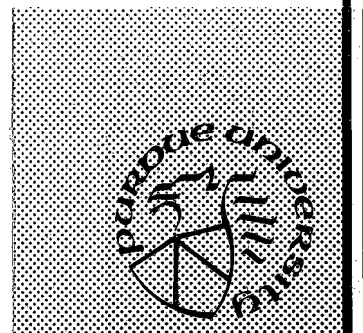
7-1-1984

Automatic Construction of CSG Representation from Orthographic Projections

Namdar Saleh
Purdue University

K. S. Fu
Purdue University



Automatic Construction of CSG Representation from Orthographic Projections

Namdar Saleh
K. S. Fu

TR-EE 84-24
July 1984

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

AUTOMATIC CONSTRUCTION OF CSG REPRESENTATION FROM ORTHOGRAPHIC PROJECTIONS

Namdar Saleh and K. S. Fu

TR-EE 84-24

July 1984

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

This work was supported by the NSF Grant ECS 81-19886. The work was also supported in part by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation and TRW.

BIBLIOGRAPHIC DATA SHEET	1. Report No. TR-EE 84-24	2.	3. Recipient's Accession No.

4. Title and Subtitle

AUTOMATIC CONSTRUCTION OF CSG REPRESENTATION FROM ORTHOGRAPHIC PROJECTIONS

5. Report Date: July 1984

6.

7. Author(s): Namdar Saleh and K. S. Fu

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Purdue University
School of Electrical Engineering
West Lafayette, Indiana 47907

10. Project/Task/Work Unit No.

11. Contract/Grant No.: ECS 81-19886

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D.C. 20550

13. Type of Report & Period Covered: Technical

14.

15. Supplementary Notes

16. Abstracts

An algorithm has been designed to construct the CSG model of an object from its 2D orthographic projections. The method proposed uses a top-down approach in which the existence of some 3D primitive (e.g. CUBE) is assumed and then different views are searched for appropriate elements to prove the assumption. The algorithm is applied to some examples and the results are demonstrated. A second algorithm has also been designed to implement the automatic input of line drawings. The drawings are first digitized using a high resolution scanner. After some preprocessing, the algorithm is applied to the image in order to extract the relevant graphical elements, such as arcs and circles. Two examples are also demonstrated.

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 165
	20. Security Class (This Page) UNCLASSIFIED	22. Price

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT ix

CHAPTER ONE - ENGINEERING DRAWING AND GEOMETRIC MODELING FUNDAMENTALS 1

1.1 Introduction 1
1.2 Fundamentals of Engineering Drawings 2
1.3 Solid Geometry Representation 7
1.3.1 A list of related fields 7
1.3.2 Properties of representation schemes 7
1.3.3 Constructive Solid Geometry 9

CHAPTER TWO - AUTOMATIC RECONSTRUCTION OF AN OBJECT FROM ITS 2D ORTHOGRAPHIC PROJECTIONS 13

2.1 Introduction 13
2.2 Previous work done 13
2.2.1 Polyhedra as the class of objects 13
2.2.2 Objects with uniform thickness 15
2.3 A new approach for solving the reconstruction problem 16
2.4 The algorithm 16
2.4.1 Explanation 16
2.4.2 Cylinder 18
2.4.3 Cone 19
2.4.4 Cube 21
2.4.5 Lugs 25
2.4.6 Corners 30
2.5 A comparison between this work

Page

and previous works 33
2.6 Examples 36
2.6.1 Data input 36
2.6.2 Output conventions 37
2.6.3 Sample Executions 38
2.7 Performance analysis 57

CHAPTER THREE - MANUAL AND AUTOMATIC INPUT OF LINE DRAWINGS 60

3.1 Introduction 60
3.2 Manual input sytem 61
3.3 Automatic input system 62
3.3.1 Image digitization and preprocessing 62
3.3.2 Digital straight lines 64
3.3.3 Digital arcs 65
3.4 The algorithm 67
3.5 Examples 70
3.6 Performance analysis 94

CHAPTER FOUR - CONCLUSION AND FUTURE RESEARCH 97

4.1 Conclusion 97
4.2 Future research 98

REFERENCES 99

APPENDIX 101

LIST OF TABLES

Table Page

2.1 Time analysis for Examples 1, 2 & 3 58

3.1 Time analysis for Examples 4 & 5 95

LIST OF FIGURES

Figure Page

1.1 Arrangement for the six principal views 3

1.2 The meaning of a line 5

1.3 Projections of a surface 6

1.4 The meaning of different operators 10

1.5 The semantics of CSG-tree representation 11

2.1 General block diagram of algorithm 1 17

2.2 Example of an object with a conical part 22

2.3 Simple cubical object 24

2.4 The major orientations of lugs 26

2.5 Combination of cubes to make a frustum 28

2.6 CSG construction of cube with round corner 31

2.7 Example of a lug and round corners 34

2.8 Example 1 40

2.9 Example 2 47

2.10 Example 3 52

3.1 Automatic input system 63

3.2 Imperfect digitization of a straight line 66

Figure Page

3.3 Example 4 72

3.4 Digital image of view 1 of Example 4 73

3.5 Digital image of view 2 of Example 4 74

3.6 Digital image of view 3 of Example 4 75

3.7 View 1 of Example 4 after preprocessing 76

3.8 View 2 of Example 4 after preprocessing 77

3.9 View 3 of Example 4 after preprocessing 78

3.10 Example 5 83

3.11 Digital image of view 1 of Example 5 84

3.12 Digital image of view 2 of Example 5 85

3.13 Digital image of view 3 of Example 5 86

3.14 View 1 of Example 5 after preprocessing 87

3.15 View 2 of Example 5 after preprocessing 88

3.16 View 3 of Example 5 after preprocessing 89

ABSTRACT

An algorithm has been designed to construct the CSG model of an object from its 2D orthographic projections. The method proposed uses a top-down approach in which the existence of some 3D primitive (e.g. CUBE) is assumed and then different views are searched for appropriate elements to prove the assumption. The algorithm is applied to some examples and the results are demonstrated. A second algorithm has also been designed to implement the automatic input of line drawings. The drawings are first digitized using a high resolution scanner. After some preprocessing, the algorithm is applied to the image in order to extract the relevant graphical elements, such as arcs and circles. Two examples are also demonstrated.

CHAPTER ONE

ENGINEERING DRAWING AND GEOMETRIC MODELING FUNDAMENTALS

1.1 INTRODUCTION

The process of automatic manufacturing of mechanical parts is currently of great importance in industry and is a field that challenges researchers in many areas of engineering and computer science. This process has several levels, from input to a solid modeler to production of an NC or CNC (computerized numerical control) part program. One would like to be able to make an engineering drawing showing projections on an electronic board, using the methods developed by generations of engineers, and then have the part program generated [PR2]. Obviously once the engineering drawing of a certain design is created, the information present in the drawing has to be conveyed to the NC machine, possibly in the form of an explicit 3D data structure. This process, which is nothing but the conversion of one form of representation of information to another, can be broken up into two parts:

1 Creation of a 2D data structure of graphical elements such as straight lines and arcs from the line drawing. After this step, the information present in the engineering drawing is stored in a computer where it can be accessed and manipulated by appropriate algorithms.

2 Interpretation and reconstruction of the 3D representation of the part from the 2D data structure. After this step, the object is represented in some form of solid model, and can be stored in an appropriate data base.

For the first part, two systems, manual and automatic are proposed and these are discussed in Chapter three. The second part is treated in Chapter two in which an approach for the reconstruction process is suggested and compared to previous work on the subject. An introduction to the rules governing engineering drawings is given in the next section. This is followed by a discussion on representations for rigid solids and an explanation of the method used in this work which is the Constructive Solid Geometry.

1.2 FUNDAMENTALS OF ENGINEERING DRAWINGS

Engineering drawing is a graphic language that is used universally by design engineers and engineering technologist to describe the shape and size of structures and mechanisms. It has developed through the centuries, much as have various spoken and written languages, until at the present time its fundamental principles are understood by trained persons [LUZ]. Three dimensional objects are represented in engineering drawings by two to six two_dimensional orthogonal views. Figure 1.1 contains the American standard arrangement for the six principal views [WEL]. When preparing an engineering drawing, a shape description

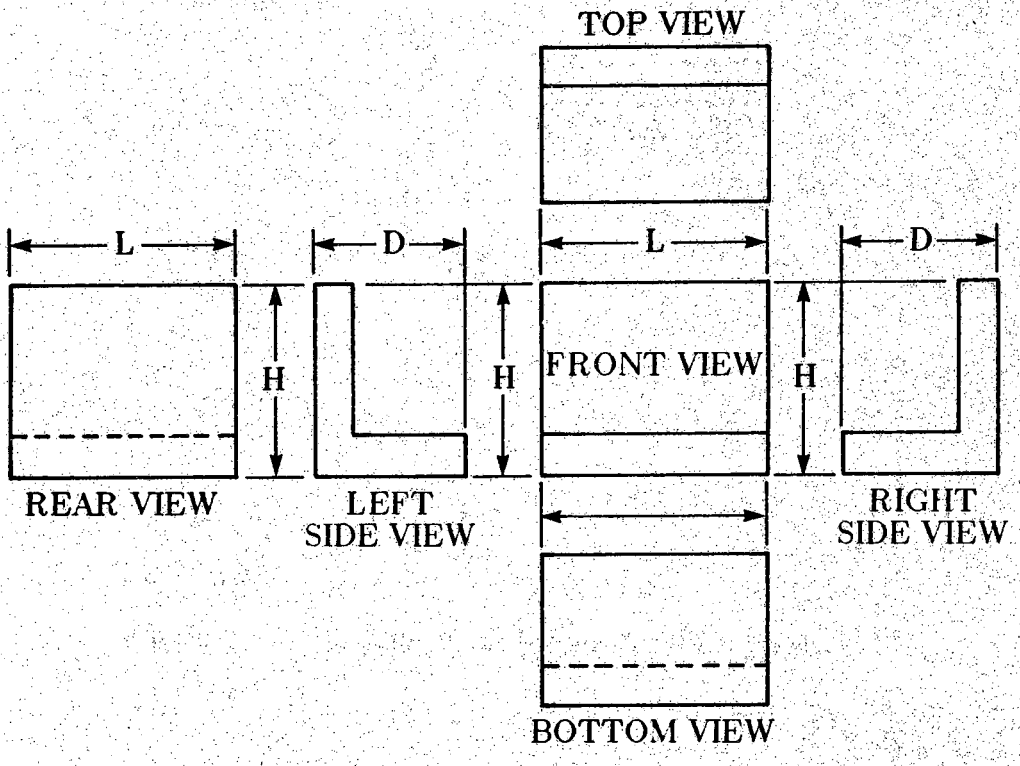


Figure 1.1 Arrangement for the six principal views

method called orthographic or parallel projection is used. The orthographic geometry governing this method has the following properties [HAR]:

Rule 1 - The lines of sight for any two adjacent views are perpendicular.

Rule 2 - Every point of the object in one view is aligned on a parallel directly opposite the corresponding point in any adjacent view.

Rule 3 - The distance between any two points on the object measured along the parallels is the same in all related views.

Rule 4 - A line can only appear as a line or a point, a point being the end view of a line.

As shown in figure 1.2 [LUZ], a visible or invisible (dashed) line may represent either the intersection of two surfaces, the edge view of a surface, or it may be the limiting element of a surface. The full circle in the front view may be considered as the edge view of the cylindrical surface of the hole. In the side view, the top line, representing the contour element of the cylindrical surface, indicates the limits for the surface and therefore can be thought of as being a surface limit line.

Rule 5 - Every face can appear only as an edge or as a figure of similar configuration. More precisely, when a surface is parallel to a plane of projection, it will appear in true size in the view on the plane of projection to which it is parallel. When it is perpendicular to the plane of projection, it will project as a line in the view. And finally when it is positioned at an angle, it will appear foreshortened. See figure 1.3.

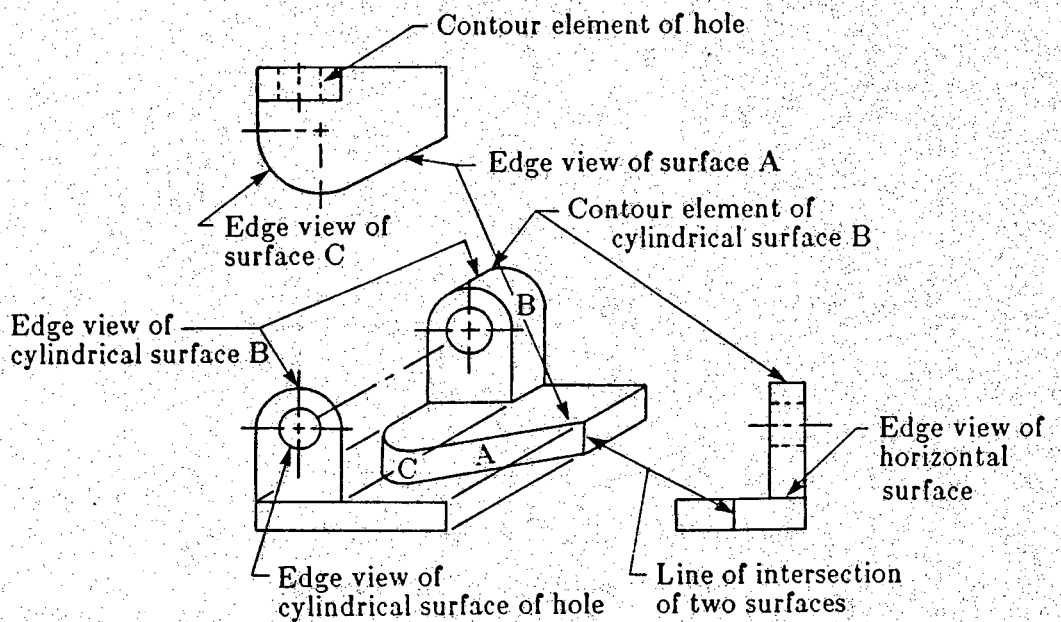
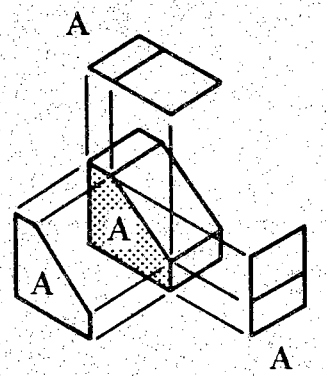
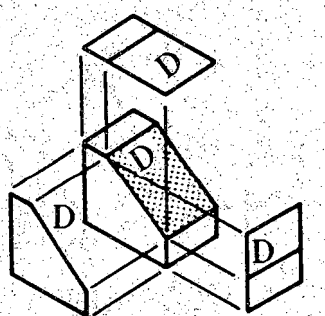


Figure 1.2 The meaning of a line



SURFACE A



SURFACE D

Figure 1.3 Projections of a surface

From the orthographic geometry it is apparent that each view contributes information not in the other views and that to understand the object portrayed by the orthographic view, the information in one view must be used in a coordinated way with the other views.

1.3 SOLID GEOMETRY REPRESENTATION

1.3.1 A LIST OF RELATED FIELDS

The problem of representing mechanical components requires talents from such fields as data structures, logic and algorithms, artificial intelligence, programming languages, numerical control, metal cutting, and operations research [WOO]. Before discussing the advantages and disadvantages of CSG, a general discussion on representation schemes and their properties is in order.

1.3.2 PROPERITIES OF REPRESENTATION SCHEMES

A representation scheme is a relation between (abstract) solids and representations. There are several methods for constructing complete representations of solids and some of them are: Constructive Solid Geometry; Sweeping and Boundary Representation. In general, representation schemes have four formal properties and they are as follows [REQ].

1. Domain: The domain of a representation scheme Characterizes the descriptive power of the scheme.

2. Validity: The range of a representation scheme is the set of representations which are valid. Validity is an important property because it ensures the integrity of databases, in that databases should not contain symbol structures which correspond to nonsense objects.

3. Completeness: A representation is complete if it corresponds to a single object, that is, there are no ambiguities. This is the most important formal characteristic of representation schemes. It is crucial when there is a wide range of applications to be supported by a practical modeling system, and especially when the range of applications is not known [VOE].

4. Uniqueness: The representation o an object is unique if it is the only possible representation of that object in that particular scheme. Representational uniqueness is important for assessing the equality of objects in automatic planning algorithms and numerically controlled (NC) machine tools. Representations which are both complete and unique are highly desirable. However, most representation schemes, are nonunique for at least two reasons.

- Substructures in a representation may be permuted.
- Distinct representations may correspond to differently positional but congruent copies of a single geometric entity.

An example of representation schemes that are complete but not unique are CSG. In the next section we study these schemes in more detail.

1.3.3 CONSTRUCTIVE SOLID GEOMETRY

Constructive Solid Geometry connotes a family of schemes for representing rigid solids as Boolean constructions or combinations of solid components via the regularized set operators, mainly Union (+), Intersection (&) and Difference (-) [REQ]. These operators are demonstrated in figure 1.4 . CSG representations are ordered binary trees. Nonterminal nodes represent operators, which may be either rigid motions or regularized union, intersection or difference, terminal nodes are either primitive leaves which represent subsets of E^3 [1] or transformation leaves which contain the defining arguments of rigid motions.

The semantics of CSG-tree representation is clear (figure 1.5): Each subtree that is not a transformation leaf represents a set resulting from applying the indicated motional/combinational operators to the sets represented by the primitive leaves. Schemes whose primitives are bounded are called "CSG based on bounded primitives", or simply CSG when no confusion is likely to arise, while schemes possessing unbounded primitives are called "CSG based on general half spaces". We consider only CSG schemes whose primitives are bounded. In fact, the main advantage in using primitive volumes in the description process is that the object constructed is always bounded and finite, since the primitives are [WOO].

When the primitive solids of a CSG scheme are bounded and hence are r-sets [2], the algebraic properties of r-sets guarantee that any CSG tree is a valid representation of an r-set if the primitive leaves are

---

1 Three dimensional Euclidean space.

2 r-sets are subsets of E^3 that are bounded, closed, regular and semianalytic.

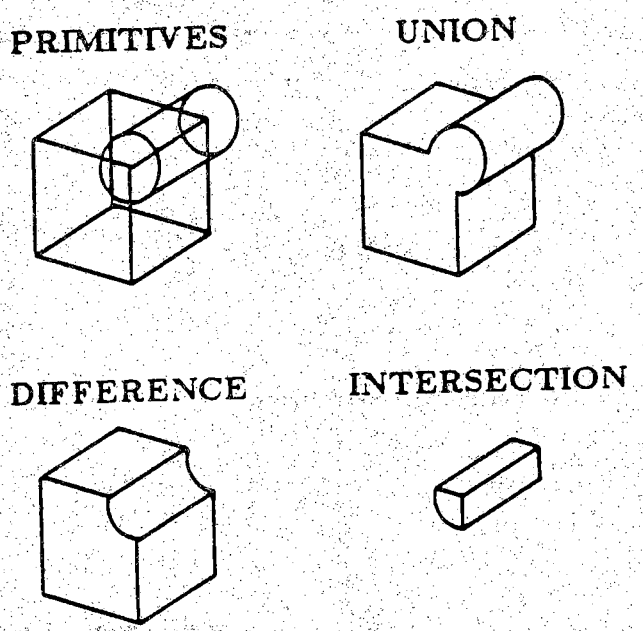


Figure 1.4 The meaning of different operators

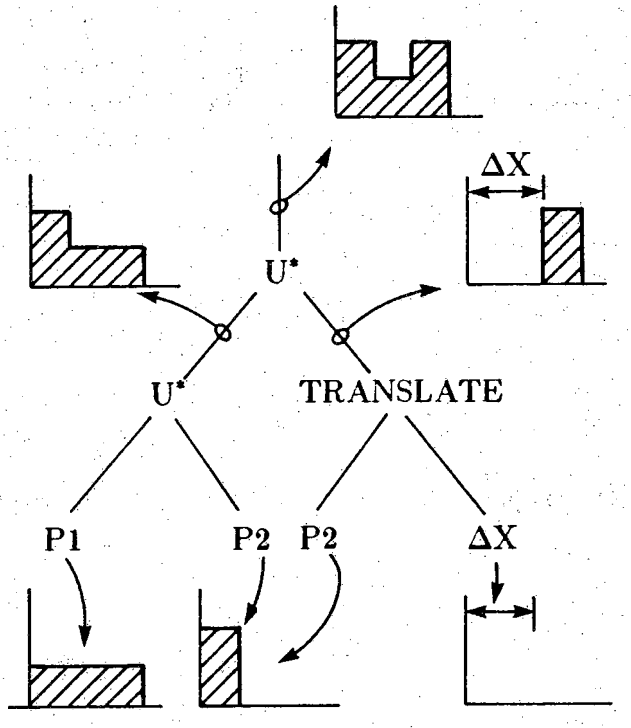


Figure 1.5 The semantics of CSG-tree representation

valid. This guaranteed validity of CSG schemes based on solid primitives applies only to schemes in which the combinational operators are general regularized set operators which may be applied to any objects in the domain of the representation schemes.

Overall the benefits of CSG are [ROT] :

- The model represents a true solid with volume.
- Curved as well as planer surfaces bound the solids.
- The combined operators are remarkably effective for modeling solid artifacts, particularly mechanical parts.

In addition, experience has showed that humans can easily create CSG representations of certain classes of objects such as mechanical parts [REQ].

CHAPTER TWO
AUTOMATIC RECONSTRUCTION OF AN OBJECT FROM ITS 2D ORTHOGRAPHIC PROJECTIONS

2.1 INTRODUCTION

The purpose of this chapter is to discuss the problem of obtaining the 3D representation of an object from its 2D projections. Comparatively little work has been done on line drawing interpretation in the context of geometry definition. This work has mainly considered the class of polyhedra ([LAF],[LIA],[HAR],[PR1]). Results have also been obtained on curved objects with uniform thickness ([ALD1],[ALD2]), and on objects with less restrictions [SAK]. In the following section, some of the previous algorithms are briefly described. Sections 2.3 - 2.4 present a different approach to solving the reconstruction problem.

2.2 PREVIOUS WORK DONE

2.2.1 POLYHEDRA AS THE CLASS OF OBJECTS

In the paper by PREISS [PR1],the emphasis for the interpretation of 2D drawings is on the connectedness properties. The approach used is similar to the approach in the theorem proving programs that have

the following general principles:

(a)- Existence of data representing the current state.

(b)- Rules by which all possible future states can be evaluated.

(c)- A definition of legal final states.

The algorithm goes as follows:

- Find the possible coordinates of each vertex. Two of the three coordinates are available in each view. Using the other views, set up a list of possible third coordinates.

- Identify the projected faces given by closed paths of solid lines in each view.

- Interpret the projected faces by identifying its vertices from an ordered depth first search.

- Interpret the dashed lines.

- Assemble the body using a technique from scene analysis programs.

The algorithm is not very hard to follow. However, the part about the interpretation of dashed lines is ambiguous. There is a possibility of modifying the process in order to be able to treat curved surfaces.

In the paper by Haralick and Queeney [HAR],the problem is treated as three consistent labeling problems. A set of rules are defined according to the properties of polyhedra. Some of the rules are as follows:

(a)- Every point of the object in one view is aligned on a parallel directly opposite the corresponding point in any adjacent view.

(b)- A line can only appear as a line or a point, a point being an end view of a line.

(c)- Every face can appear only as an edge or as a figure of similar configuration.

(d)- No two contiguous faces can lie in the same plane.

The algorithm is similar to the previous one:

- Find the set of V(x,y,z) eligible to be vertices.
- Find the set of visible surfaces for each view.
- Find the interpretation of the surfaces denoted by three or more vertices according to some rules.
- Make sure the interpretations are consistent.

Steps a,c,d are consistent labeling problems and are solved using a tree search. The main drawback of the algorithm is that there is no mention of any treatment of hidden lines. Therefore the object is always viewed from an angle where there are no hidden lines, an assumption that is not very practical.

2.2.2 OBJECTS WITH UNIFORM THICKNESS

The main distinction of the algorithm by B.Aldefeld [ALD1],[ALD2], from the previous methods is that it is able to interpret curved objects as well as plane faced polyhedra. The interpretation process has two parts. In the local part, objects are recognized by their individual patterns, irrespective of any possible global inconsistencies. Therefore several sets of candidates including spurious ones are generated. The second part which is the global interpretation step, takes care of finding the subset of real objects among the candidates and of recognizing whether each elementary object is a solid or a cavity. The algorithm is rather complicated and includes heuristic searching and matching.

Also the data structure used for representing the final 3D object is not specified, although it is said that the representation is volume oriented.

2.3 A NEW APPROACH FOR SOLVING THE RECONSTRUCTION PROBLEM

In this work, a top_down interpretation approach has been used. This means that the existence of a certain 3D primitive (cube, cylinder, cone) is assumed and then the views are searched in order to find the necessary 2D primitives that justify the assumption. If the assumption is justified then the 2D primitives are used in order to obtain the attributes needed for the 3D representation. This assumption has also been extended to some combinations of 3D primitives, namely corners and lugs.

The following section explains the reconstruction algorithm in more detail. Each subsection is devoted to the interpretation of one of the elements mentioned above.

2.4 THE ALGORITHM

2.4.1 EXPLANATION

A block diagram of the algorithm is shown in figure 2.1. In the following subsections, each step of the interpretation process is discussed and then briefly illustrated in algorithmic form. For more detail on the algorithm, the reader should refer to the programs included in the appendix.

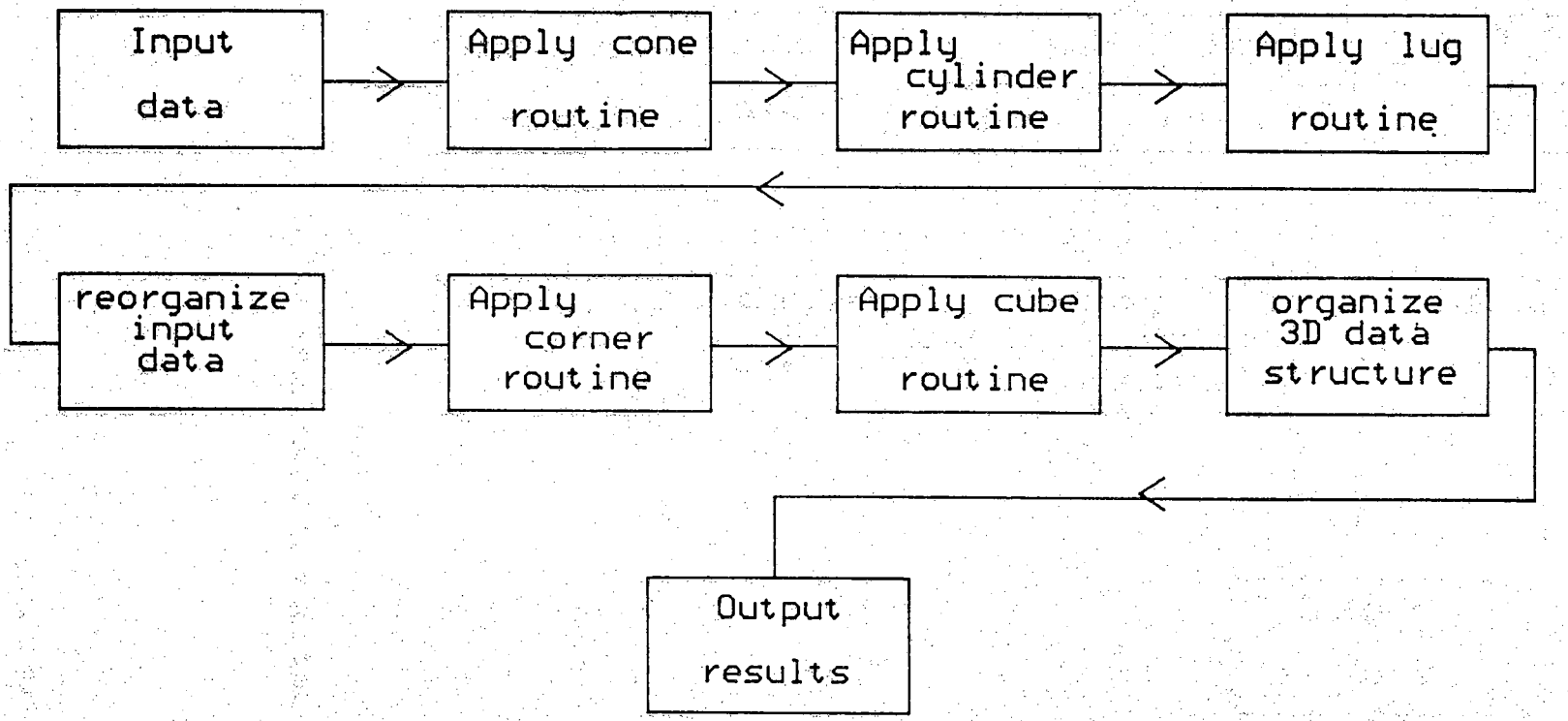


Figure 2.1 General block diagram of algorithm 1

2.4.2 CYLINDER

The planes of projection in an engineering drawing are usually selected so that in case of the existence of a cylinder, the axis would be perpendicular to one of the planes. Therefore the projection of the cylinder in that view is a circle. The projection in the other views is two parallel lines. These lines might or might not match [1] and they may be solid or dotted depending on the mode of the cylinder (solid or cavity) and the objects surrounding it.

The above facts make the cylinder the easiest primitive to detect. The views are searched for circles and if one or more are found, the other views are searched for the above mentioned lines. In order to represent a cylinder uniquely the following are needed:

- The radius which is just the radius of the circle detected in one of the views.
- The length which is given by the length of the lines in the corresponding views.
- The orientation of the axis of the cylinder which is also available from the view of the circle.

The routine goes as follows:

FOR view = 1,3 *DO*
  n = number of circles in this view;
  FOR i = 1,n *DO*
    find the horizontal and vertical extremities on $circle_i$;
    use the coordinates of the extreme points to find corresponding

1 Two lines are said to match when they are equal in length and direction.

```
        lines in the other views;
        IF in any view no such line is found THEN
          GO TO END;
        find the best candidate among the lines found;
        output the cylinder;
      END;
    END;
```

It should be noted that when the parallel lines can not be found, then we definitely do not have a cylinder. In this case the circle found earlier corresponds to some conical object. This is discussed in the next subsection.

2.4.3 CONE

In mechanical objects, there are cases where we encounter parts that have a conical shape. A whole cone however is very seldom encountered, therefore we do not need bother with teaching our system to recognize it. More often we have a part that looks like a cone whose top has been cutoff. This part has the following representation in a three view engineering drawing:

- In one of the views we have two concentric circles. The larger one is the projection of the base of the cone, and the smaller one is where the original cone has been cut.

- In the other two views we have an identical four sided figure which has the following properties : of the opposing sides, two of them are parallel but with different lengths. The other two are equal in length. It should also be mentioned that the class of objects considered requires that the parallel sides be either horizontal or vertical. As a convention, the larger of these parallel lines will be called 'base' and the slanted sides will be called 'arms'.

In terms of C.S.G., the conical object described above can be represented as a combination of a cone and cylinder, i.e.

OBJECT = CONE - CYLINDER

Therefore the following information has to be extracted from the drawings:

- The coordinates of the center and the radius of the circle representing the base of the cone.
- The height and orientation of the original cone.
- The coordinates of the center and the radius of the circle representing the cylinder.
- The height and orientation of the cylinder.

Once two concentric circles have been found in a view, we search an adjacent view for the base line. There may be more than one candidate but the right one has to be connected to two slanted and equal lines , i.e. the arms. Once the arms are found, we have enough evidence that the object is conical and we also have all the information needed for representing it. For example, the height of the cone, h, can be calculated if we have the length of the base line and the angle teta that the arm makes with the base:

$$2h = \text{baselength} \times \tan(\text{teta})$$

The routine goes as follows:

FOR view = 1,3 *DO*
n = number of circles in this view; *FOR* i = 1,n-1 *DO*

```
        FOR j = i+1,n DO
          IF center_of_circle_i = center_of_circle_j THEN
             find the horizontal and vertical extremities on the larger
             circle;
             use the coordinates of the extreme points obtained above
             to find the base line;
             use the base to find the arms;
             check to make sure the circles are the projection of a cone;
             output the cone;
          END;
        END;
      END;
    END;
```

An example of a simple object that contains a conical part and its CSG representation as a result of using the above algorithm is illustrated in figure 2.2

2.4.4 CUBE

The process of recognizing a cube is more complex than previous primitives because of a high degree of freedom in its 2D representation. A complete and isolated cube has 4 perpendicular lines in the form of a rectangle or a square as projection on each view plane. However, when other objects are combined with the cube, many of these lines are either totally missing or only partly visible. The cube algorithm expects to find a horizontal line connected to two vertical lines that match in the first view. If these elements are found, then the rules of engineering drawings require that two parallel lines corresponding to the cube be present in view 2 and view 3 each. One problem that arises here is that more than two lines may be found in those views (two or more lines may be concatenated in the same direction) and it is not always obvious which line is the projection of the cube. One way to solve this problem is

(cone1 - cylinder1) + cylinder2

+ cylinder3 - cylinder4

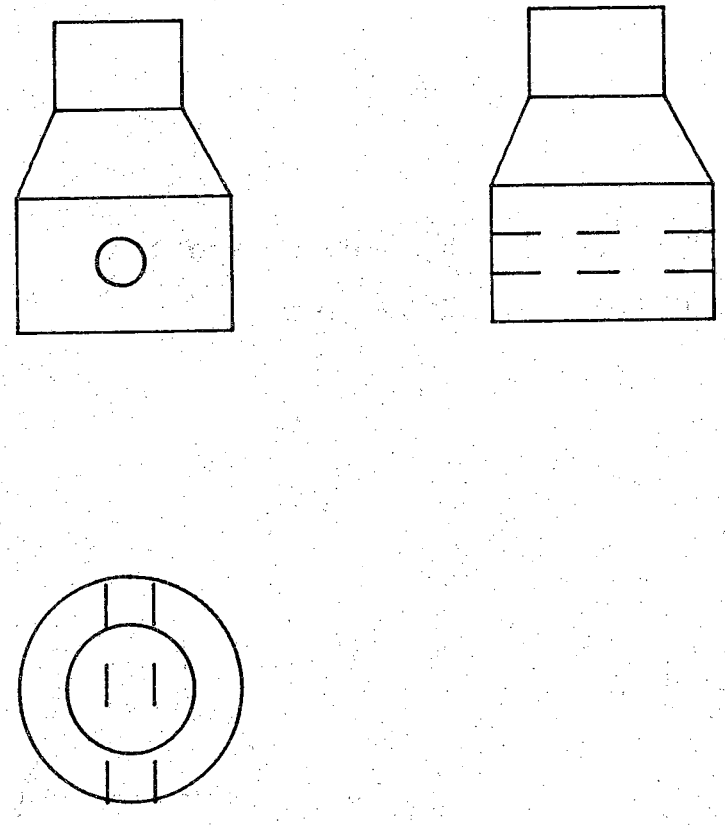


Figure 2.2 Example of an object with a conical part

to give priorities to certain configurations of lines. For example consider the simple object of figure 2.3. Lines a, b, and c have been found in view 1 and a search for corresponding lines in views 2 and 3 has resulted in lines b1, b2, c1, and c2 in view 3 and a1, a2 in view 2. All of these lines are candidates for the third dimension of the cube. Priority is given to view 2 because it contains only two lines. However the lines a1 and a2 do not match, so there is still some uncertainty. In this case line a1 is chosen because it is at an extreme location in view 2. That is it has the lowest y coordinate among the lines in view 2.

The algorithm is not limited to complete cubes only. Cubical frustums can also be detected. In this case we have two slanted lines connected to a horizontal line. The process of finding the third dimension of the frustum is the same as explained above. However, the representation of a frustum in terms of CSG is more complex than the representation of a simple cube. This problem is more thoroughly discussed in the next subsection.

The routine goes as follows:

REPEAT
  find a horizontal line;
  find two lines that are connected to the ends of the above line;
  IF the lines match *OR* the lines are slanted and equal in length *THEN*
    look for lines in views 2 and 3 that are candidates for the projection of the cube in those views, using the coordinates of the lines above;
    choose the best candidates;
    IF the lines match *THEN*
      output a cube;

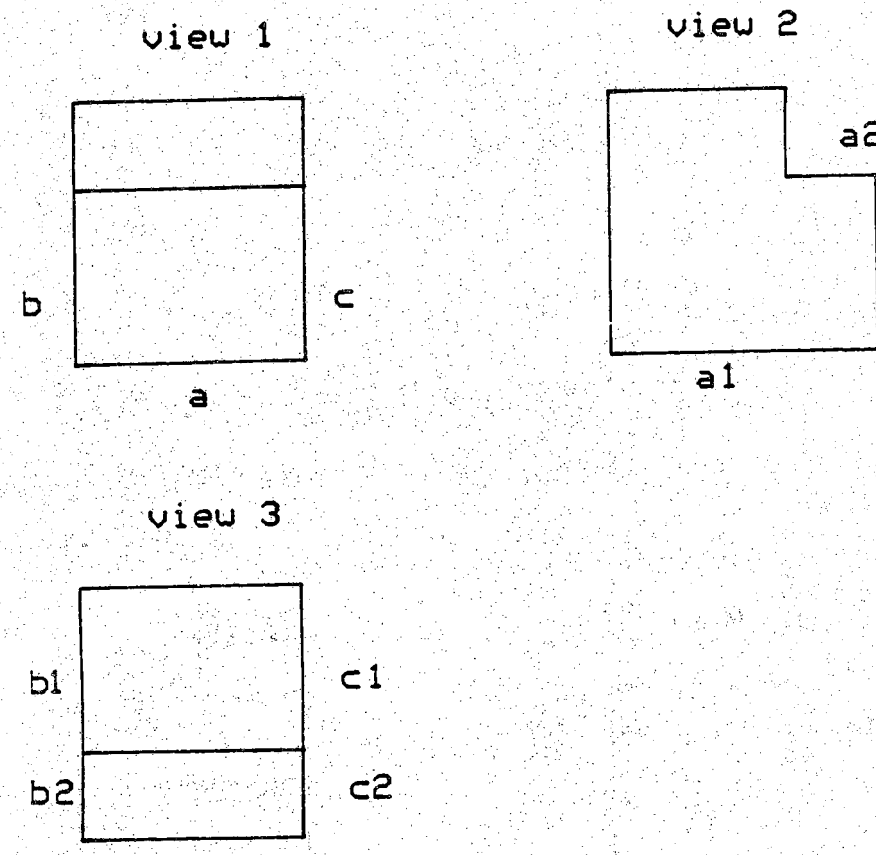


Figure 2.3 Simple cubical object

ELSE
using the angles and the length of the slanted lines, find the dimensions and position of the frustum;
output a frustum;
END;
END;
UNTIL no more horizontal lines;

2.4.5 LUGS

Objects classified as lugs are those objects that have a cylindrical part in union with a cubical part. These objects are very common in mechanical parts. Their front view representation in 2D drawings is an arc connected to two line segments at its ends. The lines could be either parallel or not and the whole object can have infinite possible rotations. However 4 major configurations are very common in engineering drawings and they are shown in figure 2.4.

Since all four configurations can occur in any of the three views, we have a total of 12 possible cases to consider. In each case the following information has to be extracted in order to uniquely represent the lug.

_The centers of the two circles of the cylinder.

_The radius of the cylinder.

_The thickness of the cylinder.

_The origin of the cube.

_The x,y,z dimensions of the cube.

In order to render the representation of the partial cylinder independent of the cube that is attached to it, it is a better idea to have the output as (CYL1 - CUBE1) + CUBE2 instead of CYL1 + CUBE2, where CUBE1 is a cube that intersects the cylinder in a manner to have the desired half cylinder as a result, and CUBE2 is the cube that

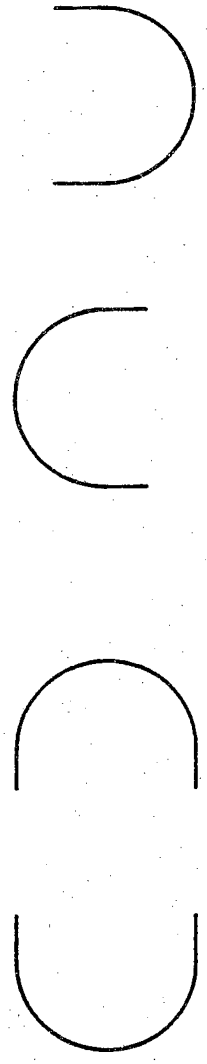


Figure 2.4 The major orientations of lugs

completes the representation of the lug.

Most of the information mentioned above is available in the view in which the curve appears. However the thickness of the cylinder (and the cube) has to be found from another view. One way to go about finding this thickness is to use the fact that the midpoint on the arc should map into a line in another view and that this line will be unique because of the class of objects considered. Therefore once the arc has been located, the coordinates of the midpoint on its body can be calculated and depending on the view in which the arc resides, and the orientation of the arc, we can determine which view should be searched for the line segment in question. After this step, the cylinder can be defined uniquely.

The problem of outputting the cubical part of the lug can be more complicated especially if the line segments connected to the endpoints of the arc are not parallel. In this case the cube in question will be a combination of three cubes. The relation between the three cubes is demonstrated in figure 2.5. As it can be seen from the figure, the information that needs to be extracted is the angle teta and the coordinates of the origin of cube C. The x,y,z dimensions of the cube have to be obtained with regard to the view we are in. The x,y,z dimensions of cubes A and B are not important as long as the cubes cover the volume that is to be extracted from cube C. Cube A and B are defined with respect to cube C using homogeneous transformation conventions. In order to ease the output process all rotations and translations involved in defining cubes A and B are done with respect to the origin of cube C and then the result of the combination of A,B,C is moved to its

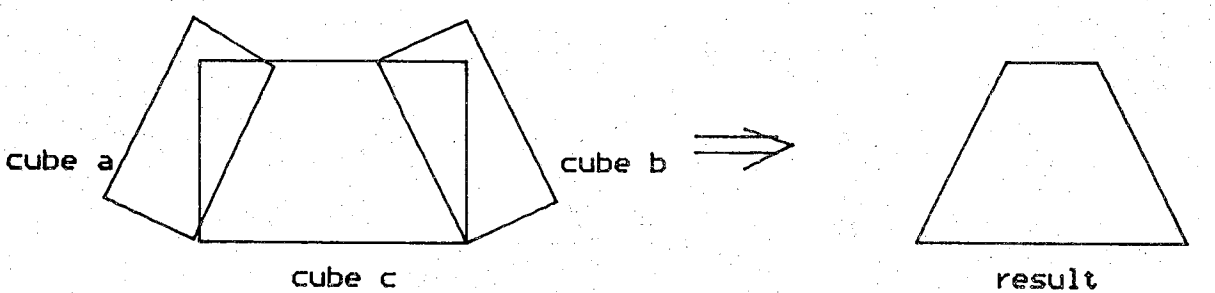


Figure 2.5 Combination of cubes to make a frustum

appropriate location. That is we define local coordinates with origin (0,0,0) at the origin of cube C and after subtracting A and B from C we move the result to the global coordinates of the origin of C.

The routine goes as follows:

```
FOR view = 1,3 DO
  n = number of arcs in this view;
  FOR i = 1,n DO
    find the midpoint of arc_i;
    using the coordinates of the midpoint, find a line that is the pro-
    jection of the half cylinder in a secondary view;
    using the line just fund and arc_i, output a cylinder;
    using the endpoints of arc_i, find the dimensions and position of
    the cube to be subtracted from the cylinder;
    output a cube;
    IF the endpoints of arc_i are not connected THEN
      find two lines that are connected to the endpoints;
      IF the lines match THEN
        output a cube;
      ELSE IF the lines are slanted and equal in length THEN
        output a frustum;
      END;
    END;
  END;
END;
```

It should be noted that after each iteration of the lug algorithm, the input data is reorganized as a preprocessing for the cube algorithm.

2.4.6 CORNERS.

Mechanical parts in many cases have round instead of sharp corners and this simple difference makes the interpretation and representation of them more complex. As an example let us consider the case where a cubical object has three sharp corners and one round corner as shown in figure 2.6. In terms of CSG schemes, the above object can be represented as follows:

((CUBE A - CUBE B) + CYLINDER C)

where the location of CUBE B and CYLINDER C is at the round corner of CUBE A and their thickness is the same as that of CUBE A. It is easy to see that without the rounding effect, the representation of the object would have simply been CUBE A.

For every round corner, the radius and center of the arc give us the radius, one of the centers of the cylinder and the x and y dimensions of the cube. The origin of the cube ,however, depends on the position of the arc. For example, suppose the horizontal line connected to the arc is LINE1 and the vertical line connected to the arc is LINE2. Then we have the following for the x and y coordinates of the origin of the cube , cube_orig:

Case A:

cube_orig(x) = LINE2(POINT2(x))[1]

cube_orig(y) = LINE2(POINT2(y))

---

1 LINEm(POINTn(x)) means the x coordinate of the n endpoint of LINE m.

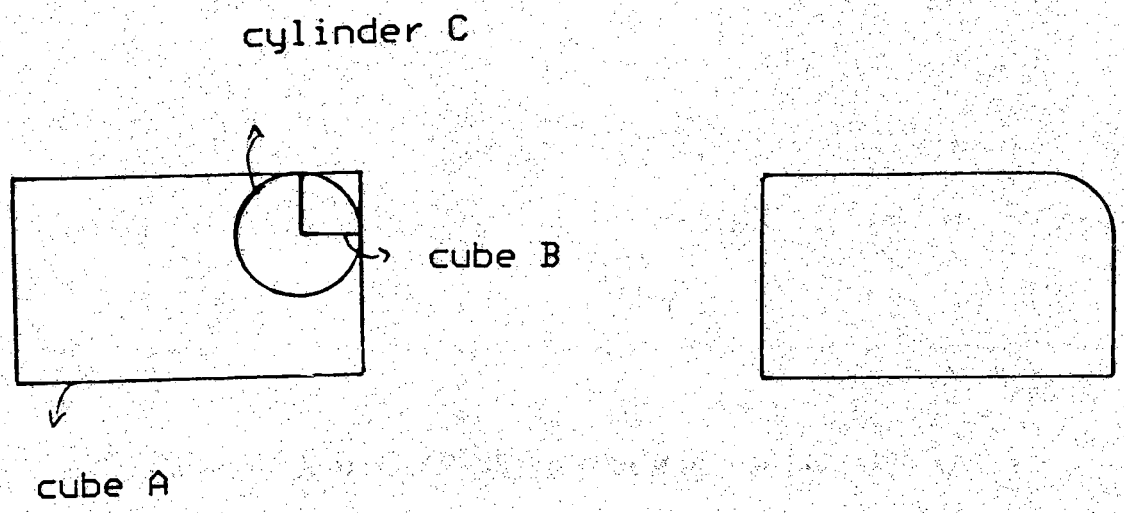


Figure 2.6 CSG construction of cube with round corner

Case B:

cube_orig(x) = LINE1(POINT2(x))

cube_orig(y) = LINE2(POINT2(y))

Case C:

cube_orig(x) = LINE2(POINT1(x))

cube_orig(y) = LINE1(POINT1(y))

Case D:

cube_orig(x) = LINE1(POINT2(x))

cube_orig(y) = LINE1(POINT2(y))

The last information needed for the representation of CUBE B and CYLINDER C is their third dimension, that is the z dimension of the cube which is the same as the thickness of the cylinder. One way to go about finding this information, call it zl, is to search the other two views. However, the same conditions and ambiguities that existed in the cube interpretation process exist here. That is, there may be more than one candidate for zl and a set of criterions has to be designed. In addition, the original cube will eventually go through the process of interpretation and its z dimension which is the same quantity that we are looking for will be available. So instead of trying to find zl at this point, a better and faster solution is to mark the cube and cylinder representations as incomplete and then complete them later when zl becomes available.

Finally, because of the requirements that the cube interpretation algorithm has, the round corners should all be replaced by sharp corners. Therefore, once the round corners have been processed, LINE1 and LINE2 should be extended to meet at a 90 degrees angle. After this step, the corner algorithm is done.

The routine goes as follows:

FOR view = 1,3 *DO*
  n = number of arcs in this view;
  FOR i = 1,n *DO*
    IF arc_i belongs to a round corner *THEN*
      determine which of the four possible cases has occurred;
      using the center, radius and endpoints of arc_i, find the dimensions and position of the cube and cylinder;
      mark the cube and cylinder just obtained as incomplete and store them so that they can be accessed later when the appropriate information is available;
      transform the round corner into a sharp corner;
    END;
  END;
END;

An example that demonstrates a lug and corners is shown in figure 2.7.

2.5 A COMPARISON BETWEEN THIS WORK AND PREVIOUS WORKS

In general there are three main differences between this 2D_3D reconstruction algorithm and the ones suggested by other researchers:

CLASS OF OBJECTS

Many authors have designed algorithms that deal with polyhedra only. This condition seriously constrains the scope and usefulness of their work since in the real world most mechanical parts contain some cylindrical or conical part. Other authors, however, have come up with ways to interpret curved faces too. The class of objects considered in

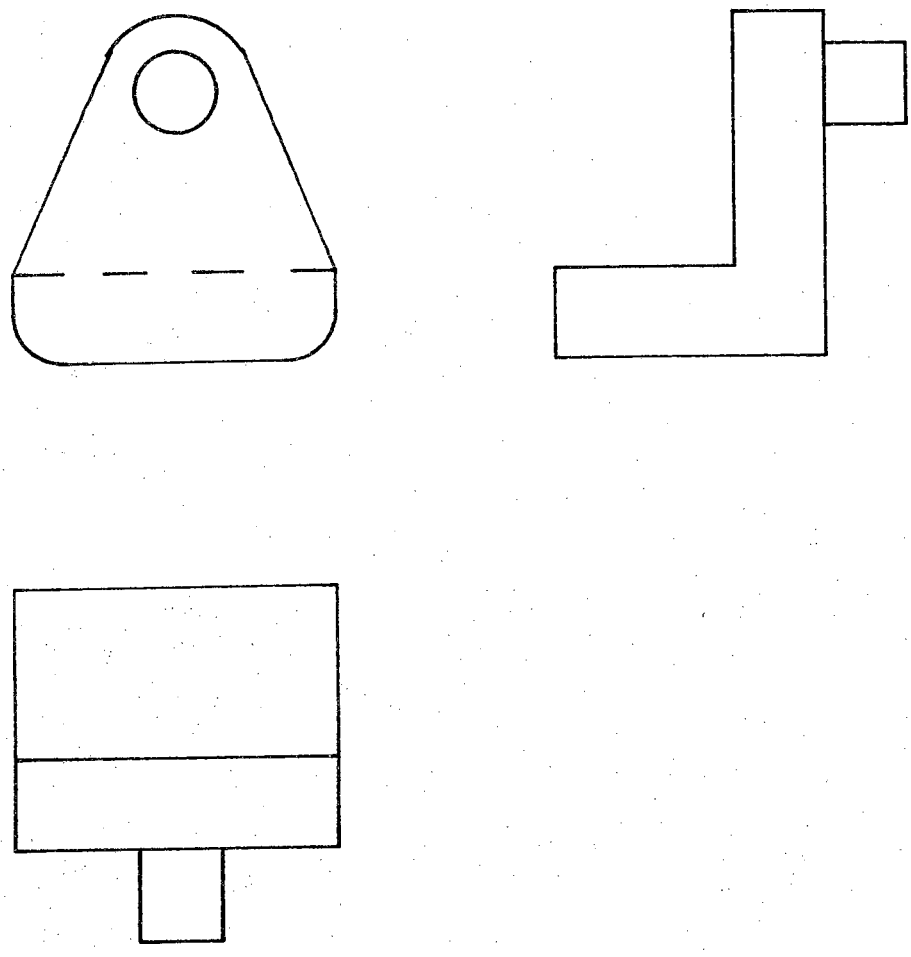


Figure 2.7 Example of a lug and round corners

this work is a subclass of the one considered by Aldefeld ([ALD1],[ALD2]), which is the uniform thickness objects. These objects generally have a plane base with arbitrary contour and a uniform thickness in the direction perpendicular to the base [ALD2]. Another constraint on the domain of the objects treated is that the curves appearing in the drawings should be either a circle or an arc belonging to a circle. This does not in general limit the domain of objects very much.

PROPOSED APPROACH

The method used in this work is like a "top_down" approach. That is the existence of a certain goal object (cube, lug,...) is assumed and then the different views are searched for primitives in order to find proof for the assumption. In the process of proof finding, the attributes needed to represent the object in terms of C.S.G. are extracted. The disadvantage of this approach is its lack of generality. However, adding more power to this algorithm, that is, making it capable of treating more complex objects does not require a major effort. This might be the case for previous polyhedra oriented algorithms because once curves are introduced in a drawing, the concept of vertex matching used in some previous approaches looses its significance. The advantage of this approach is that it is easier to have a volume oriented representation because the primitives used in this kind of modeling (e.g. cubes in CSG) are found and defined independently. In addition, the algorithm is relatively fast compared to some of the previous algorithms, when they are applied to similar line drawings.

FINAL REPRESENTATION

The output of this algorithm is the C.S.G. representation of the object depicted in the three orthogonal views. The most important advantage of this representation is that it is directly compatible with a C.A.D. system that uses Constructive Solid Geometry to represent objects that are stored in its data base. Other advantages of this volume oriented representation over the ones used previously are the lack of ambiguity (which is possible in wire frame representation) and boundedness of the object (which is not always guaranteed in surface oriented representations).

2.6 EXAMPLES

2.6.1 DATA INPUT

In order to examine the function of the algorithm, a few examples have been implemented using a manual input routine from the terminal. The conventions for inputing each 2D primitive is as follows: The first two entities to be entered are the TYPE (LINE = 1, CIRCLE = 2, ARC = 3) and the MODE (solid = 1, dashed = 0). Then, depending on TYPE the following entities are entered:

P_{ij} i = 1,2; j = 1,2. This is the jth coordinate of the ith endpoint.

C_i i = 1,2. This is the ith coordinate of the center.

POS This flag takes values from 1 to 4 depending on the position of the ARC with respect to its center.

RAD This is the radius of the CIRCLE.

Therefore a LINE is defined as follows:

$$1 \text{ MODE } P_{11} \; P_{12} \; P_{21} \; P_{22}$$

A CIRCLE is defined as follows:

$$2 \text{ MODE } C_1 \; C_2 \text{ RAD}$$

And an ARC is defined as follows:

$$3 \text{ MODE } P_{11} \; P_{12} \; P_{21} \; P_{22} \text{ POS } C_1 \; C_2$$

It should be noted that the first coordinate depends on the view. In view 1 the first coordinate is X, in view 2 it is Y, and in view 3 it is Z. The first line in the input list contains one digit which is the error margin. This is the error allowed when two coordinates are matched. That is, if the difference between two coordinates is smaller than this number, then the coordinates are said to be equal. This error margin is especially needed for the automatic input explained in Chapter 3. The views are entered in order and they are separated by -1. Finally, the order in which the primitives in a certain view are entered is not important.

2.6.2 OUTPUT CONVENTIONS

The output of the algorithm is a list of primitives separated by union (+) and difference (-) operators. All the primitives reside in a global coordinate system. Parentheses are used to separate different groups of primitives that have to be combined together. The result of the operation on the primitives in the parentheses is then added to the list. Those primitives that are not combined with other primitives in parentheses can be added or subtracted from the list globally. A MOVE operator is used when it becomes necessary to have the operation on

the primitives done locally and the result to be transferred to some global coordinates. These global coordinates are indicated by the MOVE operator.

For each primitive, a 3×4 matrix is printed which contains the information needed for the dimensions and the position of that primitive in the global coordinate system. The rows of the matrix correspond to the X, Y and Z axis. For the CUBE, the first column contains the three coordinates of one of the vertices. This point is called the origin of the CUBE. The second column contains the length of the cube in all three directions. The third and fourth columns correspond to the translation and rotation information. The concepts of translation and rotation are taken from the method of homogeneous transformations which is used in robotics and computer vision [PAUL]. For an example refer to the output of Example 2. In the case of CUBE 5, we have nonzero entries in columns 3 and 4. They should be interpreted as follows: translate the cube in the positive X direction 12 units. Then rotate the cube about the Z axis 29.743 degrees in the positive direction (using the right hand rule). For the CYLINDER and CONE, the first two columns are similar to the CUBE. The origin in the case of CYLINDER is the center of one of the circles (top or bottom). For the CONE, the origin is the center of the base circle. The radius is given in the third column and the rest of the entries are always zero.

2.6.3 SAMPLE EXECUTIONS

The algorithm has been implemented in "C" language on a Digital Equipment Corporation VAX 11/780 minicomputer under the UNIX

operating system. The following pages contain three sample executions of the algorithm. The line drawings are shown in figures 2.8, 2.9 and 2.10. Following each drawing there is the list of input primitives and the result of the execution. The meanings of the input and output lists are explained in Sections 2.6.1 and 2.6.2. Since the input is manual, an error margin of 1 unit is adequate because of the high accuracy of the coordinates entered.

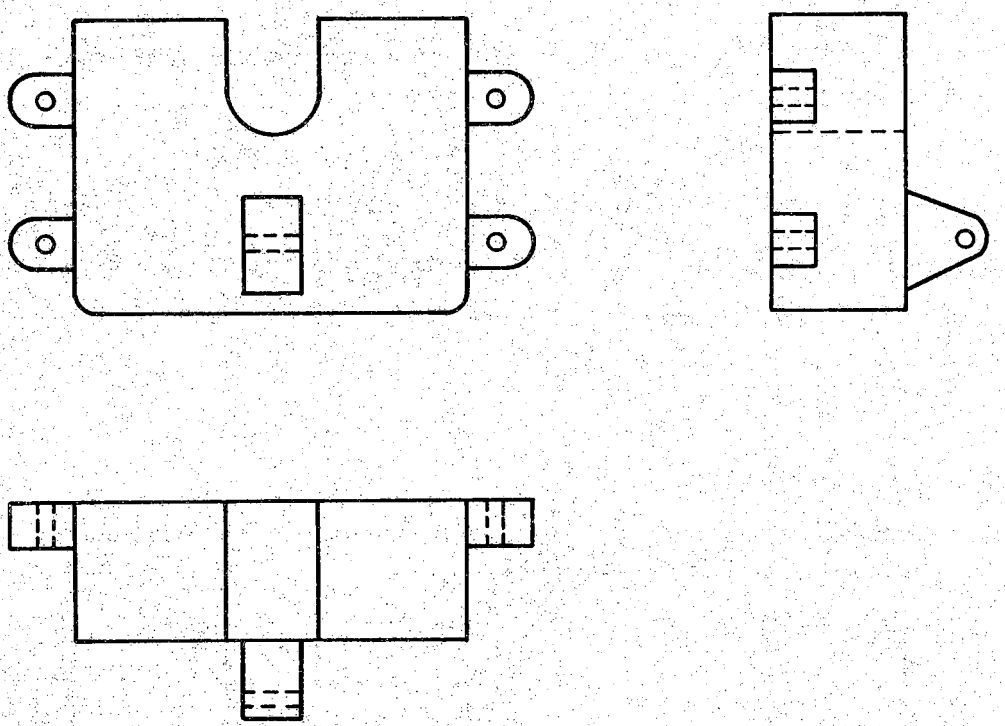


Figure 2.6 Example 1

INPUT LIST FOR EXAMPLE 1

```
1
3   1   8   15   16   15   2   12   15
3   1   2   12   2   16   4   2   14
3   1   2   4   2   8   4   2   6
3   1   22   12   22   16   3   22   14
3   1   22   4   22   8   3   22   6
3   1   4   1   5   0   10   5   1
3   1   20   1   19   0   10   19   1
2   1   2   14   .5
2   1   22   14   .5
2   1   2   6   .5
2   1   22   6   .5
1   1   4   20   8   20
1   1   16   20   20   20
1   1   20   20   20   16
1   1   20   16   20   12
1   1   20   12   20   8
1   1   20   8   20   4
1   1   20   4   20   1
1   1   19   0   5   0
1   1   4   1   4   4
1   1   4   4   4   8
1   1   4   8   4   12
1   1   4   12   4   16
1   1   4   16   4   20
1   1   11   9.5   13   9.5
1   1   11   .5   13   .5
1   1   11   .5   11   9.5
1   1   13   .5   13   9.5
1   0   11   5.75   13   5.75
1   0   11   4.25   13   4.25
1   1   8   15   8   20
1   1   16   15   16   20
1   1   2   16   4   16
1   1   2   12   4   12
1   1   20   16   22   16
1   1   20   12   22   12
1   1   2   4   4   4
1   1   2   8   4   8
1   1   20   4   22   4
1   1   20   8   22   8
-1
3   1   3   10.6   7   10.6   3   5   8.5
2   1   5   8.5   .75
1   1   20   0   20   7
1   1   20   7   9.5   7
1   1   9.5   7   .5   7
1   1   .5   7   0   7
```

```
1   1   0   7   0   0
1   1   0   0   4   0
1   1   4   0   8   0
1   1   8   0   12   0
1   1   12   0   16   0
1   1   16   0   20   0
1   1   16   0   16   3
1   1   12   0   12   3
1   1   16   3   12   3
1   1   .5   7   3   10.6
1   1   9.5   7   7   10.6
1   0   11   0   11   7
1   0   13.5   0   13.5   3
1   0   14.5   0   14.5   3
1   0   5.5   0   5.5   3
1   0   6.5   0   6.5   3
-1
1   1   0   0   0   4
1   1   0   4   0   8
1   1   0   8   0   16
1   1   0   16   0   20
1   1   0   20   0   24
1   1   0   24   3   24
1   1   3   24   3   20
1   1   3   20   7   20
1   1   7   20   7   16
1   1   7   16   7   13
1   1   7   13   11   13
1   1   11   13   11   11
1   1   11   11   7   11
1   1   7   11   7   8
1   1   7   8   7   4
1   1   7   4   3   4
1   1   3   4   3   0
1   1   3   0   0   0
1   1   0   4   3   4
1   1   0   8   7   8
1   1   0   16   7   16
1   1   0   20   3   20
1   1   7   11   7   13
1   0   0   1.5   3   1.5
1   0   0   2.5   3   2.5
1   0   0   21.5   3   21.5
1   0   0   22.5   3   22.5
1   0   7.75   11   7.75   13
1 0 9.25 11 9.25 13
-1
```

OUTPUT LIST FOR EXAMPLE 1

```
+ (CYLINDER7 - CUBE2) + CUBE3 + (CYLINDER8 - CUBE4)
+ CUBE5 + (CYLINDER9 - CUBE6) + CUBE7
+ (CYLINDER10 - CUBE8) + CUBE9
+ (CYLINDER11 - CUBE10)
+ MOVE (11.00,0.50,7.00)((CUBE11 - CUBE12) - CUBE13)
+ ((((CUBE16 - CUBE14) + CYLINDER12) - CUBE15) + CYLINDER13)
- CYLINDER1 - CYLINDER2 - CYLINDER3
- CYLINDER4 - CYLINDER5 - CYLINDER6 - CUBE1

     CYLINDER7
  2.000      0.000      2.000      0.000
 14.000      0.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CUBE2
  2.000      4.000      0.000      0.000
 10.000      8.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CUBE3
  2.000      2.000      0.000      0.000
 12.000      4.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CYLINDER8
  2.000      0.000      2.000      0.000
  6.000      0.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CUBE4
  2.000      4.000      0.000      0.000
  2.000      8.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CUBE5
  2.000      2.000      0.000      0.000
  4.000      4.000      0.000      0.000
  0.000      3.000      0.000      0.000

     CYLINDER9
 22.000      0.000      2.000      0.000
```

14.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CUBE6

18.000	4.000	0.000	0.000
10.000	8.000	0.000	0.000
0.000	3.000	0.000	0.000

CUBE7

20.000	2.000	0.000	0.000
12.000	4.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER10

22.000	0.000	2.000	0.000
6.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CUBE8

18.000	4.000	0.000	0.000
2.000	8.000	0.000	0.000
0.000	3.000	0.000	0.000

CUBE9

20.000	2.000	0.000	0.000
4.000	4.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER11

11.000	2.000	2.900	0.000
5.000	0.000	0.000	0.000
8.500	0.000	0.000	0.000

CUBE10

11.000	2.000	0.000	0.000
0.100	9.800	0.000	0.000
4.800	5.800	0.000	0.000

CUBE11

0.000	2.000	0.000	0.000
0.000	9.000	0.000	0.000
0.000	3.600	0.000	0.000

CUBE12

0.000	18.000	0.000	55.221
0.000	18.000	0.000	0.000
0.000	18.000	0.000	0.000

CUBE13

0.000	18.000	0.000	34.779
0.000	18.000	9.000	0.000
0.000	18.000	0.000	0.000

CUBE16

4.000	16.000	0.000	0.000
0.000	20.000	0.000	0.000
0.000	7.000	0.000	0.000

CUBE14

4.000	1.000	0.000	0.000
0.000	1.000	0.000	0.000
0.000	7.000	0.000	0.000

CYLINDER12

5.000	0.000	1.000	0.000
1.000	0.000	0.000	0.000
0.000	7.000	0.000	0.000

CUBE15

19.000	1.000	0.000	0.000
0.000	1.000	0.000	0.000
0.000	7.000	0.000	0.000

CYLINDER13

19.000	0.000	1.000	0.000
1.000	0.000	0.000	0.000
0.000	7.000	0.000	0.000

CYLINDER1

2.000	0.000	0.500	0.000
14.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER2

22.000	0.000	0.500	0.000
14.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER3

2.000	0.000	0.500	0.000
6.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER4

22.000	0.000	0.500	0.000
6.000	0.000	0.000	0.000
0.000	3.000	0.000	0.000

CYLINDER5

11.000	2.000	0.750	0.000
5.000	0.000	0.000	0.000
8.500	0.000	0.000	0.000

CYLINDER6

12.000	0.000	4.000	0.000
15.000	0.000	0.000	0.000
0.000	7.000	0.000	0.000

CUBE1

8.000	8.000	0.000	0.000
15.000	5.000	0.000	0.000
0.000	7.000	0.000	0.000

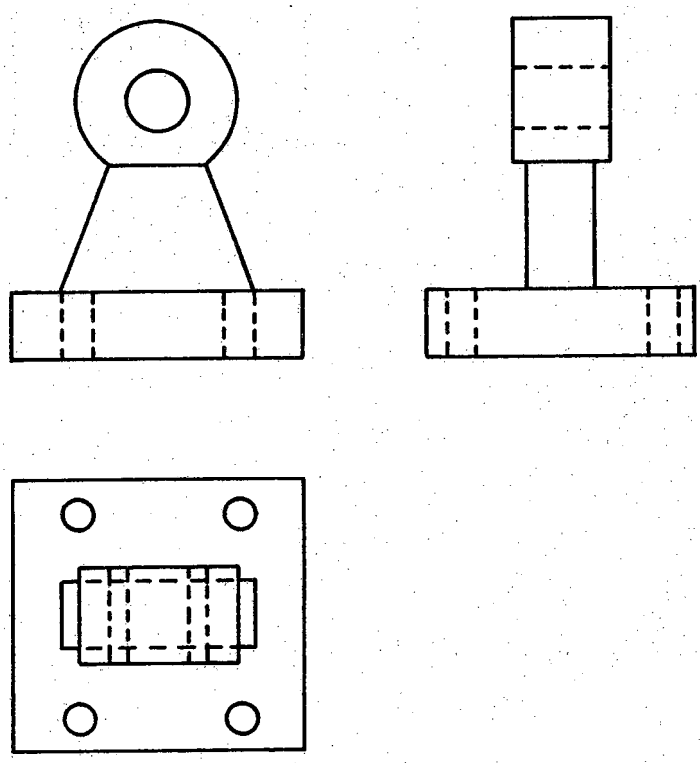


Figure 2.9 Example 2

INPUT LIST FOR EXAMPLE 2

```
1
3   1   8   11   12   11   1   10   14
2   1   10   14   1
1   1   8   11   4   4
1   1   4   4   0   4
1   1   0   4   0   0
1   1   0   0   20   0
1   1   20   0   20   4
1   1   20   4   16   4
1   1   16   4   12   11
1   1   12   11   8   11
1   1   4   4   16   4
1   0   1   0   1   4
1   0   3   0   3   4
1   0   17   0   17   4
1   0   19   0   19   4
-1
1   1   18   8   18   4
1   1   18   4   11   4
1   1   11   4   11   5
1   1   11   5   4   5
1   1   4   5   4   0
1   1   4   0   0   0
1   1   0   0   0   12
1   1   0   12   4   12
1   1   4   12   4   7
1   1   4   7   11   7
1   1   11   7   11   8
1   1   11   8   18   8
1   1   11   5   11   7
1   1   4   5   4   7
1   0   4   1   0   1
1   0   4   3   0   3
1   0   4   9   0   9
1   0   4   11   0   11
1   0   15   4   15   8
1   0   13   4   13   8
-1
2   1   2   2   1
2   1   2   18   1
2   1   10   2   1
2   1   10   18   1
1   1   0   0   0   20
1   1   0   20   12   20
1   1   12   20   12   0
1   1   12   0   0   0
1   1   5   4   5   16
1   1   5   16   7   16
```

```
1   1   7   16   7   4
1   1   7   4   5   4
1   1   4   6   4   14
1   1   4   14   8   14
1   1   8   14   8   6
1   1   8   6   4   6
1   1   4   6   4   14
1   0   5   8   7   8
1   0   4   9   8   9
1   0   4   11   8   11
1   0   5   12   7   12
-1
```

OUTPUT LIST FOR EXAMPLE 2

```
+ (CYLINDER6 - CUBE1) + CUBE2
+ MOVE (4.00,4.00,5.00)((CUBE3 - CUBE4) - CUBE5)
- CYLINDER1 - CYLINDER2 - CYLINDER3
- CYLINDER4 - CYLINDER5
```

CYLINDER6

10.000	0.000	3.606	0.000
14.000	0.000	0.000	0.000
4.000	4.000	0.000	0.000

CUBE1

4.394	11.211	0.000	0.000
3.789	7.211	0.000	0.000
4.000	4.000	0.000	0.000

CUBE2

0.000	20.000	0.000	0.000
0.000	4.000	0.000	0.000
0.000	12.000	0.000	0.000

CUBE3

0.000	12.000	0.000	0.000
0.000	7.000	0.000	0.000
0.000	2.000	0.000	0.000

CUBE4

0.000	24.000	0.000	0.000
0.000	24.000	0.000	0.000
0.000	24.000	0.000	60.257

CUBE5

0.000	24.000	12.000	0.000
0.000	24.000	0.000	0.000
0.000	24.000	0.000	29.743

CYLINDER1

10.000	0.000	1.000	0.000
14.000	0.000	0.000	0.000
4.000	4.000	0.000	0.000

CYLINDER2

2.000	0.000	1.000	0.000
0.000	4.000	0.000	0.000
2.000	0.000	0.000	0.000

CYLINDER3

18.000	0.000	1.000	0.000
0.000	4.000	0.000	0.000
2.000	0.000	0.000	0.000

CYLINDER4

2.000	0.000	1.000	0.000
0.000	4.000	0.000	0.000
10.000	0.000	0.000	0.000

CYLINDER5

18.000	0.000	1.000	0.000
0.000	4.000	0.000	0.000
10.000	0.000	0.000	0.000

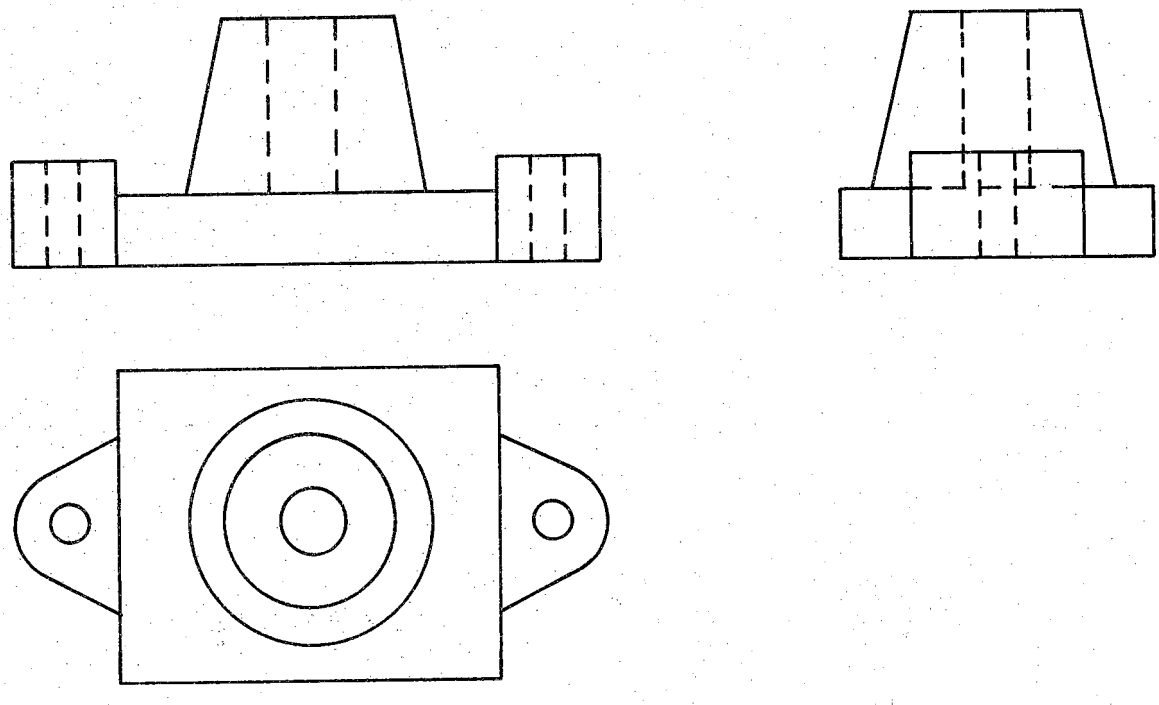


Figure 2.10 Example 3

INPUT LIST FOR EXAMPLE 3

```
1
1    1    .15    6    .15    0
1    1    .15    0    6    0
1    1    6    0    28    0
1    1    28    0    33.85    0
1    1    33.85    0    33.85    6
1    1    33.85    6    28    6
1    1    28    6    28    4
1    1    28    4    24    4
1    1    24    4    22    14
1    1    22    14    12    14
1    1    12    14    10    4
1    1    10    4    6    4
1    1    6    4    6    6
1    1    6    6    6    .15
1    1    6    4    6    0
1    1    28    4    28    0
1    1    10    4    24    4
1    0    2    6    2    0
1    0    4    6    4    0
1    0    30    6    30    0
1    0    32    6    32    0
1    0    15    4    15    14
1    0    19    4    19    14
-1
1    1    14    4    4    2
1    1    4    2    4    0
1    1    4    0    0    0
1    1    0    0    0    4
1    1    0    4    0    14
1    1    0    14    0    18
1    1    0    18    4    18
1    1    4    18    4    16
1    1    4    16    14    14
1    1    14    14    14    4
1    1    6    14    6    4
1    1    6    4    4    4
1    1    4    4    0    4
1    1    0    14    4    14
1    1    4    14    6    14
1    1    4    2    4    4
1    1    4    14    4    16
1    0    14    7    4    7
1    0    14    11    4    11
1    0    6    8    0    8
1    0    6    10    0    10
1    0    4    2    4    16
-1
```

```
2   1   9   17   2
2   1   9   17   5
2   1   9   17   7
2   1   9   3   1
2   1   9   31   1
3   1   6   2   12   2   4   9   3.5
3   1   6   32   12   32   3   9   30.5
1   1   4   6   6   2
1   1   12   2   14   6
1   1   4   28   6   32
1   1   12   32   14   28
1   1   0   6   4   6
1   1   4   6   14   6
1   1   14   6   18   6
1   1   18   6   18   28
1   1   18   28   14   28
1   1   14   28   4   28
1   1   4   28   0   28
1   1   0   28   0   6
-1
```

OUTPUT LIST FOR EXAMPLE 3

```
+ (CONE1 - CYLINDER1) + (CYLINDER5 - CUBE1)
+ MOVE(2.00,0.00,4.00)((CUBE2 - CUBE3) - CUBE4)
+ (CYLINDER6 - CUBE5)
+ MOVE (28.00,0.00,4.00)((CUBE6 - CUBE7) - CUBE8)
+ CUBE9 - CYLINDER2 - CYLINDER3 - CYLINDER4
```

```
      CONE1
  17.000      0.000      7.000      0.000
   4.000     35.000      0.000      0.000
   9.000      0.000      0.000      0.000

     CYLINDER1
  17.000      0.000      5.000      0.000
  14.000     35.000      0.000      0.000
   9.000      0.000      0.000      0.000

     CYLINDER5
   3.500      0.000      3.354      0.000
   0.000      6.000      0.000      0.000
   9.000      0.000      0.000      0.000

      CUBE1
   2.000      6.708      0.000      0.000
   0.000      6.000      0.000      0.000
   2.646     12.708      0.000      0.000

      CUBE2
   0.000      4.000      0.000      0.000
   0.000      6.000      0.000      0.000
   0.000     10.000      0.000      0.000

      CUBE3
   0.000     20.000      4.000      0.000
   0.000     20.000      0.000    206.566
   0.000     20.000      0.000      0.000

      CUBE4
   0.000     20.000      4.000      0.000
   0.000     20.000      0.000   -116.566
   0.000     20.000     10.000      0.000
```

CYLINDER6

30.500	0.000	3.354	0.000
0.000	6.000	0.000	0.000
9.000	0.000	0.000	0.000

CUBE5

25.292	6.708	0.000	0.000
0.000	6.000	0.000	0.000
2.646	12.708	0.000	0.000

CUBE6

0.000	4.000	0.000	0.000
0.000	6.000	0.000	0.000
0.000	10.000	0.000	0.000

CUBE7

0.000	20.000	0.000	0.000
0.000	20.000	0.000	63.434
0.000	20.000	0.000	0.000

CUBE8

0.000	20.000	0.000	0.000
0.000	20.000	0.000	26.566
0.000	20.000	10.000	0.000

CUBE9

6.000	22.000	0.000	0.000
0.000	4.000	0.000	0.000
0.000	18.000	0.000	0.000

CYLINDER2

17.000	0.000	2.000	0.000
4.000	10.000	0.000	0.000
9.000	0.000	0.000	0.000

CYLINDER3

3.000	0.000	1.000	0.000
0.000	6.000	0.000	0.000
9.000	0.000	0.000	0.000

CYLINDER4

31.000	0.000	1.000	0.000
0.000	6.000	0.000	0.000
9.000	0.000	0.000	0.000

2.7 PERFORMANCE ANALYSIS

Table 2.1 contains the CPU time needed for the construction of the CSG model of the examples in Section 2.6.3. This time is obviously affected by the complexity of the object. Complex parts have complicated line drawings in their projections. These drawings in turn have a large number of graphical elements which causes a larger search list for each view. A second important factor is the type of graphical elements. For example, for each horizontal line present in view 1, all three views are searched at least twice for other lines in the cube routine. On the other hand the absence of horizontal lines in view 1 causes the cube routine to terminate after only one scan through the primitives of that view.

The precision of the input data, which is expected in a manual system, causes the 3D model of the object to be very accurate. That is, there are no errors in the position coordinates and the dimensions of the 3D primitives. For instance, in the output of Example 3, CONE1 - CYLINDER1 results in the accurate representation of the conical object whose projections are seen in figure 2.10.

Another characteristic of the output model is that in general all of the holes present in the input object are represented in terms of 3D primitives that are subtracted globally from the rest of the object. For this reason, these primitives appear at the end of the output list. This is demonstrated in Example 1 where the last 7 primitives correspond to the holes present in the object of figure 2.8.

It should be noted that in the final model of the object, the absolute position of the various primitives is not important. We are concerned mainly with the dimensions and the relative position of the primitives

TABLE 2.1: Time analysis for Examples 1, 2 & 3

CPU TIME (sec)		
EXAMPLE 1	EXAMPLE 2	EXAMPLE 3
1.4	.7	.9

with respect to each other. Therefore, the origin of the global coordinate system can be chosen quite arbitrarily.

CHAPTER THREE

MANUAL AND AUTOMATIC INPUT OF LINE DRAWINGS

3.1 INTRODUCTION

As explained previously, the input to the reconstruction algorithm is the type of primitives found in each view, along with all the information needed to uniquely define those primitives. More clearly, there are three different primitives possible in our line drawings and they are a straight LINE, a CIRCLE and an ARC. For these three elements the minimum information needed is as follows.

LINE: The coordinates of the two endpoints and the mode (solid or cavity).

CIRCLE: The coordinates of the center, the radius and the mode.

ARC: The coordinates of the two endpoints, the coordinates of the center and the position of the body with respect to the center.

In the following sections, two different input systems are proposed. Because of the requirements for speed and automation, more emphasis is put on the automatic system.

3.2 MANUAL INPUT SYSTEM

A manual input system can be designed as follows: the drawings are placed on a digitizing tablet such as a TEKTRONIX 4954. Using the cursor , the operator then points to points of interest and presses the button on the cursor. Each time the button is pressed, the (x,y) coordinates of that point are recorded. The operator is also required to interact with an input routine that asks simple questions like the type and the mode of the primitive to be entered next. Depending on the type of the primitive, the routine expects two (for LINE and CIRCLE) or three (for ARC) points to be entered using the cursor.

The advantage of this process is its accuracy. The error involved in entering the position of a point is negligible when compared to the size of the drawings. Another advantage is that dashed lines and curves are entered just as easily as solid lines and curves are. The disadvantage of this method, however, is that it needs the involvement of an operator and in the case of very complex drawings, the process becomes rather tedious. One way to improve this method is by using the fact that each primitive, except CIRCLE, is connected to at least two other primitives. Therefore, we can have a rule that requires that the last point entered for a primitive be also the first point for the next primitive unless otherwise indicated. If this convention is followed, the number of points to be entered can be cut to almost a half, depending on the drawing.

3.3 AUTOMATIC INPUT SYSTEM

3.3.1 IMAGE DIGITIZATION AND PREPROCESSING

An alternate method of obtaining the information needed by the reconstruction algorithm is to digitize the drawings and then extract the primitives and their attributes from the digital picture. A block diagram of such a system is shown in figure 3.1. The advantage of this method is of course lack of operator involvement and speed. In an ideal system, the drawing is put under the camera and is digitized. The resulting data is stored in a file and an image processing algorithm is applied to it. The output of the algorithm is then fed into the reconstruction routine as explained previously.

There are several problems to be overcome with the kind of system described above. In order to obtain a good image, the lighting should be controlled so that we can avoid unnecessary bright and dark spots. Resolution can also be a problem. Since the reconstruction algorithm relies heavily on coordinate matching, it is imperative that the scanner produces an image with adequate resolution. There are also other difficulties related to scanning such as noise and distortion, but in a controlled environment, it can be assumed that the above mentioned problems are minimal. It is then safe to assume that using adequate measures when scanning the picture and also some preprocessing (e.g. thinning and thresholding), a binary picture can be obtained in which each dark (value = 1) pixel in general has at most two dark 4_neighbors unless it is at the intersection of two primitives. The importance of this condition will be seen later when the curve following algorithm is

discussed. Before proposing a method to extract the needed information from such a picture, a review of rules governing ideal digital straight lines is adequate.

3.3.2 DIGITAL STRAIGHT LINES

In a digital picture, a neighbor means any of the eight horizontal, vertical, or diagonal neighbors of the pixel. A digital arc, S, is a connected set of lattice points all but two of which have exactly two neighbors in S. Let p,q be points of the digital picture subset S, and let pq denote the (real) line segment between p and q. We say that pq lies near S if, for any (real) point (x,y) of pq, there exists a lattice point (i,j) of S such that

$$\max[\text{abs}(i - x), \text{abs}(j - y)] < 1$$

We say that S has the so called *chord property* if, for every p,q in S, the chord pq lies near S. Rosenfeld [ROS] has shown that the digitization of a line segment is a digital arc and has the chord property. In addition, if a digital arc has the chord property, it is the digitization of a straight line segment. Using the above theorems, Rosenfeld comes up with a number of useful regularity properties of digitized straight lines. Defining a *run* to be a collection of consecutive 1's in the same direction, the rules are as follows:

1- The runs in a digital arc have at most two directions, differing by 45 degrees, and for one of these directions, the run length must be 1.

2- The runs can have only two lengths, which are consecutive integers.

3- One of the run lengths can occur only once at a time.

4- For the run length that occurs in runs, these runs can themselves have only two lengths, which are consecutive integers.

The above rules apply only when we have a perfect digitization of thin lines. In practice, the characteristics of an imperfect scanner have to be held into consideration. In case of straight lines with horizontal or vertical slopes, the result of perfect digitization and thinning is one row or column of consecutive dark pixels. However, the result of a real scanner can be different as shown in figure 3.2. The break shown in the figure can happen more than once, and it can be in both left and right directions. A good algorithm should be able to handle this kind of distortion.

3.3.3 DIGITAL ARCS

A digital arc can be detected in a picture by using the fact that the curvature along the arc should be a nonzero constant within some error margin. One way to find the curvature is to use an algorithm similar to the one by Freeman and Davis [FRE1].

In their corner finding algorithm, Freeman and Davis detect the curvature of a chain coded curve by scanning the chain with a moving line segment which connects the end points of a sequence of links. As the line segment moves from one chain node to the next, the angular differences between successive segment positions are used as a smoothed measure of local curvature along the chain. Using this method, the start and end points of an arc can be detected with relatively good accuracy.

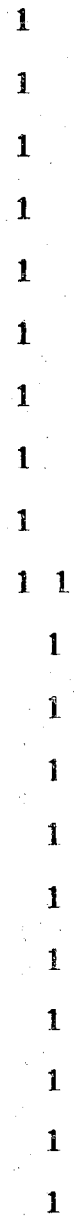


Figure 3.2 Imperfect digitization of a straight line

3.4 THE ALGORITHM

An algorithm using the results of the previous section has been designed and is illustrated at the end of this section. A brief description follows:

After the image has been preprocessed, it is scanned from left to right, top to bottom. Once a pixel with value 1 is reached the curve is followed and chain coded [FRE2],[FRE3]. The criterion for following the curve is to check the 4 neighbors first and then the other neighbors of the pixel. This way, priority is given to the 4 neighbors. It is also guaranteed that all the neighbors of the pixel are covered (unless the pixel is at an intersection). After passing each pixel, its value is turned from 1 to 2 in order to indicate that it has already been covered. This guarantees that the same curve will not be traced again. The other advantage of this procedure will become obvious later. The curve following procedure stops when no more neighbors with value equal to 1 can be found. At this point three cases are possible:

1- CLOSED CURVE

If the last pixel covered has as neighbor the start point of the chain, it means that we have a closed curve which is possibly a combination of straight line segments and arcs. In order to extract horizontal and vertical line segments, the difference array is calculated as follows for each link in the chain:

$$diff_i = link_{i+1} - link_i$$

Horizontal and vertical lines are characterized by consecutive zeroes in the difference array. The minimum number of zeroes needed in order to ensure the presence of a straight line depends on the size of the

image. The problem of breaks in the straight line as mentioned in section 3.3.2 can be handled by realizing that such breaks are characterized by two consecutive nonzero elements with opposite signs in the difference array. Except for the latter case, a nonzero element in the array means that the end of a vertical/horizontal line is reached.

Slanted lines and arcs both give nonzero values in the difference array. In order to obtain the start and end points of a slanted line or an arc, the local curvature is calculated for each pixel and summed up. If at any point the sum starts to rise above some threshold, then that is the start point of an arc. Somewhere along the line this value stops increasing and remains constant. That is the endpoint of the arc. Finally, where ever the sum is relatively constant, it is assumed that the curve is a straight line.

In case no straight lines can be found in the closed curve, the curve is assumed to be a circle. The radius and the approximate center can be calculated by solving the equation of the circle using three points on its perimeter.

2- DASHED LINE

If the last pixel covered has only one nonzero neighbor, then we have encountered a dashed line. The procedure for following a dashed line can be rather complex unless a few assumptions are made. For example if it is assumed that the dashed lines are either horizontal or vertical and with a relatively short length (so that breaks do not occur), the end of the dashed line can be found by just moving in the previous direction until a pixel with value equal to 2 is found. This procedure also decreases the possibility of error in case the thinning algorithm has

failed to totally thin the dashed lines. It should be noted that if the dashed lines are too short, they might be mistaken for noise and therefore ignored by the curve following routine.

3- OPEN CURVE

Finally, if the last pixel has two or more neighbors with values equal to two, then we have an open curve. This curve is the combination of one or more straight lines that can be extracted by using a similar method as for closed curves.

The algorithm goes as follows:

```
BEGIN:
  scan the image;
  IF no pixels with value equal to one is found THEN
    EXIT;
  ELSE
    follow curve and chain code it until end condition is met. Change
    the value of each pixel on the curve from 1 to 2;
  END
  IF the last pixel on the curve has as neighbor the first pixel on the
  curve THEN
    /* It is a closed curve */
    compute the difference array from the chain code array;
    using the difference array, find all vertical and horizontal lines;
    IF there are no straight lines THEN
      /* It is a circle */
      find the center and radius of the circle;
      output a circle;
      GO TO BEGIN;
    ELSE
      output the lines;
    END
    IF there are no breaks between the lines THEN
      GO TO BEGIN;
```

```
      ELSE /* There are arcs or slanted lines between the previous lines
      */
        find the position of the arc or slanted line or both using the cur-
        vature function;
        IF there is an arc THEN
          find its endpoints and center;
          output an arc;
        END
        IF there is a line THEN
          find its endpoints;
          output a line;
        END
        GO TO BEGIN
      END
    END
    ELSE IF the last pixel has a 2 neighbor THEN
      /* It is the combination of one or more straight lines */
      extract the lines using the difference array;
      output the lines;
    ELSE
      /* It is a dashed line */
      proceed in the last direction in the chain code to find the endpoint
      of the dashed line;
      output the dashed line;
    END
    GO TO BEGIN
  END
```

3.5 EXAMPLES

The algorithm described in section 3.4 has been implemented in "C" language on a Digital Equipment Corporation VAX 11/780 minicomputer under the Berkeley 4.2 UNIX operating system. The drawings shown in figures 3.3 and 3.10 were photographed and the negatives were digitized using an OPTRONIX P1000 drum scanner with a 100 micron resolution. The resulting images were 300 × 300 pixels of 8 bits each. The images were eventually reduced to 200 × 200 pixels by cutting out the edges of the picture that did not contain any drawings. These are

shown in figures 3.4 through 3.6 and figures 3.11 through 3.13. An appropriate threshold was then found in order to separate the drawings from the background. The original picture was then transformed into a binary image using this threshold. Because of the high resolution of the scanner, the resulting lines are in general more than one pixel thick. This may cause ambiguities for the curve following routine and therefore a thinning algorithm is applied to the thresholded image in order to eliminate this problem. In general, the amount of noise in the picture after thresholding is small because of the considerations made during digitization (e.g. uniform lighting). In addition, the algorithm has a limited capability to distinguish between noise and elements belonging to the drawings. This is discussed later in this chapter. The results of the above preprocessing are shown in figures 3.7 through 3.9 and 3.14 through 3.16. The algorithm explained in Section 3.4 was applied to these images separately. The coordinates of the primitives found in each picture were shifted appropriately in order to make the three views compatible. This is necessary because the reconstruction algorithm relies on a global coordinate system for matching appropriate primitives together, as explained in Chapter 2. The output of the algorithm is the list of primitives found in each view. The conventions used to describe the primitives are the same as in Chapter 2. This list of primitives was then fed into the reconstruction algorithm and the 3D representation of the object was obtained. These are shown in the following pages. For a complete explanation of the meanings of the input and output lists, the reader should refer to Sections 2.6.1 and 2.6.2.

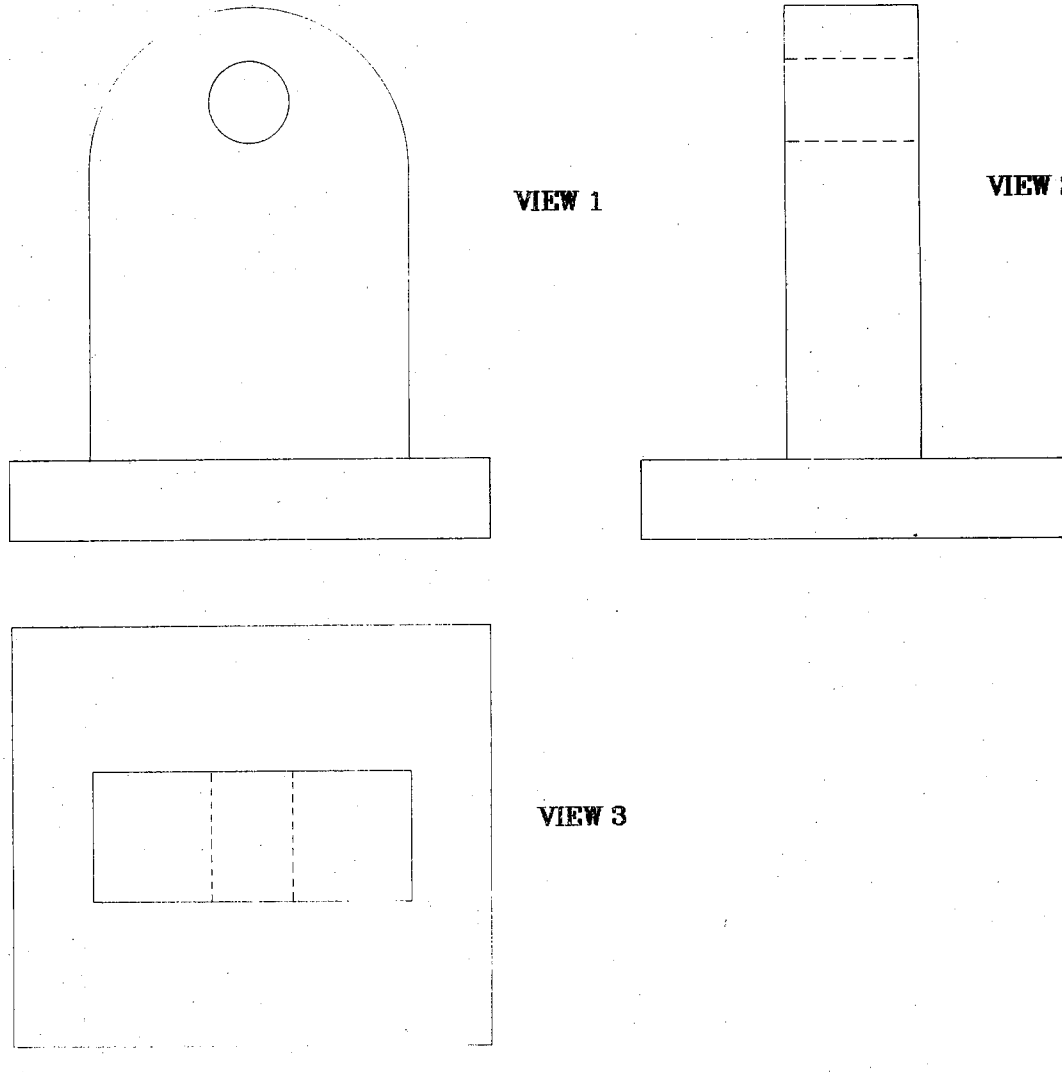


Figure 3.3 Example 4

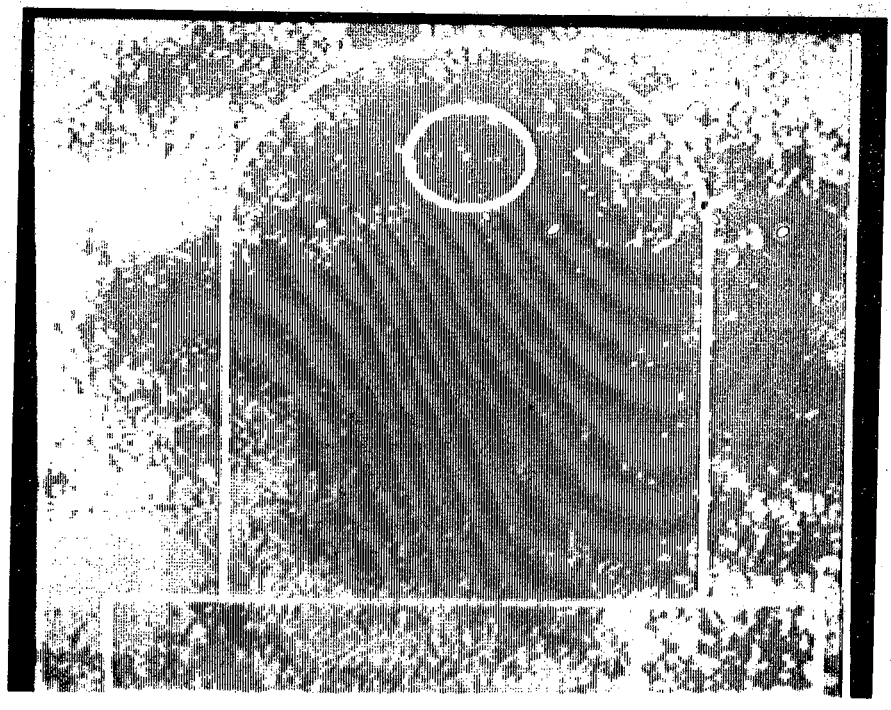


Figure 3.4 Digital image of view 1 of Example 4

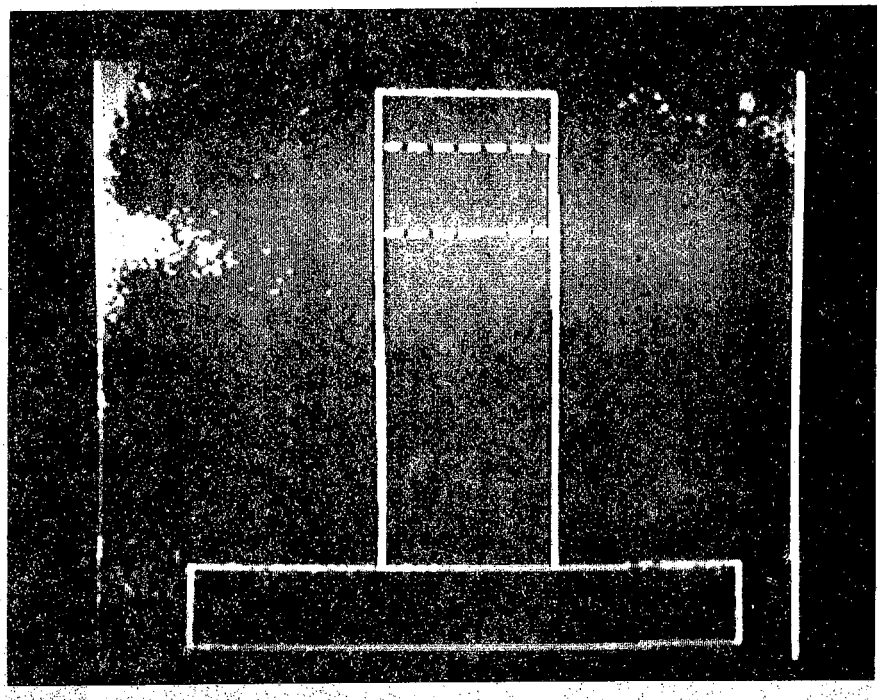


Figure 3.5 Digital image of view 2 of Example 4

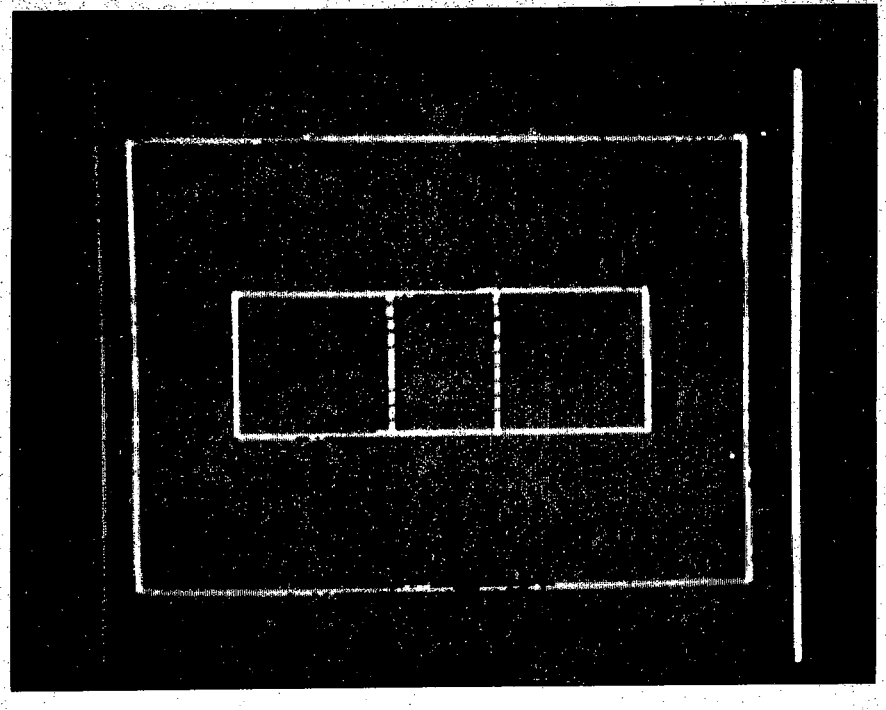


Figure 3.6 Digital image of view 3 of Example 4

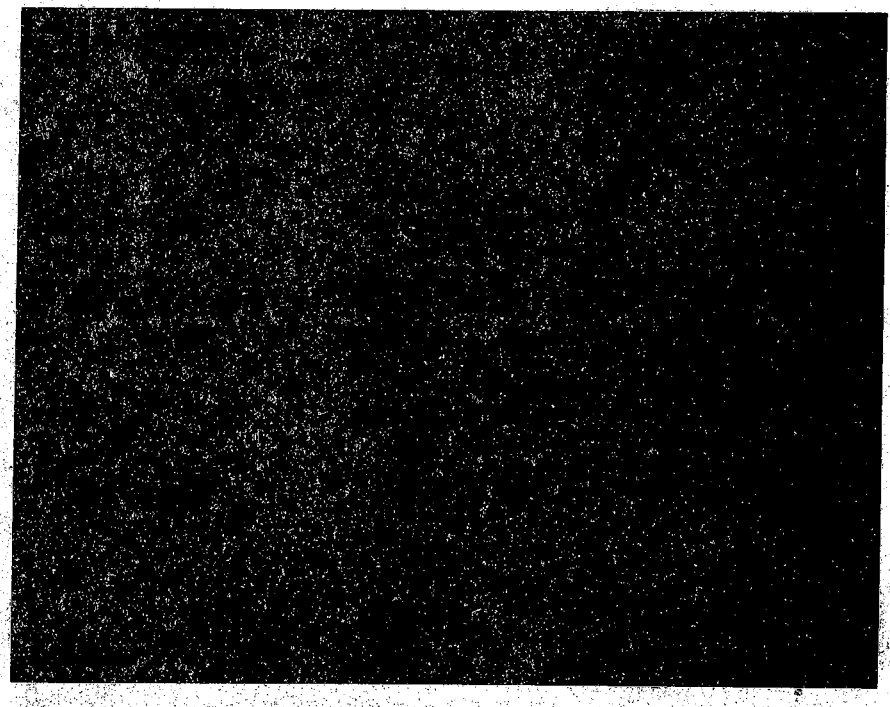


Figure 3.7 View 1 of Example 4 after preprocessing

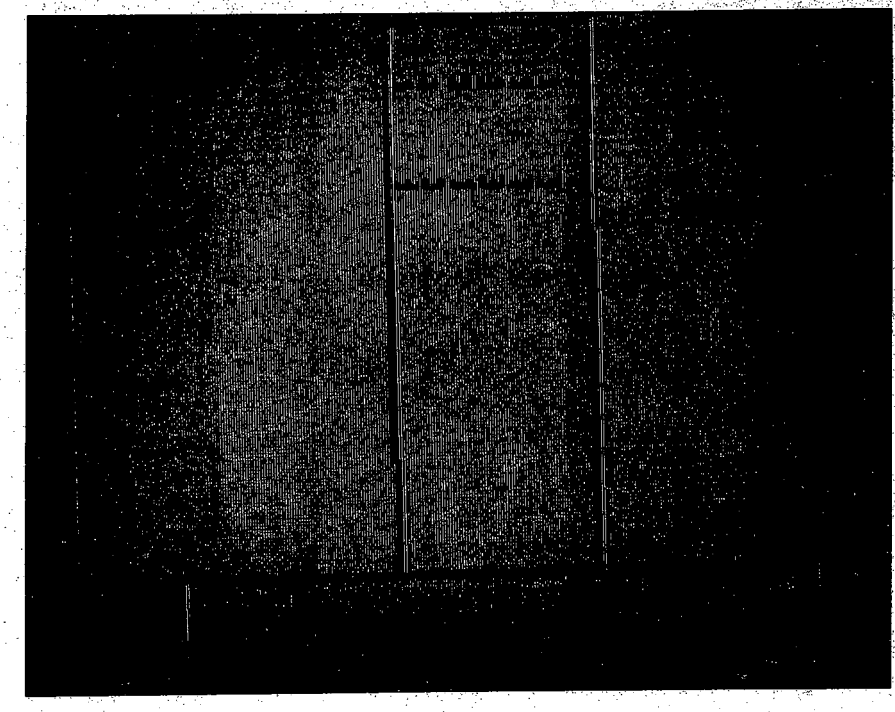


Figure 3.8 View 2 of Example 4 after preprocessing

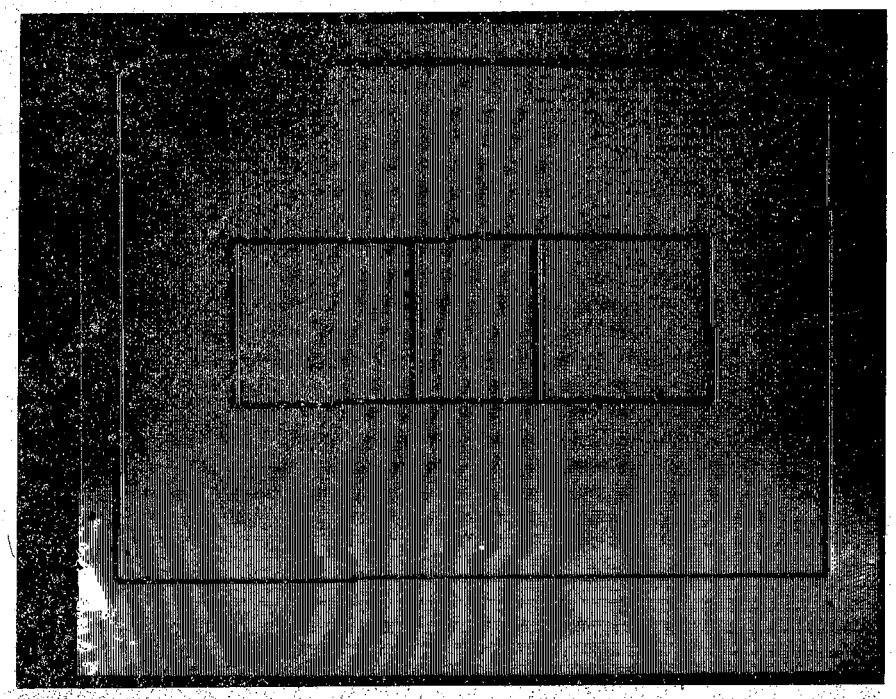


Figure 3.9 View 3 of Example 4 after preprocessing

PRIMITIVES EXTRACTED FROM VIEW 1

```
1 1  29.0 148.0  27.0  29.0

1 1  27.0  29.0   0.0  28.0

1 1   0.0  28.0   1.0   0.0

1 1   1.0   0.0 174.0   1.0

1 1 174.0   1.0 173.0  29.0

1 1 173.0  29.0 146.0  30.0

1 1 146.0  30.0 145.0 145.0

3 1 145.0 145.0  29.0 148.0 1  88.1 136.0

2 1 85.82 160.57 15.05

1 1  30.0  29.0 145.0  29.0
```

PRIMITIVES EXTRACTED FROM VIEW 2

1	1	29.0	52.0	28.0	0.0
1	1	28.0	0.0	0.0	1.0
1	1	0.0	1.0	1.0	157.0
1	1	1.0	157.0	29.0	156.0
1	1	29.0	156.0	30.0	102.0
1	1	30.0	102.0	195.0	101.0
1	1	195.0	101.0	194.0	53.0
1	1	194.0	53.0	29.0	52.0
1	0	176.0	76.0	176.0	101.0
1	0	147.0	69.0	147.0	101.0
1	1	29.0	55.0	29.0	101.0

PRIMITIVES EXTRACTED FROM VIEW 3

1	1	155.0	1.0	154.0	176.0	
1	1	154.0	176.0	0.0	175.0	
1	1	0.0	175.0	1.0	0.0	
1	1	1.0	0.0	155.0	1.0	
1	1	101.0	30.0	100.0	147.0	
1	1	100.0	147.0	52.0	146.0	
1	1	52.0	146.0	53.0	29.0	
1	1	53.0	29.0	101.0	30.0	
1	0	53.0	103.0	100.0	103.0	
1	0	63.0	73.0	100.0	73.0	

THE OUTPUT OF THE RECONSTRUCTION ALGORITHM

```
+ (CYLINDER2 - CUBE1)
+ CUBE2 + CUBE3 - CYLINDER1
```

CYLINDER2

88.100	0.000	57.607	0.000
136.000	0.000	0.000	0.000
53.000	48.010	0.000	0.000

CUBE1

-28.607	231.215	0.000	0.000
32.785	115.215	0.000	0.000
52.990	48.010	0.000	0.000

CUBE2

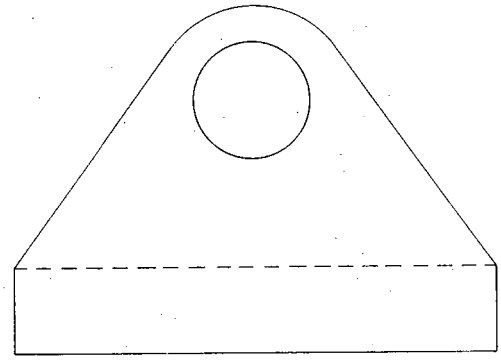
29.000	116.000	0.000	0.000
28.983	119.017	0.000	0.000
52.990	48.010	0.000	0.000

CUBE3

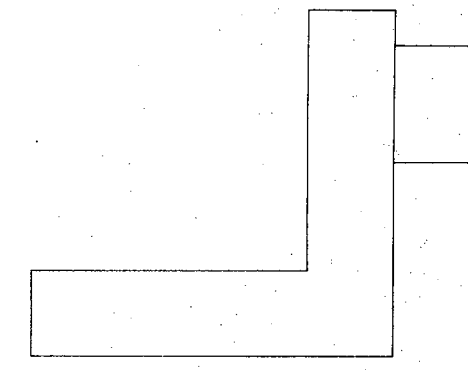
0.000	173.000	0.000	0.000
0.000	28.018	0.000	0.000
1.000	154.003	0.000	0.000

CYLINDER1

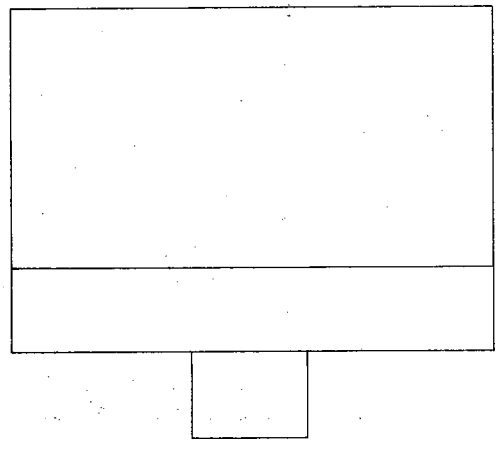
85.820	0.000	15.050	0.000
160.570	0.000	0.000	0.000
63.000	37.000	0.000	0.000



VIEW 1



VIEW 2



VIEW 3

Figure 3.10 Example 5

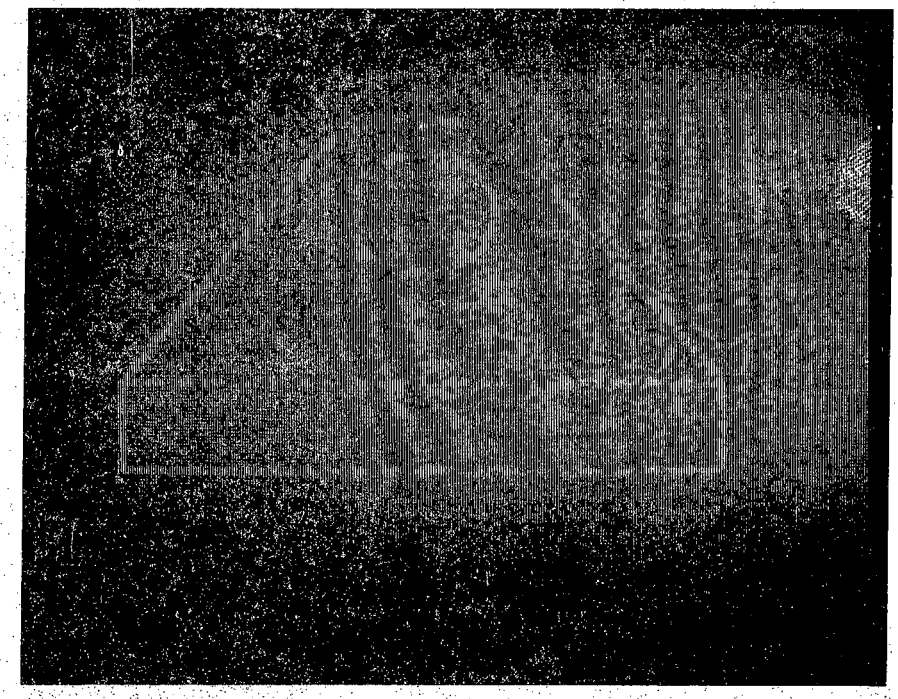


Figure 3.11 Digital image of view 1 of Example 5

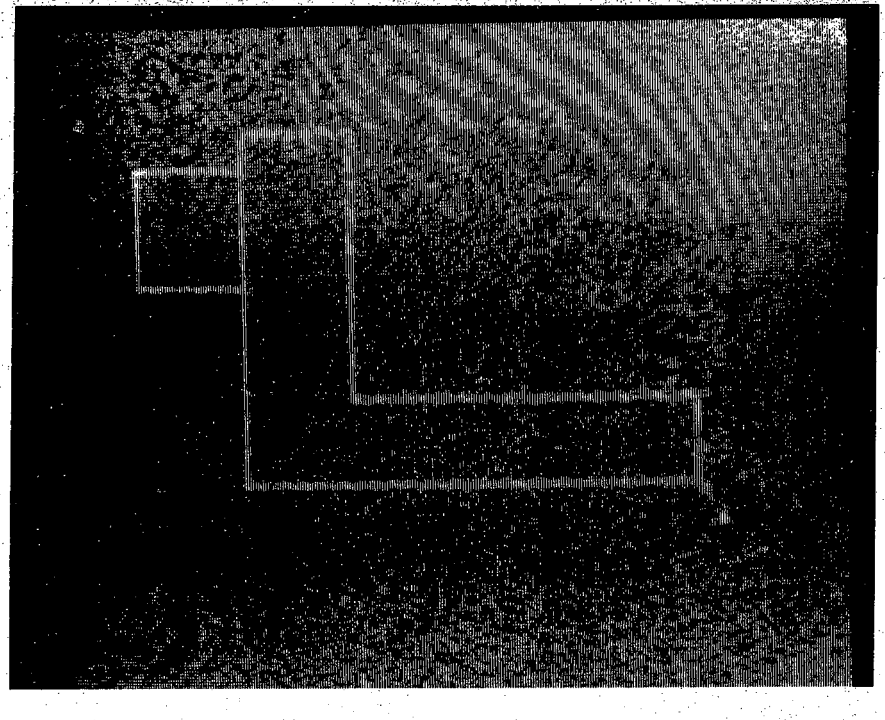


Figure 3.12 Digital image of view 2 of Example 5

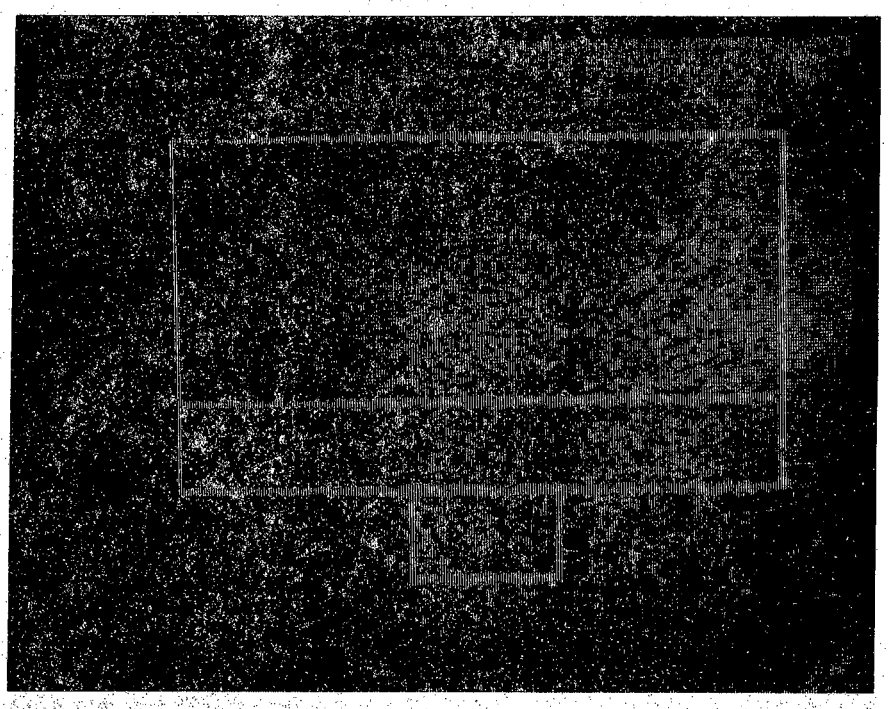


Figure 3.13 Digital image of view 3 of Example 5

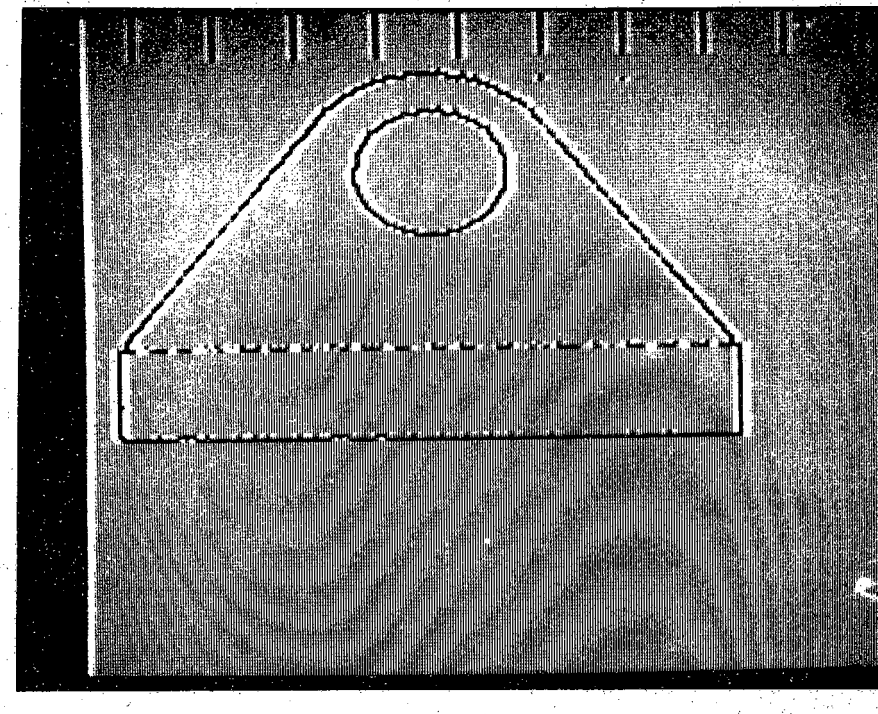


Figure 3.14 View 1 of Example 5 after preprocessing

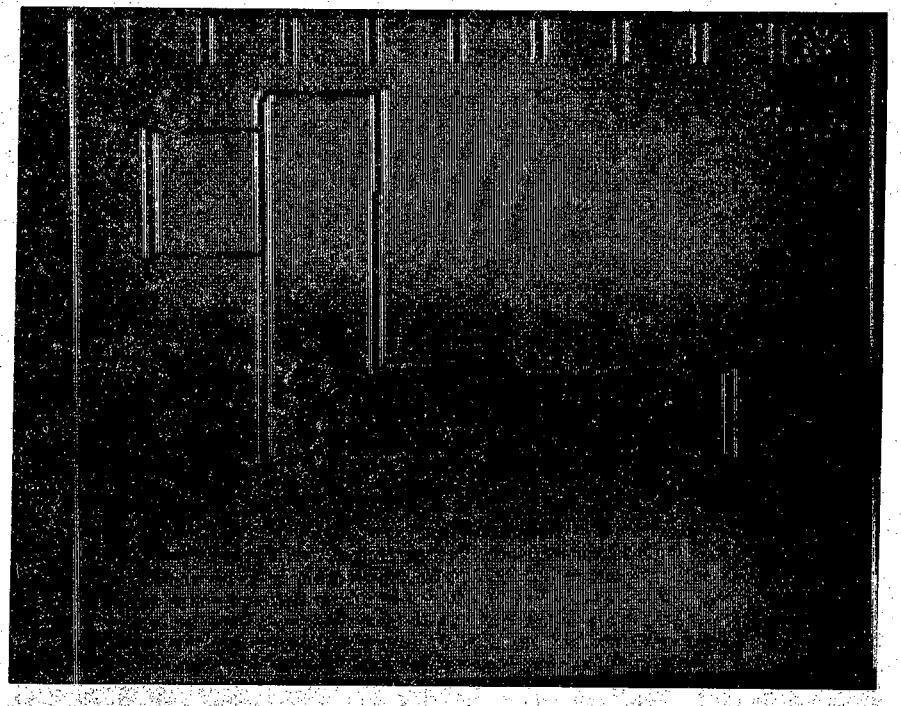


Figure 3.15 View 2 of Example 5 after preprocessing

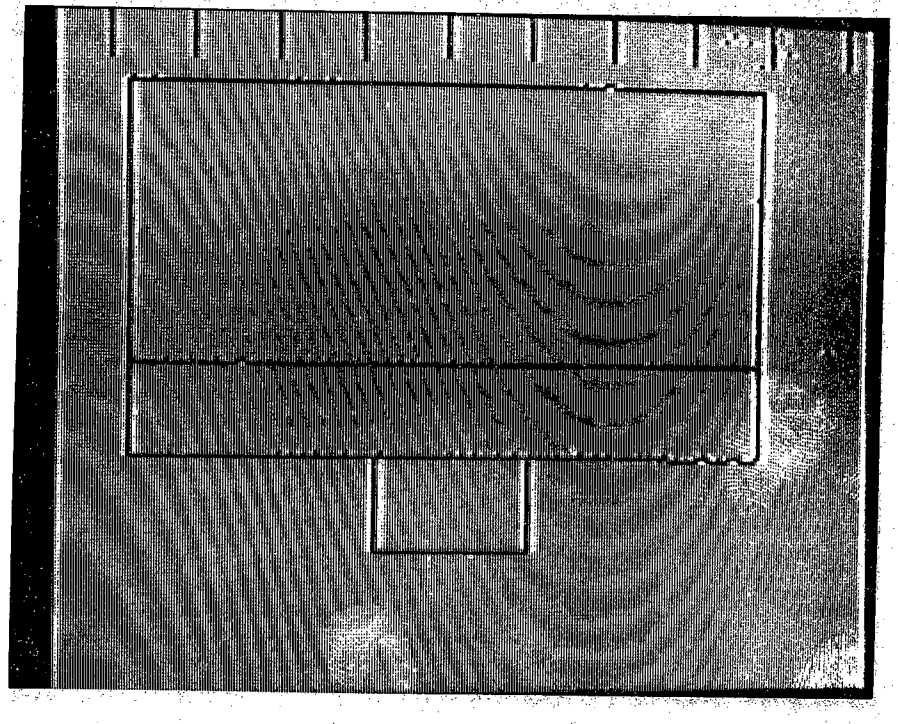


Figure 3.16 View 3 of Example 5 after preprocessing

PRIMITIVES EXTRACTED FROM VIEW 1

```
1 1   0.0  26.0   2.0   0.0

1 1   2.0   0.0 154.0   1.0

1 1 154.0   1.0 153.0  27.0

3 1  97.0 102.0  52.0  98.0 1  76.4  80.4

1 1 153.0  27.0  97.0 102.0

1 1  52.0  98.0   0.0  26.0

2 1 80.18 80.59 18.60

1 0   7.0  28.0 151.0  27.0
```

PRIMITIVES EXTRACTED FROM VIEW 2

1	1	108.0	87.0	27.0	87.0	
1	1	27.0	87.0	25.0	0.0	
1	1	25.0	0.0	0.0	1.0	
1	1	0.0	1.0	1.0	115.0	
1	1	1.0	115.0	61.0	116.0	
1	1	61.0	116.0	62.0	142.0	
1	1	62.0	142.0	98.0	141.0	
1	1	98.0	141.0	99.0	115.0	
1	1	109.0	113.0	108.0	87.0	
1	1	99.0	115.0	109.0	113.0	
1	1	96.0	115.0	62.0	115.0	

PRIMITIVES EXTRACTED FROM VIEW 3

1	1	1.0	0.0	109.0	1.0		
1	1	109.0	1.0	110.0	59.0		
1	1	110.0	59.0	137.0	60.0		
1	1	137.0	60.0	136.0	96.0		
1	1	136.0	96.0	109.0	97.0		
1	1	109.0	97.0	108.0	154.0		
1	1	108.0	154.0	0.0	153.0		
1	1	0.0	153.0	1.0	0.0		
1	1	82.0	1.0	82.0	153.0		
1	1	109.0	60.0	109.0	95.0		

THE OUTPUT OF THE RECONSTRUCTION ALGORITHM

```
+ CYLINDER1 + (CYLINDER2 - CUBE1) + MOVE (0.00,26.00,87.00)((CUBE2
- CUBE3) - CUBE4) + CUBE5

        CYLINDER1
    80.180      0.000     18.600      0.000
    80.590      0.000      0.000      0.000
   110.000     27.019      0.000      0.000

        CYLINDER2
    76.400      0.000     29.848      0.000
    80.400      0.000      0.000      0.000
    87.000     26.019      0.000      0.000

         CUBE1
    22.152    104.697      0.000      0.000
    38.303     59.697      0.000      0.000
    86.981     26.019      0.000      0.000

         CUBE2
     0.000    153.000      0.000      0.000
     0.000     76.000      0.000      0.000
     0.000     26.019      0.000      0.000

         CUBE3
     0.000    306.000      0.000      0.000
     0.000    306.000      0.000      0.000
     0.000    306.000      0.000     54.164

         CUBE4
     0.000    306.000    153.000      0.000
     0.000    306.000      0.000      0.000
     0.000    306.000      0.000     35.836

         CUBE5
     2.000    152.000      0.000      0.000
     0.000     26.077      0.000      0.000
     1.000    108.005      0.000      0.000
```

3.6 PERFORMANCE ANALYSIS

The CPU time needed for extracting the primitives from each view using the automatic input algorithm is demonstrated in Table 3.1. As it was expected, the processing time for Example 5 is longer than the time for Example 4. This is due to the fact that the images of Example 4 contained less distortion and noise as explained previously. In general, the factors that play important roles in determining the CPU time are the image size, number of primitives and the amount of noise. Each time a curve of length less than 5 pixels is detected, it is considered to be noise. This number is dependent on the size and type of the drawings in the image and should be supplied to the algorithm. For our case, considering the length of the dashed lines, a minimum length of 5 pixels turns out to be very appropriate.

One weakness of the algorithm can be the processing of short dashed lines. In some cases it might be difficult to differentiate between the start of a dashed line and noise. However once the start of such lines is detected, the length of the remaining dashes is not important. Therefore the possibility of the occurence of such a situation is rather small. Problems might also occur when the drawing contains both short lines and long arcs. In this case, the minimum length requirement for a line has to be set relatively low. Then the arc might contain a segment that is long enough to qualify as a line and an error will happen.

A comparison between the results of the manual and automatic primitive extraction processes shows that the latter is much faster (as expected) but less accurate. It is obvious that the manual input process will take more than a few minutes and therefore as far as speed is

TABLE 3.1: Time analysis for Examples 4 & 5

CPU TIME (sec)					
EXAMPLE 4			EXAMPLE 5		
View1	View2	View3	View1	View2	View3
11.0	7.6	9.7	22.5	12.0	22.7

concerned there is no match. However the automatic process lacks the precision of the manual system which was demonstrated in the previous chapter. For instance in Example 4, the endpoints of the arc should have the same vertical (Y) coordinate. As it can be seen from the results, the values are different (145.0 and 148.0). The reason behind this lack of accuracy is the fact that the presentation of the line drawings was transformed from the original analog form into a digital form and therefore some error is introduced. This error justifies the introduction of an error tolerance as discussed in section 2.6.1. A consequence of this kind of error is that the final model might not be as accurate as before. However, given the size of the drawings and the final model, the problem is not very significant.

CHAPTER FOUR

CONCLUSION AND FUTURE RESEARCH

4.1 CONCLUSION

In this work, the problem of automatic CSG construction from line drawings was studied and two algorithms (i.e. input and construction) were proposed in an attempt to demonstrate the feasibility of such a system. The results from Chapter 2 show the feasibility of an algorithm that constructs the 3D model of an object from its 2D orthographic projections. However, certain assumptions have to be made about the characteristics of the object, that is, a certain class should be defined. Once the algorithm is designed, more complex examples can be used in order to upgrade the power of the routines and therefore expand the boundaries of the initial class.

The results obtained from the automatic extraction of primitives from the digitized line drawings were better than expected. This was possible partly because the digitization process was done in a relatively controlled environment. That is, thin line drawings were photographed against a uniform back light and the negatives were digitized using a high resolution scanner. As for the algorithm, it was designed to be able to treat digital straight lines even when they do not follow the rules listed in section 3.3.2.

The combination of the above algorithms enables us to achieve with some limitations, the goal set at the beginning of this report, which is the desire to transform the information available in the form of 3 orthographic view line drawings into a form directly compatible with an NC machine.

4.2 FUTURE RESEARCH

So far, the algorithms designed in order to construct a 3D model of an object from its 2D projections have been limited to certain classes of objects. There still remains the considerable challenge of designing a system general enough to handle any kind of objects. This ideal system would certainly require some limited interaction with a human operator in order to overcome the ambiguity caused by the complexity of certain line drawings.

REFERENCES

REFERENCES

[ALD1] Aldefeld,B., "On Automatic Recognition of 3D Structures from 2D Repesentations", *Computer Aided Design*, vol. 15, no. 2, pp59-64, MARCH 1983.

[ALD2] Aldefeld,B., "Automatic 3D Reconstruction from 2D Geometric Part Description", *Proc. Conf. on Computer Vision and Pattern Recognition*, JUNE 1983.

[FRE1] Freeman,H. and Davis,L.S., "A Corner Finding Algorithm for Chain-Coded Curves", *IEEE Trans. Comput.* 26, 1977, 297-303

[FRE2] Freeman,H., "Computer Processing of Line Drawing Images", *Computing Surveys*, vol.6, pp.57-97, Mar.1974.

[FRE3] Freeman,H., "On the Encoding of Arbitrary Geometric Configurations," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 260-268, June 1961.

[HAR] Haralick,R.M. and Queeney,D., "Understanding Engineering Drawings", *Computer Graphics and Image Processing*, vol.20, pp244-258, 1982

[LAF] Lafue,G., "A Theorem Prover for recognizing 2D Representations of 3D Objects," *Proc. IFIP TC-5 Working Conf. AI & CAD*, Grenoble, France, March 17-19, 1978, pp. 391-401.

[LIA] Liardet,M.,Holmes,C.& Rosenthal,D., "Input to CAD Systems:Two Practical Examples", *Proc. IFIP TC-5 Working Conf. AI & CAD*, Grenoble, France, March 17-19, 1978, pp. 403-414.

[LUZ] Luzadder,W., *Fundamentals of Engineering Drawing*, PRENTICE-HALL, Englewood Cliffs, N.J. 1965.

[PAUL] Paul,R.P., *Robot Manipulators, Mathematics, Programming and Control*, MIT Press, Cambridge, Mass. 1981.

[PR1] Preiss,K., "Algorithms for Automatic Conversion of 3-view Drawing of a Plane Faced Part to the 3D Representation",

Computer Industry, vol.12, pp 133-139, 1981.

[PR2] Preiss,K. and Kaplansky,E., "Solving CAD/CAM Problems by Heuristic Programming", *Computers in Mechanical Engineering*, Sep. 1983.

[REQ] Requicha,A.A.G., "Representations for Rigid Solids: Theory, Methods, and Systems", *Computing Surveys*, vol. 12, no. 4, Dec. 1980.

[ROS] Rosenfeld,A., "Digital Straight Line Segments", *IEEE Trans. Comput.* 23, 1974, 1264-1269.

[ROT] Roth,S.D., "Ray Casting for Modeling Solids", *Computer Graphics and Image Processing*, pp. 109-144, ACADEMIC PRESS, 1982.

[SAK] Sakurai,H. and Gossard,D., "Solid Model Input Through Orthographic Views", *Computer Graphics*, vol. 17, no. 3, July 1983.

[VOE] Requicha,A.A.G. and Voelcker,H.B., "An Introduction To Geometric Modeling and Its Applications In Mechanical Design and Production", *Advances in Information Systems Science*, ed. by Julius T. Tou, Vol. 8, PLENUM PRESS, New York, 1981.

[WEL] Wellman,B., *Technical Descriptive Geometry*, McGRAW-HILL, New York, 1957.

[WOO] Woo,T.C., "Progress in Shape Modeling", *Computer*, Dec. 1977.

APPENDIX

APPENDIX

The following pages contain the implementation of the algorithms explained in Chapters 2 and 3 in the "C" Language.

```
        /*   ALGORITHM 1   */

#include <stdio.h>
#include <math.h>

/*
    Define utility constants
*/

#define         X       1
#define         Y       2
#define         Z       3
#define         V1      1
#define         V2      2
#define         V3      3
#define         MAXSIZE         10
#define         TRUE    1
#define         FALSE           0
#define         BASE    1000
#define         LEG     15000

/*
    Define code for primitive type
*/

#define         LINE    1
#define         CIRCLE          2
#define         ARC     3
#define         CUBE    1
#define         CYLINDER 2
#define         CONE    3

/*
    Define code for drawing mode
*/

#define         DASHED  0
#define         SOLID   1

/*
    General representation of a two dimensional primitive
*/

    struct PRIM {
        int USE;
        int TYPE;
        int MODE;
        float POINT1[3];
        float POINT2[3];
        float CENTER[2];
        float LEN_RAD;
        struct PRIM *NEXT;
    };
```

```
/*
     General representation of a three dimensional primitive
*/

     struct PRIM3 {
          int TYPE;
          int NUM;
          int MODE;
          int MOVE;
          int FLAG;
          float ATR[3][4];
          struct PRIM3 *NEXT;
     };

/*
     Global variables
*/

     int T;
     FILE *fp,*fopen();
     struct PRIM *VIEW[4];
     struct PRIM3 *O_PTR,*L_PTR;

/*
     Functions returning non integers
*/

struct PRIM *getbase() ;
struct PRIM *gethead() ;
struct PRIM *ufun1() ;
struct PRIM *pmatch() ;
struct PRIM *match1() ;
struct PRIM *findline() ;
struct PRIM *do_line() ;
struct PRIM *do_arc() ;
struct PRIM *do_circle() ;
struct PRIM *talloc();
struct PRIM3 *getnode();
float angle();

/*********************************************
               MAIN()

In the main function, the list of primitives
is read from the file "in", and the 2D data
structure is setup. Then the various routines
are called and the results are printed out
*********************************************/
main()
{
     struct PRIM *old_ptr,*new_ptr;
     int p,v;

     fp = fopen("in","r");
```

```
    if (fp == NULL) exit(0);
    fscanf(fp,"%d",&T);

    for (v = 1;v < 4;v++){
        old_ptr=0;

        /* read primitive type and process accordingly */

        while(1){
            fscanf(fp,"%d",&p);
            if (p<0)
               break;
            switch(p){
                case LINE:
                  new_ptr = do_line(v);
                      break;

                case ARC:
                  new_ptr = do_arc(v);
                  break;

                case CIRCLE:
                  new_ptr = do_circle(v);
                  break;

                default:
                  printf("Input error0);
                  new_ptr = 0;
            } /* end of switch */
            if (old_ptr == 0){
                old_ptr = new_ptr;
                VIEW[v]=new_ptr;
            }
            else {
                old_ptr->NEXT = new_ptr;
                old_ptr = new_ptr,
            }
        } /* end of while(1) */

        if (old_ptr != 0)
        old_ptr->NEXT = 0;/* last primitive points to 0 */

    } /* end of for */
/* The 2d data structure has been initialized */
/* Start processing */
    for(v=1;v<4;v++){
        Cone(v);
    }
    for(v=1;v<4;v++)
        Cylinder(v);
    for(v=1;v<4;v++)
        Lug(v);

    /* process corners */
```

```
    Corner();
    /* process cubes */
    Cube();

/* output the final result */
    out_res();
} /* end of main */


/*********************************************
            DO_LINE()

This function creates a node using talloc()
and initializes it by asking the operator for
the different attributes of a LINE.
*********************************************/
struct PRIM *do_line(v)

 int v;
{
    struct PRIM *talloc(),*ptr;
    int i,j,m,n;
    float x1,y1,x2,y2,d,dum;

    i = v;  /* i and j are the coordinate system */
    j = (v % 3) + 1;

    ptr = talloc();     /* get new node */

    /* Initialization of the node */

        ptr->USE = 0; ptr->TYPE = 1;
        fscanf(fp,"%d",&n);
        ptr->MODE = n;
        fscanf(fp,"%f",&x1); fscanf(fp,"%f",&y1);
        fscanf(fp,"%f",&x2); fscanf(fp,"%f",&y2);
    /* enter the points in order */
    if (((x1-x2) > T)  || ((y1-y2) > T )) {
        dum = x1; x1 = x2;
        x2 = dum; dum = y1;
        y1 = y2; y2 = dum;
    }
        ptr->POINT1[1] = x1; ptr->POINT1[2] = y1;
        ptr->POINT2[1] = x2; ptr->POINT2[2] = y2;

    /* calculate and enter the length */

        d = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
        ptr->LEN_RAD = sqrt((double) d);

    return(ptr);
}
/*********************************************
            DO_CIRCLE()
```

```
This function creates a node using talloc()
and initializes it by asking the operator for
the different attributes of a CIRCLE.
*********************************************/
struct PRIM *do_circle(v)

 int v;
{
     struct PRIM *talloc(),*ptr;
     int i,j,m;
     float x1,y1,rad;

     i = v;  /* i and j are the coordinate system */
     j = (v % 3) + 1;

     ptr = talloc();    /* get new node */

     /* Initialization of the node */

          ptr->USE = 0; ptr->TYPE = 2;
          fscanf(fp,"%d",&m); ptr->MODE = m;
          fscanf(fp,"%f",&x1); fscanf(fp,"%f",&y1);
          ptr->CENTER[0] = x1; ptr->CENTER[1] = y1;
          fscanf(fp,"%f",&rad);
          ptr->LEN_RAD = rad;
     return(ptr);
}
/*********************************************
               DO_ARC()

This function creates a node using talloc()
and initializes it by asking the operator for
the different attributes of an ARC.
*********************************************/
struct PRIM *do_arc(v)

 int v;
{
     struct PRIM *talloc(),*ptr;
     int i,j,m,n;
     float x1,y1,x2,y2,rad,dum;

     i = v;  /* i and j are the coordinate system */
     j = (v % 3) + 1;

     ptr = talloc();     /* get new node */

     /* Initialization of the node */

          ptr->USE = 0; ptr->TYPE = 3;
          fscanf(fp,"%d",&m); ptr->MODE = m;
          fscanf(fp,"%f",&x1); fscanf(fp,"%f",&y1);
          fscanf(fp,"%f",&x2); fscanf(fp,"%f",&y2);
          fscanf(fp,"%d",&n);
```

```
        ptr->POINT1[0] = n; ptr->POINT2[0] = n;
        /* enter the points in order */
        if (x1 > x2  || y1 > y2 ) {
            dum = x1; x1 = x2;
            x2 = dum; dum = y1;
            y1 = y2; y2 = dum;
        }
        ptr->POINT1[1] = x1; ptr->POINT1[2] = y1;
        ptr->POINT2[1] = x2; ptr->POINT2[2] = y2;
        fscanf(fp,"%f",&x1); fscanf(fp,"%f",&y1);
        ptr->CENTER[0] = x1; ptr->CENTER[1] = y1;

    /* calculate the radius of the arc */
        rad = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
        ptr->LEN_RAD = sqrt((double) rad);

    return(ptr);
}
/*********************************************
            TALLOC()

This function allocates storage for 2d primitives
*********************************************/
struct PRIM *talloc()
{
    char *calloc();

    return((struct PRIM *) calloc(1,sizeof(struct PRIM)));
}
/*********************************************
            GETNODE()

This function allocates storage for 3d primitives
*********************************************/
struct PRIM3 *getnode()
{
    char *calloc();
    struct PRIM3 *ptr;
    static int count = 1;

    ptr = (struct PRIM3 *) calloc(1,sizeof(struct PRIM3));
/* Initialize the output pointer if this is the first access */
    if (count == 1){
        O_PTR = ptr;
        O_PTR->TYPE = 0;
        ptr = (struct PRIM3 *) calloc(1,sizeof(struct PRIM3));
        O_PTR->NEXT = ptr;
        L_PTR = ptr;
    }
    count++;
    return(ptr);
}
/*********************************************
        EQ()
```

```
Function Equal. T is the error margin (Threshold).
**********************************************/
EQ(x,y)

float x,y;
{

     if(fabs(x - y) < T)
          return(1);
     return(0);
}
/*********************************************
          NEQ()

Function Not Equal
**********************************************/
NEQ(x,y)

float x,y;
{

     if(fabs(x - y) > T)
          return(1);
     return(0);
}
/*********************************************
          UFUN3()

Find two lines that share the given coordinates
If they have the same slope,merge them together.
**********************************************/
ufun3(co1,co2,view)

 float co1,co2;
 int view;
{
     int sco,dco;
     float co11,co12,co21,co22;
     struct PRIM *ptr,*dum,*line1,*line2,*nptr;

     ptr = 0;
     while((line1=findline(view,co1,co2,ptr))!=NULL){
       ptr = line1;
       if (line1->USE == 0){

         while((line2=findline(view,co1,co2,ptr))!=NULL){
           ptr = line2;
           if (line2->USE == 0){

          co11 = line1->POINT1[1];
          co12 = line1->POINT1[2];
          co21 = line1->POINT2[1];
          co22 = line1->POINT2[2];
```

```
            if(EQ(co11,co21)) sco = 1;
            else
               if(EQ(co12,co22)) sco = 2;
            else
               return(0);

            if(NEQ(line2->POINT1[sco],line2->POINT2[sco]))  return(0);
            dco = (sco == 1) ? 2 : 1;
            if(EQ(line1->POINT1[dco],line2->POINT2[dco])){
                 co11 = line2->POINT1[1];
                 co12 = line2->POINT1[2];
                 co21 = line1->POINT2[1];
                 co22 = line1->POINT2[2];
            }
            else{
                 co21 = line2->POINT2[1];
                 co22 = line2->POINT2[2];
            }
          /* The lines have the same slope.
             Create a third line that can replace the above two
           */

            nptr = talloc();
            nptr->USE = 0;
            nptr->TYPE = LINE;
            nptr->MODE = SOLID;
            nptr->POINT1[1] = co11;
            nptr->POINT2[1] = co21;
            nptr->POINT1[2] = co12;
            nptr->POINT2[2] = co22;
            nptr->LEN_RAD = line1->LEN_RAD + line2->LEN_RAD;

            dum = VIEW[view];
            VIEW[view] = nptr;
            nptr->NEXT = dum;

         /* Delete the original two lines */

            delete(line1,view); delete(line2,view);
            return(1);
             }/* end of second if */
           }/* end of second while */
         }/* end of first if */
       }/* end of first while */
       return(0);
}
/*********************************************
         OUT_CUBE()

Add a cube to the 3d data structure
*********************************************/
out_cube(buf,sign,flag,move)

 float buf[3][4];
```

```
 int sign,flag,move;
{
     static int i = 1;
     int j,k;
     struct PRIM3 *ptr;

     ptr = getnode(); ptr->TYPE = CUBE;
     ptr->NUM = i++; ptr->MODE = sign;
     ptr->MOVE = move; ptr->FLAG = flag;

     for(j = 0;j < 3;j++){
          for(k = 0;k < 4;k++)
               ptr->ATR[j][k] = buf[j][k];
     }
     L_PTR->NEXT = ptr;
     L_PTR = ptr;
     ptr->NEXT = 0;
     return;
}
/*********************************************
          OUT_CYL()

Add a cylinder to the 3d data structure
*********************************************/
out_cyl(buf,sign,flag)

 float buf[3][4];
 int sign,flag;
{
     static int i = 1;
     int j,k;
     struct PRIM3 *ptr;

     ptr = getnode(); ptr->TYPE = CYLINDER;
     ptr->NUM = i++; ptr->MODE = sign;
     ptr->FLAG = flag;

     for(j = 0;j < 3;j++){
          for(k = 0;k < 4;k++)
               ptr->ATR[j][k] = buf[j][k];
     }
     L_PTR->NEXT = ptr;
     L_PTR = ptr;
     ptr->NEXT = 0;
     return;
}
/*********************************************
          DELETE()

Remove the given element from the input
data structure.
*********************************************/
delete(ptr,view)
```

```
struct PRIM *ptr;
int view;
{
     struct PRIM *ptr1,*ptr2;

     ptr1 = VIEW[view];
     if (ptr1 == ptr) {
          VIEW[view] == ptr->NEXT;
          return;
     }
     while((ptr2 = ptr1->NEXT) != NULL){
          if (ptr2 == ptr){
               ptr1->NEXT = ptr->NEXT;
               return;
          }
          ptr1 = ptr2;
     }
}
/*********************************************
          OUT2()

Outputs the three cubes needed to represent
a slanted cube.
*********************************************/
out2(leg1,leg2,ptr,view,pos)

 struct PRIM *leg1,*leg2,*ptr;
 int view,pos;
{
     float cx,cy,cz,xl,yl,zl,max,teta;
     float buf1[3][4],buf2[3][4],buf3[3][4];
     float len,tco;
     float co1max,co1min,co2max,co2min,dum;
     int i,j,

     for(i=0;i<3;i++){
          for(j=0;j<4;j++){
               buf1[i][j] = 0.0;
               buf2[i][j] = 0.0;
               buf3[i][j] = 0.0;
          }
     }
     teta = angle(leg1,view);
/* get the third coordinate and length */
     len = ptr->LEN_RAD;
     j = (EQ(ptr->POINT1[1],ptr->POINT2[1])) ? 2 : 1;
     tco = ptr->POINT1[j];

/* get the maximum and minimum coordinates */

     co1min = (leg1->POINT1[1] < leg1->POINT2[1]) ?
          leg1->POINT1[1] : leg1->POINT2[1];
     dum = (leg2->POINT1[1] < leg2->POINT2[1]) ?
          leg2->POINT1[1] : leg2->POINT2[1] ;
```

```
    co1min = (dum < co1min) ? dum : co1min;
    co1max = (leg1->POINT1[1] > leg1->POINT2[1]) ?
         leg1->POINT1[1] : leg1->POINT2[1];
    dum = (leg2->POINT1[1] > leg2->POINT2[1]) ?
         leg2->POINT1[1] : leg2->POINT2[1] ;
    co1max = (dum > co1max) ? dum : co1max;
    co2min = (leg1->POINT1[2] < leg1->POINT2[2]) ?
         leg1->POINT1[2] : leg1->POINT2[2];
    dum = (leg2->POINT1[2] < leg2->POINT2[2]) ?
         leg2->POINT1[2] : leg2->POINT2[2] ;
    co2min = (dum < co2min) ? dum : co2min;
    co2max = (leg1->POINT1[2] > leg1->POINT2[2]) ?
         leg1->POINT1[2] : leg1->POINT2[2];
    dum = (leg2->POINT1[2] > leg2->POINT2[2]) ?
         leg2->POINT1[2] : leg2->POINT2[2] ;
    co2max = (dum > co2max) ? dum : co2max;

    /* Depending on the view, use the above
       information to find the attributes for the
       Cubes.
    /*

    if (view == 1){
         switch(pos) {
              case 1:
                   buf2[2][3] = teta;
                   buf3[0][2] = co1max - co1min;
                   buf3[2][3] = 90 - teta;
                   break;
              case 3:
                   buf2[1][2] = co2max - co2min;
                   buf2[2][3] = - teta;
                   buf3[2][3] = 90 - teta;
                   break;
              case 4:
                   buf2[0][2] = co1max -co1min;
                   buf2[1][2] = co2max - co2min;
                   buf2[2][3] = 90 + teta;
                   buf3[0][2] = co1max - co1min;
                   buf3[2][3] = 180 - teta;
                   break;
         }

         cx = co1min; xl = co1max - co1min;
         cy = co2min; yl = co2max - co2min;
         cz = tco;    zl = len;
    }
    else if (view == 2) {
         switch(pos) {
              case 1:
                   buf2[0][3] = - teta;
                   buf3[2][2] = co2max - co2min;
                   buf3[0][3] = teta - 90;
                   break;
```

```
            case 3:
                buf3[1][2] = col1max - col1min;
                buf3[0][3] = teta;
                buf2[0][3] = 90 - teta;
                break;
            case 4:
                buf2[2][2] = co2max -co2min;
                buf2[1][2] = col1max - col1min;
                buf2[0][3] = - (90 + teta);
                buf3[2][2] = co2max - co2min;
                buf3[0][3] = - (180 - teta);
                break;
        }

        cy = col1min; yl = col1max - col1min;
        cz = co2min; zl = co2max - co2min;
        cx = tco;    xl = len;
    }
    else if (view == 3) {
        switch(pos) {
            case 1:
                buf2[1][3] = teta;
                buf3[0][2] = co2max - co2min;
                buf3[1][2] = 90 - teta;
                break;
            case 3:
                buf3[2][2] = col1max - col1min;
                buf3[1][3] = teta;
                buf2[1][3] = 90 - teta;
                break;
            case 4:
                buf2[0][2] = co2max -co2min;
                buf2[1][3] = 180 + teta;
                buf3[0][2] = co2max - co2min;
                buf3[2][2] = col1max - col1min;
                buf3[1][3] = - (90 + teta);
                break;
        }

        cz = col1min; zl = col1max - col1min;
        cx = co2min; xl = co2max - co2min;
        cy = tco;    yl = len;
    }
    max = (xl > yl) ? xl : yl;
    max = (max > zl) ? max : zl;

    buf1[0][0] = cx; buf1[1][0] = cy; buf1[2][0] = cz;
    buf1[0][1] = xl; buf1[1][1] = yl; buf1[2][1] = zl;

    /* output the middle cube */

    out_cube(buf1,1,-2,1);

    /* The dimensions of the slanted cube are not
```

```
        important as long as they cover the right area
      /*

      buf2[0][1] = 2*max; buf2[1][1] = 2*max;
      buf2[2][1] = 2*max; buf3[0][1] = 2*max;
      buf3[1][1] = 2*max; buf3[2][1] = 2*max;

      /* Output the slanted cubes */

      out_cube(buf2,-1,1,0);
      out_cube(buf3,-1,1,0);

      return;
}
/*********************************************
          ANGLE()

Find the angle that the line makes with the
horizontal. Teta will be between 0 and 90
degrees.
*********************************************/
float angle(line,view)

 struct PRIM *line;
 int view;
{
     int ver,hor;
     float delx,dely,dum;

     switch(view){
          case V1:
               ver = 2; hor = 1; break;
          default:
               ver = 1; hor = 2; break;
     }
     dely = fabs(line->POINT1[ver] - line->POINT2[ver]);
     delx = fabs(line->POINT1[hor] - line->POINT2[hor]);

     if (delx == 0) return(90.0);
     else{
          dum = atan((double) (dely/delx));
          dum = dum * 180.0/3.1415;
          return(dum);
     }
}
/*********************************************
              OUT_RES()

Reorganize the 3d data structure if necessary
and output the results
*********************************************/
out_res()
{
     struct PRIM3 *ptr1,*ptr2,*dum,*s_ptr1,*s_ptr2,*s_ptr3,*s_ptr4;
```

```
    int i,count,j,k;
    float xm,ym,zm;

    printf(" THE RESULT  0);
    fp = fopen("res","w");

/* Send all single cavity cylinders and cubes to the end of the list */

    ptr2 = O_PTR;
    count = 0;
    dum = 0;
    while ((ptr1 = (ptr2->NEXT)) != NULL){
        if ((ptr1->TYPE == CYLINDER || ptr1->TYPE == CUBE)&&
                ptr1->MODE == -1 &&
                ptr1->FLAG == 0){
            if(ptr1 == dum) break;
            L_PTR->NEXT = ptr1;
            L_PTR = ptr1;
            ptr2->NEXT = ptr1->NEXT;
            ptr1->NEXT = 0,
            dum = (count++) ? dum : ptr1;
        }
        else
            ptr2 = ptr1;
    }

    /*  See if any Cubes or Cylinders have to be updated */

    ptr1 = O_PTR;
    while ((ptr2 = ptr1->NEXT) != NULL){
        if(ptr2->TYPE == 0 && ptr2->FLAG > 40 && ptr2->FLAG < 50)
            break;
        ptr1 = ptr2;
    }
    if ( ptr2 != NULL ){
        s_ptr1 = ptr1;
        s_ptr2 = ptr2;
        ptr1 = ptr2;
        while ((ptr2 = ptr1->NEXT) != NULL){
            if(ptr2->TYPE == CUBE && ptr2->FLAG == s_ptr2->FLAG){
                s_ptr3 = ptr1;
                s_ptr4 = ptr2;
                break;
            }
            ptr1 = ptr2;
        }
        if ( ptr2 == NULL ) {
            fprintf(stderr,"Error in OUT_RES");
            exit(1);
        }
        s_ptr1->NEXT = s_ptr4;
        s_ptr3->NEXT = s_ptr4->NEXT;
        s_ptr4->NEXT = s_ptr2->NEXT;
        s_ptr4->FLAG = s_ptr4->FLAG - 50;
```

```
/* update the related cube(s) and cylinder(s) */

        ptr1 = s_ptr4;
        s_ptr1 = ptr1;
        for(j = ptr1->FLAG; j<0; j++){
             ptr1 = ptr1->NEXT;
             ptr1->ATR[2][1] = s_ptr1->ATR[2][1];
             ptr1->ATR[2][0] = s_ptr1->ATR[2][0];
        }
    }

    /* Start printing the 3D primitves out */

    ptr1 = O_PTR;
    printf("0);
    count = 0;
    while ((ptr1 = ptr1->NEXT) != NULL){
       if (ptr1->TYPE != NULL){
        count++;
        if (ptr1->MODE < 0)
             printf(" - ");
        else if (ptr1->MODE > 0)
             printf(" + ");
        if (ptr1->FLAG > 10 && ptr1->TYPE == CUBE)
             ptr1->FLAG = ptr1->FLAG - 50;
        if (ptr1->MOVE == 1){
             xm = ptr1->ATR[0][0];
             ym = ptr1->ATR[1][0];
             zm = ptr1->ATR[2][0];
             for (j=0; j<3; j++)
                  ptr1->ATR[j][0] = 0;
             printf(" MOVE (%5.2f,%5.2f,%5.2f)", xm, ym, zm);
        }
        for ((i = ptr1->FLAG); i<0; i++)
                  printf("(");

        switch(ptr1->TYPE) {
             case CUBE:
                  printf(" CUBE");
                  fprintf(fp,"0CUBE");
                  break;
             case CYLINDER:
                  printf(" CYLINDER");
                  fprintf(fp,"0CYLINDER");
                  break;
             case CONE:
                  printf(" CONE");
                  fprintf(fp,"0CONE");
                  break;
             default:
                  printf(" primitive unknown0);
        }
        printf("%d",ptr1->NUM);
        fprintf(fp,"%d0,ptr1->NUM);
```

```
            for(i = ptr1->FLAG;i > 0;i--)
                    printf(")");
            if (count == 5){
                 printf("0);
                 count = 0;
            }
            for(j=0;j<3;j++){
                 for(k=0;k<4;k++)
                      fprintf(fp,"%10.3f",ptr1->ATR[j][k]);
                 fprintf(fp,"0);
            }
           }
      }
      printf("0);
}
/*********************************************
          CORNER

Find all the curved corners and save the
information necessary to rebuild this part
of the object with a combination of cubes
and cylinders.
*********************************************/
Corner()
{
      struct PRIM *buff[MAXSIZE],*ptr1,*ptr2,*dum,*tmp;
      int count,i,j,k,flag;
      float p11,p12,p21,p22,rad,tem;
      float teta1,teta2,cube[3][4],cyl[3][4];

      /* Initialize the buffers to zero */
      for(i=0;i<3;i++){
           for(j=0;j<4;j++){
                cube[i][j] = 0;
                cyl[i][j] = 0;
           }
      }
      /* Get all the curves */
      if ((count = get_curve(V1,buff,ARC)) == NULL) return;
      for (k=0;k<count;k++){
        tmp = buff[k];
        rad = tmp->LEN_RAD;
        p11 = tmp->POINT1[1];
        p12 = tmp->POINT1[2];
        p21 = tmp->POINT2[1];
        p22 = tmp->POINT2[2];
        if(tmp->POINT1[0] > 5){

     /* Find the lines connected to the curve. */

        dum = 0;
        ptr1 = findline(V1,p11,p12,dum);
        if (ptr1 == NULL)
           fprintf(stderr,"Error in corner,ptr10);
```

```
    ptr2 = findline(V1,p21,p22,dum);
    if (ptr2 == NULL)
       fprintf(stderr,"Error in corner,ptr20);

  /* Make sure the lines are perpendicular */

    teta1 = angle(ptr1,V1);
    teta2 = angle(ptr2,V1);
    teta1 = fabs(teta1 - teta2);
    while(fabs(teta1 - 90.0) < 5.0){
       /* Initialize the buffers to zero */
       for(i=0;i<3;i++){
            for(j=0;j<4;j++){
                 cube[i][j] = 0;
                 cyl[i][j] = 0;
            }
       }
       if(NEQ(ptr1->POINT1[2],ptr1->POINT2[2])){
            tmp = ptr1; ptr1 = ptr2; ptr2 = tmp;

            tem = p11; p11 = p21; p21 = tem;

            tem = p12; p12 = p22; p22 = tem;
       }

  /* Using the p's, find the location of the corner */
  /* Then, find the correct attributes for the Cube
     and Cylinder */

       if (p11 < p21){
            if(p12 > p22){
                 cube[0][0] = p11; cube[1][0] = p22;
                 cyl[0][0] = p11; cyl[1][0] = p22;
                 flag = 1;
            }
            else{
                 cube[0][0] = p11; cube[1][0] = p12;
                 cyl[0][0] = p11; cyl[1][0] = p12 + rad;
                 flag = 2;
            }
       }
       else if(p11 > p21){
            if (p12 > p22){
                 cube[0][0] = p21; cube[1][0] = p22;
                 cyl[0][0] = p21 + rad; cyl[1][0] = p22;
                 flag = 3;
            }
            else{
                 cube[0][0] = p21; cube[1][0] = p12;
                 cyl[0][0] = p21 + rad;cyl[1][0] = p12 + rad;
                 flag = 4;
            }
       }
       cube[0][1] = rad;
```

```
            cube[1][1] = rad;
            cyl[0][2] = rad;

            switch(flag) {
                 case 1:
                      ptr1->POINT2[1] = p21;
                      ptr2->POINT2[2] = p12;
                      break;
                 case 2:
                      ptr1->POINT2[1] = p21;
                      ptr2->POINT1[2] = p12;
                      break;
                 case 3:
                      ptr1->POINT1[1] = p21;
                      ptr2->POINT2[2] = p12;
                      break;
                 case 4:
                      ptr1->POINT1[1] = p21;
                      ptr2->POINT1[2] = p12;
                      break;
            }
            ptr1->LEN_RAD = ptr1->LEN_RAD + rad;
            ptr2->LEN_RAD = ptr2->LEN_RAD + rad;

        /* Adjust the flag for cube and cylinder so that it would
           be possible to find them again */

            flag = ((int) ptr1) + ((int) ptr2);

        /* Output the nodes */

            out_cube(cube,-1,flag,0);
            out_cyl(cyl,1,flag);
            break;
          }/* end of while */
          }/* end of if(corner...)*/
        }/* end of for */
}
/*************************************************
          UFUN4()

See if any nodes in the 3D data structure
has to be modified. If so, return the appropriate
number for flag
*************************************************/
ufun4(base,leg1,leg2,head)

 struct PRIM *base,*leg1,*leg2,*head;
{
     int s1,s2,s3,s4,flag,dum,found;
     struct PRIM3 *ptr,*ptr1,*ptr2,*s_ptr1,*s_ptr2;


     s1 = ((int) base) + ((int) leg1);
```

```
        s2 = ((int) base) + ((int) leg2);
        s3 = ((int) head) + ((int) leg1);
        s4 = ((int) head) + ((int) leg2);

        ptr1 = 0_PTR;
        found = 0;
        flag = 0;
        while ((ptr2 = ptr1->NEXT) != NULL){
             dum  = ptr2->FLAG;
             if(dum == s1 || dum == s2 || dum == s3 || dum == s4){
                  if (flag-- == 0){
                       s_ptr1 = ptr1;
                       s_ptr2 = ptr2;
                  }
                  ptr2->FLAG = 1;
                  found++;
             }
             ptr1 = ptr2;
        }
        if (found == 0) return(0);
        ptr = getnode();
        ptr->TYPE = 0;
        ptr->FLAG = flag + 50;
        s_ptr1->NEXT = ptr;
        ptr->NEXT = s_ptr2;
        return(flag + 50);
}
/*********************************************
          CONE()

Find and output all the cones.
*********************************************/
Cone(view)
{
     int count,i,j,nview,cord,mco,found1;
     float co11,co12,co21,co22,max,min,max1,max2,min1,min2;
     float teta1,teta2,h;
     struct PRIM *arm1,*arm2,*circle[10];
     struct PRIM *c1,*c2,*cand,*top;

     switch(view) {
          case V1:
               nview = V3; cord = Z; mco = X;
               break;
          case V2:
               nview = V1; cord = X; mco = Y;
               break;
          case V3:
               nview = V1; cord = Y; mco = X;
               break;
     }

     /* Find all the Circles in this view */
```

```
count = get_curve(view,circle,CIRCLE);
for(i = 0;i < count - 1;i++){
  c1 = circle[i];
  for(j = i+1;j < count;j++){
    c2 = circle[j];
    if (EQ(c1->CENTER[0],c2->CENTER[0]) &&
              EQ(c1->CENTER[1],c2->CENTER[1])){

     max_co(c1,view,mco,&max1,&min1);
     max_co(c2,view,mco,&max2,&min2);

/* find the lines representing the surfaces
   of the circles */

     max = (max1 > max2) ? max1 : max2;
     min = (min1 < min2) ? min1 : min2;
     cand = 0;
     while ((cand = getbase(nview,cord,cand)) != NULL ){
      if (EQ(cand->POINT1[mco],min)
          && EQ(cand->POINT2[mco],max)){
          co11 = cand->POINT1[1];
          co12 = cand->POINT1[2];
          co21 = cand->POINT2[1];
          co22 = cand->POINT2[2];
          arm1 = 0; found1 = FALSE;

/* Find the arms */

          while ((arm1 = findline(nview,co11,co12,
              arm1)) != NULL){
              teta1=angle(arm1,nview);
              if((teta1 - 0.0) > 5.0 && (90.0 - teta1) > 5.0){
                arm2 = 0;
                while ((arm2 = findline(nview,co21,co22,arm2)) != NULL){
                   teta2 = angle(arm2,nview);
                   if(fabs(teta1-teta2)<5.0
                        &&
                    EQ(arm1->LEN_RAD,arm2->LEN_RAD)){
                        found1 = TRUE;
                         break;
                   }
                }/* end of while(arm2....)*/

                if(found1 == TRUE) break;

              }/* end of if teta */
          }/* end of while(arm=...)*/
          if(found1 == TRUE) break;
      }/* end of if(cand=...)*/
     }/* end of while(cand=...)*/
     if(found1 == TRUE){

/* The arms are found */
```

```
                max = (max1 > max2) ? max2 : max1;
                min = (min1 < min2) ? min2 : min1;
                top = 0;
                while ((top = getbase(nview,cord,top)) != NULL){
                  if (EQ(top->POINT1[mco],min)
                   && EQ(top->POINT2[mco],max))
                       break;
                }
                if(top != NULL){
                    h = (cand->LEN_RAD)*(tan(teta1 * 3.1415 /180.0));
                    h = h/2;

    /* Output a Cone */

                    out4(c1,c2,cand,arm1,h,view);
                    arm1->USE = LEG; arm2->USE = LEG;
                }
            }
           }/* end of if (center1 = center2)*/
         }/* end of for(j)*/
       }/* end of for(i)*/
}
/*************************************************
        OUT4()

 Prepare the information needed for out_cone.
*************************************************/
out4(c1,c2,base,arm1,h,view)

 struct PRIM *c1, *c2,*base,*arm1;
 float h;
 int view;
{
    int i,j;
    struct PRIM *tmp;
    float buf1[3][4],buf2[3][4],dum1,dum2;

    for (i = 0;i < 3;i++){
        for (j = 0;j < 4;j++){
            buf1[i][j] = 0;
            buf2[i][j] = 0;
        }
    }
    if (c1->LEN_RAD < c2->LEN_RAD) {
        tmp = c1; c1 = c2; c2 = tmp;
    }
    buf1[0][2] = c1->LEN_RAD;
    buf2[0][2] = c2->LEN_RAD;
    dum1 = (EQ(arm1->POINT1[1],base->POINT1[1])) ?
        arm1->POINT2[1] : arm1->POINT1[1];
    dum2 = (EQ(arm1->POINT1[2],base->POINT1[2])) ?
        arm1->POINT2[2] : arm1->POINT1[2];

    /* Depending on the view, find the right attributes
```

```
    for the Cone using the above info.
  */

  switch(view) {
      case V1:
           buf1[0][0] = c1->CENTER[0];
           buf1[1][0] = c1->CENTER[1];
           buf1[2][0] = base->POINT1[1];
           buf1[2][1] = (base->POINT1[1] < dum1) ?
                    h : -h;
           buf2[0][0] = c2->CENTER[0];
           buf2[1][0] = c2->CENTER[1];
           buf2[2][0] = dum1;
           buf2[2][1] = buf1[2][1];
           break;
      case V2:
           buf1[1][0] = c1->CENTER[0];
           buf1[2][0] = c1->CENTER[1];
           buf1[0][0] = base->POINT1[1];
           buf1[0][1] = (base->POINT1[1] < dum1) ?
                    h : -h;
           buf2[1][0] = c2->CENTER[0];
           buf2[2][0] = c2->CENTER[1];
           buf2[0][0] = dum1;
           buf2[0][1] = buf1[0][1];
           break;
      case V3:
           buf1[2][0] = c1->CENTER[0];
           buf1[0][0] = c1->CENTER[1];
           buf1[1][0] = base->POINT1[2];
           buf1[1][1] = (base->POINT1[2] < dum2) ?
                    h : -h;
           buf2[2][0] = c2->CENTER[0];
           buf2[0][0] = c2->CENTER[1];
           buf2[1][0] = dum2;
           buf2[1][1] = buf1[1][1];
           break;
  }
  out_cone(buf1,1,-1);

  /* Output the cylinder to be extracted from
     the Cone.
  */

  out_cyl(buf2,-1,1);
}
/*********************************************
        OUT_CONE()

Add a cone to the 3d data structure
*********************************************/
out_cone(buf,sign,flag)

float buf[3][4];
```

```
 int sign,flag;
{
     static int i = 1;
     int j,k;
     struct PRIM3 *ptr;

     ptr = getnode();
     ptr->TYPE = CONE;
     ptr->NUM = i++;
     ptr->MODE = sign;
     ptr->FLAG = flag;

     for(j = 0;j < 3;j++){
          for(k = 0;k < 4;k++)
               ptr->ATR[j][k] = buf[j][k];
     }
     L_PTR->NEXT = ptr;
     L_PTR = ptr;
     ptr->NEXT = 0;
     return;
}
/***********************************************
          GET_CURVE()

Find all curves of type "type" in VIEW[view_num]
and store pointers to them in array buff.
Return the number of curves found.
***********************************************/
get_curve(view_num,buff,type)

 int view_num,type;
 struct PRIM *buff[MAXSIZE];
{
     struct PRIM *ptr;
     int count;

     /* initialize counter and pointer */

     count = 0;
     ptr = VIEW[view_num];

     /* go through the list of primitives and find the curves */

     while(ptr != NULL) {

          if ((ptr->TYPE) == type) {
               buff[count++] = ptr;
          }
          ptr = ptr->NEXT;
     }
     return(count);
}
/***********************************************
          MAX_CO()
```

```
Find the extreme points on the circle pointed
to by ptr. Co_num specifies the coordinate
on which the points are to be found.
**********************************************/
max_co(ptr,view_num,co_num,max,min)

 struct PRIM *ptr;
 int co_num,view_num;
 float *max,*min;
{
     int i;
     float radius;

     radius = ptr->LEN_RAD;

     /* Find out which coordinates we are trying to match */

     if (co_num == view_num )
          i=0;
     else if (co_num == ((view_num % 3) + 1))
          i=1;
     else {    /* The given coordinate is not in this view */
          printf(" error in max_co0);
          return(0);
     }
     *max = (ptr->CENTER[i]) + radius;
     *min = (ptr->CENTER[i]) - radius;
     return;
}
/**********************************************
          CYLINDER()

Find and output all the Cylinders.
**********************************************/
Cylinder(view)

 int view;
{
     struct PRIM *circle[MAXSIZE],*ptr,*pptr,*match1();
     struct PRIM *v2_arm1[5],*v2_arm2[5],*v3_arm1[5],*v3_arm2[5];
     int v1,v2,co1,co2,i,i1,i2,i3,i4,count;
     float max1,min1,max2,min2;

     /* find the appropriate views and cooordinates for matching */

     if (view == 1){
          v1 = 3; v2 = 2; co1 = 1; co2 = 2;
     }

     else if (view == 2){
          v1 = 1; v2 = 3; co1 = 2; co2 = 3;
     }

     else{
```

```
        v1 = 2; v2 = 1; co1 = 3; co2 = 1;
    }

    /* find all the circles in this view */

    count = get_curve(view,circle,CIRCLE);

    /* process each circle */

    for(i=0;i<count;i++){

        max_co(circle[i],view,co1,&max1,&min1);
        max_co(circle[i],view,co2,&max2,&min2);

        /* try to find the matching lines */
            i1 = 0;
            pptr = 0;
            while((v2_arm1[i1]=match1(v1,co1,min1,pptr)) != NULL){
                pptr = v2_arm1[i1++];
            }
            i2 = 0;
            pptr = 0;
            while((v2_arm2[i2]=match1(v1,co1,max1,pptr)) != NULL){
                pptr = v2_arm2[i2++];
            }
            i3 = 0;
            pptr = 0;
            while((v3_arm1[i3]=match1(v2,co2,min2,pptr)) !=NULL){
                pptr = v3_arm1[i3++];
            }
            i4 = 0;
            pptr = 0;
            while((v3_arm2[i4]=match1(v2,co2,max2,pptr)) !=NULL){
                pptr = v3_arm2[i4++];
            }

    /* Depending on the number of lines found in each case,
       find the best candidate for the third dimension of
       Cylinder
    */

            while(i2!=0 && i1!=0 && i3!=0 && i4!=0){

                if(i2==1 && i1==1 && i3==1 && i4==1){
                  if(match_lines(v2_arm1[0],v2_arm2[0]) == TRUE){
                     out3(circle[i],v2_arm1[0],view,0);
                     break;
                  }
                  if(match_lines(v3_arm1[0],v3_arm2[0]) == TRUE){
                     out3(circle[i],v3_arm1[0],view,0);
                     break;
                  }
                  if ((ptr=ufun1(v2_arm1[0],v2_arm2[0],V2,Y)) != NULL){
                     out3(circle[i],ptr,view,0);
```

```
                    break;
                  }
                  if ((ptr=ufun1(v3_arm1[0],v3_arm2[0],V3,X)) != NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                  if ((ptr=pmatch(v2_arm1[0],v2_arm2[0],V2,Z)) != NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                  if ((ptr=pmatch(v3_arm1[0],v3_arm2[0],V3,Z)) != NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                }/* end of if(all of them = 1) */

/* If only two lines are found in the first view
   then choose between them
*/

                else if(i2==1 && i1==1){
                  if(match_lines(v2_arm1[0],v2_arm2[0]) == TRUE){
                    out3(circle[i],v2_arm1[0],view,0);
                    break;
                  }
                  if((ptr=ufun1(v2_arm1[0],v2_arm2[0],V2,Y))!= NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                  if((ptr=pmatch(v2_arm1[0],v2_arm2[0],V2,Z))!= NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                  else{
                    fprintf(stderr,"impossible interpretation0);
                    exit(1);
                  }
                }/* end of if(i2,i1 = 1) */

/* If only two lines are found in the second view
   then choose between them
*/

                else if(i3==1 && i4==1){
                  if(match_lines(v3_arm1[0],v3_arm2[0]) == TRUE){
                    out3(circle[i],v3_arm1[0],view,0);
                    break;
                  }
                  if((ptr=ufun1(v3_arm1[0],v3_arm2[0],V3,X))!= NULL){
                    out3(circle[i],ptr,view,0);
                    break;
                  }
                  if((ptr=pmatch(v3_arm1[0],v3_arm2[0],V3,Z))!= NULL){
                    out3(circle[i],ptr,view,0);
```

```
                    break;
                }
                else{
                    fprintf(stderr,"impossible interpretation0);
                    exit(1);
                }

            }/* end of if(i3,i4 = 1)*/

            else if(i1==1){
                out3(circle[i],v2_arm1[0],view,0);
                break;
            }
            else if(i2==1){
                out3(circle[i],v2_arm2[0],view,0);
                break;
            }
            else if(i3==1){
                out3(circle[i],v3_arm1[0],view,0);
                break;
            }
            else if(i4==1){
                out3(circle[i],v3_arm2[0],view,0);
                break;
            }
            else
                break;
        }/* end of while() */
    }/* End of for(i<count) */
    return;
}
/*********************************************
        OUT3();

This function finds the necessary coordinates
for the cylinder, using the pointers circle
and ptr. The result is used by OUT_CYL()
*********************************************/
out3(circle,ptr,view,flag)

 struct PRIM *circle,*ptr;
 int view,flag;
{
    int sign,co,i,j;
    float rad,buf[3][4],o1[4],len;

    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            buf[i][j] = 0;
        }
    }
    sign = (ptr->MODE == SOLID) ? 1 : -1;
    rad = circle->LEN_RAD;
    co = (EQ(ptr->POINT1[1],ptr->POINT2[i])) ? 2 : 1;
```

```
    len = ptr->LEN_RAD;

    /* Depending on the view, find the right attributes
       for the Cylinder.
    */

    switch(view){
        case V1:
            o1[X] = circle->CENTER[0];
            o1[Y] = circle->CENTER[1];
            o1[Z] = ptr->POINT1[co];
            buf[2][1] = len;
            break;
        case V2:
            o1[X] = ptr->POINT1[co];
            o1[Y] = circle->CENTER[0];
            o1[Z] = circle->CENTER[1];
            buf[0][1] = len;
            break;
        case V3:
            o1[X] = circle->CENTER[1];
            o1[Y] = ptr->POINT1[co];
            o1[Z] = circle->CENTER[0];
            buf[1][1] = len;
            break;
    }
    for(i=0;i<3;i++)
        buf[i][0] = o1[i+1];

    buf[0][2] = rad;
    out_cyl(buf,sign,flag);
    return;
}
/*********************************************
        MATCH_LINES()

Return 1 if the two lines pointed to by ptr1 &
ptr2 have the same projection on the appropriate
axis and are parallel. Else return 0.
  Also return 0 if the pointers are the same.
*********************************************/
match_lines(ptr1,ptr2)

 struct PRIM *ptr1,*ptr2;
{
    int co,dum;

    if (ptr1 == ptr2)
        return(0);
    /* find the appropriate coordinates to be compared */

    if (NEQ(ptr1->POINT1[1],ptr1->POINT2[1])){
        co = 1; dum = 2;
    }
```

```
    else{
         co = 2; dum = 1;
    }
    /* make sure the lines are either vertical or horizontal */

    if ((NEQ(ptr1->POINT1[dum],ptr1->POINT2[dum]))   ||
       (NEQ(ptr2->POINT1[dum],ptr2->POINT2[dum])))
         return(0);

    /* check whether the two lines have the same projection
       on the above coordinate axis */

    if (EQ((ptr1->POINT1[co]),(ptr2->POINT1[co])) &&
         EQ((ptr1->POINT2[co]),(ptr2->POINT2[co])))
         return(1);
    else
         return(0);

}
/*************************************************
              MATCH1()

This function takes as input the view number,
coordinate number and the coordinate to be
matched. Looking in the list of primitives it
finds a line whose endpoints have the input
coordinate. It then returns a pointer to that
line. Else it returns 0 .
*************************************************/
struct PRIM *match1(view_num,co_num,co,prev_ptr)

  int view_num,co_num;
  float co;
  struct PRIM *prev_ptr;
{
    struct PRIM *ptr;
    int i;

    /* Find out which coordinates we are trying to match */

    if (co_num == view_num )
         i=1;

    else if (co_num == ((view_num % 3) + 1))
         i=2;

    else {    /* The given coordinate is not in this view */
         printf(" Error in match1");
         return(0);
    }
    /* Start searching the primitives for the appropriate match */

    if (prev_ptr != NULL)
         ptr = prev_ptr->NEXT;
```

```
    else
        ptr = VIEW[view_num] ;   /* view to be searched */

    while (ptr != NULL)  {

        if ((ptr->TYPE) == LINE) {
            if (EQ(ptr->POINT1[i],co) &&
                EQ(ptr->POINT2[i],co))
                 break;
        }
        ptr = ptr->NEXT;
    }/* end of while */
    return(ptr);
}
/*********************************************
        GETBASE()

Find a solid line in view(view_num) which has
the following property :
    POINT1[cor] = POINT2[cor]
*********************************************/
struct PRIM *getbase(view_num,co_num,pptr)

int view_num,co_num;
struct PRIM *pptr;
{
    struct PRIM *ptr;
    int cor;

    ptr = (pptr == NULL) ? VIEW[view_num] : pptr->NEXT;
    cor = (view_num == co_num) ? 1 : 2;
    while(ptr != NULL){
        while(ptr->TYPE == LINE && ptr->USE != BASE){

            if (EQ(ptr->POINT1[cor],ptr->POINT2[cor]))
             return(ptr);
            else
             break;
        }
        ptr = ptr->NEXT;
    }
    return(NULL);
}
/*********************************************
        GETHEAD()

This function finds a line that is parallel
with the base and has the same length.
*********************************************/
struct PRIM *gethead(base,leg,view_num)

 struct PRIM *base,*leg;
 int view_num;
{
```

```
    int i,j,co_num;
    float co1,co2,co3,dum1,dum2,dum3;
    struct PRIM *ptr;

    i = (view_num == 1) ? 2 : 1;
    j = (i == 1) ? 2 : 1;
    co_num = (view_num == 3) ? 3 : 2;

    co1 = base->POINT1[i]; co2 = base->POINT2[i];
    dum1 = leg->POINT1[j]; dum2 = leg->POINT2[j];
    dum3 = base->POINT1[j]; co3 = (EQ(dum1,dum3)) ? dum2 : dum1;

    ptr = VIEW[view_num];
    while ((ptr = match1(view_num,co_num,co3,ptr)) != NULL){
        if (ptr->MODE == SOLID && ptr->USE != BASE){
            if ((EQ(ptr->POINT1[i],co1) &&
                EQ(ptr->POINT2[i],co2))

                ||

                (EQ(ptr->POINT2[i],co1) &&
                 EQ(ptr->POINT1[i],co2)))

                 break;
        }
    }
    return(ptr);
}
/*********************************************
        CUBE()

Find and output all the remaining Cubes
*********************************************/
Cube()
{
    int i_h,i_b,i_l1,i_l2,found,sign,flag;
    float co11,co12,co21,co22,dum1,dum2,dum,teta1,teta2;
    struct PRIM *ptr,*pptr,*base,*head,*leg1,*leg2;
    struct PRIM *v2_arm1[5],*v2_arm2[5],*v3_arm1[5],*v3_arm2[5];

    ptr = 0;

    base = 0;

    /* Look for a horizontal line in view 1 */

    while ((base=getbase(V1,Y,base)) != NULL ){
      if(base->MODE == SOLID){

        co11=base->POINT1[1];
        co12=base->POINT1[2];
        co21=base->POINT2[1];
        co22=base->POINT2[2];
```

```
/* Flag will be used to determine whether
   we have a cube or a frustum */

     leg1=0;
     found = 0;
     while ((leg1 = findline(V1,co11,co12,leg1))!=NULL){
          leg2=0;
          while ((leg2=findline(V1,co21,co22,leg2))
                                        !=NULL){
               if(match_lines(leg1,leg2) == TRUE  &&
               ((leg1->USE != leg2->USE) || (leg1->USE == NULL))){

/* It is a cube */

                    found = TRUE;
                    flag = 0;
                    break;
               }
               else if(EQ(leg1->LEN_RAD,leg2->LEN_RAD)
                    && base != leg1){
                    dum1 = (EQ(leg1->POINT1[2],co12)) ?
                    leg1->POINT2[2] : leg1->POINT1[2];
                    dum2 = (EQ(leg2->POINT1[2],co22)) ?
                    leg2->POINT2[2] : leg2->POINT1[2];
                    if(EQ(leg1->LEN_RAD,leg2->LEN_RAD) &&
                    EQ(dum1,dum2) && NEQ(dum1,co12) &&
                    leg1->USE == NULL && leg2->USE == NULL){
                        teta1 = angle(leg1,V1);
                        teta2 = angle(leg2,V1);
                        if(EQ(teta1,teta2) && (teta1 - 0.0) > 5.0
                         && (90 - teta1) > 5.0){

/* We have a frustum  */

                            found = TRUE;
                            flag = 1;
                            break;
                            }
                    }
               }
          }
          if (found)
               break;
     }

     if (found){
          leg1->USE = ((int) base);
          leg2->USE = ((int) base);
          head = gethead(base,leg1,V1);

/* Look for corresponding lines in the other views */

          i_b = 0;
          pptr = 0;
```

```
        while((v2_arm1[i_b]=match1(V2,Y,co12,pptr)) != NULL){
             pptr = v2_arm1[i_b++];
        }
        i_h = 0;
        pptr = 0;
        dum1 = leg1->POINT1[2];
        dum = (EQ(co12,dum1)) ? leg1->POINT2[2] : dum1;
        while((v2_arm2[i_h]=match1(V2,Y,dum,pptr)) != NULL){
             pptr = v2_arm2[i_h++];
        }
        i_l1 = 0;
        pptr = 0;
        while((v3_arm1[i_l1]=match1(V3,X,co11,pptr)) !=NULL){
             pptr = v3_arm1[i_l1++];
        }
        i_l2 = 0;
        pptr = 0;
        while((v3_arm2[i_l2]=match1(V3,X,co21,pptr)) !=NULL){
             pptr = v3_arm2[i_l2++];
        }

/* Depending on the number of lines found in each view,
   choose the best candidate for the depth of the Cube,
   and output it.
*/

        while(i_h!=0 && i_b!=0 && i_l1!=0 && i_l2!=0){

             if(i_h==1 && i_b==1 && i_l1==1 && i_l2==1){
               if(match_lines(v2_arm1[0],v2_arm2[0]) == TRUE){
                  sign = cavity(v2_arm1[0]);
                  out(base,leg1,leg2,head,v2_arm1[0],sign,flag);
                  break;
               }
               if(match_lines(v3_arm1[0],v3_arm2[0]) == TRUE){
                  sign = cavity(v3_arm1[0]);
                  out(base,leg1,leg2,head,v3_arm1[0],sign,flag);
                  break;
               }
               if ((ptr=ufun1(v2_arm1[0],v2_arm2[0],V2,Y)) != NULL){
                  sign = cavity(ptr);
                  out(base,leg1,leg2,head,ptr,sign,flag);
                  break;
               }
               if ((ptr=ufun1(v3_arm1[0],v3_arm2[0],V3,X)) != NULL){
                  sign = cavity(ptr);
                  out(base,leg1,leg2,head,ptr,sign,flag);
                  break;
               }
               if ((ptr=pmatch(v2_arm1[0],v2_arm2[0],V2,Z)) != NULL){
                  sign = cavity(ptr);
                  out(base,leg1,leg2,head,ptr,sign,flag);
                  break;
               }
```

```
                if ((ptr=pmatch(v3_arm1[0],v3_arm2[0],V3,Z)) != NULL){
                   sign = cavity(ptr);
                   out(base,leg1,leg2,head,ptr,sign,flag);
                   break;
                }
              }/* end of if(all of them = 1) */
              else if(i_h==1 && i_b==1){

/* If only two lines are found in view 2
   give priority to them
*/

                if(match_lines(v2_arm1[0],v2_arm2[0]) == TRUE){
                   sign = cavity(v2_arm1[0]);
                   out(base,leg1,leg2,head,v2_arm1[0],sign,flag);
                   break;
                }
                if((ptr=ufun1(v2_arm1[0],v2_arm2[0],V2,Y))!= NULL){
                   sign = cavity(ptr);
                   out(base,leg1,leg2,head,ptr,sign,flag);
                   break;
                }
                if((ptr=pmatch(v2_arm1[0],v2_arm2[0],V2,Z))!= NULL){
                   sign = cavity(ptr);
                   out(base,leg1,leg2,head,ptr,sign,flag);
                   break;
                }
                else{
                   fprintf(stderr,"impossible interpretation0);
                   exit(1);
                }
              }/* end of if(i_h,i_b = 1) */

              else if(i_l1==1 && i_l2==1){

/* If only two lines are found in view 3
   give priority to them
*/

                if(match_lines(v3_arm1[0],v3_arm2[0]) == TRUE){
                   sign = cavity(v3_arm1[0]);
                   out(base,leg1,leg2,head,v3_arm1[0],sign,flag);
                   break;
                }
                if((ptr=ufun1(v3_arm1[0],v3_arm2[0],V3,X))!= NULL){
                   sign = cavity(ptr);
                   out(base,leg1,leg2,head,ptr,sign,flag);
                   break;
                }
                if((ptr=pmatch(v3_arm1[0],v3_arm2[0],V3,Z))!= NULL){
                   sign = cavity(ptr);
                   out(base,leg1,leg2,head,ptr,sign,flag);
                   break;
                }
```

```
                    else{
                        fprintf(stderr,"impossible interpretation0);
                        exit(1);
                    }

                }/* end of if(i_l1,i_l2 = 1)*/

                else if(i_b==1){
                        sign = cavity(v2_arm1[0]);
                        out(base,leg1,leg2,head,v2_arm1[0],sign,flag);
                        break;
                }
                else if(i_h==1){
                        sign = cavity(v2_arm2[0]);
                        out(base,leg1,leg2,head,v2_arm2[0],sign,flag);
                        break;
                }
                else if(i_l1==1){
                        sign = cavity(v3_arm1[0]);
                        out(base,leg1,leg2,head,v3_arm1[0],sign,flag);
                        break;
                }
                else if(i_l2==1){
                        sign = cavity(v3_arm2[0]);
                        out(base,leg1,leg2,head,v3_arm2[0],sign,flag);
                        break;
                }
                else
                        break;
            }/* end of while(it is a cube) */
        }/* end of if(found) */
      }
      base->USE = BASE;
    }/* end of while (base....) */
}
/*********************************************
        OUT()

This function finds the coordinates of the cube
from its input arguments. The results are used
by out_cube()
*********************************************/
out(base,leg1,leg2,head,ptr,sign,flag)

 struct PRIM *base,*leg1,*leg2,*head,*ptr;
 int sign,flag;
{
    float buf[3][4];
    int cor,i,j;

    if (flag == 1){

    /* It is a frustum */
```

```
        out2(leg1,leg2,ptr,v1,1);
        return;
    }
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            buf[i][j] = 0.0;

    flag = ufun4(base,leg1,leg2,head);
    cor = (EQ(ptr->POINT1[1],ptr->POINT2[1])) ? 2 : 1;

    buf[0][0] = base->POINT1[i];
    buf[1][0] = (EQ(base->POINT1[2],leg1->POINT2[2])) ?
        leg1->POINT1[2] : base->POINT1[2];
    buf[2][0] = ptr->POINT1[cor];

    buf[0][1] = base->POINT2[1] - base->POINT1[1];
    buf[1][1] = leg1->LEN_RAD;
    buf[2][1] = ptr->LEN_RAD;

    out_cube(buf,sign,flag,0);
    return;
}
/*************************************************
        PMATCH()

 Partially match two lines and return the result.
(ptr or FALSE)
*************************************************/
struct PRIM *pmatch(line1,line2,view_num,co_num)

 struct PRIM *line1,*line2;
 int view_num,co_num;
 {
    int i;
    float dum1,dum2,co11,co12,co21,co22;

    i = (view_num == co_num) ? 1 : 2;
    co11 = line1->POINT1[i]; co12 = line1->POINT2[i];
    co21 = line2->POINT1[i]; co22 = line2->POINT2[i];

    if (co11 <= co21 && co12 >= co22)     {
        return(line1);
    }
    else if (co21 <= co11 && co22 >= co12) {
        return(line2);
    }
    else
        return(FALSE);
}
/*************************************************
        UFUN1()

This function checks to see whether any of the
given lines are at an extreme position in the
```

```
 given view.
 ************************************************/
 struct PRIM *ufun1(ptr1,ptr2,view_num,co_num)

  struct PRIM *ptr1,*ptr2;
  int  view_num,co_num;
 {
      int cor;
      float max,min,cmax,cmin,c1,c2;
      struct PRIM *maxptr,*minptr,*ptr;

      cor = (view_num == co_num) ? 1 : 2;
      min = 1000.0; max = 0.0;
      minptr = 0; maxptr = 0;
      ptr = VIEW[view_num];
      while (ptr != NULL) {
           if (ptr->TYPE == LINE &&
               EQ(ptr->POINT1[cor],ptr->POINT2[cor])){
             maxptr = (ptr->POINT1[cor] > max) ? ptr : maxptr;
             minptr = (ptr->POINT1[cor] < min) ? ptr : minptr;
           }
           ptr = ptr->NEXT;
      }
      cmax = maxptr->POINT1[cor]; cmin = minptr->POINT1[cor];
      c1 = ptr1->POINT1[cor]; c2 = ptr2->POINT1[cor];

      if (EQ(c1,cmax) || EQ(c1,cmin))
           return(ptr1);
      else if (EQ(c2,cmax) || EQ(c2,cmin))
           return(ptr2);
      else
           return(NULL);
 }
 /*********************************************
          CAVITY()

 Return the mode of the input line.
 ************************************************/
 cavity(line)

 struct PRIM *line;
 {
     return(line->MODE);
 }
 /*********************************************
          UFUN2()

 Finds the two lines (leg1 & leg2) connected to
 the two endpoints of the semicircle.
 ************************************************/
 ufun2(sem_circ,view,pleg1,pleg2)

  struct PRIM *sem_circ,*(*pleg1),*(*pleg2);
  int view;
```

```
{
     int i;
     float co11,co12,co21,co22;
     struct PRIM *pp1,*pp2;

     co11 = sem_circ->POINT1[1]; co12 = sem_circ->POINT1[2];
     co21 = sem_circ->POINT2[1]; co22 = sem_circ->POINT2[2];
/* i is the coordinate that is  the same in POINT1 & POINT2. */
     if(EQ(co11,co21))
          i = 1;
     else
          i = 2;
     pp1 = 0;
     while(( (*pleg1) = findline(view,co11,co12,pp1)) != NULL ){
          pp1 = (*pleg1); pp2 = 0;
          while(( (*pleg2) = findline(view,co21,co22,pp2)) != NULL){
               pp2 = (*pleg2);
               if(EQ((*pleg1)->POINT1[i],(*pleg2)->POINT1[i]) &&
                  EQ((*pleg1)->POINT2[i],(*pleg2)->POINT2[i]))
                    return;
               else if(EQ((*pleg1)->POINT2[i],(*pleg2)->POINT1[i]) &&
                  EQ((*pleg1)->POINT1[i],(*pleg2)->POINT2[i]))
                    return;
          }
     }
     printf("legs not found for semicircle0);
     return;
}
/*********************************************
          FINDLINE()

Finds a line in the given view that has an
endpoint with the given coordinates.
*********************************************/
struct PRIM *findline(view,co1,co2,pptr)

 int view;
 float co1,co2;
 struct PRIM *pptr;
{
     struct PRIM *ptr;

     ptr = (pptr == NULL) ? VIEW[view] : pptr->NEXT;
     while (ptr != NULL){
          if(ptr->TYPE == LINE && ptr->MODE == SOLID){
               if ((EQ(ptr->POINT1[1],co1) && EQ(ptr->POINT1[2],co2))
                         ||
                   (EQ(ptr->POINT2[1],co1) && EQ(ptr->POINT2[2],co2)))

                    return(ptr);
          }
          ptr = ptr->NEXT;
     }
     return(ptr);
```

```
}
/*********************************************
        LUG()

Find and output all the lugs
*********************************************/
Lug(view)

 int view;
 {
     int xi,xj,count,found,i,sign,flag1,flag2,pos;
     float ci,cj,rad,co11,co12,co21,co22,co1,co2;
     float teta1,teta2;
     struct PRIM *buff[MAXSIZE],*ptr,*tmp,*pptr,*leg1,*leg2,*dum;

     /* Get all the curves */

     count = get_curve(view,buff,ARC);
     if (count == NULL)   return;
     for (i=0;i<count;i++){
        tmp = buff[i];
        if (tmp->POINT1[0] < 5){
          rad = tmp->LEN_RAD;
          co11 = tmp->POINT1[1];
          co12 = tmp->POINT1[2];
          co21 = tmp->POINT2[1];
          co22 = tmp->POINT2[2];
          leg1 = 0;  leg2 = 0;

     /* Find the coordinates of c which is
        the midpoint on the curve . xi is
        the coordinate that is not the same
        in POINT1 & POINT2
     */

          xi = (EQ(tmp->POINT1[1],tmp->POINT2[1])) ? 2 : 1;
          xj = (xi == 1) ? 2 : 1;
          ci = tmp->CENTER[xi-1];
          if(view != V3){
            if(tmp->POINT1[0] == 1 || tmp->POINT1[0] == 3 )
              cj = tmp->CENTER[xj-1] + rad;
            else
              cj = tmp->CENTER[xj-1] - rad;
          }
          else{
            if(tmp->POINT1[0] == 1 || tmp->POINT1[0] == 4 )
              cj = tmp->CENTER[xj-1] - rad;
            else
              cj = tmp->CENTER[xj-1] + rad;
          }
          pptr = 0;  ptr = 0;

     /* Depending on the view and position of the
        Lug, find its depth from a secondary view */
```

```
        switch(view) {
             case V1:
                  if(xi==1)
                    ptr=match1(V2,Y,cj,pptr);
                  else
                    ptr=match1(V3,X,cj,pptr);
                  break;

             case V2:
                  if(xi==1)
                    ptr=match1(V3,Z,cj,pptr);
                  else
                    ptr=match1(V1,Y,cj,pptr);
                  break;

             case V3:
                  if(xi==1)
                    ptr=match1(V1,X,cj,pptr);
                  else
                    ptr=match1(V2,Z,cj,pptr);
        }
        if ( ptr == NULL){
             fprintf(stderr,"Error in DO_LUG");
             exit(1);
        }

        /* output the cylinder */

        flag1 = (ptr->MODE == SOLID) ? -1 : 0;
        out3(tmp,ptr,view,flag1);

        /* output the cube to be subtracted if necessary*/

        if (flag1 == -1)
        out1(tmp,leg1,ptr,view,-1,1);

      /* if the endpoints are connected,
         go to the next ARC */

        dum = VIEW[view];
        found = FALSE;
        while (dum != NULL){
             if(dum->TYPE == LINE){
               if((EQ(dum->POINT1[1],co11) && EQ(dum->POINT1[2],co22)) &&
                  (EQ(dum->POINT2[1],co21) && EQ(dum->POINT2[2],co22))){
                  found = TRUE;
                  break;
               }
             }
             dum = dum->NEXT;
        }
        if (found == FALSE){

      /* The endpoints are not connected
```

```
        find the lines connected to them
    */
            ufun2(tmp,view,&leg1,&leg2);
            if(match_lines(leg1,leg2) == TRUE){

        /* output the cube */

            sign = (ptr->MODE == SOLID) ? 1 : -1;
            out1(tmp,leg1,ptr,view,sign,0);
            (leg1->USE) = LEG; (leg2->USE) = LEG;
            (ptr->USE)++;
            }
            else if (EQ(leg1->LEN_RAD,leg2->LEN_RAD)){
                teta1 = angle(leg1,view);
                teta2 = angle(leg2,view);
                if(EQ(teta1,teta2)){
                    pos = tmp->POINT1[0];
                    out2(leg1,leg2,ptr,view,pos);
                     (leg1->USE) = LEG;
                      (leg2->USE) = LEG;
                     (ptr->USE)++;
                }
            }

    /* Use ufun3 to create new lines in this view */

            co1 = (EQ(co11,leg1->POINT1[1])) ?
                 leg1->POINT2[1] : leg1->POINT1[1];
            co2 = (EQ(co12,leg1->POINT1[2])) ?
                 leg1->POINT2[2] : leg1->POINT1[2];
            ufun3(co1,co2,view);

            co1 = (EQ(co21,leg2->POINT1[1])) ?
                 leg2->POINT2[1] : leg2->POINT1[1];
            co2 = (EQ(co22,leg2->POINT1[2])) ?
                 leg2->POINT2[2] : leg2->POINT1[2];
            ufun3(co1,co2,view);
        }
      }
    }
    return;
}
/*********************************************
        OUT1()

Output the cube to be subtracted from a cylinder
in order to have a half a cylinder
*********************************************/
out1(cyl,leg,ptr,view,sign,flag)

 struct PRIM *cyl,*ptr,*leg;
 int view,sign,flag;
 {
```

```
float co11,co12,co21,co22,buf[3][4],len;
float a[4],b[4],c[4],e[4],rad;
int i,j;

for(i=0;i<3;i++)
     for(j=0;j<4;j++)
          buf[i][j] = 0.0;
co11 = cyl->POINT1[1];
co12 = cyl->POINT1[2];
co21 = cyl->POINT2[1];
co22 = cyl->POINT2[2];
if(sign == 1){
     len = leg->LEN_RAD;
     rad = 0;
}
else if (flag == 1){
     len = 2*(cyl->LEN_RAD);
     rad = cyl->LEN_RAD;
}
else if (sign == -1 && flag == 0){
     len = leg->LEN_RAD;
     rad = 0;
}

/* Depending on the position and view of the Lug,
  use the above info to find the coordinates
  of the cubical part of the lug */

/* cyl->POINT1[0] gives us the position of
   the Lug. We have four possible positions */

switch(view){
     case V1:

/* The Lug is in View 1 */

          if(cyl->POINT1[0]==1){
               a[X] = co11 - rad;
               a[Y] = co12;
               a[Z] = ptr->POINT2[2];
               b[X] = co21 + rad;
               c[Y] = a[Y] - len;
          }
          else if(cyl->POINT1[0] == 2){
               a[X] = co11 - rad;
               a[Y] = co12 + len;
               a[Z] = ptr->POINT2[2];
               b[X] = co21 + rad;
               c[Y] = co12;
          }
          else if(cyl->POINT1[0] == 3){
               a[X] = co21 - len;
               a[Y] = co22 + rad;
               a[Z] = ptr->POINT2[1];
```

```
                b[X] = co11;
                c[Y] = co12 - rad;
            }
            else{
                a[X] = co11;
                a[Y] = co22 + rad;
                a[Z] = ptr->POINT2[1];
                b[X] = a[X] + len;
                c[Y] = co12 - rad;
            }
            e[Z] = a[Z] - ptr->LEN_RAD;
            break;

        case V2:

    /* The Lug is in View 2 */

            if(cyl->POINT1[0]==1){
                a[X] = ptr->POINT1[1];
                a[Y] = co11;
                a[Z] = co22 + rad;
                b[X] = ptr->POINT2[1];
                c[Y] = a[Y] - len;
                e[Z] = co12 - rad;
            }
            else if(cyl->POINT1[0] == 2){
                a[X] = ptr->POINT1[1];
                a[Y] = co11 + len;
                a[Z] = co22 + rad;
                b[X] = ptr->POINT2[1];
                c[Y] = co11;
                e[Z] = co12 - rad;
            }
            else if(cyl->POINT1[0] == 3){
                a[X] = ptr->POINT1[2];
                a[Y] = co21 + rad;
                a[Z] = co22;
                b[X] = ptr->POINT2[2];
                c[Y] = co11 - rad;
                e[Z] = co12 - len;
            }
            else{
                a[X] = ptr->POINT1[1];
                a[Y] = co21 + rad;
                a[Z] = co22 + len;
                b[X] = ptr->POINT2[2];
                c[Y] = co11 - rad;
                e[Z] = co12;
            }
            break;

        case V3:

    /* The Lug is in View 3 */
```

```
            if(cyl->POINT1[0]==1){
                a[X] = co12 - rad;
                a[Y] = ptr->POINT2[1];
                a[Z] = co11 + len;
                b[X] = co22 + rad;
                c[Y] = ptr->POINT1[1];
                e[Z] = co11;
            }
            else if(cyl->POINT1[0] == 2){
                a[X] = co12 - rad;
                a[Y] = ptr->POINT2[1];
                a[Z] = co11;
                b[X] = co22 + rad;
                c[Y] = ptr->POINT1[1];
                e[Z] = co11 - len;
            }
            else if(cyl->POINT1[0] == 3){
                a[X] = co12 - len;
                a[Y] = ptr->POINT2[2];
                a[Z] = co21 + rad;
                b[X] = co12;
                c[Y] = ptr->POINT1[2];
                e[Z] = co11 - rad;
            }
            else{
                a[X] =   co12;
                a[Y] =   ptr->POINT2[2];
                a[Z] =   co21 + rad;
                b[X] =   a[X] + len;
                c[Y] =   ptr->POINT1[2];
                e[Z] =   co11 - rad;
            }
    }

    /* The following is always true. */

    c[X] = a[X]; c[Z] = a[Z];

    buf[0][0] = c[X]; buf[1][0] = c[Y];
    buf[2][0] = e[Z]; buf[0][1] = b[X] - c[X];
    buf[1][1] = a[Y] - c[Y]; buf[2][1] = c[Z] - e[Z];

    out_cube(buf,sign,flag,0);
    return;
}
```

```
     /*    ALGORITHM 2    */


#include <stdio.h>
#include <math.h>

/*
     GLOBAL VARIABLES
*/

int pic[200][200],X0,Y0,Z0,VIEW;
float angle();
/*****************************************
          MAIN

In main, the picture is scanned. If any curve
is found, it is followed and chain coded.
Then it is determined which routine should
be called in order to process the curve.
*****************************************/
main()
{
     int i,j,chain[1000],dif[1000],lines[50][2],n,code,count;
     int found,start_X,start_Y,end_X,end_Y,link,nlink;
     int k,k1,k2,*ptr,pix,dist;
     extern int pic[][200],VIEW,X0,Y0,Z0;
     FILE *fp,*fopen();

/* Read view number and the values by which the picture should be
  shifted to obtain a global coordinate system. */

     scanf("%d",&VIEW); scanf("%d",&X0);
     scanf("%d",&Y0); scanf("%d",&Z0);

     fp = fopen("inimage","r");
     if(fp == NULL)
          exit(1);
     ptr = (&pic[0][0]);

     /* Read in picture into the 2D array pic */

     while((pix = getc(fp)) != EOF){
          *ptr = pix; ptr++;
     }
     start_X = 0; start_Y = 0;
begin:
     link = 7;  found = 0;

     /* Look for a pixel with value = 1 */

     for(i = 0;i < 200;i++){
          for(j = 0;j < 200;j++){
               if(pic[i][j] != 1)
                    continue;
```

```
            else{
                found = 1; break;
            }
        }
        if(found == 1)  break;
    }
    if (!found)  exit(0); /* DONE */
    start_X = j;   start_Y = i;  k = 0;

    /* Follow and chain code the curve */

    while(i < 200  &&  j < 200){
        if((nlink = getnbor(i,j,link,1)) < 0)
            break;
        chain[k++] = nlink;  link = nlink;
        pic[i][j] = 2;
        i = i + dely(nlink);
        j = j + delx(nlink);
    }
    pic[i][j] = 2;
    n = k;  chain[k] = -1;

    /* See if it is noise */

    if(n < 5)  goto begin;
    nlink = getnbor(i,j,link,2);
    if (nlink == -1){

    /* It is a dotted line */

        dotline(start_X,start_Y,i,j,link);
        goto begin;
    }
    i = i + dely(nlink);
    j = j + delx(nlink);
    if ((abs(i-start_Y)<2) && (abs(j-start_X)<2)){

    /* CLOSED  LOOP  */

        for(i = 0;i < n-1;i++)
            dif[i] = chain[i+1] - chain[i];
        dif[i] = chain[0] - chain[n-1];
        dif[n] = 8;
        count = getlines(dif,lines,n);
        if (count == 0){

    /* It is a circle */

            do_circle(chain,start_X,start_Y,n);
            goto begin;
        }
        /* find the coordinates of the lines and output them */

        do_lines(chain,n,lines,start_X,start_Y,count);
```

```
        /* see whether there is anything else
           besides straight lines */

            found = 0; j = count - 1;
            for(i = 0;i < count - 1;i++){
                if(lines[i][1] == lines[i+1][0])
                    continue;
                else{
                    found = 1; j = i;
                }
            }
            if(lines[0][0] != lines[count-1][1]){
                    found = 1; j = count - 1;
            }
            if(found == 0)

        /* Nothing but straight lines in this chain */

                goto begin;
            k1 = lines[j][1]; k2 = lines[(j+1)%count][0];
            if (k2 < k1)
                dist = k2 + n + 1 -k1;
            else
                dist = k2 - k1;
            if (dist < 25){

        /* This is a short line */

                lines[0][0] = k1; lines[0][1] = k2; count = 1;
                do_lines(chain,n,lines,start_X,start_Y,count);
                goto begin;
            }

        /* An arc or a slanted line lies between k1 & k2 */

            segment(chain,n,k1,k2,start_X,start_Y);
            goto begin;
        }
        /* If not a closed loop or a dotted line */
        /* Assume one or more straight lines */
        for(i = 1;i < n-1;i++)
            dif[i] = chain[i+1] - chain[i];
        dif[i] = 8; dif[0] = 8;
        count = getlines(dif,lines,n);
        if (count == 0)
            fprintf(stderr," ERROR 0);
        else
            do_lines(chain,n + 1,lines,start_X,start_Y,count);
        goto begin;
}
/*****************************************
        DO_LINES()

Find the coordinates of the lines using the
```

```
array lines[][]
*******************************************/
do_lines(chain,n,lines,x0,y0,count)

int chain[],n,lines[][2],x0,y0,count;
{
     int i,k1,k2,p1x,p2x,p1y,p2y;

     for(i = 0;i < count;i++){
          k1 = lines[i][0];
          k2 = lines[i][1];
          findco(chain,n,x0,y0,k1,&p1x,&p1y);
          findco(chain,n,x0,y0,k2,&p2x,&p2y);
          out_line(p1x,p1y,p2x,p2y,1);
     }
     return;
}
/*******************************************
          DO_CIRCLE()

Find the center and radius of the circle
*******************************************/
do_circle(chain,x0,y0,n)

int chain[],x0,y0,n;
{
     int count,k1,k2,k3,x[3],y[3],i;
     float av_ox,av_oy,ox,oy,av_rad,rad,delx,dely;

     count = 0; k1 = 0, av_ox = 0; av_oy = 0; av_rad = 0;
     while(1){
          k2 = (k1 + 10)%n;
          k3 = (k1 + 20)%n;
          findco(chain,n,x0,y0,k1,&x[0],&y[0]);
          findco(chain,n,x0,y0,k2,&x[1],&y[1]);
          findco(chain,n,x0,y0,k3,&x[2],&y[2]);

          getcenter(x,y,&ox,&oy);
          if(ox > 0.0){
               av_ox = av_ox + ox;
               av_oy = av_oy + oy;
               count++;
          }
          if(k3 < k1) break;
          k1 = k3 + 5;
     }
     ox = av_ox /((float) count);
     oy = av_oy /((float) count);
/* find the radius */
     for(i = 0;i < 3;i++){
          delx = ((float) x[i] - ox) * ((float) x[i] - ox);
          dely = ((float) y[i] - oy) * ((float) y[i] - oy);
          av_rad = av_rad + sqrt((double) (delx + dely));
     }
```

```
        rad = av_rad / 3.0;
        out_circle(ox,oy,rad);
        return;
}
/*****************************************
        DO_ARC()

Find the coordinates and the position
of the arc
*****************************************/
do_arc(chain,n,k1,k2,x0,y0)

int chain[],n,k1,k2,x0,y0;
{
        float av_ox,ox,av_oy,oy;
        int x[3],y[3],count,i,p1x,p2x,p1y,p2y,cx,cy,pos,k;

        findco(chain,n,x0,y0,k1,&p1x,&p1y);
        findco(chain,n,x0,y0,k2,&p2x,&p2y);
/* Need to know how many links between k1 & k2 */
        if(k2 > k1)
                count = k2 - k1;
        else
                count = k2 + n + 1 - k1;
/* find out position of arc */
        k = (k1 + (count/2))%n;
        findco(chain,n,x0,y0,k,&cx,&cy);
        if(abs(p1x - p2x) < 6) {/* vertical */
                if(cx < p1x)
                        pos = 4;
                else
                        pos = 3;
        }
        else{ /* horizontal */
                if(cy > p1y)
                        pos = 1;
                else
                        pos = 2;
        }
/* Estimate the coordinates of the center */
        av_ox = 0; av_oy = 0;
        k = (k1 + (count/4))%n;
        x[1] = cx; y[1] = cy;
        x[2] = p1x; y[2] = p1y;
        for(i = 0;i < 2;i++){
                while(1){
                        findco(chain,n,x0,y0,k,&x[0],&y[0]);
                        getcenter(x,y,&ox,&oy);
                        if(ox > 0.0 ) break;
                        k = (k + 1)%n;
                        if(k == k2){
                                printf("det = 0");
                                exit(1);
                        }
```

```
            }
            av_ox = av_ox + ox;
            av_oy = av_oy + oy;
            k = (k1 + (3*count/4))%n;
            x[2] = p2x; y[2] = p2y;
        }
        ox = av_ox/2.0;
        oy = av_oy/2.0;

        out_arc(p1x,p1y,p2x,p2y,ox,oy,pos);
        return;
}
/*******************************************
        DOTLINE()

Follow a dotted line and find its endpoints
*******************************************/
dotline(x0,y0,ii,jj,link)

int x0,y0,ii,jj,link;
{
        int i,j,nlink,end_x,end_y;
        extern int pic[][200];

        j = jj; i = ii;
        while((i < 200) && (j < 200)){
            if((nlink = getnbor(i,j,link,2)) != -1)
                break;
            pic[i][j] = (pic[i][j] == 1) ? 2 : 0;
            i = i + dely(link);
            j = j + delx(link);
        }
            pic[i][j] = 2;
        end_x = j;  end_y = i;
        out_line(x0,y0,end_x,end_y,0);
}


/*******************************************
        FINDCO()

Given the chain array and the coordinates
of the start of the chain, find the
coordinates of the pixel corresponding
to the chain link k.
*******************************************/
findco(chain,n,x0,y0,k,px,py)

int chain[],n,x0,y0,k,*px,*py;
{
        int k1,i,X,Y;

        k1 = (k + 1)%n;
        X = 0; Y = 0;
```

```
    for(i = 0;i<k1;i++){
        X = X + delx(chain[i]);
        Y = Y + dely(chain[i]);
    }
    *px = x0 + X;
    *py = y0 + Y;

    return;
}
/*****************************************
        GETCENTER()

Using the coordinates of three points on the
circle or arc, find an estimate for the
coordinates of the center.
*****************************************/
getcenter(x,y,pox,poy)

int x[],y[];
float *pox,*poy;
{
    float x02,x12,x22,y02,y12,y22;
    float a,b,c,d,det1,det2,det3;

    x02 = x[0] * x[0]; x12 = x[1] * x[1]; x22 = x[2] * x[2];
    y02 = y[0] * y[0]; y12 = y[1] * y[1]; y22 = y[2] * y[2];

    /* for ox */

    a = x02 - x12 + y02 - y12;
    b = 2 * (y[0] - y[1]);
    c = x02 - x22 + y02 - y22;
    d = 2 * (y[0] - y[2]);
    det1 = (a * d) - (b * c);

    a = 2 * (x[0] - x[1]);
    b = 2 * (y[0] - y[1]);
    c = 2 * (x[0] - x[2]);
    d = 2 * (y[0] - y[2]);
    det2 = (a * d) - (b * c);
    if(det2 == 0){
        *pox = -1;
        return;
    }

    /* for oy */
    a = 2 * (x[0] - x[1]);
    b = x02 - x12 + y02 - y12;
    c = 2 * (x[0] - x[2]);
    d = x02 - x22 + y02 - y22;
    det3 = (a * d) - (b * c);

    *pox = det1 / det2;
    *poy = det3 / det2;
```

```
    return;
}
/*******************************************
        GETNBOR()

Scan the neighbors of pic[i][j],in a
counter clockwise direction, 4 neighbors
first. If any with value equal to pix is
found, return the link code. Else return -1
*******************************************/
getnbor(i,j,link,pix)

int i,j,link,pix;
{
    int nlink,dum,k,ni,nj,i1,j1,i2,j2;
    extern int pic[][200];

    nlink = (link + 5) % 8;
    if ((dum = nlink % 2) != 0) nlink = (nlink + 1) % 8;
/* check 4 neighbors */
    for(k = 0;k < 3;k++){
        ni = i + dely(nlink);
        nj = j + delx(nlink);
        if((ni<200) && (ni>0) && (nj<200) && (nj>0)){
            if (pic[ni][nj] == pix){
             if(nlink != link && pix == 1 && (link == 0 ||
                 link == 2 || link == 4 || link == 6)){
                 i1 = i + dely(link);
                 j1 = j + delx(link);
                 i2 = ni + dely(nlink);
                 j2 = nj + delx(nlink);
                 if(pic[i1][j1] == 1 && pic[i2][j2] != 1){
                     nlink = link;
                     pic[ni][nj] = 2;
                 }
             }
                return(nlink);
            }
        }
        nlink = (nlink + 2) % 8;
    }
    nlink = (link + 5) % 8;
    if (dum == 0) nlink = (nlink + 1) % 8;
    for(k = 0;k < 4;k++){
        ni = i + dely(nlink);
        nj = j + delx(nlink);
        if((ni<200) && (ni>0) && (nj<200) && (nj>0)){
            if (pic[ni][nj] == pix)
              return(nlink);
        }
        nlink = (nlink + 2) % 8;
    }
    return(-1);
}
```

```
/*****************************************
        GETLINES()

Find all horizontal and vertical lines using
the difference array.
*****************************************/
getlines(dif,lines,n)

int dif[],lines[][2],n;
{
    int i,k,k2,dist,firstk,count;

    i = 0; count = 0;
    k = scan(dif,0,1,n);
    firstk = -1;
    while(k != firstk){
        if(count == 0){
            firstk = k; count++;
        }
        k2 = scan(dif,k,1,n);
        if (k2 < k)
            dist = k2 + n + 1 -k;
        else
            dist = k2 - k;
        if (dist > 25){
            lines[i][0] = k;
            lines[i++][1] = k2;
        }
        k = k2;
    }
    return(i);
}
/*****************************************
        SCAN()

Scan the dif array in the direction dir, from
position start. If a nonzero entry is found
return its position. Also take care of
nonzero entries due to distortion.
*****************************************/
scan(dif,start,dir,n)

int start,dir,n,dif[];
{
    int i,k,dum,save[3],count,save_i;

    i = start + dir;
    if ((start == 0) && (dir == -1))
        i = n - 1;
    else if ((start == n-1) && (dir == 1))
        i = 0;
    k = start; count = 0;
    while (1){
        k = k + dir;
```

```
            if (k < 0)
                i = k + n;
            else
                i = k%n;
            if(dif[i] != 0){ /* make sure it is not distortion */
                if(dif[i] == 8) return(i);
                dum = - dif[i];
                k = k + dir;
                if (k < 0)
                    i = k + n;
                else
                    i = k%n;
                if(dif[i] != dum)
                    break;
                if(count == 0) save_i = i;
                save[count++] = -dum;
                if(count > 1){
                    if(save[0] == save[1])
                        return(save_i);
                    count = 0;
                }
            }
        }
        return(i);
}
/*********************************************
        SEGMENT()

Using the curvature function, segment the
array into lines and arcs.
*********************************************/
segment(chain,n,k1,k2,x0,y0)

int chain[],n,k1,k2,x0,y0;
{
    int i,j,s,dum[700],count,lines[50][2],kk1,kk2;
    float teta,teta1,teta2,del,totdel[700];

    j = 1; dum[0] = chain[k1]; i = k1;
    while (i != k2){
        i = (i + 1) % n;
        dum[j++] = chain[i];
    }
    count = j; s = 15;
    for(j = 0; j < count+1; j++){
        teta = angle(dum,j,s,count);
        teta = (teta < -90.0) ? teta + 360.0 : teta;
        teta1 = angle(dum,j-3,s,count);
        teta2 = angle(dum,j+3,s,count);
        del = teta2 - teta1;
        if (del < -180) del = del + 360.0;
        else if (del > 180) del = del - 360.0;
        totdel[j] = (j < 20) ? 0 :totdel[j-1] + del;
```

```
        }
        j = 20;
        while(totdel[j++] < 50);
        if (j < 35){ /* It is only an ARC */
             do_arc(chain,n,k1,k2,x0,y0);
             return;
        }
        kk1 = (k1 + j) %n;
        lines[0][0] = k1; lines[0][1] = kk1;
        i = 0;
        while(j < count-5 && i < 4){
             if((totdel[j+5] - totdel[j]) < 10)
                  i++;
             j++;
        }
        j = j - s; /* take into account the lead */
        kk2 = (k1 + j) %n;
        do_arc(chain,n,kk1,kk2,x0,y0);
        lines[1][0] = kk2; lines[1][1] = k2;
        do_lines(chain,n,lines,x0,y0,2);
}
/*******************************************
          ANGLE()

Given that point A corresponds to chain[j] and
point B corresponds to chain[j-s], find the
angle that the line AB makes with the horizontal
*******************************************/
float angle(chain,j,s,n)

int j,s,n,chain[];
{
     int i,k;
     float X,Y,teta;

     X = 0;
     Y = 0;
     j = j % n;
     for(k = j-s+1; k < j+1; k++){
          i = (k<0) ? k + n : k;
          X = X + delx(chain[i]);
          Y = Y + dely(chain[i]);
     }
     teta = atan2(-Y,X);
     return(teta * 180.0/3.1415);
}
/*******************************************
          OUT_LINE();

Output a line
*******************************************/
out_line(p1x,p1y,p2x,p2y,mode)

int p1x,p2x,p1y,p2y,mode;
```

```
{
     extern int VIEW,X0,Y0,Z0;
     float p11,p12,p21,p22;

     switch(VIEW){
          case 1:
     /* shift elements to X0,Y0 */
               p12 = p1y - Y0; p22 = p2y - Y0;
               p11 = p1x - X0; p21 = p2x - X0;
     /* reverse with respect to z axis */
               p12 = 200 - p12; p22 = 200 - p22;
               break;
          case 2:
     /* shift elements to Y0,Z0 */
               p11 = p1y - Y0; p21 = p2y - Y0;
               p12 = p1x - Z0; p22 = p2x - Z0;
     /* reverse with respect to z axis */
               p11 = 200 - p11; p21 = 200 - p21;
               break;
          case 3:
     /* shift elements to X0,Z0 */
               p11 = p1y - Z0; p21 = p2y - Z0;
               p12 = p1x - X0; p22 = p2x - X0;
               break;
     }
     printf("1 %d %5.1f %5.1f %5.1f %5.1f0,mode,p11,p12,p21,p22);
}

/*******************************************
          OUT_CIRCLE();

Output a circle
*******************************************/
out_circle(ox,oy,rad)

float ox,oy,rad;
{
     float o1,o2;
     extern int VIEW,X0,Y0,Z0;

     switch(VIEW){
          case 1:
     /* shift elements to X0,Y0 */
               o1 = ox - X0; o2 = oy - Y0;
     /* reverse with respect to x axis */
               o2 = 200 - o2;
               break;
          case 2:
     /* shift elements to Y0,Z0 */
               o1 = oy - Y0; o2 = ox - Z0;
     /* reverse with respect to z axis */
               o1 = 200 - o1;
               break;
```

```
        case 3:
    /* shift elements to X0,Z0 */
            o1 = oy - Z0; o2 = ox - X0;
            break;
    }
    printf("2 1 %5.2f %5.2f %5.2f0,o1,o2,rad);
}
/*****************************************
        OUT_ARC(),

Output an arc
*****************************************/
out_arc(p1x,p1y,p2x,p2y,ox,oy,pos)

int p1x,p2x,p1y,p2y,pos;
float ox,oy;
{

    float p11,p12,p21,p22,o1,o2;
    extern int VIEW,X0,Y0,Z0;

    switch(VIEW){
        case 1:
    /* shift elements to X0,Y0 */
            p12 = p1y - Y0; p22 = p2y - Y0;
            p11 = p1x - X0; p21 = p2x - X0;
            o1 = ox - X0; o2 = oy - Y0;
    /* reverse with respect to x axis */
            p12 = 200 - p12; p22 = 200 - p22;
            o2 = 200 - o2;
            break;
        case 2:
    /* shift elements to Y0,Z0 */
            p11 = p1y - Y0; p21 = p2y - Y0;
            p12 = p1x - Z0; p22 = p2x - Z0;
            o1 = oy - Y0;  o2 = ox - Z0;
    /* reverse with respect to z axis */
            p11 = 200 - p11; p21 = 200 - p21;
            o1 = 200 - o1;
            break;
        case 3:
    /* shift elements to X0,Z0 */
            p11 = p1y - Z0; p21 = p2y - Z0;
            p12 = p1x - X0; p22 = p2x - X0;
            o1 = oy - Z0; o2 = o2 - X0;
            break;
    }
    if(VIEW != 3){
        if(pos == 1)
            pos = 2;
        else if(pos == 2)
            pos = 1;
    }
    printf("3 1 %5.1f %5.1f %5.1f %5.1f ",p11,p12,p21,p22);
```

```
    printf("%d %5.1f %5.1f0,pos,o1,o2);
}
/*********************************************
        DELX()

Return the displacement in the horizontal
direction due to the chain link "code"
*********************************************/
delx(code)

int code;
{

    switch(code){
        case 0: return(1);
        case 1: return(1);
        case 7: return(1);
        case 2: return(0);
        case 6: return(0);
        case 3: return(-1);
        case 4: return(-1);
        case 5: return(-1);
        default:
            printf(" Unknown code0);
            return(0);
    }
}
/*********************************************
        DELY()

Return the displacement in the vertical
direction due to the chain link "code"
*********************************************/
dely(code)

int code;
{

    switch(code){
        case 1: return(-1);
        case 2: return(-1);
        case 3: return(-1);
        case 0: return(0);
        case 4: return(0);
        case 5: return(1);
        case 6: return(1);
        case 7: return(1);
        default:
            printf(" Unknown code0);
            return(0);
    }
}
```