv2].y ;
```

```
      If Edge.NumErasures = 0 Then
```

```
        Clip(x1,y1,x2,y2,Color,Xmin,Xmax,Ymin,Ymax)
```

```
      Else
```

```
        Begin
```

```
          SortErasure(Edge.Erasure,Edge.NumErasures) ;
```

```
          q1 := Edge.Erasure[1].t1 ;
```

```
          Clip(x1,y1,x1+q1*(x2-x1),y1+q1*(y2-y1),Color,Xmin,Xmax,Ymin,Ymax) ;
```

```
          k := 1 ;
```

```
          While k <= Edge.NumErasures - 1 Do
```

```
            Begin
```

```
              q1 := Edge.Erasure[k].t2 ;
```

```
              q2 := Edge.Erasure[k].t1 ;
```

```
              Clip(x1+q1*(x2-x1),y1+q1*(y2-y1),x1+q2*(x2-x1),y1+q2*(y2-y1),Color,  
                Xmin,Xmax,Ymin,Ymax) ;
```

```
              k := k+1 ;
```

```
            End ;
```

```
          q1 := Edge.Erasure[Edge.NumErasures].t2 ;
```

```
          Clip(x1+q1*(x2-x1),y1+q1*(y2-y1),x2,y2,Color,Xmin,Xmax,Ymin,Ymax) ;
```

```
        End ;
```

```
    End ;
```

```
End ;
```

```
Procedure CalcNormalVector(R3_SINRD_VERTEX:R3VertexType ;  
                          Surface:SurfaceRecord ;  
                          Var Normal:R3Vector) ;
```

```
Var a,b : R3Vector ;
```

```
Begin
```

```
  R3Diff(R3_SINRD_VERTEX[Surface.Vertices[2]],  
        R3_SINRD_VERTEX[Surface.Vertices[1]],a) ;
```

```
  R3Diff(R3_SINRD_VERTEX[Surface.Vertices[3]],
```

```
R3_STNRD_VERTEX[Surface.Vertices[1]],b) ;
```

```
Normal.x := a.y * b.z - b.y * a.z ;
```

```
Normal.y := b.x * a.z - a.x * b.z ;
```

```
Normal.z := a.x * b.y - b.x * a.y ;
```

```
End ;
```

```
Procedure AddEdge(Var Edges:EdgesType ; Var EdgesCount:Integer ;
                  nv1,nv2,NumErasures:Integer ; BackEdge,Flag:Boolean ) ;
```

```
Begin
```

```
If EdgesCount < EdgesMax Then
```

```
Begin
```

```
EdgesCount := EdgesCount + 1 ;
```

```
If nv1 > nv2 Then SwitchIntegers(nv1,nv2) ;
```

```
Edges[EdgesCount].nv1 := nv1 ;
```

```
Edges[EdgesCount].nv2 := nv2 ;
```

```
Edges[EdgesCount].BackEdge := BackEdge ;
```

```
Edges[EdgesCount].Flag := Flag ;
```

```
Edges[EdgesCount].NumErasures := NumErasures ;
```

```
End ;
```

```
End ;
```

```
Procedure BuildEdgeTable(Var Edges:EdgesType ; Var EdgesCount:Integer ;
                        Surface:SurfaceType ; SurfaceCount:Integer) ;
```

```
Var j,k,nv1,nv2,EdgeNumber : Integer ; Found : Boolean ;
```

```
Begin
```

```
EdgesCount := 0 ;
```

```
For j := 1 To SurfaceCount Do
```

```
Begin
```

```
For k := 1 To Surface[j].NumVertices Do
```

```
Begin
```

```
nv1 := Surface[j].Vertices[k] ;
```

```
nv2 := Surface[j].Vertices[k+1] ;
```

```
FindEdgeNumber(Edges,EdgesCount,nv1,nv2,EdgeNumber,Found) ;
```

```
If Not Found Then AddEdge(Edges,EdgesCount,nv1,nv2,0,False,False) ;
```

```
End ;
```

```
End ;
```

```
End ;
```

```

Procedure DrawObject(Edges:EdgesType ; EdgesCount:Integer ;
                    R2Vertex:R2VertexType) ;

Var k : Integer ;

Begin
  For k := 1 To EdgesCount Do DrawEdge(Edges[k],R2Vertex,Green) ;
End ;

Procedure RemoveHiddenLines(R3_SINRD_VERTEX,R3_EYE_VERTEX:R3VertexType ;
                           R2Vertex:R2VertexType ; Surface:SurfaceType ;
                           Var Edges:EdgesType ; EdgesCount,VertexCount,
                           SurfaceCount : Integer ; Theta,Phi,Rho,D:Real) ;

Var Surf,K,NumInt,ntv1,ntv2,NumSurfVertices,EdgeNumber : Integer ;
  e,f,g,h,minpolyx,maxpolyx,minpolyy,maxpolyy : Real ;
  TestEdge,PolyEdge : LineType ;
  p1,p2 : R2Vector ;
  PolyPoints : VerticesType ;
  h1,h2,h3,View,Normal : R3Vector ;
  Dependent,Found,Behind1,Behind2,Insidel,Inside2 : Boolean ;

Begin
  For Surf := 1 To SurfaceCount Do
  Begin
    CalcViewVector(Surface[Surf],R3_SINRD_VERTEX,Theta,Phi,Rho,View) ;
    CalcNormalVector(R3_SINRD_VERTEX,Surface[Surf],Normal) ;

    If R3DotProduct(View,Normal) > 0 Then
    Begin
      NumSurfVertices := Surface[Surf].NumVertices ;

      For K := 1 To NumSurfVertices+1 Do
        PolyPoints[K] := R2Vertex[Surface[Surf].Vertices[K]] ;

      MarkSurfaceEdges(Edges,EdgesCount,Surface[Surf]) ;

      RectangularBoundry(NumSurfVertices,PolyPoints,minpolyx,maxpolyx,
                        minpolyy,maxpolyy) ;

      h1 := R3_EYE_VERTEX[Surface[Surf].Vertices[1]] ;
      h2 := R3_EYE_VERTEX[Surface[Surf].Vertices[2]] ;
      h3 := R3_EYE_VERTEX[Surface[Surf].Vertices[3]] ;

      CalcR3Plane(h1,h2,h3,e,f,g,h) ;

      For EdgeNumber := 1 To EdgesCount Do
      Begin
        If ( Not Edges[EdgeNumber].BackEdge) And

```



```

( Not Edges[EdgeNumber].Flag ) Then
Begin

ntv1 := Edges[EdgeNumber].nv1 ;
ntv2 := Edges[EdgeNumber].nv2 ;

If Edges[EdgeNumber].Flag Then
  Nothing
Else
Begin
  TestEdge.end1 := R2Vertex[ntv1] ;
  TestEdge.end2 := R2Vertex[ntv2] ;

  Behind1 := BehindPlane(R3_EYE_VERTEX[ntv1],e,f,g,h) ;
  Behind2 := BehindPlane(R3_EYE_VERTEX[ntv2],e,f,g,h) ;

  If ((Not Behind1) And (Not Behind2)) Or
    (OutsideBox(TestEdge,minpolyx,maxpolyx,minpolyy,maxpolyy)) Then
  Nothing
  Else
  Begin
    FindIntersectionPoints(TestEdge,NumSurfVertices,PolyPoints,
      p1,p2,NumInt,Dependent) ;

    If Not Dependent Then
    Begin
      Inside1 := InsidePoly(NumSurfVertices,PolyPoints,
        TestEdge.end1) ;
      Inside2 := InsidePoly(NumSurfVertices,PolyPoints,
        TestEdge.end2) ;
      ProcessTestEdge(TestEdge,EdgeNumber,
        R3_EYE_VERTEX[ntv1],R3_EYE_VERTEX[ntv2],
        R2Vertex,p1,p2,NumInt,NumSurfVertices,
        PolyPoints,e,f,g,h,Theta,Phi,Rho,D,
        Edges,Behind1,Behind2,Inside1,Inside2)

      End ;
    End ;
  End ;
End ;
End ;
End ;
UnMarkSurfaceEdges(Edges,EdgesCount,Surface[Surf]) ;
End ;
End ;
End ;

```

```

Procedure Set_BackEdge_Flags(Var Edges:EdgesType ; EdgesCount:Integer ;
  R3_STNRD_VERTEX:R3VertexType ;
  Surface:SurfaceType ; SurfaceCount:Integer ;
  Theta,Phi,Rho,D:Real) ;

```

```

Var View,Normal : R3Vector ; EdgeNumber,J,K : Integer ;
Found : Boolean ;

```

```

Begin

```

```
For K := 1 To EdgesCount Do Edges[K].BackEdge := True ;
```

```
For K := 1 To SurfaceCount Do
```

```
Begin
```

```
  CalcViewVector(Surface[K],R3_SINRD_VERTEX,Theta,Phi,Rho,View) ;
```

```
  CalcNormalVector(R3_SINRD_VERTEX,Surface[K],Normal) ;
```

```
  If R3DotProduct(View,Normal) > 0 Then
```

```
  Begin
```

```
    For J := 1 To Surface[K].NumVertices Do
```

```
    Begin
```

```
      FindEdgeNumber(Edges,EdgesCount,Surface[K].Vertices[J],  
                    Surface[K].Vertices[J+1],EdgeNumber,Found) ;
```

```
      Edges[EdgeNumber].BackEdge := False ;
```

```
    End ;
```

```
  End ;
```

```
End ;
```

```
End ;
```

```
Procedure ProcessObject(R3_SINRD_VERTEX:R3VertexType ;
```

```
                        R3_EYE_VERTEX:R3VertexType ;
```

```
                        R2Vertex:R2VertexType ;
```

```
                        VertexCount:Integer ; Surface:SurfaceType ;
```

```
                        SurfaceCount:Integer ;
```

```
                        Edges:EdgesType ; EdgesCount:Integer ;
```

```
                        Theta,Phi,Rho,D:Real ; Stats:StatsRecord) ;
```

```
Var K : Integer ;
```

```
Begin
```

```
  CalcR3EyeVertex(R3_SINRD_VERTEX,VertexCount,R3_EYE_VERTEX,  
                 Theta,Phi,Rho,D) ;
```

```
  CalcR2Vertex(R3_EYE_VERTEX,VertexCount,R2Vertex,Theta,Phi,Rho,D) ;
```

```
  If Stats.Hidden Then
```

```
  Begin
```

```
    For K := 1 To EdgesCount Do
```

```
    Begin
```

```
      Edges[K].NumErasures := 0 ;
```

```
      Edges[K].Flag       := False ;
```

```
      Edges[K].BackEdge  := False ;
```

```
    End ;
```

```
  Set_BackEdge_Flags(Edges,EdgesCount,R3_SINRD_VERTEX,Surface,  
                    SurfaceCount,Theta,Phi,Rho,D) ;
```

```
  Write(Chr(7)) ;
```

```
  RemoveHiddenLines(R3_SINRD_VERTEX,R3_EYE_VERTEX,R2Vertex,Surface,Edges,  
                   EdgesCount,VertexCount,SurfaceCount,Theta,Phi,Rho,D) ;
```

```
End ;
```

```

ZapScreen ;
Write(Chr(7)) ;
DrawObject(Edges,EdgesCount,R2Vertex) ;

If Stats.Axis Then DrawAxes(Theta,Phi,Rho,D) ;

```

```
End ;
```

```

Procedure Initialize(Var R3_STNRD_VERTEX:R3VertexType ;
                    Var R3_EYE_VERTEX:R3VertexType ;
                    Var Surface:SurfaceType ;
                    Var Edges:EdgesType ;
                    ObjectName:FileNameType ;
                    Var Theta,Phi,Rho,D:Real ;
                    Var EdgeCount,SurfaceCount,VertexCount:Integer ;
                    Var Stats:StatsRecord) ;

```

```
Var Infile : Text ; k:Integer ;
```

```
Begin
```

```

Assign(Infile,ObjectName + '.Ver') ;
Reset(Infile) ;

```

```

LoadVertex(Infile,R3_STNRD_VERTEX,VertexCount) ;
Close(Infile) ;

```

```

Assign(Infile,ObjectName + '.Sur') ;
Reset(Infile) ;

```

```

LoadSurface(Infile,Surface,SurfaceCount) ;
Close(Infile) ;

```

```

Theta := Pi/4 ;
Phi   := Pi/2 ;
Rho   := 425 ;
D     := 1549 ;

```

```
With Stats Do
```

```
Begin
```

```

StatsOn   := True ;
Axis      := False ;
Hidden    := True ;
ObjName   := ObjectName ;
Th        := Theta ;
Ph        := Phi ;
Rh        := Rho ;
Dist     := D ;

```

```
End ;
```

```
BuildEdgeTable(Edges,EdgesCount,Surface,SurfaceCount) ;
```

```
End ;
```

APPENDIX H

The following procedures and functions are located in the include files GRAPH1.TPU and GRAPH2.TPU. These procedures and functions are all common to the programs HLR01.PAS, HLR02.PAS, and HLR03.PAS.

```
{ GRAPH1.TPU }
```

```
Function Deg(x:Real):Real ;
```

```
Begin
```

```
  Deg := x * 180/pi ;
```

```
End ;
```

```
Procedure ShowStats(Stats:StatsRecord) ;
```

```
Begin
```

```
  With Stats Do
```

```
  Begin
```

```
    If StatsOn Then
```

```
      Begin
```

```
        GotoXY(01,01) ;
```

```
        Write('Theta Phi Rho D Object HLR Axes') ;
```

```
        GotoXY(01,02) ;
```

```
        Write(' ') ;
```

```
        GotoXY(03,02) ; Write(Deg(Th):3:0) ;
```

```
        GotoXY(07,02) ; Write(Deg(Ph):3:0) ;
```

```
        GotoXY(11,02) ; Write(Rh:5:0) ;
```

```
        GotoXY(17,02) ; Write(Dist:5:0) ;
```

```
        GotoXY(23,02) ; Write(ObjName) ;
```

```
        GotoXY(32,02) ; If Hidden Then Write('ON') ELSE Write('OFF') ;
```

```
        GotoXY(36,02) ; If Axis Then Write('ON') ELSE Write('OFF') ;
```

```
      End ;
```

```
    End ;
```

```
End ;
```

```
Procedure EraseStats(Stats:StatsRecord) ;
```

```
Begin
```

```
  GotoXY(01,01) ; WriteLn(' ') ;
```

```
  GotoXY(01,02) ; WriteLn(' ') ;
```

```
End ;
```

```
Procedure LoadVertex(Var Infile:Text ; Var R3_SINRD_VERTEX:R3VertexType ;
```

```
Var VertexCount:Integer) ;
```

```
{ This procedure reads the vertex definition file from the disk. The
vertex definition file contains a listing of the standard three-dimensional
coordinates for each vertex. Examples of vertex definition files are
HOUSE.VER, DODEC.VER SIXTY.VER, SPHERE.VER, and TETRA.VER which are all
located on the program disk. }
```

```
Begin
```

```
VertexCount := 0 ;
```

```
While (Not Eof(InFile)) And (VertexCount < VertexMax) Do
```

```
Begin
```

```
VertexCount := VertexCount + 1 ;
```

```
ReadLn(InFile,R3_SINRD_VERTEX[VertexCount].x,
        R3_SINRD_VERTEX[VertexCount].y,
        R3_SINRD_VERTEX[VertexCount].z ) ;
```

```
End ;
```

```
End ;
```

```
Procedure LoadSurface(Var Infile:Text ; Var Surface:SurfaceType ;
                        Var SurfaceCount:Integer) ;
```

```
{ This procedure reads the surface definition file from the disk. The
surface definition file contains a listing of the vertices that form each
surface. Examples of surfac definition files are HOUSE.SUR, DODEC.SUR,
SIXTY.SUR, SPHERE.SUR, and TETRA.SUR which are all located on the
program disk. }
```

```
Var k : Integer ;
```

```
Begin
```

```
SurfaceCount := 0 ;
```

```
While (Not Eof(Infile)) And (SurfaceCount < SurfaceMax) Do
```

```
Begin
```

```
SurfaceCount := SurfaceCount + 1 ;
```

```
Read(Infile,Surface[SurfaceCount].NumVertices) ;
```

```
For k := 1 To (Surface[SurfaceCount].NumVertices + 1) Do
```

```
Read(Infile,Surface[SurfaceCount].Vertices[k]) ;
```

```
ReadLn(Infile) ;
```

```
End ;
```

```
End ;
```

```
Procedure ExitProgram ;
```

```

Begin
  TextMode(C80) ;
End ;

```

```

Procedure DrawBorder ;

```

```

Begin
  Draw(Xmin, Ymin, Xmax, Ymin, BorderColor) ;
  Draw(Xmax, Ymin, Xmax, Ymax, BorderColor) ;
  Draw(Xmax, Ymax, Xmin, Ymax, BorderColor) ;
  Draw(Xmin, Ymax, Xmin, Ymin, BorderColor) ;
End ;

```

```

Procedure ZapScreen ;

```

```

Begin
  GraphColorMode ;
  Palette(0) ;
  ShowStats(Stats) ;
  DrawBorder ;
End ;

```

```

Procedure R3Diff(a,b:R3Vector ; Var diff:R3Vector) ;

```

```

Begin
  diff.x := a.x - b.x ;
  diff.y := a.y - b.y ;
  diff.z := a.z - b.z ;
End ;

```

```

Procedure EyeXYZ(Theta,Phi,Rho,D:Real ; Standard:R3Vector ;
  Var Eye:R3Vector) ;

```

{ Given the standard coordinates of a point in three-space, this procedure calculates the eye coordinates for that particular point. }

```

Var s1,s2,c1,c2,x,y,z : Real ;

```

```

Begin
  x := Standard.x ; y := Standard.y ; z := Standard.z ;

  s1 := Sin(Theta) ;
  c1 := Cos(Theta) ;
  s2 := Sin(Phi) ;
  c2 := Cos(Phi) ;

  Eye.x := -x * s1 + y * c1 ;
  Eye.y := -x * c1 * c2 - y * s1 * c2 + z * s2 ;
  Eye.z := -x * c1 * s2 - y * s1 * s2 - z * c2 + Rho ;
End ;

```

```
Procedure ScreenXY(Theta,Phi,Rho,D:Real ; R3Eye:R3Vector ;
                    Var Screen:R2Vector) ;
```

{ Given the eye coordinates for a point in three-space, this procedure calculates the screen coordinates for that particular point. }

```
Begin
```

```
Screen.x := CenterX + D * R3Eye.x/R3Eye.z ;
Screen.y := CenterY - D * R3Eye.y/R3Eye.z * AspectRatio ;
```

```
End ;
```

```
Procedure WrapAroundDec(Var x:Real; Delta,Lower,Upper:Real) ;
```

```
Begin
```

```
x := x - Delta ;
If x < Lower Then x := Upper ;
```

```
End ;
```

```
Procedure WrapAroundInc(Var x:Real; Delta,Lower,Upper:Real) ;
```

```
Begin
```

```
x := x + Delta ;
If x > Upper Then x := Lower ;
```

```
End ;
```

```
Procedure Inc(Var x:Real; Delta:Real) ;
```

```
Begin
```

```
x := x + Delta ;
```

```
End ;
```

```
Procedure Dec(Var x:Real; Delta:Real) ;
```

```
Begin
```

```
x := x - Delta ;
```

```
End ;
```

```
Procedure Clip(x1,y1,x2,y2 : Real ; Color,Xmin,Xmax,Ymin,Ymax : Integer) ;
```

```
Label Return ;
```

```
Type Edge = (Left,Right,Bottom,Top) ;
```

```
OutCode = Set of Edge ;
```

```
Var c,c1,c2 : OutCode ;
```

```
x,y : Real ;
```

```
Procedure Code(x,y : Real ; Var c : OutCode) ;
```

```
Begin
```

```
C := [ ] ;
```

```

If x < Xmin Then c := [Left]
Else
If x > Xmax Then c := [Right] ;

If y < Ymin Then c := c + [Top]
Else
If y > Ymax Then c := c + [Bottom] ;
End ;

```

```
Begin
```

```
Code(x1,y1,c1) ;
Code(x2,y2,c2) ;
```

```
While (c1  $\diamond$  []) Or (c2  $\diamond$  []) Do
```

```
Begin
```

```
  If (c1 * c2)  $\diamond$  [] Then Goto Return ;
```

```
  c := c1 ; If c = [] Then c := c2 ;
```

```
  If Left In c Then
```

```
  Begin
```

```
    y := y1 + (y2-y1)*(Xmin-x1)/(x2-x1) ;
```

```
    x := Xmin ;
```

```
  End
```

```
  Else
```

```
  If Right In c Then
```

```
  Begin
```

```
    y := y1 + (y2-y1)*(Xmax-x1)/(x2-x1) ;
```

```
    x := Xmax ;
```

```
  End
```

```
  Else
```

```
  If Bottom In c Then
```

```
  Begin
```

```
    x := x1 + (x2-x1)*(Ymax-y1)/(y2-y1) ;
```

```
    y := Ymax ;
```

```
  End
```

```
  Else
```

```
  If Top In c Then
```

```
  Begin
```

```
    x := x1 + (x2-x1)*(Ymin-y1)/(y2-y1) ;
```

```
    y := Ymin ;
```

```
  End ;
```

```
  If c = c1 Then
```

```
  Begin
```

```
    x1 := x ; y1 := y ; Code(x,y,c1) ;
```

```
  End
```

```
  Else
```

```
  Begin
```

```
    x2 := x ; y2 := y ; Code(x,y,c2) ;
```

```
  End ;
```

```
End ;
```



```
Draw(Trunc(x1),Trunc(y1),Trunc(x2),Trunc(y2),Color) ;
```

```
Return ;
```

```
End ;
```

```
Procedure DrawAxes(Theta,Phi,Rho,D:Real) ;
```

```
Const AxisLength = 10 ;
```

```
Var i3s,j3s,k3s,i3e,j3e,k3e: R3Vector ; i2,j2,k2 : R2Vector ;
```

```
Begin
```

```

i3s.x := AxisLength ; i3s.y := 00 ;      i3s.z := 00 ;
j3s.x := 00           ; j3s.y := AxisLength ; j3s.z := 00 ;
k3s.x := 00           ; k3s.y := 00 ;      k3s.z := AxisLength ;
```

```
EyeXYZ(Theta,Phi,Rho,D,i3s,i3e) ;
```

```
EyeXYZ(Theta,Phi,Rho,D,j3s,j3e) ;
```

```
EyeXYZ(Theta,Phi,Rho,D,k3s,k3e) ;
```

```
ScreenXY(Theta,Phi,Rho,D,i3e,i2) ;
```

```
ScreenXY(Theta,Phi,Rho,D,j3e,j2) ;
```

```
ScreenXY(Theta,Phi,Rho,D,k3e,k2) ;
```

```
Clip(CenterX,CenterY,i2.x,i2.y,Green,Xmin,Xmax,Ymin,Ymax) ;
```

```
Clip(CenterX,CenterY,j2.x,j2.y,Red ,Xmin,Xmax,Ymin,Ymax) ;
```

```
Clip(CenterX,CenterY,k2.x,k2.y,Brown,Xmin,Xmax,Ymin,Ymax) ;
```

```
End ;
```

```
Function R3DotProduct(Vector1,Vector2:R3Vector):Real ;
```

```
Begin
```

```

R3DotProduct := Vector1.x * Vector2.x +
                Vector1.y * Vector2.y +
                Vector1.z * Vector2.z  ;
```

```
End ;
```

```
Procedure GetKeySequence(Var A,B:Char) ;
```

```
Begin
```

```
Read(Kbd,A) ;
```

```
If Ord(A) = 27 Then Read(Kbd,B) ;
```

```
End ;
```

```
Function DecodeFunctionKey(A : Char):Integer ;
```

```
{ This function is used to decode the two consecutive characters
returned after a function key has been pressed. }
```

```

Begin
  If Ord(A) > 103 Then
    DecodeFunctionKey := Ord(A)-103+30 (Alt F1 - F10 : returns 31-40 )
  Else
  If Ord(A) > 93 Then
    DecodeFunctionKey := Ord(A)-93+20 (Ctrl F1 - F10 : returns 21-30 )
  Else
  If Ord(A) > 68 Then
    DecodeFunctionKey := Ord(A)-83+10 (Shift F1 - F10 : returns 11-20 )
  Else
  If Ord(A) > 58 Then
    DecodeFunctionKey := Ord(A)-58 ; ( F1 - F10 : returns 01-10 )
  End ;

```

Procedure DrawBox(x1,y1,x2,y2 : Real ; Color:Integer) ;

Begin

```

  Clip(x1,y1,x2,y1,Color,Xmin,Xmax,Ymin,Ymax) ;
  Clip(x2,y1,x2,y2,Color,Xmin,Xmax,Ymin,Ymax) ;
  Clip(x2,y2,x1,y2,Color,Xmin,Xmax,Ymin,Ymax) ;
  Clip(x1,y2,x1,y1,Color,Xmin,Xmax,Ymin,Ymax) ;

```

End ;

Procedure CalcViewVector(Surface:SurfaceRecord ;
 R3_SINRD_VERTEX:R3VertexType ;
 Theta,Phi,Rho:Real ;
 Var ViewVector:R3Vector) ;

{ Given a particular surface, this procedure computes the view vector
 which is the vector directed from the first vertex of the surface to the
 viewpoint whose spherical coordinates are (Theta,Phi,Rho). }

Var x : R3Vector ;

Begin

```

  x.x := Rho * Sin(Phi) * Cos(Theta) ;
  x.y := Rho * Sin(Phi) * Sin(Theta) ;
  x.z := Rho * Cos(Phi) ;

```

```

  R3Diff(x,R3_SINRD_VERTEX[Surface.Vertices[1]],ViewVector) ;

```

End ;

{ GRAPH2.TPU }

Procedure CalcR2Vertex(R3_EYE_VERTEX:R3VertexType ; VertexCount:Integer ;
 Var R2Vertex:R2VertexType ; Theta,Phi,Rho,D:Real) ;

```
Var k : Integer ;
```

```
Begin
```

```
For k := 1 To VertexCount Do
  ScreenXY(Theta,Phi,Rho,D,R3_EYE_VERTEX[k],R2Vertex[k]) ;
```

```
End ;
```

```
Procedure FindSlope(x1,y1,x2,y2:Real ; Var m:Real ;
  Var Undefined:Boolean) ;
```

```
Begin
```

```
  Undefined := True ;
```

```
  If (x2 - x1) <> 0 Then
```

```
    Begin
```

```
      m := (y2 - y1)/(x2 - x1) ;
```

```
      Undefined := False ;
```

```
    End ;
```

```
End ;
```

```
Function Find_t(x1,y1,x2,y2,px,py:Real):Real ;
```

(A line segment with endpoints (x1,y1) and (x2,y2) can be defined by a parametric equation $R(t) = (x1,y1) + t(x2-x1,y2-y1)$. Given a point (px,py) not necessarily on this line segment but reasonably close, this procedure calculates a value for t such that R(t) is approximately equal to the point (px,py).)

```
Var slope : Real ; undefined : Boolean ;
```

```
Begin
```

```
  FindSlope(x1,y1,x2,y2,slope,undefined) ;
```

```
  slope := Abs(slope) ;
```

```
  If Abs(slope - 1) < 0.01 Then
```

```
    Find_t := ((px - x1)/(x2 - x1) + (py - y1)/(y2 - y1))/2
```

```
  Else
```

```
    If (slope >= 0) And (slope < 1) Then
```

```
      Find_t := (px - x1)/(x2 - x1)
```

```
    Else
```

```
      Find_t := (py - y1)/(y2 - y1) ;
```

```
End ;
```

```
Function Min(x,y:Real):Real ;
```

```
Begin
```

```

If x <= y Then Min := x Else Min := y ;
End ;

```

```

Function Max(x,y:Real):Real ;

```

```

Begin
  If x >= y Then Max := x Else Max := y ;
End ;

```

```

Function R2VectorsEqual(v1,v2:R2Vector):Boolean ;

```

```

Begin
  R2VectorsEqual :=
    (Abs(v1.x - v2.x) < 0.001) And (Abs(v1.y - v2.y) < 0.001) ;
End ;

```

```

Procedure AddErasure(Var Edges:EdgesType ; EdgeNumber:Integer ;
  newt1,newt2:Real ) ;

```

```

Var NumErasures : Integer ;

```

```

Begin

```

```

  NumErasures := Edges[EdgeNumber].NumErasures ;

```

```

  If NumErasures < ErasureMax Then

```

```

    Begin

```

```

      NumErasures := NumErasures + 1 ;

```

```

      Edges[EdgeNumber].Erasure[NumErasures].t1 := Min(newt1,newt2) ;

```

```

      Edges[EdgeNumber].Erasure[NumErasures].t2 := Max(newt1,newt2) ;

```

```

      Edges[EdgeNumber].NumErasures := NumErasures ;

```

```

    End ;

```

```

  End ;

```

```

Procedure SwitchReals(Var a,b:Real) ;

```

```

Var Temp : Real ;

```

```

Begin

```

```

  Temp := a ; a := b ; b := Temp ;

```

```

End ;

```

```

Procedure SwitchIntegers(Var a,b:Integer) ;

```

```

Var Temp : Integer ;

```

```

Begin

```

```

  Temp := a ; a := b ; b := Temp ;

```

```
End ;
Procedure SortErasure(Var Erasure:ErasureType ; N:Integer) ;
```

```
Var i,j : Integer ;
```

```
Begin
```

```
  For i := 1 To N-1 Do
```

```
    For j := 1 To N-i Do
```

```
      If Erasure[j].t1 > Erasure[j+1].t2 Then
```

```
        Begin
```

```
          SwitchReals(Erasure[j].t1,Erasure[j+1].t1) ;
```

```
          SwitchReals(Erasure[j].t2,Erasure[j+1].t2) ;
```

```
        End ;
```

```
End ;
```

```
Procedure FindEdgeNumber(Edges:EdgesType ; EdgesCount:Integer ;
                          nv1,nv2:Integer ; Var EdgeNumber:Integer ;
                          Var Found:Boolean) ;
```

```
Begin
```

```
  Found := False ;
```

```
  If EdgesCount > 0 Then
```

```
    Begin
```

```
      EdgeNumber := 0 ;
```

```
      If nv1 > nv2 Then SwitchIntegers(nv1,nv2) ;
```

```
      Repeat
```

```
        EdgeNumber := EdgeNumber + 1 ;
```

```
        If (Edges[EdgeNumber].nv1 = nv1) And
```

```
           (Edges[EdgeNumber].nv2 = nv2) Then Found := True ;
```

```
      Until (Found) Or (EdgeNumber = EdgesCount) ;
```

```
    End ;
```

```
End ;
```

```
Procedure MarkSurfaceEdges(Var Edges:EdgesType ; EdgesCount:Integer ;
                             Surface:SurfaceRecord) ;
```

```
{ Given a particular surface, this procedure sets the process flag for all
the edges defining this surface to TRUE. Thus none of the edges that
define this surface will be tested for visibility with respect to this
surface. }
```

```
Var k,EdgeNumber : Integer ; Found : Boolean ;
```

Begin

For k := 1 To Surface.NumVertices Do

Begin

FindEdgeNumber(Edges,EdgesCount,Surface.Vertices[k],
Surface.Vertices[k+1],EdgeNumber,Found) ;

Edges[EdgeNumber].Flag := True ;

End ;

End ;

Procedure UnMarkSurfaceEdges(Var Edges:EdgesType ; EdgesCount:Integer ;
Surface:SurfaceRecord) ;

{ Given a particular surface, this procedure sets the process flag for all
the edges defining this surface to FALSE. }

Var k,EdgeNumber : Integer ; Found : Boolean ;

Begin

For k := 1 To Surface.NumVertices Do

Begin

FindEdgeNumber(Edges,EdgesCount,Surface.Vertices[k],
Surface.Vertices[k+1],EdgeNumber,Found) ;

Edges[EdgeNumber].Flag := False ;

End ;

End ;

Procedure ProcessOverlap(oldd1,oldd2,newt1,newt2:Real ;
Var uniont1,uniont2:Real ; Var Overlap:Boolean) ;

Begin

Overlap := True ;

If oldt1 > oldt2 Then SwitchReals(oldd1,oldd2) ;

If newt1 > newt2 Then SwitchReals(newt1,newt2) ;

If ((newt1 \geq oldt1) And (newt1 \leq oldt2)) And (newt2 \geq oldt2) Then

Begin

uniont1 := oldt1 ;

uniont2 := newt2 ;

End

Else

If ((newt2 \geq oldt1) And (newt2 \leq oldt2)) And (newt1 \leq oldt1) Then

Begin

uniont1 := newt1 ;

uniont2 := oldt2 ;

End

Else

```
If (newt1 <= oldt1) And (newt2 >= oldt2) Then
```

```
Begin
```

```
  uniont1 := newt1 ;
```

```
  uniont2 := newt2 ;
```

```
End
```

```
Else
```

```
Overlap := False ;
```

```
End ;
```

```
Procedure RemoveErasure(Var Edges:EdgesType ; EdgeNumber,k:Integer ) ;
```

```
Var NumErasures,p : Integer ;
```

```
Begin
```

```
  NumErasures := Edges[EdgeNumber].NumErasures ;
```

```
  p := k+1 ;
```

```
  While p <= NumErasures Do
```

```
  Begin
```

```
    Edges[EdgeNumber].Erasure[p-1] := Edges[EdgeNumber].Erasure[p] ;
```

```
    p := p+1 ;
```

```
  End ;
```

```
  NumErasures := NumErasures - 1 ;
```

```
  Edges[EdgeNumber].NumErasures := NumErasures ;
```

```
End ;
```

```
Procedure EatTestEdge(TestEdge:LineType ; EdgeNumber:Integer ;
```

```
  p1,p2:R2Vector ; R2Vertex:R2VertexType ;
```

```
  Var Edges:EdgesType) ;
```

{ Given a particular edge and a portion of that edge to be erased (called an erasure), this procedure does the following:

1. If there are no existing erasures to be preformed on this particular edge, then the erasure is stored. (ie. the very first erasure).
Goto step 3.
2. If there are existing erasures to be preformed on this particular edge, then the new erasure is compared to the existing ones.
 - a.) If the new erasure is contained completely inside one of the existing erasures then the new erasure is discarded. Goto step 3.
 - b.) If the new erasure overlaps an existing erasure, then remove the existing erasure from the erasure table and make
new erasure = (new erasure) union (existing erasure) and

repeat step 2.

c.) If the new erasure does not overlap an existing erasure, then add the new erasure to the erasure table. Goto step 3.

3. Exit Procedure

}

Label 1,2,3 ;

Var Discard,Overlap : Boolean ; NumErasures,k,nv1,nv2 : Integer ;
 x1,y1,x2,y2,newt1,newt2,union1,union2 : Real ;

Begin

If Not R2VectorsEqual(p1,p2) Then

Begin

nv1 := Edges[EdgeNumber].nv1 ;

nv2 := Edges[EdgeNumber].nv2 ;

x1 := R2Vertex[nv1].x ; y1 := R2Vertex[nv1].y ;

x2 := R2Vertex[nv2].x ; y2 := R2Vertex[nv2].y ;

newt1 := Find_t(x1,y1,x2,y2,p1.x,p1.y) ;

newt2 := Find_t(x1,y1,x2,y2,p2.x,p2.y) ;

If newt1 > newt2 Then SwitchReals(newt1,newt2) ;

NumErasures := Edges[EdgeNumber].NumErasures ;

If NumErasures = 0 Then

AddErasure(Edges,EdgeNumber,newt1,newt2)

Else

Begin

Discard := False ;

k := 0 ;

NumErasures := Edges[EdgeNumber].NumErasures ;

Repeat

k := k + 1 ;

If (newt1 >= Edges[EdgeNumber].Erasure[k].t1) And

(newt2 <= Edges[EdgeNumber].Erasure[k].t2) Then Discard := True ;

Until (Discard) Or (k = NumErasures) ;

If Not Discard Then

Begin

1: k := 0 ;

NumErasures := Edges[EdgeNumber].NumErasures ;

2: $k := k + 1$;

ProcessOverlap(Edges[EdgeNumber].Erasure[k].t1,
Edges[EdgeNumber].Erasure[k].t2,
newt1,newt2,union1,union2,Overlap) ;

If Overlap Then

Begin

RemoveErasure(Edges,EdgeNumber,k) ;

newt1 := min(union1,union2) ;

newt2 := max(union1,union2) ;

If newt1 > newt2 Then SwitchReals(newt1,newt2) ;

If Edges[EdgeNumber].NumErasures > 0 Then Goto 1 Else Goto 3 ;

End ;

If $k < \text{NumErasures}$ Then Goto 2 ;

3: AddErasure(Edges,EdgeNumber,newt1,newt2) ;

End ;

End ;

End ;

End ;

Procedure R2MidPoint(a,b:R2Vector ; Var midpoint:R2Vector) ;

Begin

midpoint.x := (a.x + b.x)/2 ;

midpoint.y := (a.y + b.y)/2 ;

End ;

Procedure R3MidPoint(a,b:R3Vector ; Var midpoint:R3Vector) ;

Begin

midpoint.x := (a.x + b.x)/2 ;

midpoint.y := (a.y + b.y)/2 ;

midpoint.z := (a.z + b.z)/2 ;

End ;

Procedure Nothing ;

Begin

End ;

Function Inbetween(a,b,x:Real):Boolean ;

Begin

Inbetween := (x \geq Min(a,b)-0.001) And (x \leq Max(a,b)+0.001) ;

End ;

```
Function R2Magnitude(Vector1:R2Vector):Real ;
```

```
Begin
  R2Magnitude := Sqrt(Sqr(Vector1.x)+Sqr(Vector1.y)) ;
End ;
```

```
Function R2Angle(v1:R2Vector):Real ;
```

```
Begin
  If (v1.x = 0) And (v1.y > 0) Then R2Angle := Pi/2
  Else
  If (v1.x = 0) And (v1.y < 0) Then R2Angle := 3*Pi/2
  Else
  If (v1.y >= 0) And (v1.x > 0) Then R2Angle := ArcTan(v1.y/v1.x)
  Else
  If (v1.y >= 0) And (v1.x < 0) Then R2Angle := ArcTan(v1.y/v1.x) + Pi
  Else
  If (v1.y < 0) And (v1.x < 0) Then R2Angle := ArcTan(v1.y/v1.x) + Pi
  Else
  If (v1.y < 0) And (v1.x > 0) Then R2Angle := ArcTan(v1.y/v1.x) + 2*Pi ;
End ;
```

```
Function Det2x2(a,b,c,d:Real):Real ;
```

```
Begin
  Det2x2 := a*d - b*c ;
End ;
```

```
Function Det3x3(a,b,c,d,e,f,g,h,i:Real):Real ;
```

```
Begin
  Det3x3 := a*(i*e - h*f) - b*(i*d - g*f) + c*(h*d - e*g) ;
End ;
```

```
Procedure CramersRule2x2(a1,b1,c1,a2,b2,c2:Real ; Var solution:R2Vector ;
  Var Inconsistent,Dependent : Boolean) ;
```

```
Var d,n1,n2 : Real ;
```

```
Begin
  Inconsistent := False ;
  Dependent := False ;

  d := Det2x2(a1,b1,a2,b2) ;
  n1 := Det2x2(c1,b1,c2,b2) ;

  If d = 0 Then
  Begin
    If n1 = 0 Then Dependent := True Else Inconsistent := True ;
  End
```

```

Else
Begin
  n2 := Det2x2(a1,c1,a2,c2) ;
  solution.x := n1/d ;
  solution.y := n2/d ;
End ;
End ;

```

```

Procedure CalcR3Plane(p1,p2,p3:R3Vector ; Var e,f,g,h : Real) ;

```

```

Var x1,y1,z1,x2,y2,z2,x3,y3,z3 : Real ;

```

```

Begin
  x1 := p1.x ; y1 := p1.y ; z1 := p1.z ;
  x2 := p2.x ; y2 := p2.y ; z2 := p2.z ;
  x3 := p3.x ; y3 := p3.y ; z3 := p3.z ;

  e := Det3x3(1,y1,z1,1,y2,z2,1,y3,z3) ;
  f := Det3x3(x1,1,z1,x2,1,z2,x3,1,z3) ;
  g := Det3x3(x1,y1,1,x2,y2,1,x3,y3,1) ;
  h := Det3x3(y1,x1,z1,y2,x2,z2,y3,x3,z3) ;

  If h < 0 Then
  Begin
    e := -e ; f := -f ; g := -g ; h := -h ;
  End ;

```

```

End ;

```

```

Procedure CalcR2Line(p1,p2:R2Vector ; Var a,b,c : Real) ;

```

```

Begin
  a := (p1.y - p2.y) ;
  b := (p2.x - p1.x) ;
  c := (p2.x * p1.y) - (p1.x * p2.y) ;

```

```

End ;

```

```

Procedure CalcR2Intersection(Edge1,Edge2:LineType ; Var Point:R2Vector ;
  Var Inconsistent,Dependent:Boolean ) ;

```

```

Var a1,b1,c1,a2,b2,c2 : Real ;

```

```

Begin
  CalcR2Line(edge1.end1,edge1.end2,a1,b1,c1) ;
  CalcR2Line(edge2.end1,edge2.end2,a2,b2,c2) ;

  CramersRule2x2(a1,b1,c1,a2,b2,c2,point,Inconsistent,Dependent) ;

```

```

End ;

```

```

Procedure R2Diff(v1,v2:R2Vector ; Var diff:R2Vector) ;
{ Number : 050 }

Begin
  diff.x := v1.x - v2.x ;
  diff.y := v1.y - v2.y ;
End ;

Function Sgn(x:Real):Integer ;

Begin
  If x > 0 Then Sgn := +1
  Else
  If x < 0 Then Sgn := -1
  Else
  Sgn := 0 ;
End ;

Function InsidePoly(NumPoints:Integer ; Points:VerticesType ;
                    Point:R2Vector):Boolean ;

{ Given a point and a polygonal surface, this function determines
  whether the point is located inside or outside of the surface. }

Label 2 ;

Var Ang,TotalAngle : Real ; K : Integer ; v1,v2 : R2Vector ;

Begin
  InsidePoly := False ;

  For K := 1 To NumPoints Do
    If R2VectorsEqual(Point,Points[K]) Then Goto 2 ;

  TotalAngle := 0 ;

  For K := 1 To NumPoints Do
    Begin
      R2Diff(Points[K] ,Point,v1) ;
      R2Diff(Points[K+1],Point,v2) ;

      Ang := Abs(R2Angle(v1) - R2Angle(v2)) ;
      If Ang > Pi Then Ang := 2*Pi - Ang ;
      If Abs(Ang-Pi) < 0.001 Then Goto 2 ;
      TotalAngle := TotalAngle + Sgn(Det2x2(v1.x,v1.y,v2.x,v2.y))*Ang ;
    End ;

  If Abs(Abs(TotalAngle) -2*Pi) < 0.001 Then InsidePoly := True ;

2:
End ;

```

```
Procedure RectangularBoundry(NumPoints:Integer ; Points:VerticesType ;
                               Var minpolyx,maxpolyx,minpolyy,maxpolyy:Real) ;
```

```
Var i,p : R2Vector ; k : Integer ;
```

```
Begin
```

```
  i := Points[1] ;
```

```
  minpolyx := i.x ; maxpolyx := i.x ;
```

```
  minpolyy := i.y ; maxpolyy := i.y ;
```

```
  For k := 2 To NumPoints Do
```

```
  Begin
```

```
    p := Points[K] ;
```

```
    If p.x < minpolyx Then minpolyx := p.x ;
```

```
    If p.x > maxpolyx Then maxpolyx := p.x ;
```

```
    If p.y < minpolyy Then minpolyy := p.y ;
```

```
    If p.y > maxpolyy Then maxpolyy := p.y ;
```

```
  End ;
```

```
End ;
```

```
Function OutsideBox(edge:LineType ;
                     minpolyx,maxpolyx,minpolyy,maxpolyy:Real):Boolean ;
```

```
Begin
```

```
  OutsideBox := ((edge.end1.x <= minpolyx) And (edge.end2.x <= minpolyx)) Or
                ((edge.end1.x >= maxpolyx) And (edge.end2.x >= maxpolyx)) Or
                ((edge.end1.y <= minpolyy) And (edge.end2.y <= minpolyy)) Or
                ((edge.end1.y >= maxpolyy) And (edge.end2.y >= maxpolyy)) ;
```

```
End ;
```

```
Procedure FindIntersectionPoints(TestEdge:LineType ; NumPoints:Integer ;
                                   PolyPoints:VerticesType ; Var p1,p2:R2Vector ;
                                   Var NumInt:Integer ; Var Dependent:Boolean) ;
```

```
Var T : Integer ; PolyEdge : LineType ; intpoint : R2Vector ;
    Inconsistent : Boolean ;
```

```
Begin
```

```
  Dependent := False ;
```

```
  NumInt := 0 ;
```

```
  T := 1 ;
```

```
  Repeat
```

```
    PolyEdge.end1 := PolyPoints[T] ;
```

```
    PolyEdge.end2 := PolyPoints[T+1] ;
```

```

CalcR2Intersection(TestEdge,PolyEdge,intpoint,
                   Inconsistent,Dependent) ;

If (Not Dependent) And (Not Inconsistent) Then
Begin
  If Inbetween(PolyEdge.end1.x,PolyEdge.end2.x,intpoint.x) And
     Inbetween(PolyEdge.end1.y,PolyEdge.end2.y,intpoint.y) And
     Inbetween(TestEdge.end1.x,TestEdge.end2.x,intpoint.x) And
     Inbetween(TestEdge.end1.y,TestEdge.end2.y,intpoint.y)
  Then
  Begin

    Case NumInt Of

      0 : Begin
          NumInt := NumInt + 1 ;
          p1 := intpoint ;
          End ;

      1 : Begin
          If (Not R2VectorsEqual(p1,intpoint)) Then
          Begin
            NumInt := NumInt + 1 ;
            p2 := intpoint ;
            End ;
          End ;
        End ;
      End ;
    End ;
  End ;

  T := T+1 ;

Until (T > NumPoints) Or (Dependent) Or (NumInt = 2) ;

End ;

Function BehindPlane(Point:R3Vector ; e,f,g,h : Real):Boolean ;
Begin
  BehindPlane := (Point.x * e + Point.y * f + Point.z * g + h < 0 ) ;
End ;

Function InfrontPlane(Point:R3Vector ; e,f,g,h : Real):Boolean ;
Var t : Real ;
Begin
  t := Point.x * e + Point.y * f + Point.z * g + h ;
  InfrontPlane := t >= 0 ;
End ;

```

```

Procedure FindPreImage(Theta,Phi,Rho,D:Real ;
    R2Find:R2Vector ;
    R3k1,R3k2:R3Vector ;
    R2k1,R2k2:R2Vector ;
    Var R3Find:R3Vector) ;

```

(Given a point P' on a projected line segment R'S', this procedure finds the point P located on the actual segment RS that projects onto P'. A binary search method is used.)

```

Var R2m,Diff,R2FindDiff,R2mDiff : R2Vector ; R3m : R3Vector ;
    Found : Boolean ;

```

```

Begin

```

```

    Found := False ;

```

```

Repeat

```

```

    R3MidPoint(R3k1,R3k2,R3m) ;

```

```

    ScreenXY(Theta,Phi,Rho,D,R3m,R2m) ;

```

```

    R2Diff(R2m,R2Find,Diff) ;

```

```

    R2Diff(R2Find,R2k1,R2FindDiff) ;

```

```

    R2Diff(R2m,R2k1,R2mDiff) ;

```

```

    If R2Magnitude(Diff) <= 0.2 Then Found := True

```

```

    Else

```

```

        If R2Magnitude(R2FindDiff) < R2Magnitude(R2mDiff) Then R3k2 := R3m

```

```

    Else

```

```

        R3k1 := R3m ;

```

```

Until Found ;

```

```

    R3Find := R3m ;

```

```

End ;

```

```

Procedure ProcessTestEdge(TestEdge:LineType ; EdgeNumber:Integer ;
    R3Endp1,R3Endp2:R3Vector ;
    R2Vertex:R2VertexType ; intp1,intp2:R2Vector ;
    numint,Numpoints:Integer ;
    PolyPoints:VerticesType ;
    e,f,g,h,Theta,Phi,Rho,D:Real ;
    Var Edges:EdgesType ;
    Behind1,Behind2,Insidel,Inside2:Boolean) ;

```

```

Var R3Midpt : R3Vector ; R2midpt : R2Vector ;

```

```

Begin

```

```

If NumInt = 0 Then
Begin
  R2MidPoint(TestEdge.End1,TestEdge.End2,R2midpt) ;
  If InsidePoly(NumPoints,PolyPoints,R2midpt) Then
    EatTestEdge(TestEdge,EdgeNumber,TestEdge.End1,TestEdge.End2,R2Vertex,
      Edges)
End
Else
If NumInt = 1 Then
Begin
  If Insid1 Then
  Begin
    If Behind1 Then
      EatTestEdge(TestEdge,EdgeNumber,intp1,TestEdge.end1,R2Vertex,Edges) ;
    End
  Else
  If Inside2 Then
  Begin
    If Behind2 Then
      EatTestEdge(TestEdge,EdgeNumber,intp1,TestEdge.end2,R2Vertex,Edges) ;
    End ;
  End
End
Else
Begin
  If (Behind1) And (Behind2) Then
    EatTestEdge(TestEdge,EdgeNumber,intp1,intp2,R2Vertex,Edges)
  Else
  Begin
    R2MidPoint(intp1,intp2,R2midpt) ;
    FindPreImage(Theta,Phi,Rho,D,R2midpt,R3endp1,R3endp2,
      TestEdge.end1,TestEdge.end2,R3midpt) ;

    If BehindPlane(R3midpt,e,f,g,h) Then
      EatTestEdge(TestEdge,EdgeNumber,intp1,intp2,R2Vertex,Edges) ;
    End ;
  End ;
End ;

```

```

Procedure CalcR3EyeVertex(R3_SINRD_VERTEX:R3VertexType ;
  VertexCount:Integer ;
  Var R3_EYE_VERTEX:R3VertexType ;
  Theta,Phi,Rho,D:Real) ;

```

```

Var K : Integer ;

```

```

Begin
  For K := 1 To VertexCount Do
    EyeXYZ(Theta,Phi,Rho,D,R3_SINRD_VERTEX[K],R3_EYE_VERTEX[K]) ;
  End ;

```