COMPUTER INTERPRETATION OF ENGINEERING DRAWINGS AS SOLID MODELS

Abderrezak KARGAS

Doctor of Philosophy

University of Aston in Birmingham

June 1988

The University of Aston in Birmingham

Computer Interpretation of Engineering Drawings as Solid Models

by Abderrezak KARGAS Doctor of Philosophy June 1988

Summary

Much of the geometrical data relating to engineering components and assemblies is stored in the form of orthographic views, either on paper or computer files. For various engineering applications, however, it is necessary to describe objects in formal geometric modelling terms. The work reported in this thesis is concerned with the development and implementation of concepts and algorithms for the automatic interpretation of orthographic views as solid models.

The various rules and conventions associated with engineering drawings are reviewed and several geometric modelling representations are briefly examined.

A review of existing techniques for the automatic, and semi-automatic, interpretation of engineering drawings as solid models is given. A new theoretical approach is then presented and discussed. The author shows how the implementation of such an approach for uniform thickness objects may be extended to more general objects by introducing the concept of 'approximation models'. Means by which the quality of the transformations is monitored, are also described.

Detailed descriptions of the interpretation algorithms and the software package that were developed for this project are given. The process is then illustrated by a number of practical examples.

Finally, the thesis concludes that, using the techniques developed, a substantial percentage of drawings of engineering components could be converted into geometric models with a specific degree of accuracy. This degree is indicative of the suitability of the model for a particular application. Further work on important details is required before a commercially acceptable package is produced.

2D-3D Reconstruction
Engineering Drawing
Geometric Modelling
Computer-aided Design
Computer Graphics

ACKNOWLEDGEMENTS

I would like to express my thanks to Dr. T.H. Richards and Dr. P. Cooley, for their unflagging supervision and guidance throughout this project.

I am most grateful to the SERC for funding this project, especially to Dr M.A. Sabin and Dr. E. Warman for their constructive and helpful comments and suggestions.

Thanks are due to Pafec Ltd., especially to Dr. Y. T. Lee, for their technical contribution to the project

I would like also to thank my loving wife Zahia Kathryn and my parents for their support.

List of Contents

List of illustrations

<u>CHAPTER ONE</u>

INTRODUCTION

1.1) INTRODUCTION :

It was over twenty years ago that the application of computers to problems in Mechanical Design and Manufacture, also known as CAD/CAM (Computer-Aided Design and Computer-Aided Manufacture), was first recognized [1]. Since that time, much work has been done on the development of computer based systems for the input of product definitions into a computer database, and for the use of mechanical design information in design analysis and manufacture processes.

In the field of database entry, a substantial amount of two-dimensional geometric product-data has been transferred into computer stored files by :

- manual digitization, and sophisticated techniques such as video scanning, of existing engineering drawings,
- computer draughting that allows the designer to interact with the a display via a tablet, or other device, to directly produce and store new drawings of objects, in the classic two-dimensional projections of the edges of a three-dimensional object.

Computer based systems have also been extensively used in the field of design analysis, manufacture and assembly of objects. For example, Finite-element methods may be used for the analysis of heat flow [2]; parts can be checked for interference [3,4]; numerically controlled machine tool tapes can be generated to allow the manufacture of a part [5]; the constraints between objects and mechanisms can be simulated [6]; robot motions to assemble parts may be generated [7,8] and many more engineering applications such as Volume and Mass property computations, Process-planning, and High-realism Displays can be achieved. The software for these applications is extensive and commercially available to the Engineering Industry.

In the early stages of development of such software there was a tendency for each application to require the description of a part in a form that was suitable for that application only. Fortunately, it soon became clear that a solid geometric model was the uniquely versatile form of description which could be exploited for all the above applications.

Clearly, there is need for a bridge between database entry and engineering applications. The gap that needs to be bridged is illustrated in Figure 1.1; on the one hand, much product definition was already stored as two-dimensional information in the form of paper engineering drawings or computer stored files, and on the other hand, there is a wealth of valuable application software that requires three-dimensional volumetric information about the object to be stored as a solid

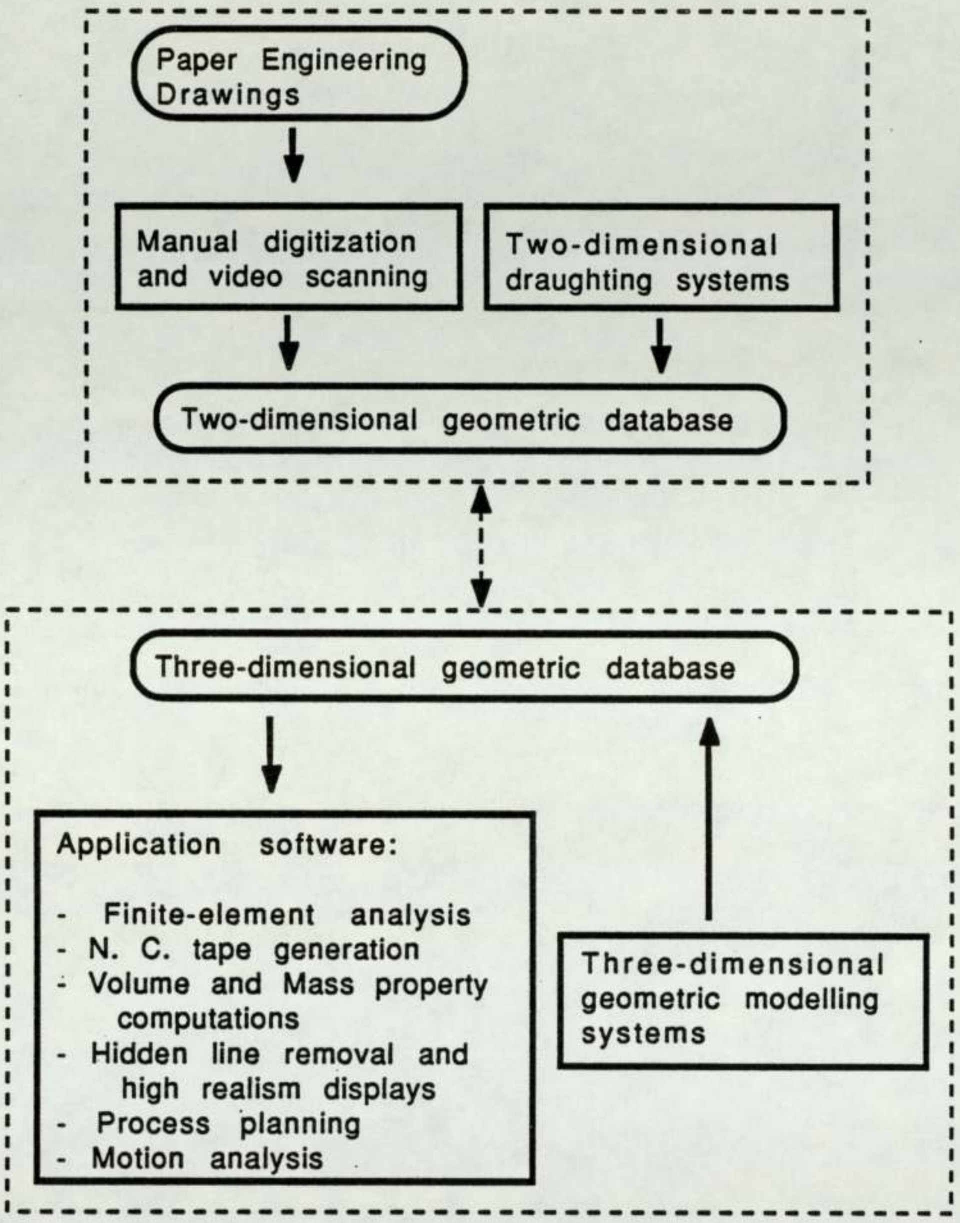

Paper Engineering Drawings
Manual digitization and video scanning
Two-dimensional draughting systems
Two-dimensional geometric database
Three-dimensional geometric database
Application software:
- Finite-element analysis
- N. C. tape generation
- Volume and Mass property computations
- Hidden line removal and high realism displays
- Process planning
- Motion analysis
Three-dimensional geometric modelling systems

Fig. 1.1: The gap between the developments in 2D and 3D CAD systems

geometric model. In attempts to bridge this gap, new geometric modelling tools have been created to enable the designer to generate interactively three-dimensional models. However, these tools can only be used to generate solid models for new products. Furthermore, they are far from easy to use by creative designers who still prefer to develop sketches into engineering drawings, but geometric modelling systems which allow them to do that are not yet commercially available.

It became obvious that in order to bridge this gap and reap the benefits of engineering applications software, it was necessary to develop means by which to interpret the considerable wealth of existing two-dimensional information as three-dimensional geometric models. There are, of course, various ways in which this may be done, but such activity is considered difficult, expensive and unacceptably labour-intensive. For example, in the case of numerically controlled machine tools, the path of the cutter has to be entered interactively over a drawing at a graphics terminal. Hence, there is a need for interpretive software which can process orthographic views of a product and automatically generate a geometric solid model.

Several research workers have recognized such need and have attempted to develop algorithms to "reconstruct" a 3D object from its orthographic projections. A number of techniques have been developed but none has yet been implemented commercially. By adopting a completely novel approach, the author has sought to develop a number of algorithms which automatically interpret a set of orthographic projections as a solid model.

1.2) OBJECTIVES AND SCOPE OF THE PROJECT:

The aim of this work was to develop a number of algorithms and, subsequently, a computer program to interpret an engineering drawing as a solid object, in formal geometric modelling terms. Technically, what is required are algorithms to read a data file that represents the orthographic projections of a mechanical part, process the information, and output a file which defines that object as a formal solid model, as shown in Figure 1.2 .

In essence, the following activities were involved:

- Study and selection of solid geometric representation for the project.
- Review of existing interpretation techniques of engineering drawings as solid objects.
- Development of algorithms and corresponding software for the interpretation process, and

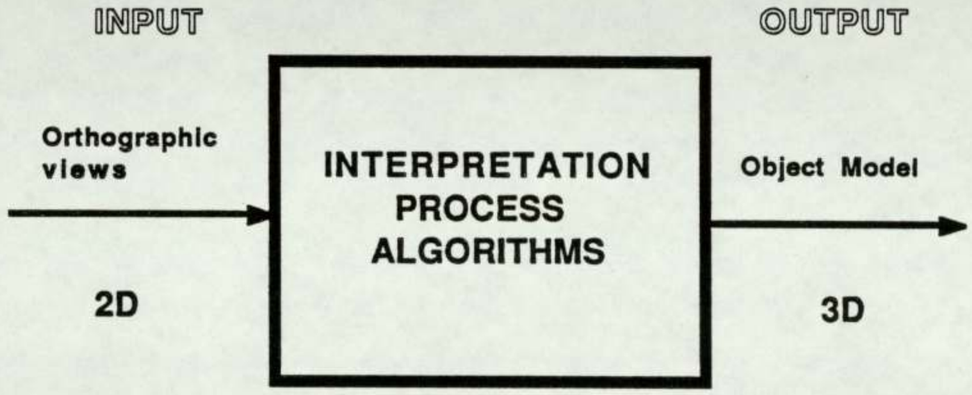


Fig. 1.2: Overall process

implementation on the computer workstation.

- Monitoring of the quality of these transformations by comparison of the input orthographic views with the orthographic views generated directly from a solid modeller, with the emphasis on the complete automation of the process and generation of the complete solid model.

The input data files in which the two-dimensional information is stored, were assumed to exist within a computer system and to represent an assemblage of straight lines and circular arcs.

1.3) <u>TOOLS FOR THE PROJECT</u> :

In the early stages of the project, the environmental hardware consisted of an ICL Perq 2 graphic workstation, illustrated in plate 1 . The computer has one megabyte (Mb) of random access memory (RAM) with built in 8-inch Winchester-type hard disk with a formatted storage capacity of up to 34 Mb and a 1/2 Mb single density (8-inch) floppy disc. The display is a high resolution (768 x 1024 pixels) monochrome portrait monitor. Two RS232 interfaces are also available for serial input and output: one of these ports was used as a link to a VAX 11/750 mainframe in which the solid modeller BOXER (PAFEC Ltd.) was stored; the other port is used to link up a DPX 2000 plotter (Roland DG Ltd.) and an Epson EX-1000 printer

Plate 1: The ICL Perq 2 Minicomputer

Plate 2: The Apollo DN3000 Minicomputer

for hard copy generation. The PNX Operating System - a 32-bit implementation of UNIX - had been installed.

The software has subsequently been transferred and developed further on an Apollo DN3000 workstation, illustrated in plate 2. The Apollo computer has 2 Mb of RAM and a built in 72 Mb Winchester disc (formatted capacity) together with a 1.2 Mb (5.25-inch) floppy disc. The display is a 15-inch bit-mapped, high resolution (1024 x 800 pixels) monitor. Links to the peripherals, such as the DPX2000 plotter and the EX-1000 Epson printer, are provided via an 8-serial port expansion unit. The AEGIS Operating system is used and the complete software of a subroutine version of the solid modeller (BOXER), provided by PAFEC Ltd, has also been installed in the Apollo workstation.

1.4) THESIS PLAN :

The ultimate aim of the work was the automatic interpretation of a set of three orthographic projections of an object stored in the form of two-dimensional information, as a complete and unambiguous solid model.

The initial input to the procedures of the interpretation process is data corresponding to this set of orthographic views. Hence, the subject of chapter Two is Orthographic Projections, where special reference is made to the orthographic views of *Prismatic* objects, i.e. those whose cross-section does not vary with respect to an "axial" direction. The data structure which has been developed for fast storage and retrieval of the data for the input views is also described, with the emphasis on the problem area of interpreting this data into a complete solid model. The latter is the output to the interpretation process and hence, Geometric Modelling is the subject of the next chapter. In chapter Three, the main techniques of geometric modelling are discussed, with special reference *Solid Modelling*, and in particular to the technique known as *Constructive Solid Geometry* which was chosen as the most appropriate for the purpose of the project.

Chapter Four presents a literature survey on the different approaches and techniques that have been developed to interpret orthographic views as solid models.

The author's approach to the problem is discussed in chapter Five which first presents the theoretical foundations of the automatic interpretation process that was developed. An overview of the process, followed by a detailed description of the different stages, is also given.

Several algorithms have been designed and developed to process the input data of the orthographic projections of an object, and automatically output a file which describes the object as a solid model. The transformation process for prismatic and so-called orthoprismatic objects, with the extension to the concept of approximation models for general three-dimensional solids, is also described in this chapter. Some of these algorithms have been designed to be used in conjunction with a commercial solid modeller. The details of all the algorithms are given in chapter Six.

Chapter Seven presents the details of C.I.E.D.S.M. (Computer Interpretation of Engineering Drawings as Solid Models) - the software developed for the interpretation process. Its portability and interface with commercial solid modellers are described. The process is intended to be fully automatic, hence the user interaction with the software has been limited to the input of the system and a brief user guide is given for this purpose.

Examples illustrating the interpretation of prismatic, orthoprismatic, and arbitrary objects, and corresponding results from C.I.E.D.S.M., are given in chapter Eight. In chapter Nine, the project is discussed and some conclusions are drawn, the areas where work remains to be done are identified, and some potential benefits are listed .

<u>CHAPTER TWO</u>

ORTHOGRAPHIC PROJECTIONS

2.1) <u>INTRODUCTION</u>:

Data corresponding to an engineering drawing is the primary input to the interpretation algorithms that are discussed in later chapters. The various rules and conventions associated with engineering drawings are first briefly reviewed to provide a convenient reference for the work which follows.

An engineering drawing conveys a considerable amount of information about the design and manufacture of engineering components. This information may comprise:

i) geometric and topological data in the form of a number of orthographic and auxiliary views of the solid object. These may comprise a number of points or nodes, straight lines, circles, circular arcs and higher order curves.

ii)text in the form of symbols and alphanumerics which indicates dimensioning, tolerances, material, surface finish and other data.

It was clear from the start of the project that some consideration had to be given on whether, or not, to use all the above data as input to the interpretation algorithms. Textual information may have indeed been useful, however, it has been rejected as being beyond the scope of the work for two main reasons:

1) The textual data are not of equal significance but there is no readily available means to distinguish vital information from mere comment.

2) Character recognition algorithms would have been needed in order to extract and make use of such information. Such software was not available for the project and an attempt to develop such algorithms was rejected because it would have been difficult and time consuming.

The input data thus comprise geometric and topological information only. Furthermore, orthographic projections are the only views of the solid object that are considered. These may comprise straight lines and circular arcs, and are assumed to be stored in the computer memory. A simple and efficient data structure, discussed in section 2.3, has been developed to provide fast storage and retrieval for the input orthographic views.

A preliminary analysis of a number of data structures corresponding to several sets of orthographic views, has led to the classification of all objects into two main classes: *prismatic* and *non-prismatic* objects. These are discussed in section 2.4.

The nature of the problem in the interpretation of orthographic views as solid objects is discussed in the last section of this chapter.

2.2) PROJECTION CONVENTIONS:

A number of orthographic views are usually used to represent a solid object on an engineering drawing, and the identity between the solid body and the views can be established only if certain rules are observed. The following section describes some of the rules used to generate these views.

In practice, orthographic projections of a solid object are generated using systems of parallel projectors from its boundaries onto a number of planes. The projectors are normal to these planes. Often, only two planes are required and they are known as the principal planes of projection. One is horizontal and the other vertical. Four quadrants or angles are produced by the intersection of these planes, as shown in Figure 2.1. The object to be drawn is placed in one of these angles and the orthographic views of it are projected onto the planes. The orthographic projections that are widely used are produced using the First and Third Angles, illustrated in Figures 2.2(a) and 2.3(a), respectively. In both systems, the view on the vertical plane is called the elevation and the view on the horizontal plane is called the plan. To obtain these views as they appear in an

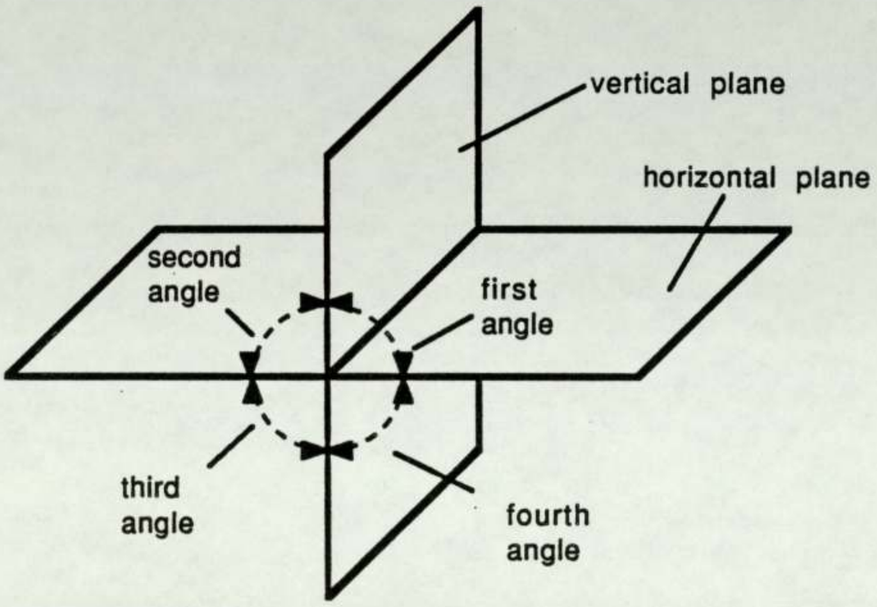


Fig. 2.1: Principal planes of projection

engineering drawing, as shown in Figures 2.2(b) and 2.3(b), the horizontal plane is rabatted about the intersection of the planes or ground line. It can be seen that the projectors cross the ground line, at right angles.

An elevation and plan of an object are not always sufficient to describe it completely. In such a case a third view, called an end or side elevation, is drawn on another vertical plane which is perpendicular to both principal planes, as shown in Figure 2.4(a). The equivalent views which would appear on an engineering drawing are obtained by rabatting this vertical plane with the horizontal plane, as shown in Figure 2.4(b). The plan, elevation and end views are also commonly known as the top, front and side views, respectively. It can be seen that there are a number of relationships between a point and its projections in the adjacent views. For instance, the vertex v in the top view, and its projection v' in the front view are located on a projector line normal to the X-axis; the vertex v' in the front view, and its projection v" in the side view are located on a horizontal projector line perpendicular to the Z-axis. These relations are independent of the system chosen.

Two or three views are usually adequate to represent a simple object. For more complicated objects, such as those which have complex inclined faces,

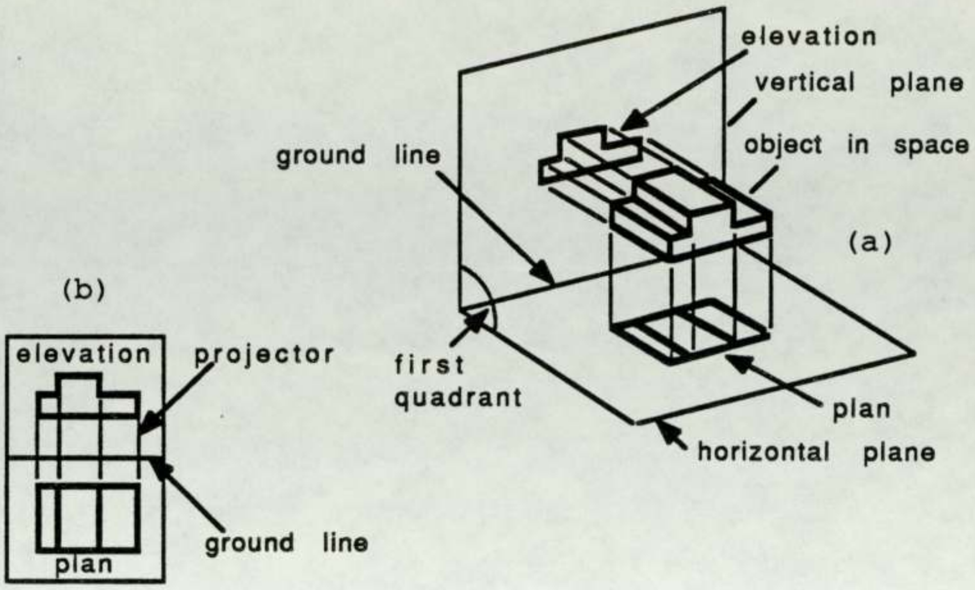


Fig. 2.2: First angle orthographic projection system

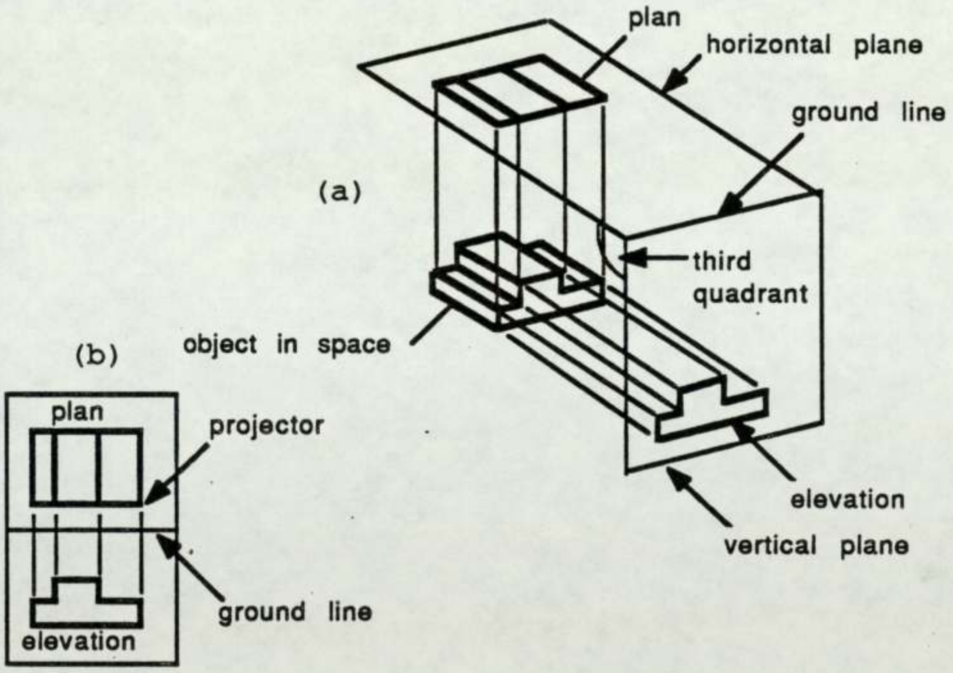


Fig. 2.3: Third angle orthographic projection system

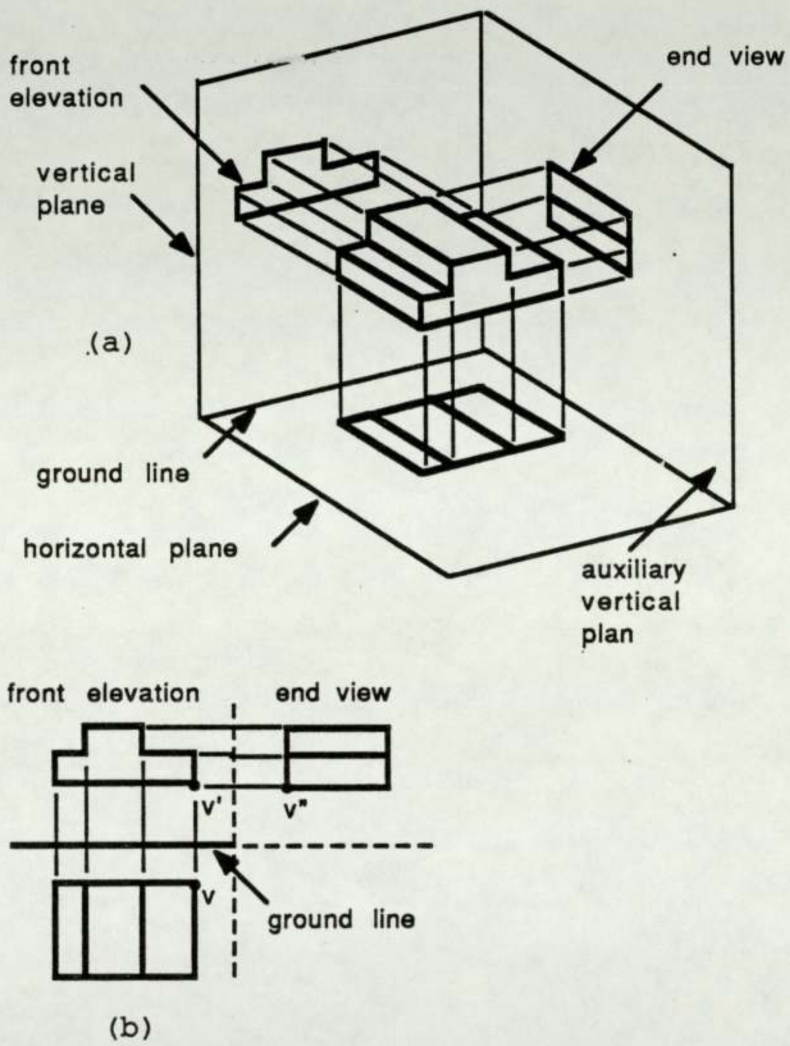


Fig. 2.4: Three orthographic views projection system

additional views may be necessary. These are drawn on auxiliary planes inclined to the principal planes and are called auxiliary views. The same principle of parallel projectors, normal to the plane, are used. Cross-sectional views are also commonly used to define the interior of an object when required.

Apart from the projection system and number of views, there are other conventions that are important to this work. These are:

i) the drawing of additional lines referred to thereafter as 'tangency edges'. These artificial lines which are not normally drawn in engineering drawings, are used to represent the edge where a curved surface is tangent to a plane surface.

ii)the type of lines, or line-styles, used to draw the views of the object. Some of the line-styles that have been recommended by the British Standards Institution in B.S. 308: Part 1: 1984, are as follow:

- continuous thick lines should be drawn for the visible outlines of the object
- continuous thin lines must be used for projection or extension lines, hatching or sectioning.
- hidden detail lines must be made up of short thin dashes

- centre lines must be thin long chain lines

- thick long chain lines should be used for cutting planes or section planes

- irregular boundaries and short break lines should be drawn using thick continuous wavy lines.

As far as the work developed in this project is concerned, the following conventions have been adopted:

- the first angle projection system

- three orthographic views are required

- the attribute which defines an edge as 'visible' or 'hidden', is not required, hence, the type of lines does not have to be specified. This has the advantage of minimizing the amount of input data to the interpretation algorithms.

2.3) <u>**DATA STRUCTURE**</u>:

At the lowest level, a single view of an engineering drawing may be regarded as an assemblage of unordered segments, or edges. For orthographic views accepted as input for this project, these segments may consist of straight lines or circular arcs. In the data structure developed, each segment is stored and represented by :

- its type, which determines whether the edge is a straight line or circular arc,

- a pair of points or nodes representing the start and end points of line (except for complete circles for which the start and end nodes are the same point). Each node, as well as each centre of arc or circle, is specified by a pair of coordinate values.

Figure 2.5(a) shows the front, or XY view of a solid object and Figure 2.5(b) shows a structure for the data items defining that view. These data are referred to below as the 'initial data'. Table A in Figure 2.5(b) contains the number, the type and the start and end nodes of all the edges. The type is set to zero for a straight line, or to an integer signed according to the sense of rotation of an arc: clockwise is positive; the integer value is used as a pointer to specify in table C, the storage location of the coordinates of the centre of the corresponding arc. Table B contains the coordinates of all the nodes in that view. For instance, edge number 1 is a clockwise arc which starts at node number 2 and finishes at node number 4, and the x and y coordinate values of its centre are 6.00 and 4.00, respectively; nodes 2 and 4 are located at (6.00, 3.00) and (5.00, 4.00), respectively.

The initial data are the only required input to the interpretation algorithms. Furthermore, the user does not need to specify nor enter edges in any prescribed order.

(a)

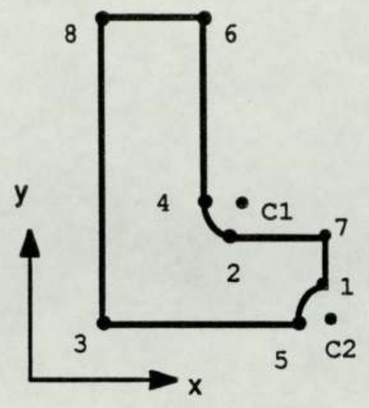


Table A: Topology

Edge Number	Start Node	End Node	Type
1	2	4	1
2	7	1	0
3	3	8	0
4	1	5	-2
5	4	6	0
6	5	3	0
7	8	6	0
8	7	2	0

Table B: Nodal Coordinates

Node	X	Y
1	9.0	2.0
2	6.0	3.0
3	1.0	1.0
4	5.0	4.0
5	8.0	1.0
6	5.0	9.0
7	9.0	3.0
8	1.0	9.0

Table C: Arc centre Coordinates

Centre No.	X	Y
C1	6.0	4.0
C2	9.0	1.0

Tables B & C : Geometry

(b)

Fig. 2.5: a) Object XY view
b) Data structure

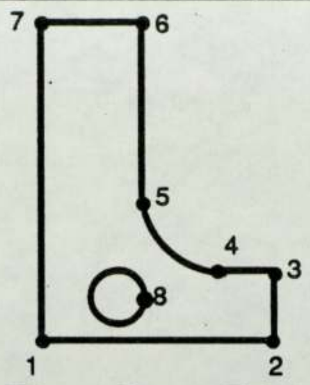
At a higher level, a single view of an engineering drawing may be regarded as a graph and is best described using the terminology of Graph Theory. Some of the basic concepts of Graph Theory are given in appendix A. A graph is represented by the connections between its elements, and in the case of a view of an engineering drawing, these connections exist between edges and nodes.

Formally, a graph may consist of closed paths or circuits, Multiple Edges, and/or Loops. In an engineering drawing, the projections of three-dimensional surfaces are closed contours which can be described as circuits, Multiple Edges or Loops. A complete circle is the only way that a loop which consists of a single edge, can occur in an engineering drawing, since the start and end points are the same node.

On this basis, the user's initial input is converted into a number of circuits (or closed paths, loops or contours). These contours are the basic elements that are processed by the algorithms described below. In order to guarantee that only closed paths will be processed, it is necessary to check that the user did not input nodes which belong to one edge only, except for Loops. The case may also arise if tangency edges have not been included in the input orthographic views. The check is simple and is carried out by counting the number of times each node appears in the data structure. In Figure

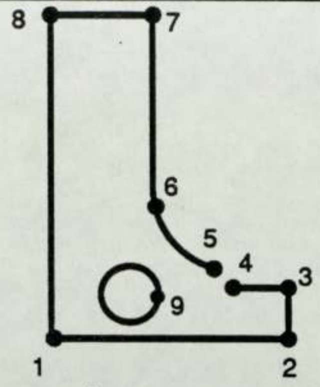
2.6 where all the contours are identified as closed, each node appears more than once in the data structure while in Figure 2.7, each of node numbers 4 and 5, appears only once indicating that there will be an open path which does not bound an area in the corresponding view. In that case the user is immediately informed that the orthographic view can not be accepted as input, and is prompted to enter the correct data. The initial data are also checked for self-intersecting loops to ensure that all the nodes are included in the input data. This is achieved by examining all the intersections between edges in the view. If a valid point of intersection is found but not entered as a node then a node is generated automatically at that point and the data updated by dividing the intersecting segments into pairs of segments. Figure 2.8(a), shows a view where a self-intersection exists between two loops. The points of intersection are computed, numbered and stored in the updated data structure, as shown in Figure 2.8(b), where edge numbered as 3 has been divided into three new edges: 3, 9 and 10.

When examining a view of an engineering drawing, it is possible to distinguish different types of contours. Some may be isolated (or disjoint) from all the other contours, and some may be connected to others. An algorithm developed in this project and described in later chapters, has been designed to determine which contour is isolated and which is connected. Moreover, the same



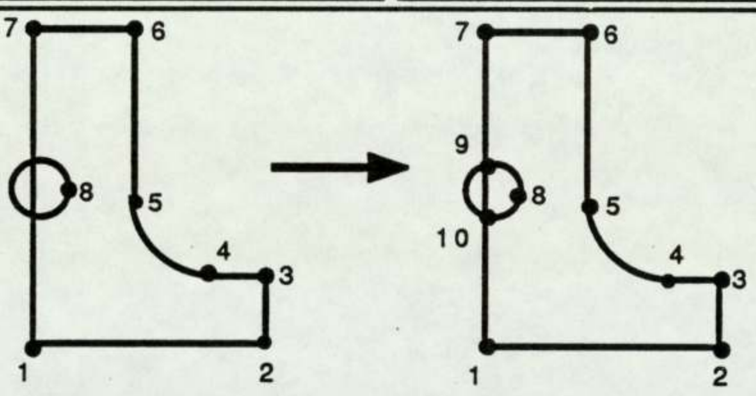
Edge	Start	Finish
1	1	2
2	5	6
3	8	8
4	7	1
5	3	4
6	2	3
7	4	5
8	6	7

Fig. 2.6: Object view and topology



Edge	Start	Finish
1	1	2
2	5	6
3	9	9
4	8	1
5	3	4
6	2	3
7	7	8
8	6	7

Fig. 2.7: Checking for closed loops



Edge	Start	Finish
1	1	2
2	5	6
3	8	8
4	7	1
5	3	4
6	2	3
7	4	5
8	6	7

(a)

Edge	Start	Finish
1	1	2
2	5	6
3	8	9
4	7	1
5	3	4
6	2	3
7	4	5
8	6	7
9	9	10
10	10	8

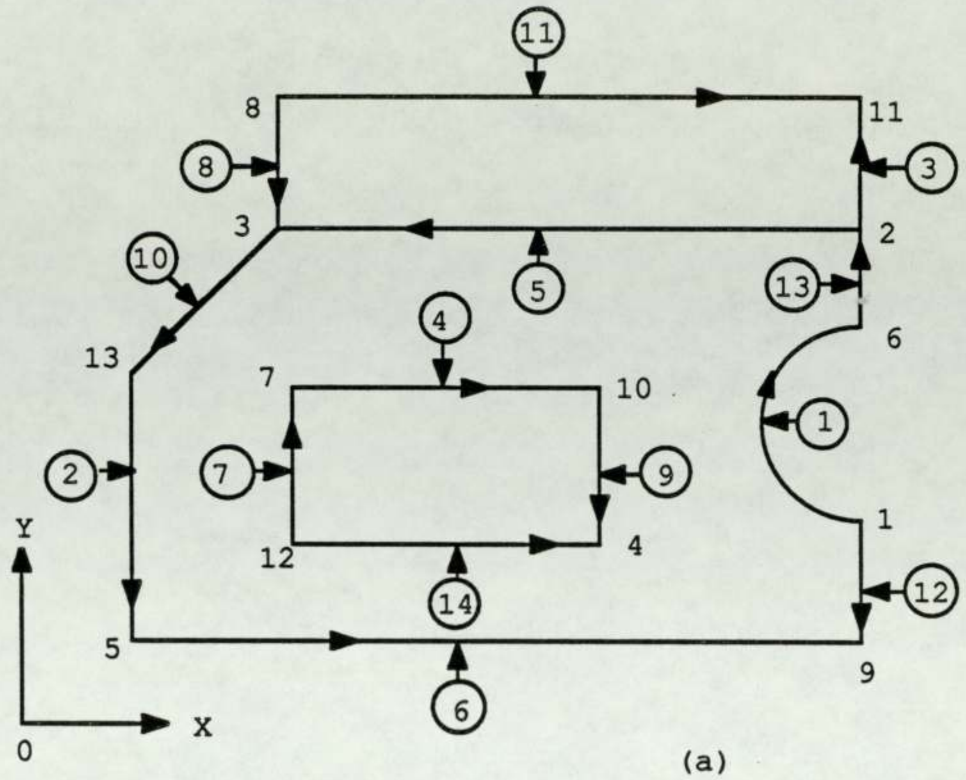
(b)

Fig. 2.8: a) A self-intersecting loop
b) Updating the topology

algorithm is used to determine the boundary or perimeter contour in any view. Figure 2.9(a), shows a view that comprises both connected and disjoint closed contours, where the positive sense of each edge is indicated by an arrow pointing from the start node towards the end node; an edge which is traversed in the opposite sense is indicated by a negative integer. The type of each loop is represented by an integer value which is set to zero to indicate a 'disjoint' attribute, or to 1 to indicate a 'connected' attribute. The number and type of contours in the xy view shown in Figure 2.9(a) are shown in Figure 2.9(b), where loop L3 and L4 represent in effect the same loop which is traversed in both directions. These contours are represented and stored by specifying the number and types of edges which define it. Direct access files are used to store such data. The structure of these files is described in chapter 7.

2.4) <u>ORTHOGRAPHIC VIEWS OF PRISMATIC AND ARBITRARY OBJECTS</u>:

In a preliminary analysis of the input data, it has been found that, among the vast range of mechanical components manufactured in the engineering industry, there exists a class of simple objects which can readily be identified from their orthographic views. These solid objects, known as prismatic objects, are those having a



(a)

View	Loop No	Edge Nos	Type or 'label'
XY	L1	6, -12, 1, 13, 3, -11, 8, 10, 2	connected or '1'
	L2	-2, -10, -5, -13, -1, 12, -6	connected or '1'
	L3	-4, -7, 14, -9	disjoint or '0'
	L4	4, 9, -14, 7	disjoint or '0'
	L5	5, -8, 11, -3	connected or '1'

(b)

Fig. 2.9: a) XY view of an object
b) Number and type of loops in the above XY view

fixed cross-section in at least one direction. Any other object which does not belong to this class of objects is hereafter referred to as an arbitrary or non-prismatic object.

The class of prismatic objects may be divided further into two distinct subclasses: simple and complex. A complex prismatic has one or more holes drilled through it, while a simple prismatic object has none.

The class of any object, as defined above, is determined by the number of loops, as well as the type and shape of each loop, contained in each view of the object.

Prismatic objects play an important role in the interpretation process developed in this project. This role is discussed in later chapters. Therefore, the ability readily to identify a prismatic object from its orthographic views is considered here as one of the important milestones in this work.

2.4.1) **ORTHOGRAPHIC VIEWS OF PRISMATIC OBJECTS**:

A simple prismatic object is shown in Figure 2.10(a). It can be seen from the orthographic projections, Figure 2.10(b), of such object that there will always be at least one view consisting of a single closed loop only.

This loop represents the boundary or perimeter loop in that view. Such a view is defined here as the *base* view. Furthermore, the remaining views may consist of one or more connected rectangular loops whose nodes also belong to the perimeter loop.

The simple prismatic object shown in Figure 2.10(a), may be transformed into a complex one (having a multiply-connected cross-section) by drilling holes through it, as shown in Figure 2.11(a). In this case, the base view consists of two or more closed disjoint loops. Again, the two views adjacent to the base view comprise a number of connected rectangular loops whose nodes also belong to the perimeter loop, as shown in Figure 2.11(b).

2.4.2) **ORTHOGRAPHIC VIEWS OF ARBITRARY OBJECTS**:

In the case of arbitrary or non-prismatic objects, the orthographic projections may comprise any number, type and shape of loops. Those features which identify a prismatic object are not found in the views. Figure 2.12(a), shows an arbitrary solid object. In the three orthographic projections of this rather simple object there is at least one view, in this case the xy view, that has a single closed loop, as shown in Figure 2.12(b). This may be regarded as a 'base' view; this is one feature found in the views of prismatic objects. However, the

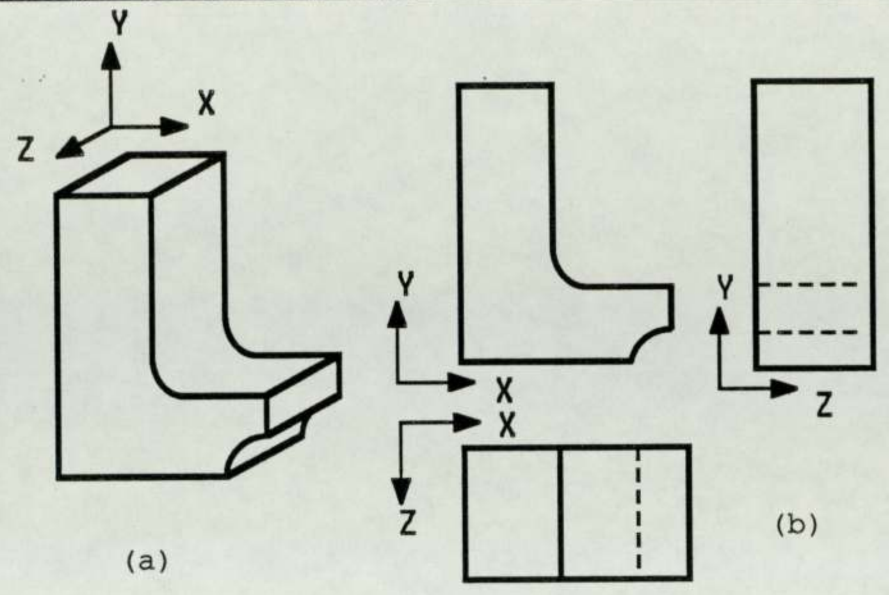


Fig. 2.10: a) A simple prismatic object
b) its orthographic views

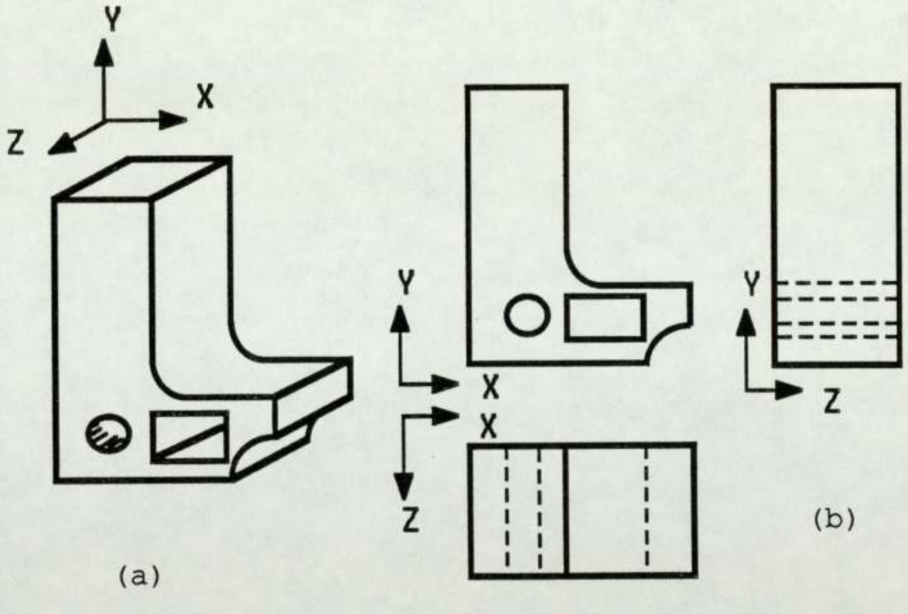


Fig. 2.11: a) A complex prismatic object
b) its orthographic views

object will not be defined as a prismatic object because the views adjacent to the 'base' comprise loops that are not rectangular.

Another example is illustrated in Figure 2.13(a). In this case, a view consisting of two closed disjoint loops exists amongst the set of orthographic views in Figure 2.13(b), and may be considered as the 'base' view of a prismatic object; however, because the loops contained in the remaining views do not share their nodes with the perimeter loops in the corresponding views, the object is identified as non-prismatic.

2.5) <u>INTERPRETING PROJECTIONS AS SOLID OBJECTS</u>:

The process of generating orthographic views is rather straightforward (see section 2.2), and has been implemented on computers for a relatively long time. However, the generation of solid models from orthographic views by machines has not yet been achieved because of the complex nature of the process involved. To appreciate the complexity of such a problem it is necessary to understand the process by which a human interprets three views as a solid object.

The engineering drawing has been a successful form of communication only because of the intelligent

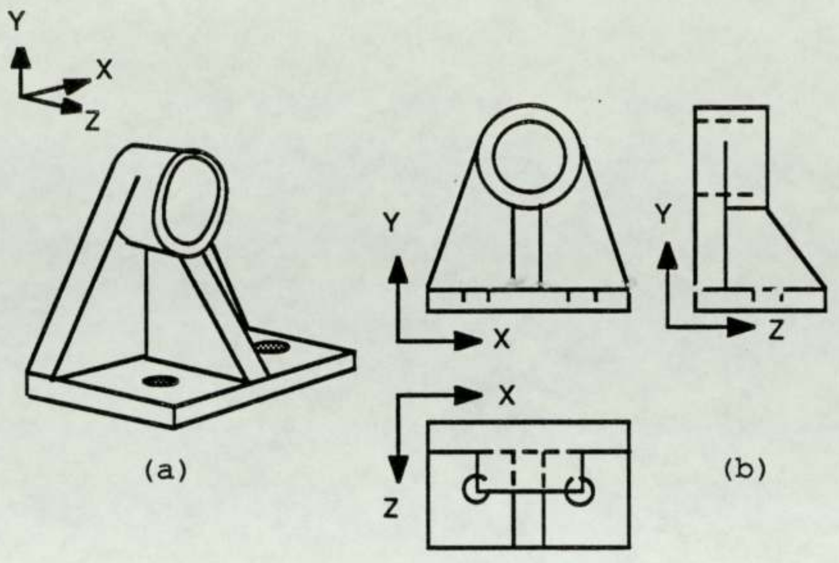


Fig. 2.12: (a) A general 3D object
(b) its orthographic projections

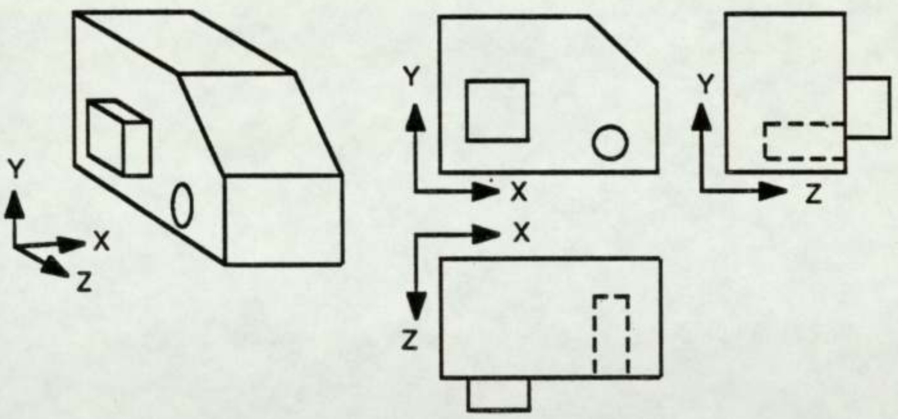


Fig. 2.13: The XY view of this arbitrary object is similar to the base view of a complex prismatic object.

interpretation which is applied to it. The drawing can only communicate a precise description of a component when both the draughtsman and the user are well-versed with the implicit information it conveys, such as the conventions and mechanisms by which the two-dimensional structural elements have been generated. The draughtsman knows that the inferences he makes will only be correctly understood by a user who has a knowledge of this implicit information.

Furthermore, apart from using his knowledge and experience, a human also has the ability to process a large number of two-dimensional entities from the two-dimensional views, such as faces, edges and vertices, simultaneously and qualitatively. Partial solids are generated from the corresponding parts of three views, and if they agree with his experience and knowledge, they are composed to form a complete solid object; if inconsistencies are discovered, then there is a return to the three views and a generation a different solid. Therefore, the process has certain characteristics which can be described as follows:

a) More empirical than logical, since it is based on past experience rather than deduction,

b) entities are grasped qualitatively,

c) the processes proceed in parallel,

d) the feed back is continuous.

In contrast, available mini-computers and, probably, those on which any commercial software would be mounted, operate sequentially. Therefore, it is not feasible to construct a system which generates a solid model from three views by emulating human thought-process. Such a system would therefore require the processing of automatic three-dimensional interpretation not only to be formulated in a logical (serial) manner, but also to involve some degree of expertise to represent the knowledge that both draughtsman and user have gained.

Technically, the problems of automatically interpreting orthographic views as a solid object are translated as a loss of semantics occurring when the object is represented with a two-dimensional description. For instance, one line in any view can represent more than one edge; moreover, some lines do not represent true edges such as silouhette lines which are used to represent curved surfaces. For this reason, the straightforward approach of matching each line to the lines of the other views to construct a set of edges and faces, may lead to the generation of nonexistent or ghost faces, and hence impossible objects. Orthographic views are described using a 'wireframe' representation, and for this reason it may be possible for a set of three orthographic views to have several solutions, i.e. to represent the two-dimensional description of several objects.

In order to avoid these problems, it is necessary to select a three-dimensional output data structure that guarantees the representation of a valid and unambiguous object model. Furthermore, it is desired that this data structure must explicitly describe the output model in terms of a solid volumetric representation as required for design analysis and manufacture processes. The main three-dimensional representation techniques are described in the following chapter, with special reference to the one selected for the project.

CHAPTER THREE

GEOMETRIC MODELLING

3.1) INTRODUCTION :

A central activity in the Computer-Aided Design (CAD) process is the evolution of a comprehensive representation, or *geometric model* (also called *product model)* of a designed object. The designer gives concrete form to his ideas by building a model in dialogue with a computer. The model is then developed and optimized by design analysis: a set of calculations and simulations to predict the properties and behaviour of the object. The model may also be used for the preparation of manufacturing processes. Therefore, the role of *modelling* in CAD can be summarized as a foundation of the design cycle of synthesis, representation, analysis and optimization.

By definition, *geometric modelling* is the computer-aided input, representation, interrogation and display of the shape of three-dimensional (3D) objects. It is a collection of methods used to create data structures and algorithms for representation and calculation of data on the shape of 3D objects. Formally, the representations are mainly defined in terms of Geometry (point coordinates, curve and surface equations,...), and Topology (connections between points, edges and faces).

In the field of geometric modelling representations, several authors distinguish two main subareas: *surface* modelling and *solid* modelling. However, it is recognized here that *wireframe* modelling also deserves to be mentioned, since it is widely used by several geometric modelling systems, and it also has a very important historical value. For this reason, it is presented first in the following sections, as an introduction to the subject of geometric modelling representations. Excellent surveys and reviews on the fundamentals of geometric modelling representations and their potential can be found in the literature [9,10].

3.2) WIREFRAME MODELLING :

Wireframe modelling was first used in the early two-dimensional (2D) drafting systems to represent simple 2D designs such as for circuit diagrams and printed circuit board (PCB) layouts. The wireframe model consisted of lists of points and lines in 2D space.

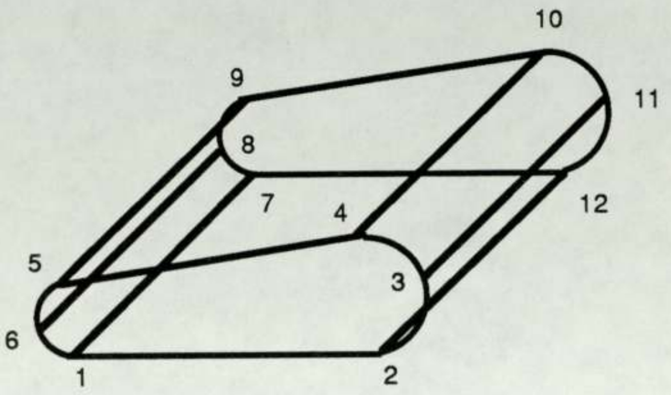
In recent mechanical engineering drafting systems, objects may be displayed in one or more orthogonal views. These views, as 2D wireframe models, are independent of each other, and thus can be incompatible, in which case there is no true representation of a 3D object.

Compatibility of views has been guaranteed with the introduction of 3D wireframe models which allow several views to be derived from a single representation. A 3D wireframe model still has the same data structure but in 3D space; it consists of a set of vertices and a set of edges which indicate the interconnection between vertices. Each vertex is specified by its position in space in terms of (x, y, z) coordinates, and each edge is defined in terms of its two end vertices. This data structure is very simple and can be represented by two arrays: a 3-column geometry array of real numbers to store the coordinates of all the vertices, and a 2-column topology array of integers for edge definitions. Figure 3.1 shows a 3D wireframe model and the associated data structure.

This modelling technique owes its well-established use in several commercial CAD systems to the simplicity and efficiency of data storage. The main advantages can be illustrated as follows:

- geometric entities (vertices and edges) can be retrieved and updated quickly
- model creation and display are fast
- computer requirements, such as storage capacity, are low.

However, the representation has severe limitations which are mainly due to the lack of geometric completeness and loss of surface and volume information. These are as follows:



Segment No.	Start node	End node
1	1	2
2	4	5
3	9	10
4	7	12
5	1	7
6	2	12
7	5	9
8	4	10
9	6	8
10	3	11
11	1	5
12	2	4
13	7	9
14	10	12

Fig. 3.1: A simple wireframe model and its data structure

- there can be ambiguity and loss of definition as illustrated in Figure 3.2, which shows a wireframe model and three possible interpretations

- input of a large amount of low-level data is required to define even simple objects

- the visualization of a complex wireframe model may be impaired and may lead to confusion

- impossible (invalid) objects, as illustrated in Figure 3.3, may be generated

- mass and volume property computations cannot be carried out

- sectioning and hidden line removal cannot be generated automatically

- in the case of objects with curved surfaces, silouhette lines (profile edges) cannot be adequately represented, as illustrated in Figure 3.4.

3.3) SURFACE MODELLING :

Surface modelling overcomes some of the above problems of wireframe modelling since it provides more information describing the surface of an object. It is concerned with mathematical methods for description of all kind of surfaces. These may be simple flat plane models created between pairs of parallel straight lines or may be much more complex surfaces, often referred to as *free-form* or *sculptured* surfaces. It also deals with operations on

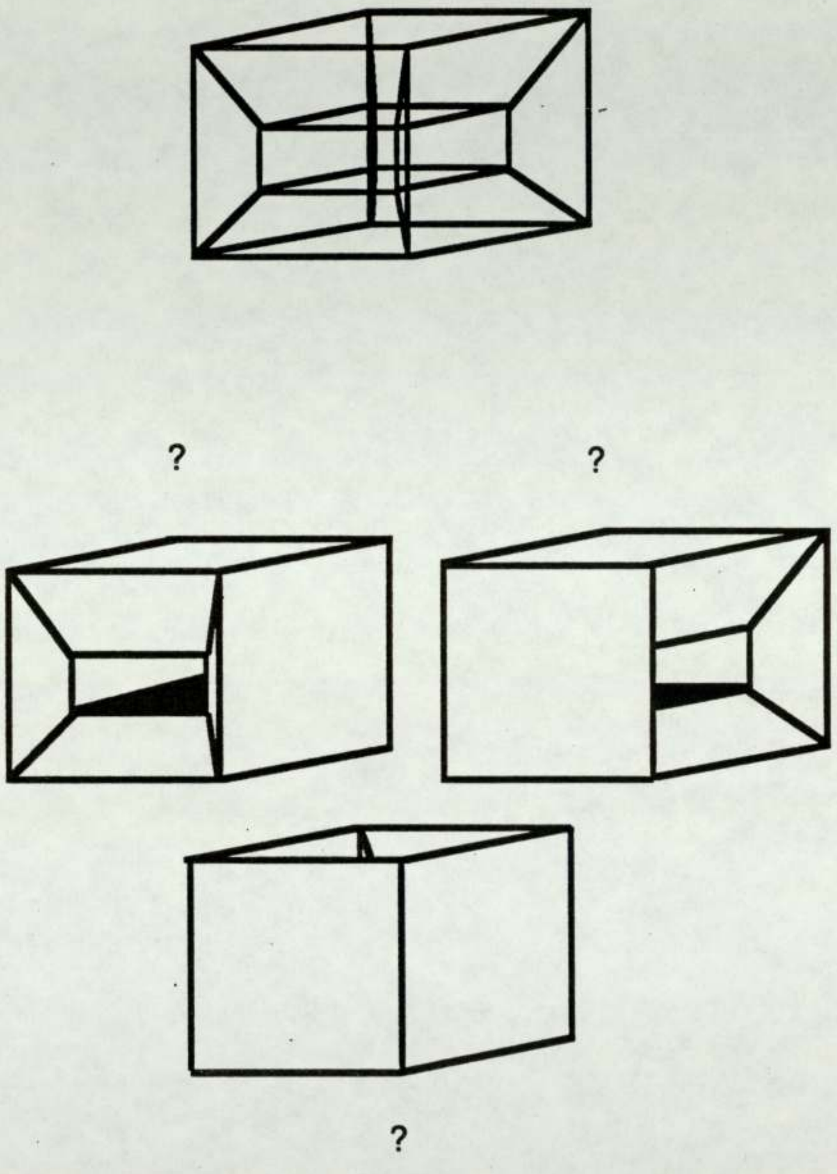


Fig. 3.2: a) A wireframe model and
b) three possible interpretations

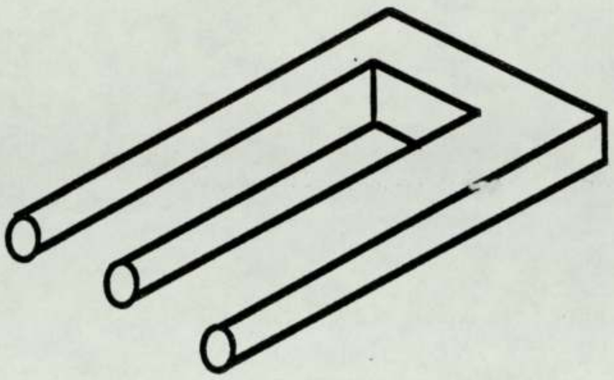


Fig. 3.3: Validity of a model
(the devil's fork)

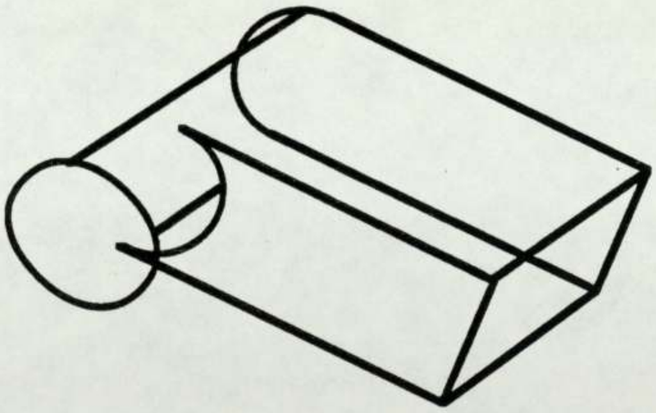


Fig. 3.4: Profile edges in a wireframe
model with curved surfaces

these surfaces, such as intersections and modifications.

Surface modelling was first introduced to replace lofting techniques used in design of bodies such as ship's hulls, turbine blades, aircraft and car panels. This modelling technique has since been developed to the extent that the theoretical background has become a new field of study known as *computational geometry* [11], that uses methods from matrix and vector algebra, differential geometry and approximation theory.

One of the earliest techniques of surface modelling was developed by Fergusson [12], and was known as the Fergusson Patch. Mathematical definitions of curves and surfaces were made possible by using *parametric* rather than Cartesian co-ordinates, and transformations could easily be carried out using matrix algebra.

Basically, three-dimensional surfaces may be formulated by interpolation or approximation of two or more *parametric* space curves; a parametric representation of a curve is given by:

$$\mathbf{r} = \mathbf{R(u)}$$

where $\mathbf{r}$ is the position vector (x y z) of a point on a curve described by the vector function:

$$\mathbf{R(u)} = [\ \mathbf{X(u)}\ \ \mathbf{Y(u)}\ \ \mathbf{Z(u)}\ ]$$

A segment on the curve is then described on some closed interval $a \le \mathbf{u} \le b$, and is usually defined in terms of data at a number of points; Figure 3.5 shows a parametric curve segment described on the interval:

$$t_1 \le \mathbf{u} \le t_2$$

For surfaces, two parameters are required:

$$\mathbf{r} = \mathbf{R(u,v)}$$
$$\mathbf{R(u,v)} = [\ \mathbf{X(u,v)}\ \ \mathbf{Y(u,v)}\ \ \mathbf{Z(u,v)}\ ]$$

where parameters **u** and **v**, may take on values in a specified range, usually 0 to 1. Figure 3.6 illustrates the parametric description of a three-dimensional surface.

Several schemes of interpolation or approximation have been used in the design of complex surfaces, most of which were based on cubic and rational polynomial segments [13,14,15], and others involve higher degree polynomials.

The major advantage of surface modelling is the ease with which complex surfaces may be generated. This technique also allows fast local, and global, modifications to be carried out and needs only a small amount of data storage. There are however, some disadvantages:

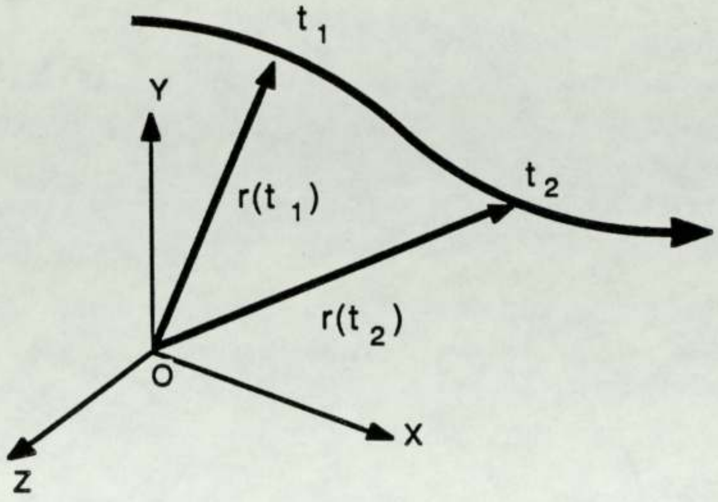


Fig. 3.5: Parametric description of a curve

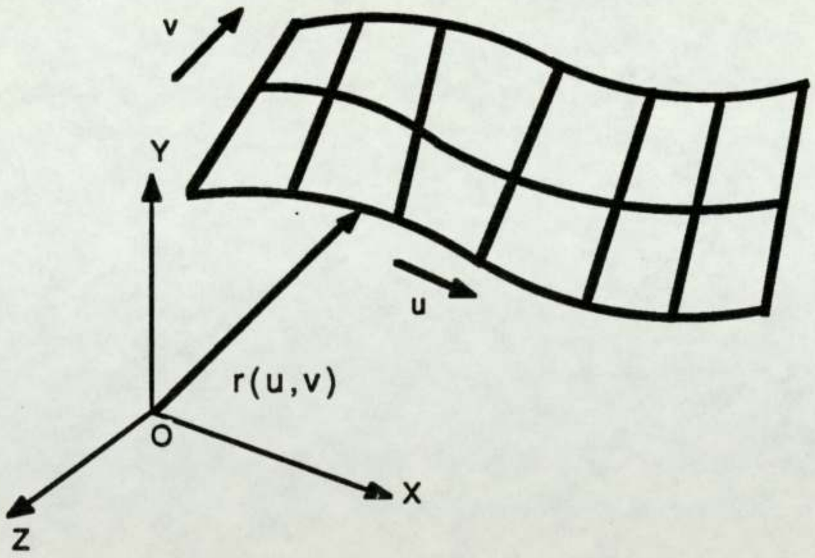


Fig. 3.6: Parametric description of a surface

- the generation of unambiguous models is not ensured because of possible lack of connectivity between surfaces

- calculation of intersections between sculptured surfaces is complex

- mass property calculations are limited to single surfaces

- interference between surfaces is not guaranteed and relies on user detection.

3.4) <u>**SOLID MODELLING**</u> :

Solid modelling deals with data structures for informationally complete and unambiguous description of solid objects, and algorithms operating on these data structures. This modelling technique has been developed to overcome the limitations mentioned above. Solid modelling has geometric completeness, thus it allows the automation of several engineering applications. The technique was pioneered in Britain, by Braid [16,17,18]. Extensive developments followed and several solid geometry modelling representations have been developed. In this section, the four main techniques are discussed: *Cellular Decomposition, Sweeping, Boundary representation* and *Constructive Solid Geometry* [19,20].

3.4.1) CELLULAR DECOMPOSITION :

In this scheme objects are represented by a collection of disjoint cells. There are several variants of cellular decomposition. The simplest variant is known as *Spatial Enumeration*, illustrated in Figure 3.7 . In this, space is divided into a large number of equally-sized cubes, or *voxels*, that are positioned in a fixed and regular three-dimensional grid. The object is represented by the voxels in which it resides. Each cube is marked or 'enumerated' as 'inside' or (1) if it lies inside the object, and as 'outside' or (2) if it is outside the object. For voxels that are partly inside and partly outside the object, a decision is made on the basis of whether the centre of the cube is inside the object.

The data structure of spatial enumeration is very simple. It consist of a three-dimensional Boolean array, where each voxel is represented by one element which indicates whether it is inside or outside the object.

It is clear that this scheme is well adapted to applications such as the computation of mass and volume properties of the object it represents. The other advantage of spatial enumeration lies in its simplicity. However, most objects, notably those with curved surfaces, can only be approximated at the boundaries, resulting in an inaccurate and jagged representation. To provide any

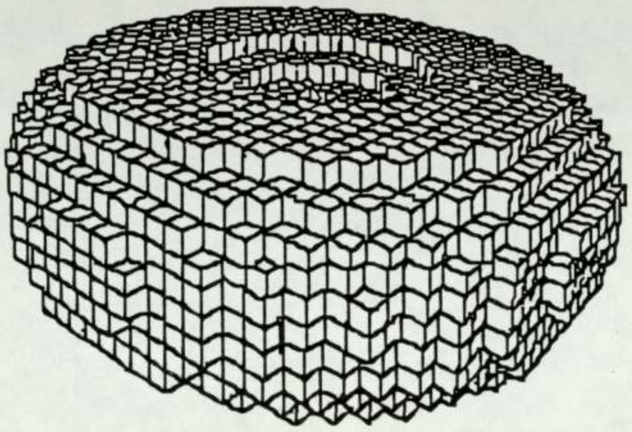
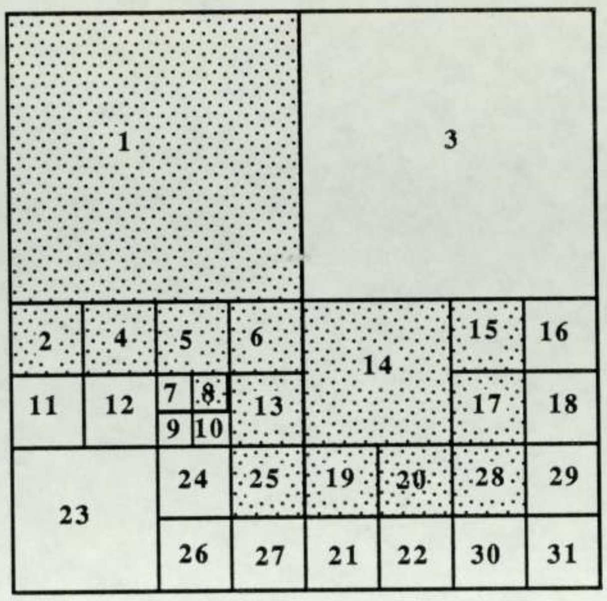


Fig. 3.7: Spatial Enumeration

reasonable geometric resolution and close approximation, it is necessary to use very small voxels which requires an excessive amount of memory; for instance, a grid of 10^3 by 10^3 by 10^3 voxels would require more than 100 Megabytes of memory!

A more recent development amongst spatial enumeration schemes [21] attempts to reduce the amount of memory required by only using small cubes where fine resolution is needed, such as the boundary. A coarse grid of cubes is used everywhere else. In this method, known as *octtree*, the model space is first divided into large cubes, and these may be marked as completely inside the object (full), or completely outside the object (empty), or partially inside and partially outside the object (partially occupied). Partially occupied cubes are then subdivided into eight smaller cubes of equal size and these again are marked, which can lead to further subdivisions. This process of subdivision continues recursively for partially occupied cubes until either the object is represented exactly, or until a predefined minimum size of the cubes is reached. This technique is similar to the quadtree representation in two-dimensional space which is illustrated in Figure 3.8 .

The data structure of an octtree representation is a tree where each node is a record of the state (inside, outside or partially inside and partially outside) of the



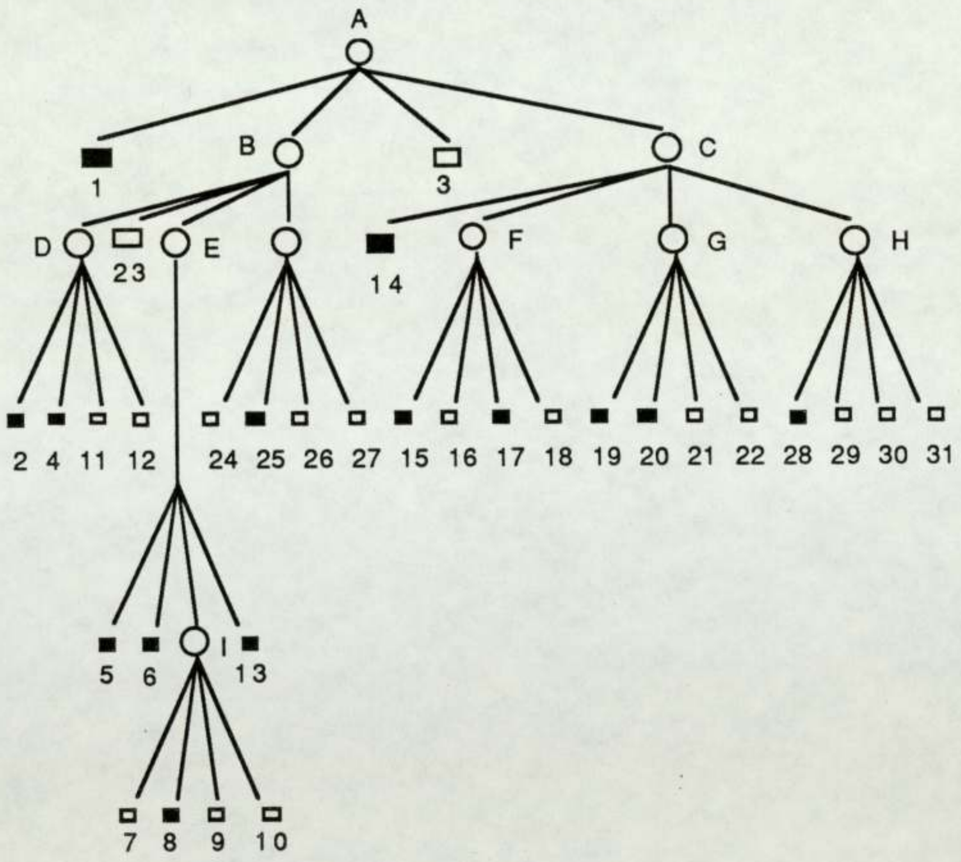


Fig. 3.8: The quadtree representation

cube residing at that node. That record also holds pointers to the eight cubes into which the cube is subdivided. This structure is illustrated in Figure 3.9, where the cube, enumerated as 0, has been subdivided into eight more cubes and the cube 4 is marked as outside of the object. Figure 3.10 shows an alternative data structure, known as *linear octtree* [22]. In this method, only the information about the cubes that are inside the object is stored. An octal code is used for all these cubes - the octal code is a sequence of numbers between 0 and 7 - and the length of the code depends on the size of the cube: for every further subdivision, an extra number is taken. The number depends on the cube concerned. The complete linear octtree data structure is also condensed to save memory space.

Although the data structure of the octtree representation is somewhat more complicated, the advantages are similar to those of spatial enumeration. The disadvantages are also the same except that the octtree takes less memory than the tree data structure of spatial enumeration.

Other variants of cellular decomposition with cells of different shapes are also used, in particular for application such as finite-element analysis.

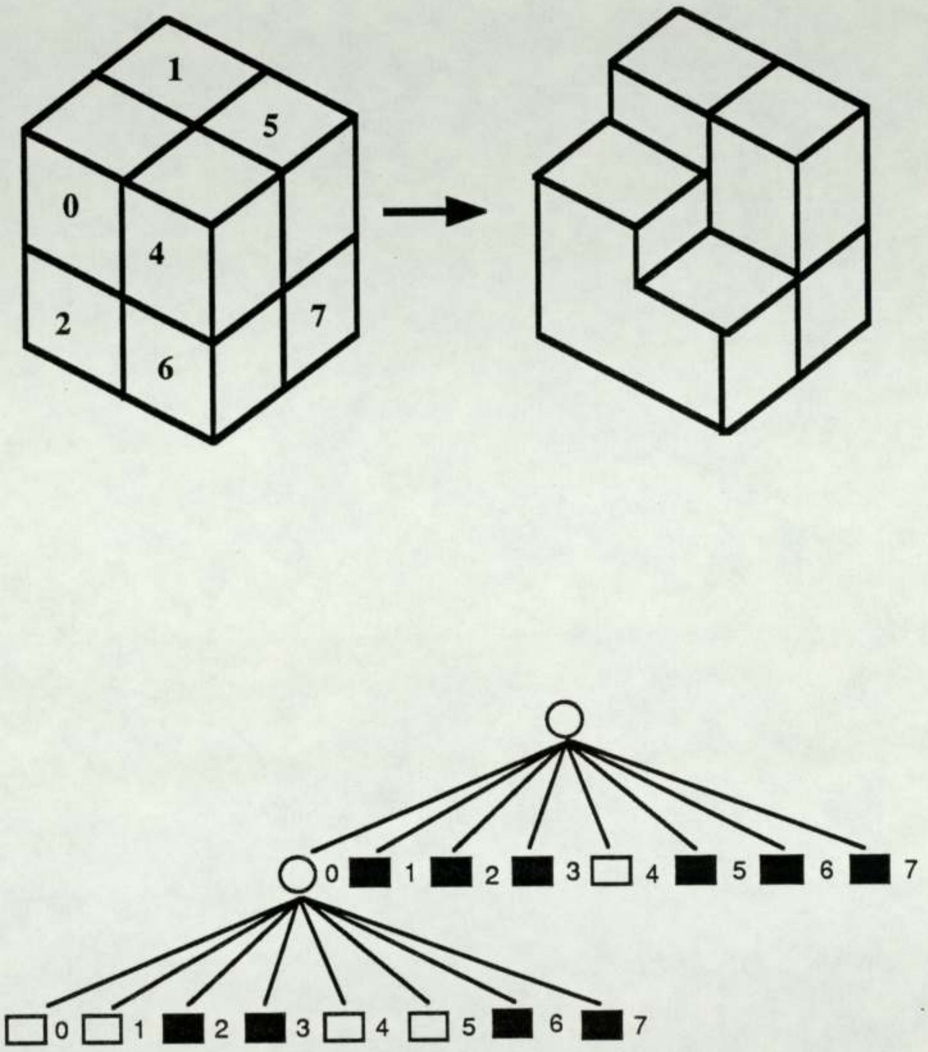
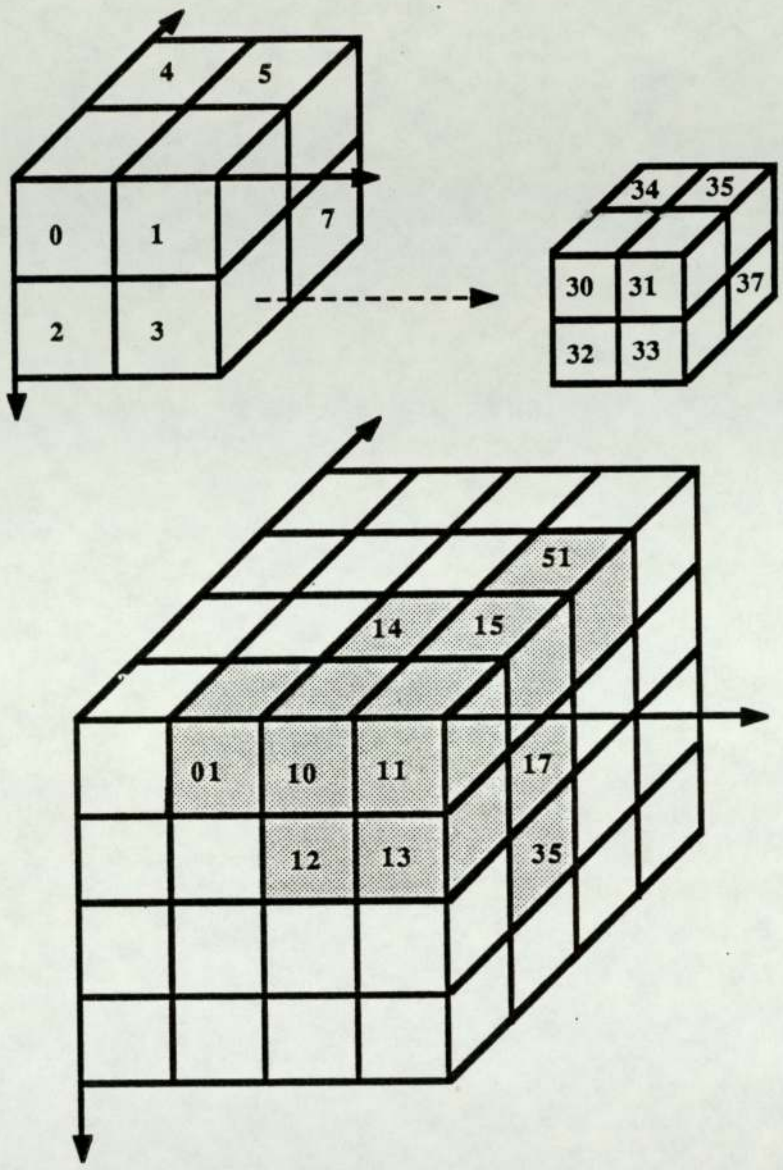


Fig. 3.9: Octtree representation



Linear encoding :

{01, 10, 11, 12 , 13, 14, 15, 16, 17, 35, 51}

After condensation: {01, 1X, 35, 51}

Fig. 3.10: Linear octtree encoding

3.4.2) <u>SWEEPING</u> :

In sweeping, an object is defined by a two-dimensional *contour* curve that is moved along a three-dimensional *trajectory* curve. The cross-section of the object is defined by the contour and the spine of the object is defined by the trajectory. Four different types of sweep objects can be distinguished, depending on the contour and trajectory definitions :

- translational sweep: the contour is arbitrary but the trajectory is a straight line [23]. Prismatic objects are easily defined by translational sweep, as shown in Figure 3.11
- rotational sweep: the arbitrary contour is rotated about an axis, i.e. the trajectory is a circle, as shown in figure 3.12. All axisymmetric objects may be defined using rotational sweep
- circle or sphere sweep: the contour is a circle (or a sphere [24]) and the trajectory is arbitrary, as shown in Figure 3.13.
- general sweep: both the contour and the trajectory are arbitrary [25], as shown in figure 3.14 .

Simple curves (straight lines, arcs of circles, and other quadrics) as well as general curves, such as parametric curves (Bezier and B-splines) may be used in a continuous sequence to define arbitrary contours and trajectories.

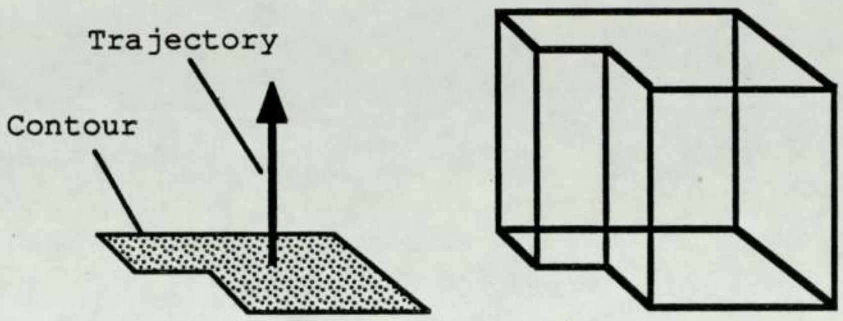


Fig. 3.11: Translational Sweep

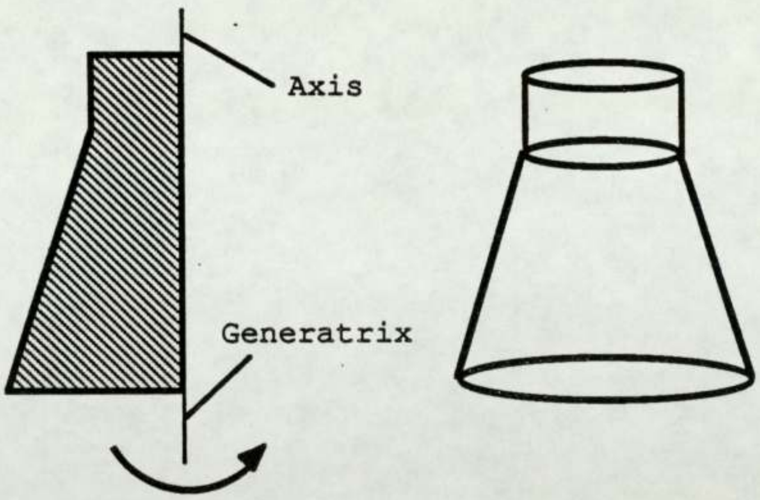


Fig. 3.12: Rotational Sweep

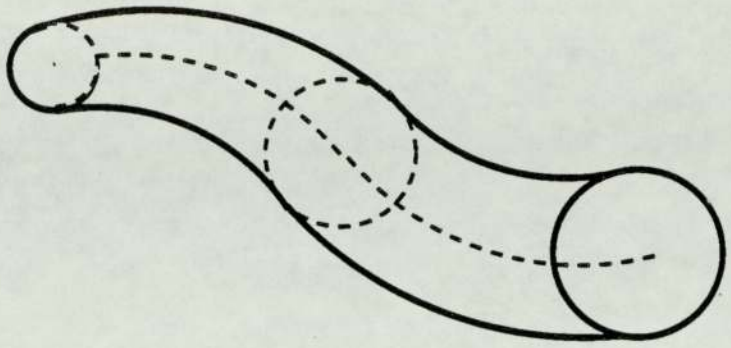


Fig. 3.13: Circle or sphere sweep

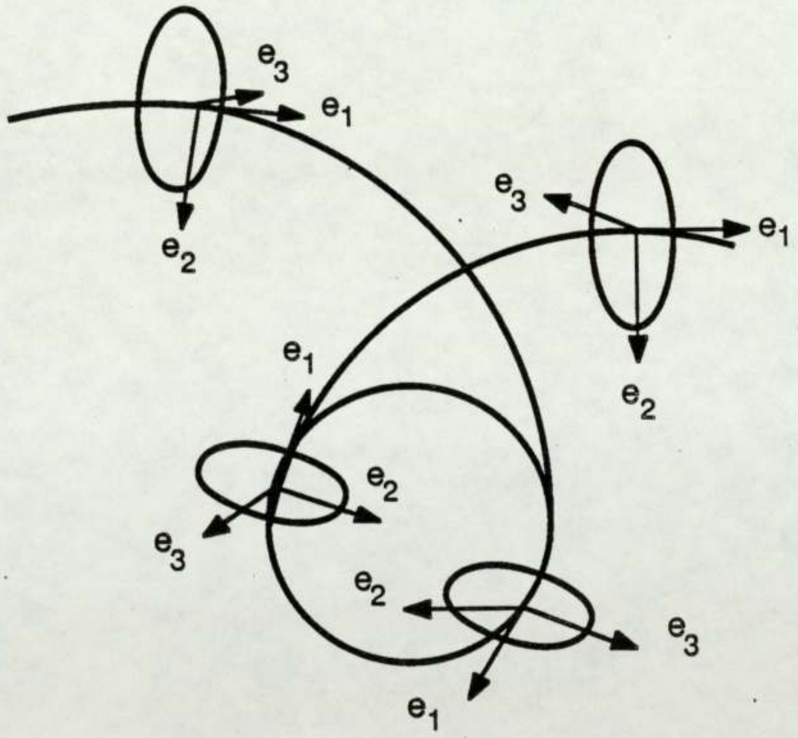


Fig. 3.14: General sweep

A recent development in sweeping allows tapered and twisted sweep objects to be produced [26]. This is achieved by allowing the size and the orientation of contour to vary as it is moved along the trajectory. The data structure stores scaling and rotation factors at a number of points along the trajectory.

One of the advantages of sweeping is that the representation is compact and does not require large amounts of storage. It is also suitable for input of models since it is relatively simple to specify a contour and a trajectory. The main disadvantage of sweeping is the restriction of the shape domain it may represent; only certain classes of object can be modelled with sweeping. For example, only objects with rotational symmetry can be modelled with rotational sweeping.

3.4.3) BOUNDARY REPRESENTATION :

In this representation, a solid object is defined in terms of its boundary elements, and these are specified in terms of a finite number of bounded faces. Several kinds of regular surfaces can be used as the basic face elements for describing the object. These include planar surfaces (polygons) and parametrically described surfaces, such as cylindrical, conical and spherical surfaces.

One B-rep data structure comprises a set of surfaces where each surface is represented by a set of directed edges that bound it, and each edge is represented by two vertices. These are held in a graph structure, known as the face-edge-vertex graph, which indicates the way in which they are connected. The topology information, i.e. the relationship between faces, edges and vertices, is specified by means of pointers which are in fact addresses of records in the data structure. For instance, in Figure 3.15, face F1 is bounded by edges E1, E2, and E3, and in the record of F2 there are therefore pointers to records E1, E2, and E3. Each entity in the face-edge-vertex graph has pointers back to the entities that own it, and to other related entities within the structure.

The face-edge-vertex graph is manipulated using the so-called Euler-Poincarre rule:

$$V - E + F - H = 2 * (M - G)$$

where:

V = Number of vertices

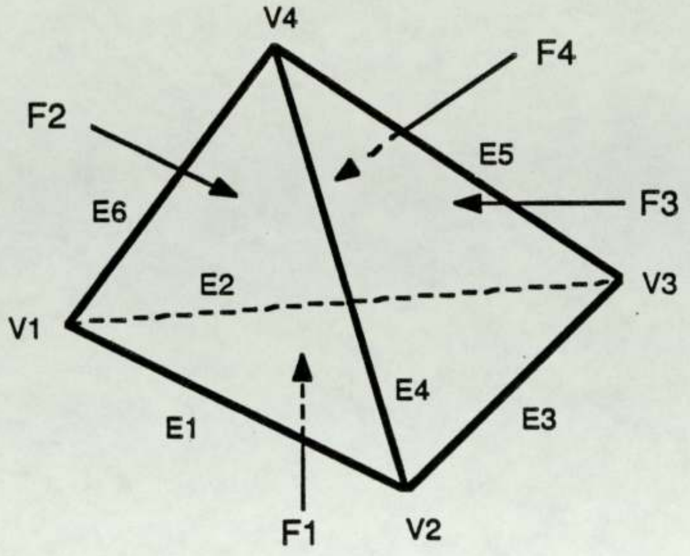
E = " " edges

F = " " faces

H = " " hole loops

M = " " separate pieces of solid

G = Genus of object



$$\begin{tikzcd}
 & & & \text{Tetrahedron} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & & \\
 & F1 \arrow[dl] \arrow[d] \arrow[dr] & F2 \arrow[dll] \arrow[dr] \arrow[drrr] & F3 \arrow[dl] \arrow[d] \arrow[dr] & F4 \arrow[dlll] \arrow[d] \arrow[dr] & \\
E1 \arrow[d] \arrow[drr] & E2 \arrow[dl] \arrow[drr] & E3 \arrow[d] \arrow[dr] & E4 \arrow[dl] \arrow[drr] & E5 \arrow[dl] \arrow[dr] & E6 \arrow[dlllll] \arrow[d] \\
V1 & & V2 & V3 & & V4
\end{tikzcd}$$

Fig. 3.15: Boundary representation of a tetrahedron

The hole loops are the internal boundaries at which several faces are joined together as well as to the perimeter boundary. The genus of an object is the number of holes it has; For example, a block has a genus of zero, and a torus has a genus of one.

Geometric information is also specified using pointers to appropriate geometric elements which serve to fix the object in space and define its geometry, as follows :

```
Face   --->  Surface (coefficient of equation)
Edge   --->  Curve   (coefficient of equation)
Vertex --->  Point   (coordinate triple x,y,z )
```

The geometry is therefore defined in terms of surface and curve equations to define faces and edges, and in terms of coordinate triples to define vertices, in space. For instance, the information about the equation of a planar surface:

$$ax + by + cz + d = 0$$

are the coefficients a, b, c and d. Quadric analytical surfaces can also be represented by the coefficients of their equations.

Most objects can be represented exactly with boundary representations using planar and quadric surfaces. However, in many boundary representation systems, curved surfaces are usually approximated by a mesh of polygons, in order to simplify the data structure and to make operations on the representation much faster. The disadvantage of using polygons or planar surfaces only, is that it may not provide adequate approximation of the object and uses a large amount of memory. In the boundary representation where only planar surfaces are allowed, the geometric information may be restricted to the coordinates of vertices, and the face and edge equations are derived from this information whenever it is necessary. The amount of topological information that is stored differs from one variant of boundary representation to another. For instance, it is not possible to determine the two faces which intersect at a particular edge; for this, in some boundary modelling systems, pointers from every edge to the faces intersecting at that edge are also stored.

In the *winged-edge* data structure [27], illustrated in Figure 3.16, every edge is assigned a direction, and from each edge there are pointers to:

- the two faces intersecting at that edge; these are called Fleft and Fright, as seen from the outside of the object
- the next edge in the sequence of edges bounding

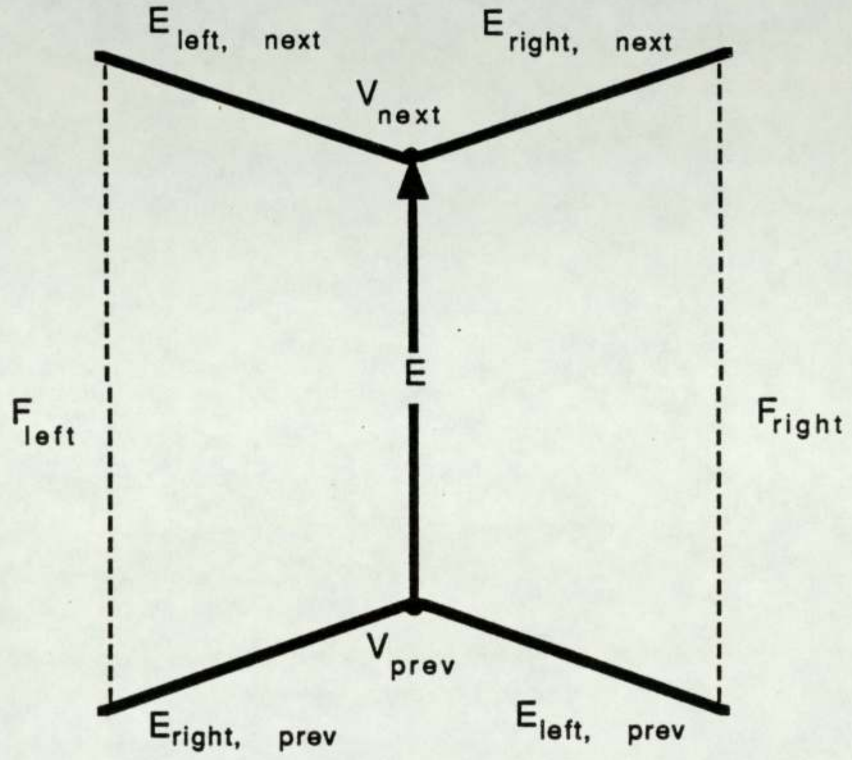


Fig. 3.16: Winged-edge representation

Fleft in clockwise order

- the next edge in the sequence of edges bounding Fleft in counterclockwise order
- the next edge in the sequence of edges bounding Fright in clockwise order
- the next edge in the sequence of edges bounding Fright in counterclockwise order
- the two vertices bounding the edge.

The boundary representation may be extended to cover a much larger domain, such as objects bounded by free-form surfaces [28].

The advantages of boundary representation are:

- the information about faces, edges and vertices, is explicitly present in the data structure, which allows applications such as fast display of the model to be achieved. It is well adapted for straightforward interrogating programs
- models can be generated step by step and local shape modifications are relatively easy to perform.

The disadvantages are:

- the data structure takes up a large amount of storage because of the amount of explicit information it contains. Most of this data is redundant

- the data structure is also complex, which may lead to the generation of invalid models of objects

- input to create models is difficult and tedious. This can be eliminated only by a sophisticated and well-designed user interface.

- inside/outside tests are more time-consuming

3.4.4) <u>CONSTRUCTIVE SOLID GEOMETRY</u>:

With constructive solid geometry, commonly abbreviated to CSG, a complex object may be synthetized from a finite number of much simpler shapes or *primitives*, like cubes and cylinders. These can be positioned in three-dimensional space by means of *transformations*, and then be combined to produce more complex objects using the Set (Boolean) operations of *union*, *intersection* and *difference*.

The primitives can be defined in a manner similar to that used in boundary representation (discussed in section 3.4.3) or can be specified in terms of low level entities called *half-spaces*. A half-space is generated by an infinite surface that divides the three-dimensional space into two parts, and may be defined, for example, by an inequality such as: $x \geq 0$

Simple objects can be represented as the intersection of a number of half-spaces. For instance, a unit cube may thus be represented as the intersection of the following half-spaces:

$x \geq -1$, $x \leq 1$, $y \geq -1$, $y \leq 1$,, $z \geq -1$ and $z \leq 1$

and the half-space:

$x^2 + y^2 + z^2 \leq 1$

is itself the unit sphere.

The user does not have to specify the half-spaces, but has at his disposal a number of simple primitive objects predefined with half spaces. This guarantees that no unbounded model can inadvertently be built. The primitives such as cubes and cylinders are common to most CSG modellers; the pyramid, cone, wedge, torus and sphere are useful for mechanical engineering components and are also available. Each primitive has a number of parameters that have to be specified. For example, for a block the parameters are the length, width and the height, and for a sphere, the radius.

The data structure for a constructive solid geometry representation of a solid model, consists of a binary tree, also called the CSG tree. At a leaf node of the tree there is information about a primitive: its type, the values of its parameters, and the transformations applied to it. At an internal node, there is the type of the Boolean operator (union, intersection or difference) to be applied to the objects defined by the left and right branches of that node, and pointers to these branches. Figure 3.17 shows a CSG tree, where ($\cup$*) stands for union, and (-*) stands for difference. The *union* of two objects A and B, (A $\cup$* B), is the object which consists of the points that lie within either A or B. The *intersection* of A and B, (A $\cap$* B), is the object which consists of the points that lie within both A and B. The difference of A and B, (A -* B), is the object which consists of all the points that belong to A and not to B. It is important to note that the 'difference' Boolean operator is not commutative; thus the object obtained by the Boolean operation B -* A is not the same as the object obtained by the difference A -* B.

In Figure 3.17, the operators are starred to indicate the difference between these operators and those used in classical Set Theory. The straightforward application of set theoric Boolean operators to the set of points defined by a three-dimensional solid may lead to anomalous results, such as 'dangling-edges', as shown in Figure 3.18(a). In order to avoid such problems, Boolean operators are refined in such a way that they operate on and produce "regular sets" [29], as shown in Figure 3.18(b).

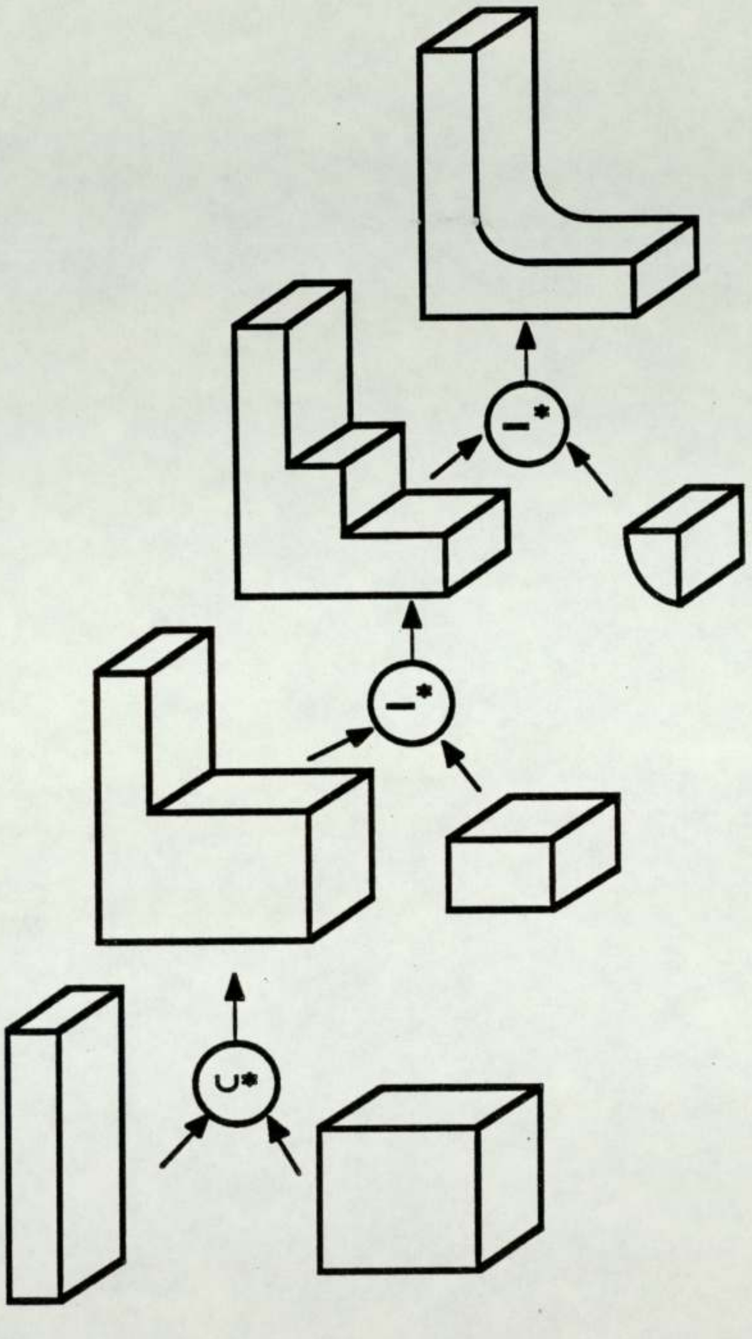
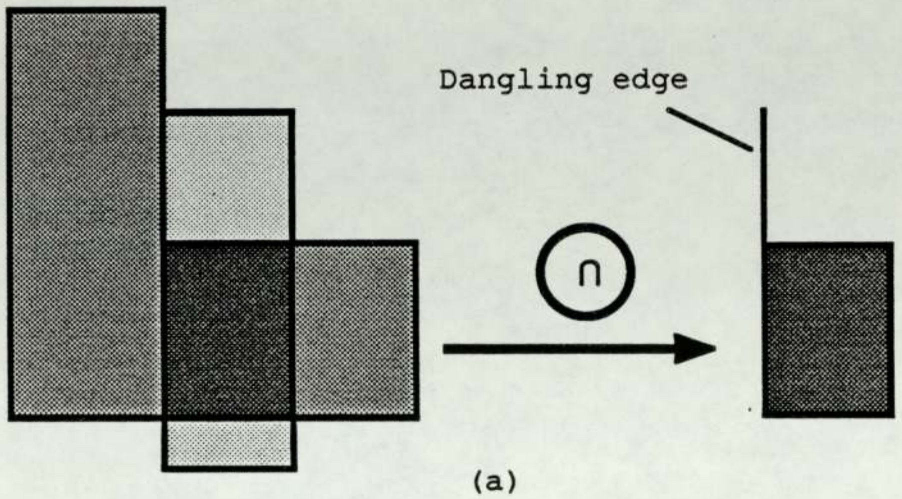
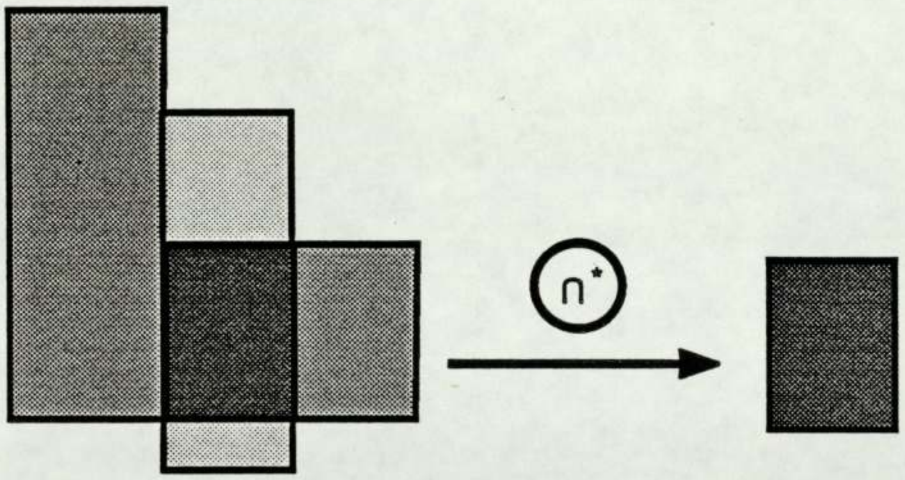


Fig. 3.17: Constructive Solig Geometry representatiun



(a)



(b)

Fig. 3.18: a) Non-regularised Boolean operation
b) Regularised Boolean operation

A number of transformations are available to move a primitive to the correct position and orientation in space:

- translation to move it in the X, Y or Z direction
- rotation to rotate it through an angle about the X, Y or Z axis
- scaling to change its size with a factor in the X, Y or Z direction
- skewing or shearing to change the angles between the X, Y and Z axis.

Constructive solid geometry representation is used in several commercial solid modelling systems, because it is compact, uses relatively little storage in comparison to cell decomposition or boundary representation, and all the objects that can be modelled are guaranteed to be valid. Another important advantage is the ease with which models can be built; very complex objects, in particular the majority of common mechanical parts, can readily be modelled, by using a restricted number of primitives, as illustrated in Figure 3.19. Moreover, conceptually, the CSG method has several similarities to engineering practice for designing and manufacturing mechanical components. For example, the 'difference' operation resembles cutting and the 'union' operation resembles bonding.

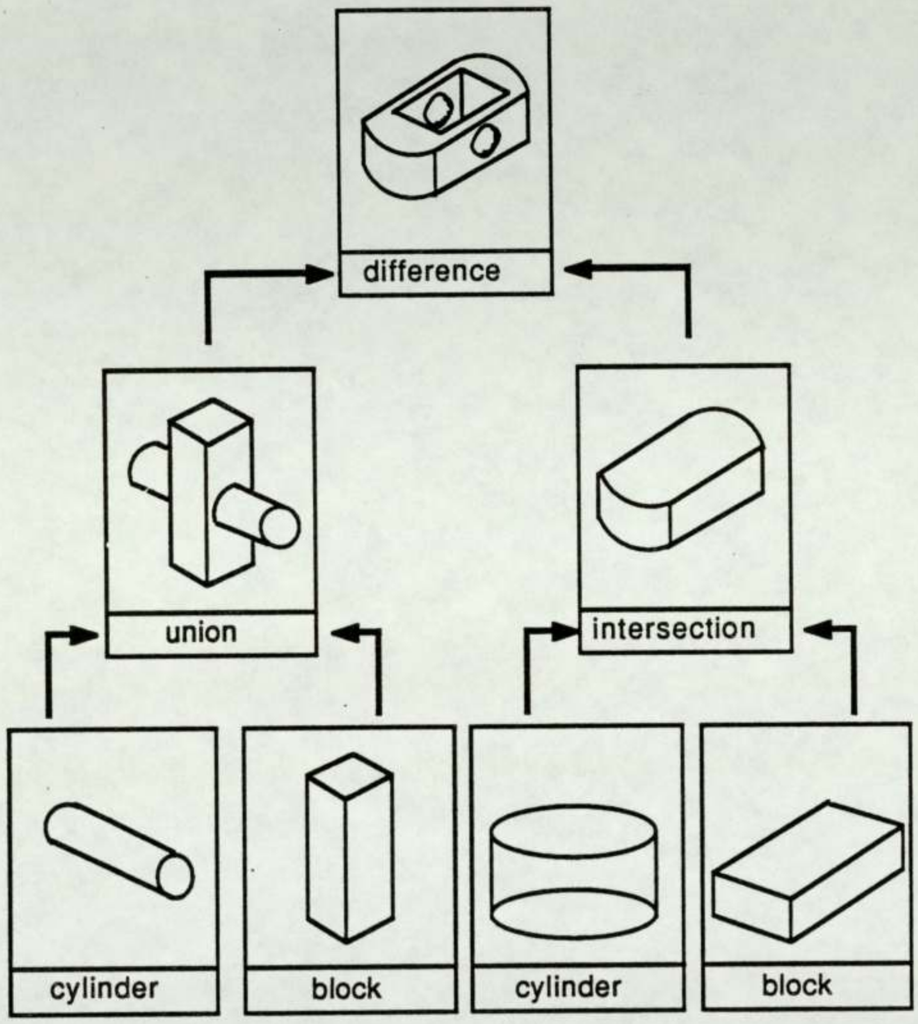


Fig. 3.19: Constructive Solid Geometry representation of a complex object

The disadvantage is that there no explicit information about the edges and vertices of the object in the data structure, as in boundary representation.

35) <u>SELECTION OF GEOMETRIC MODELLING REPRESENTATION FOR THE PROJECT</u>:

Several geometric modelling schemes have now been examined, all of which have their specific advantages and disadvantages. For instance, with constructive solid geometry, model input of mechanical parts is easy, but it is not the most suitable representation for making line drawings. On the other hand, a boundary representation is very suitable for making such drawings, but it in turn requires a large amount of memory. In general, the selection of a geometric modelling scheme depends on the applications: domain, input, applications and storage.

The work developed in the project is mainly concerned with the domain of mechanical engineering parts. This class of objects requires the provision for a complete volumetric information, to enable engineering applications, such as those described in section 1.1, to be performed. Clearly, these objects are best described using solid, rather than wireframe or surface modelling representation schemes.

The selection was initially narrowed down to choosing one of the solid modelling representations. The primary criteria used in the selection of one of the solid representations schemes was model input. The interpretation process developed in the project, requires the use of a solid modeller, (section 5.3.4). The input to the modeller is generated by a number of subroutines which have been developed in this work. Therefore, the amount of input data must be small to allow fast transfer to the solid modeller. Cellular decomposition was rejected on this criterion, since it requires that all the cells in the three-dimensional grid to be indicated and stored. The same applies to variants like octtrees. Moreover, these representations can only give an approximate description of the object. Boundary representation is also not suitable for model input since the validity of the input model is not guaranteed which may result in generating nonsense objects. This representation also requires a large amount of storage. Sweeping, especially translation and rotation sweeping, is very suitable for model input, since it requires only a small amount of data to specify an object. However, it has a limited domain of application, even in its generalized form.

Constructive solid geometry was adopted for this project because it is suitable for describing most mechanical engineering parts, and for model input. The PADL-1 development team at Rochester University [30], found that about forty percent of mechanical engineering components could be represented in terms of just two primitives: rectangular blocks and circular cylinders - subject to the restriction that block edges and cylinder axes were aligned with the coordinate axes. The addition of further primitive types (cones, spheres and tori), together with the removal of any restriction on the orientation of the primitives, allows modelling of more than ninety percent of mechanical parts. Model input in constructive solid geometry, prevents the generation of invalid objects and only a small amount of data is required to specify complex objects. Constructive solid representation has however, one drawback; it is not suitable for making line drawings. This problem has been solved by converting one modelling representation into another (appendix B), and many commercially available modellers provide such conversions. Input and storage may be effected by a constructive solid geometry representation, and if line drawings are required, the representation is then converted into a boundary representation.

CHAPTER FOUR

REVIEW OF EXISTING TECHNIQUES

4.1) INTRODUCTION:

Attempts to tackle the problem of reconstructing a solid object from its orthographic projections is not completely new, and some useful work in this area has been reported in the literature. A number of different approaches have been adopted with some early methods utilizing both hardware and software techniques. Such an approach was adopted by Sutherland in his work on three-dimensional input [31] which was focused on hardware and software for digitizing. He introduced a tablet with multiple pens so that a 3D vertex could be generated by digitizing vertices in two views. He also discussed how to treat digitized data from perspective views. Thornton's work [32] was also based on the same approach, and was concerned with interactive techniques for three dimensional input from two-dimensional views. However, neither Sutherland nor Thornton investigated algorithms for constructing solid models from projections.

The first algorithmic effort to construct solid models from their orthographic projections was initiated by Idesawa [33,34]. His method which focused on the domain of polyhedral objects, was largely based on labelling corresponding information in different views. The algorithms employed edge "tracing" techniques which mainly consist of tracing around labelled edges (lines) of engineering drawings and extracting the projected surfaces (which were closed loops of edge lines) in order to determine possible planes for those features. These algorithms also required the elimination of false elements such as "ghost" faces generated during the process of assembling projected faces.

Idesawa's method, briefly reviewed in section 4.2, was regarded as the basic method. His approach has been adopted by several other researchers in this field, mainly to improve the method and to extend the domain to non-polyhedral objects. For instance, Lafue [35], in his work on the recognition of three dimensional objects from orthographic views, added a procedure for removing false elements and finding true elements. His method had two drawbacks. First, it cannot remove all the possible false elements and can remove some true elements in multiple solution cases. Second, it constrained the user to a predetermined format when describing features such as faces; for example, two-dimensional lines are required to be input in such a way that a sequence of lines bounds a face. Preiss [36] attempted to free the user from as many constraints as possible. However, the relaxation of constraints has led to the possibility of multiple solutions, including "impossible" objects, to a given problem. The recent introduction of a heuristic approach to find the probable solution [37] has been applied to plane-faced bodies.

A completely different approach was adopted by Aldefeld [38], where mechanical parts were regarded as assemblages of separate prismatic objects, each object was required to have a base parallel to one of the coordinate planes. This method which utilized model recognition techniques, is reviewed in section 4.3 because of the new concept it introduced.

The work described in references [33-38] was not based on formal geometrical and topological definitions, and led in all cases to wireframe representations of the objects. This mode of representation has a serious limitation in that it does not provide the volumetric information required for manufacture, assembly and design analysis purposes.

Wesley and Markowsky used algebraic topology concepts and rigorous definitions of geometric entities to allow a volumetric description to be obtained in terms of solid material, empty space and topology of surfaces and edges for objects described in terms of their wireframe [39]. They used the same approach to obtain objects described in terms of their projections [40]. However, the algorithms were still restricted to objects having straight line edges and planar surfaces. This concept was developed further by Sakurai and Gossard [41] to extend the interpretation process to include objects with rotational symmetry such as cylinders, cones, tori and spheres for which the axes are parallel to one of the coordinate axes. This reconstruction algorithm developed by Wesley and Markowsky was also improved by Kaining [42], who made use of the idea of pattern recognition expressed in the Aldefeld algorithm, to include cylindrical objects for which the axes are parallel to one coordinate plane rather than to one coordinate axis. This improvement allows the input views to comprise elliptic arcs, hyperbolas and regular higher order curves with their symmetry axes parallel to one coordinate axis. Kaining's method is reviewed in depth in section 4.4. This is because it illustrates the basic approach adopted by Wesley and Markowsky, and to describes the improvements it makes on their algorithm. Moreover, much of Kaining's work has direct relevance to this project and requires detailed exposition.

In all the above techniques, boundary representation has been used to describe the output object model. As previously discussed, in section 3.4.3, this mode of representation does not guarantee the validity of the object. It is for this reason that most of the above methods required algorithms to deal with pathological cases. Constructive Solid Geometry, a more adequate representation, has been adopted in an interesting work reported in [43,44] for the interpretation of orthographic views as solid models. However, in this case, the interpretation process is not fully automatic and requires a 'man in the loop', i.e. the user, to carry out most of the interpretation tasks, such as the identification of three-dimensional primitives, the input of their corresponding data and the comparison of input and output orthographic views. A method based on such an interactive, or 'semi-automatic', approach is described in section 4.5, to illustrate the extent to which human intervention is required by such techniques.

Constructive Solid Geometry principles, together with the 'man in the loop' concept, have also provided the basis for the process of interpreting engineering drawings as solid models, developed in the present work. However, the extent to which the process relies on the user is far less than the one used in the interactive method described in section 4.4. This chapter is concluded by a discussion which highlights the reasons for adopting these concepts and the differences between the tasks carried out in the process developed in this work, as compared to those required by the so-called semi-automatic methods.

4.2) <u>IDESAWA'S METHOD</u>:

Idesawa describes his approach as the inverse transformation of the operation which is used to produce orthographic projections of a given object. His algorithm is divided into five main steps:

(a) generate 3D vertices from 2D vertices

(b) generate 3D edges from 3D vertices

(c) elimination of ghost elements

(d) generate 3D faces from 3D edges

(e) assemble true faces into an object.

Each of the above steps can be briefly described as follows:

(a) <u>GENERATION OF 3D VERTICES FROM 2D VERTICES</u>:

In any given orthographic view, a two-dimensional vertex is defined in terms of a pair of coordinates; For instance, any vertex in the XY view has an x-coordinate value and y-coordinate value, and an unknown z-coordinate value in the direction of sight.

The purpose of this step is to determine the missing coordinate value for each two-dimensional vertex in order to generate the corresponding three-dimensional vertex. This is achieved by the following matching rule:

Consider, three points: P(x,y) in the XY view, P'(x,z) in the XZ view and P"(z,y) in the ZY view, as shown in Figure 4.1(a). A 3D vertex V(x,y,z) is defined by the views, if:

$$x(P) = x(P')$$
$$y(P) = y(P'')$$
$$z(P'') = z(P')$$

The corresponding three-dimensional point created, V(x,y,z) is shown in Figure 4.1(b).

(b) <u>GENERATION OF 3D EDGES FROM 3D VERTICES</u>:

In a three-dimensional object each edge is defined by a pair of three-dimensional vertices. The purpose of this step is to obtain each pair of vertices that define three-dimensional edges. Idesawa devised a function which takes as its main input all the combinations of generated three-dimensional vertices in pairs and outputs some value signifying whether or not a given pair of vertices are connected. The function is specified by the Boolean operations required for each set of entities. The interested reader may find more details in [33].

(c) <u>ELIMINATION OF GHOST ELEMENTS</u>:

The three-dimensional elements (vertices and edges) generated in the previous steps may not all be true elements. Idesawa refers to those elements which are not true one as ghost figures. These are partially eliminated according to a set of twelve rules.

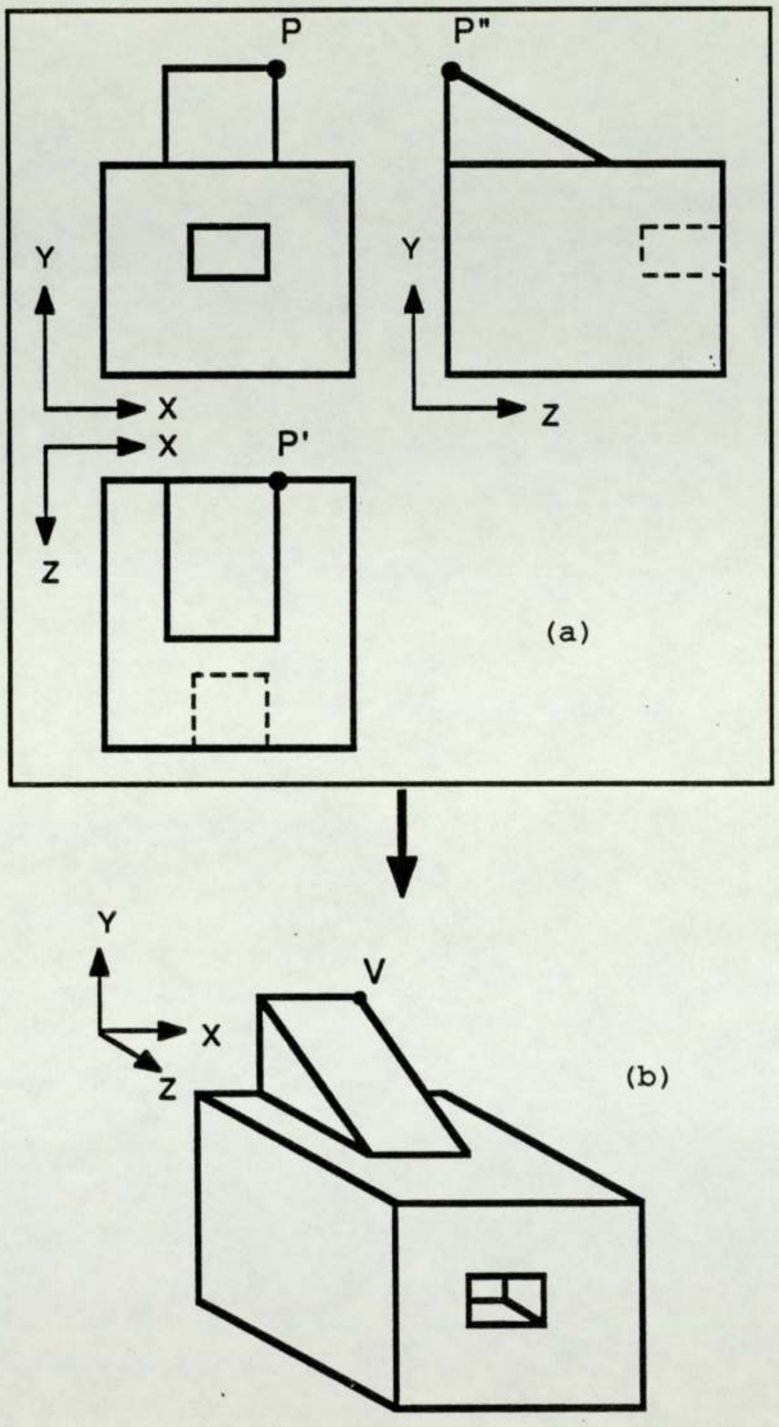


Fig. 4.1: a) Matching 2D points and
b) corresponding 3D vertex

(d) GENERATION OF 3D FACES FROM 3D EDGES:

In this step, a search for three-dimensional edges that are likely to bound a face is carried out together with a further elimination of ghost figures. Faces are defined only if the following conditions are met:

i) There are n faces which contain a vertex as a given intersection of n edge lines.

ii) An edge line constitutes the boundary of two faces, and runs in opposite direction to each other in the row of boundaries.

iii) A boundary of a face is enclosed.

An edge line which can not be in any boundary of faces is eliminated as a ghost line. Finally, the object is described in terms of a number of planar faces.

The main disadvantage of Idesawa's method is its domain of application which is limited to polyhedral objects, as curved surfaces can not be treated. Furthermore, false elements can not really be distinguished from true elements, and thus true elements can easily be deleted.

4.3) ALDEFELD'S METHOD:

The underlying philosophy of Aldefeld's method is to view a complex part as being composed of elementary objects belonging to a set of predefined classes, and these elementary objects may be recognized by making use of the knowledge about class-dependent patterns of their two-dimensional representations. Each elementary object will have, in each view, a two-dimensional pattern which will identify the object. Each two-dimensional pattern comprises a number of 2D primitives, such as lines, arcs and circles. Primitives may be concatenated to form line segments. Line segments and arcs may be grouped to form closed loops, and finally, an object view comprises line segments and loops. A number of different attributes are used in the data structure to define the relationship between these entities; for instance, the attribute CONTACT(p,q), between primitives, means that primitives p and q have at least one common node, and the attribute CONSISTS_OF relates line segments to loops. These attributes are used in the recognition of the 2D pattern they form in the views of the elementary object. Figure 4.2 illustrates the types of entities and their relationships in the data structure.

To avoid the whole complexity of possible geometries, the method is confined to a subset of structures and the following restrictions are placed on

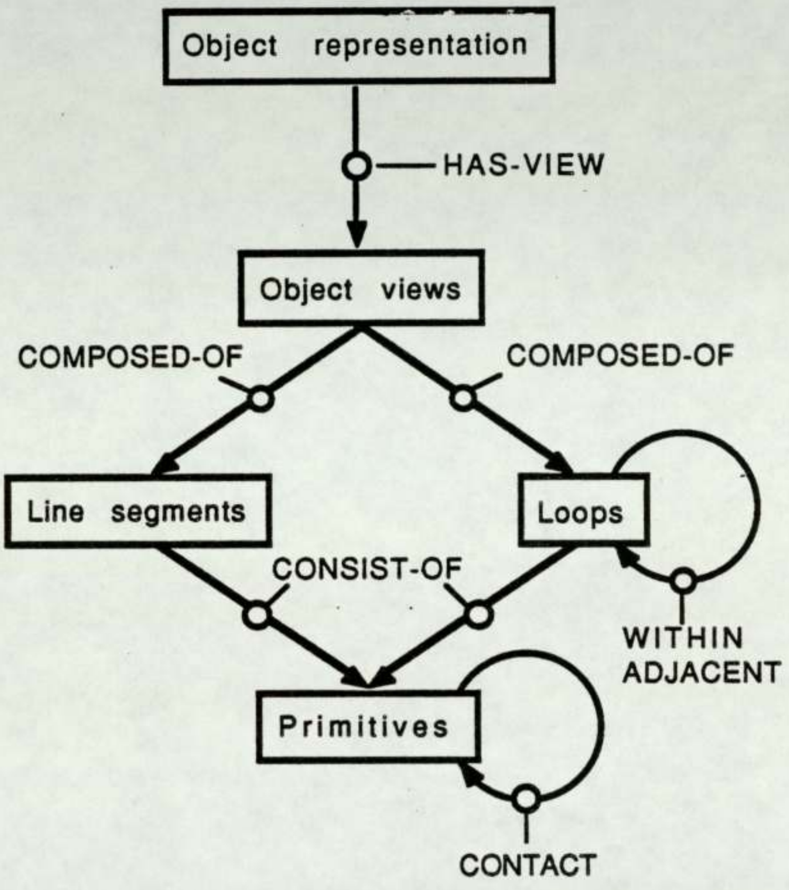


Fig. 4.2: Types of entities and relationships defining the data structure

the structure to be interpreted:

1) All elementary objects must be of uniform thickness, i.e. prismatic objects.

2) The base of each elementary object must be parallel to one of the coordinate planes.

Figure 4.3 shows an object which complies with the above restrictions and its 3-view orthographic projections. For an object restricted in such a manner, one of the views, V_1, will consist of a single loop, also referred to as the 'silouhette', of an arbitrary shape and the other views will comprise a rectangle subdivided by line segments, with the sides of the rectangle and the line segments being parallel to a coordinate axis.

The model-guided recognition algorithm used to recognize a uniform thickness object for which a loop, L, in a given view V_1, represents the base silouhette, may be described as follows:

1) Search in an arbitrary view, where the chosen view is not V_1, to find all the rectangles that 'match' the silouhette loop, L, in the given view V_1. - A *match* between two loops l_1 and l_2 from different views is defined if the minimum coordinate of l_1 is equal to the

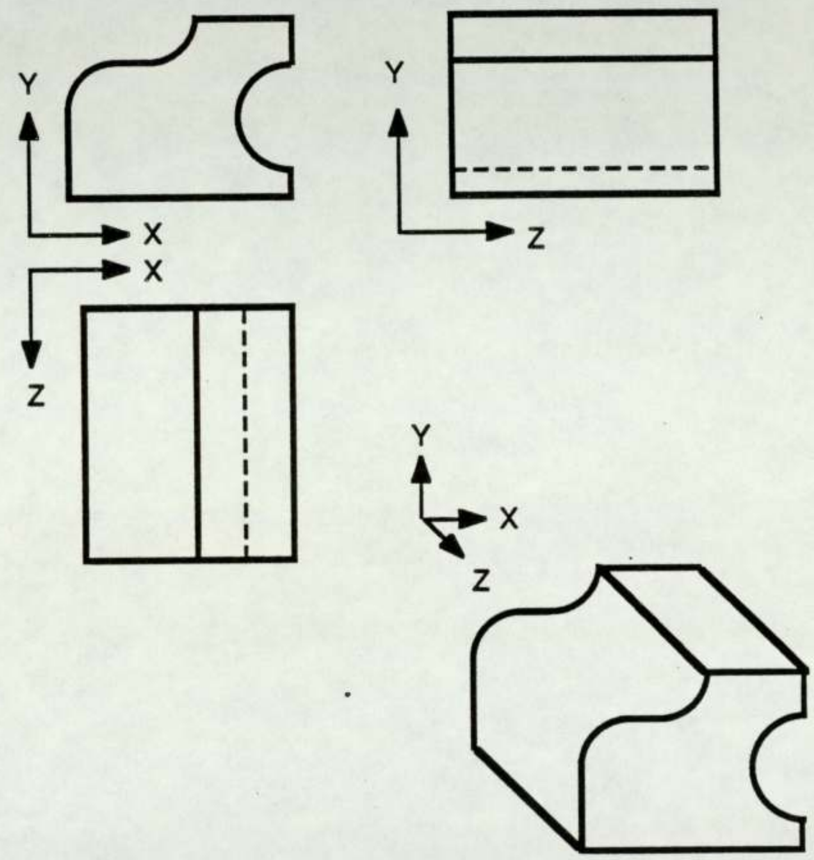


Fig. 4.3: A uniform-thickness object

minimum coordinate of l_2 and the maximum coordinate of l_1 is equal to the maximum of l_2, in the common coordinate direction.

2) Search the remaining view for all the loops that match the loops generated from step (1) and the loop L in the view V_1. A list of 'matching' loops is generated.

3) Loop L is scanned for features that signify the presence of line segments in one or both of the remaining views. For instance, features such as corners formed by primitives of loop L.

4) For each pair of matching loops listed in step (2), attempt to find the complete set of line segments required by the features. If this is successful, a complete object pattern given by the union of these matching loops and generated line segments has been found.

The model-guided algorithm will only work if true patterns of elementary objects are offered to it. Unfortunately, it is not always easy to extract true patterns from the views due to the overlapping of faces and edges when an object is represented as a set of orthographic views. In a bid to overcome this problem, Aldefeld uses heuristic techniques so that subpatterns can be extracted on the basis of hypotheses. The strategy which Aldefeld calls the "Best First Search" is based on an evaluating function that assigns scores (number of points) to patterns on the merit of their characteristics.

The patterns are then chosen in the order dictated by their accumulated score for the input to the recognition process. Two main scoring methods are used; Each pattern is first assigned a score according to the number of primitives it comprises and which have not yet been recognized as a part of an object representation. Each pattern is then assigned another set of points depending on the attributes it may have in relation to other patterns. For instance, a pattern that is not adjacent to any other pattern, i.e. isolated, is given a higher score than a pattern which has the attribute 'adjacent' assigned to it. This hypothesis is true since an isolated pattern must necessarily represent the silouhette of at least one partial solid.

Finally, Aldefeld's reconstruction algorithm can be briefly described as follows:

1) Find all the relationships between primitives.

2) Find all closed loops and assign them their various attributes, i.e. 'circular', irregular, etc..

3) Assign a score to each loop using the evaluating function, and select the loop which has the highest score.

4) The loop selected from step (3) is assumed to be the base silouhette of one or more partial solids. This assumption is verified or rejected using the model-guided recognition algorithm. A three-dimensional structure is generated if the assumption is held as true.

5) The loop is 'expanded' to include itself and an adjacent loop. The expanded loop is checked if it already exist; if not, add it to the set of data, find its attributes and relationships with other loops as before. Mark the new loop as 'open' and the original loop as 'closed'.

6) Verify whether the generated object complies with the input data; if so exit the algorithm, else continue from step (3).

The main disadvantage of Aldefeld's method is that it only works on a local basis since it deals with one partial solid at a time, and ignores the global context. For this reason, the reconstruction algorithm can not distinguish between solid bodies and cavities, and may also generate false partial solids due to silouhette interference. Furthermore, the generalization of the method to true three-dimensional non-uniform objects would require not only more sophisticated heuristics, but also the extension of the domain of partial objects to include those which, for instance, have rotational symmetry objects, and the relaxation of the restriction on their spatial orientation.

4.4) **KAINING'S METHOD**:

Kaining's method is based on the algorithm developed by Wesley and Markowsky. Their algorithm resembles the one presented by Idesawa, briefly discussed above (section 4.3), in the sense that it 'fleshes out' projections hierarchically from lower levels to higher ones, but by making use of rigorous mathematical and topological definitions, Wesley-Markowsky's algorithm gives better results on handling pathological cases and multisolution problems. Basically, the algorithm can be described as follows:

1) Generate 3D vertices from 2D vertices.
2) Generate 3D edges from 3D vertices.
3) Generate 3D subfaces from 3D edges.
4) Assemble 3D subfaces to form 3D subobjects.
5) Assemble 3D subobjects to form objects matching the input 2D projective representations.

However, as in Idesawa's algorithm, the domain of objects that may be interpreted is limited to those having planar faces only.

Kaining's algorithm extends the interpretation process to include objects having cylindrical faces. Furthermore, the axis of any cylinder is restricted to be parallel to one coordinate plane. The different steps of the algorithm are illustrated in Figure 4.4, the details of which are described as follows:

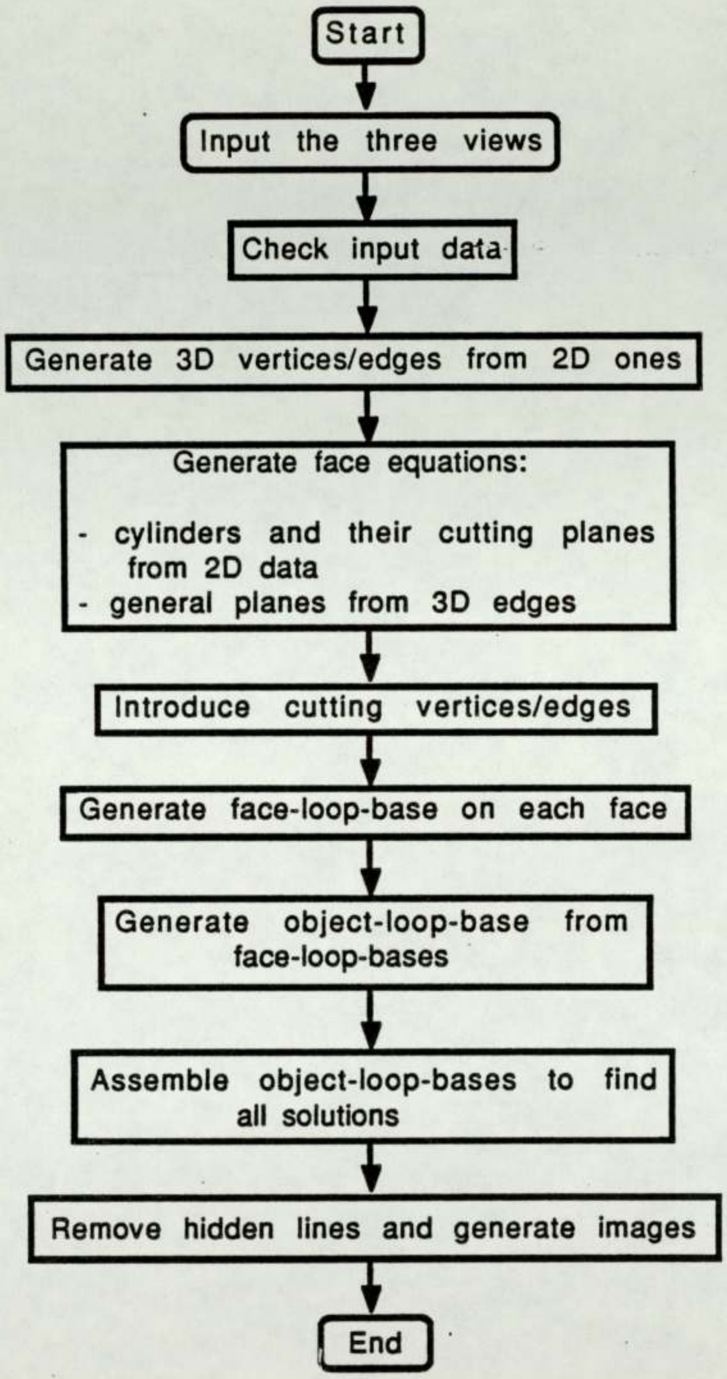

Start
Input the three views
Check input data
Generate 3D vertices/edges from 2D ones
Generate face equations:
- cylinders and their cutting planes from 2D data
- general planes from 3D edges
Introduce cutting vertices/edges
Generate face-loop-base on each face
Generate object-loop-base from face-loop-bases
Assemble object-loop-bases to find all solutions
Remove hidden lines and generate images
End

Fig. 4.4: Flow chart of KAINING's algorithm

1) GENERATE 3D VERTICES/EDGES FROM 2D ONES:

The following principles are applied to derive 3D vertices and 3D edges:

a) Matching principle:

If E_f, E_t and E_s are projected edges (or vertices) on the front, top and side views respectively, then they may be referred to as a group of matching edges if their surrounding rectangles defined by their maximum and minimum coordinates, i.e. $(x_{fmin}, z_{fmin}, x_{fmax}, z_{fmax})$, $(x_{tmin}, y_{tmin}, x_{tmax}, y_{tmax})$ and $(y_{smin}, z_{smin}, y_{smax}, z_{smax})$ respectively meet the following conditions:

$$x_{fmin} = x_{tmin}, \qquad y_{tmin} = y_{smin}, \qquad z_{smin} = z_{fmin}$$

$$x_{fmax} = x_{tmax}, \qquad y_{tmax} = y_{smax}, \qquad z_{smax} = z_{fmax}$$

b) Line mode:

A 3D straight line can be derived from a group of matching edges E_f, E_t and E_s if and only if:

i) E_f, E_t and E_s are 2D straight lines (at most one of which can be a 2D vertex);

ii) there is a group of endpoints of them satisfying the matching principle.

c) Ellipse mode:

A 3D elliptical edge can be derived from a group of matching edges E_f, E_t and E_s if and only if:

i) E_f, E_t and E_s are 2D ellipses with their axes parallel to the coordinate axes, or 2D straight lines, and there is at least one ellipse and one straight line among them;

ii) If there are two ellipses amongst E_f, E_t and E_s, then their centres have the same coordinate value in the shared coordinate;

iii) Each group of endpoints of elliptical arcs and another group of points on the elliptical arcs, satisfy the matching principle.

The elliptical mode is illustrated in Figure 4.5.

d) Higher order curve mode:

Higher order curves can be derived from a group of matching edges E_f, E_t and E_s, when two of them are circular arcs and the other is either hyperbolic or a regular higher order curve. Higher order curves arise from the intersection of two cylinders, with different radii or non-intersecting axes, whose axes are parallel to coordinate axes, as shown in Figure 4.6.

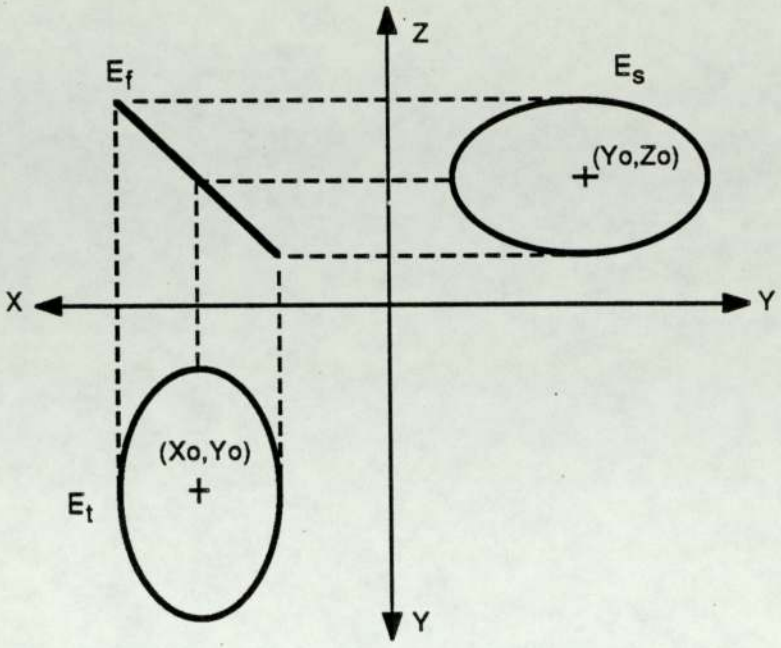


Fig. 4.5: Ellipse mode

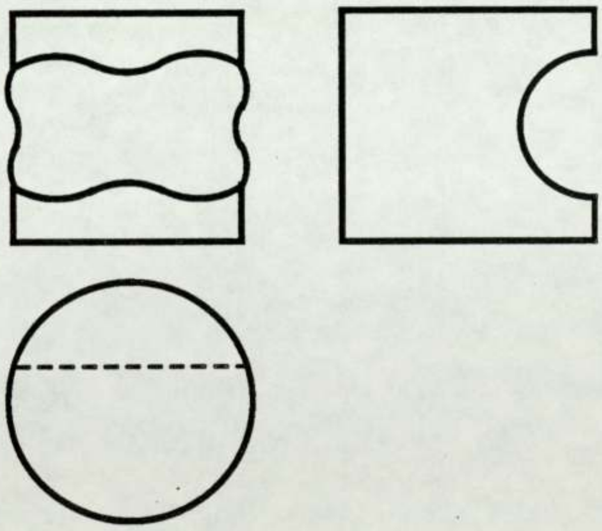


Fig. 4.6: Higher order curve mode

2) GENERATE FACE EQUATIONS:

A face equation may be generated from pairs of non-colinear 3D straight line edges sharing a common endpoint. The equation of a cylindrical face can be defined by three geometric parameters:

- a point on the axis of the cylinder, referred to as the location point,
- a radius
- the orientation of the axis of the cylinder.

The location point is obtained by using the matching principle to recover the centre of the 3D ellipse, while the radius and the orientation of the axis can be derived as follows:

i) Derivation of the radius:

Let G be the generating cylinder of a 3D ellipse E, as shown in Figure 4.7. If the axis I of such a cylinder is parallel to the OXY plane, then the generating plane P of E will be perpendicular to the OXY plane. If, in addition, P is not perpendicular to the OXZ plane, then:

a) the orthographic projection of E on OXZ is an ellipse S with its axes parallel to either the X or Z axis;

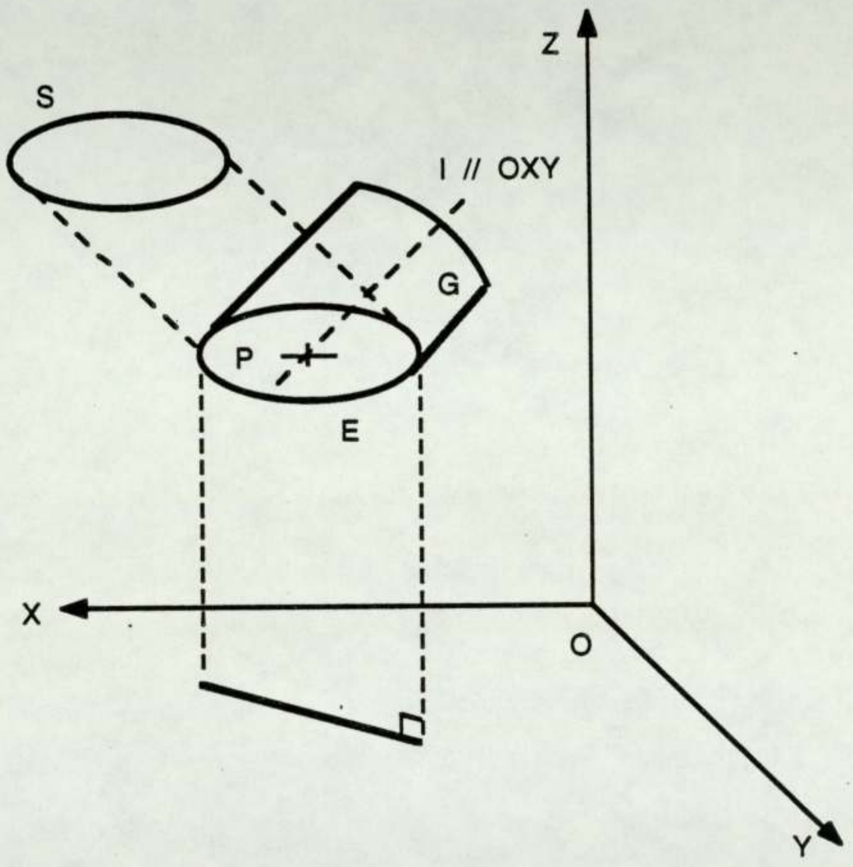


Fig. 4.7: Derivation of the radius

b) the radius of the generating cylinder G is equal to half the length of the axis of S parallel to the Z axis.

In the above derivation of the radius, it is assumed that the axis I of the generating cylinder G is parallel to the OXY plane. It is possible to determine the coordinate plane to which the axis I of the cylinder G is parallel. Two examples are given here in order to illustrate how to deal with this problem; In Figure 4.8(a), only one straight line exists in the group of matching edges. In this case, the axis I must be parallel to the plane in which this line lies. In Figure 4.8(b), only one ellipse E_f exists in the group of matching edges. In this case, the length of the minor axis of E_f is equal to the diameter of the cylinder G.

ii) <u>Derivation of the orientation of the cylinder axis</u>:

Figure 4.9 represents the orthographic views of an object composed of two intersecting cylinders at an oblique angle. The point $C(x_o, z_o)$ on the line E_f corresponds to the centre point (xo,yo) of the ellipse E_t, and the point (x_1, y_o) on E_t is the endpoint of the major axis of the ellipse E_t. This point (x_1, y_o) also corresponds to the point $P(x_1, z_1)$ on the line E_f. An auxiliary circle with C as centre and with radius equal to the radius of

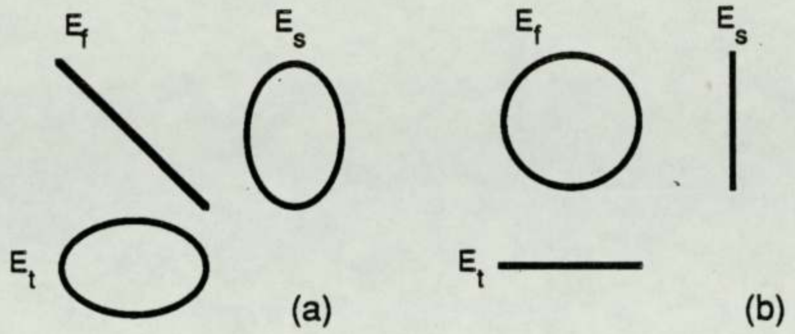


Fig. 4.8: Two cases in deriving the radius

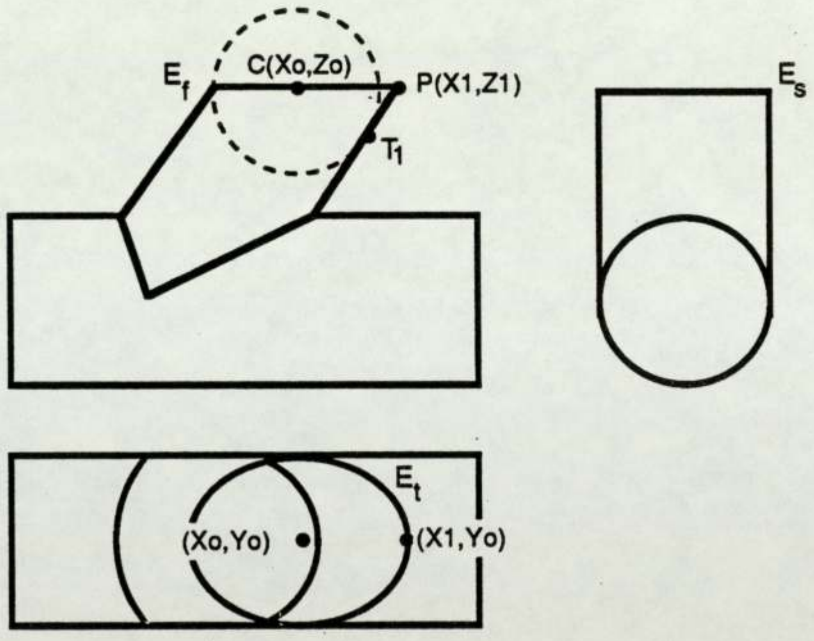


Fig. 4.9: Derivation of cylinder axis

the generating cylinder, may be constructed to have the line PT_1 as tangent at the point T_1. The orientation (dx,dz) of line PT_1 is that of the projection of the cylinder's axis. The orientation (dy,dz') of the cylinder's axis on the side view can be derived in similar fashion, and by scaling one orientation vector so that:

$$dz = dz'$$

the orientation in 3D-space (dx,dy,dz) can be obtained.

3) <u>INTRODUCE CUTTING VERTICES AND CUTTING EDGES</u>:

There are two types of pathological cases which may arise with the 3D edges and faces generated above. One is when two edges intersect at one of their interior points rather than endpoints. Such an intersection can appear as an endpoint in a set of orthographic projections as those shown in Figure 4.10 where pathological point P exists between edges AC and BD. The other pathological case may arise when two different faces intersect at their interior lines rather than boundary edges. The faces AEGC and BFHD intersect at such interior line PQ. These two types of pathological cases should not appear in well defined geometric objects.

The pathological intersecting point P, referred to as a cutting vertex is introduced to separate its two generating edges, AC and BD, into four edges, AP, BP, DP and PC, so that the pathological case is removed. Similarly, the pathological intersecting line PQ, referred as cutting edge, is introduced to separate its two generating faces, AEGC and BFHD, into four separate faces, AEQP, PQGC, BFQP and PQHD.

4) GENERATE FACE-LOOP-BASES:

In order to define a face-loop base the following definition are first introduced:

- A face-loop on a face F is defined by Kaining as a "simply interconnected" area bounded by a subset of edges on F. For example, in Figure 4.11, $f_1, \ldots, f_7$ are face-loops on face F.

- A boundary edge set $E(f_1 + f_2 + \ldots + f_m)$ of the union of the face-loops $f_1, f_2, \ldots, f_m$ is defined as follows:

$$E(f1 + f2 + \ldots + fm) = \bigcup_{i=1}^{m} E(fi) - \left(\bigcup_{i=1}^{m-1} \bigcup_{j=i+1}^{m} E(fi) \cap E(fj)\right)$$

where the sign '+' denotes the union of some different face-loops on one face and (fi) denotes the boundary edge set of the face-loop fi.

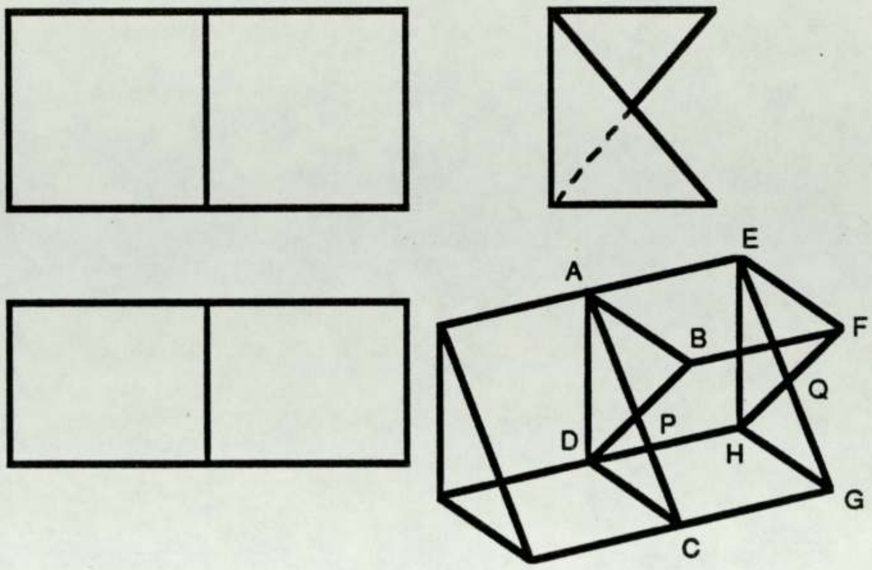


Fig. 4.10: Cutting vertex/edge

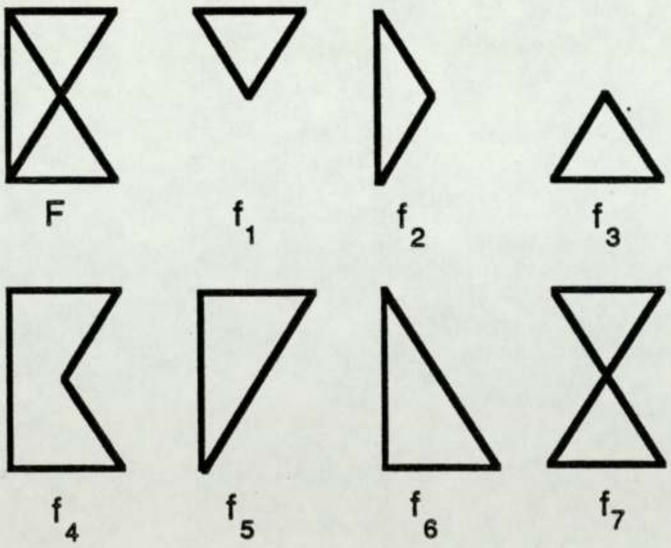


Fig. 4.11: Face-loops

From the above definitions, the face-loops f4,..., f7 in Figure 4.11 can be generated by the union of set $f_1,\dots,f_3$, i.e.:

$f_4 = f_1 + f_2 + f_3$ $f_5 = f_1 + f_2$

$f_6 = f_2 + f_3$ $f_7 = f_1 + f_3$

A set of face-loops on face F, $B_f = f_1,\dots.,f_k$, is defined as a face-loop-base if any face-loop on F can be generated from one or more faces in Bf and each face-loop in Bf can not be generated from other face-loops in Bf. For example, in Figure 4.11, the set $B_f = f_1,f_2,f_3$ is the face-loop-base on face F.

Kaining devised the following algorithm to determine the face-loop-base of each face:

a) For each vertex v_i on F, sort its incident edges on F in a counter clockwise order such as e_1 e_2 ... e_k, the e_1 is the left-adjacent-edge of e_2 at v_i,..., and e_k is the left-adjacent-edge at e_1.

b) Pick an ordered edge $e_i(v_i,v_j)$ at v_j, followed by picking its left-adjacent-edge $e_j(v_j,v_k)$ at v_k, then pick the left-adjacent-edge of e_j at v_k,..., the face-loop L will be formed when edge e_m (v_m,v_i) jointing the first edge

e_i is picked. The left side of each ordered edge is defined as the interior of L.

c) Since there are only two ways to traverse each edge, either from v_i to v_j or from v_j to v_i, all the face-loops will be obtained when each edge is picked twice in different directions.

d) The face-loop-base is made from all of the bounded face-loops except unbounded ones.

5) <u>GENERATE OBJECT-LOOP-BASE</u>:

The philosophy of object-loop base and the union of object-loops can be derived by extending the concept face-loop-base and the union of face-loops. In the example shown in Figure 4.12, if the ordered face -f1 is first picked, then in order to ensure that there is no face-loop in the interior of the object loop, it is necessary to pick the ordered face -f3. But if the face +f1 is picked at first, then face -f2 should be picked next. An object-loop B will be formed if the ordered face-loop are picked as shown above repeatedly until each edge in B has been contained by two face-loops in B.

All of object-loops are found when each face-loop is traversed twice in two direction. The object-loop-base is composed of all the bounded object-loops except unbounded ones.

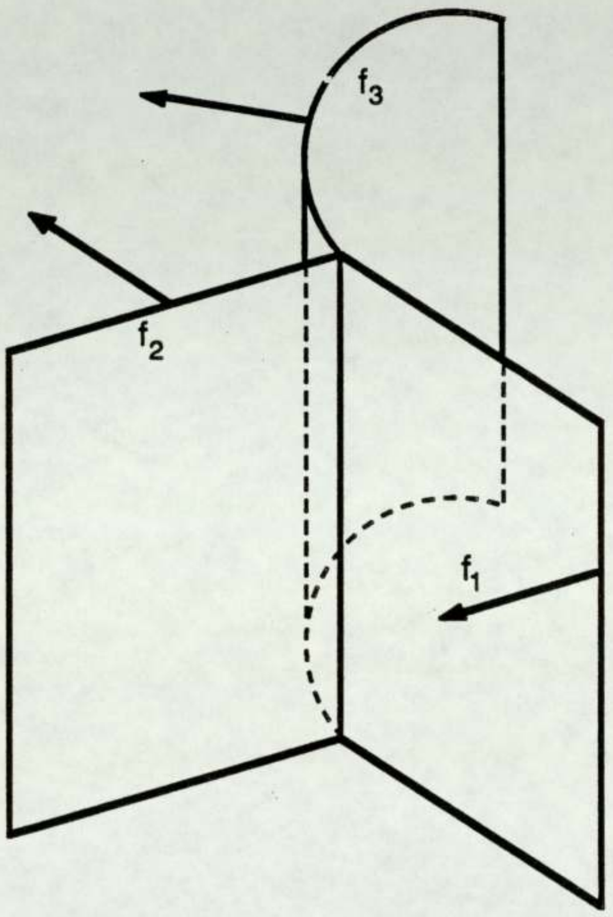


Fig. 4.12: Incident faces

6) <u>ASSEMBLE OBJECT-LOOPS TO GENERATE SOLUTIONS</u>:

Object-loops in the base may either be disjoint, or have some vertices, edges or face-loops in common. Therefore, the rules to assemble object-loops are simple and may be described as follows:

a) Delete face-loops shared by only two object-loops, since it is not allowed for a face to be in the interior of an object.

b) Delete edges shared by only two face-loops which are on the same face, because an edge is the intersection of two different faces in an object.

Finally, the orthographic views of the object generated above are compared with the input three views to establish whether it is a solution or not. All the solutions matching the input two-dimensional views can be found by checking all the assemblies of the object-loops.

The main disadvantage of Kaining's method is that the range of object that may be treated is limited to those having planar and cylindrical surfaces only.

4.5) HO BIN'S METHOD:

The basic approach adopted in this method is that parts can be thought of as an assemblage of elementary volumes or 'solid primitives', which also forms the Constructive Solid Geometry representation of those objects, (see section 3.4.4). Ho Bin's approach is based on interactively inputting these representations directly from the two-dimensional orthographic views. The interpretation process may be described as a semi-automatic rather than automatic process since most of the tasks of recognizing each primitive from the input views are carried out by the user rather than by means of algorithms such as the ones employed in the methods described above. The amount of input required from the user is rather large and for each primitive the basic input cycle consists of four steps:

STEP 1: Input the type of primitive. Five types of primitives can be input: cuboid, tetra pyramid, cylinder, cone and sphere. All of these are defined so as to have their axes (or heights) perpendicular to one of the projection planes, or oblique to two of the projection planes.

STEP 2: Input the sign of the primitive. If the primitive represents a solid, part of space, its sign must be '+' (positive). If the primitive represents a hole or cavity, i.e. a "virtual" primitive, then its sign is '-' (negative).

STEP 3: Input three points for the base of the primitive.

STEP 4: Input two points for the height of the primitive. The first input point of the height is regarded as the projection of the points located on the primitive base contour.

Steps 1, 2 and 3 are carried out using a menu of commands, shown in Figure 4.13, on a digitizer. The five points of base and height (steps 3 and 4), are input in a prescribed order from the given engineering drawing.

At the end of each input cycle, a three-dimensional model of the corresponding primitive is constructed and the two-dimensional representations, comprising three orthographic views and a hidden-line isometric view of the primitive) are displayed on the output device. This feed back enables the user to check whether his input of that primitive is correct before beginning the next input cycle. This interactive process is continued in this fashion until the complete Boolean tree of the object model is obtained.

The algorithm concentrates mainly on :

i) using the type and sign of all the primitives sequentially to build simple Constructive Solid Geometry expressions of the kind A-B+C-D+E, where A, B, C, D and E

Command	Type	Sign
Continue	Cuboid	SOLID (+)
Delete	Pyramid	
Hard copy	Cylinder	
STOP	Cone	VIRTUAL (-)
	Sphere	

Fig. 4.13: Menu of commands

represent the primitives, and the signs '+' and '-' represent the Boolean operators Union and Difference respectively. The special Boolean tree along a single direction with only two operators (+ and -) may be obtained, as shown in Figure 4.14.

ii) using the coordinates of the input five points (steps 3 and 4) to derive the following data:

a) the number of the view on which each primitive base and height are projected,

b) the three coordinates of the base centre,

c) the radius of the base circle, if the primitive is a cylinder , a cone or a sphere,

d) the length, the width and the angle (between the length direction and the horizontal line) of the base rectangle, if the primitive is a cuboid or a pyramid,

e) the value of the angle between the axis (height) and the XY projection plane, or between the axis (height) and the XZ projection plane if the axis is parallel to XY plane,

f) the value of the primitive height. For the sphere, which is a primitive that has no 'height', the two input points are used to define the third coordinate of the centre of the sphere.

The algorithm may be described as follows:

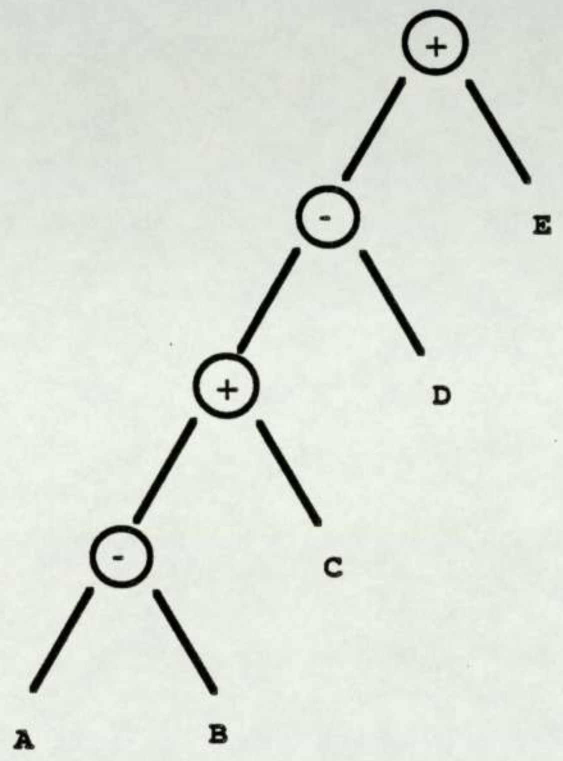


Fig. 4.14: Two operators CSG tree along a single direction

i) If the axis of the input primitive is perpendicular to one of the projection planes, the real shape of its base contour and the true length of its height are shown in the principal views. From the three input points of the base, for instance points 1, 2, and 3 in Figure 4.15, the parameters of the base are obtained easily. These parameters are :

- for the rectangular contour: the length, width, the angle β between the length direction and the horizontal line, and the coordinates of the centre,
- for a circular contour (eg. Figure 4.16): the radius and the coordinates of its centre.

From the two input points of the height, the value of the height is obtained by subtracting the coordinates along the coordinate axis parallel to the primitive axis.

ii) If the axis of an input primitive is oblique to the projection plane, the real shape of its base no longer appears in the principal views. For instance, the projection of a circle or a rectangle becomes an ellipse or a parallelogram, respectively. In this case it is necessary to reconstruct the real shape in order to obtain the input primitive dimensions, construct its geometric model and draw its three orthographic views and isometric view. A cone, and its orthographic views, shown in Figure 4.16 is chosen here as an example to illustrate this

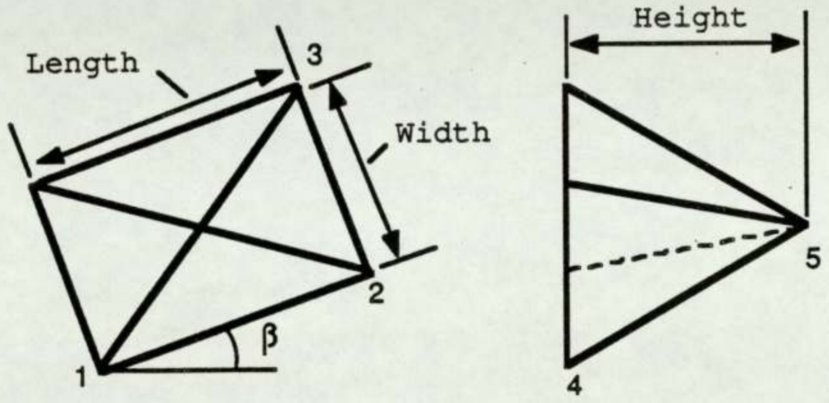


Fig. 4.15: Views of an input PYRAMID primitive

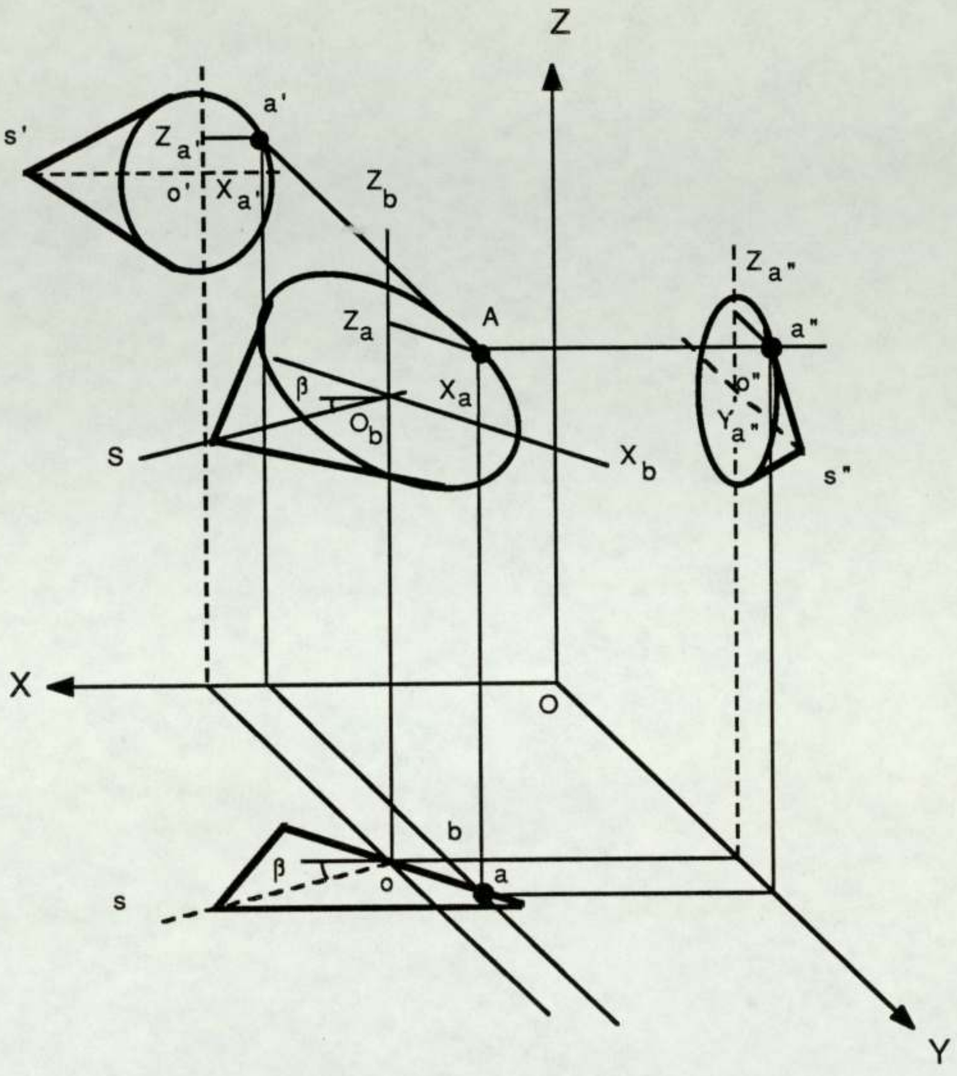


Fig. 4.16: A primitive cone and its three principal views

transformation algorithm.

The axis SO_b is parallel to the XY projection plane and has an angle β with the XZ projection plane. An arbitrary point A located on its circular base contour is projected on the three projection planes as points a, a' and a". O_bX_b and O_bZ_b are the reference coordinate axes of the circular base contour of the cone. The X_b coordinate of point A, AZ_a, on the reference coordinate axis O_bX_b, is equal to ao in the top view. The X and Z coordinates of projection point a' relative to the cone centre o' in the front view, are $a'Z_{a'}$ and $a'X_{a'}$. The Y and Z coordinates of the projection point a" relative to o" in the side view are $a''Z_{a''}$ and $a''Y_{a''}$. The X and Y coordinates of the projection of point a to the projection o in the top view are bo and ab :

Since:

$a'Z_{a'} = bo,$

$a''Z_{a''} = ab,$

then:

$AZ_a = a'Z_{a'} / \sin \beta$

or:

$$AZ_a = a''Z_{a''} / \cos\beta$$

and:

$$AX_a = a'X_{a'} = a''Y_{a''}$$

It is therefore possible, by using the above transformation algorithm, to calculate the coordinates Xb and Zb of an arbitrary point A from the coordinates of its projection points. However, this is only possible if the coordinates of the centre point of the ellipse projection are known. After transforming the coordinates of the three input points of the ellipse projection, a circle base contour of that cone is determined.

4.6) DISCUSSION:

The first fundamental point that emerges from the above review is that some of the methods adopted an approach based on the Boundary representation scheme while others used Constructive Solid Geometry representation. The former methods require thorough checks on the validity of the reconstructed object to be carried throughout the interpretation process. These checks hold an important place in the algorithm and they serve to identify and eliminate pathological cases which may lead to the generation of impossible objects. Those methods which adopted Constructive Solid Geometry do not require such checks since the validity of the reconstructed object is guaranteed, whether it is a solution or not. However, one can argue that such methods do not attempt 'automatically' to interpret orthographic views as a solid object; instead they rely heavily on the user to carry out the most difficult task in the interpretation process, i.e. the identification of the three-dimensional primitives. The other main disadvantage of adopting this approach is the speed at which the interaction between the user and machine is carried out. However, there is now a tendency to adopt such user-guided interpretation techniques, may be because of the increasing processing power in terms of speed present machines can offer, or maybe because researchers have come to the conclusion that human parallel processing power can never be matched by any algorithm in solving such problem; Such attitude has been reflected by Aldefeld and Richter in their work on semi-automatic three-dimensional interpretation of line drawings [43].

The other point common to all the methods developed so far, is that whatever the approach or modelling representation adopted in each method, they all fall short of what is really needed in engineering practice. This is either because there is a significant number of failing cases, or because the scope of the technique is insufficient for mechanical engineering; there is still a wide range of mechanical parts that can not be reconstructed because of their complexity.

A novel approach, based on Constructive Solid Geometry concepts, has been adopted in the work reported in this thesis for the automatic interpretation of engineering drawings as solid models. The method developed in this project also uses the 'man in the loop' concept but not to the same extent as the so-called "semi-automatic" techniques described above.

CHAPTER FIVE

INTERPRETATION OF ENGINEERING DRAWINGS AS SOLIDS: A NEW APPROACH

5.1) INTRODUCTION:

A new approach has been adopted by the author to solve the problem of automatic interpretation of orthographic projections as solid objects. The basic concepts underlying this approach are discussed in section 5.2. The manner in which these concepts are applied to the present work is novel. A process of converting orthographic views into a solid model has been developed by initially implementing these concepts to uniform thickness, or prismatic, objects. Experience gained from the work on prismatic objects has yielded a technique for implementing those same concepts to more general 3D objects.

An overview of the process is presented in section 5.3, and its implementations to both prismatic and general 3D objects are described in section 5.4.

5.2) THEORETICAL FUNDATIONS FOR THE PROCESS:

The approach being used exploits the concepts of Constructive Solid Geometry in which a complex object is considered to be an assemblage of three-dimensional primitive elements, or building blocks, synthesized by means of Boolean operations to represent the complete object. Further, each primitive may be represented by the two-dimensional elementary patterns, referred to as *primitive* loops, contained in its set of three orthographic views. This set of patterns constitutes a unique 'signature' which identifies the primitive within a tree structure used to describe the total object. For instance, a set of three rectangles (one rectangle in each view of the primitive) will identify a primitive block, or a set of three circles (one circle in each view of the primitive) will identify a sphere, as shown, along with other signatures, in Figure 5.1.

The manner in which these concepts are applied to the present work is novel. The starting point is to assume that an object can be 'cut out' from a single 'raw block' rather than being built up of several building blocks. The raw block is itself a three-dimensional primitive (a unit block) which is transformed (scaled) and to which a sign (positive) is allocated to represent a volume of material from which the object is cut out. The task of reconstructing the object from its orthographic views is then to find and identify the volumes of material (the three-dimensional primitives) to be removed from the raw block to yield the true object. Figure 5.2 illustrates the reconstruction of an L-shaped model starting from the raw block at the bottom, using the above concept. It will be observed that only subtractions of primitives are required in the total process.

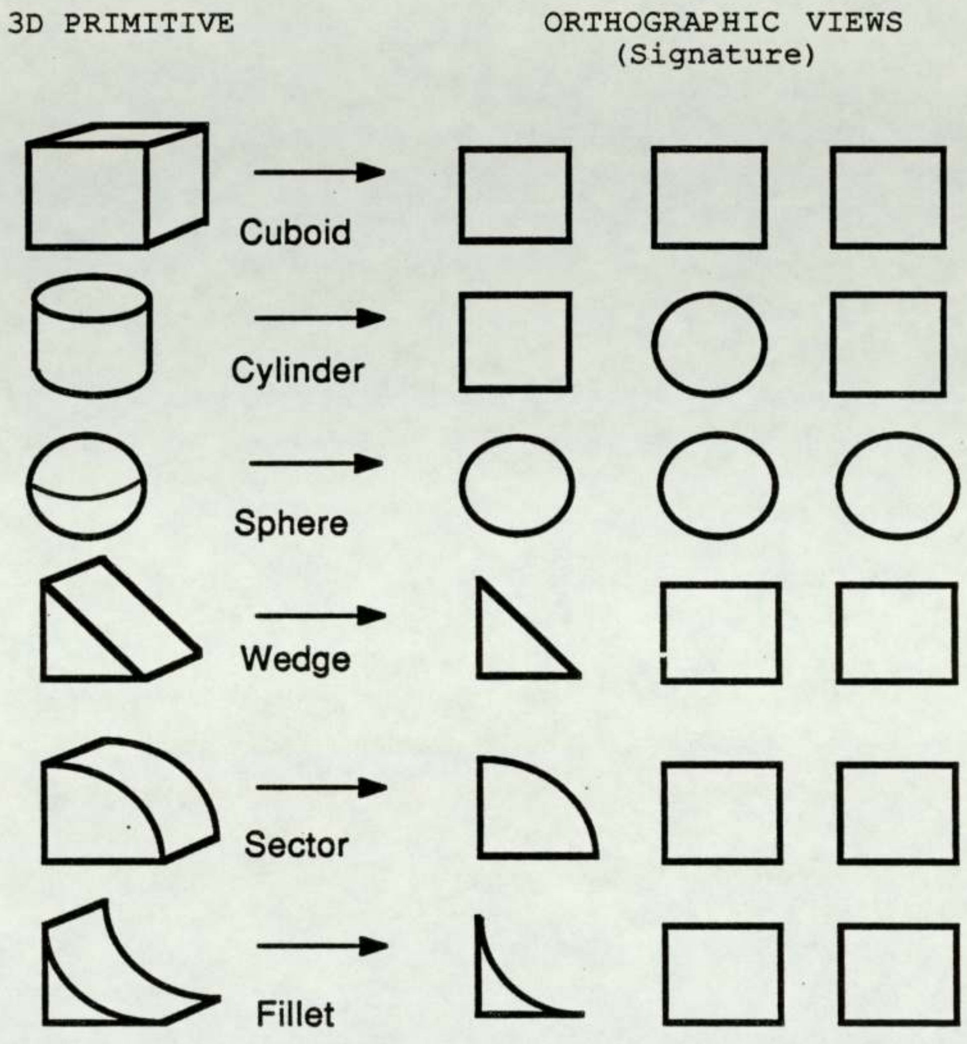


Fig. 5.1: 3D primitives and their othographic projections (signatures)

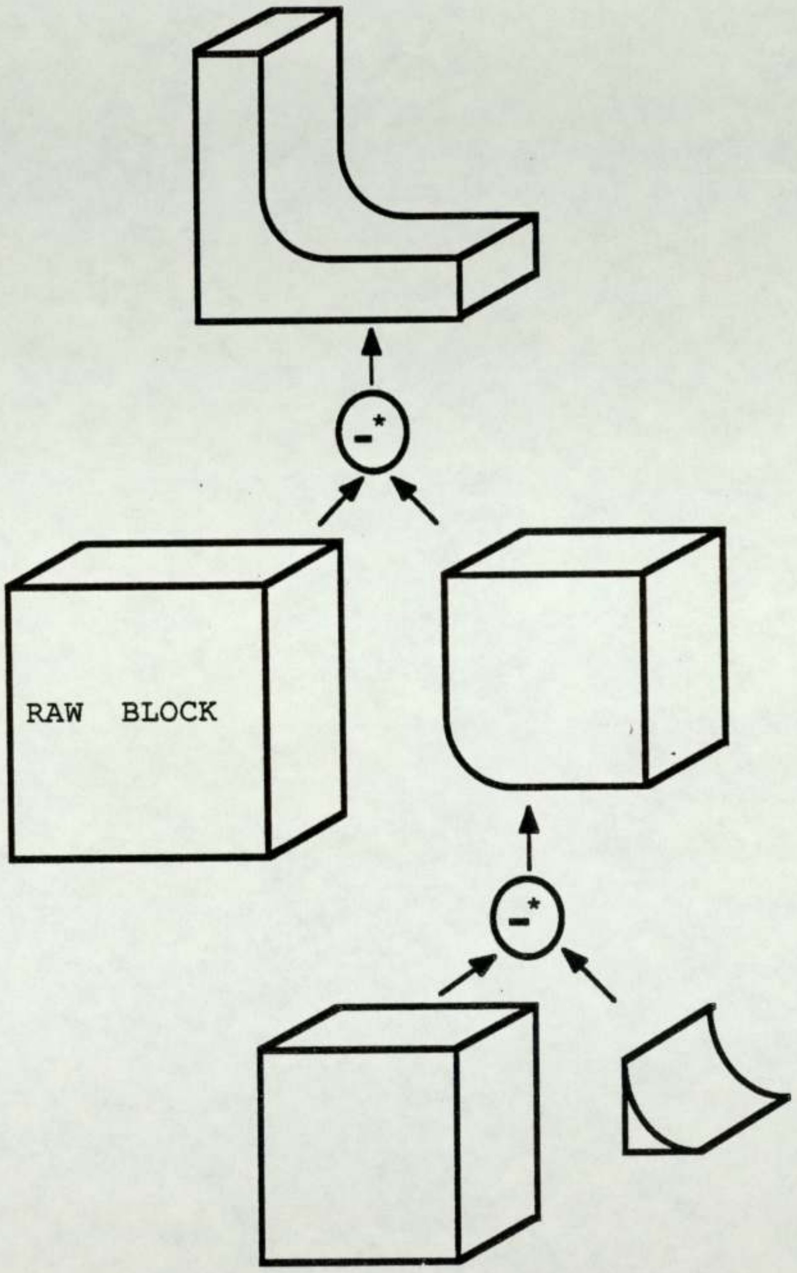


Fig. 5.2: Object 'cut out' from a 'Raw Block'

5.3) OVERVIEW OF THE PROCESS:

The process of transforming the orthographic views of an object into a formal three-dimensional representation has been designed to comprise five elements (stages or subprocesses), requiring a minimum of user interaction, and to provide feed back of data corresponding to discrepancies between input and output orthographic views, when the need arises. These subprocesses are:

a) Raw Data Interpretation

b) Data Analysis

c) Solid Modelling

d) Output Verification

e) Feed back

The flow diagram of Figure 5.3 shows the relationship between these elements in the forward path of a closed loop where feed back is provided. The human operator interaction with the process resides after the last subprocess in the forward path of the cycle, i.e., before feed back. A description of each subprocess is given in the following sections while the details of the technique employed in each of these subprocesses and associated algorithms are described in chapter 6.

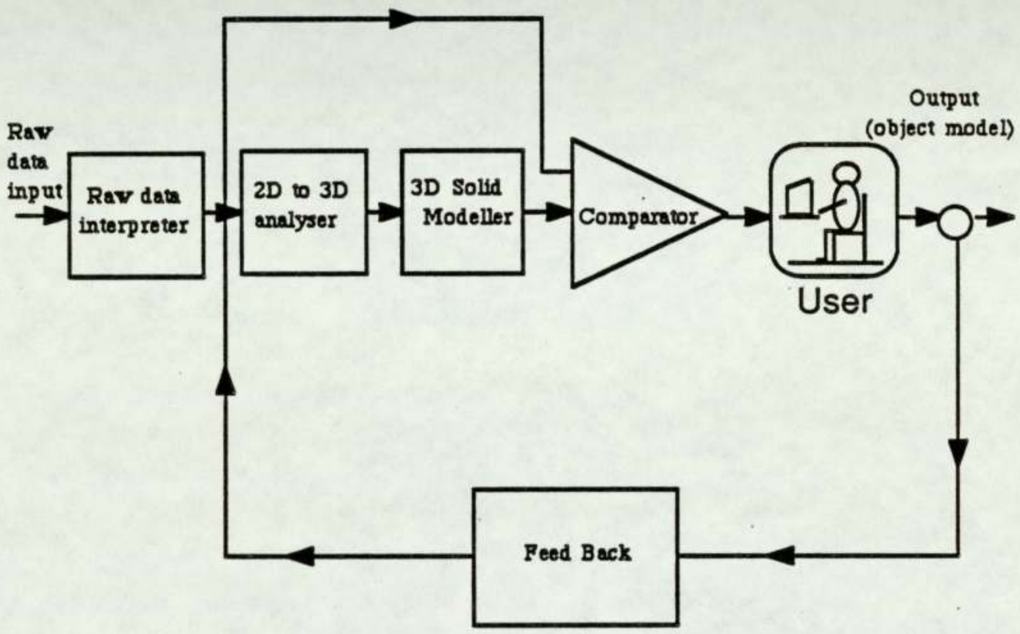


Fig. 5.3: Process main components

5.3.1) TERMINOLOGY:

The following terms are used throughout the description of the interpretation process, and are illustrated in Figure 5.4:

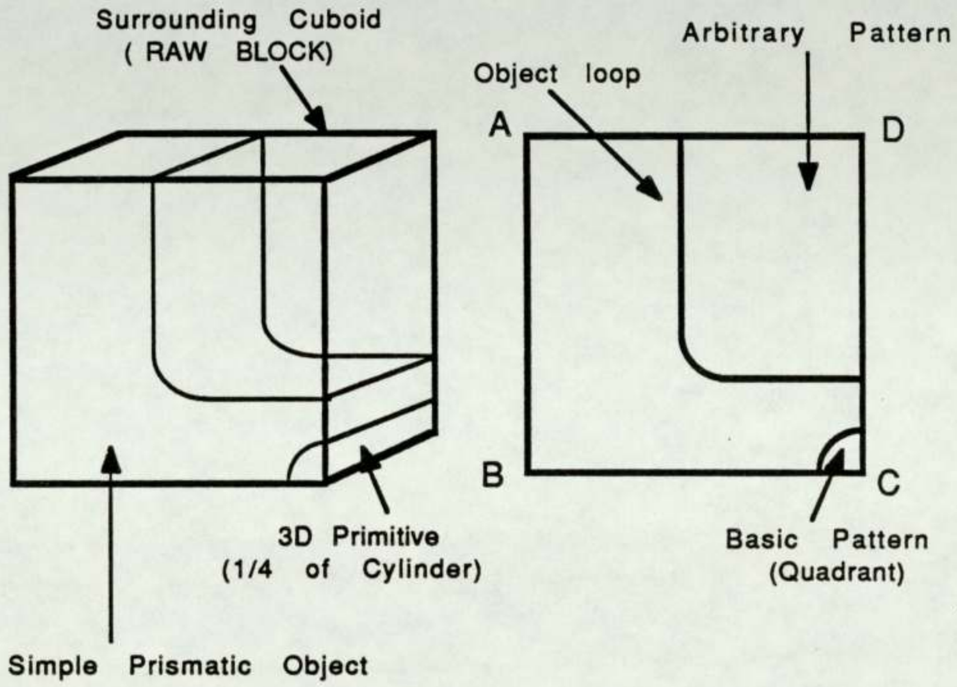
a) The *Surrounding Cuboid* is the three-dimensional block from which the object is to be cut, i.e. the raw block (see section 5.2). Therefore, it is a cuboid whose dimensions equal those of the maximum values for the solid object in the X, Y and Z directions.

b) The *Surrounding Rectangle* is the closed loop representing an orthographic projection of the Surrounding Cuboid.

c) The *Object Loop* is the closed loop defined in an orthographic projection of the solid object.

d) The *Perimeter Loop* is the object loop that defines the outline of the object when viewed in the direction of projection.

e) A *Subobject Loop* is a closed loop formed between the surrounding rectangle and the object loop. It may also be defined as an orthographic projection of a subobject formed by difference between the surrounding cuboid and the solid object.



ABCD = Surrounding Rectangle

Fig. 5.4: Terminology

f) A *Basic Loop* is a closed loop which can be identified as an elementary two-dimensional shape, such as a rectangle, a quadrant, a circle, a triangle ... .

g) A *Primitive Loop* is a basic loop that is an orthographic projection of a three-dimensional primitive. It may also be a subobject loop.

h) An *Arbitrary Pattern* is a closed loop which cannot be identified as an elementary two-dimensional shape and which requires further processing in order to generate Basic Patterns.

i) A *Parent Loop* is a closed loop which has been identified as an arbitrary pattern and then, directly decomposed further into a number of subobject loops or *children* loops, i.e. a parent loop can be directly reconstructed by using its children loops. An object, or a subobject, loop may be either a basic or an arbitrary loop.

5.3.2) <u>RAW DATA INTERPRETATION</u>:

The purpose of the raw data interpreter is to check for incomplete, inconsistent or false information, such as, for instance, edges of order less than 2 (known as dangling edges, discussed in section 3.4.4), or for self-intersecting loops, in the input orthographic views. The data of the orthographic views are assumed to have already been stored in the computer in the form of three separate data files (one file per view). The raw data interpreter is also used to transform the raw data into a structured format required by the next step.

5.3.3) <u>DATA ANALYSIS</u>:

In the analysis stage, the data are examined in order to:

i) identify the class of the object, i.e. to determine whether the input views represent a primitive object (the trivial case), a prismatic object or an arbitrary and more general object. The output of this step predetermines the next steps of the process since, for efficiency in processing, the implementation of the interpretation process differs from one class of objects to another although the process itself is the same in all cases.

ii) to extract the object and subobject loops in order to locate and identify all the basic two-dimensional patterns from each orthographic view and the 'signatures' they may form as a set to establish the identity of the three-dimensional primitives. The transformations and associated Boolean operations, necessary for the reconstruction of the shape of these loops, are also defined at this stage. The underlying technique uses the knowledge about a number of predefined elementary or 'basic' patterns. Those loops that can not be readily identified as basic patterns are classified as arbitrary patterns, or parent loops, which are then decomposed further until all the loops are identified as basic patterns. The details of this recursive technique, including the algorithms associated with it, are presented in chapter 6.

The application of this technique to the interpretation of prismatic and ortho-prismatic objects provides the fundamental basis from which generalization to more general three-dimensional objects can be developed. Details of the implementations of the process to each class of objects, including the trivial case where the object is itself represented by a single primitive, are discussed later in section 5.4.1.

The output from this stage is a text file which contains the identified primitives and the required manipulations to provide all the necessary data for the reconstruction of the object in the solid modelling stage. These files must be written in a format appropriate to the solid modeller in use.

5.3.4) THREE-DIMENSIONAL MODELLING:

A true solid modeller is at the core of this process. It is incorporated, at this stage, to:

1) reconstruct the solid model by performing the various transformations and Boolean operations on the identified three-dimensional primitives.

2) generate a parametric ASCII data file which describes the two-dimensional orthographic views of the object model. These are used later for comparing the views of the generated model with the original orthographic views in order to assess the quality of the model and, if necessary, to refine it.

The solid modeller used in this project is the PAFEC 'BOXER' solid modeller. Like most contemporary modelling systems, it offers a finite set of concise, compact primitives whose size, shape, position and orientation are determined by a small set of user-specified parameters. The type and parameters of each primitive are specified, using the PAFEC 'BOXER' syntax, either interactively, or stored in a text file and then transferred to the solid modeller.

A particular primitive, such as the block illustrated in Figure 5.5(a) may be specified using the following text statements:

```
   BLOCK (Xlen,Ylen,Zlen )  AT (Xpos,Ypos,Zpos)
```

where:

Xlen = block length in the X direction

Ylen = block length in the Y direction

Zlen = block length in the Z direction

Xpos = block centroid X coordinate

Ypos = block centroid Y coordinate

Zpos = block centroid Z coordinate

A primitive cylinder, illustrated in Figure 5.5(b), must be defined using the following text format :

```
CYL (Cylen, Radius) AT (Xo, Yo,Zo)
```

where:

Cylen = length of the cylinder.

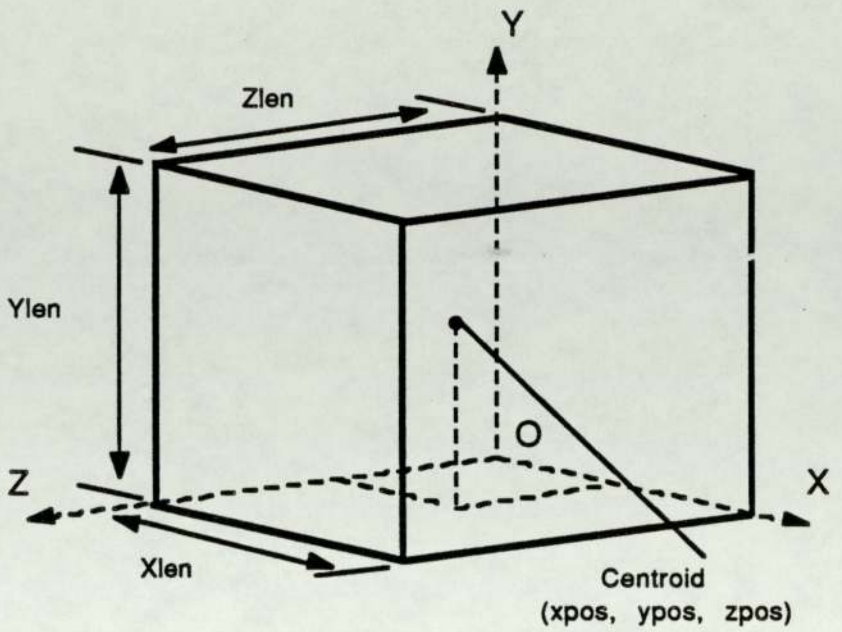
Radius = radius of the cylinder base

Xo = X coordinate of the centre of the base

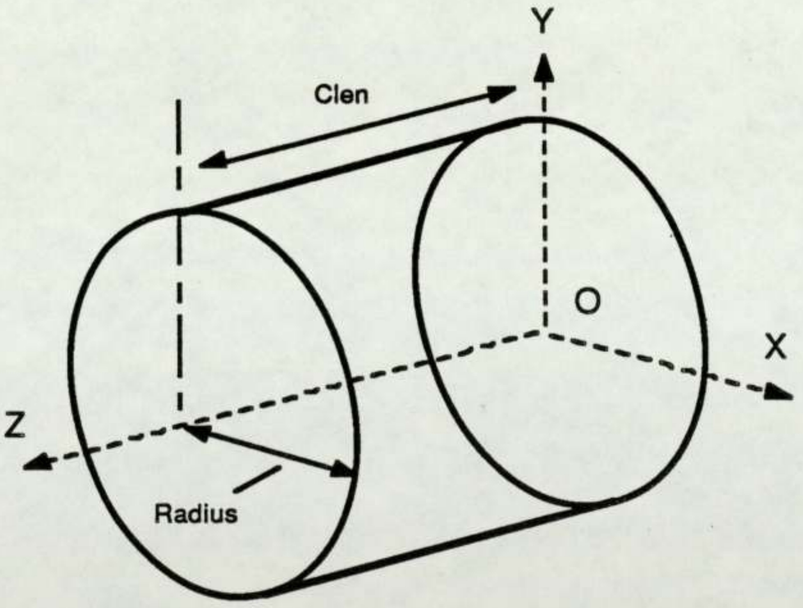
Yo = Y coordinate of the centre of the base

Zo = Z coordinate of the centre of the base

There are other formats which can be used to define a block, and a cylinder, but the above have been found to be most convenient, and are used here. Other common primitives such as wedge, fillet, cylindrical



(a)



(b)

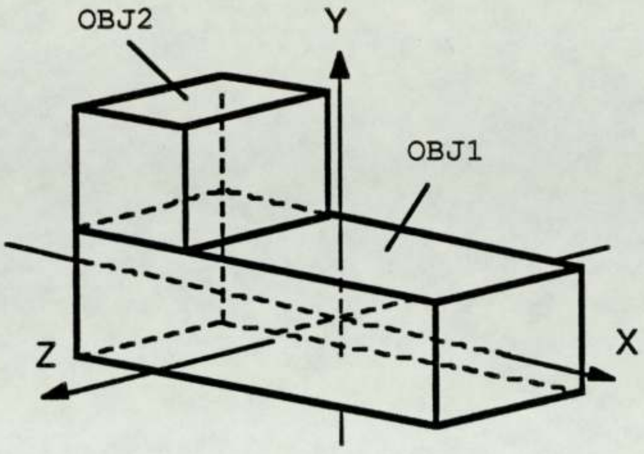
Fig. 5.5: a) Primitive CUBE definition
b) primitive CYLINDER definition

segment, sphere, tetrahedron, cone and torus, are also available. The format corresponding to each of these primitives may be found in [45].

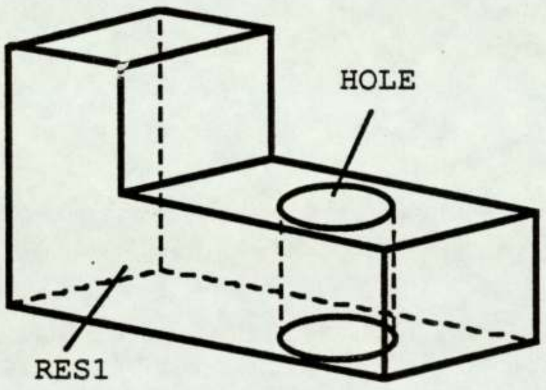
The solid modeller generates solid models by combining different primitives using a number of specified Boolean operators, such as UNION, DIFFERENCE and INTERSECTION, (see section 3.4.4). As an example, for the object illustrated in Figure 5.6, the following modelling statements are required:

```
LENGTH = 6.0
WIDTH  = 0.5
HEIGHT = 3.0
OBJ1 <- BLOCK (LENGTH, HEIGHT, WIDTH)
OBJ2  <-  BLOCK  (LENGTH/3,  HEIGHT,  WIDTH)  AT  (MOVEX =$
      -LENGTH/3,MOVEY = 1/2)
RES1 <-    OBJ1 +  OBJ2
DRAW  RES1
HOLE <-  CYL (1/2, 1) AT (1, 0, -1/2, ROTX =  90)
RES2 <-    RES1 -  HOLE
DRAW  RES2
```

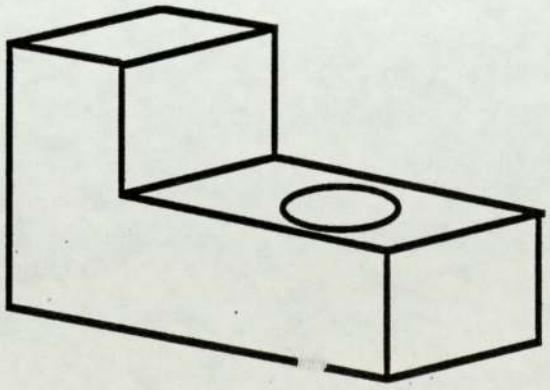
OBJ1 is a primitive block of length 6, height 1/2 and width 3, whose centroid is at: x = 0, y = 0 and z = 0, i.e. at the origin of the coordinate reference system.



a) OBJ1 and OBJ2 definitions



b) RES1 definition



c) RES2 definition

Fig. 5.6: PAFEC "BOXER" Output

OBJ2 is also a primitive block; length 2, height 1/2 and the width 3. This block is positioned by translating its centroid, from the origin, in the negative X direction over a distance equal to 2/3, and in the positive Y direction over a distance equal to 1/2. The " $ " sign is used to indicate a continuation of statement.

RES1 is the object resulting from the Boolean union, represented by the " + " sign, of OBJ1 and OBJ2. This object is then drawn, as shown in figure 5.6(a).

HOLE is a cylinder primitive whose length is equal to 1/2 and base radius equal to 1. The cylinder is positioned by defining the coordinates of the centre of its base, x =1, y = 0 and z = 1/2, and rotating it by a 90 degrees angle about the X axis, as shown in Figure 5.6(b).

Finally, RES2 is the object resulting from the Boolean difference, represented by the " - " sign, of RES1 and HOLE. RES2 is then drawn, as shown in Figure 5.6(c).

The input to this subprocess comprises the data files obtained from the analysis step. The size, shape, position and orientation parameters of primitives whose type has been defined, are here specified by extracting the necessary data from these files. The solid modeller input file is then generated by converting the data associated with each primitive and corresponding Boolean operators, into the appropriate syntax. The process of generating the solid modeller input file is discussed later in section 6.3.

The output from the solid modelling subprocess is:
a) a set of data files which comprise the topological and geometrical data to describe and display the object model.
b) an ASCII file which describes the orthographic views of the object model.

5.3.5) OUTPUT VERIFICATION:

The output verification is the final stage in the forward path of the process. This subprocess has two inputs:

1) the orthographic projections data obtained from the raw data interpreter,
2) the text file, generated by the solid modeller, from which the data corresponding to the orthographic projections of the output model are extracted.

These two sets of data are compared to establish any discrepancy between the three-dimensional model generated by the process and the actual object model. If the two sets of data agree the interpretation process is automatically terminated on the basis that the exact object has been reconstructed. However, in the case where there are some discrepancies, a decision has to be made on whether to terminate the process, since the discrepancies may be deemed negligible, or to allow it to continue for further iterations. Two options are open here:

a) The discrepancies are indicated to the user, who is then prompted to make a decision on whether to terminate the process or to allow it to continue. The user may terminate the process if he decides that the model generated (approximation model) is similar to the actual model to within the required tolerances, or he may require the process to continue until the exact model is reconstructed or until the tolerance conditions between the approximation model and the true object are met.

b) Use an objective function whose criterion for terminating the process consists of detecting when there only minor differences between two successive approximation models. The acceptable level of accuracy would then be dictated by the application for which the output model is required. For instance, in the case where the level of accuracy is not required to be high, such as for preliminary design, a simple value, such as volume, or mass, whose decrement has reached a certain level between successive approximation models may provide a practical test of model acceptability. These points are discussed further in chapter 9.

When the process is allowed to continue, the two sets of data (original and approximation model orthographic views data) are passed onto the feed back process.

5.3.6) FEED BACK:

In the feed back stage, the two set of data are examined to extract the two-dimensional geometrical and topological information which comprises the sets of three orthographic views representing one or more subobjects. The orthographic views data of these subobjects are then fed back as input to the data analysis subprocess in order to reconstruct the corresponding subobjects. Once reconstructed, a subobject is then subtracted from the previous approximation model to either the exact object, or to generate a further, but more accurate, approximation model. This process may be continued to provide 'nth' order approximation models which may converge to the exact input model.

5.4) IMPLEMENTATION OF THE PROCESS:

The process briefly described above was first developed for the case of prismatic objects (section 5.4.2) and extended to ortho-prismatic objects. The interpretation process for more general three-dimensional objects has been achieved by recursively employing exactly the same technique used in interpreting prismatic and ortho-prismatic objects within an iterative process.

Real mechanical engineering components cannot usually be represented by a single three-dimensional primitive. However, parts within any object, such as holes or pockets, may be represented by single primitives. Thus, it is necessary that the process should also be able to deal with the trivial case where the object, or subobject, is itself a primitive. The implementation of the process to this special case is the first to be discussed in the following sections.

5.4.1) THE TRIVIAL CASE: THE OBJECT IS A PRIMITIVE

The interpretation of orthographic views which represent an object comprising a single three-dimensional primitive, is simplest in the case where the primitive principal axis is parallel to one of the coordinate axes. Then, the two-dimensional pattern in each view of the primitive will consist of only one single closed loop readily identified as one of a number of predefined patterns, such as those shown in Figure 5.1. These patterns must also form a valid predefined 'signature' in order to be acknowledged by the computer. If the primitive, i.e. the object, has an arbitrary orientation, then the case may arise where at least one view may comprise more than one loop, or one pattern may not be readily be identified as a predefined 'known' pattern, as shown in Figure 5.1. In that case, the object is classified as either a prismatic or arbitrary object.

5.4.2) <u>IMPLEMENTATION TO PRISMATIC OBJECTS</u>:

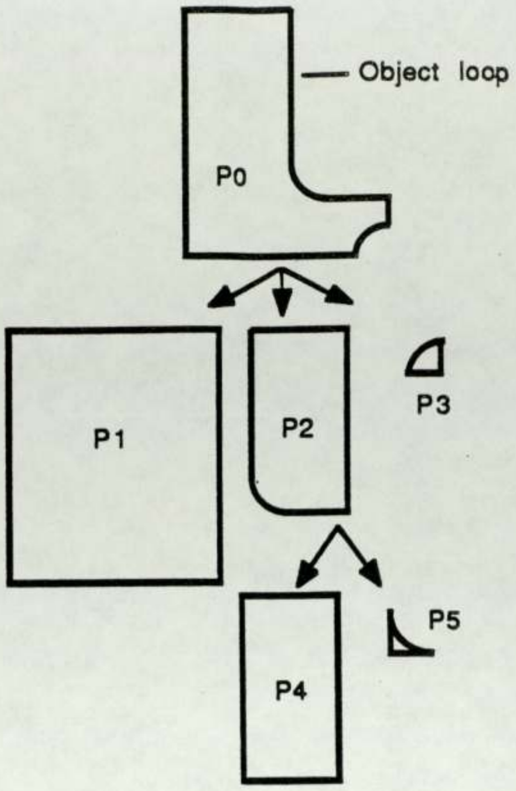
Any object that has at least one 'base' view which may consist of one, or more, closed disjoint loops, and two views which consist of only rectangular interconnected loops, is classified as simple or complex prismatic object, respectively (section 2.4.1).

The implementation of the interpretation process to both simple and 'complex' prismatic objects is basically the same, except that in the case of simple prismatic object only the perimeter loop in the base view needs to be analysed, while in the case of complex prismatic object all the loops including those corresponding to axial holes in the base view are analysed.

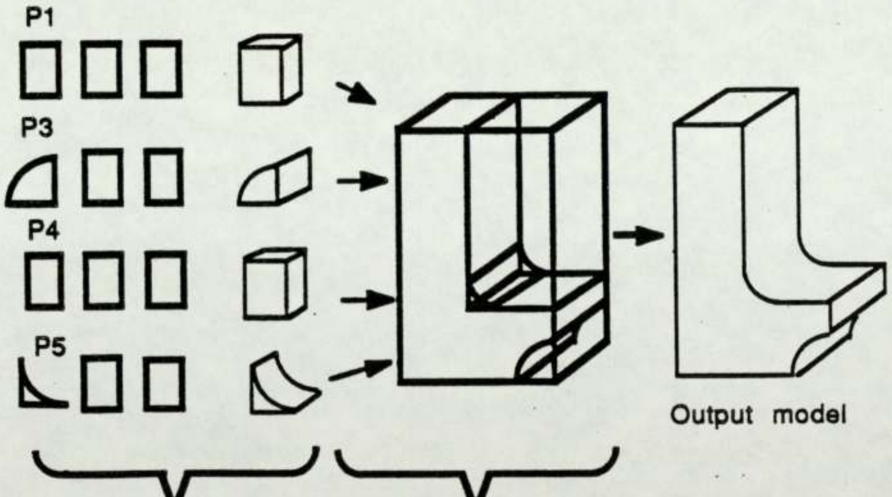
In both cases, only the base view is processed (since the other views are known to comprise only rectangles), and the analysis of each object loop consists of extracting the subobject loops formed between the surrounding rectangle and the loop itself in that view. The shape formed by each subobject loop is identified as either a basic pattern or as an arbitrary pattern.

Arbitrary patterns are then decomposed further until only basic patterns are identified. Figure 5.7(a), shows the perimeter loop in the XY view of a prismatic object, and the basic patterns generated during the decomposition process of each arbitrary pattern. The object loop is situated at the root of the Boolean tree, the arbitrary loops at its nodes, and the basic patterns at its leaves.

Each basic pattern is then associated with a rectangle in each of the remaining views to form sets of basic patterns which comprise the 'signatures', (see section 5.2), of a number of three-dimensional primitives. Thus a three-dimensional primitive is determined for each basic pattern identified in the base view. In Figure 5.7(b), the basic pattern P1 which is, in effect, the surrounding rectangle to the perimeter loop, P0, of the object, is interpreted as the loop obtained in the XY view of a primitive block. Similarly, the loop P4 is also interpreted as the loop obtained in the XY view of a primitive block, while a primitive fillet is associated



a) Decomposition of an object loop



b) Identification of 3D primitives c) 3D Modelling

Fig. 5.7: Processing of an object loop to generate a solid model

with the loop P5, and a primitive cylindrical segment, or sector, is associated with loop P3. These 3D primitives are then used by the solid modeller to produce the complete solid object model, as illustrated in Figure 5.7(c).

In the case of complex prismatic objects, there are hole and perimeter loops to be considered. Figure 5.8 shows the Boolean tree obtained by the decomposition of all the loops in the base view. Loop P2 which is a through hole loop, is identified as a circular basic pattern, while loops P1, the perimeter loop, and loop P3, another through hole loop, are identified as arbitrary patterns which are then decomposed further until all the loops are identified as basic patterns, i.e. loops P4, P6, P7, P8, P9 and P10.

The primitives identification and 3D modelling steps are similar to those described above. For instance, loop P2 will be identified as the orthographic projection in the XY view of a cylinder and P8 as the loop obtained in the XY view of a primitive block.

In the case of both, simple and complex, prismatic objects, discrepancies will not be found when comparing the input orthographic views with those of the generated model. The output of the process will be a Constructive Solid Geometry model which is the exact interpretation of

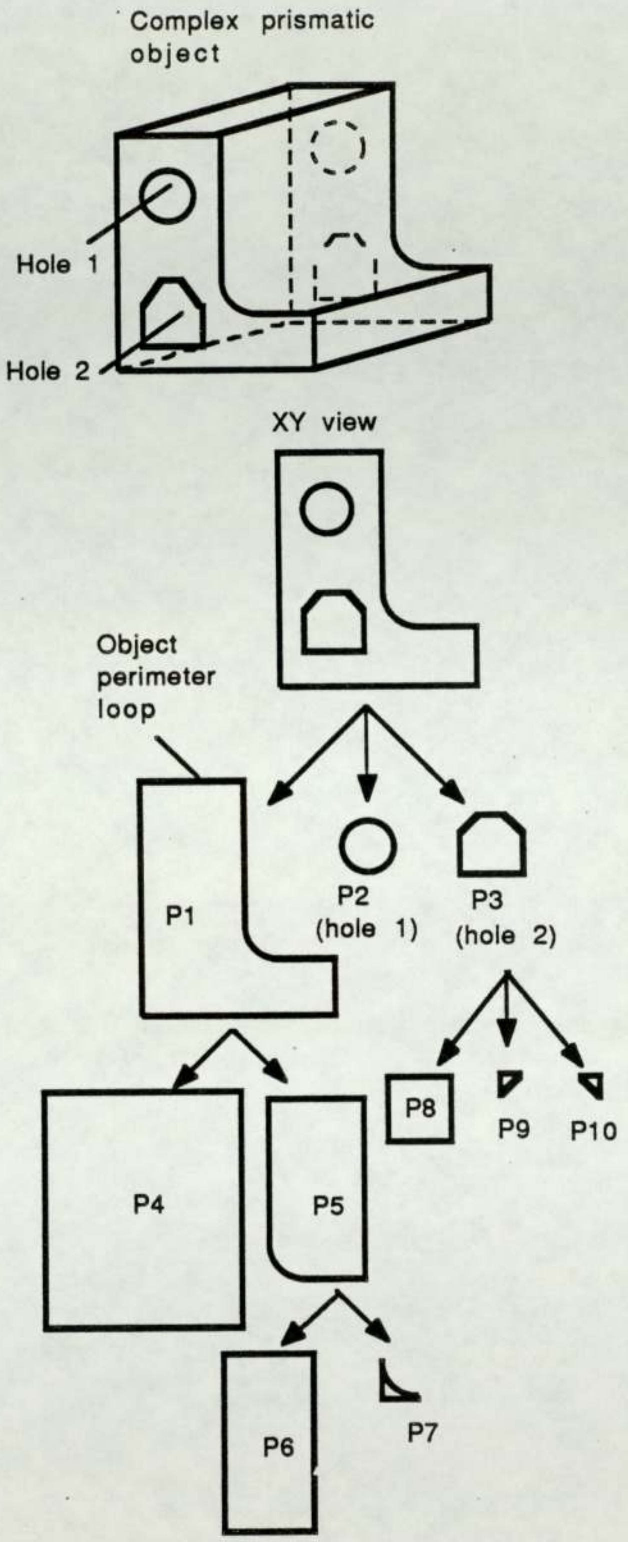


Fig. 5.8: Boolean tree of a complex prismatic object view

the input orthographic views, and thus, feed back will not be required.

5.4.3) <u>IMPLEMENTATION TO NON-PRISMATIC OBJECTS</u>:

The interpretation of non-prismatic objects, i.e. more general three-dimensional ones, requires a formal consideration of all views. Initially only one loop, the perimeter loop, shown in bold in Figure 5.9, is analysed in each view. Thus, each view is treated in a similar fashion to the base view of a simple prismatic object, and the inside loops are ignored. A prismatic object is obtained from each view . Such uniform thickness object is hereafter referred to as the X-profile, Y-profile, or Z-profile, depending on wether the view being analysed is the yz, xz or xy view, respectively. A *First approximation* model is defined as the intersection of these prismatic objects.

Figure 5.10(a), (b) and (c) show the three prismatic objects obtained from the analysis and three-dimensional modelling processes performed on the perimeter loop in each of the three views of the object shown in Figure 5.9. A prismatic object whose length in the Z direction equals Zmax, the maximum length of object also the length of the surrounding cuboid, in that direction, is produced by analysing the perimeter loop of

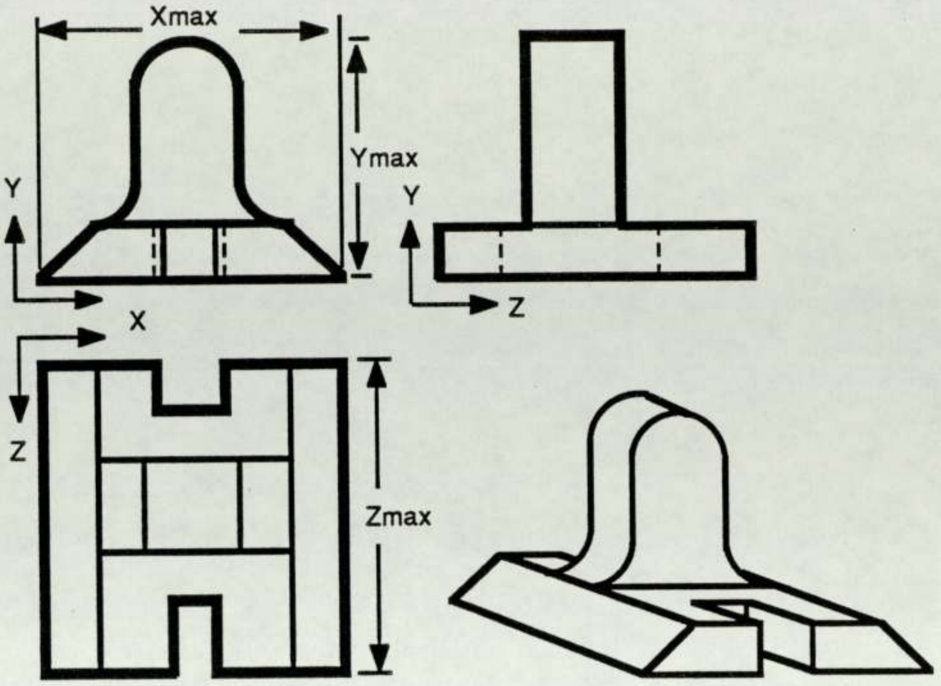


Fig. 5.9: An ortho-prismatic object and its orthographic projections

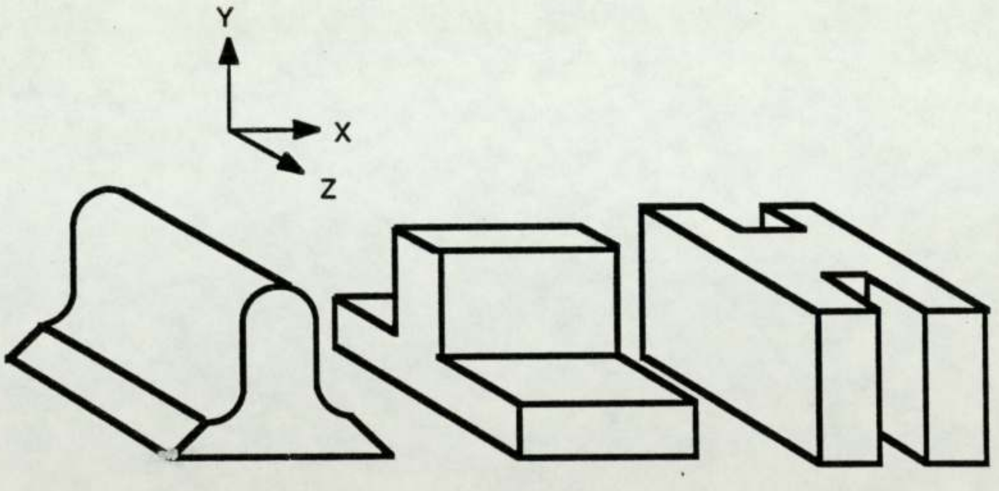


Fig. 5.10: a) Z-profile, (b) X-profile, and (c) Y-profile

its XY view, (the Z-profile), as shown in Figure 5.10(a). Another prismatic object whose length in the X direction equal Xmax, is produced by analysing the perimeter of its YZ view, (the X-profile), as shown in Figure 5.10(b). Similarly, a third prismatic object whose length in the Y direction equals Ymax, is produced by analysing the perimeter of its XZ view, (the Y-profile), as shown in Figure 5.10(c).

Figure 5.11 illustrates a solid model, referred to as the ZX model, obtained by the intersection of the Z-profile and the X-profile:

ZX-model = Z-profile $\cap$ X-profile

Figure 5.11 also illustrates the output solid model produced by the intersection of the ZX-profile and the Y-profile.

For an ortho-prismatic object, which is the case of Figure 5.9, the intersection shown is a complete description of the object represented by the input orthographic views. Similarly to the case of prismatic objects, the interpretation process to the case of ortho-prismatic objects results in the generation of the exact solid model, and thus the feed back step is not required.

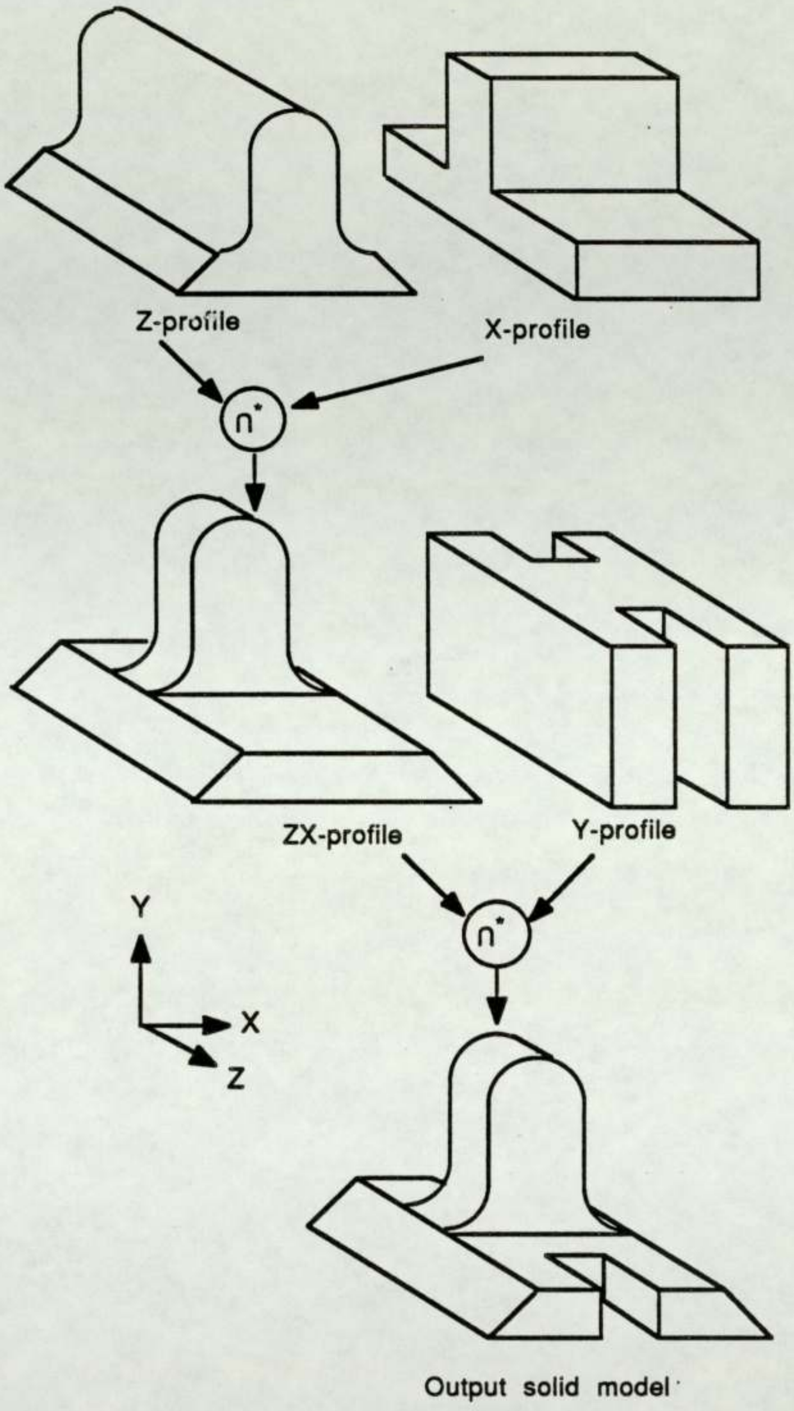
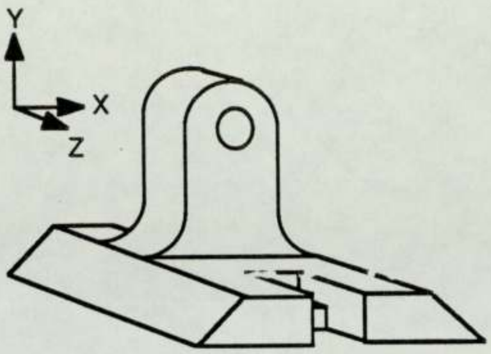


Fig. 5.11: Generation of a solid model

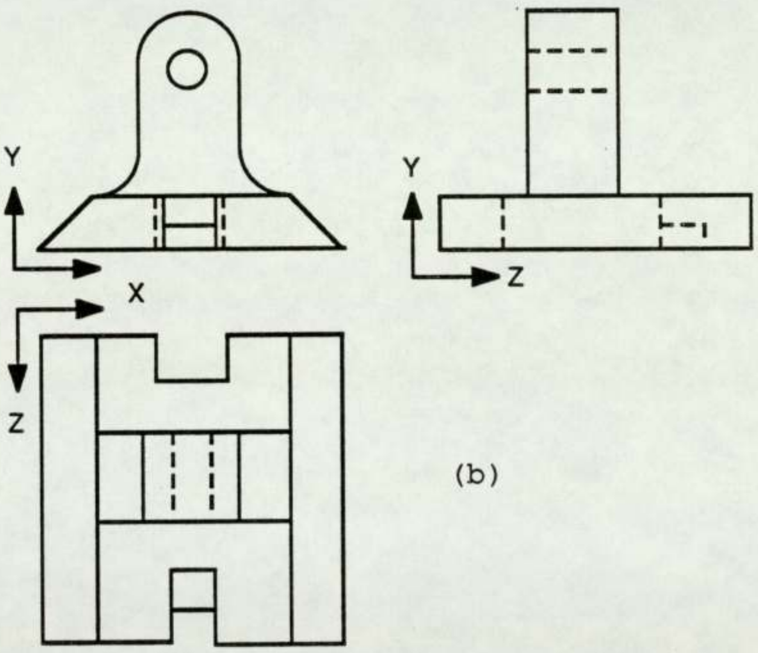
For all other objects (the great majority of mechanical engineering components), such intersections lead to a *First-approximation* model which is not exactly the object represented by the input orthographic views, but an approximate model only. For example, the intersection shown at the bottom of Figure 5.11, can be regarded as the First-approximation model of the three-dimensional object shown in Figure 5.12.

It is clear from the orthographic views of this intersection, i.e. the output views shown in Figure 5.13(a), that various details are either missing from it, and in some cases, added to it, when compared with the original input orthographic views, Figure 5.13(b).

In order either to generate a complete, or an 'adequate', object model, the differences between input and output views will have to be either completely eliminated, or reduced to within some agreed tolerance, respectively, by subjecting them to a minimization procedure. Such a procedure is initiated by the detection of these differences and a search is then carried out, at the feed back step, to extract from these differences sets of loops which may represent orthographic projections of one or more subobjects. These loops are then fed back to the analysis process where they are treated in similar fashion to the input orthographic views, i.e. to reconstruct the corresponding subobjects. Since the

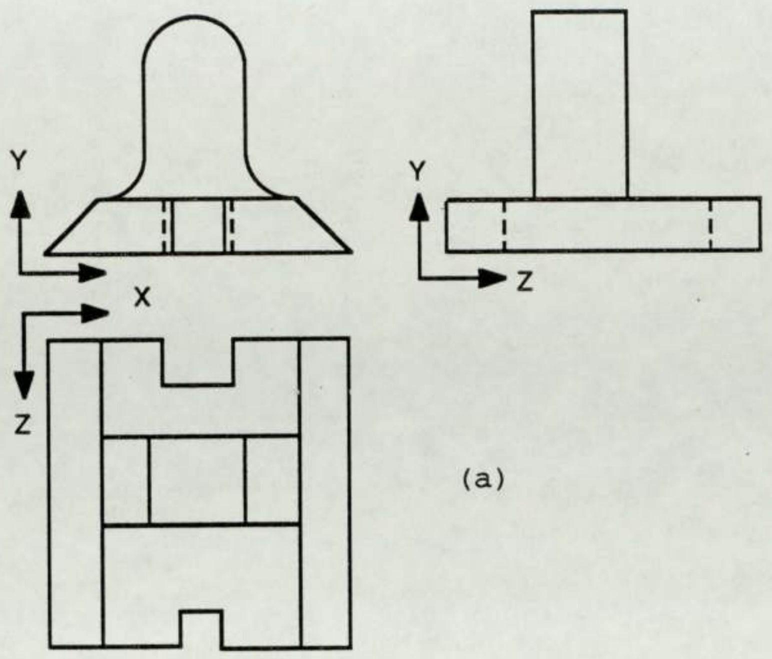


(a)

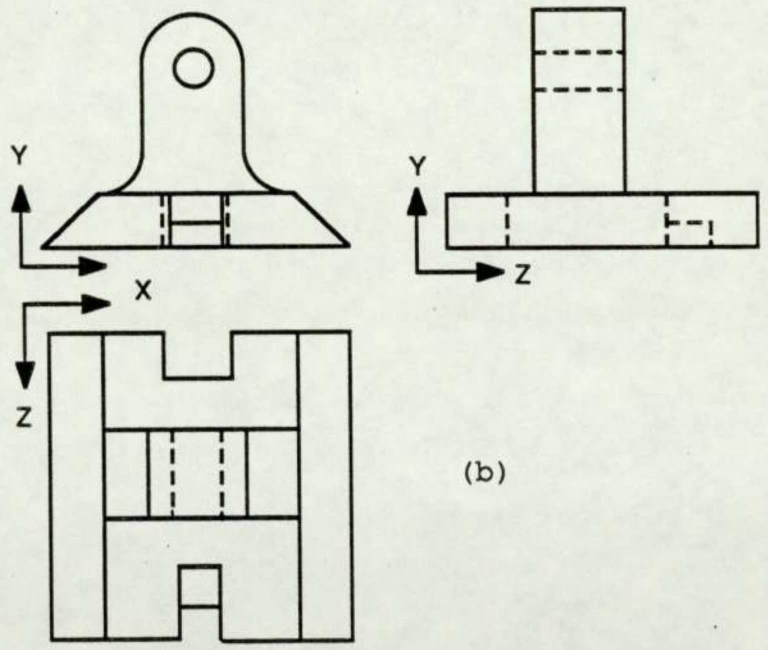


(b)

Fig. 5.12: a) A general 3D object and,
b) its orthographic projections



(a)



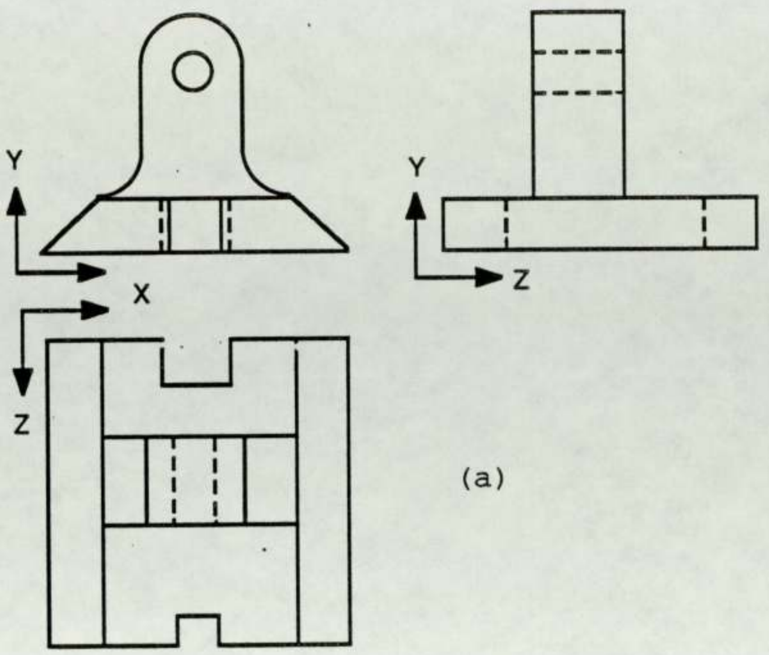
(b)

Fig. 5.13: a) Orthographic projections of 1st approximation model
b) Original input orthographic projections

First-approximation model represents, in effect, an envelop of maximum dimension, such subobjects, should be subtracted from it to generate an improved model. Figure 5.13(b) contains such sets of loops whose absence from the orthographic views of Figure 5.13(a), may be readily detected and identified as the signature of a cylinder primitive. This primitive is then subtracted from the First-approximation model and the orthographic views of the output model (the second approximation model) will be as shown in Figure 5.14(a); comparison with the input orthographic views, Figure 5.14(b), shows that a number of details are still missing from it. These details can be identified as the signature of a cuboid which should be subtracted from the already improved model. The orthographic views would then be as shown in Figure 5.15(a), which match the input orthographic views, Figure 5.15(b), thus confirming that a complete object model has been generated.

A summary of the implementation process to prismatic and non-prismatic objects is provided in the form of a flow chart as shown in Figure 5.16.

The basic approach and concepts adopted in the this project to develop a process which automatically reconstructs a solid model from a set of orthographic



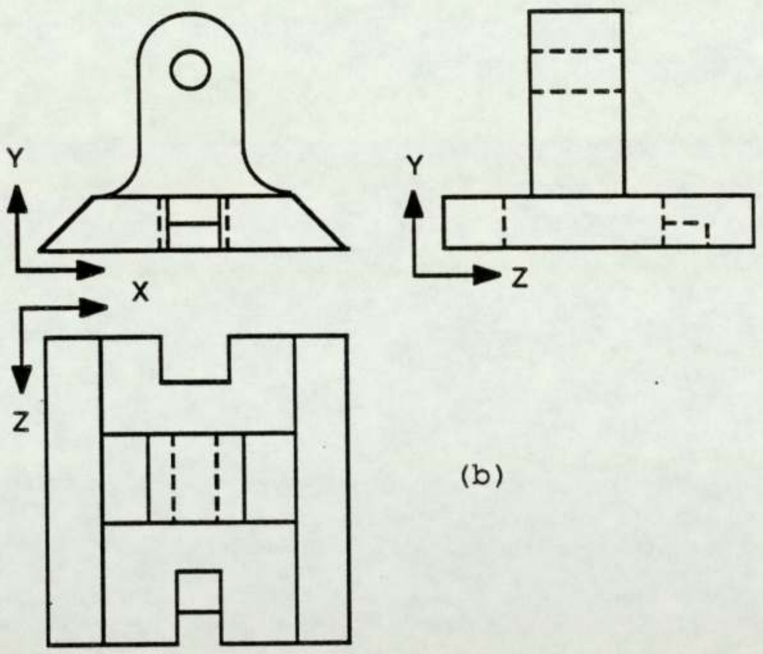
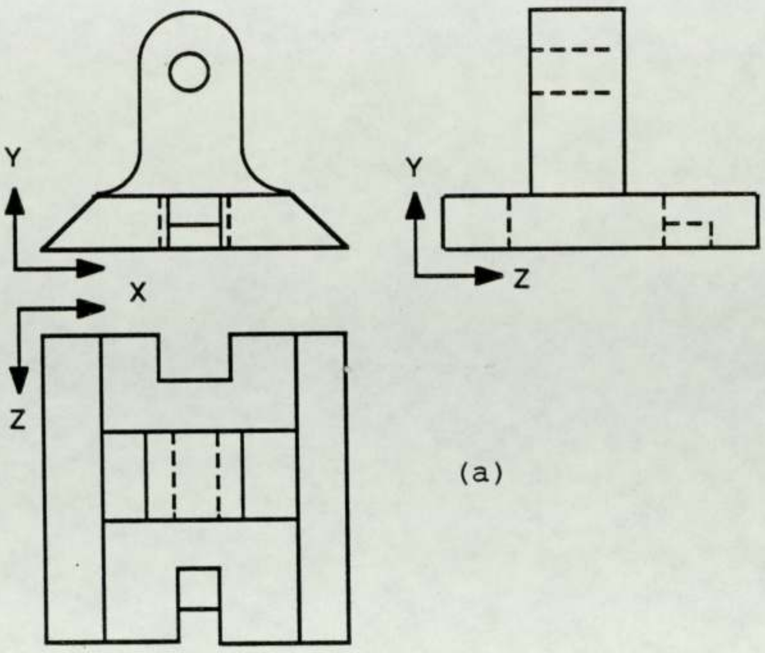


Fig. 5.14: a) Orthographic projectio.s of 2nd approximation model
b) Original input orthographic projections



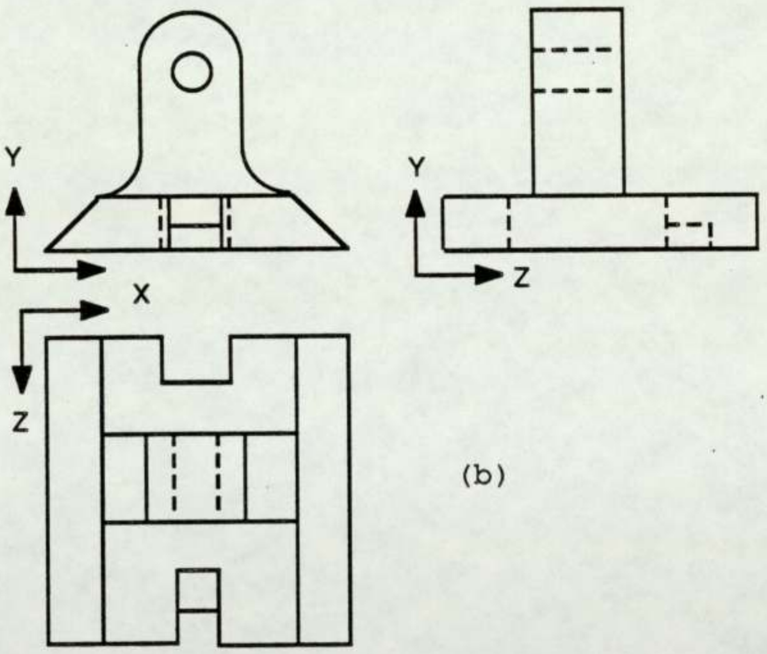


Fig. 5.15: a) Orthographic projections of final output model
b) Original input orthographic projections

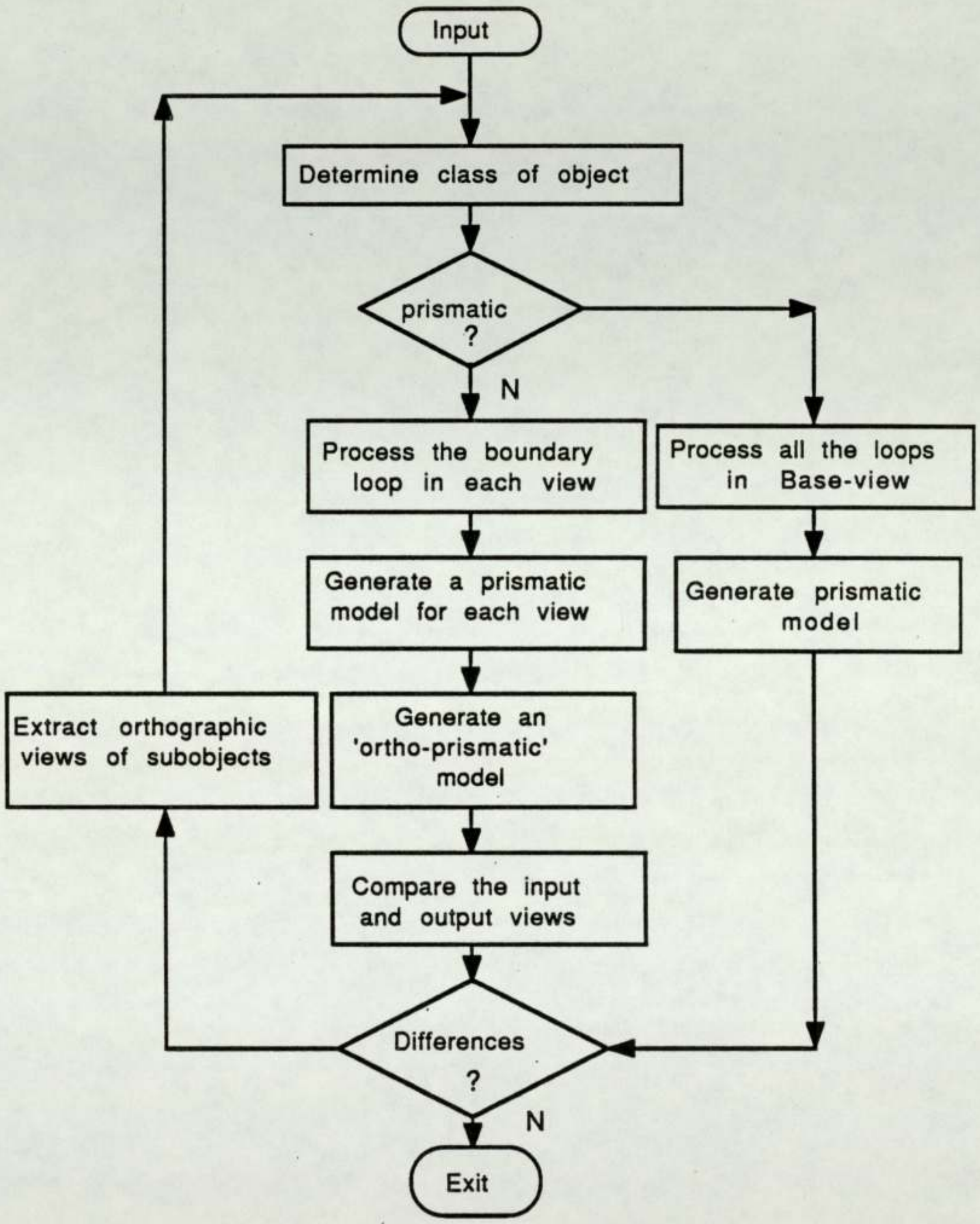

Input
Determine class of object
prismatic ?
N
Process the boundary loop in each view
Process all the loops in Base-view
Generate a prismatic model for each view
Generate prismatic model
Extract orthographic views of subobjects
Generate an 'ortho-prismatic' model
Compare the input and output views
Differences ?
N
Exit

Fig. 5.16: Interpretation process flow chart
(A summary)

views have been described. The application of such process to prismatic objects, and its generalization to non-prismatic ones have been presented. It has also been shown that the interpretation process consists of five distinct subprocesses. The corresponding formal algorithms developed, except for Raw Data Interpretation, are described in chapter 6. It was clear from the start of the project that time and effort must be concentrated on the development of algorithms related to the interpretation process, rather then its input. For this reason, the initial input orthographic projections, are assumed to be correct, complete and unambiguous. This point is picked up in chapter 9.

CHAPTER SIX

ALGORITHMIC INTERPRETATION OF THE PROCESS

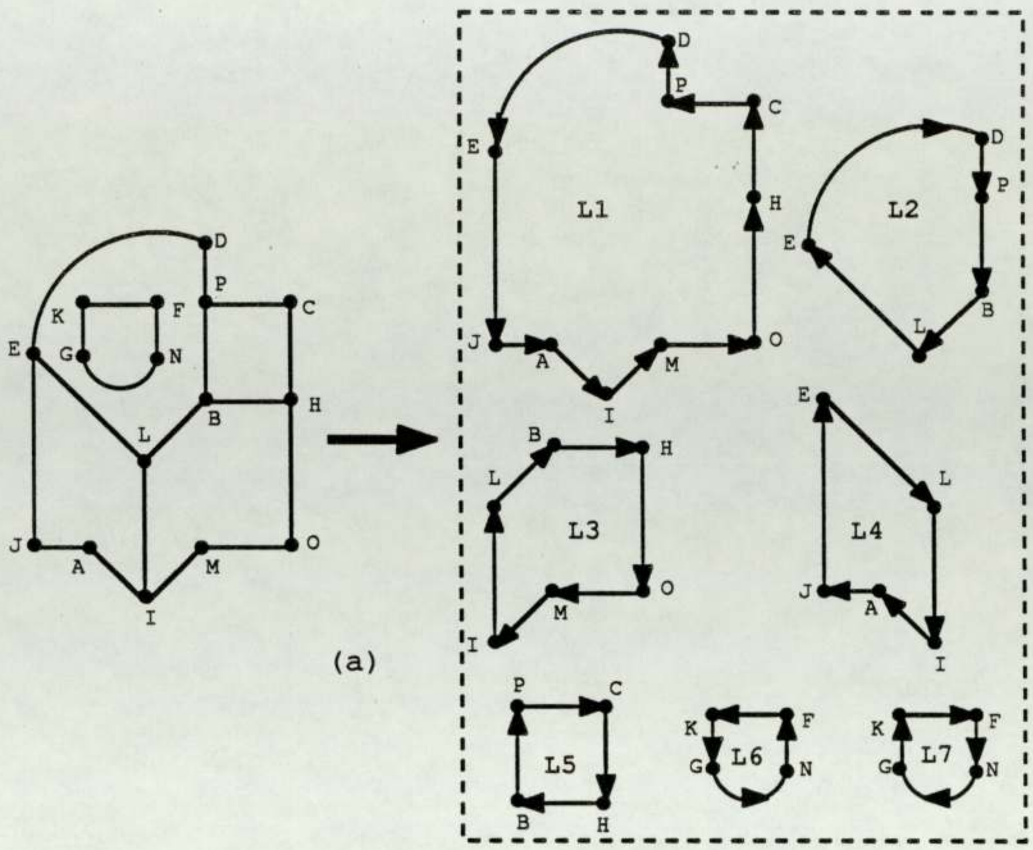
6.1) <u>DATA ANALYSIS ALGORITHMS</u>:

The technique used in the analysis process consists of extracting the object loop from each of the orthographic views and identifying the two-dimensional patterns and associated Boolean operations necessary for the reconstruction of the shape of this loop. These patterns are then interpreted as three-dimensional primitives which are combined to yield a uniform thickness (prismatic) object for each object loop that has been identified. The class of the object determines the view, or views, and loop, or loops, that are to be processed. Thus, the first task in the analysis stage, is to examine the data in each view in order to determine the class of the object which may either be a three-dimensional primitive, a prismatic object, or a general and arbitrary (non-prismatic) object. The class of any object is determined by the number, type and shape of loops in each view.

In general, an orthographic view of any object consists of interstitial spaces, each bounded by a closed loop, and each loop consists of a finite number of nodes and edges. An edge-following algorithm, based on the 'First-Right' rule, has been developed to examine these interstitial spaces in order to determine the number of loops in each view. The algorithm has been developed further to determine the type of each loop, i.e. whether it is a perimeter or an inside (connected or disjoint) loop. This algorithm is referred as the 'Loop Detector' and is described below. An algorithm has also been developed to identify the shape of each loop, i.e. whether the loop is a simple geometric shape (a basic loop), or an arbitrary shape (an arbitrary loop). This algorithm, referred to as the 'Loop Identifier', is designed to recognize geometrical characteristics. For example, a loop will be classified as a right-angled triangle if it has three straight edges and the angle between any two of them is equal to 90 degrees. The information generated by the 'Loop Detector' and the 'Loop Identifier' algorithms is used by another algorithm, the 'Object Classifier', to determine the class of the object.

6.1.1) **THE 'LOOP DETECTOR' ALGORITHM**:

Orthographic projections consist of a number of interconnected nodes and edges which may be regarded as the elements of a directed graph, or digraph (appendix A). Each edge within the graph is defined as a di-edge by a pair of ordered nodes: a start node and an end node. For instance, in Figure 6.1 which shows such a graph in the view of an object, node P and node C are the start and end nodes of edge number 8. Furthermore, each node has a degree which is equal or higher than two, i.e. each node belongs to two or more adjacent edges.



(a)

TOPOLOGY		
Edge no.	Start node	End node
1	J	A
2	H	O
3	B	H
4	L	B
5	A	I
6	E	D
7	J	E
8	P	C
9	D	P
10	G	N
11	K	G
12	K	F
13	L	I
14	L	E
15	N	F
16	M	O
17	C	H
18	B	P
19	M	I

(b)

CIRCUITS (LOOPS)	EDGE NUMBERS	LOOP TYPE
L1	5,19,16,-2,-17,-8,-9,-6,-7,1	P
L2	-4, 14, 6, 9, -18	C
L3	4, 3, 2, -16, -19, -13	C
L4	13, -5, -1, 7, -14	C
L5	-3, 18, 8, 17	C
L6	15, -12, 11, 10	D
L7	-15, -10, -11, 12	

P = perimeter loop C = connected loop
D = disjoint loop

Note: a negative edge number indicates that the edge is traversed from end to start nodes .

(c)

Fig. 6.1: a) An arbitrary object view and object loops
b) its corresponding topological data
c) number and type of loops in the view

Edges and nodes may be sequentially grouped to form closed circuits, and in orthographic projections these circuits represent closed loops. All sequences of edges and nodes of all the closed circuits within the graph, may be determined using the 'First-Right' rule. The rule consists of selecting the next edge in the sequence amongst three or more adjacent edges. For instance, assuming that edge 5, in Figure 6.1, has been selected as the first edge in the sequence. The next edge will either be edge 13 or edge 19. According to the 'First-Right' rule, edge 19 will be selected as the next edge in the sequence since its anticlockwise angle from edge 5, at node I, is smaller than that of edge 13; thus edge 19 is the nearest, from the right, to the edge 5.

The perimeter loop may also be determined by starting the sequence from a node that is known to be at an extreme position, and adopting the 'First Right' choice consistently at each node. For example, node I is at an extreme position (on the boundary or perimeter loop) since it has the minimum Y coordinate for the entire graph. There are three adjacent edges at node I: edges 13, 5, and 19. Edge 19 is chosen as the next in the sequence since it has the smallest angle to the positive X-axis. At node M the only possible edge to follow is edge 16, but at node H, edge 17 is selected since its anticlockwise angle from edge 16 is smaller than that of edge 3. The process is continued until the path has returned to the starting node

I. The other loops, referred to as 'inside' loops, as detailed in Figure 6.1, may be discovered by traversing each edge twice, the second time in the opposite sense from the first. One result of this is that loop number L6 and L7, through nodes F, K, G and N, are identical in every way except sense, and it is axiomatic that any such loop is disjoint. Any other loop is referred to as a 'connected' loop.

The algorithm is as follows:

STEP 1: Label all edges in the graph as 0

STEP 2: Find a node that has the minimum Y coordinate

STEP 3: Find an edge that has an end point at the above node, and makes the smallest angle with the positive X-axis

STEP 4: Mark this edge as +1 if the above node is the start node, or as -1 if it is the end node

STEP 5: Starting at the other node of the above edge , make a right turn to select the next edge

STEP 6: Mark each traversed edge as +1 if it has a flag 0 and traversed in the positive sense, or as -1 if it has a flag 0 and traversed in the negative sense

STEP 7: Once the start node (node chosen in step 2), is reached again, search for an edge that has been marked as 0, +1 or -1.

STEP 8: Reverse the sense of the edge found in step 7 and make first right turn

STEP 9: Mark each edge as +2 if traversed in both opposite senses

STEP 10: Repeat steps (5), (6), (7), (8) and (9 until all the edges have been marked as +2, i.e. traversed in both positive and negative senses

STEP 11: Label the first circuit (loop) as PERIMETER, and the remaining ones as INSIDE

STEP 12: Compare each pair of circuits, and if a circuit repeats itself in the opposite sense then label it as DISJOINT (Type =0), else label it as CONNECTED (Type =1).

The results of applying the above algorithm to the arbitrary view in Figure 6.1(a) are displayed in Figure 6.1(c). The first loop, circuit L1, is identified as the perimeter loop while the remaining circuits as inside loops. Circuits L6 and L7 represent the same loop traversed in both opposite senses and is therefore identified as a disjoint loop. The others are recognized as being connected loops.

6.1.2) <u>THE 'LOOP IDENTIFIER' ALGORITHM</u>:

This algorithm exploits the geometric and topological characteristics of simple two-dimensional shapes in order to identify the pattern of each closed loop in the orthographic views. The actual technique used is similar to the production rules approach employed in the design of expert systems, in the sense that the knowledge about the characteristics of each geometric shape is used. For example, a loop is classified as a rectangle if it has four straight edges and three inside angles each equal to ninety degrees. The characteristics associated with a number of geometric shapes can also be recognized. Figure 6.2, shows the five simple 2D shapes that may readily be identified by the algorithm. These may be defined as follows:

a) a rectangle, or a square, as a loop that has :

- four straight edges,
- all the inside angles are ninety degrees angles,
- opposite edges are parallel and equal.

b) a right-angled triangle as a closed loop that has :

- three straight edges,
- at least one of the inside angles is ninety degrees.

c) a circle as a closed loop that has :

- one circular arc edge
- the start and end nodes of the arc edge are the same point

or

- two or more circular arc edges,
- all the arcs have the same radius and centre coordinates,
- the start node of the first edge is the same point

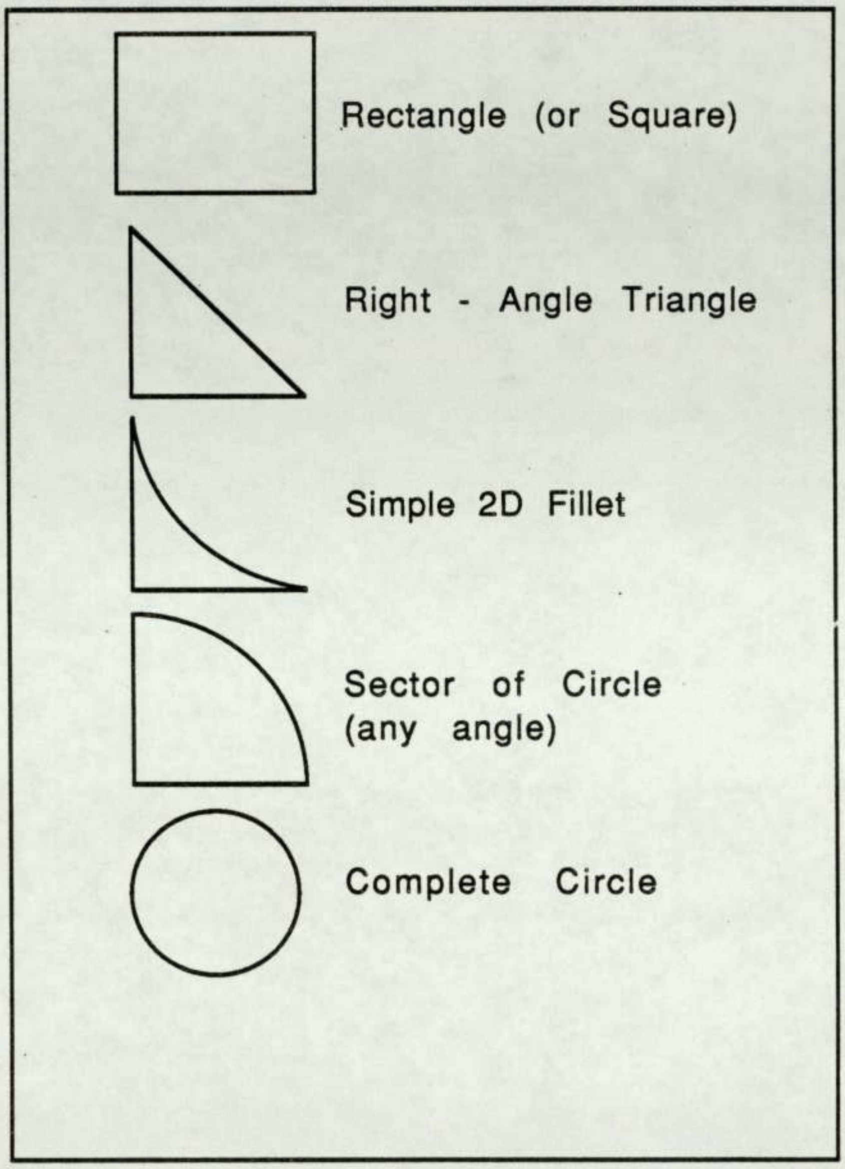


Fig. 6.2: The five 2D geometric shapes identified as 'Basic Patterns'

as the end node of the last edge.

d) a quadrant as a closed loop that has :

- three edges two of which are straight and the other is an arc,

- the inside angle formed by the straight edges is equal to ninety degrees,

- the coordinates of the centre of the arc are equal to the coordinates of the intersection point of the straight edges.

e) a fillet as a closed loop that has :

- three edges two of which are straight and the other is an arc,

- the inside angle formed by the straight edges is equal to ninety degrees,

- the coordinates of the centre of the arc are equal to the coordinates of the mirror image of the intersection point of the straight edges about the chord joining the nodes of the arc.

Each shape has several characteristics, however the algorithm only uses those which are necessary and sufficient to identify it. For example, in order to identify a rectangle, the algorithm only looks for the following two characteristics:

- four straight edges,

- at least three inside angles must be equal to ninety degrees.

A loop that has been identified as one of the above shapes, is stored as a basic pattern, and is allocated a flag, LPF, according to the shape it has been identified with:

- LPF = 1 : for a rectangle or a square loop,
- LPF = 2 : for a right-angled triangle loop,
- LPF = 3 : for a fillet loop,
- LPF = 4 : for a quadrant loop,
- LPF = 5 : for a full circle loop.

It is obviously possible to include a much larger number of simple shapes in the algorithm, such as parallelograms, semicircles and so on ..; however it has been found that the shapes described above are sufficient for the interpretation process, since they are the only ones found in the orthographic views of the three-dimensional primitives considered in this work. Furthermore, each loop which can not be identified as one of the above shapes, is stored as an arbitrary pattern with a flag set to zero (LPF = 0).

Assuming that the loop that is being processed has N edges, then the algorithm for each loop is as follows:

```
STEP 1: Are all the edges straight edges ?
   1.1: if YES then go to step(2)
   1.2: if NO then go to step(5)
STEP 2: if N < 3 then exit with error message.
   2.1: if N = 3 then go to step (3)
   2.2: if N = 4 then go to step (4)
   2.3: if N > 4 then go to step (14)
STEP 3: Is the inside angle between two adjacent edges equal to ninety degrees ?
   3.1: if YES then the loop is a right-angled triangle, LPF = 2. EXIT.
   3.2: if NOT then go to step (14)
STEP 4: Are there at least three inside angles equal to ninety degrees ?
   4.1: if YES then the loop is a rectangle or square, LPF = 1. EXIT.
   4.2: if NOT then go to step (14)
STEP 5: Are all the edges circular arcs ?
   5.1: if YES then go to (6)
   5.2: if NOT then go to (9)
STEP 6: All edges are circular arcs;
   6.1: if N = 1 go to step (7)
   6.2: if N > 1 go to step (8)
STEP 7: Are the coordinates of the start node equal to the coordinates of the end node ?
   7.1: if YES then the loop is a full circle, LPF = 5. EXIT.
   7.2: if NOT then go to step (15).
```

```
STEP 8: Are the coordinates of the centres of all the arcs equal ?
   8.1: if YES then the loop is a full circle, LPF = 5. EXIT.
   8.2: if NOT then go to step (14)
STEP 9: Some of the edges are straight edges and others are circular arcs.
   9.1: if N = 3 then go to (10)
   9.2: if N < 3 then go to step (15)
   9.3: if N > 3 then go to step (14)
STEP 10: Is there only one arc among the edges ?
   10.1: if YES then go to step (11)
   10.2· if NOT then go to step (14)
STEP 11: Are the two straight edges perpendicular ?
   11.1: if YES then go to step (12)
   11.2: if NOT then go to step (14)
STEP 12: Are the coordinates of the centre of the arc equal those of the point of intersection of the two straight edges ?
   12.1: if YES then the loop is a quadrant, LPF = 4. EXIT.
   12.2: if NOT then go to step (13)
STEP 13: Are the coordinates of the centre of the arc equal to those of a point which is the mirror image of the point of intersection of the two straight edges about the line joining the two noaes of the arc ?
   13:1 if YES then the loop is a fillet, LPF = 3. EXIT.
   13:2 if NOT then go to (14)
STEP 14: The loop is an arbitrary loop, LPF = 0. EXIT.
```

STEP 15: Error message and EXIT.

6.1.3) <u>THE 'CLASS IDENTIFIER' ALGORITHM</u>:

The class of the object depends on the number, type and shape of all the loops in all the views, as shown in sections 2.4.1 and 2.4.2. This algorithm uses the information about the number and type of loops gathered from the 'loop detector' algorithm, and the shape information obtained from the 'loop identifier' algorithm, to determine the class of the object.

By assuming that:

a) the orthographic views are labelled as 1 for the XY view, 2 for the XZ view and 3 for the YZ view, and the number of loops in each view is stored in the array NL(i), i = 1, 2, 3,

b) the type of each loop is assumed to be stored in the array TYPE(n), n = 1, 2, 3, ... NL(i), and equal to either 0 for disjoint loop, or to 1 for a connected loop,

c) the information concerning the shape of the loop is assumed to have been stored in the array SHAPE(n) and is equal to LPF (the flag set by the loop identifier).

The algorithm is as follows:

```
STEP 1:  i = 0; KOUNT = 0;
STEP 2:  Is i > 3 ?
   2.1: if YES then go to step (14)
   2.2: if NOT then i = i + 1 and go to step (3)
STEP 3:  Does view NL(i) consists of one loop only ?
   3.1:  if YES then KOUNT = 0
   3.2:  if NOT then KOUNT = KOUNT + 1
   3.3:  Go to step (2)
STEP 4:  All the loop have been examined,
   4.1:  if KOUNT = 0 then go to step (5)
   4.2:  if KOUNT = 1 then go to step (7)
   4.3:  if KOUNT = 2 then go to step (8)
   4.4:  if KOUNT = 3 then go to step (9)
STEP 5:  Each view consists of one closed loop. Are all the
loops identified as basic patterns, 1 ≤SHAPE(1)≤ 5 ?
   5.1:  if YES then go to step (6)
   5.2:  if NOT then EXIT.
STEP 6:  Do these set of patterns form a 'signature' ?
   6.1:  if YES then the object is itself a primitive.
EXIT.
   6.2:  if NOT then EXIT.
STEP 7:  The object has only one loop in at least one view.
The object may be a simple prismatic or an arbitrary
object. Go to step (11).
STEP 8:  The object has at least two views each of which
comprises only one loop. This is not consistent. EXIT.
```

STEP 9: Each view has more than one loop. The object may either be a complex prismatic or an arbitrary object. Check if the loops are disjoint or not. Is there at least one view which comprises only disjoint loops (TYPE(n) =0)

9.1: if YES then go to step (10)

9.2: if NOT then go to step (14)

STEP 10: Is there more than one view which comprises only disjoint loops ?

10.1: if YES then EXIT.

10.2: if NOT then go to step (11)

STEP 11: Do all the nodes in the views other than the one which consists of either one, or more, closed disjoint loops, belong to the perimeter loop.

11.1: if YES then go to step (12)

11.2: if NOT then EXIT.

STEP 12: Are all the loops in the views other than the one which consists of either one, or more, closed disjoint loops, identified as rectangular shapes ?

12.1: if YES then go to step (13)

12.2: if NOT then EXIT.

STEP 13: If I (I = 1, 2, or 3) is the base view, i.e. the view which comprises either one, or more, disjoint loops, then :

13.1: if NL(I) = 1 then the object is a simple prismatic object. EXIT.

13.2: if NL(I) > 1 then the object is a complex prismatic object. EXIT.

STEP 14: The object is an arbitrary object. EXIT.

6.1.4) <u>THE 'LOOP PROCESSOR' ALGORITHM</u>:

The class of the object, once identified, determines the object loop, (or loops) that is (are) to be processed in order to identify the three-dimensional primitives together with the transformations and Boolean operations associated with them. For instance, in the case of prismatic objects, the object loops that are found in the base view, are the only loops to be processed since the other views comprise only rectangles (section 5.4.2). In the case of non-prismatic objects, only the object loop identified as the boundary, or perimeter, loop in each view, is analysed (section 5.4.3).

The first task in processing an object loop is to define the node coordinates of its surrounding rectangle. This is achieved by computing the extreme node coordinates of the object loop. The next task is to locate the loops formed by the intersection of the surrounding rectangle and the object loop. This is achieved by generating a list, referred to as the 'control list', which contains the information about the position of the nodes of the object loop in relation to the nodes and sides of the surrounding rectangle. Each located loop is then examined in order to classify it as either a specific basic pattern or as an arbitrary pattern. This task is in effect carried out by the 'Loop Identifier' algorithm described above. A flag which identifies the shape of the pattern is then attached to each loop. Any loop that has been identified as an arbitrary pattern is then decomposed into either a number basic patterns or into further arbitrary patterns that require further processing.

Finally, using these basic patterns, all the three-dimensional primitives are identified, and the transformations together with the Boolean operations associated with them, are defined. These are then stored in a Boolean tree according to the order in which the primitives are generated. The data stored in the Boolean tree is then converted into a specific format which depends on the solid modeller that is used in the interpretation process, and which, in the present work, is the PAFEC "BOXER" text definition structure described section 5.3.4. The converted data is in effect the output of the analysis step, and the file in which it is stored, is used as input file by the solid modeller.

The specifications of the 'Loop Processor' algorithm are as follows:

STEP 1: Read and extract the coordinates of the nodes of the loop from the input file, and compute the maximum and minimum values to define the coordinates of the surrounding rectangle.

STEP 2: Generate a 'control list' by performing a series of tests to check the position of all the object loop nodes in relation with the nodes and sides of the surrounding rectangle.

STEP 3: Use the control list generated in step (2) to locate the primitive loops formed by the intersection of the surrounding rectangle and the object loop.

STEP 4: Examine the characteristics of the shape of each primitive loop and identify the pattern associated with it. Store the data of the loop together with the flag, which identifies its shape, in a file.

STEP 5: Scan the above file for a flag which identifies a loop as an arbitrary pattern. If such flag exists, then extract the data of that particular loop from that file and store them in the input file, and repeat steps (1) to (5). Otherwise go to step (6).

STEP 6: Identify the three-dimensional primitive related to each basic pattern. Generate and store the transformations and Boolean operations associated with each primitive.

STEP 7: Convert the data obtained from step (6) into BOXER format and store it into a file.

STEP 8: EXIT.

The tasks associated with the steps of the above algorithm, are in effect carried out by the algorithms described in the following sections.

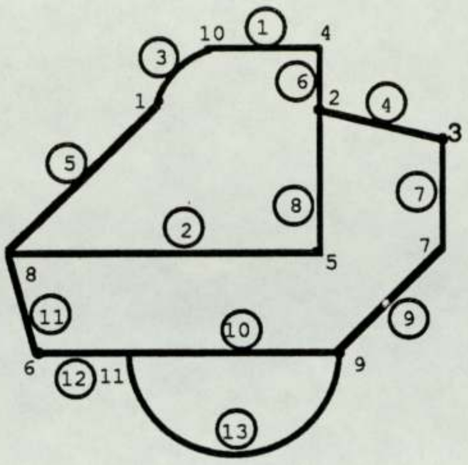
6.1.5) THE 'EXTREME COORDINATES SEARCH' ALGORITHM:

This is a straightforward search routine which performs the first step of the 'Loop Processor' algorithm. Its function is to scan the coordinates of the nodes of a particular loop and finds the maximum and minimum values. It is possible that an extreme value lies on a circular arc where no node exists, then it is necessary to split the arc into smaller arcs and generate a node at this extremity. Figure 6.3(a) shows an orthographic view where such a case may arise; initially, the view comprises 11 nodes; node 12 is then added to it by splitting edge {11,9} into edge {11,12} and {12,9}, in Figure 6.3(b) which also shows the updated topology.

These extreme values are used to define nodes coordinates for the surrounding rectangle which is, as defined previously in section 5.3.1, the orthographic projection of the surrounding cuboid. The specifications of this algorithm may be briefly described as follows:

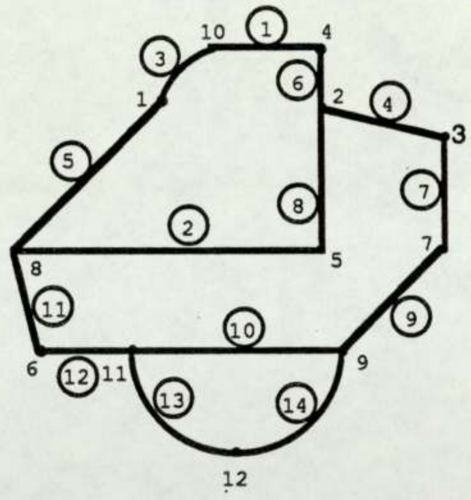
Assuming that there are N edges in the view, then:

```
STEP 1: I = 0, and set Pmax and Qmax to infinitely small
values, and Pmin and Qmin to infinitely large values.
STEP 2: I = I + 1. Is I greater than N ?
   2.1: If YES then go to step (9)
   2.2: If NOT then go to step (3)
```



TOPOLOGY		
Edge no.	Start node	End Node
1	10	4
2	5	8
3	1	1
4	2	3
5	1	8
6	4	2
7	7	3
8	2	5
9	7	9
10	9	11
11	6	8
12	6	11
13	11	9

(a)



NEW TOPOLOGY		
Edge no.	Start node	End Node
1	10	4
2	5	8
3	1	10
4	2	3
5	1	8
6	4	2
7	7	3
8	2	5
9	7	9
10	9	11
11	6	8
12	6	11
13	11	12
14	12	9

(b)

Fig. 6.3: Search for extreme coordinate values and topology update

```
STEP 3: Read the coordinate values of the nodes of the Ith edge in the loop and compute the maximum and minimum coordinate values, Xmax, Ymax, Xmin and Ymin.
   3.1: If Xmax greater than Pmax, then Pmax = Xmax
   3.2: If Xmin smaller than Pmin, then Pmin = Xmin
   3.3: If Ymax greater than Qmax, then Qmax = Ymax
   3.4: If Ymin smaller than Qmin, then Qmin = Ymin
STEP 4: Is the Ith edge a circular arc ?
   4.1: If YES then go to step (5)
   4.2: If NOT then go to step (2)
STEP 5: Does the arc intersect a horizontal straight edge passing through its centre ?
   5.1: If YES then go to step (6)
   5.2: If NOT then go to step (7)
STEP 6: If XA and YA are the coordinates of such intersection point, then
   6.1: If XA greater than Pmax, then Pmax = XA
   6.2: If XA smaller than Pmin, then Pmin = XA
   6.3: If YA greater than Qmax, then Qmax = YA
   6.4: If YA smaller than Qmin, then Qmin = YA
STEP 7: Does the arc intersect a vertical straight edge passing through its centre ?
   7.1: If YES then go to step (8)
   7.2: If NOT then go to step (2)
STEP 8: If XB and YB are the coordinates of such intersection point, then
   8.1: If XB greater than Pmax, then Pmax = XB
   8.2: If XB smaller than Pmin, then Pmin = XB
   8.3:  If YB greater than Qmax, then Qmax = YB
   8.4:  If YB smaller than Qmin, then Qmin = YB
   8.5: go to step (2).
STEP 9: EXIT.
```

The coordinates, XR and YR, of the nodes of the surrounding rectangle may then defined according to the sense of the object loop it surrounds. Figure 6.4(a) shows an anticlockwise loop surrounded by a rectangle defined by the following nodes :

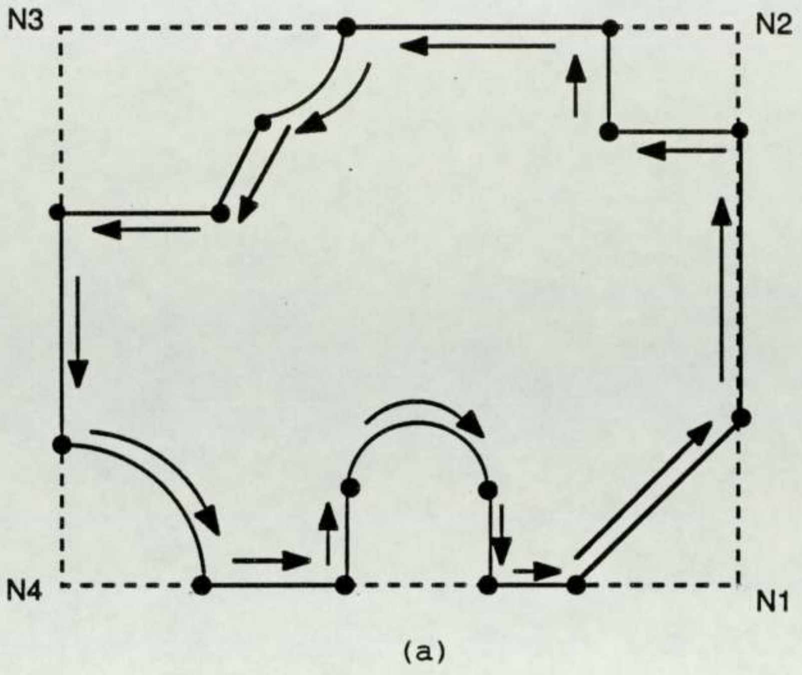
XR(1) = Pmin	YR(1) = Qmin
XR(2) = Pmax	YR(2) = Qmin
XR(3) = Pmax	YR(3) = Qmax
XR(4) = Pmin	YR(4) = Qmax

and Figure 6.4(b) shows a clockwise loop surrounded by a rectangle defined by the following nodes:

XR(1) = Pmin	YR(1) = Qmin
XR(2) = Pmin	YR(2) = Qmax
XR(3) = Pmax	YR(3) = Qmax
XR(4) = Pmax	YR(4) = Qmin

6.1.6) <u>THE 'CONTROL LIST GENERATOR' ALGORITHM</u>:

This algorithm performs step (2) of the 'Loop



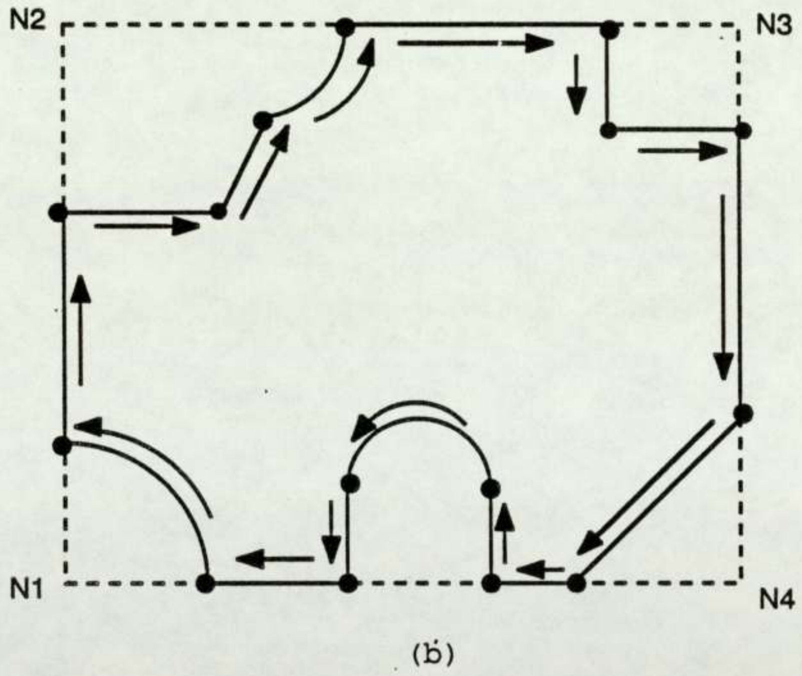
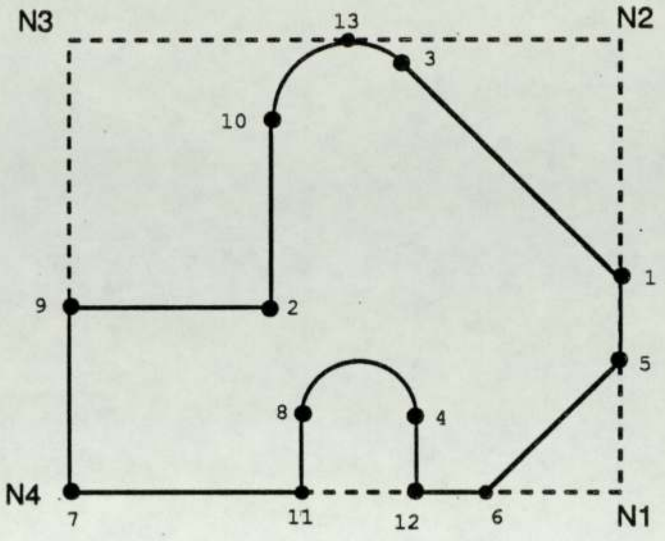


Fig. 6.4: a) An anticlockwise lo[illegible] and its surrounding rectangle
b) A clockwise loop and its surrounding rectangle

Processor' algorithm. Its function is to produce a list that is needed for the detection of primitive loops formed by the intersection of the object loop and the surrounding rectangle. It performs a series of tests, using homogeneous coordinates, (see appendix C), to check the position of all the nodes of the object loop, in relation with sides and nodes of the surrounding rectangle.

An object loop and its surrounding rectangle are shown in Figure 6.5(a). The 'control list' consists of three arrays, (A), (B) and (C), shown in Figure 6.5(b) as columns A, B and C, respectively. Array (A) stores the node number of any node which belongs to the object loop and which lies on a side of the surrounding rectangle. Array (B) stores the number of the side of the surrounding rectangle which contains that node, and array (C) stores the digit 1, to indicate that the node lies on one of the sides of the surrounding rectangle, or the digit 0 to indicate that the node coincides with one of the nodes of the surrounding rectangle. For example, in Figure 6.5(b), the first row of the control list indicates that node 5 of the object loop, shown in Figure 6.5(a), lies on the surrounding rectangle side number 1, and does not coincide with any node of the surrounding rectangle, while the fifth row indicates that node 7 of the object loop, lies on side 4 of the surrounding rectangle and coincides with one of the surrounding rectangles nodes, shown in Figure 6.5(a) as node N4.



(a)

A	B	C
5	1	1
1	1	1
13	2	1
9	3	1
7	4	0
11	4	1
12	4	1
6	4	1

(b)

Fig. 6.5: a) An object loop and its surrounding rectangle
b) the corresponding 'control list'

The specifications of the 'Control List Generator' algorithm are as follows:

Assuming that the object loop consists of N edges, and that the surrounding rectangle nodes and sides are labelled as NR_j and SR_j, respectively, where j = 1, 2, 3, and 4, then :

```
STEP 1:  I = 0 and KOUNT = 0 .
STEP 2:  I = I + 1; J = 0. If I > N, then go to step (6) .
STEP 3:  J = J + 1. If J > 4 then go to step (2).
STEP 4:  Does the Ith node of the object loop lie on side
$SR_j$ of the surrounding rectangle ?
   4.1:  If YES then KOUNT = KOUNT + 1
         and A(KOUNT) = I and B(KOUNT) = j.
   4.2:  If NOT then go to step (5)
STEP 5:  Does the Ith node of the object loop coincide with
a node of the surrounding rectangle ?.
   5.1:  If YES then C(KOUNT) = 1
   5.2:  If NOT then C(KOUNT) = 0
   5.3:  Go to step (3) .
STEP 6: EXIT .
```

6.1.7) <u>THE 'PRIMITIVE LOOP LOCATOR' ALGORITHM</u>:

This algorithm processes the 'control list' generated by the previous algorithm in order to:

1) locate the primitive, or subobject, loops formed by the intersection of the surrounding rectangle and the object loop, thus performing step 3 of the 'Loop Processor' algorithm.

2) compute the number of segments, and the coordinates of each node, of each primitive loop it locates.

This algorithm also detects a number of characteristics which identify a primitive loop, referred hereafter as an 'unstable' loop, as one which will later require more processing in the decomposition stage of arbitrary loops into basic patterns. These loops and their special treatment are discussed in the following section.

The specifications of the algorithm are as follows:

It is considered that M nodes of the object loop, have been found to lie on the surrounding rectangle. The numbers of such nodes are listed in the array (A) of the control list. The nodes of the object loop are labelled sequentially according to whether the loop has an anticlockwise or a clockwise sense.

STEP 1: I = 1

```
STEP 2: I = I + 1. If I is greater than M then go to step
(9).
STEP 3: Compute:
         D1 = B(I+1) - B(I)
         D2 = A(I+1) - A(I).
   3.1: If C(I) = 0 or C(I+1) = 0 then go to step (4)
   3.2: If C(I) = 0 and C(I+1) = 0 then go to step (7)
   3.3: If D2 < 0 then go to step (8)
STEP 4: D1 = D1 - 1
   4.1: If D1 = 0 then go to step (5)
   4.2: If D1 = 1 then go to step (6)
   4.3: If D1 > 1 then go to step (8)
STEP 5: Check if there is any primitive loop between node
numbers stored in (A):
   5.1: If D2 = 0 then the node number in A(I+1) is the
same as the node number stored in A(I), thus there is an
error. Go to step (7).
   5.2: If D2 = 1 then there is only one edge that can be
defined between the node stored in A(I+1) and A(I). If
that edge is an arc, then there is a primitive loop which
has two edges, but if the edge is a straight edge then
there is not any primitive at that position.
   5.3: If D2 = 2 then there is a primitive loop which has
three edges.
   5.4: If D2 = 3 then there is a primitive loop which has
four edges.
   5.5: If D2 = 3 + n then there is a primitive loop which
has 4 + n edges.
```

5.6: Go to step (2).

STEP 6: Check if there is any primitive loop between node numbers stored in (A):

6.1: If D2 = 0 then the node number in A(I+1) is the same as the node number stored in A(I), thus there is an error. Go to step (7).

6.2: If D2 = 1 then there is a primitive loop which has three edges.

6.3: If D2 = 2 then there is a primitive loop which has four edges.

6.4: If D2 = 2 + n then there is a primitive loop which has 4 + n edges

6.5: Go to step (2)

STEP 7: Check if there is any primitive loop between node numbers stored in (A):

7.1: If D2 = 1 then there is a primitive loop which has three edges.

7.2: If D2 > 1 then there is a primitive loop which is identified as an 'unstable' loop.

7.3: Go to step (2).

STEP 8: Error.

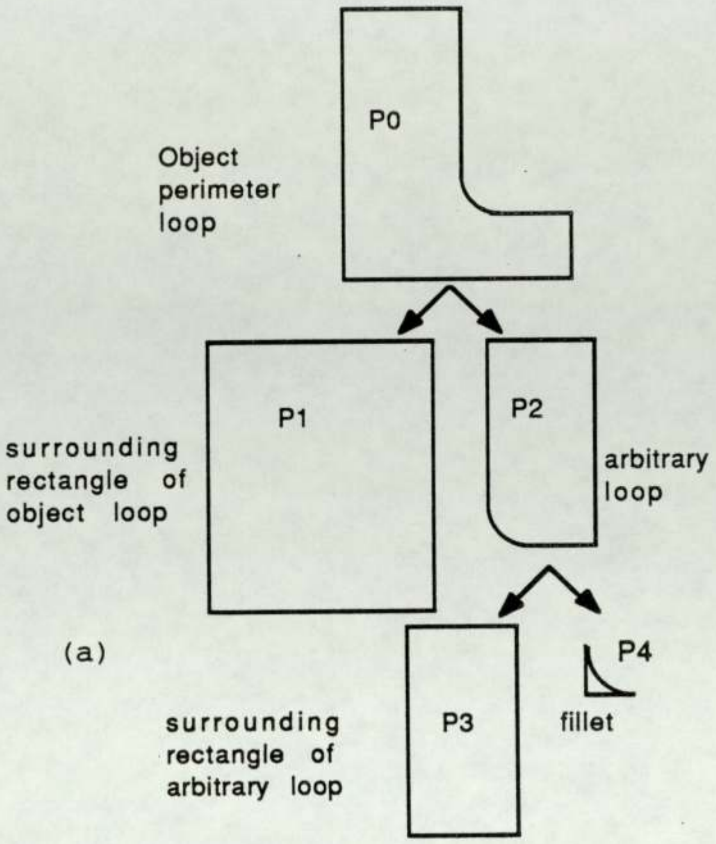
STEP 9: Exit.

The output of the above algorithm comprises the geometric and topological data of each primitive loop that has been located. These data are then used by the 'Loop Identifier' algorithm, in step 4 of the 'Loop Processor' algorithm, to determine the shape of each primitive loop which is then stored in a file, referred to thereafter as file MAINDATA, together with the appropriate flag, LPF, to indicate whether the shape is a basic pattern, LPF = 1, 2, 3, 4, or 5, or an arbitrary pattern, LPF = 0.

6.1.8) THE 'ARBITRARY PATTERN ANALYSER' ALGORITHM:

The function of this algorithm is to decompose any arbitrary loops into further primitive loops. The output file, MAINDATA, generated from the 'Loop Identifier' routine, is scanned in order to search for any flags, LPF, equal to 0, which indicate that the corresponding loop has an arbitrary pattern. If such a flag is found then the geometric and topological data of the corresponding loop is retrieved and stored in another file, referred to as ADATA, in order to be used as input to the whole process again. The processing of the file ADATA may result in a number of primitive loops that are either basic or arbitrary patterns, or both. This decomposition process is illustrated in Figure 6.6(a), where the object loop PO, which has an arbitrary pattern, is decomposed into the following loops:

loop P1: a positive rectangle (LPF = 1)
loop P2: a negative arbitrary loop (LPF = 0)


P0
Object perimeter loop
P1
P2
surrounding rectangle of object loop
arbitrary loop
(a)
P4
P3
fillet
surrounding rectangle of arbitrary loop

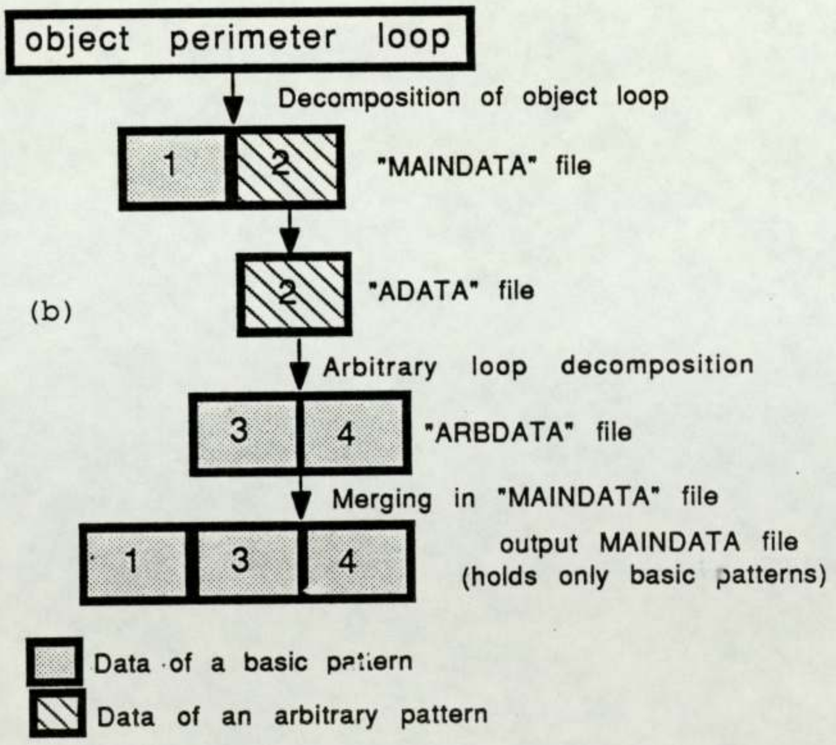

object perimeter loop
Decomposition of object loop
1
2
"MAINDATA" file
(b)
2
"ADATA" file
Arbitrary loop decomposition
3
4
"ARBDATA" file
Merging in "MAINDATA" file
1
3
4
output MAINDATA file (holds only basic patterns)
Data of a basic pattern
Data of an arbitrary pattern

Fig. 6.6: a) Decomposition of an object loop
b) Transfer and merging of data files

The results of processing the file ADATA are then stored in a new file, called ARBDATA, instead of being stored in MAINDATA. For example, in Figure 6.6(a), loop P2 is decomposed further into two loops, P3 and P4, which are identified as basic patterns and whose data is stored in ARBDATA. All the above files are direct access files, designed to have the same structure, a description of which is given in section 7.5.

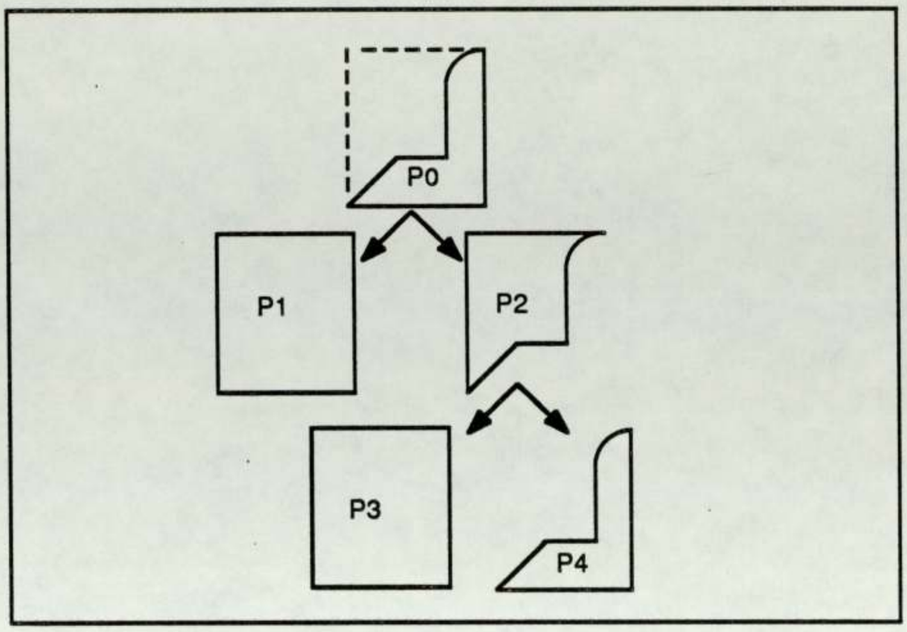
The algorithm consists mainly of a fast merging routine that merges two direct access files together into one. In this case, it merges files MAINDATA and ARBDATA into the original file MAINDATA from which the file ADATA has originated, as shown in Figure 6.6(b). The process of scanning the file MAINDATA, generating and processing ADATA and merging ARBDATA to MAINDATA is repeated again until all the flags (LPF) in MAINDATA are found to be not equal to 0, which would indicate that the data stored in the file, correspond to loops which have been identified as basic patterns only.

There are, however, some particular loops, referred to earlier, in section 6.1.7, as 'unstable' loops, which can not be directly decomposed into further patterns. The direct application of the decomposition process to such a loop always leads to the generation of a 'child' loop that is identical to its ancester. These type of loops are readily detected by the 'primitive loop locator' algorithm described above. In this case, each 'unstable' loop is dealt with by simply dividing it into three loops for which the data are then stored back in the file MAINDATA. Figure 6.7 shows such a loop, P0, and those generated from its decomposition, one of which, P4, is identical to it.

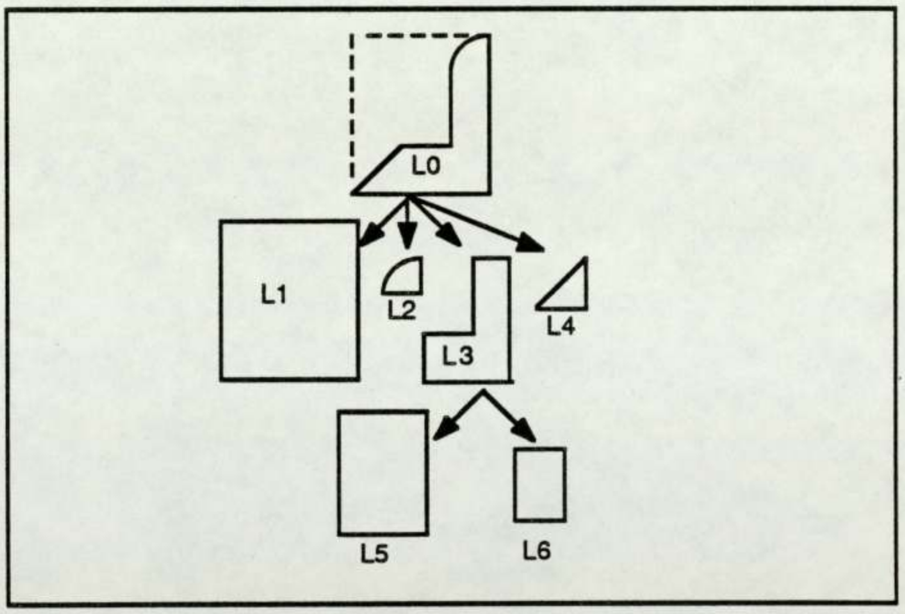
The specifications for such an algorithm are as follows:

It is considered that the input file has been already processed to the stage where the file MAINDATA has been generated, and that it comprises the data of a number, NL, of loops, some of which have basic patterns and others arbitrary ones. Thus, the file may comprise flags that are equal to either 0, 1, 2, 3, 4, or 5.

```
STEP 1: I = 0.
STEP 2: I = I + 1. Is I greater than NL ?
   2.1:  If YES then go to step (11)
   2.2:  If NOT then go to step (3)
STEP 3: Scan the file MAINDATA and read the flag (LPF) of
the Ith loop.
   3.1:  If LPF = 0 then go to step (4)
   3.2:  If LPF = 1, 2, 3, 4 or 5 then go to step (2)
STEP 4:  Check if the loop is 'unstable'.
   4.1:  If YES then go to step (5)
   4.2:  If NOT go to step (6)
```



(a)



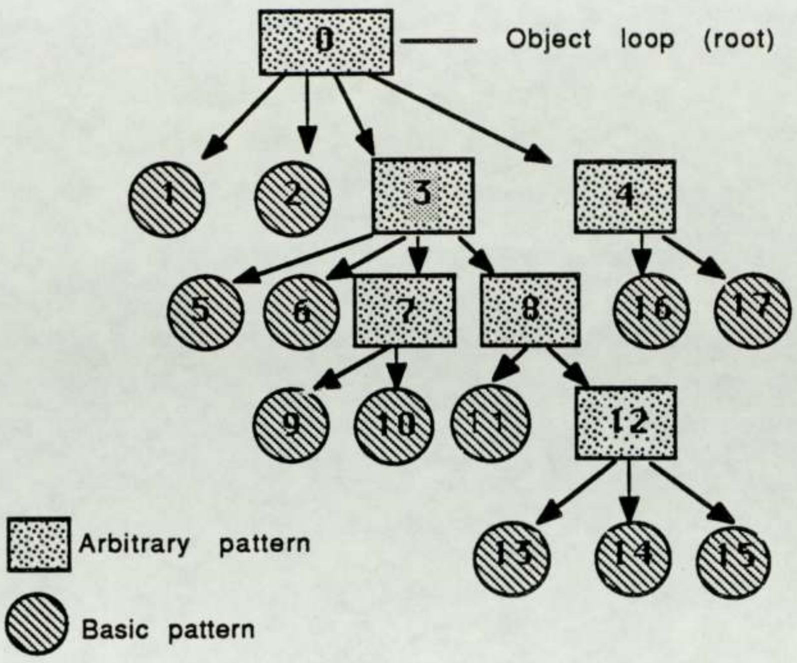
(b)

Fig. 6.7: a) An 'unstable' loop which can not be decomposed directly
b) The decomposition of an 'unstable' loop

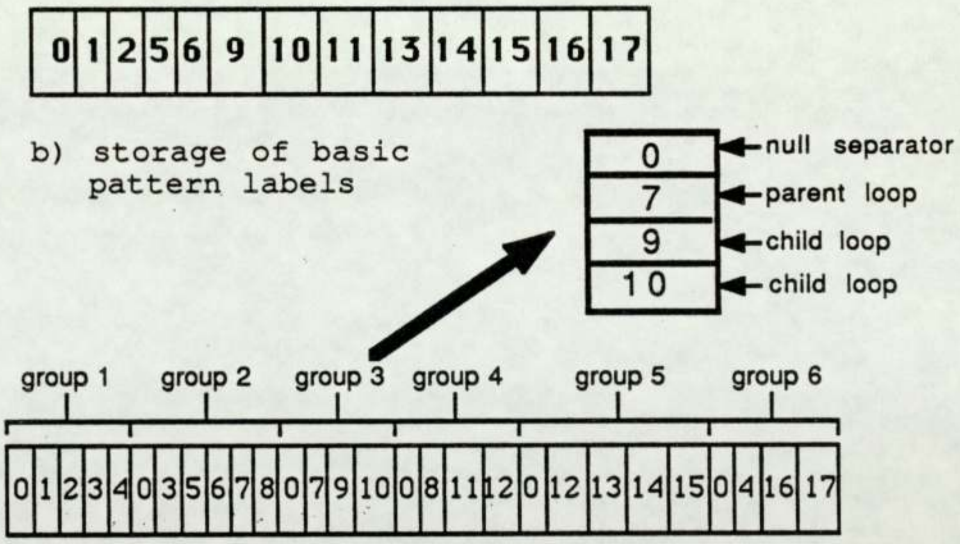
STEP 5: Split the 'unstable' loop into three loops and store their corresponding data in file 'MAINDATA' with a flag LPF = 0. Go to step (1).
STEP 6: Store the data of the Ith loop (arbitrary pattern) in the file 'ADATA'.
STEP 7: Use the file 'ADATA' as the input of the analysis process, i.e. locate and identify the primitive loops formed by the intersection of the arbitrary loop and its surrounding rectangle.
STEP 8: Store all the loops generated from step (5) in the file 'ARBDATA' with their corresponding flags (the file ARBDATA has the same structure as the file MAINDATA, and may contain both loops which are basic patterns and loops which are arbitrary patterns).
STEP 9: Merge the file 'ARBDATA' into the file 'MAINDATA'.
STEP 10: Repeat steps (1), (2), (3), (4), (5), (6), (7), (8), and (9), until all the loops stored in file 'MAINDATA' are identified as basic patterns, i.e., all the flags, LPF, are not equal to zero.
STEP 11: EXIT.

The output of the above algorithm is a file which contains the geometric and topological data of all the two-dimensional basic patterns generated from the analysis of one object loop in a given view. However, it has been found that such data is not enough for the reconstruction of the corresponding object loop. This reconstruction process also requires the storage of the information, in the form of pointers, corresponding to the relationship between 'parent' and 'children' loops, and the order in which these are generated. This information is stored in the form of a tree, as shown in Figure 6.8(a). The object loop is always at the root of the tree where each node represents an arbitrary primitive loop, and each leaf, or terminal node, represents a primitive loop which has been identified as a basic pattern, i.e. one which does not require any further decomposition. This tree is, in effect, stored in two one-dimensional arrays. The first array stores the number of the basic pattern in the order in which they have been generated, as shown in Figure 6.8(b). The second array, Figure 6.8(c), holds a series of pointers, which are separated in a number of groups, by a null parameter. Each group determines a parent loop and its corresponding children loops. For example, the arbitrary loop numbered as 7, in the general tree, has been decomposed further into two primitive loops, 9 and 10, identified both as basic patterns.

The order in which the primitive loops are generated, stored and retrieved is very important since it determines the resulting object loop; this is because the Boolean difference operator is not commutative, and the order in which the primitives are subtracted from each other may yield different results. This problem is illustrated in Figure 6.9(a) which shows the decomposition of an object loop, P0, into a number of primitive loops;



(a) General tree generated by decomposition of an object loop



b) storage of basic pattern labels

c) pointers/groups storage

Fig. 6.8: Arbitrary loop decomposition and data storage

these are labelled according to the order in which they have been generated, and only loops identified as basic patterns are stored, i.e., primitive loops P1, P3, P4, P5 and P6. In order to reconstruct the object loop P0, the Boolean operations performed on these basic patterns have to be carried out in the right order, as shown in Figure 6.9(b), as follows:

P2 = P4 + P5 + P6

and P0 = P1 + P2 + P3

However, a completely different object loop would have been obtained if the Boolean operation have been performed in a different order, as shown in Figure 6.9(c). The reconstruction process, in this case, may produce not only the incorrect object loop, but also self intersecting and impossible loops.

The file which stores the geometric and topological data of all these basic patterns is, in effect, used to define the associated 3D primitives, their corresponding transformations and Boolean operations. Whereas, the two arrays determine which primitives are to be combined by these Boolean operations to generate, for each object loop, a prismatic object, also referred to here as a profile.

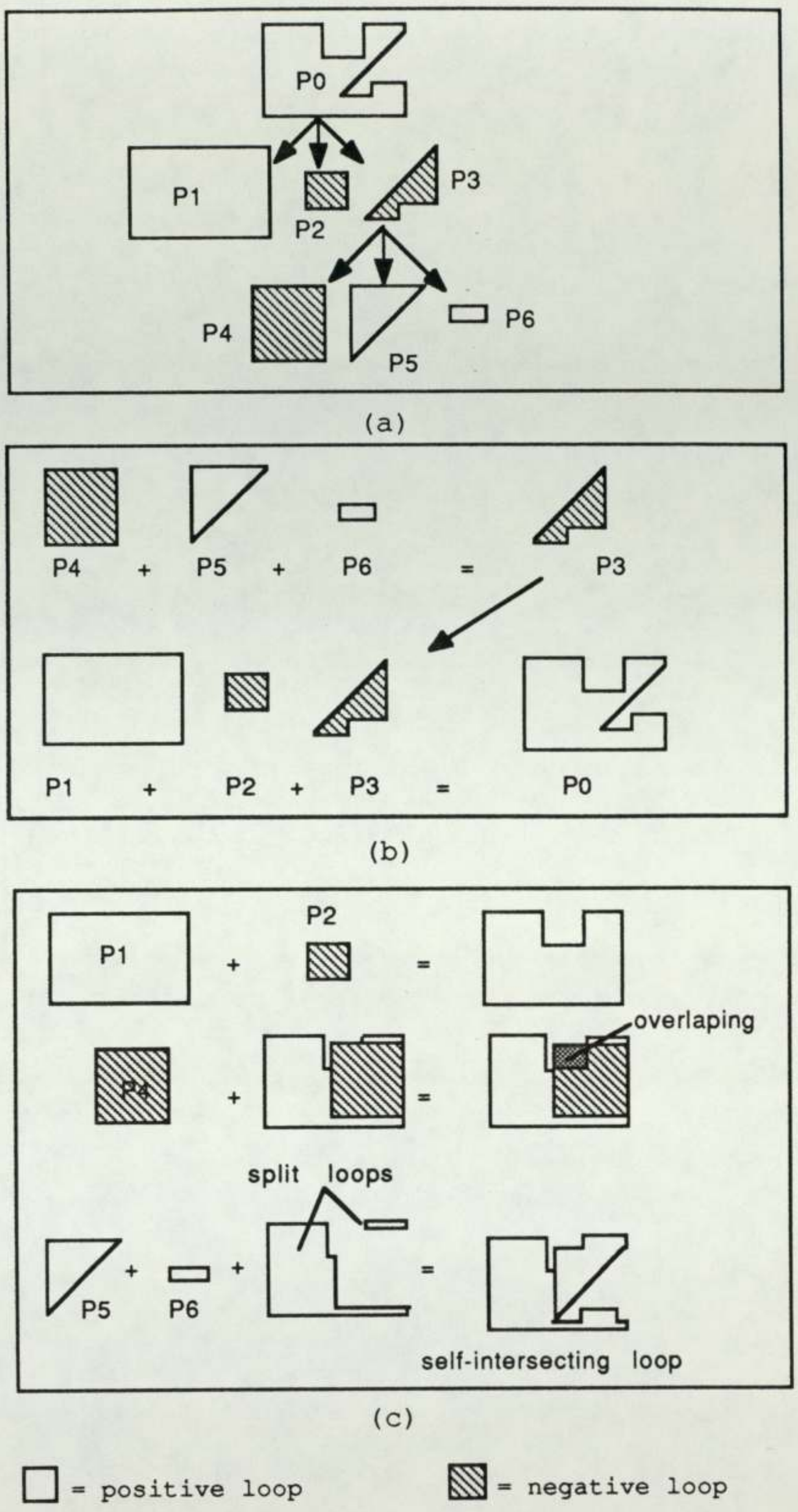


Fig. 6.9: a) Decomposition of an object loop,
b) its correct reconstruction,
c) its incorrect reconstruction

6.1.9) THE '3D PRIMITIVES IDENTIFIER' ALGORITHM:

This algorithm performs step 6 of the 'Loop Processor' algorithm. Its function is to identify the three-dimensional primitives by associating with the three views the primitive loops identified by the previous algorithm. Since the strategy is to always generate a prismatic object, or profile, for each object loop, and since a prismatic object has always two views which comprise only rectangular loops, two of the basic patterns which constitute a primitive signature are, therefore, always rectangles. Thus a complete signature may be obtained by associating two rectangles with each identified and stored basic pattern. For example, if two rectangles are associated with a primitive loop which has been identified as a circle, the prismatic shape is known to be a cylinder, whereas if two rectangles are associated with a primitive loop which has been identified as a right-angled triangle, then the three-dimensional primitive is a wedge.

The input to this algorithm is the file 'MAINDATA' generated by the 'Arbitrary Pattern Analyser' algorithm, according to which, only the data associated with basic patterns is stored. The flag, LPF (see pages 184 and 186), which identifies the geometric shape of each pattern is also stored in this file.

Assuming that N basic patterns have been generated during the process of a given object loop, the specifications of the algorithm are as follows:

```
STEP 1: I = 0
STEP 2: I = I + 1
STEP 3: Is I greater then N ?
   3.1: If YES then go to step (11)
   3.2: If NOT then go to step (4)
STEP 4: Read the value of the flag LPF.
   4.1: If LPF ≤ 0 or LPF > 5 the  go to (10)
   4.2: If LPF = 1 then go to step (5)
   4.2: If LPF = 2 then go to step (6)
   4.3: If LPF = 3 then go to step (7)
   4.4: If LPF = 4 then go to step (8)
   4.5: If LPF = 5 then go to step (9)
STEP 5: The basic pattern is a rectangle and the signature
of a three-dimensional primitive BLOCK is obtained. Go to
step (2).
STEP 6: The basic pattern is a right-angle triangle and
the signature of a three-dimensional primitive WEDGE is
obtained. Go to step (2).
STEP 7: The basic pattern is a fillet and the signature of
a three-dimensional primitive FILLET is obtained. Go to
step (2).
STEP 8: The basic pattern is a quadrant and the signature
of a three-dimensional primitive CYLINDRICAL SEGMENT is
obtained. Go to step (2).
```

STEP 9: The basic pattern is a circle and the signature of a three-dimensional primitive CYLINDER is obtained. Go to step (2).

STEP 10: Error. Exit

STEP 11: The type of all the three-dimensional primitive has now been defined. Exit.

6.2) SOLID MODELLING INPUT FILE GENERATION:

It has been shown in chapter 5, that, in order to generate the solid modelling input file it is necessary to:

1) extract the size, shape, position and orientation parameters of each identified three-dimensional primitive, from the MAINDATA file obtained from the analysis step. The type of each primitive is defined by the '3D Primitives Identifier', described above.

2) generate and store the text structure definition of each primitive, together with the Boolean operations which represent the output model.

The MAINDATA file comprises the data associated with all the two-dimensional basic patterns, and hence, with all the three-dimensional primitives, generated from the processing of one object loop. The file is a direct access one which is composed of a number of sections equal to the number of primitives. Thus, the first section of the file contains the data of the first primitive, the second section to the next primitive, and so on. Each section of the file has the same structure as the file used to store the input data of each view, except for an additional record which is appended at the end of each section to store the identification flag, LPF, of the corresponding primitive. The structure of the files that store the topological and geometrical data of the input views is describer later in section 7.4.

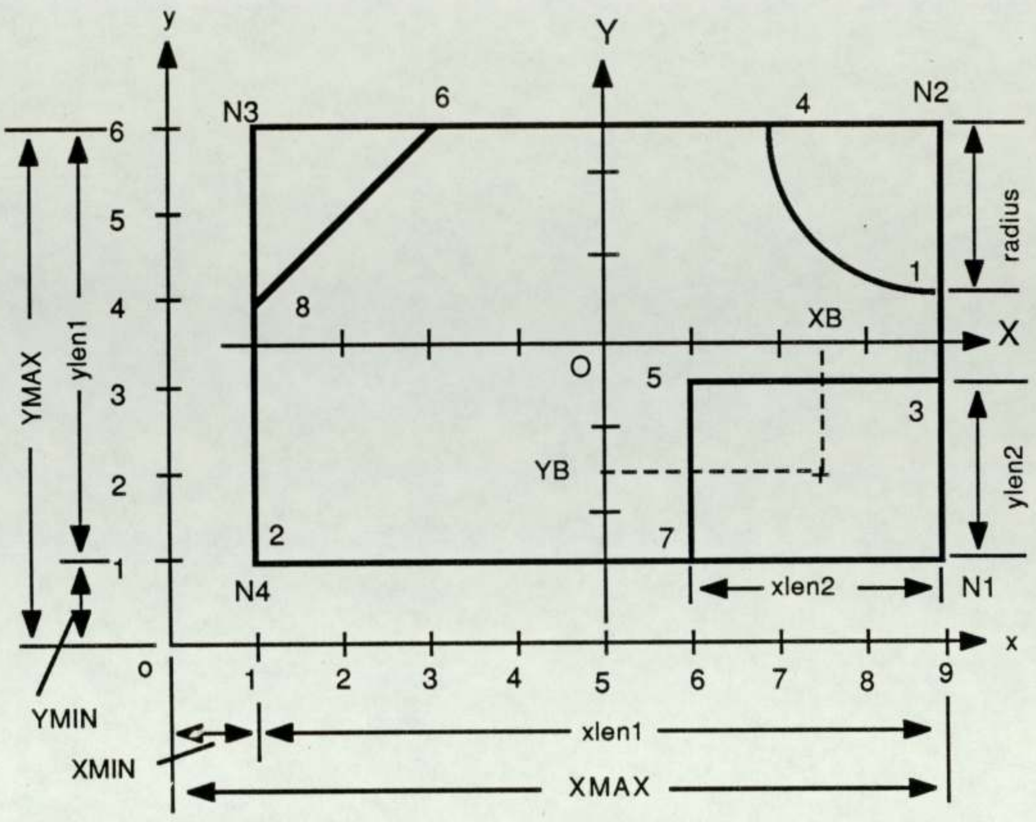
The maximum and minimum X and Y coordinate values for any one of the adjacent views, are also required for the specification of some of the parameters, such as size and position, of some primitives. For instance, if a primitive block has been identified by processing an object loop in the XY view, then these values are used to specify the length of a primitive block in the Z direction. These extreme values are computed in the analysis step.

The data stored in the file MAINDATA, described above, together with the extreme coordinates values of one of the adjacent views, provide enough information to specify all the necessary parameters of the identified primitives. The shape, size, and position parameters of each primitive, are computed using the node coordinate values of the corresponding loop and the maximum and minimum coordinate values obtained from the adjacent view. Whereas, the orientation parameter depends on the view from which the pattern is extracted. A primitive is rotated by an angle equal to 90 degrees about either the X, or Y axis, only if the corresponding pattern is contained in the XZ, or YZ view, respectively. The solid modelling input file is finally completed by specifying the Boolean operations, required to combine the primitives according to the Boolean tree generated by the 'Arbitrary Pattern Analyser' algorithm (section 6.1.8).

A simple prismatic object whose XY view has been identified as a base-view, is illustrated in Figure 6.10(a), where oxyz defines the coordinate system used by the author, and OXYZ represents the solid modeller coordinate system. Figure 6.10(b) shows the solid modeller input file which contains, in 'BOXER' text structure, all the primitive definitions and Boolean operations necessary for the reconstruction of the object.

The first primitive that has been identified is a primitive block, represents, in effect, the surrounding cuboid, or raw block, from which the object is to be 'cut-out'. The corresponding "BOXER' syntax may be written as follows:

```
OBJNAME <- BLOCK(xlen, ylen, zlen) AT (xcen, ycen, zcen)
```



(a)

```
XYO1  <-  BLOCK  (8.0,  5.0,  3.0)

XYO2  <-  CYL  (3.0,  2.0)  AT  (2.5,  4.0,  -1.5)

XYO3  <-  BLOCK  (3.0,  2.5,  3.0)  AT  (2.5,  -1.5,  0.0)

XY04  <-  WEDGE  (2.0,  2.0,  3.0)  AT  (-4.0,  2.5,  -1.5)

FAMOD  <-  XYO1  -  XY02  -  XYO3  -  XY04
```

(b)

Fig. 6.10: a) Computation of 3D primitive parameters from geometry of 2D patterns
b) corresponding solid modelling input file

OBJNAME represents the name of the primitive, or object. The size and shape of a primitive block are defined by its length, xlen, width, ylen, and depth, zlen, which, in this case, may be specified by using the minimum and maximum coordinate values of the base view and one of the adjacent views, in the following equations:

xlen = ABS (XMIN - XMAX) (1)

ylen = ABS (YMIN - YMAX) (2)

zlen = ABS (ZMIN - ZMAX) (3)

where:

XMIN = minimum X coordinate value in the XY view
XMAX = maximum X coordinate value in the XY view
YMIN = minimum Y coordinate value in the XY view
YMAX = maximum Y coordinate value in the XY view
ZMIN = minimum Z coordinate value in the XZ or YZ view
ZMAX = maximum Z coordinate value in the XZ or YZ view

In this example, the name of the primitive block, OBJNAME is automatically set to XY01, and its length, xlen1, is equal to 8.00, height, ylen1, is equal to 5.00 and width, zlen1, to 3.00.

The position parameters of the 3D primitive are defined by specifying the coordinates of its centroid, xcen, ycen and zcen, with reference to the solid modeller coordinate system, OXYZ. Since positioning parameters have not been specified, in this case, the centroid of the primitive block is automatically positioned at the origin of the solid modeller coordinate system. Thus, xcen, ycen and zcen are all equal to 0 in OXYZ, but in oxyz they are as follows:

```
xo = (xmin + xmax) / 2.0
yo = (ymin + ymax) / 2.0
zo = (zmin + zmax) / 2.0
```

where xmin, xmax, ymin, ymax, zmin and zmax are the coordinate values of the object surrounding block or 'Raw Block'.

The next primitive, XY02, is a cylinder segment whose length, cylen, is equal to 3.00 and radius equal to 2.00. The 'BOXER' syntax for a primitive cylinder may be written as follows:

```
OBJNAME <- CYL(cylen, radius) AT ( xcyl, ycyl, zcyl)
```

The cylinder is positioned by defining the coordinates xcyl, ycyl and zcyl, of the centre of its base. In this example, xcyl is equal to 2.5, ycyl to 4.0 and zcyl to -1.5. The cylinder length, radius, position and orientation parameters are computed with reference to the solid modeller coordinate system, OXYZ, which origin is at the centroid of the previous primitive block. The following equations are used:

cylen = ABS(zmin - zmax)

radius = $\sqrt{\{(xc - xn)^2 + (yc - yn)^2)\}}$

xcyl = xc + xo

ycyl = yc + yo

zcyl = - cylen / 2.0

where:

xn = x coordinate, in oxyz, of the start node (node 1) of the circular arc in the base view.

xc = x coordinate, in oxyz, of the centre of the circular arc in the base view.

yn = y coordinate, in oxyz, of the start node (node 1) of the circular arc in the base view.

yc = y coordinate, in oxyz, of the centre of the circular arc in the base view.

xo = x coordinate, in oxyz, of origin of OXYZ coordinate system, as before.

yo = y coordinate, in oxyz, of origin of OXYZ coordinate system, as before.

The third primitive, XY03 is a block whose length, xlen2 equal to 3.0, height, ylen2, equal to 2.5 and width, zlen2, equal to 3.0, are computed using equations (1), (2), and (3) respectively. The primitive is positioned in the solid modeller coordinate system by defining the coordinate values of its centroid which are computed using the following equations:

```
xcen = xo + ((xrmin + xrmax) / 2.0)
ycen = yo + ((yrmin + yrmax) / 2.0)
zcen = - zlen / 2.0
```

where xrmin, xrmax, yrmin, yrmax are the minimum and maximum coordinate values of the primitive loop (rectangle). The coordinate values xo and yo are as previously defined.

The next primitive is a wedge whose name is set to XY04. The 'BOXER' syntax for a wedge may be written as follows:

```
OBJNAME <- WEDGE(xlen, ylen, zlen) AT (xcor, ycor, zcor )
```

The length, height and width of the primitive wedge are computed using the following equations:

```
xlen = ABS(xwmin - xwmax)
ylen = ABS(ywmin - ywmax)
zlen = ABS(zmin - zmax)
```

where xwmin, xwmax, ywmin, ywmax are the minimum and maximum coordinate values of the primitive loop (triangle). The maximum and minimum values, zmin and zmax, are the extreme coordinate values in the z direction, of the surrounding rectangle of the object. The primitive wedge is positioned by specifying the coordinate values xcor, ycor and zcor, of its far corner, which is a node that joins the two perpendicular edges. Theses values are computed using the following equations:

$$xcor = xo + xr$$
$$ycor = yo + yr$$
$$zcor = - zlen / 2.0$$

where:
xr = the x coordinate value of the node at which the 90 degrees angle of the right-angle triangle is sustended.
yr = the y coordinate value of the node at which the 90 degrees angle of the right-angle triangle is sustended.
xo, yo = as previously defined.

Finally, the primitives are combined by the Boolean operation specified, in this case, by the last statement shown in figure 6.10(b). The object, whose name is set to FAMOD, may be reconstructed by subtracting the primitives XY02, XY03 and XY04 from the surrounding block defined as XY01.

6.3) <u>OUTPUT VERIFICATION ALGORITHMS</u>:

There are two sets of data inputs to the output verification process (section 5.3.5): one corresponds to the input orthographic views, and the other, having an ASCII format, is a PAFEC 'BOXER' file which represents the orthographic views of the output model. The principal functions of the output verification algorithms are:

a) to extract the data associated with the orthographic views of the output solid model from the parametric file generated by the solid modeller,
b) to compare the input and output views data in order to detect differences (if any) between the input and output orthographic views.

In the case of prismatic objects, the output verification is carried out to confirm that the output orthographic views are the same as the input orthographic views. Thus, the model generated is verified to be the exact object and the interpretation process is successfully terminated.

In the case of more general three-dimensional objects, the comparison between input and output views may have the same result as above, or may lead to the detection of a number of discrepancies between the input and output views. The presence of such discrepancies indicates that the output model is not the exact object but an approximation model. Thus, there exist a number of subobjects which need to be removed from the output model to generate another output model, which may again be the exact object, or yet another approximation model.

6.3.1) **EXTRACTION OF OUTPUT VIEWS DATA**:

The first task in the output verification subprocess is the extraction of the data corresponding to the orthographic views of the output model from the parametric text file generated by the solid modeller. A small section of such a file, shown in Figure 6.11, contains:

- Three lines of 'REM' statements, where each line is used to indicate that the following block of data corresponds to one view.
- A number of lines that comprise the character string 'LT' followed by an integer whose value is set to 1 to indicate that the following edges are drawn in a dotted line style (hidden edges), or to 2 to indicate that the edges are solid lines (visible edges).
- Several lines that comprise the character string 'LN' followed by an integer whose value is set to either 2 to indicate that the edge is a straight edge, or to 5 to indicate that the edge is a circular arc.
- Several groups of lines comprising the character strings 'X =' and 'Y =' followed by real numbers. In the case where the edge is a straight edge, each pair of lines

```
START / 3.1
 FA 12
PROMPT INDICATE BOTTOM LEFT POINT
ST / C XBL, YBL
REM NEW VIEW X AND Y PAPER SIZE :                  9.9422            9.942
LT   1
LN   2
X=          3.000000 + XBL,  Y=          -1.500000 + YBL
X=         -6.000000,  Y=         0.000000
X=         -3.000000 + XBL,  Y=           1.500000 + YBL
X=          6.000000,  Y=         0.000000
X=          3.000000 + XBL,  Y=           1.500000 + YBL
X=          0.000000,  Y=        -3.000000
X=         -3.000000 + XBL,  Y=          -1.500000 + YBL
X=         0.000000,  Y=          3.000000
X=         -1.000000 + XBL,  Y=          -1.500000 + YBL
X=          0.000000,  Y=         3.000000
X=          0.000000 + XBL,  Y=           1.500000 + YBL
X=          0.000000,  Y=        -3.000000
X=          2.000000 + XBL,  Y=          -1.500000 + YBL
X=          0.000000,  Y=         3.000000
X=          2.000000 + XBL,  Y=          -1.500000 + YBL
X=          0.000000,  Y=         3.000000
LT   2
LN   2
X=         -1.000000 + XBL,  Y=          -1.500000 + YBL
X=          0.000000,  Y=         3.000000
REM NEW VIEW X AND Y PAPER SIZE :                  9.9422            9.942
L1   1
LN   2
X=         -3.000000 + XBL,  Y=           3.000000 + YBL
X=         2.000000,  Y=          0.000000
X=          2.000000 + XBL,  Y=          -3.000000 + YBL
X=        -5.000000,  Y=          0.000000
X=          3.000000 + XBL,  Y=          -1.000000 + YBL
X=         0.000000,  Y=         -1.000000
X=         -3.000000 + XBL,  Y=          -3.000000 + YBL
X=         0.000000,  Y=          6.000000
LN   5
X=          3.000000 + XBL,  Y=          -3.000000 + YBL
X=         0.000000,  Y=          1.000000
A=         90.000000
LN   2
X=         -3.000000 + XBL,  Y=          -1.000000 + YBL
X=         -3.000000,  Y=          0.000000
X=         -1.000000 + XBL,  Y=           0.000000 + YBL
X=         0.000000,  Y=          1.000000
X=         -1.000000 + XBL,  Y=           2.000000 + YBL
X=         0.000000,  Y=          1.000000
X=          0.000000 + XBL,  Y=           2.000000 + YBL
X=         0.000000,  Y=         -1.000000
|                    |                    |              |
|                    |                    |              |
                     |                    |
END
```

Fig. 6.11: Solid modeller output parametric file

represents an edge. The first line contains the X coordinate and Y coordinate values of the start node of the edge, whereas the following line contains the X coordinate and Y coordinate values of the end node relative to the start node of the edge. If, however, the edge is an arc, then the first line contains the X coordinate and the Y coordinate values of the start node and the following line contains the X coordinate and Y coordinate values of the centre of the arc relative to the start node of the edge. An additional line containing the character string 'A =' followed by a real number gives the angular position of the end node with respect to the start node.

It is therefore possible to extract from such a file, the complete geometric and topological data of the orthographic projections of the output model. The algorithm which carries out such a task is a simple routine which manipulates strings of characters.

It was found that the extracted data required a minor adjustment because the coordinate system used by the PAFEC 'BOXER' modeller is different to the one used by the author. A shift in the X and Y directions is computed for each view, by comparing the minimum X and Y coordinate values of the extracted data with the minimum X and Y coordinate values of the corresponding input orthographic views. All the X and Y coordinate values in the output views are then adjusted by deducting the corresponding shift. The output of this routine is therefore the data which represents the orthographic views of the output model in the coordinate system adopted by the author. The data structure is similar to the one which holds the input orthographic views data.

6.3.2) <u>THE 'COMPARISON' ALGORITHM</u>:

The purpose of this algorithm is to detect the differences that may exist between the input and output orthographic views. The most obvious and simplest test is to compare the number of nodes and edges in the input views with the number of nodes and edges in the corresponding output views. However, it is possible for two views to comprise the same number of nodes and edges and yet may not be similar. For this reason, in addition to this simple test, the algorithm has been developed to include the following steps:

a) Search for a 'matching node' in the output view for each node in the corresponding input view. A node is defined as having a 'matching' node only if the coordinate values differ by not more than a preset tolerance. The matching of two nodes is independent of the node numbers, since the nodes in the input views are numbered in a different sequence from the nodes in the output views.

b) Search for a 'matching edge' in the output view for each edge in the input view. An edge is defined as having a 'matching edge' only if both edges have matching start and end nodes, and they are both of the same type, i.e. either both straight edges, or both circular arcs in which case they must also have the same centre.

The search for matching edges is initiated only if all the nodes in three input views have a matching nodes in the corresponding output views. The specifications of the algorithm may be as follows:

STEP 1: Compare the number of nodes in the input XY view with the number of nodes in the output XY view. If these numbers are equal then go to step (2), otherwise go to step (13).

STEP 2: Compare the number of edges in the input XY view with the number of edges in the output XY view. If these numbers are equal then go to step (3), otherwise go to step (13).

STEP 3: Compare the number of nodes in the input YZ view with the number of nodes in the output YZ view. If these numbers are equal then go to step (4), otherwise go to step (13).

STEP 4: Compare the number of edges in the input YZ view with the number of edges in the output YZ view. If these numbers are equal then go to step (5), otherwise go to step (13).

STEP 5: Compare the number of nodes in the input XZ view with the number of nodes in the output XZ view. If these numbers are equal then go to step (6), otherwise go to step (13).

STEP 6: Compare the number of edges in the input XZ view with the number of edges in the output XZ view. If these numbers are equal then go to step (7), otherwise go to step (13).

STEP 7: For each node in the input XY view, find a 'matching node' in the output XY view. If such a matching node does not exist then go to step (13).

STEP 8: For each edge in the input XY view, find a 'matching edge' in the output XY view. If such a matching edge does not exist then go to step (13).

STEP 9: For each node in the input XZ view, find a 'matching node' in the output YZ view. If such a matching node does not exist then go to step (13).

STEP 10: For each edge in the input XZ view, find a 'matching edge' in the output XZ view. If such a matching edge does not exist then go to step (13).

STEP 11: For each node in the input YZ view, find a 'matching node' in the output YZ view. If such a matching node does not exist then go to step (13).

STEP 12: For each edge in the input YZ view, find a 'matching edge' in the output YZ view. If such a matching edge does not exist then go to step (13), otherwise go to step (15).

STEP 13: If the object has been classified as a prismatic object then go to step (14), otherwise go to step (15).

STEP 14: The input and output views should have been similar. Inform the user that there has been an error. Exit.

STEP 15: All the input and corresponding output orthographic views are similar. Inform the user that the generated model is the exact object. Exit.

STEP 16: The input and output orthographic views. Produce a list of differences (nodes and edges numbers). Inform the user that the generated model is an approximation model, and that further processing is required in order to reconstruct the exact object, or to obtain a more refined model. Exit.

The last step of the above algorithm represents, in effect, the only instance where interaction with the user may be required. A choice is here given to the user on whether to terminate, or to allow the interpretation process to continue in order to generate another model which may then either be the exact object, or another but more refined approximation model.

6.4) <u>FEED BACK ALGORITHMS</u>:

Discrepancies between the input and output orthographic views indicate that the output model requires further processing in order to generate either the exact object, or a more refined model. This process, which consists of identifying and subtracting one or more subobjects from the output model, is initiated by the feed back subprocess algorithms whose main function is to examine the original input views and the orthographic views of the output model, in order to generate the orthographic views of such subobjects. The geometric and topological data related to the orthographic projections of these subobjects are then fed back as input to the analysis process, and interpreted as solid models in a similar fashion to the original input views.

The first step in the feed back subprocess consists of generating the views of a wireframe which is defined by combining the wireframe of the input object with the wireframe of the output model. It can be clearly observed from Figure 6.12, that the purpose of such a wireframe is to define the wireframes of the subobjects that are to be removed from the output model. Such a wireframe, referred to here as the 'pseudo-wireframe', may not be directly generated since the input object is yet to be reconstructed; however, the orthographic projections of the pseudo-wireframe, shown in Figure 6.13, and referred to hereafter as the pseudo-views, can be obtained directly from the input views and the orthographic projections of the output model. Furthermore, the orthographic views of subobjects defined by the pseudo-wireframe, are clearly visible in the pseudo-views, shown in Figure 6.13 as

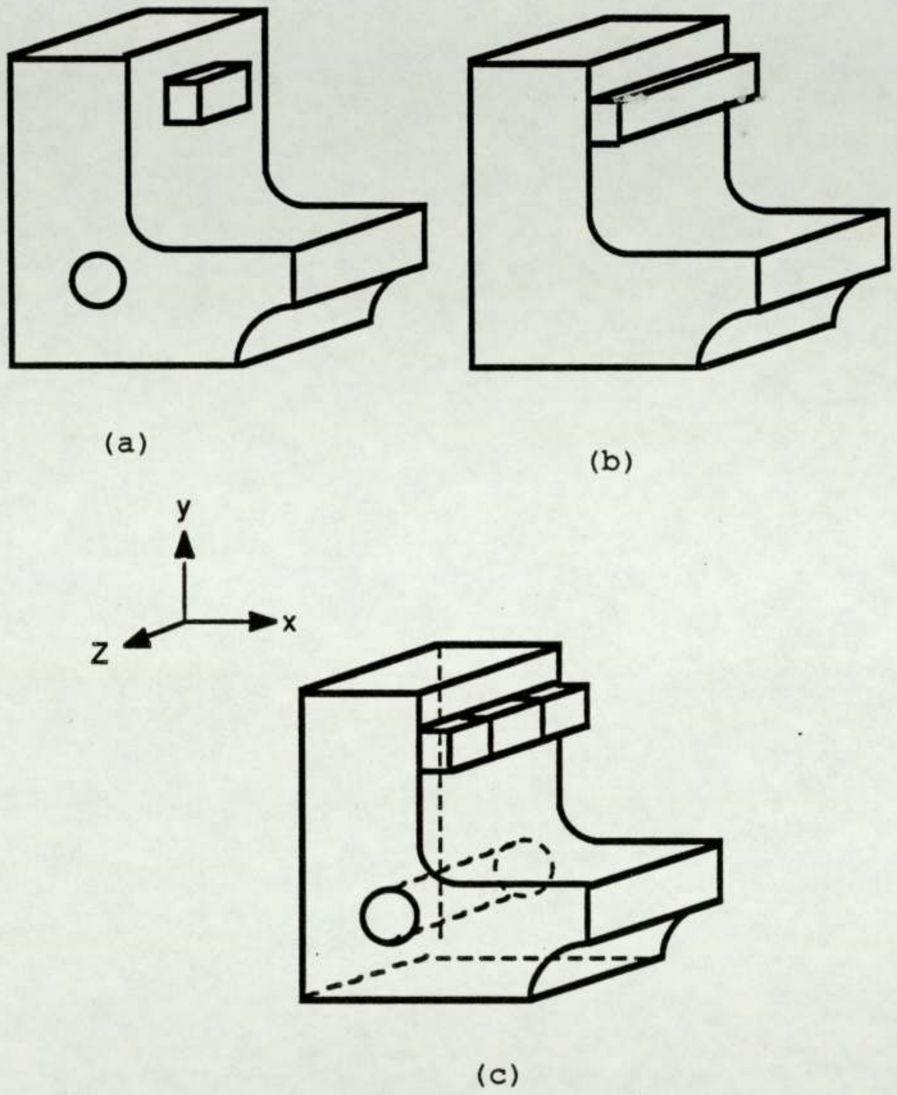
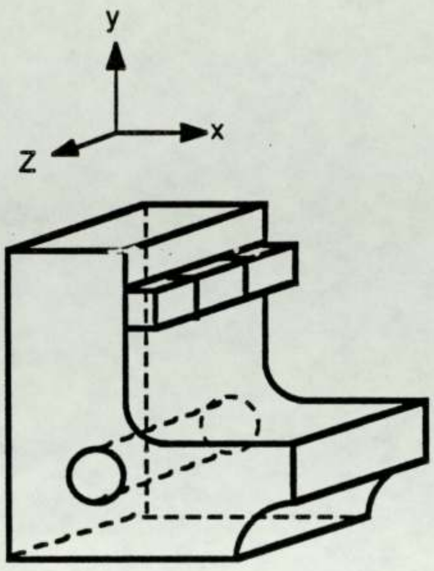
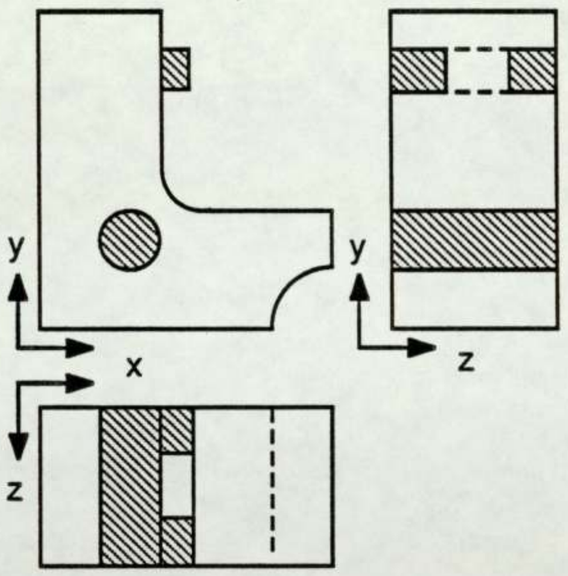


Fig. 6.12: a) Input object
b) 1st approximation model
c) 'Pseudo-wireframe'



(a)



(b)

Fig. 6.13: a) A pseudo-wireframe, and
b) its orthographic projections (pseudo-views)

hatched areas. The pseudo-views are generated by the 'Pseudo-views Generator' algorithm described below in section 6.4.1.

The next step in the feed back subprocess is to extract from the pseudo-views the orthographic projections corresponding to each subobject that is to be removed from the output model. This task is performed by the 'Feed Back Data Generator' algorithm described below in section 6.4.2.

6.4.1) THE 'PSEUDO-VIEWS GENERATOR' ALGORITHM:

Pseudo-views are generated by 'assembling' all the nodes and edges of the original input views with those which describe the orthographic views of the output model. The first step in the process of 'assembling' these entities is to identify the nodes which are not common to both set of projections. Such nodes are referred to hereafter as either input, or output, 'Active' nodes. Input active nodes, such as nodes 1, 4,7 and 10 in the YZ in Figure 6.14(a), are nodes which exist in one of the original input views but which do not have matching nodes in the corresponding view of the output model. The definition of a match is similar to the one used in the 'Comparison' algorithm, described in section 6.3.2.

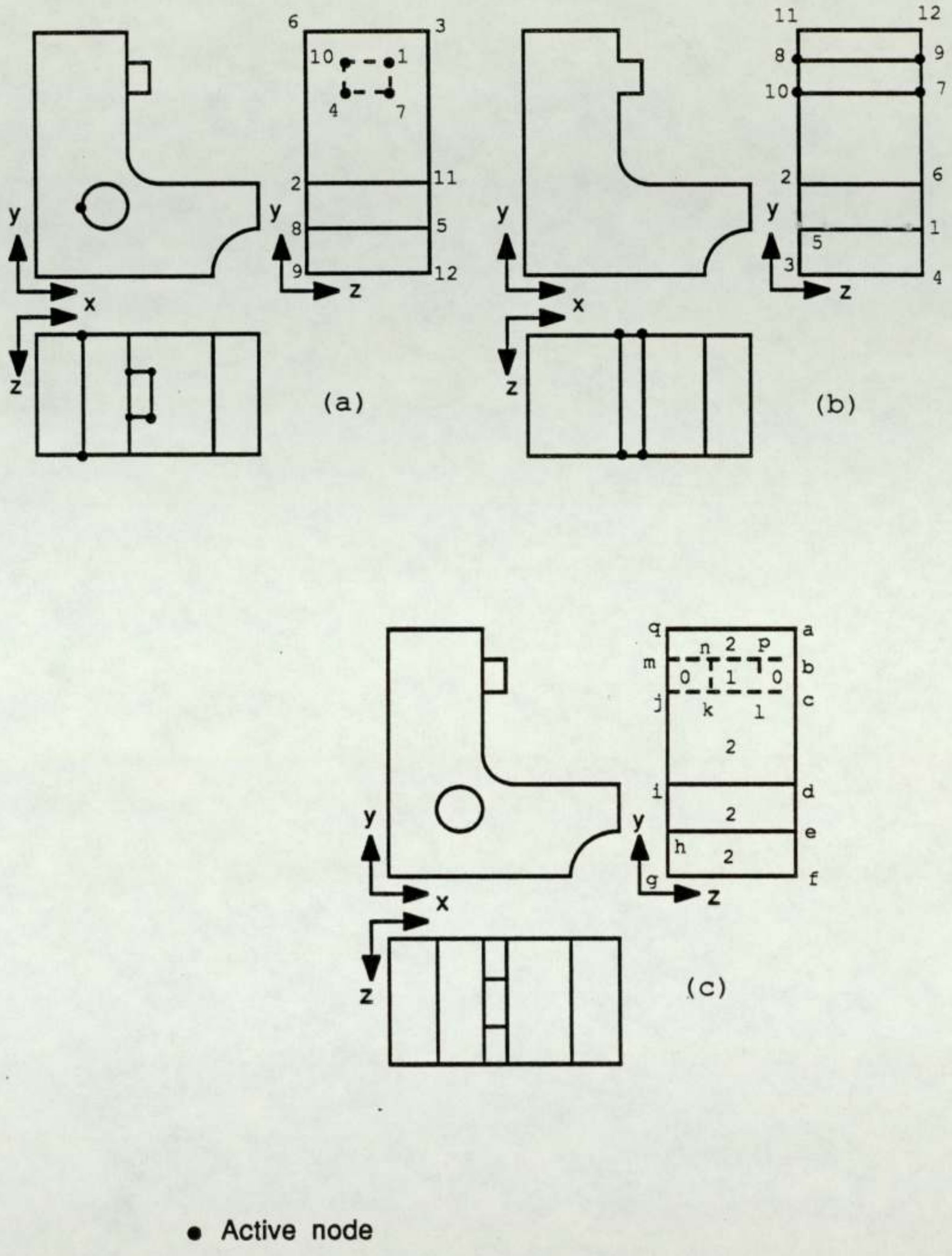


Fig. 6.14: a) Original input views,
b) orthographic views of first approximation model, and
c) corresponding pseudo-views

The next step is to 'fit' any input, or output, active node to the output, or input, views, respectively. A 'fit' consists of adding an active node to a view by splitting the edge on which it lies, in that view, into two new edges. For example, the input active nodes 7 and 9, in the YZ view of the output model, shown in Figure 6.14(b), are fitted into the input YZ view, in Figure 6.14(a), by splitting input edge {3,11}, into three new edges, which are shown as edges {A,B}, {B,C} and {C,D} in Figure 6.14(c). A pseudo-view is then generated by adding to the input view those edges which are not common to both input and corresponding output views.

The specifications of the algorithm may be described as follows:

STEP 1: I = 0

STEP 2: I = I + 1; If I > 3 then go to step (8)

STEP 3: Search in the Ith input view for nodes that do not have a matching node in the corresponding Ith output view. Store these nodes, if any, as input active nodes.

STEP 4: Search in the Ith output view for nodes that do not have a matching node in the corresponding Ith input view. Store these nodes, if any, as output active nodes.

STEP 5: Search in the Ith output view for edges which may be colinear with the input active nodes of the corresponding Ith input view. Split such edges, and update the geometry and topology of the Ith output view, accordingly.

STEP 6: Search in the Ith input view for edges which may be colinear with the output active nodes of the corresponding Ith output view. Split such edges, and update the geometry and topology of the Ith input view, accordingly.

STEP 7: Generate the Ith pseudo-view by adding to the Ith input view, all the edges which are in the Ith output view only. Go to step (2)

STEP 8: Exit.

6.4.2) <u>THE 'FEED BACK DATA GENERATOR' ALGORITHM</u>:

This algorithm has been developed to perform the task of extracting from the pseudo-views, the data corresponding to sets of orthographic projections of any subobject that is to be removed from the output model. The first step of the algorithm consists of identifying all the loops in the pseudo-views. This is achieved by applying the 'Loop Detector' algorithm described in section 6.1.1. The loops are then labelled as follows:

0 for a loop which does not exist in either input or corresponding output views.

1 for a loop which exists in the input views only.

2 for a loop which exists in the output views only.

3 for a loop which exists in both input and

corresponding output views.

A loop in a view is said to exist in another view, only if there is a loop, in the that view, which meets the following conditions:

a) Both loops have exactly the same number of nodes and edges, where each node in one loop has a matching node in the other loop.
b) Each edge in one loop has a similar edge in the other loop. Similar edges are defined as edges which are of the same type, i.e straight edges or circular arcs, and the start and end nodes of one edge are the matching nodes of the other edge. Furthermore, in the case where the edges are circular arcs, the centres of both arcs must have the same coordinate values.

For example, in Figure 6.14(c), the loop defined by nodes b, c, l and p, in the YZ pseudo-view, has been labelled as '0' because it does not exist in either the YZ input view, or in the corresponding YZ output view. Whereas, loop (k, l, p, n) in the YZ pseudo-view, has been labelled as '1' because it exists in the YZ input view, but not in the YZ output view.

The purpose of such a labelling process is to enable loops, which represent projections of subobjects, to be readily identified from the pseudo-views. It has been found that such loops may always be identified as follows:

- any 'connected' loops labelled as '0',
- any disjoint loop labelled as '1'.

It has been found, as expected, that some loops in the pseudo-views can only have specific labels, because of the manner in which the interpretation process is implemented for non-prismatic objects, and the manner in which pseudo-views are generated. For instance:

a) perimeter, or boundary, loops in the pseudo-views, will always be labelled as '3', since the initial step in interpreting non-prismatic objects consists of processing the perimeter loop of each input view only (section 5.4.3), and resulting in an output model whose perimeter loops are the same as the perimiter loops in the corresponding output views.
b) disjoint loops in the pseudo-views can only be labelled as '1' or '3'. A disjoint loop in the input views may, or may not, exist in the corresponding output views, in which cases, it is labelled as '3', or '1', respectively, in the pseudo-views. A disjoint loop in the output views also, may or may not, exist in the corresponding input views. If it exists in both views then, as before , it is labelled as '3'; But, because the process of generating pseudo-views generally consists of splitting edges in both

input and output views, and adding edges to the input view, any disjoint loop in the output projections which does not exist in the input views, is always divided into a number of connected loops. Thus, a disjoint loop in the pseudo-views may never be labelled as '2', nor as '0'. These interesting results may be used to cross-check the data associated with the orthographic projections of the output model and those corresponding to the pseudo-views. For example, if a perimeter loop in a pseudo-views has been labelled as '0', '1' or '2', then it can be said that the output model generated is not the correct model. Also, if a disjoint loop in the pseudo-views has been labelled as '0', or '2', then it would indicate that the output model is again not the correct model, or that an error has occurred in generating the pseudo-views.

The next step in the algorithm consists of searching in the pseudo-views, for matching subobject loops. A subobject loop in a XY pseudo-view is defined as having matching subobject loops in adjacent YZ and XZ pseudo-views, only when their surrounding rectangles defined, respectively, by:

(X_{xymin}, Y_{xymin}, Y_{xymax}, y_{xymax}), (Z_{yzmin}, Y_{yzmin}, Z_{yzmax}, Y_{yzmax}), and (X_{xzmin}, Z_{xzmin}, X_{xzmax}, Z_{xzmax})

meet the following conditions:

$X_{xymin} = X_{xzmin}$ $\quad X_{xymax} = X_{xzmax}$

$Y_{xymin} = Y_{yzmin}$ $\quad Y_{xymax} = Y_{yzmax}$

$Z_{xzmin} = Z_{yzmin}$ $\quad Z_{xzmax} = Z_{yzmax}$

where:

X_{xymin}, X_{xymax} = minimum and maximum of the X-coordinate values of the loop in the XY view.

X_{xzmin}, X_{xzmax} = minimum and maximum of the X-coordinate values of the loop in the XZ view.

Y_{xymin}, Y_{xymax} = minimum and maximum of the Y-coordinate values of the loop in the XY view.

Y_{yzmin}, Y_{yzmax} = minimum and maximum of the Y-coordinate values of the loop in the YZ view.

Z_{xzmin}, Z_{xzmax} = minimum and maximum of the Z-coordinate values of the loop in the XZ view.

Z_{yzmin}, Z_{yzmax} = minimum and maximum of the Z-coordinate values of the loop in the YZ view.

It is assumed that all the loops in the pseudo-views have been determined, and that the number of loops in each pseudo-view is stored the array NLP. The specifications of the algorithm are as follows:

```
STEP 1: I = 0;
STEP 2: I = I + 1; J = 0; If I > 3 then go to step (6)
```

STEP 3: J = J + 1; If J > NLP(I) then go to step (2)

STEP 4: Search in the Ith output view, for a loop which is similar to the Jth loop in the Ith pseudo-view. If such a loop exists, then label the Jth loop in the Ith pseudo-view, as '2', otherwise as '0'.

STEP 5: Search in the Ith input view, for a loop which is similar to the Jth loop in the Ith pseudo-view. If such a loop exists, then label the Jth loop in the Ith pseudo-view, as '3' if it is already labelled as '2', or as '1' if it is already labelled as '0', otherwise label it as '0'.

STEP 6: Check if the perimiter loop in the Ith pseudo-view is labelled as '3'; If not then go to step (10).

STEP 7: Check if a disjoint loop (if any) in the Ith pseudo-view is labelled as '0' or '2'. If yes then go to step (10), otherwise go to step (3).

STEP 8: For each 'connected' loop labelled as '0' in a pseudo-view, search in adjacent pseudo-views for matching loops. If a match exists, store the data of the three matching loops, as they represent the orthographic views of a subobject.

STEP 9: For each disjoint loop labelled as '1' in a pseudo-view, search in adjacent pseudo-views for matching loops. If a match exists, store the data of the three matching loops, as they represent the orthographic views of a subobject. Go to step (11)

STEP 10: Error.

STEP 11: Exit.

The output of the algorithm is a set of data files which comprise the data corresponding to the orthographic projections of a number of subobjects. These subobjects are then identified as solid models which are then subtracted from the output model to yield, as mentioned above, either the exact model represented by the original input orthographic views, or another, but more accurate, approximation model. In the latter case, the complete process represents one complete iteration. An example illustrating such iteration, is presented later in section 8.5.2.

All the algorithms described above, have been implemented, and the corresponding source code, has been written in FORTRAN 77, on the Apollo DN3000 workstation. Such software, referred to as C.I.E.D.S.M. (Computer Interpretation of Engineering Drawings as Solid Models), has been designed with speed as a major criterion, because of the iterative aspect of the process. The overall design of C.I.E.D.S.M. software is described in the next chapter.

CHAPTER SEVEN

THE C.I.E.D.S.M. SOFTWARE

7.1) INTRODUCTION:

The aim of this chapter is briefly to describe the software developed in this project, with special emphasis on its implementation on the Apollo DN3000 workstation. For convenience, the suite of programs has been dubbed C.I.E.D.S.M. (**C**omputer **I**nterpretation of **E**ngineering **D**rawing as **S**olid **M**odels).

Many of the routines were originally developed on an ICL Perq 2 workstation, but subsequently transferred to the Apollo DN3000 workstation. Direct down-loading from the ICL Perq 2 to the Apollo DN3000 was not possible because of hardware incompatibilities and lack of communication software. The transfer has been made possible by first transferring it from the Perq 2 computer to a VAX 11/780 mainframe, and then down-loading it to the Apollo Domain workstation. Very few modifications to the software were necessary. Such a transfer was required because a compatible subroutine version of the solid modeller 'BOXER' was not supported on the ICL Perq2 computer, whereas such software was readily available on the Apollo Domain workstation.

The software objectives are outlined in section 7.2 and a brief description of a number of subroutines is given in section 7.3. Some of the routines have been written to integrate the PAFEC 'BOXER' solid modeller into the software. These are highlighted to indicate the possible modifications that must be made, should any other CSG solid modeller be contemplated.

Considerations about software design, such as portability, have been made from the start of the project, as it is highly possible that the software developed in this work may be implemented on a different system. Thus the subject of section 7.4 is software portability. Some aspects concerning data storage and execution speed are also discussed in section 7.5. A set of operating instructions for the package are given in section 7.6. These instructions are mainly associated with data acquisition at the input stage, since the program has been designed to require a minimum of user interaction.

7.2) SOFTWARE OBJECTIVES:

The principal objectives of the software are:

1) the implementation of all the algorithms developed in this project, (described in chapters 6 and 7), on a computer workstation.
2) Design and development of an interface with a solid modeller
3) Design and development of graphic facilities capable of displaying the input views and the orthographic

projections of the generated model

4) Design and development of facilities capable of plotting the input views and the orthographic projection of the generated model, on a Roland DG DPX 2000 plotter.

7.3) <u>THE C.I.E.D.S.M. PROGRAM</u>:

The C.I.E.D.S.M. program has been designed to take full advantage of the structured nature of the FORTRAN language. It comprises 54 overlaid subroutines which are called from the main program 'MAIN'. The subroutines may be conveniently divided into two categories:

a) 'Utility' routines
b) 'Process' routines

Utility subroutines, have been developed to perform simple geometric computations. For example, the routine 'PERPD' computes the coordinates of a point P at which a perpendicular from another point D intersects a line; the perpendicular distance PD is also computed; the subroutine 'POLAR' converts Cartesian coordinates into polar coordinates; etc. Other utility routines have been designed and developed to perform simple tasks such as files and data handling, and plotting. For example, the routine called 'TRANSF', reads a direct access file and transfers the data to a number of integer and real arrays, and the routine PLOTTER provides a plotting facility on the DPX 2000 plotter. Space does not permit the inclusion of details of these subroutines in this thesis. Suffice it to say that considerable use was made of the techniques of 'Homogeneous Coordinates', some of which are briefly described in appendix C.

The 'Process' subroutines, briefly described below, are the direct conversion of the interpretation process algorithms into FORTRAN 77 source code. Some of these routines are marked by asterisks (*) to indicate that they are dependent on the solid modeller in use; they form the necessary 'interface' between the solid modeller and the algorithms developed in this project. The term 'interface' refers to the generation of the solid modelling input file as described in section 6.2, and the extraction of the 2D data from the solid modeller output file described in section 6.3.1. The 'process' subroutines are:

INPUT: prompts the user to present input orthographic views data which are then stored in three separate random access files - one file per view.

CYCLES: scans the input data in order to determine the number of loops in each view. The perimeter loop is always the first to be determined.

EXTREM: scans the input data to compute the extreme coordinate values in each view.

<u>TOPUP</u>: updates the topology and geometry data.

<u>LINK</u>: finds all the nodes joined (adjacent) to a given node.

<u>MARKER</u>: labels edges according to the 'Loop Detector' algorithm which was presented in section 6.1.1.

<u>RELATE</u>: determines relationships between all the loops in a given view. It also labels each loop as 'DISJOINT' or 'CONNECTED'.

<u>TEST3</u>: checks if a given loop is a rectangle.

<u>PROCLP</u>: selects object loops for processing in order to generate a 'profile', as defined in section 5.4.3.

<u>PLOOPS</u>: Processes the object loops selected by the subroutine PROCLP.

<u>PATT1</u>: scans the topological and geometrical data of a given object loop to determine its 2D geometric pattern.

<u>PATT2</u>: scans the topological and geometrical data of a given primitive loop to determine its 2D geometric pattern.

SETREC: sets the coordinate values of the nodes of the rectangle surrounding a given loop.

RNUMR: renumbers the nodes of the surrounding rectangle according to the sense of the loop it is surrounding.

CFLAGS: allocates a flag to each node of a given loop, according to its position with reference to the nodes and sides of the surrounding rectangle. This routine generates, in effect, the 'control list', as described in section 6.1.6.

LOCAT: scans the control list generated by the routine CFLAGS, in order to locate all the primitive loops formed by the intersection of a given loop and its surrounding rectangle.

MERGEP: merges direct access files into the MAINDATA file.

TESTAR: scans the MAINDATA file in order to search for loops identified and stored as arbitrary patterns. If such a loop is found, then its corresponding geometric and topological data is stored in the random access file called ADATA.

UNTEST: scans the data stored in the ADATA file to test the arbitrary loop for 'instability' conditions, mentioned in section 6.1.7. If such conditions are found, then the loop is divided into two loops and their data stored back into the MAINDATA file.

OUTLIN: reads the final MAINDATA file to extract the data corresponding to all the basic patterns, and performs 2D Boolean operations to reconstruct the corresponding object loop. This routine is used as a means of cross-checking the reconstruction process.

BTREE1: stores the pointers which relate parent loops to their children loops. It generates and stores a Boolean tree.

PRIMID: scans the final MAINDATA file to read the flag LPF of all the basic patterns, in order to identify the 3D primitives.

OBNAME*: sets the object name according to text format required by the solid modeller 'BOXER'.

BLOCK*: computes the parameters of a 3D primitive block, as required by the solid modeller 'BOXER'.

WEDGE*: computes the parameters of a 3D primitive wedge, as required by the solid modeller 'BOXER'.

FILLET*: computes the parameters of a 3D primitive fillet, as required by the solid modeller 'BOXER'.

CYL1*: computes the parameters of a 3D primitive cylinder, as required by the solid modeller 'BOXER'.

CYL2*: computes the parameters of a 3D primitive cylinderical segment, as required by the solid modeller 'BOXER'.

BTREE2*: generates and stores character strings which represent the syntax of the solid modeller input file. It uses the pointers generated by the routine BTREE1, the object names generated by OBNAME, and the parameters from BLOCK, WEDGE, FILLET, CYL1 or CYL2.

WTREE*: writes the character strings generated by BTREE2 onto a file called 'BOXER.DAT', which is used as input to solid modeller 'BOXER'.

EXTRACT*: scans the solid modeller output file, DOGSDAT, in order to extract the 2D data which represents the orthographic projections of the generated model. This routine also rectifies the extracted data by adding or subtracting the amount of shift which exists between the data which represent the input views and the extracted data.

CVIEWS: scans and compares the input views data and the data which represent the orthographic views of the output model. If discrepancies exist between the two sets of views, then a flag, NCHECK, is set to 1, otherwise it remains equal to zero.

ACTIVN: detects 'active' nodes (if any) in both input and output views.

UVIEWS: generates the data which represents the 'Pseudo-views', as defined in section 6.4.1.

LABELV: labels each loop obtained in the pseudo-views by comparing the data associated with each loop found in the input and the views of the output model.

MATCH: determines the matching set of loops, in the pseudo-views, which represent the orthographic projections of a subobject that must be removed from the output model. The topological and geometric data associated with a set of matching loops, are stored in random access files whose structure is similar to the original input files.

DRAW: draws and displays the input views, the orthographic projections of the output model and the corresponding pseudo-views.

All the routines, including the main program, are grouped and stored in different files, which are compiled separately to generate the corresponding binary files. The binary files are then bound together using a link file.

The link file must also comprise the binary file of the subroutine version of the solid modeller.

7.4) <u>SOFTWARE PORTABILITY</u>:

The C.I.E.D.S.M. software has been developed with portability as one of the major requirements. It is desirable for any software readily to be able to be converted to suit a new operating environment, since all prospective users are unlikely to have the same computer system. Of course, it is not yet possible to produce any software which is universally portable, however it is possible to design the software, so that a minimum effort is required in modifying it for transfer from one system to another.

It is highly probable that the software developed in this project may be implemented on a different system in the near future. For such reasons, the C.I.E.D.S.M. suite of routines have all been written in FORTRAN 77, a high level language which is highly portable and which is one of the most popular scientific languages used in software practice. Such portability was demonstrated when the software was transferred from the ICL Perq2 computer to the Apollo Domain workstation.

Portability considerations have been expanded further by keeping the number of routines associated with the solid modeller to a minimum, as it is also unlikely that all prospective users will use the same solid modeller. These routines, highlighted above, have been developed to provide the necessary syntax as required by the PAFEC BOXER solid modeller, and extract data from files generated by it. If, in the future, a different solid modeller is used with the software developed in this work, then it is necessary to modify these routines accordingly.

Considerations have also been given to graphic languages. Initially, when the software was being developed on the ICL Perq2, graphics routines were written using GKS (Graphic Kernel System) libraries [46,47]. This choice is justified as GKS is considered as a standard, although it is still to be improved. Unfortunately, GKS was not available on the Apollo DN3000 workstation at the University of Aston. New graphics routines have been developed and implemented on the Apollo DN3000 workstation, initially using the Domain 2D Graphic Primitives Resources (GPR) [48], and later, the Domain 2D Graphic Metafiles Resources (GMR) [49] packages. GMR, an extension of GPR, provides extended facilities for developing and storing graphics data.

7.5) EXECUTION SPEED AND DATA STORAGE:

One of the major requirements in designing software for iterative problem solving techniques, is execution speed, because of the potentially slow and repetitive nature of the process involved in reaching a solution. Generally, the execution speed of such programs can be improved but at the cost of memory and storage space.

The interpretation process developed in this project is iterative, and thus, execution speed was one of the major requirements that had been taken into consideration in designing and developing the C.I.E.D.S.M. software. Speed has been optimized by making considerable use of the speed optimizing and data storage saving capabilities of FORTRAN 77. For example, using unformatted random access files rather than sequential ones, and making use of common blocks.

Furthermore, the construction of a solid model by the solid modeller is a time consuming operation. Thus, in order to improve the execution speed of the whole process, excessive and unnecessary use of the solid modeller during iterations, has been avoided; the construction of any solid model is performed only once, and never repeated. A considerable improvement in the execution speed was also observed when the software was transferred from the ICL Perq2 computer to the Apollo DN3000 workstation.

Little computer storage is necessary to implement the routines developed in this work. The source code and compiled binary files, occupy at least 590 Kb on the hard disk. However, the solid modeller 'BOXER' requires at least 6 Mb of storage capacity. Thus, the acquired system should provide at least 6.6 Mb of storage capacity. The Apollo DN3000 workstation has an ample amount of virtual memory (2 Mb) and data storage capacity (72 Mb).

With the exception of the solid modeller input and output files, all the files used to store the total set of data used by the C.I.E.D.S.M. software are random access ones. These files may be grouped into two types. The first store the topological and geometric data which represent orthographic projections. These are:

- files that store the xy, xz and yz input view data, conveniently called 'xyi', 'xzi' and 'yzi'
- files that store the xy, xz and yz orthographic projections data of the output model, called 'xyo', 'xzo' and 'yzo', respectively
- files that store the xy, xz and yz pseudo-views data, called 'xyp', 'xzp' and 'yzp', respectively.

All the above files, including others, such as EXDAT, ADATA and ARBDATA files, have the same structure, illustrated in Figure 7.1.

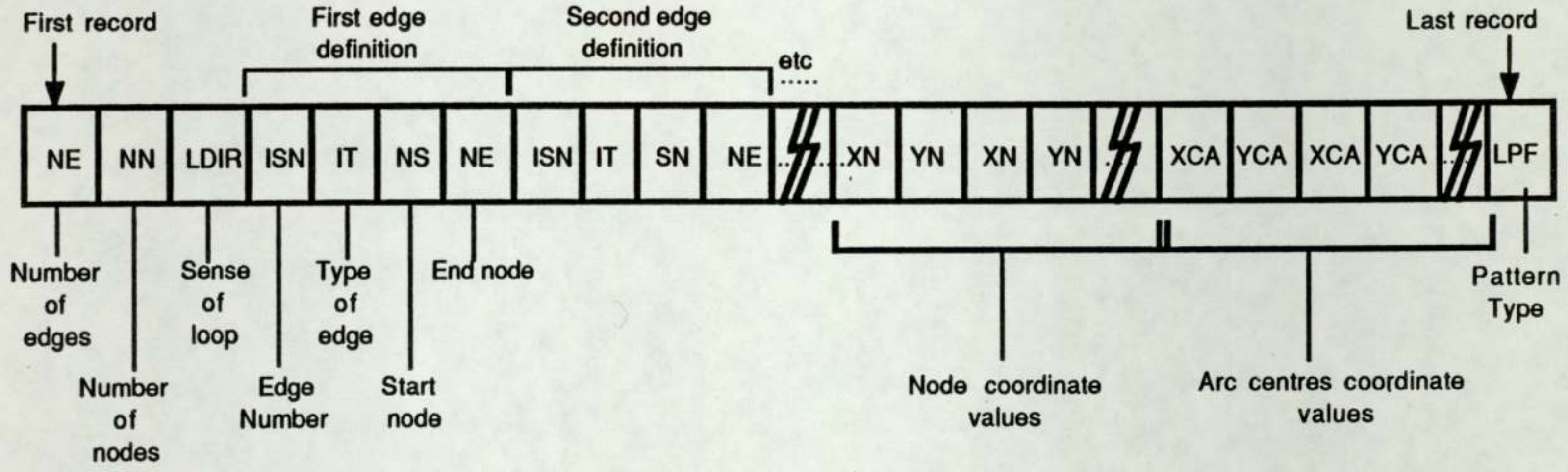


Fig. 7.1: Structure of "ADATA" file

The first two records are two 4-byte integers, NG and NN, which represents the number of edges and nodes, respectively, in each view. The third record is also a 4_byte integer, LDIR, which is set to:

i) C to indicate that a loop has a clockwise sense
ii) 1 to indicate that a loop has an anticlockwise sense.

The subsequent records are grouped in three sets. The first set of records is divided into a number of groups of four 4-byte integers. Each group corresponds to the definition of an edge. Thus, the number of such groups, in this first set of records, is equal to the number of edges. The first integer, in each group, is the edge number, ISN. The second integer, IT, is set to either 0, to indicate that the edge is a straight edge, or to a positive (or negative) number to indicate that the edge is an anticlockwise (or clockwise) arc whose centre coordinates values may be found at the address specified by the absolute value of IT. The last two integers in each group, are the start and end nodes, NS and NE, defining each edge.

The next set, comprises groups of pairs of records. Each pair of records stores the X and Y coordinate real values, XN and YN, of each node. The last set is also divided into groups of pairs of records which store the X and Y coordinate real values, XCA and YCA, of the centre of each circular arc (if any). The last record in the each section is the 4-byte integer, LPF, which, as defined previously, identifies the 2D geometric shape, or pattern, of the loop.

The other type of files store the data associated with the loops in each view, and these are as follows:

- 'XYLI', 'XZLI' and 'YZLI' which store the data of all the loops in the xy, xz and yz input views
- 'XYLO', 'XZLO' and 'YZLO' which store the data of all the loops in the xy, xz and yz views of the output model
- 'XYLP', 'XZLP' and 'YZLP' which store the data of all the loops in the xy, xz and yz pseudo-views.

The structure of one of these files is illustrated in Figure 7.2. The first record is a 4-byte integer which represents the number, NLP, of 'circuits', or loops, in a given view. The subsequent records are grouped into sets, where each set of records comprises the data associated with each loop. Thus, the number of sets is equal to the number of loops in the view. The first set of records always stores the data of the perimeter loop. The first record in each set, is also a 4-byte integer which indicates the number, LN, of nodes contained in the loop. The next records store the numbers, N_1, N_2, N_3 .. $N_{LN,}$ of all the nodes in the loop. The last record in the set, store the 4-byte integer, LT, which identifies the loop as

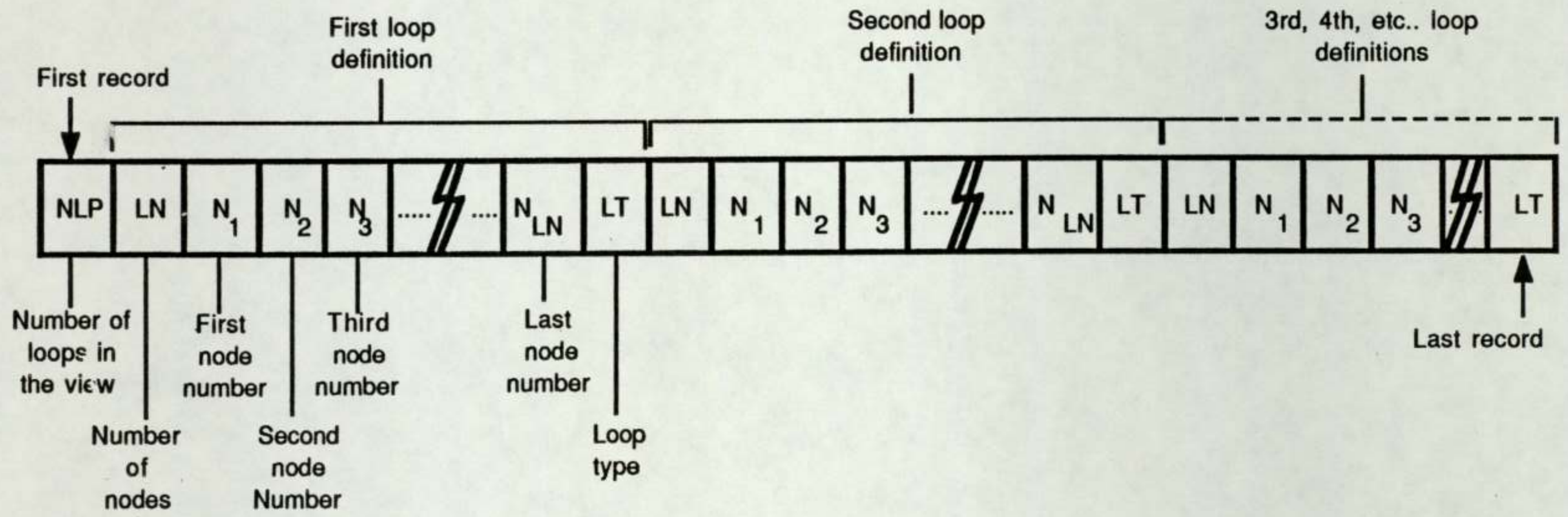


Fig. 7.2: Structure of "XYLP" file

'connected' if set to 1, or 'disjoint' if set to 0.

7.6) OPERATING INSTRUCTIONS FOR C.I.E.D.S.M.:

The C.I.E.D.S.M software has been designed and developed for automatically converting the 2D data which represent orthographic projections of an object, into a solid model. Interaction with the user is restricted to data acquisition, except where iteration is required, in which case user intervention is also requested. Prior to running the program, the user must have prepared the input data which comprises the topological and geometric data of three orthographic views described in the first-angle projection system. Such data comprise edge numbers, and types, node numbers and coordinate values, as well as arc centres (if any).

The executable file of the program is already stored on the hard disk of the Apollo workstation. To run the program, the user must type CIEDSM, and hit the RETURN key. He is then warned about certain file names that he must not enter, as they are used by the system. The CIEDSM dialogue has been designed so that the user responds to a question that has a Yes/No answer by simply hitting the RETURN key for 'Yes'. Hitting any other key indicates a 'No' answer. User information is then structured as follows:

Is data for XY, XZ or YZ view ?

Please, enter 'XY', 'XZ' or 'YZ'.

Type either XY, XZ or YZ, which indicates the view to which the data correspond. There are no restrictions on the order in which the view data are entered. For instance, if you wish to enter the XZ view first, then you may do so. The next prompt is:

Is it a new file (RETURN = 'YES') ?

If you have used the program before, then you may have a number of files already stored on the hard disk, which you would like to use. In this case, you might type 'x', and the program would respond by the following question:

Old file. File name ?

File names may have up to 10 characters: you should type a name and hit RETURN. Since you have indicated that the file name is for a file which is already stored on the hard disk, CIEDSM searches for such a name. If the name is not found, the following message is displayed:

Error. Such a file does not exist.

Do you wish to continue (RETURN = Yes) ?

If you wish to continue, you must hit RETURN, and the above steps are repeated.

New input data may be entered by hitting RETURN at the following prompt:

Is it a new file (RETURN = 'YES') ?

The program then responds with the following question:

New file. File name ?

Type a name and hit RETURN. The program sets the interactive input mode for entering the topological and geometric data of the input view. The first prompt is as follows:

XY view:

Enter number of edges.

to which you must type an integer value which indicates the total number of edges in the given view. This is then followed by:

XY view:

Enter number of nodes.

Type another integer value which indicates the total number of nodes in the view. The program allows you to check the input data by displaying the number of edges and nodes entered. You may then either hit RETURN to indicate that the data entered is correct, or hit any other key to indicate that the data is to be modified. In the latter case, the program displays the last two prompts inviting you to re-enter the number of edges and nodes. These last two steps are repeated until the input is acknowledged to be correct. The program then carries on requesting data associated with each edge, by displaying the following prompt:

Edge 1:

Enter: 0 if edge is a straight line

1 if the edge is a clockwise arc

-1 if the edge is an anticlockwise arc

Type either one of the integer values displayed depending on the type of edge number 1. The next prompt invites you to input the start and end node numbers of the edge, which in this case is edge number 1. The prompt is as follows:

Edge 1:

Enter start and end node numbers.

You should respond by typing two integer values; the first one indicates the start node number, and the second the end node number. The last two prompts are then displayed again, in sequence, for edge number 2, and then edge number 3, and so on. Thus, allowing the data corresponding to the type, start and end node numbers of all the edges of that view, to be entered. The program then provides an instant check by displaying all the previous input data followed by this prompt:

Is data correct (RETURN = 'YES') ?

Again you may either acknowledge that the data is correct by hitting RETURN, or you may wish to modify a specific value, by hitting any other key, in which case the following prompt is displayed:

Enter edge number to modify

Type an integer value indicating the edge he wishes to modify. You are then invited to reenter the correct type, start and end nodes of that specific edge. The check is repeated until the data is acknowledged to be correct, thus completing the topology input data. The next steps are concerned with the input of the geometrical data, which comprises the coordinate values of all the nodes and centre of arcs (if any). The program displays the following prompt:

<u>Node 1:</u>

Enter X and Y coordinate values.

Type two real values; the first one is the X coordinate value, and the second is the Y coordinate value of node number 1. This step is repeated until all the coordinate values of all the nodes in the view are entered. The program then scans the type of all the edges in order to check for circular arcs; If the view comprises such an edge, then the edge number is displayed, and you are informed that this particular edge is a circular arc, and then prompted to input the coordinate values of the centre of that arc. Assuming that you have previously specified that edge number 11 is of type 1, which indicated that edge number 11 is a clockwise circular arc, thus the prompt would be as follows:

<u>Edge 11 is a circular arc</u>

Please, enter X and Y coordinate values of its centre.

You must then input two real values which represent the X and Y coordinate values of the centre of that particular circular arc edge, in this case edge number 11. If the view comprises several circular arcs, then this step is repeated a number of times equal to the number of such edges. Again, the coordinate values of all the nodes, followed by the coordinate values of centre of arcs (if any), are displayed to enable you to check your input data. These values may be modified at this stage, if you wish to do so. The modifications are carried out as previously indicated.

All the input steps must then be repeated for the two remaining views. After completing the input of the data of all the views, you are then immediately informed that the process of converting the 2D data into a 3D solid model has started, by the following message:

```
***************************
*** ANALYSIS STEP ***
***************************
```

which is shortly followed by another message informing you that the data has been analysed, and the input data represent the orthographic views of either a prismatic, or non-prismatic, object. If the object has been identified as prismatic then the message will appear as:

```
******************************
*** PRISMATIC OBJECT ***
******************************
```

otherwise, the message will be as follows:

```
*************************************
*** NON-PRISMATIC OBJECT ***
*************************************
```

Either message is then followed by another indicating that the analysis step has been completed, and that the solid modelling step has started. A prompt requesting you to enter the type of terminal you are using, is displayed. A list of the different terminal supported by the solid modeller software, can be obtained by typing the on-line help command 'H'. For instance, typing 3000 would indicate that the workstation is an Apollo DN3000 with colour monitor, and that the whole screen would be used for graphics display; whereas, typing 3001, would indicate that the same workstation is used, but only the present window, and not the whole screen, would be used for graphics display.

The whole screen (or window) is then cleared to display the orthographic projections of the generated model. This is then followed by a message informing you that the solid modelling stage has now been completed, and that the next step, which consists of extracting the orthographic views data of the output model, has started. The screen is cleared again to display either two or three sets of orthographic projections. If no differences have been found to exist between the input views and the projections of the output model, such as in the case of prismatic and ortho-prismatic objects, then only the input views and the orthographic projections of the output model are displayed. In this case, the program displays the following message:

```
**************************************************
*******  SUCCESSFUL CONVERSION  *******
*** EXACT OBJECT IS RECONSTRUCTED ***
**************************************************
```

However, if discrepancies have been found to exist between these two sets of views, then the pseudo-views are also displayed, highlighting those differences. You are thus provided with a facility for checking and examining the differences between the input views and the corresponding projections of the output model. The program then displays the following prompt:

```
The output model is NOT an exact solution.
Do you wish to continue (RETURN = 'YES') ?
```

Hit RETURN if you decide that the generated model requires more refinement. You may, on the other hand, decide that the output model is accurate enough for the application you have in mind, in which case you should hit any other key to exit from the CIEDSM program, hence terminating the interpretation process.

In the former case, the program displays the following message:

```
************************************
*** ITERATION REQUESTED ***
************************************
**** FEED BACK STARTED ****
************************************
```

which indicates that the orthographic projections of one or more subobjects are being retrieved from the pseudo-views. The analysis process is then repeated and the orthographic views of a new output model are displayed. The process continues in this fashion until the output model is identified to be the exact object, or until you decide to terminate the process.

The C.I.E.D.S.M. software has now been described, and a set of operating instructions have been given. These are simple and easy to follow as they are self-explanatory. The software has been tested using a number of practical examples; These examples have been chosen to illustrate the implementation of the interpretation process for prismatic and non-prismatic objects, and are described in the next chapter.

CHAPTER EIGHT

PRACTICAL APPLICATIONS OF THE DEVELOPED PROCESS

8.1) INTRODUCTION:

A number of practical examples have been selected to illustrate the interpretation process algorithms developed in this project, and their implementation to the different classes of objects. These have been chosen to illustrate the range of objects that may successfully be processed, thereby enabling the reader to complement his understanding of the scope and nature of the process. Furthermore, the differences between prismatic (simple and complex), ortho-prismatic and more general 3D objects, are highlighted by selecting objects which slightly differ from one example to another. For instance, the object chosen to represent simple prismatic objects, is transformed into a complex one by drilling holes through it.

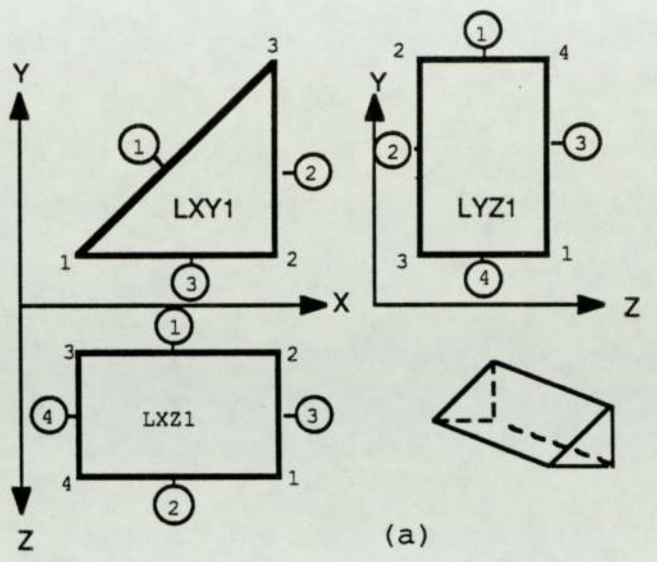
The first example illustrates the reconstruction process of an object which is itself composed of a single primitive. Although trivial, the example not only serves its purpose as an introduction, but it also demonstrates that the implementation has been so designed that such simple objects may be identified, and reconstructed without the need for the full process to be followed in a formal manner.

The second and third examples illustrate to simple and complex prismatic objects, respectively. Whilst example 4 treats what has previously been described as an ortho-prismatic object: it shows that in such cases, the reconstruction is exact and did not require any iterations. Example 5 corresponds to cases where the notion of 'approximation models' and iteration to an 'adequate' model arise.

8.2) EXAMPLE 1: A PRIMITIVE OBJECT:

Figure 8.1(a) shows a set of three views which represent the first angle projections of an object, and Figure 8.1(b) the corresponding topology. The process of converting these views into a solid model is initiated by the 'Loop Detector' algorithm (section 6.2.1) whose function is to determine the number and type of loops in each view. Each loop is then processed by the 'Pattern Identifier' algorithm which determines its geometric shape. The results of applying both algorithms to all the views are summarized in Figure 8.1(c), which indicates that:

a) each view is composed of a single loop,
b) the 2D pattern of the loop in the xy view has been identified as a right-angle triangle,
c) the 2D pattern of the loops in the xz and yz views have been identified as rectangles.



TOPOLOGY											
XY view				XZ view				YZ view			
EN	TY	SN	NE	EN	TY	SN	NE	EN	TY	SN	NE
1	0	1	3	1	0	2	3	1	0	2	4
2	0	2	3	2	0	1	4	2	0	3	2
3	0	1	2	3	0	1	2	3	0	1	4
				4	0	4	3	4	0	3	1

EN = Edge Number TY = Edge Type SN & NE = Start & End Nodes

(b)

View	Loop No	Edge Nos.	
XY	LXY1	3, 2, -1	D
		1, -2, -3	
XZ	LXZ1	1, -4, -2, 3	D
		-1, -3, 2, 4	
YZ	LYZ1	4, 3, -1, -2	D
		1, -3, -4, 2	

D = Disjoint loop

(c)

Fig. 8.1: a) Orthographic views of a primitive object
b) topology
c) number and type of loops

According to the 'Class Identifier' algorithm, this set of patterns forms the signature of a 3D primitive (section 6.2.3). The primitive is identified as a wedge. A solid model is then immediately reconstructed by generating the solid modelling input file which, in this trivial case, consists of the following statements:

```
XY01 <- WEDGE (4.0, 3.0, 5.0)
PRIM <- XY01
```

where XY01 and PRIM are names that are automatically given to the 3D primitive, and the final object, by the software, respectively.

The computation of the wedge length, WL, height, WH, and width, WW, is illustrated in Figure 8.2. In this example, WL is equal to 4.0, WH to 3.0 and WW to 5.0. The centroid of the object is automatically positioned at the origin of the solid modeller coordinate system, shown in Figure 8.2 as the set of OXYZ axes.

The solid modeller reconstructs the 3D model PRIM and generates a parametric ASCII file from which the orthographic views of the model are extracted. These data are then compared with the input data by the 'Comparison' algorithm. Because the algorithm is independent of the labelling of nodes, or edges, the two sets of views are found to be the same, as shown in Figure 8.3, although the

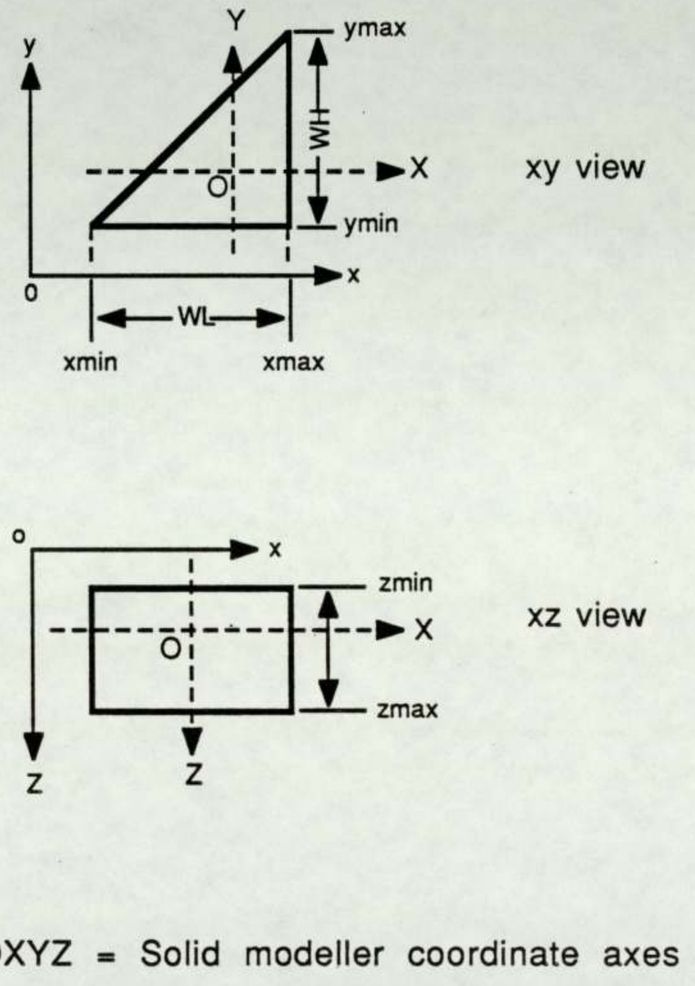
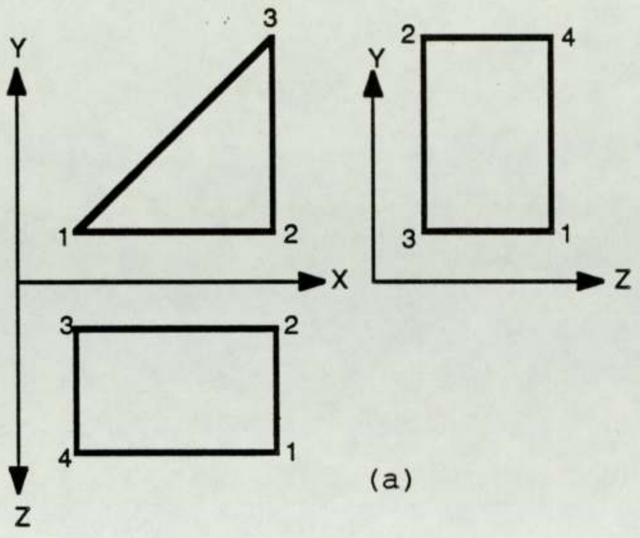


Fig. 8.2: Computation of WEDGE primitive parameters



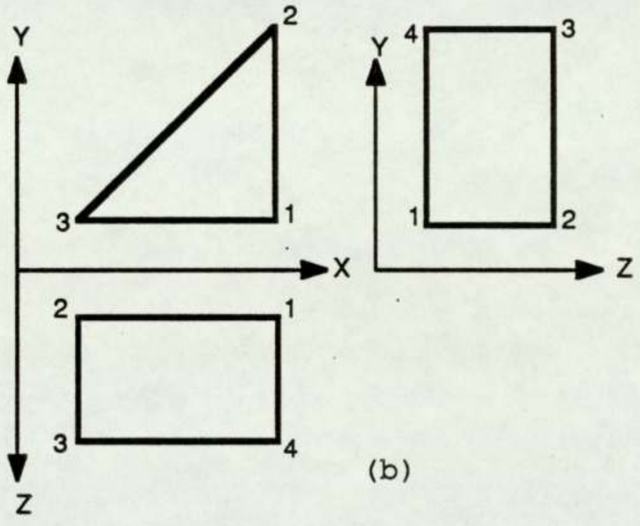


Fig. 8.3: a) Original input views
b) Orthog_aphic views of output model

nodes and edge numbers in the input views do not correspond to the node and edge numbers of the corresponding view of the output model. The complete match between the input and output views confirms that the generated model is the exact object.

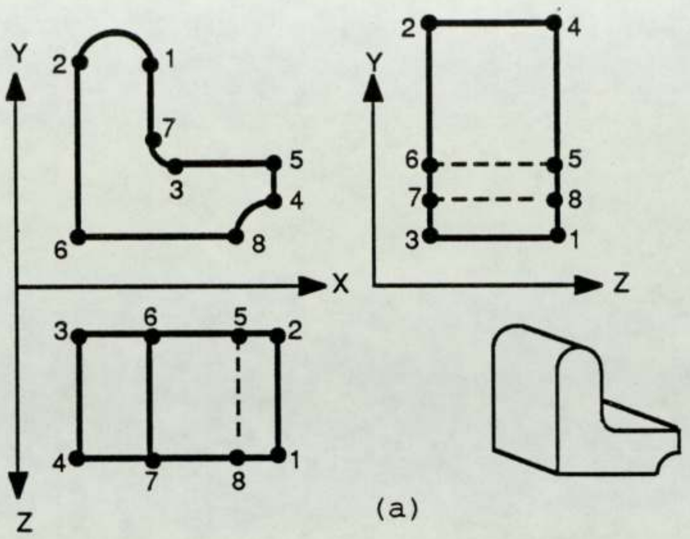
8.3) EXAMPLE 2: A SIMPLE PRISMATIC OBJECT:

Figure 8.4(a) shows the three orthographic views of an object, and Figure 8.4(b) the corresponding topological data. The results of the search for the number and type of loops in each view, Figure 8.4(c), shows that each edge is traversed twice, in opposite senses, thus clearly illustrating the execution of the 'Loops Detector' algorithm; these results indicate that:

a) there is a view, in this case the XY view, which may be identified as a base-view since it comprises only one loop
b) all the loops in the remaining views (XZ and YZ views) are rectangles.

In this case, the object is identified as a simple prismatic object, since, in addition to the above conditions (a) and (b), all the nodes in each of the remaining views belong to the perimeter loop.

The interpretation of such an object consists of



(a)

XY view				XZ view				YZ view			
EN	TY	SN	NE	EN	TY	SN	NE	EN	TY	SN	NE
1	0	6	8	1	0	7	8	1	0	8	5
2	0	5	4	2	0	6	5	2	0	5	6
3	-1	1	2	3	0	6	7	3	0	2	4
4	0	6	2	4	0	3	4	4	0	6	7
5	1	3	7	5	0	8	5	5	0	1	3
6	0	7	1	6	0	2	5	6	0	4	5
7	1	8	4	7	0	3	6	7	0	2	6
8	0	3	5	8	0	7	4	8	0	8	7
				9	0	1	8	9	0	1	8
				10	0	1	2	10	0	3	7

EN = Edge Number TY = Edge Type SN & NE = Start & End Nodes

(b)

View	Loop No	Edge Nos.	Pattern shape	Type
XY	L1	1, 7, -2, -8, 5, 6, 3, -4	arbitrary	P
		-1, 4, -3, -6, -5, 8, 2, -7		
XZ	L1	-2, -7, 4, -8, 1, -9, 10, 6	rectangle	P, C
	L2	-1, -3, 2, -5	rectangle	C
	L3	7, 3, 8, -4	rectangle	C
	L4	5, -6, -10, 9	rectangle	C
YZ	L1	-5, 9, 1, -6, -3, 7, 4, -10	rectangle	P, C
	L2	-1, 8, -4, -2	rectangle	C
	L3	2, -7, 3, 6	rectangle	C
	L4	5, 10, -8, -9	rectangle	C

P = Perimeter loop C = Connected loop

(c)

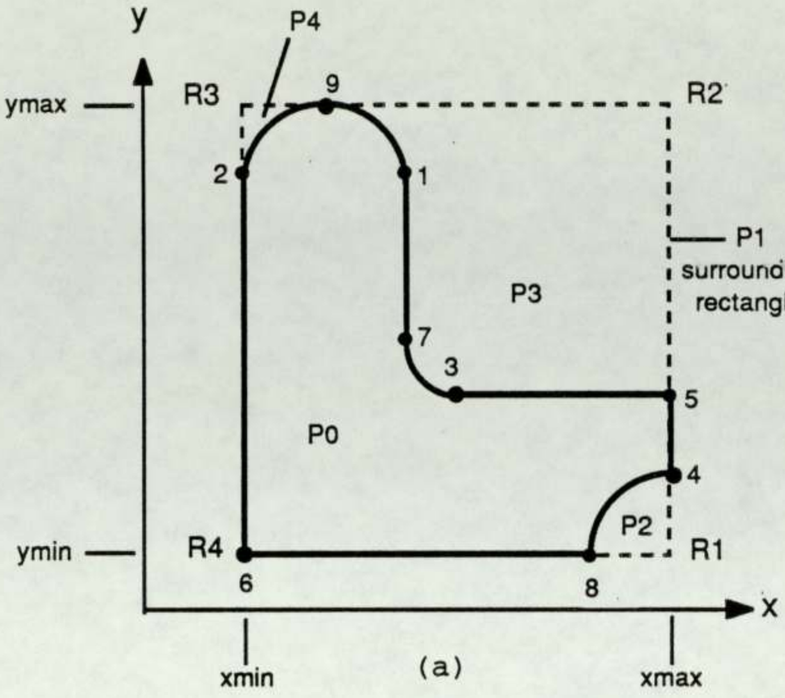
Fig. 8.4: a) Orthographic views of a simple prismatic object
b) topological data
c) Number and type of loops

processing the single loop contained in the base-view, i.e., the perimeter loop in the xy view. The first step of such a process consists of computing the extreme coordinate values of the object loop in order to define its surrounding rectangle. These values are determined by the 'Extreme Coordinate Search' algorithm (section 6.1.5). A point with such an extreme value is found to lie on the circular arc {1,2}, Figure 8.5(a). This point is added to the xy view, as node 9, by splitting the arc {1,2} into two smaller arcs {1,9}, and {9,2}. The coordinates of the surrounding rectangle are then as follows:

```
xr(1) = xmax        yr(1) = ymin
xr(2) = xmax        yr(2) = ymax
xr(3) = xmin        yr(3) = ymax
xr(4) = xmin        yr(4) = ymin
```

The next step consists of generating the 'control list' (section 6.2.6). The control list for such an object loop, shown in Figure 8.5(b), is then used by the 'Primitive Loop Locator' algorithm the function of which is to locate the loops obtained by the intersection of the object loop and its surrounding rectangle. These 'primitive' loops are shown in Figure 8.5(a), as loops P2, P3, and P4, located between the object loop P0 and its surrounding rectangle (loop P1).

The data for each primitive loop are then examined



A	B	C
8	1	1
4	2	1
5	2	1
9	3	1
2	4	1
6	4	0

(b)

Fig. 8.5: a) Object loop and surrounding rectangle
b) the corresponding control list'

by the 'Loop Identifier' algorithm to identify the geometric shape of its pattern, and label each pattern with a flag, LPF, according to its shape. The flags and patterns of all the primitive loops have been found to be as follows:

Loop P1 = basic pattern => LPF = 1 or 'rectangle'

Loop P2 = basic pattern => LPF = 4 or 'quadrant'

Loop P3 = arbitrary pattern => LPF = 0

Loop P4 = basic pattern => LPF = 3 or 'fillet'

The data for all the loops, and their corresponding flags are stored in the MAINDATA file. After scanning by the 'Arbitrary Pattern Analyser' algorithm, the MAINDATA file is found to comprise a loop, P3, that has an arbitrary pattern which is decomposed further into loops P5 (its surrounding rectangle) and P6.

Again, loop P6 is identified as an arbitrary pattern. Furthermore, it is found to possess the characteristics of an 'unstable' loop (section 6.1.7). Such characteristics prevent loop P6 from being directly decomposed into further patterns, and hence, is divided into three further loops (section 6.1.8), shown in Figure 8.6 as loops P7, P8 and P9. The data of these loops are then stored back into the MAINDATA file, and all the loops are identified as basic patterns, shown in Figure 8.6 as loops P1, P2, P4, P5, P7, P8 and P9.

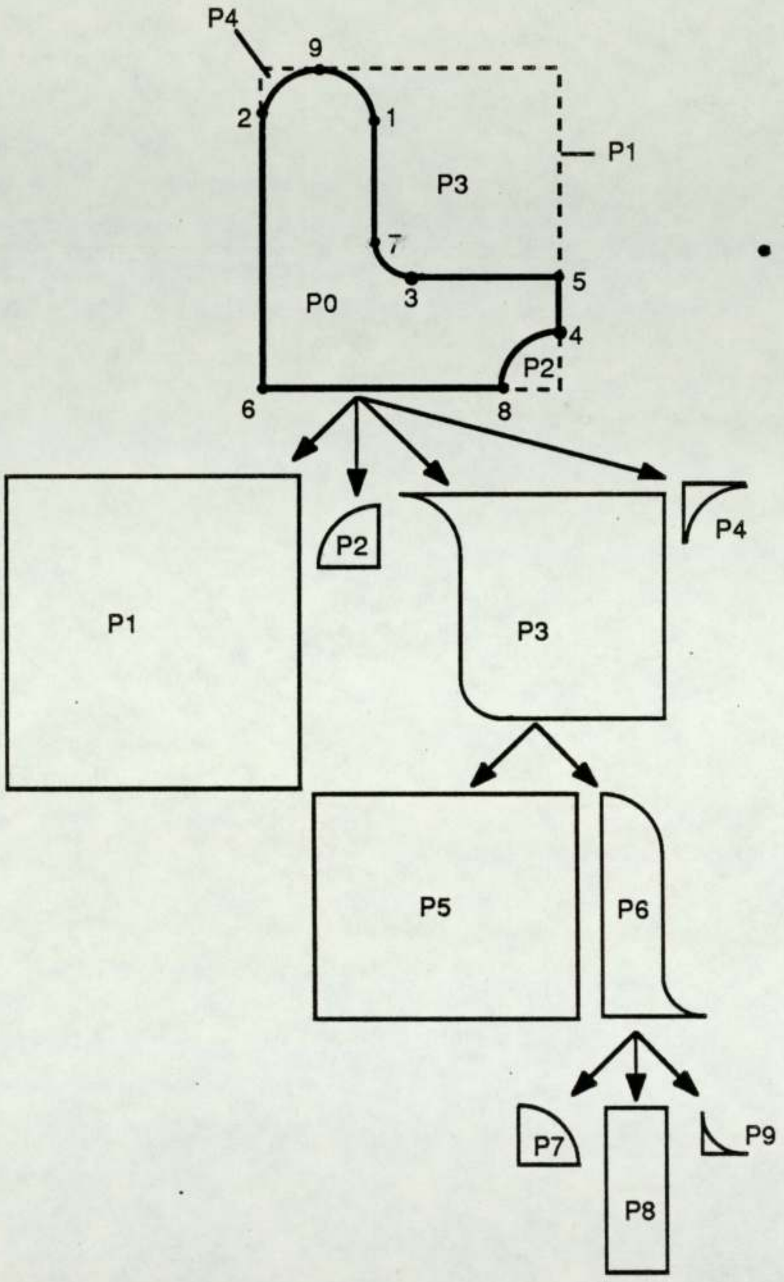


Fig. 8.6: Decomposition tree of an object loop into basic patterns

Each of the basic patterns identified above represents the xy view of a 3D primitive. The remaining views of each primitive are composed of rectangles. Each set of three patterns represents the signature of a 3D primitive, as shown in Figure 8.7(a). For example, loop P1 has been identified as a rectangle which, together with a rectangle in each of the remaining views, form the signature of a primitive block.

The solid modeller input file is then generated and consists of the following statements:

```
XY01 <- BLOCK (5.0, 5.0, 4.0)
XY02 <- CYL (4.0, 2.0) AT (2.5, -2.5, -2.0 )
XY04 <- FILLET (1.0, 1.0, 4.0) AT (-2.5, 2.5, -2.0)
XY05 <- BLOCK (4.0, 3.0, 4.0) AT (0.5, 1.5, 0.0)
XY07 <- CYL (1.0, 1.0, 4.0) AT (-1.5, -1.5, -2.0)
XY08 <- BLOCK (1.5, 3.0, 4.0) AT (-0.75, 1.0, 0.0)
XY09 <- FILLET (0.5, 0.5, 4.0) AT (-0.5, 1.5, -2.0)
XY06 <- XYO7 - XY08 - XY09
XY03 <- XY05 - XY06
FAMOD <- XY01 - XY02 - XY03
```

Each primitive is automatically given a name. The object XY06 is obtained by subtracting primitives XY08 and XY0.9 from primitive XYO7; object XY03 is obtained by removing object XY06 from primitive XY05 and, finally, the output model, called FAMOD, is then produced by

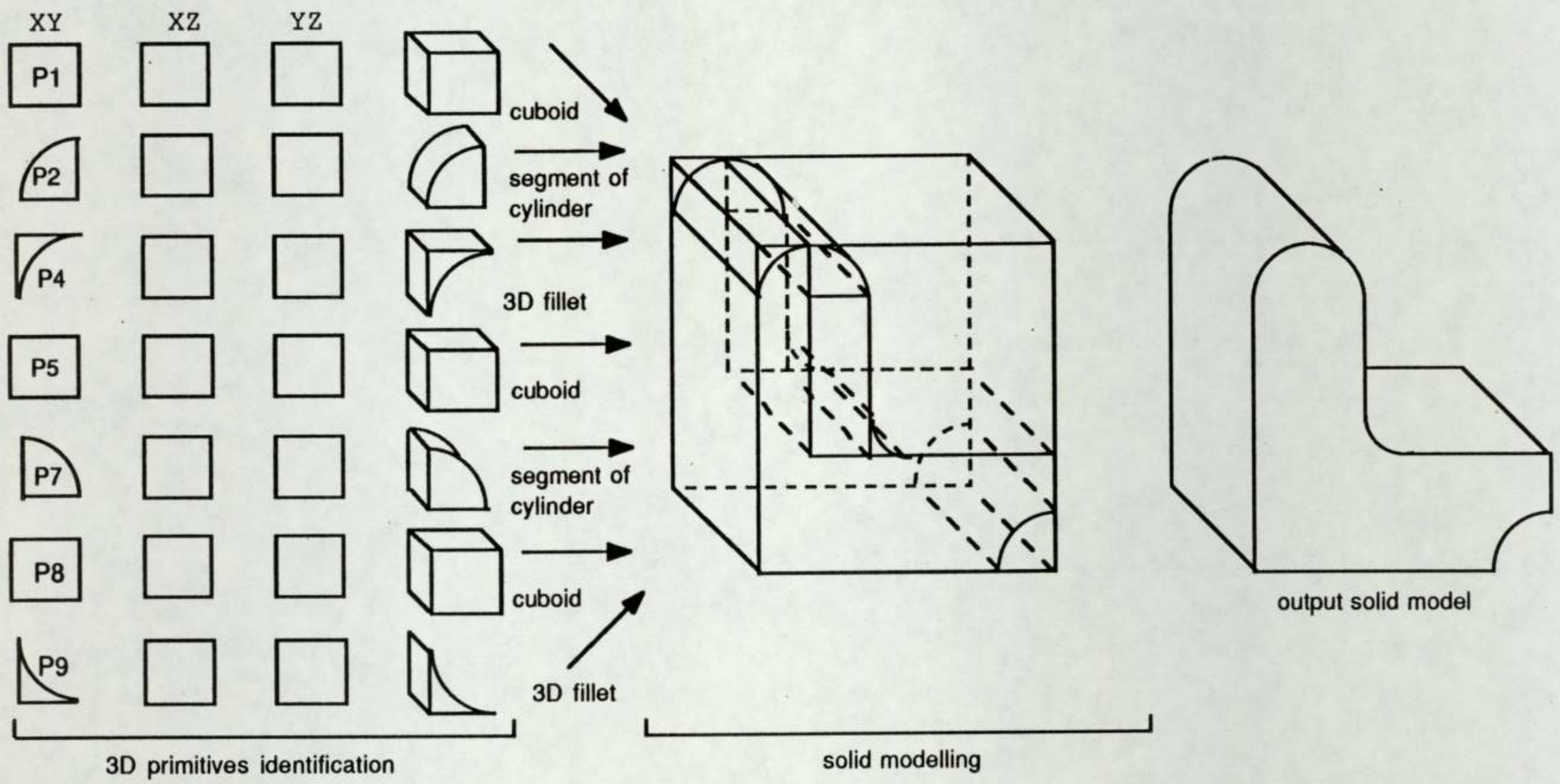
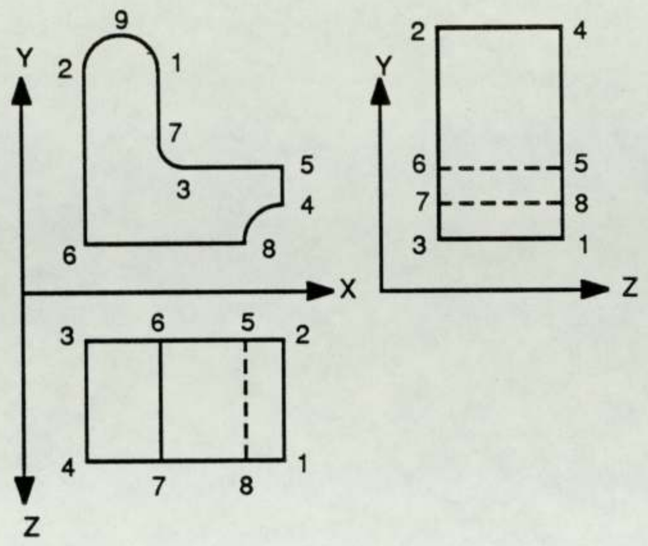


Fig. 8.7: 3D primitives identification and generation of solid model

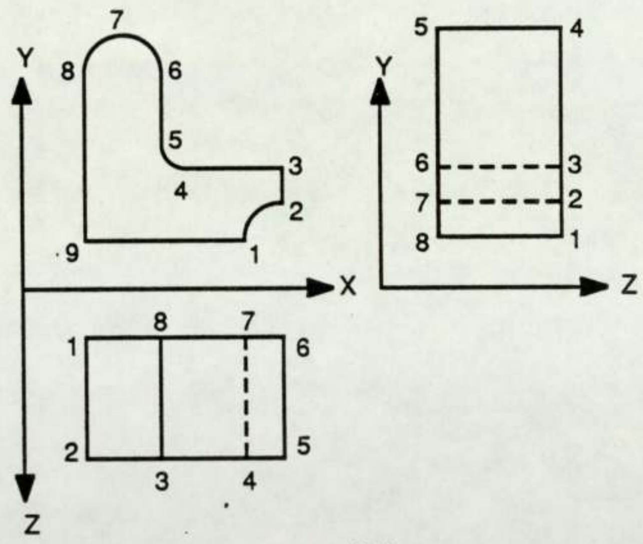
subtracting object XY03 and primitive XY02 from the surrounding block represented by the primitive XYO1. Clearly, the generation of output models consists of a gradual removal, not addition, of objects from the raw block.

The solid modeller combines the objects and primitives defined in the solid modeller input file to generate the output model, Figure 8.7(b). A text file, which comprises the parametric description of the orthographic views of the output model, is also generated. This file is then scanned and the data representing the orthographic views of the output model are retrieved. The minimum node coordinate values of the output model views are computed and then subtracted from the minimum node coordinate values of the corresponding input views, in order to calculate the amount of shift required to adjust the node coordinates in all the views of the output model.

The original input views are finally compared to the corresponding orthographic projections of the output model, FAMOD, by the 'Comparison' algorithm, and are found to be exactly the same, as shown in Figure 8.8; the generated model thus corresponds exactly to the object whose views comprised the input to the interpretation process.



(a)



(b)

Fig. 8.8: a) Input orthographic views
b) Output orthographic views

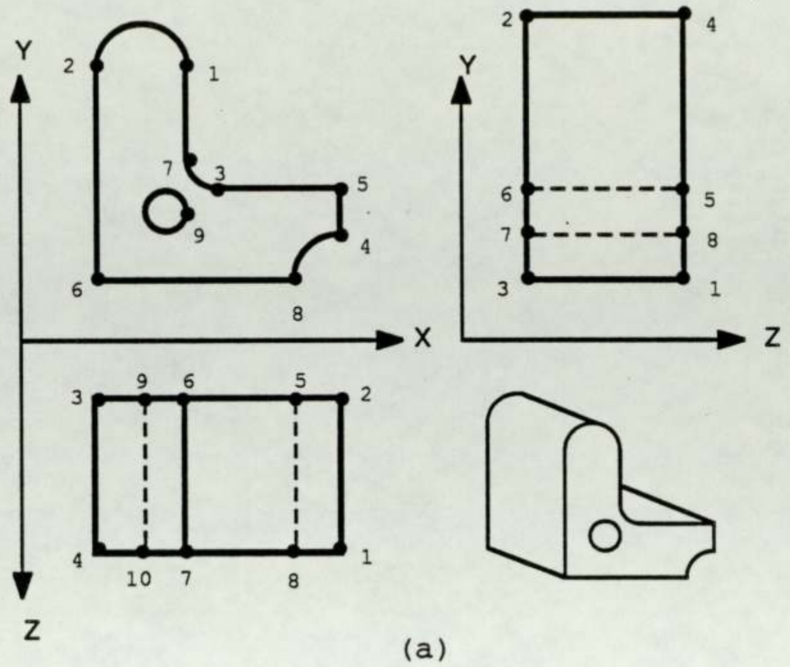
8.4) EXAMPLE 3: A COMPLEX PRISMATIC OBJECT:

The object in the previous example is transformed into a complex prismatic one by drilling a hole through it, so that the modified orthographic views are those in Figure 8.9(a). The topological data is shown in Figure 8.9(b). Those views are immediately identified, by the 'Class Identifier' algorithm (section 6.1.3), to be the orthographic projections of a complex prismatic object since, as shown in Figure 8.9(c) :

a) there is a view, in this case the xy view, which consists of two disjoint loops.
b) Each of the two remaining views comprises a number of connected rectangular loops only. Furthermore, all the nodes in these views belong to the perimiter loop.

The interpretation of process, now consists of not just treating a perimeter loop in a base-view, but all the loops in that view. In this example, loops P01 and P02, in the xy view are processed. Loop P01 is decomposed into the primitive loops P1, P2, P4, P5, P7, P8 and P9, as described in the previous example. Loop P02 is identified as a circle (basic pattern), which does not require any further decomposition. The resulting tree is shown in Figure 8.10.

The 3D primitives, associated with each basic



(a)

XY view				XZ view				YZ view			
EN	TY	SN	NE	EN	TY	SN	NE	EN	TY	SN	NE
1	0	6	8	1	0	7	8	1	0	8	5
2	0	5	4	2	0	6	5	2	0	5	6
3	-1	1	2	3	0	6	7	3	0	2	4
4	0	6	2	4	0	3	4	4	0	6	7
5	1	3	7	5	0	8	5	5	0	1	3
6	0	7	1	6	0	2	5	6	0	4	5
7	1	8	4	7	0	3	9	7	0	2	6
8	0	3	5	8	0	7	10	8	0	8	7
9	1	9	9	9	0	1	8	9	0	1	8
				10	0	1	2	10	0	3	7
				11	0	10	4				
				12	0	9	6				

EN = Edge Number TY = Edge Type SN & NE = Start & End Nodes

(b)

Fig. 8.9: a) Orthographic views of a complex prismatic object
b) topology

View	Loop No	Edge Nos.	Pattern shape	Type
XY	L1	1, 7, -2, -8, 5, 6, 3, -4	arbitrary	P, D
		-1, 4, -3, -6, -5, 8, 2, -7		
	L2	9	circle	D
		-9		
XZ	L1	-2, -12, -7, 4, -11, -8, 1, -9, 10, 6	rectangle	P, C
	L2	-1, -3, 2, -5	rectangle	C
	L3	3, 8, 13, 12	rectangle	C
	L4	-4, 7, -13, 11	rectangle	C
	L5	5, -6, -10, 9	rectangle	C
YZ	L1	-5, 9, 1, -6, -3, 7, 4, -10	rectangle	P, C
	L2	-1, 8, -4, -2	rectangle	C
	L3	2, -7, 3, 6	rectangle	C
	L4	5, 10, -8, -9	rectangle	C

P = Perimeter loop   C = Connected loop   D = Disjoint loop

Fig. 8.9(c): Number and type of loops

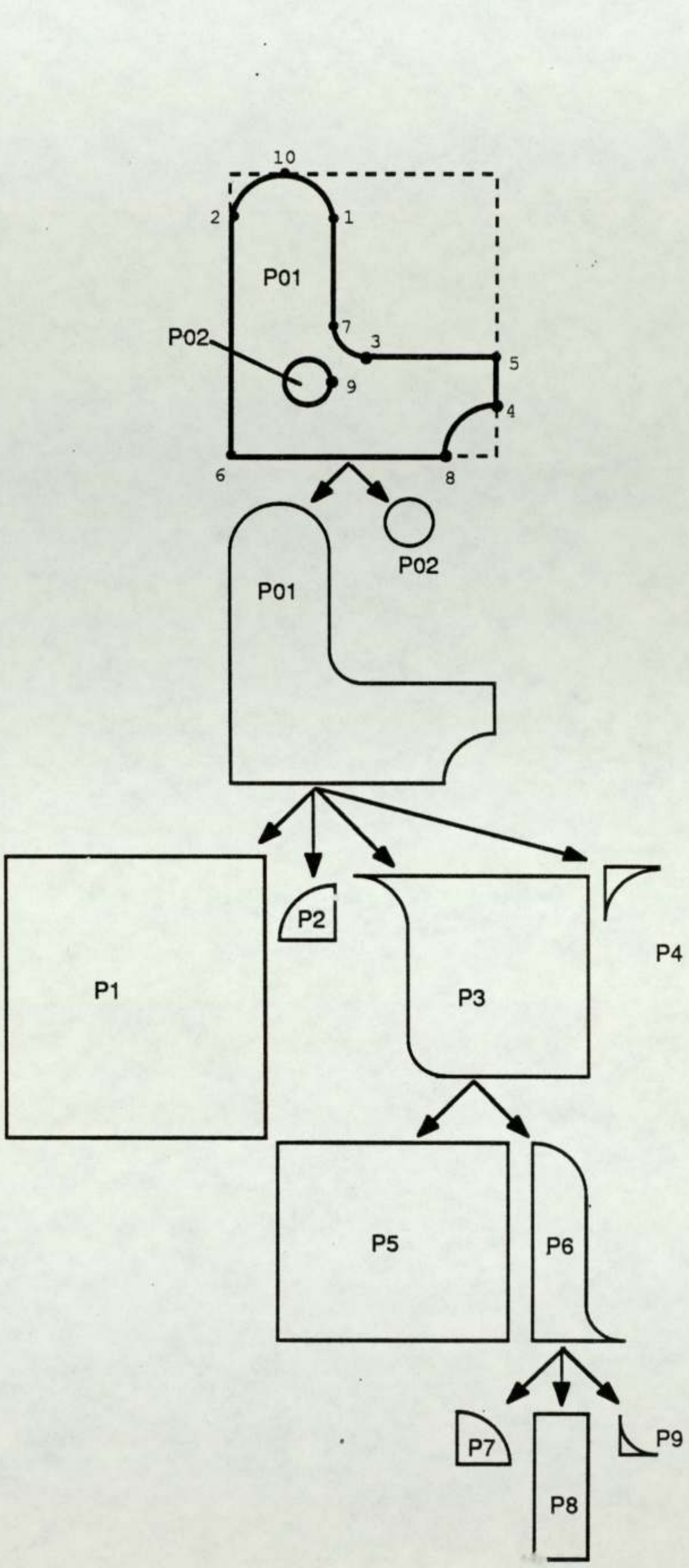


Fig. 8.10: Decomposition tree of object loops in the xy view

pattern, are then identified, Figure 8.11, and the syntax defining all the primitives and the output model, are specified in the solid modeller input file, as follows:

```
XY01 <- BLOCK (5.0, 5.0, 4.0)
XY02 <- CYL (4.0, 2.0) AT (2.5, -2.5, -2.0 )
XY04 <- FILLET (1.0, 1.0, 4.0) AT (-2.5, 2.5, -2.0)
XY05 <- BLOCK (4.0, 3.0, 4.0) AT (0.5, 1.5, 0.0)
XY07 <- CYL (1.0, 1.0, 4.0) AT (-1.5, -1.5, -2.0)
XY08 <- BLOCK (1.5, 3.0, 4.0) AT (-0.75, 1.0, 0.0)
XY09 <- FILLET (0.5, 0.5, 4.0) AT (-0.5, 1.5, -2.0)
XY06 <- XYO7 - XY08 - XY09
XYH01 <- CYL (4.0, 0.25) AT (-0.75, -0.75, -2.0)
XY06 <- XYO7 - XY08 - XY09
XY03 <- XY05 - XY06
FAMOD <- XY01 - XY02 - XY03 - XYH01
```

The file specifies that the output model, FAMOD, is obtained by:

1) generating a number of prismatic objects, XY03 and XY06, which are the result of processing the perimeter loop P1 in the xy view
2) generating a cylindrical object, XYH01, which is the result of processing loop P02
3) generating the output model by subtracting these objects from the surrounding cuboid, specified, in the file, as object XY01.

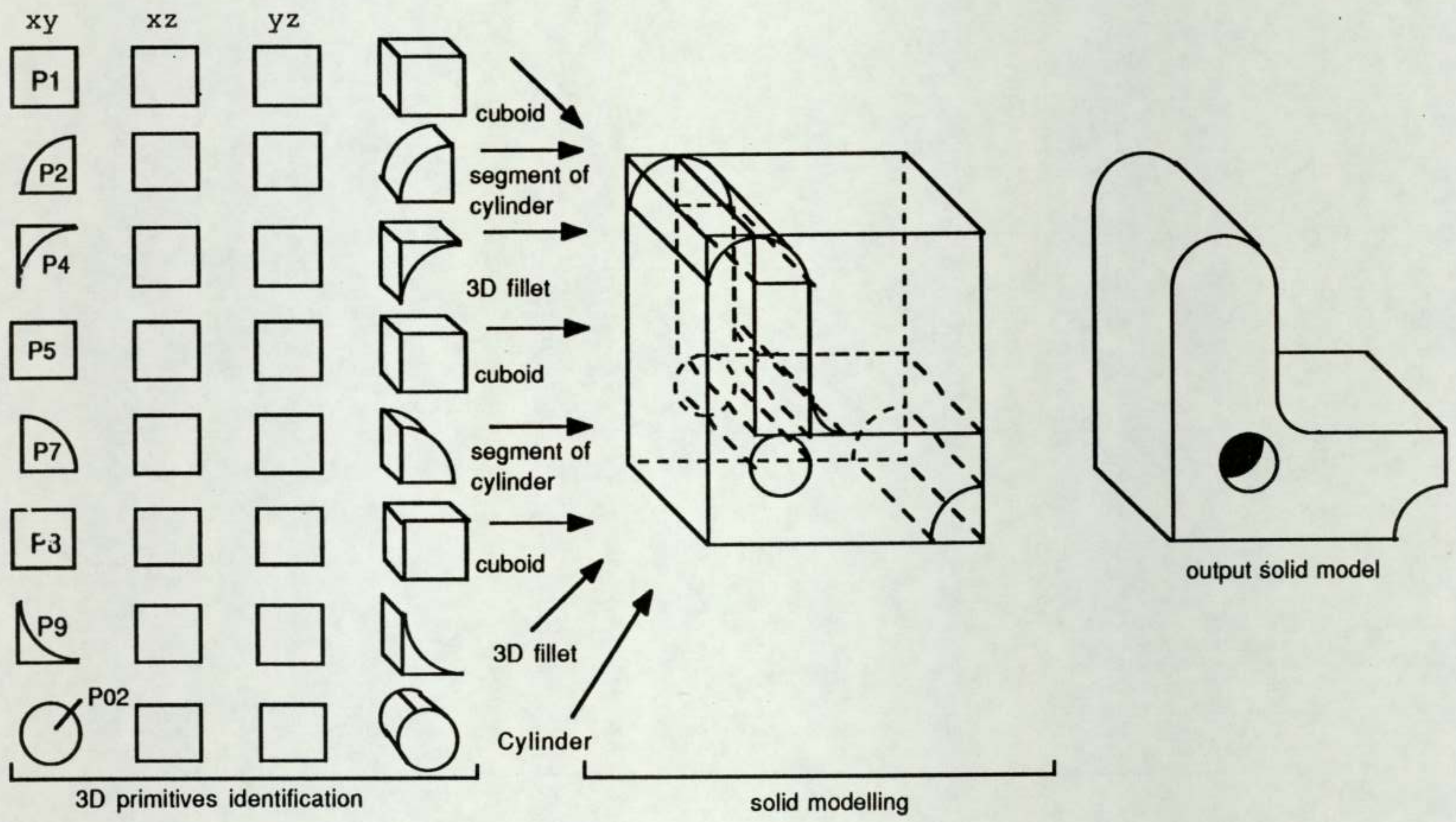


Fig. 8.11: 3D primitives identification and generation of sclid model

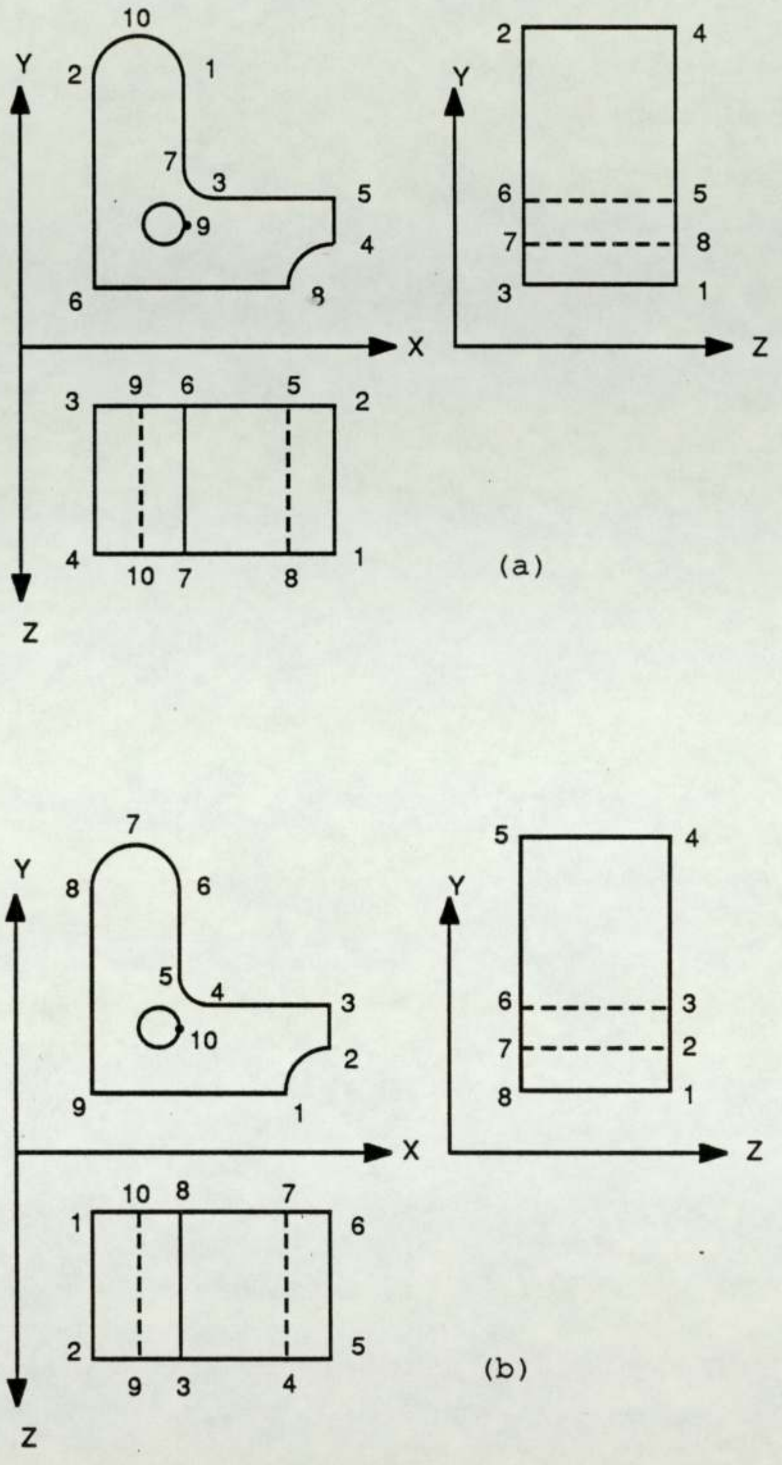


Fig. 8.12: a) Input orthographic views
b) Output orthographic views

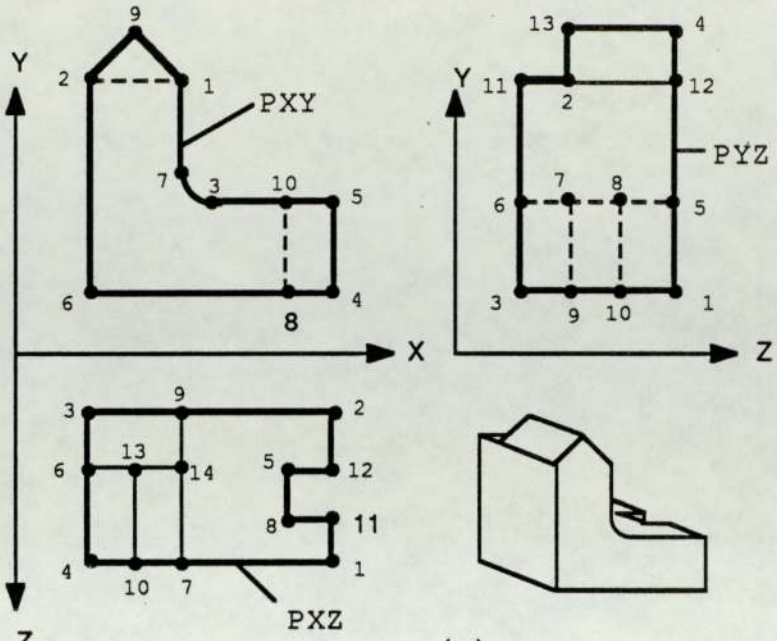
The orthographic views data of the output model are then retrieved from the parametric file generated by the solid modeller 'BOXER', and adjusted in order to be compared with the original input views data, as shown in Figure 8.12. The input views are found to correspond exactly to the orthographic projections of the output model, thus confirming that the exact object has been reconstructed.

8.5) EXAMPLES FOR NON-PRISMATIC OBJECTS:

The two previous examples have shown that exact solutions are obtained for prismatic objects without iteration. In section 5.4.3, a class of objects called ortho-prismatic were introduced. An example illustrating the reconstruction process for such an object is now given and it will be seen that an exact solution will be obtained without iteration. A more general example requiring an iterative solution follows.

8.5.1) EXAMPLE 4: AN ORTHO-PRISMATIC OBJECT:

Figure 8.13(a) shows a set of three views which represent the first angle projections of an object, and the corresponding topological and geometrical data is shown in Figure 8.13(b). These input views are not



(a)

XY view				XZ view				YZ view			
EN	TY	SN	NE	EN	TY	SN	NE	EN	TY	SN	NE
1	0	6	8	1	0	7	10	1	0	8	5
2	0	5	4	2	0	6	13	2	0	3	6
3	0	1	2	3	0	6	4	3	0	2	11
4	0	6	2	4	0	3	6	4	0	6	11
5	1	3	7	5	0	8	5	5	0	1	10
6	0	7	1	6	0	2	12	6	0	4	12
7	0	8	4	7	0	3	9	7	0	7	6
8	0	3	10	8	0	10	4	8	0	8	7
9	0	10	5	9	0	1	7	9	0	10	8
10	0	1	9	10	0	1	11	10	0	3	9
11	0	9	2	11	0	5	12	11	0	7	9
12	0	8	10	12	0	9	2	12	0	2	12
				13	0	13	10	13	0	4	13
				14	0	14	9	14	0	13	2
				15	0	14	7	15	0	5	1
				16	0	14	13	16	0	9	10
				17	0	8	11	17	0	5	12

EN = Edge Number, TY = Edge Type, SN & NE = Start & End Nodes

(b)

Fig. 8.13: a) Orthographic views of an ortho-prismatic object
b) topology

View	Loop No	Edge Nos.	Pattern shape	Type
XY	L1 (PXY)	1, 7, -2, -9, -8 , 5, 6, 10, 11, -4	arbitrary	P, C
	L2	-1, 4, -3, -6, -5, 8, -12	arbitrary	C
	L3	2, -7, 12, 9	rectangle	C
	L4	3, -11, -10	arbitrary	C
XZ	L1 (PXZ)	-8, -1, -9, 10, -17, 5, 11, -6, -12, -7, 4, 3	arbitrary	P, C
	L2	1, -13, -16, 15	rectangle	C
	L3	2, 13, 8, -3	rectangle	C
	L4	-2, -4, 7, -14, 16	rectangle	C
	L5	-5, 17, -10, 9, -15, 14, 12, 6,-11	arbitrary	C
YZ	L1 (PYZ)	10, 16, -5, -15, 17, -6, 13, 14, 3, -4, -2	arbitrary	P, C
	L2	1, 15, 5, 9	rectangle	C
	L3	-1, 8, 7, 4, -3, 12, -17	rectangle	C
	L4	2, -7, 11, -10	rectangle	C
	L5	6, -12, -14, -13	rectangle	C
	L6	-8, -9, -16, -11	rectangle	C

P = Perimeter loop, C = Connected loop

Fig. 8.13(c): Number and type of loops

immediately identified as those of an ortho-prismatic object. Instead, all the views are observed to comprise one, or more, connected loops as shown in Figure 8.13(c); a feature which shows that the object is not prismatic.

The process of constructing a solid model from orthographic views now consists of processing the perimeter loop in each view; in Figure 8.13(a) these are shown in bold lines, and labelled as PXY, PXZ and PYZ, in the xy, xz and yz views, respectively.

The perimeter loop in the xy view, PXY, is decomposed into basic patterns, shown as loops PXY1, PXY3, PXY4, PXY6, PXY7 and PXY8, in Figure 8.14(a). Similarly, the perimeter loop in the xz view, PXZ, is decomposed into loops PXZ and PXZ2, Figure 8.14(b), and for the yz view, PYZ yields loops PYZ1 and PYZ2, Figure 8.14(c).

Each basic pattern, generated from the decomposition of loop PXY, is then used to identify the corresponding 3D primitive, and a prismatic object, previously referred to as the Z-profile (section 5.4.3) may then be generated, as shown in Figure 8.15(a). Similarly, all the 3D primitives associated with all the basic patterns obtained from the decomposition of loops PXZ and PYZ, are identified, and this time, the Y-profile and the X-profile may be generated, as illustrated in Figure 8.15(b), and 8.15(c), respectively.

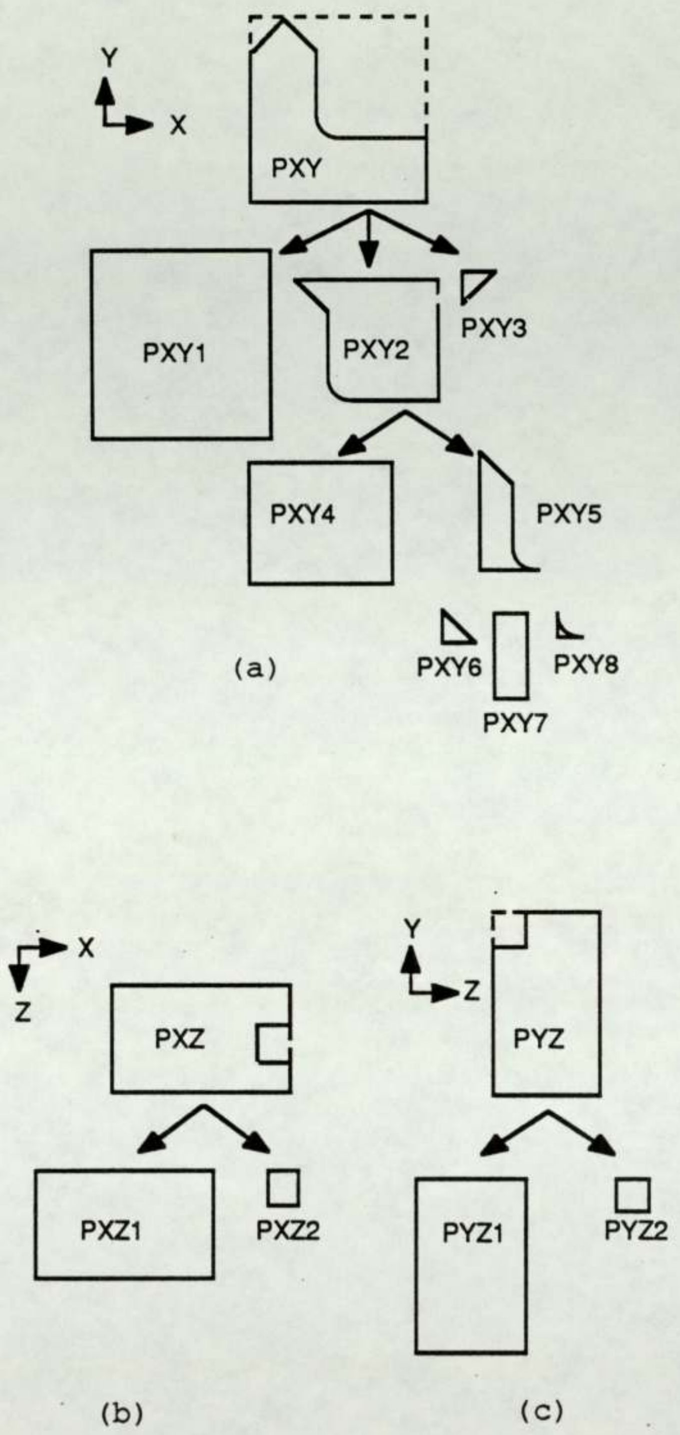


Fig. 8.14: a) Decomposition tree of PXY loop
b) Decomposition tree of PXZ loop
c) Decomposition tree of PYZ loop

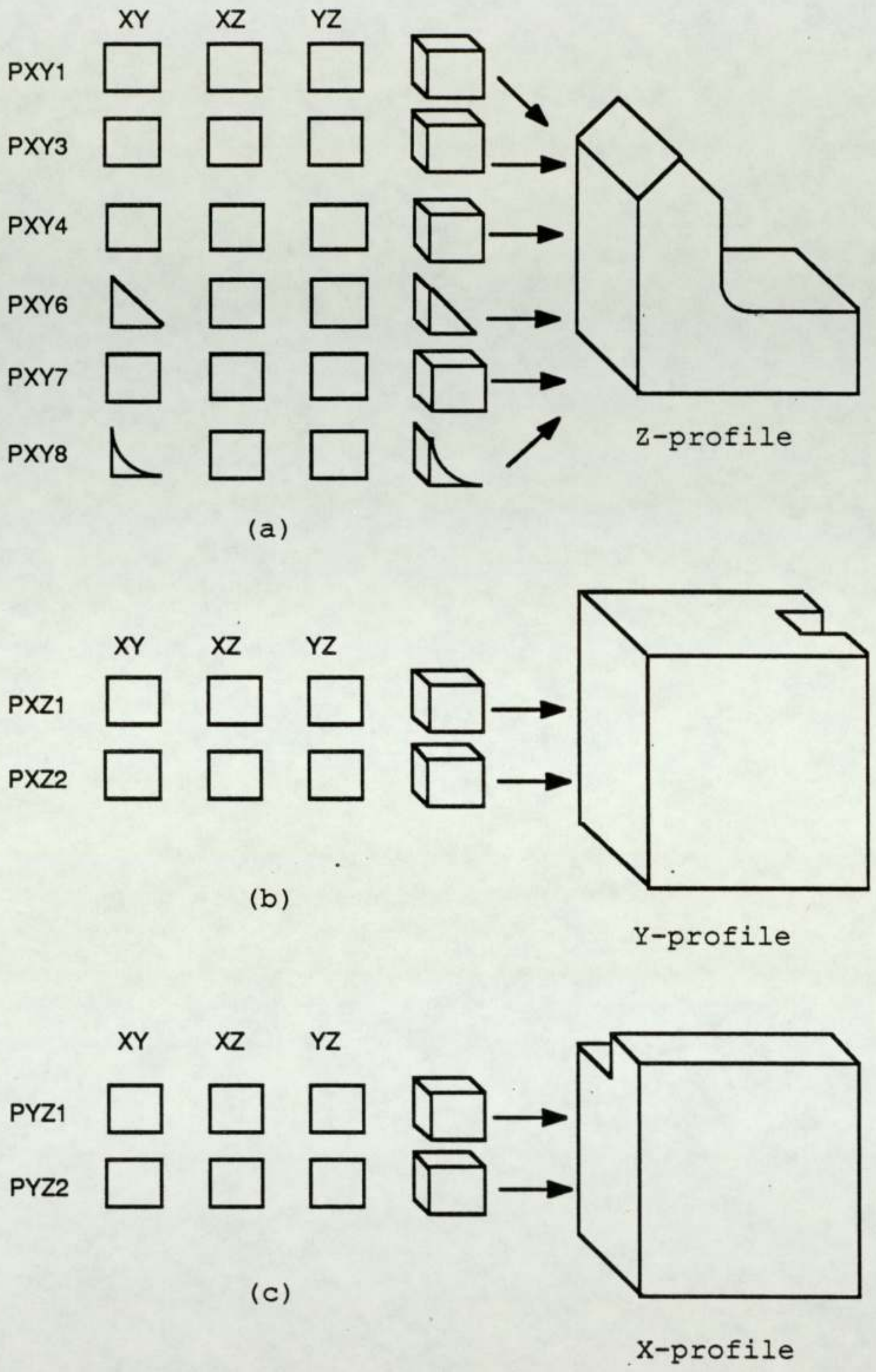


Fig. 8.15: 3D primitives identification and generation of: (a) Z-profile, (b) Y-profile, and (c) X-profile

The solid modeller input file consists of the following statements:

```
XY01 <- BLOCK(8.0, 6.0, 5.0)
XY03 <- WEDGE(1.5, 1.5, 5.0) AT (-4.0, 3.0, -2.5)
XY04 <- BLOCK(6.5, 4.0, 5.0) AT (0.75, 1.0, 0.0)
XY06 <- WEDGE(1.5, 1.5, 5.0) AT (1.5, -2.5, -2.5)
XY07 <- BLOCK(1.5, 2.5, 5.0) AT (-1.75, 0.5, 0.0)
XY08 <- FILLET(1.0, 1.0, 5.0) AT (-1.0, -1.0, -2.5)
XY02 <- XY04 - XY06 - XY07 - XY08
ZPROF <- XY01 - XY02 - XY03
XZ01 <- BLOCK(8.0, 5.0, 6.0)
XZ02 <- BLOCK(2.0, 2.0, 6.0) AT (3.0, 0.0, 0.0)
XZ00 <- XZ01 - XZ02
YPROF <- (XZ00) AT (ROTX = 90.0)
YZ01 <- BLOCK(5.0, 6.0, 8.0)
YZ02 <- BLOCK(2.0, 2.0, 8.0) AT (-1.5, 3.0, 0.0)
YZ00 <- YZ01 - YZ02
XPROF <- (YZ00) AT (ROTY = 90.0)
FAMOD <- ZPROF * YPROF * XPROF
```

The above statements indicate that:

a) the Z-profile is specified by the object named ZPROF, which is defined by suotracting the wedge primitive XY03 and object XY02 from the surrounding block XY01. Object XY02 is defined by subtracting primitives XY06, XY07 and XY08 from the primitive block XY04.

b) the Y-profile is specified by the object called YPROF which is defined by removing cuboid primitive XZ02 from the corresponding surrounding block XZ01, and by rotating it through a 90 degrees angle about the X axis.

c) the X-profile is specified by the object XPROF, defined as the result of subtracting the cuboid primitive YZ02 from the cuboid primitive YZ01, and rotating it through a 90 degrees angle about the Y axis

d) the output model is finally defined as the result of the intersection (*) of the three profiles ZPROF, YPROF and XPROF.

The above file is then used by the solid modeller to construct the solid model, Figure 8.16, according to the specifications described above, and to generate a text file in which the orthographic views of the solid model are parametrically described. The topological and geometrical data corresponding to the output model views are extracted from this parametric file and then compared to the original input data, as shown in Figure 8.17. Again, similarly to the case of prismatic objects, the two sets are found to be exactly identical, thus confirming that the intersection shown in Figure 8.16, i.e., the output model is a complete description of the original object.

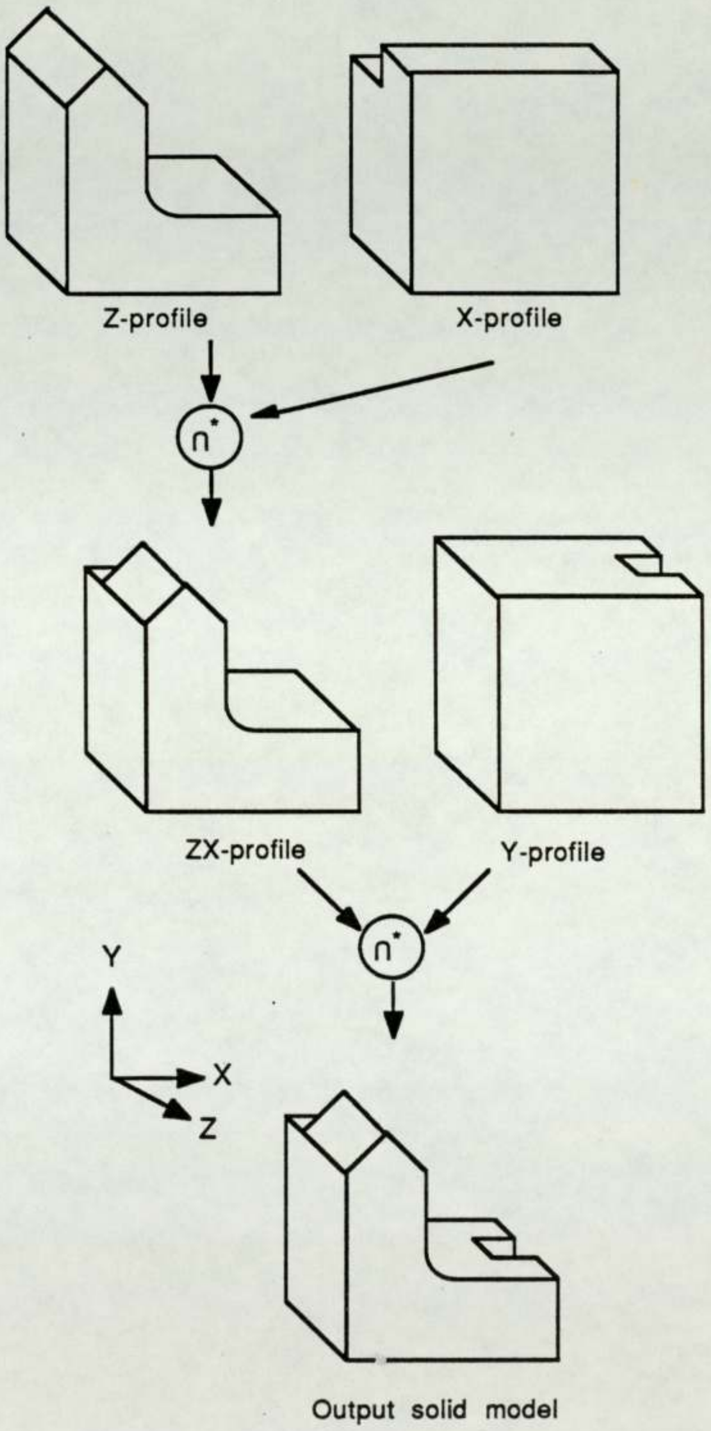
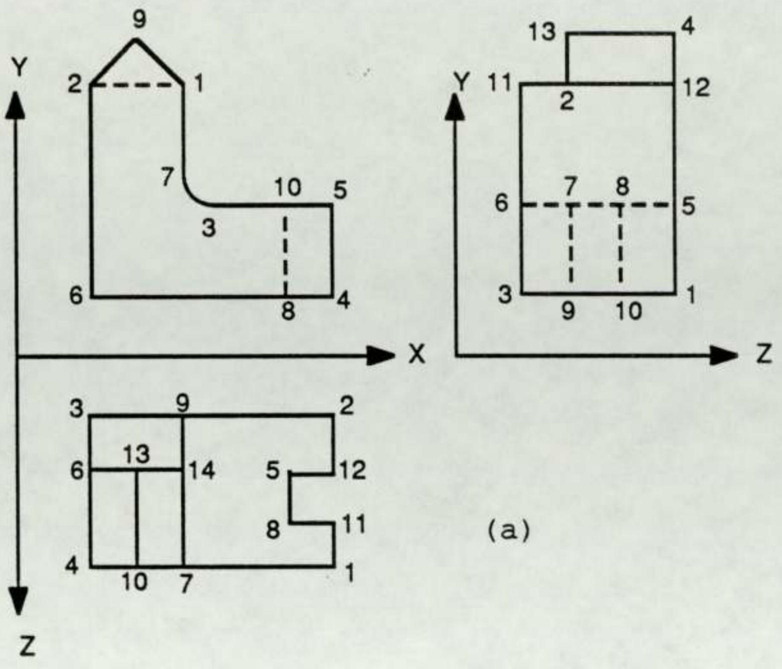


Fig. 8.16: Generation of a solid model from the intersection of the three mutually perpendicular 'profiles'



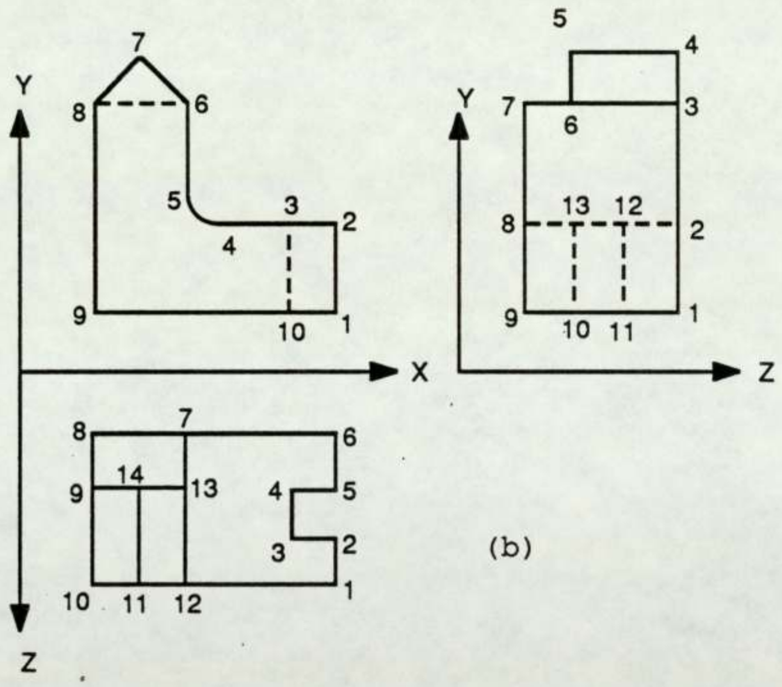


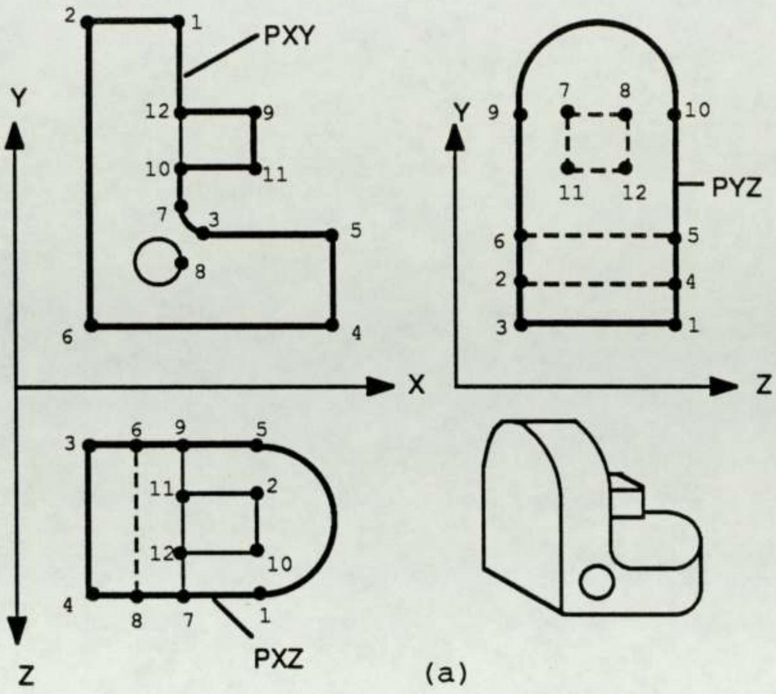
Fig. 8.17: a) Original input views
b) Orthographic views of output model

8.5.2) <u>EXAMPLE 5: A GENERAL 3D OBJECT</u>:

The iterative aspect of the interpretation process developed in this project, is clearly demonstrated by this example. The orthographic views and associated topology, shown in Figure 8.18(a) and 8.18(b), respectively, are identified as those of a non-prismatic object, for the same reasons as the ones described in the previous example. The results of the search for the number and type of loops in all the views are presented in Figure 8.18(c).

Again, only the perimeter loop in each view is processed initially. The process consists of decomposing loops PXY, PXZ and PYZ, shown in Figure 8.18(a) in bold. The results of such decomposition is illustrated in Figure 8.19. The 3D primitives are identified and the corresponding profile in each view is reconstructed as shown in Figure 8.20. The output model is then generated from the intersection of the three profiles, Z-profile, Y-profile and X-profile, as specified by the solid modeller input file which is as follows:

```
XY01 <- BLOCK(8.0, 10.0, 6.0)
XY03 <- BLOCK(5.0, 7.0, 6.0) AT (1.5, .15, 0.0)
XY04 <- BLOCK(2.0, 2.0, 6.0) AT (0.0, 1.0, 0.0)
XY05 <- FILLET(1.0, 1.0, 6.0) AT (-2.0, -1.0, -3.0)
XY02 <- XY03 - XY04 - XY05
ZPROF <- XY01 - XY02
```



(a)

XY view				XZ view				YZ view			
EN	TY	SN	NE	EN	TY	SN	NE	EN	TY	SN	NE
1	0	6	4	1	0	7	12	1	0	6	5
2	0	5	4	2	0	6	9	2	0	3	1
3	0	1	2	3	0	6	3	3	0	2	6
4	0	6	2	4	0	3	4	4	0	6	9
5	1	3	7	5	0	8	4	5	0	1	4
6	0	7	10	6	0	2	11	6	0	4	2
7	0	1	12	7	0	5	9	7	0	7	11
8	0	11	10	8	0	10	2	8	0	8	7
9	0	10	12	9	0	1	7	9	0	12	8
10	1	8	8	10	-1	1	5	10	1	9	10
11	0	9	11	11	0	11	12	11	0	5	10
12	0	9	12	12	0	12	10	12	0	12	11
13	0	3	5	13	0	6	8	13	0	4	5
				14	0	11	9	14	0	3	2
				15	0	8	7				

EN = Edge Number, TY = Edge Type, SN & NE = Start & End Nodes

(b)

Fig. 8.18: a) Orthographic views of a non-prismatic object
b) topology

View	Loop No	Edge Nos.	Pattern shape	Type
XY	L1 (PXY)	1, -2, -13, 5, 6, -8, -11, 12, -7, 3, -4	arbitrary	P, C
	L2	-1,4,-3, 7, 9,-6, -5, 13, 2	arbitrary	C
	L3	8, 9, -12, 11	rectangle	C
	L4	10 -10	circle	D
XZ	L1 (PXZ)	-5, 15, -9, 10, 7, -2, 3, 4	arbitrary	P, C
	L2	1, 12, 8, 6, 14, -7, -10	arbitrary	C
	L3	-1, -15, -8, 2, -14, 11	rectangle	C
	L4	-3, 13, 5, -4	rectangle	C
	L5	-6, -8, -12, -11	rectangle	C
YZ	L1 (PYZ)	2,5,13,11,-10, -4, -3,-14	arbitrary	P, C
	L2	1, -13, 6, 3	rectangle	C
	L3	-1, 4, 10, 11	arbitrary	C
	L4	-2, 14, -6, -5	rectangle	C
	L5	7, -12, 9, 8 -7, -8, -9, 12	rectangle	D

P = Perimeter loop, C = Connected loop
D = Disjoint loop

Fig. 8.18(c): Number and type of loops

```
XZ01 <- BLOCK(8.0, 6.0, 10.0)
XZ02 <- FILLET(3.0, 3.0, 10.0) AT (4.0, -3.0, -5.0)
XZ01 <- FILLET(3.0, 3.0, 10.0) AT (4.0, 3.0, -5.0)
XZ00 <- XZ01 - XZ02 - XZ03
YPROF <- (XZ00) AT (ROTX = 90)
YZ01 <- BLOCK(6.0, 10.0, 8.0)
YZ02 <- FILLET(3.0, 3.0, 8.0) AT (-3.0, 5.0, -4.0)
YZ01 <- FILLET(3.0, 3.0, 8.0) AT (3.0, 5.0, -4.0)
YZ00 <- YZ01 - YZ02 - YZ03
XPROF <- (YZ00) AT (ROTY = 90)
FAMOD <- ZPROF * YPROF * XPROF
```

which indicate that:

a) the Z-profile is specified by the object named ZPROF, which is defined by removing the object XY02 from the surrounding block XY01. Object XY02 is defined by subtracting primitives XY04 and XY05 from the primitive block XY03.

b) the Y-profile is specified by the object called YPROF which is defined by removing two fillet primitives, XZ02 and XZ03, from the corresponding surrounding block XZ01, and by rotating it through a 90 degrees angle about the X axis.

c) the X-profile is specified by the object XPROF, also defined as the result of subtracting two fillet primitives, YZ02 and YZ03 from the cuboid primitive YZ01, and rotating it through a 90 degrees angle about the Y

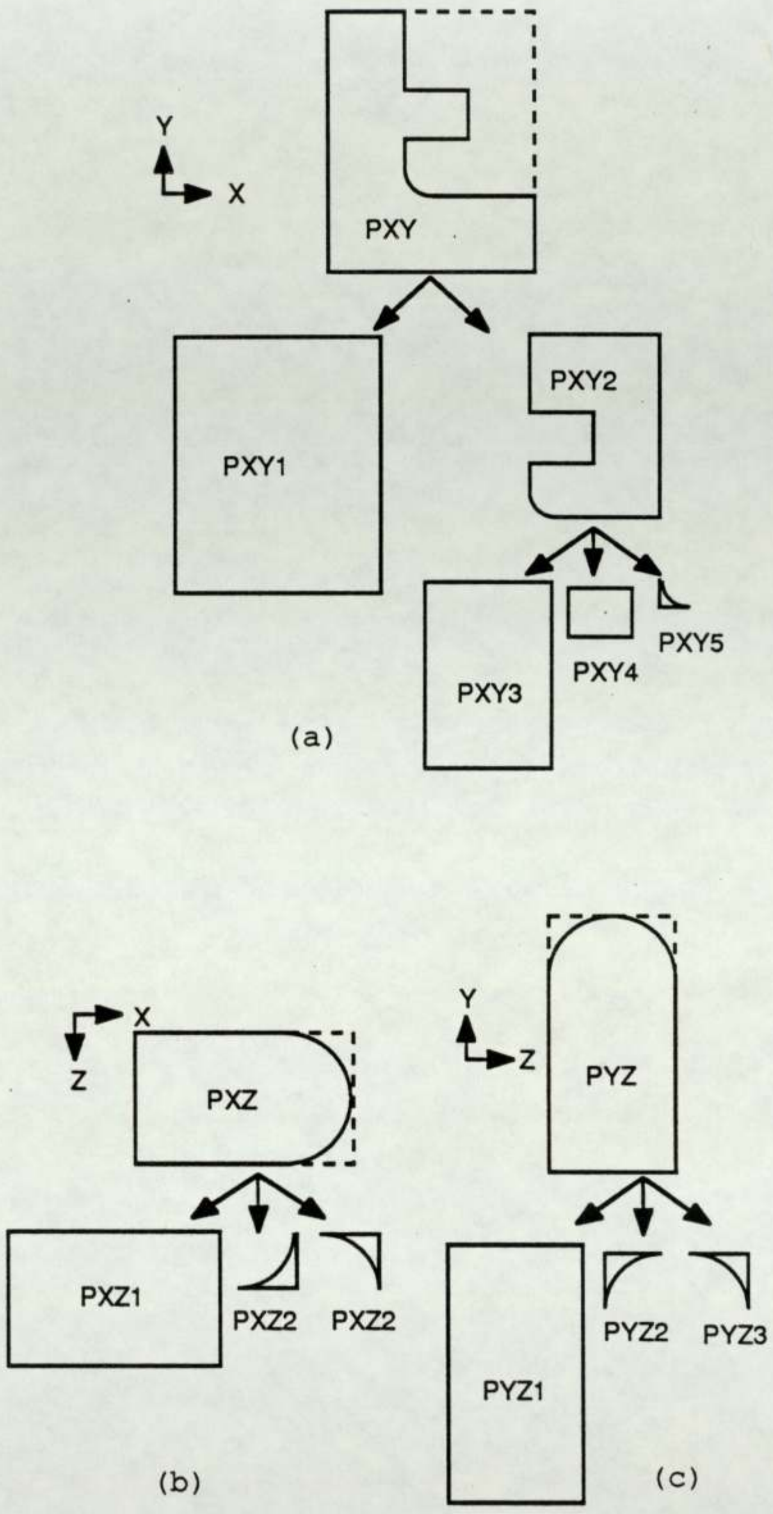
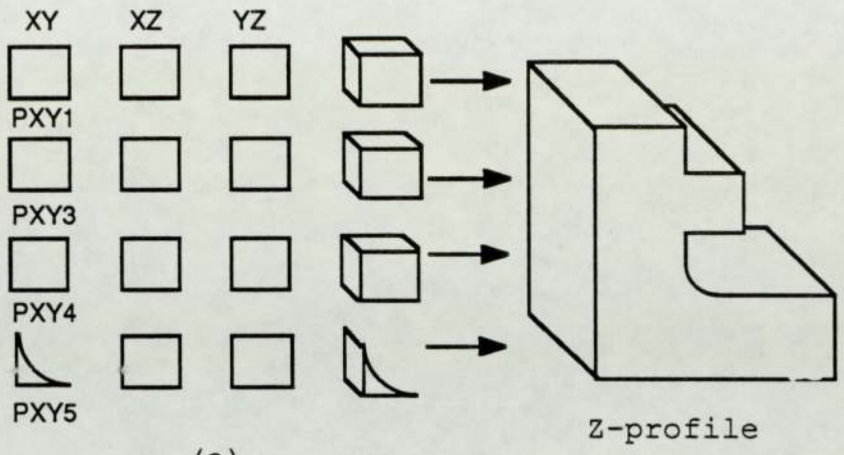
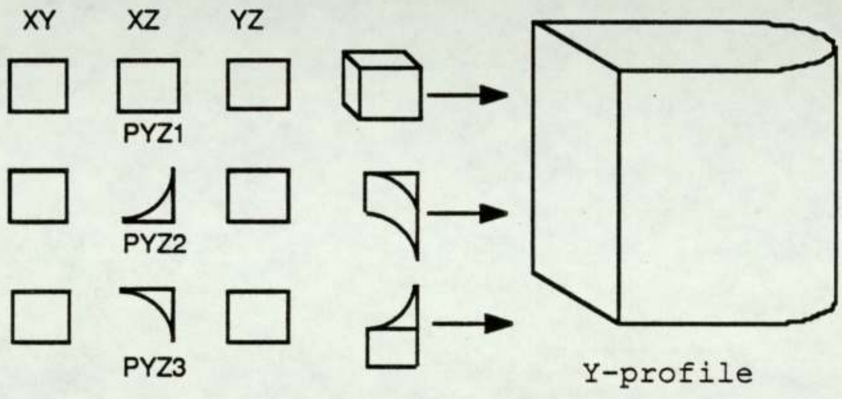


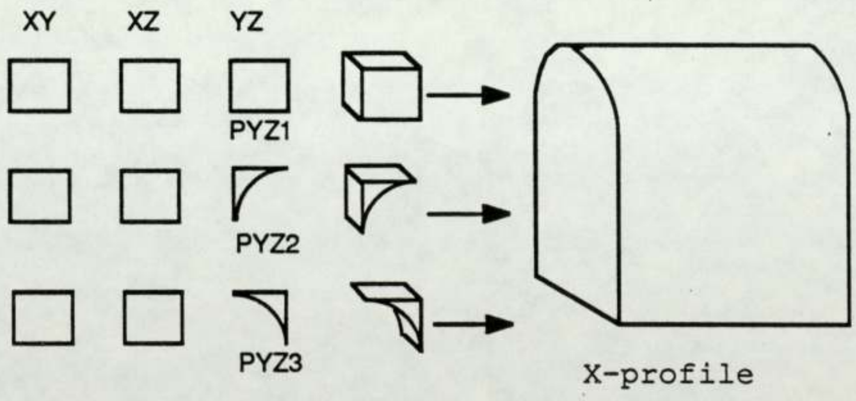
Fig. 8.19: a) Decomposition tree of PXY loop
b) Decomposition tree of PXZ loop
c) Decomposition tree of PYZ loop



(a)



(b)



(c)

Fig. 8.20: 3D primitives identification and generation of: (a) Z-profile, (b) Y-profile, and (c) X-profile

axis

d) the output model is finally defined as the result of the intersection (*) of the three profiles ZPROF, YPROF and XPROF.

The data that represent the orthographic projections of the output model, FAMOD, are then retrieved from the parametric file generated by the solid modeller. In this case, it is clear that a number of differences can be clearly identified to exist between the orthographic views of this intersection, Figure 8.21(a), and the original input orthographic views, Figure 8.21(b). These discrepancies indicate that the output 3D model is not the exact object, but an approximation model, which, in this case is the First-approximation model.

In order to generate either a complete, or a more refined, object model, the differences between the input and output views will have to be subjected to a minimization procedure. Such a procedure consists of identifying a number of subobjects which are to be removed from the output solid model. The procedure is initiated by the generation of the so-called pseudo-views from which the orthographic views of such subobjects may be retrieved. The pseudo-views are generated by the 'Pseudo-views Generator' algorithm, as shown in Figure 8.21(c). Loops are then labelled accordingly in order to identify those which represent orthographic projections of

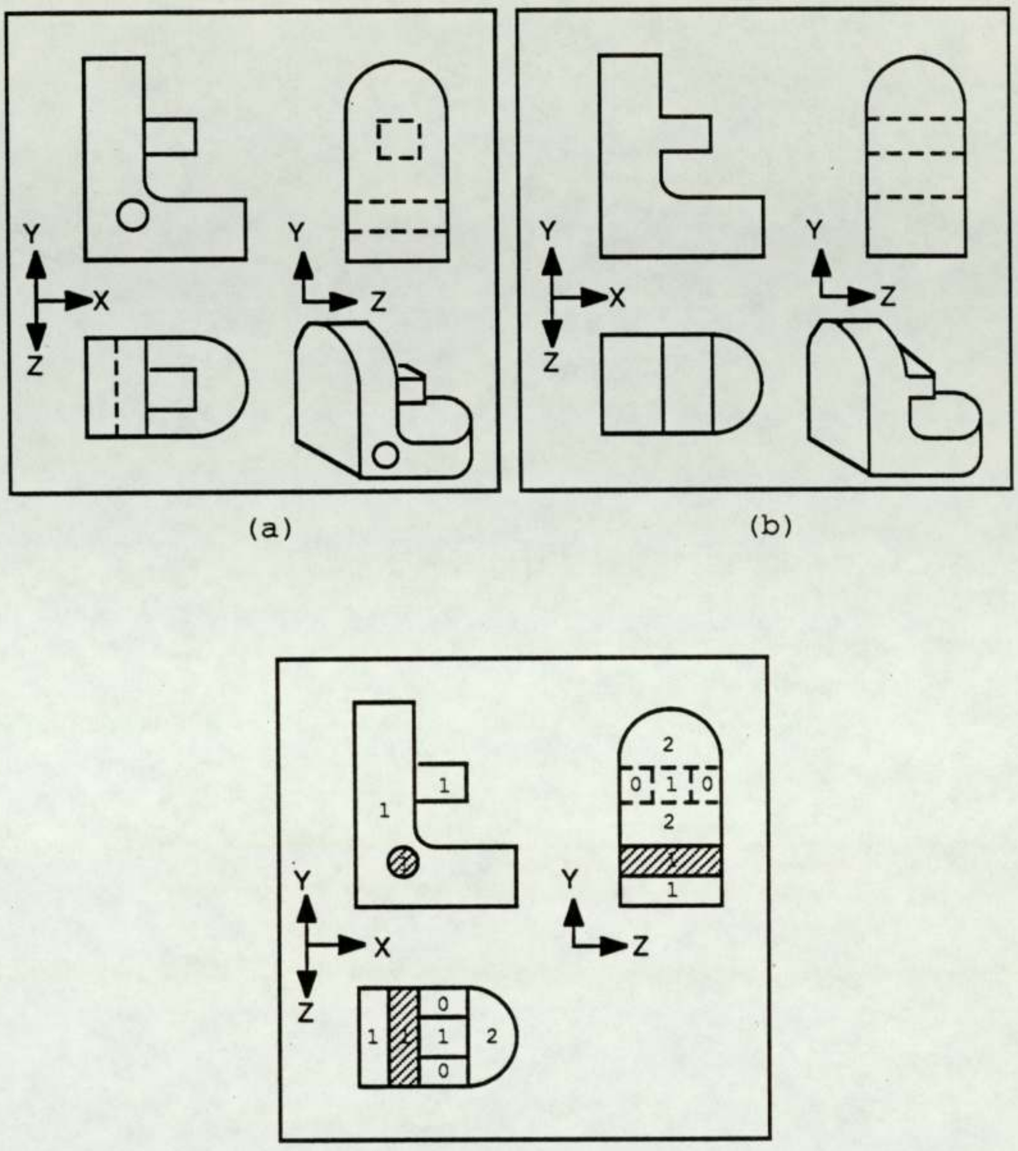


Fig. 8.21: a) Original input views and object
b) 1st approximation model and corresponding orthographic views
c) pseudo-views

subobjects. According to the 'Feed Back Data Generator' algorithm (section 6.4.2), the loops shown in Figure 8.21(c) as hatched areas, form a set of 'matching' loops which represent the orthographic projections of a subobject. The data for these loops are then fed back into the analysis process which identifies them as the signature of a cylindrical 3D primitive. The cylindrical subobject is then subtracted from the First-approximation model to yield a second approximation model, Figure 8.22, having views which are still different from the original input views, as shown in Figure 8.23.

The process of generating pseudo-views and identifying orthographic views of subobjects is again repeated; this results in the identification of another set of matching loops, shown in figure 8.23, again as hatched areas. The data associated with these loops is then used as input to the analysis step to be interpreted as two cuboids. Next, the cuboids are removed from the second-approximation model, Figure 8.24, to yield a new solid model which, this time, is identified as the exact object. This is confirmed by comparing the orthographic views of such model with the original input views, as shown in Figure 8.25. For demonstration purposes, the exact object is shown in this example, to be reconstructed from its orthographic views after two iterations, but it is actually reconstructed only after one iteration, since the identification of all the subobjects, together with

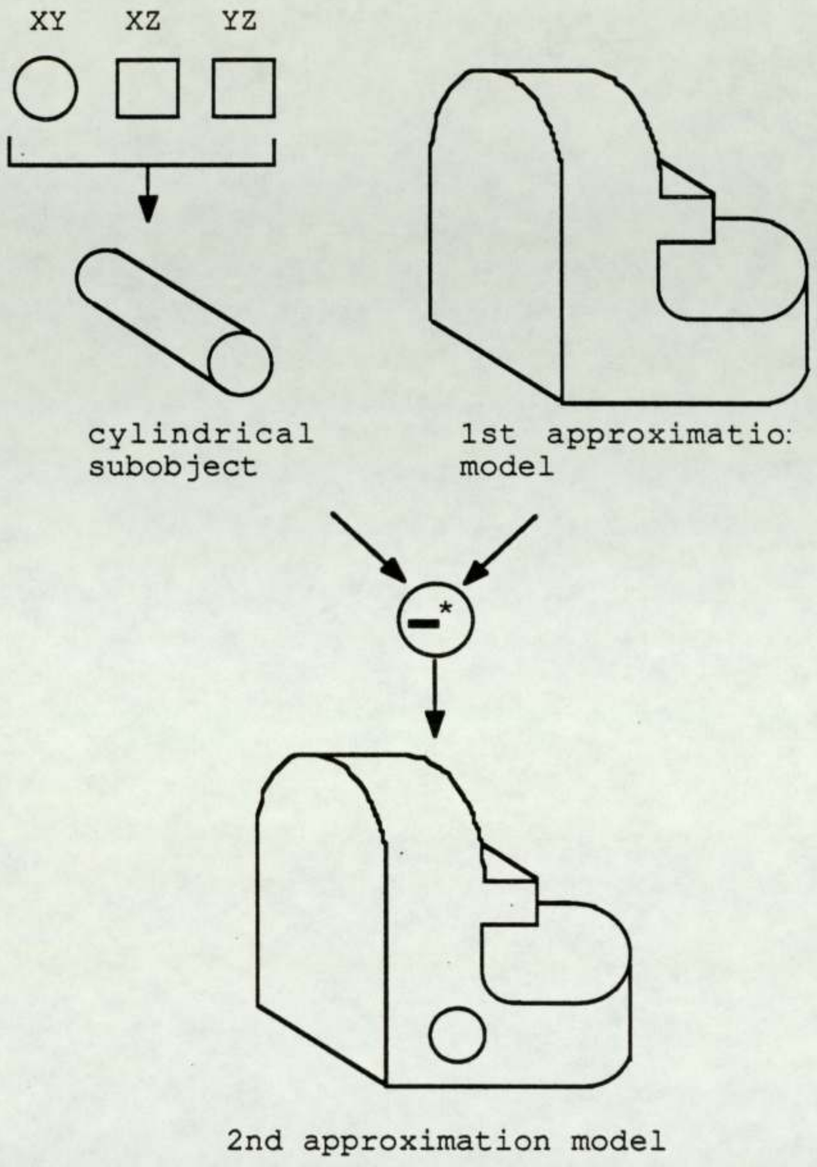
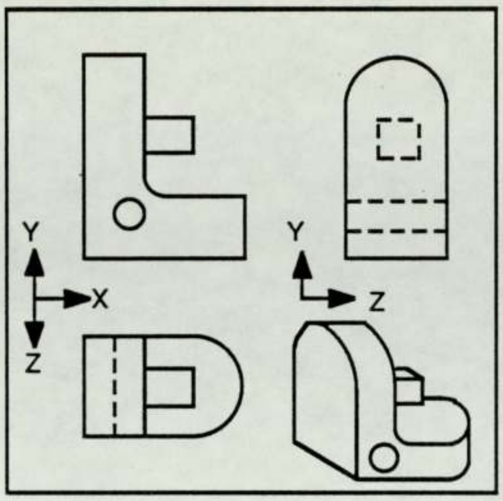
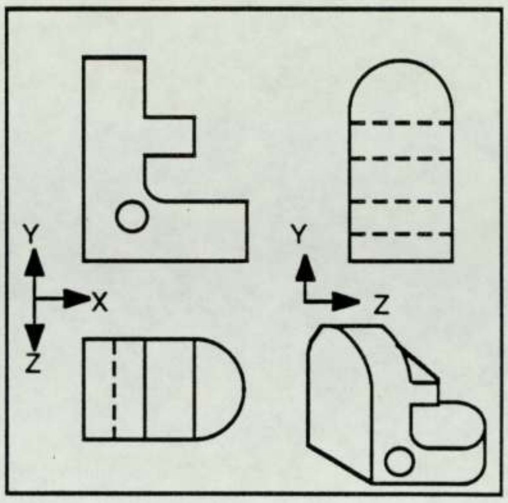


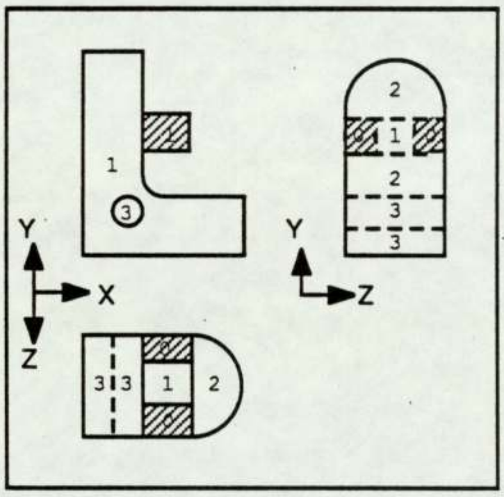
Fig. 8.22: Subtracting an identified cylindrical subobject from the 1st approximation model to generate the 2nd approximation model



(a)



(b)



(c)

Fig. 8.23: a) Original input views and object
b) 2nd approximation model and corresponding orthographic views
c) pseudo-views

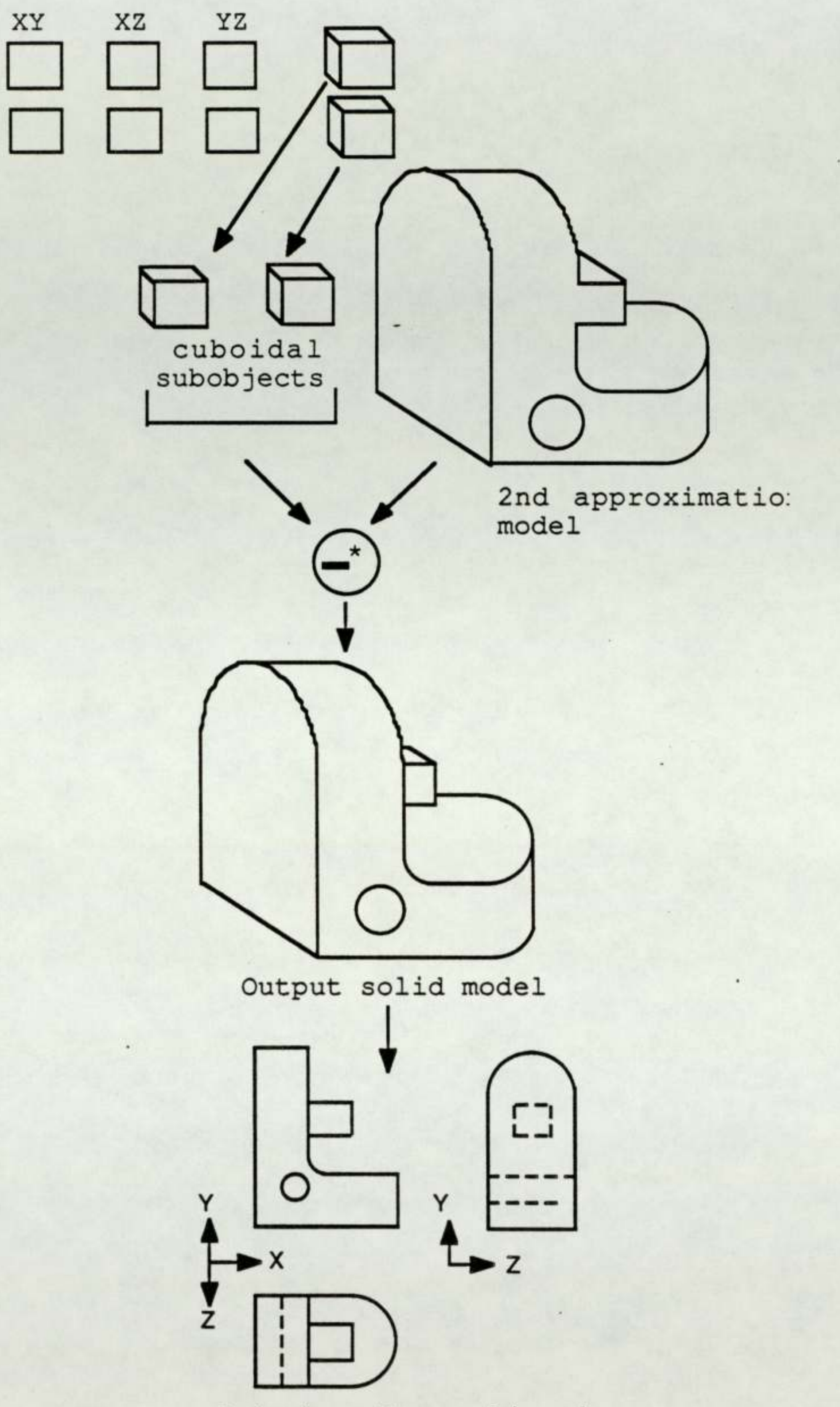
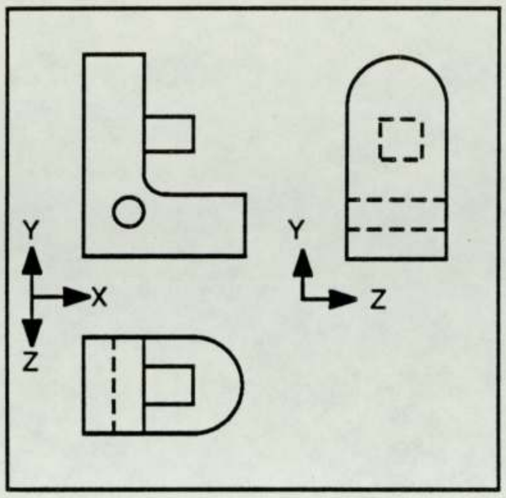
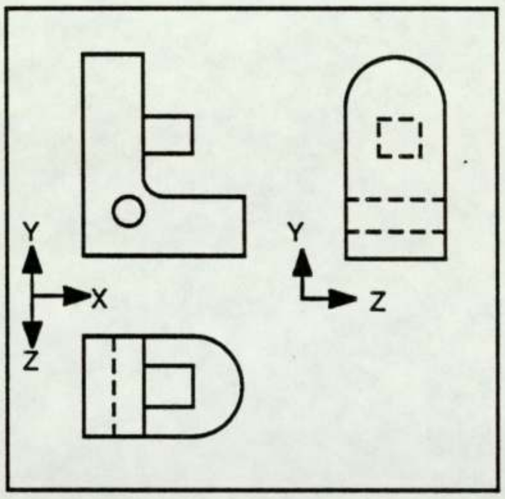


Fig. 8.24: Subtracting two identified cuboidal subobjects from the 2nd approximation model to generate the exact object



(a)



(b)

Fig. 8.25: a) Original input views
b) Orthographic views of generate
solid model

their removal from the first approximation model, is actually carried out in one step.

The above examples clearly illustrate the interpretation process developed in this project. They also demonstrate that the process of generating output models consists always of a removal, and not addition, of primitive objects from the initial surrounding cuboid or 'Raw Block'.

CHAPTER NINE

DISCUSSION AND SUGGESTIONS FOR FUTURE WORK

9.1) DISCUSSION:

The problem of converting engineering drawings into solid models is still regarded as a very complex one, mainly because of the vastness of the domain of mechanical engineering components. This project has been undertaken to contribute to the solution of the problem. To this end, a number of algorithms have been developed and implemented on an Apollo DN3000 workstation. An interpretation process which converts orthographic projections into solid models has been developed by adopting a novel approach based on the concepts of Constructive Solid Geometry.

The problem of interpreting engineering drawings as solid objects has been a topic of research for many years and various techniques have been developed in attempts to solve it. However, none of these techniques is yet known, to the author, to have been implemented for the whole domain of mechanical engineering objects. Drawing from this experience, the author has adopted, from the start of the project, the basic philosophy of minimizing the formidable complexity of the problem by using the 'Divide and Conquer' approach. The domain of objects was initially divided into two different classes: prismatic and non-prismatic objects, and work was concentrated on the development of an interpretation process which converts orthographic projections of prismatic objects, the simplest of the two classes, into solid models. Experience gained from the work on prismatic objects has yielded a technique which extends the interpretation process to a wide range of objects which may be represented by the techniques of Constructive Solid Geometry.

The algorithms described above accept data which must represent three views of a solid, and each view may consist of straight lines and circular arcs only, but this is not a severe limitation to many of the mechanical engineering applications. Because of the 'Divide and Conquer' approach, the addition of facilities to accept more complex geometry is seen as an evolutionary process rather than one requiring major changes to the philosophy upon which the software is based. For instance, one such facility is the addition of further primitives such as spheres, cones and toroids, together with the removal of any restriction on the orientation of primitives.

The main problem is that the true shape of a primitive, whose axes are not aligned with the coordinate axes, may not be readily identified from the "global" orthographic views of the solid object. Such a problem, however, does not arise during the generation of the "first-approximation" model since it only requires the analysis of the boundary loop in each view. Furthermore, as mentioned at the beginning of this thesis, it has been reported that the PADL-1 development team at Rochester University [30], found that 40 percent of parts designed by a range of Mechanical Engineering companies could be represented in terms of just two primitives: rectangular blocks and circular cylinders - subject to the restriction that block edge and cylinder axes were aligned with the coordinate axes. According to this encouraging report, the domain of objects that may be interpreted by the process that has been developed in this project, can be regarded as fairly large. The report also mentions that the addition of further primitives such as those mentioned above, together with the removal of any restriction on the orientation of primitives allows the modelling of more than 90 percent of the parts from the same companies.

9.2) <u>SUGGESTIONS FOR FUTURE WORK</u>:

To quote an appropriate comment by Cooley [50]: "It is an unfortunate but inescapable fact of software development that, when the fundamental problems have been largely solved, one has merely reached the end of the beginning". Further enhancements are required to extend the interpretation process developed in this project to a wider, if not the whole, domain of objects which may be modelled by the Constructive Solid Geometry representation. A number of areas of work also remains to be done, such as technical deficiencies and enhancements of the software for commercial acceptability and exploitation. Suggestions for future work are described in the following two sections, the first of which lists the technical problems and deficiencies which are yet to be solved, and the second discusses enhancements that may be required for commercial exploitations. The following suggestions are concerned solely with the generation of a solid model from already stored orthographic projections.

9.2.1) TECHNICAL DEFICIENCIES:

1. The algorithms described above can only accept input data which describes complete and unambiguous orthographic projections of objects. In practice, however, most drawings are either over or under defined. Aspects of this problem have been addressed in this research (chapter 2), and some suggestions for checking the input data were presented as a series of tests. These tests (which are to be performed by the 'Raw Data Interpreter' subprocess described chapter 5) have not yet been implemented. The problem of detecting redundant, incomplete, or conflicting information in an engineering drawing is a complex one, and the above tests merely address certain aspects of the problem, such as dangling edges and self-intersecting loops.

2. An important objective of this research was to automate

the interpretation process of orthographic projections as solid models. Whilst this has been achieved for a limited (although important) class of objects, the basic strategy has been shown to be valid in the context of an iterative process for which the concept of approximation models was introduced. In this connection, the following issues arise:

a) Will such an iterative process fail to converge to an adequate solution in accordance with some criterion ?

b) Will the generation of an exact object require an unacceptable number of iterations or, indeed, be impossible? On which criterion should the process be terminated in order to generate an 'acceptable' model ? What is defined as an 'acceptable' model ?

Question (a) addresses the problem of instability which is an aspect found in any iterative process. This undesirable condition may be detected either by the 'man in the loop', i.e. the user, or alternatively, by computing well-defined properties with engineering significance, such as the masses, or volumes, of two consecutive output models; the mass of one approximation model must always be smaller than the previous approximation model, since an iteration consists of subtracting one, or more, smaller volumes, i.e. masses, from the previous approximation model. Divergent conditions may then be detectable when the mass of an nth approximation model is greater than the mass of the (n-1)th approximation model.

The next questions address the problem of model acceptability. The feed back step, i.e. the iteration process, is not always required, such as in the case for the interpretation of orthographic views of prismatic and ortho-prismatic objects since discrepancies will not be found when comparing the input views data and the orthographic views data of output geometric model. If discrepancies are detected, the geometric model may nevertheless be acceptable for some applications such as volume and mass calculations in preliminary design, or even for Finite Elements Analysis purposes. The same geometric model, however, may be unacceptable for other applications such as those where fine detail is important, as in NC machining operations, or in the design and manufacture of high precision engineering components. For example, in the design of hydraulic components, a very narrow but vital fluid conduit would represent a very small volume which cannot be ignored. A criterion for model acceptability is therefore required. It is suggested that the level of accuracy which may be acceptable for some applications is that at which there are minor differences in some significant quantity between successive iterations. For example, if the application for the geometric model is the computation of heat transfer from the surface, then a minute change in surface area per iteration is indicative of an acceptable model. Alternatively, a simple value, again such as volume, whose decrement has reached a certain level between successive approximations may provide a practical test of model acceptability. At present, the decision to allow the process to continue or to terminate the iterations, is carried out by the user.

3- The interpretation process developed in this work is not only iterative but recursive too, as mentioned in chapter 5. Arbitrary loops are recursively decomposed into basic patterns. This recursive aspect of the process also arises when enhancement of the first approximation geometric model is required. Subobjects which are to be identified from the pseudo-views, and subtracted from the first approximation model, may be arbitrary objects which may not be readily identified as primitives. These subobjects must be interpreted as solid models before subtracting them from the approximation model. The case may then arise where, in order to reconstruct these subobjects, it is again necessary to identify further subobjects which are to be removed from the approximation model of the previous subobjects. At present, the interpretation process caters only for general 3D objects which can be enhanced by subtracting subobjects which are readily identified as 3D primitives. This technical problem, however, can be solved in exactly the same manner as for the decomposition of arbitrary loops into basic patterns; thus by using pointers which store the information concerning the relationship between parent and children subobjects.

4- One of the objectives of this research was to develop algorithms which require minimum user input. For this reason, hidden edges, represented as dashed lines in engineering drawings, do not have to be specified as such in the input data. Such depth information, however, is available in the description of the orthographic views of the first approximation model, and stored in the parametric file generated by the solid modeller, as described in section 6.3.1. This information may prove useful in the feed back subprocess to identify subobjects from the pseudo-views.

9.2.2) <u>REQUIREMENTS FOR COMMERCIAL SOFTWARE</u>:

The C.I.E.D.S.M. software is, in itself, an example of a useable research system which is not yet in a commercially acceptable form. This is mainly because it has been developed on the basis of a number of assumptions which impose restrictions on the user input which may be commercially unacceptable. Input flexibility is regarded as one of the major requirements for commercial software because users generally prefer systems which provide various options from which they can make their choice. C.I.E.D.S.M. software has been developed to accept a minimum of three orthographic projections which must also be defined in the first angle projection system. In practice, objects may be represented by only two orthographic views, and very often may require the use of auxiliary and cross sectional views, especially where complex mechanical engineering components are concerned. Furthermore, in engineering drawing practice, the first and third angle projection system are used by different companies. To cater for third angle would required rather straightforward changes to the conventions adopted when raw data are interpreted. Options for auxiliary and cross sectional views are a somewhat larger issue, but must be provided if the software is to be commercially exploited.

One desirable feature, commonly found in popular commercial software, is a user-friendly interface. In the above software, dialogue between user and machine could be made more attractive by :

a) designing and implementing a front end which consists of a series of menus, to display a number of options, such as those described above, from which the user is able to select those that suit him most.

b) providing a link to commercial 2D draughting packages, such as Autocad® [51], MacDraft™ [52], or PAFEC Ltd. "DOGS"[53], in order to generate and transfer input data describing orthographic projections.

c) generating intermediate files which would be used to store, perhaps in the form of a journal, the dialogue between user and machine, such as error status and user input commands. The above software already provides two text files (the solid modeller input and output files) which could be used for such a purpose.

d) improving the graphics to enable the user to view the progress that has been made with the generation of models, perhaps by displaying two or more consecutive approximation models, or by dynamically showing the changes of a solid model, provided that the display proved neither detrimental to the progress of the main computation nor irritating to watch.

The other major requirement for commercial software is processing speed. This is a function of both the algorithms and the processing hardware. A great deal of effort has been concentrated during this research on the improvement of execution speed of the algorithms, as discussed in chapter 7. The most apparent delay occurs when running the PAFEC "BOXER" solid modelling software on the Apollo DN3000 workstation. This is almost certainly caused by the complexity of the Boolean operation algorithms; a factor which is unlikely to be improved in the foreseeable future.

9.3) POTENTIAL BENEFITS:

Discussions with major companies such as PAFEC Ltd., Deltacam Systems Ltd., Radan Computational and Superdraft Systems, have revealed great interest in the work undertaken in this project. Such interest tends to the conclusion that the commercial benefits of a successful application of this work are already apparent to such companies.

One such application would mean that solid models could be produced from existing engineering drawings at a modest cost. The difficult, time-consuming and labour-intensive task of generating solid models using 3D modellers, would be completely eliminated. As a result of this, much faster links to applications such as Finite-element analysis, CNC machine tool tape generation, mechanism simulation and other engineering applications requiring a solid object description, would be achieved. Furthermore, engineering designers would be able to develop and improve their products much more rapidly since they would only need to modify design sketches and drawings. Hence, the working environment of the engineering designer would be significantly improved and valuable 2D data need not be discarded.

Engineering education is also one area which would benefit from this work. The automatic generation of solid objects from engineering drawings would be a helpful tool in the training of engineering draughtsmen, as they would be able to observe their progress and to obtain immediate feed back during a draughting exercise. It would also help in preserving and enhancing draughting skills. As far as PAFEC Ltd. is concerned, the above software would also obviate the need to train users to use the 'BOXER' solid modeller.

Lastly, the algorithms developed in this work can potentially contribute to the field of conversions between geometric modelling representations; it has been reported that the exact, or even the approximate, conversion of Simple Sweep modelling representation into Constructive Solid Modelling representation have yet to be achieved (appendix B). The work reported, in this thesis, on the generation of uniform-thickness (prismatic) models is fundamentally a process of converting a contour into a Constructive Solid Geometry representation. Since in Translational Sweep representation (section 3.4.2) objects are defined in terms of contours and trajectories, it is therefore possible, to implement the above algorithms to achieve the exact conversion of such a geometric modelling scheme into a Constructive Solid Geometry representation.

APPENDIX A

ELEMENTS OF GRAPH THEORY

A.1) INTRODUCTION:

Figures A.1 and A.2 depict, respectively, an electrical network and a sectional view from an engineering drawing. It is clear that both of them can be represented diagrammatically by means of points and lines as in Figure A.3. The points A, B, C, D, E, F, G, H, I, J, H, K, L, M, N, O, P AND Q are referred to as 'Nodes', and the lines connecting them are called 'Edges'. The whole diagram is a 'Graph'.

The 'Degree' of a node is the number of edges which have that node as an endpoint. A node of degree n is referred to as a n-node and a 2-node is one of degree 2. Nodes B, C, E, H, I, L and M are 2-nodes, whereas Nodes A, D, F, G, J, K, N, O, P and Q are 3-nodes. It is not possible for 1-nodes to exist in a graph which accurately represents an orthographic view of a solid object, but nodes of all higher degrees are possible.

The graph shown in Figure A.3 is a 'Simple' graph, which by definition, is a graph where there is never more than one edge joining a given pair of nodes. If there is more than one edge joining a pair of vertices then they are called 'Multiple Edges'. One instance where multiple edges occur in orthographic views is shown in Figure A.4; there are two nodes, A and B, which are joined by a semi-circular edge and by a straight edge. Another example

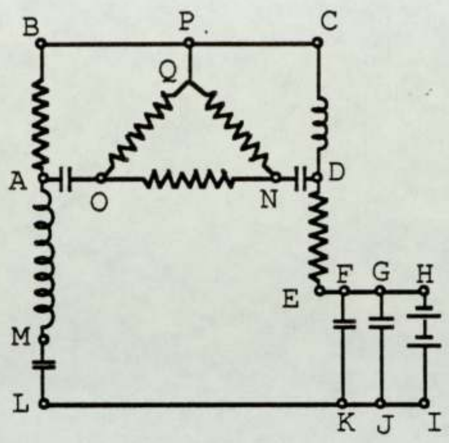


Fig. A.1: An electrical network

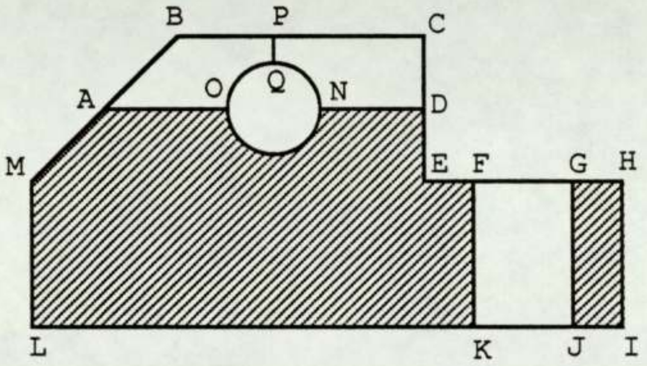


Fig. A.2: Sectional view from an engineering drawing

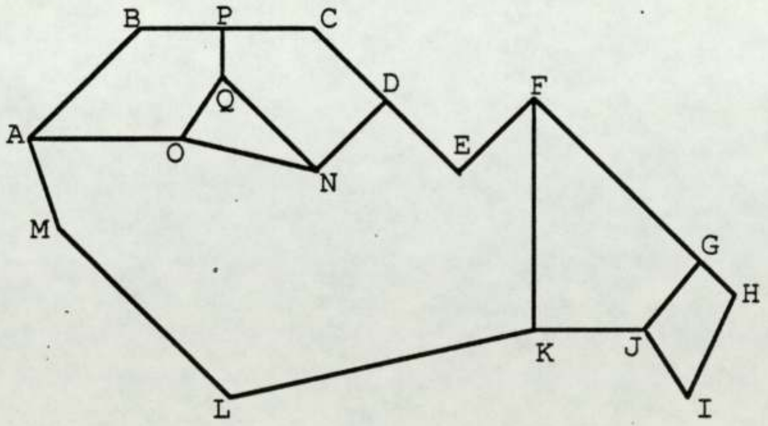


Fig. A.3: Graph equivalent to Figures A.1 and A.2

where multiple edges can arise is when two lines of equal projected length are superposed on a view.

A 'Loop' is an edge which has both endpoints at the same node. This may occur on engineering drawings whenever there is a complete circle. A circle may be adequately modelled by storing the coordinates of its centre and the coordinates of an arbitrary point on its circumference. Figure A.5 shows that a circle which is a loop having both endpoints at the shown arbitrary node. In this thesis, the term 'loop' is also used to refer to a closed 'path' or 'circuit' as defined in the following section.

A.2) <u>DEFINITIONS</u>:

Formally, a 'Graph' G is defined to be a pair [N(G),E(G)], where N(G) is a non-empty finite set of elements called 'Nodes' (or Vertices, or Points), and E(G) is a finite 'family' of unordered pairs of elements of N(G) called 'Edges'. The word 'family' is used to represent a collection of elements, some of which may occur several times; for example, {a,b,c} is a set, but (a,a,b,c,c,c) is a family. Note that the use of the word 'family' permits the existence of multiple edges. Thus, in Figure A.3, N(G) is the set { A, B, C, D, E, F, G, H, I, J, K, L, M, N, O ,P , Q} and E(G) is the family consisting

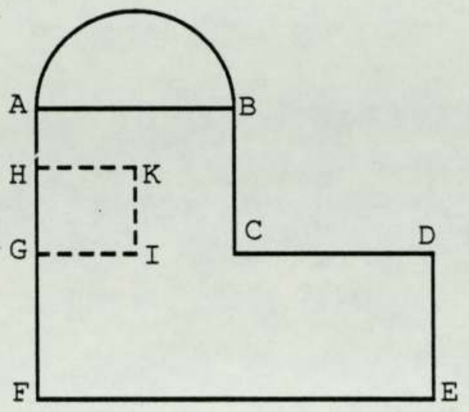


Fig. A.4: View of an engineering drawing

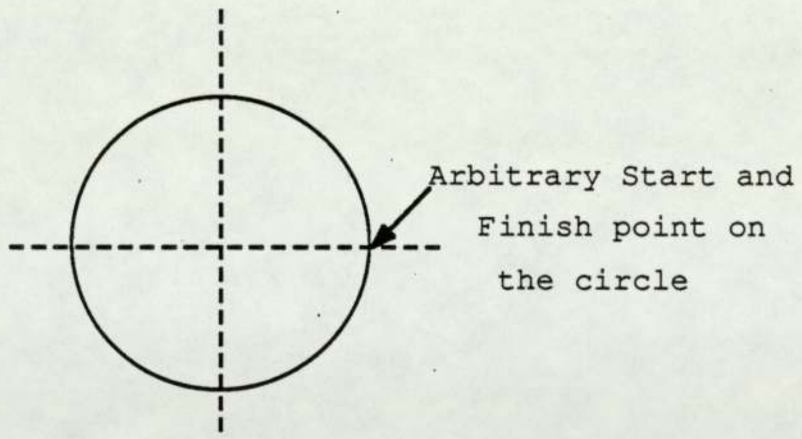


Fig. A.5: Occurence of a single edge loop on an engineering drawing

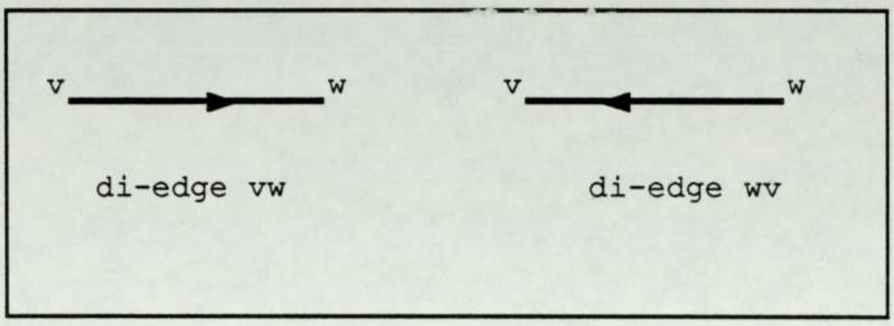
of the Edges {A,B}, {B,P}, {P,C}, {C,D}, {D,E}, {E,F}, {F,G}, {G,H}, {H,I}, {I,J}, {J,K}, {K,L}, {L,M}, {M,A}, {A,O}, {P,Q}, {N,D}, {F,K}, {G,J}, {O,N}, {N,Q} and {Q,O}.

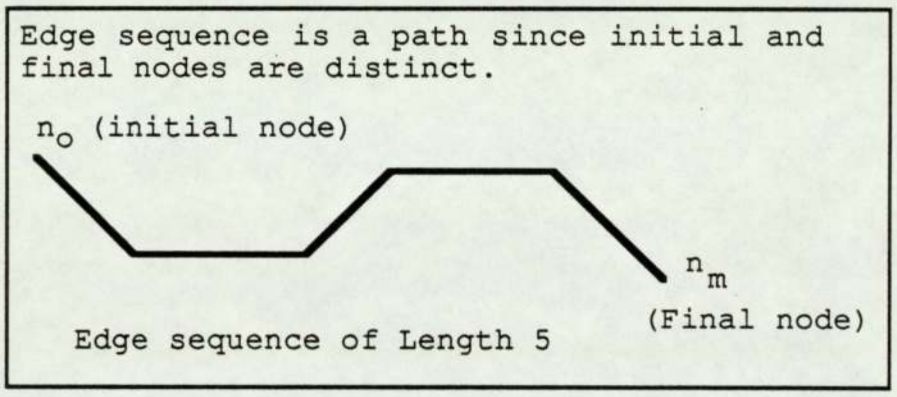
A 'Digraph' D, is defined to be a pair [N(D),E(D)], where N(D) is a non-empty finite set of elements called Nodes and E(D) is a finite family of ordered pairs of elements of N(D) called 'Di-edges'. A di-edge whose first element is v and whose second element is w is called a di-edge from v to w and is written {v,w}, or simply vw, as shown in Figure A.6. The di-edge vw is different from the di-edge wv.

An 'Edge-sequence' of a given graph G, is defined as a finite sequence of edges of the form

$n_o n_1, n_1 n_2, \ldots\ldots\ldots\ldots, n_{m-1}, n_m$

(also denoted by $n_o \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots\ldots\ldots \rightarrow n_{m-1} \rightarrow n_m$). It is clear that an edge-sequence has the property that any two consecutive edges are either adjacent or identical. The node no is called the initial node and the node nm is called the final node of the edge-sequence which may then be referred to as an edge-sequence from no to nm. The number of nodes in an edge-sequence is called its 'Length'. Thus, the length of the edge-sequence in Figure A.3 is 21. An edge-sequence in which all the edges are distinct is called a 'Trail'. If, in addition, the nodes $n_o, n_1, .., n_m$ are distinct (except possibly $n_o = n_m$),




Edge sequence is a path since initial and final nodes are distinct.
n_o (initial node)
n_m
(Final node)
Edge sequence of Length 5

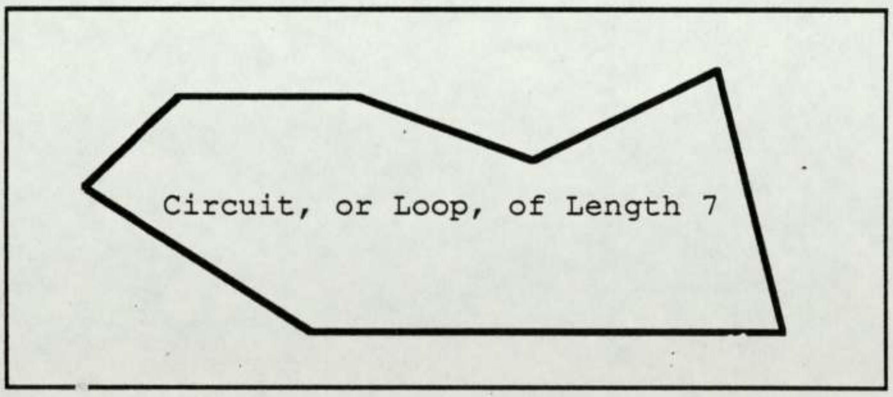


Fig. A.6: Definitions of Digraphs, Paths and Circuits

then the trail is called a 'Path'. A path, or trail, is defined as 'closed' if $n_o = n_m$, and a 'Circuit' is a closed path containing at least one edge. For convenience, a circuit is also referred to, in this work, as a loop.

The above definitions are all illustrated in Figure A.6, and are to be found in the book by Wilson [54].

APPENDIX B

CONVERSIONS BETWEEN GEOMETRIC MODELLING REPRESENTATIONS

B.1) INTRODUCTION:

The geometric modelling representation schemes discussed in chapter 3 all have their specific advantages and disadvantages. For instance, a boundary representation is very suitable for making line drawings, but it requires a large mount of memory space. On the other hand, with constructive solid geometry input of models of mechanical parts is easily achieved, but it in turn is less suitable for making such drawings.

Baer, Eastman and Henrion [9], distinguish four categories of model: the 'Definition Language' or input, the 'Data Representation' or data storage of the model, 'Conceptual Model, and 'Applications'. Each of these categories impose different requirement on the representation schemes. For this reason, many modellers provide multiple object representations. For instance, input may be done by a constructive solid geometry representation, since such representation has the merit of being adequate for input and for representing only true solids. In this form, the input may then be stored in a database. If line drawings are required, the representation is converted into a boundary representation. This specific conversion, from constructive solid geometry representation to a boundary representation, is known as the 'Boolean Evaluation', whose algorithm is roughly described in section B.2. Some of other examples of conversion algorithms are briefly discussed in section B.3. Requicha and Voelcker [55] outlined all the possible conversions between representations. These are given in Figure B.1, where an 'exact' conversion is defined as one which produces a representation of exactly the same object, and an 'approximate' conversion is a conversion which produces an approximate representation of the original object, for instance by using only planar faces or cubical cells. It can be observed from Figure B.1, that not all conversions between any two representation schemes are possible. One reason is that a conversion is impossible in principle, for example from constructive solid geometry to sweeping, and from an approximate to an exact representation. Another reason is that a conversion is possible in principle but the algorithm has not yet been developed.

B.2) **BOOLEAN EVALUATION**:

The conversion from constructive solid geometry into a boundary representation is very important, because a combination of these two representations in many modellers. Algorithms for boundary evaluation that allows primitives with curved surfaces are very difficult to implement, mainly because the intersection curves between any combination of curved surfaces is very difficult to be determined, especially for complex surfaces. For this

	EXACT				APPROXIMATE	
FROM / TO	CD	BR	CSG	SS	SE	BR
CD		K	E	I		
BR	E		E	I	KNOWN	
CSG	E	K		I		
SS	K	K	E			
SE						K
BR		IMPOSSIBLE			K	

K = Known, E = Experimental, I = Impossible

CD = Cellular Decomposition
BR = Boundary Representation
CSG = Constructive Solid Geometry
SS = Simple Sweep
SE = Spatial Enumeration

Fig. B.1: Possible conversions between representations after Requicha and Voelcker (1983)

reason, most algorithms that have been implemented only work on primitives with planar faces, so that primitives with curved surfaces have to be approximated.

In 1983, Mantyla [56] proposed an algorithm for Boolean evaluation of two primitives bounded by planar surfaces. The evaluation is achieved stepwise by first combining two primitives, and then combining the result with the third primitive, etc. The input to the algorithm consists of a number of primitives described by a boundary representation and a CSG tree indicating how these primitives have to be combined. A boundary representation of the object is obtained.

The algorithm can be divided into two steps:

1) determine the intersection lines of the input primitives

2) determine the resulting object by combining the relevant parts of the input primitives.

The first step of the algorithm can be achieved as follows:

a) determine the intersection points of all edges from one primitive with all faces from the other, and the other way around

b) determine for each face of the primitives, starting from the intersection points, chains of intersection lines, as shown in Figure B.2.

The second step of the algorithm can be divided further into two steps:

2.1) subdivide the primitives into two parts. The faces of the primitives are subdivided at the chain, or chains, of the intersection lines. The resulting parts of the faces from one primitive (A) are classified as inside or outside the other primitive (B), as shown in Figure B.2, where A is subdivided into AinB, which consists of the parts of A inside B, and AoutB, which consists of the parts of A outside B. Likewise, B is subdivided into BinA and BoutA, also shown in Figure B.2.

2.2) select the relevant parts, depending on the Boolean operator, and combine these parts. To determine the resulting object, the relevant parts of A and B have to selected, which depends on the Boolean operator:

$A \cup B$: AoutB and BoutA

$A \cap B$: AinB and BinA

A - B: AoutB and BinA

B - A: BoutA and AinB.

The two selected parts are then combined to produce the boundary representation of the resulting object. It can be seen from Figure B.2 that in all case such Boolean evaluation results in the correct object.

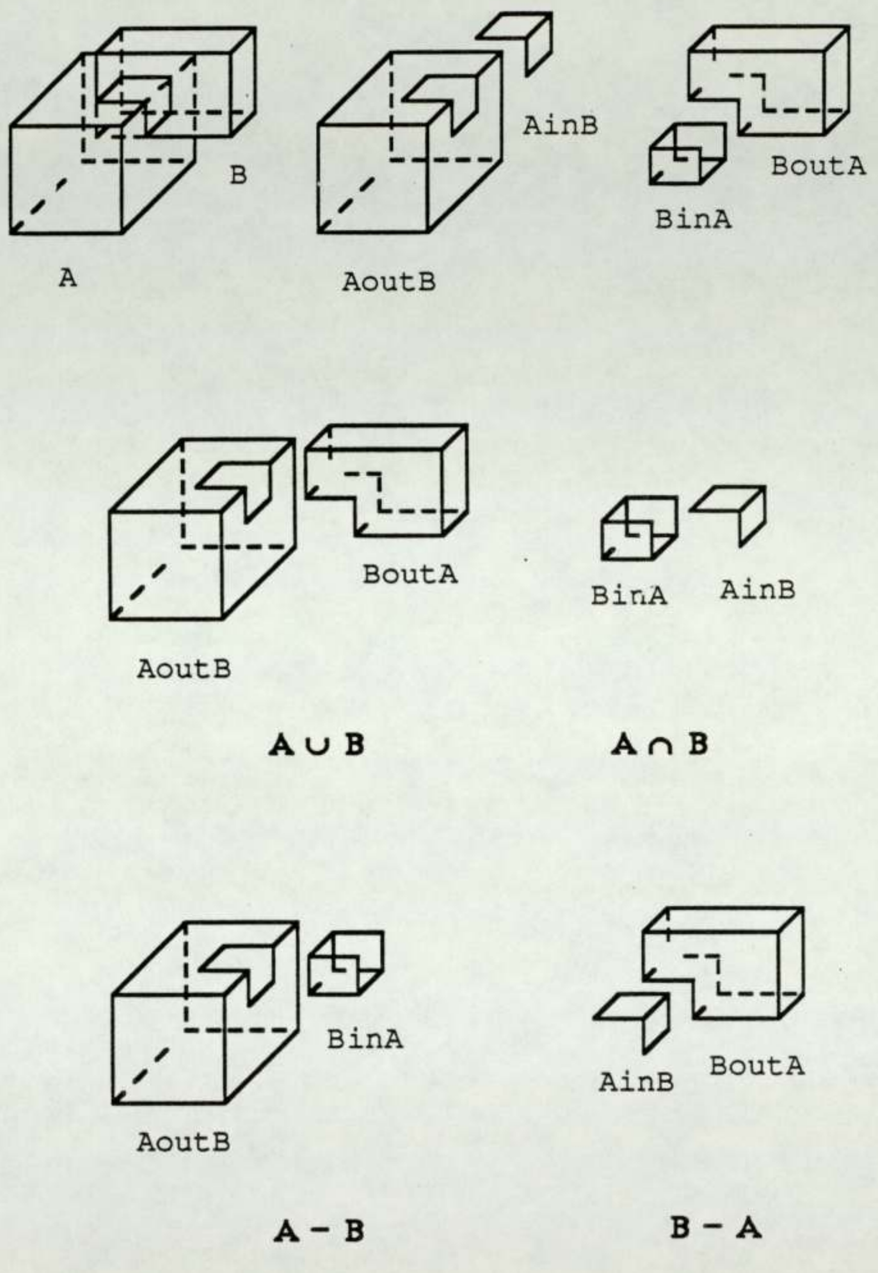


Fig. B.2: Boolean evaluation

B.3) <u>SOME CONVERSION ALGORITHMS</u>:

There are a number of algorithms which have been developed to convert one representation scheme into another. Some of these are briefly discussed here.

1) From translational sweep to boundary representation:

a) make faces perpendicular to the trajectory bounded by the contour in both endpoints of the trajectory

b) make faces parallel to the trajectory bounded by parts of the contour and edges parallel to the trajectory.

2. From constructive solid geometry to spatial enumeration:

a) determine for every voxel in the grid whether it is inside the object

b) for a composite object in the CSG tree this can be done (recursively) by applying its operator ti its left and right branches. For instance, if a node in the CSG tree with the union operator the voxel inside the left or the right branch, the voxel is inside the combined objects represented at that node.

c) for a primitive object in the CSG tree this can be achieved by substituting the coordinates of the centre of the voxel in the equations of the half spaces defining the object.

3) From boundary representation or constructive solid

geometry to arbitrary cellular decomposition:

These algorithms are particularly important because of their potential use for automatic generation of meshes of cells for finite element analysis. Unfortunately, they exist only in an experimental form.

APPENDIX C

HOMOGENEOUS COORDINATES

C.1) INTRODUCTION:

As early as 1965, L. G. Roberts [57] suggested that homogeneous coordinates could be used to describe the most commonly required transformations and projections. Since then, the technique has become commonplace and is taught as part of the standard graphics curriculum [58,59].

Homogeneous coordinate representations of points and planes are particularly useful for describing and transforming geometric models. The term 'homogeneous' is applied to the representations because each class of object is modelled by an equation which has no explicit parameter. The familiar explicit equation which describes a two-dimensional line is $y = ax + b$. The homogeneous (implicit) equation for the same line is $ax - y + b = 0$.

The homogeneous representation of a two-dimensional point (x,y) is written as $[wx,wy,w]$, where w is any non-zero scalar which is sometimes referred to as the 'Scale Factor'. the symbols 'wx' and 'wy' are diphthongs; they are single numbers, not multiplications. The mapping from a homogeneous point $[wx\ wy\ w]$ back to its two-dimensional image is simply $(wx/w,\ wy/w)$.

Three-dimensional objects are treated in an analogous fashion. The implicit form of the equation of a plane is $a_1x + a_2y + a_3z + a_4 = 0$. The homogeneous representation of the three-dimensional point (x,y,z) is written as [wx wy wz w] for any non-zero value of w. Again, the mapping from this homogeneous point back to its three-dimensional image is (wx/w, wy/w, wz/w).

C.2) TWO-DIMENSIONAL POINTS AND LINES:

C.2.1 A two -dimensional point (x,y) is represented by the homogeneous row vector **p** = [wx wy w], as described above. Any non-zero scalar multiple of this representation represents the same two-dimensional point. The homogeneous point **p** is converted back to its ordinary coordinates (wx/w, wy/w).

C.2.2 A line in 2D-space is represented by a column vector:

$$\gamma = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

C.2.3 The condition that a point **p** is on the line γ is:

$$\mathbf{p} \cdot \gamma = 0$$

This is an inner product which is equivalent to the scalar equation:

$$a(wx) + b(wy) + c(w) = 0$$

If the scalar product is not zero then **p** does not lie on the line. The product is however, proportional to the distance of the point to the line, where:

$$\text{distance} = \frac{a(wx) + b(wy) + c(w)}{w\sqrt{(a^2 + b^2)}}$$

C.2.4 The line γ between a point **p** = [p1 p2 p3] and a point **q** = [q1 q2 q3] is given by the vector product:

$$\gamma = \begin{bmatrix} q_2p_3 - q_3p_2 \\ q_3p_1 - q_1p_3 \\ q_1p_2 - q_2p_1 \end{bmatrix}$$

This is the result of solving the following implicit equations simultaneously:

$$ap1 + bp2 + cp3 = 0$$
$$aq1 + bq2 + cq3 = 0$$

It can be easily verified that the points **p** and **q** are on the line γ, i.e., $\mathbf{p} \cdot \gamma = 0$ and $\mathbf{q} \cdot \gamma = 0$.

C.2.5 The point at the intersection of two lines given by:

$$\gamma = \begin{bmatrix} a_1 \\ b_1 \\ c_1 \end{bmatrix} \quad \text{and} \quad \lambda = \begin{bmatrix} a_2 \\ b_2 \\ c_2 \end{bmatrix}$$

is given by:

$$\mathbf{p} = [\ (b_1c_2 - b_2c_1) \quad (c_1a_2 - c_2a_1) \quad (a_1b_2 - a_2b_1)\ ]$$

C.3) THREE-DIMENSIONAL POINTS, LINES AND PLANES:

C.3.1 A three-dimensional point (x,y,z) is represented by the row vector $\mathbf{p}$ = [wx wy wz w]. Any non-zero scalar multiple of this representation represents the same three-dimensional point.

C.3.2 A line is represented as a function of some parameter **t** which ranges from zero at one endpoint of this to unity at the other endpoint. Γhis parametric formulation is:

$$\mathbf{p} = [t \quad 1]\ \mathbf{L}$$

where **L** is a 2x4 matrix which may be found by requiring that :

the row vector at one endpoint of line = [0 1] **L**

the row vector at other endpoint of line = [1 1] **L**

C.3.3 A plane is represented by a column vector:

$$\gamma = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

C.3.4 The condition that a point **p** = [wx wy wz w] is on a plane is **p** . $\gamma = 0$. This is equivalent to the scalar equation :

$$a(wx) + b(wy) + c(wz) + d(w) = 0$$

If the scalar product is zero then the point **p** lies on the plane. The product is proportional to the perpendicular distance of the point to the plane, where:

$$\text{distance} = \frac{a(wx) + b(wy) + c(wz) + d(w)}{w \sqrt{(a^2 + b^2 + c^2)}}$$

C.3.5 Three non-colinear homogeneous points:

$$\mathbf{p} = [wp_1 \ wp_2 \ wp_3 \ w]$$

$$\mathbf{q} = [wq_1 \ wq_2 \ wq_3 \ w]$$

$$\mathbf{r} = [wr_1 \ wr_2 \ wr_3 \ w]$$

determine a plane. The equation of the plane can be determined by solving the following simultaneous equations:

$$a\ (wp_1) + b\ (wq_1) + c\ (wr_1) + dw = 0$$

$$a\ (wp_2) + b\ (wq_2) + c\ (wr_2) + dw = 0$$

$$a\ (wp_3) + b\ (wq_3) + c\ (wr_3) + dw = 0$$

The plane equation from a set of more than three points may be obtained by using the following equations:

$$a = \sum_{i=1}^{n} (y_i - y_j)\ (z_i + z_j)$$

$$b = \sum_{i=1}^{n} (z_i - z_j)\ (x_i + x_j)$$

$$c = \sum_{i=1}^{n} (x_i - x_j)\ (y_i + y_j)$$

where:

$j = i \pmod n + 1$

and:

n = number of points.

The value of d is found by requiring any one of the n points to lie on the plane.

REFERENCES

List of references:

1 **Sutherland, I.E.**, "SKETCHPAD: A Man-Machine Graphical Communication System", Proc. SJCC 23, 329 (1963).

2 **Brown, B.E.**,"Modelling of Solids for Three-Dimensional Finite Element Analysis", Ph.D. Dissertation, Department of Computer Sciences, Univ. of Utah, Salt Lake City, UT June 1977.

3 **Boyse, J.**, "Interference Detection among Solids and Surfaces", *Commun. ACM* 22, 3-9 (January 1979).

4 **Wesley, M.A., Lozano-Perez, T., Lieberman, L.I., Lavin, M.A.** & **Grossman, D.D.**, "A Geometric Modelling System for Automated Mechanical Assembly", *IBM J. Res. Develop.* 24, 64-74 (January 1980).

5 **Woo, T.C.H.**, "Computer Understanding of Design", Ph.D. Thesis, University of Illinois at Urbana-Champaign,1975.

6 **Taylor, R.H.**, "A Synthesis of Manipulation Control Programs from Task Level Specifications", *Report No. STANCS-76-560*, Stanford Artificial Intelligence Laboratory, Computer Sciences Department, Stanford Univ., Palo Alto, CA, July 1976.

7 **Udupa, S.**, "Collision Detection and Avoidance in Computer Controlled Manipulators", Ph.D. Thesis, California Institute of Technology, Pasadena, CA, 1977.

8 **Lozano-Perez, T.** & **Wesley, M.A.**, "An Algorithm for Planning Collision-Free Paths among Polyhedral Objects", *Commun. ACM* 22, 560-570 (October 1979).

9 **Baer, A., Eastman, C.** & **Hervion, M.**, "Geometric Modelling: A survey", *Computer-Aided Design J.*,vol.11, no 5, pp. 253-271, September 1979.

10 **BCS Conference Documentation Displays Group**, "Fundamentals of Geometric Modelling - Review and potential", London, U. K., 26th February 1986.

11 **Faux, I.D.** & **Pratt, M.J.**, "Computational Geometry for Design and Manufacture", Ellis Horwood Ltd., Chichester.

12 **Fergusson, J.C.**, "Multivariate Curve Interpolation ", *Journal ACM*, vol. 11, no. 2 (1964).

13 **Bezier, P.**, "Numerical Control: Mathematics and Applications", John Wiley & Sons Ltd., (1972).

14 **Bezier, P.**, "Mathematical and Practical Possibilities of UNISURF", in R.E. Barnhill & R. F. Riesenfeld (eds), *Computer Aided Geometric Design*, Academic Press, New York, (1974)

15 **Gordon, W.J.** & **Riesenfeld, R.F.**, " B-spline Curves and Surfaces", in *Computer Aided Geometric Design*, Academic Press, pp. 95-123, New York (1974).

16 **Braid, I.C.**, "Designing with Volumes", Ph.D. Thesis, Cambridge University, U. K., (1972).

17 **Braid, I.C.** & **Lang, C.A.**, "Computer-Aided Design of Mechanical Components with Building Blocks",*Proceedings of the Second IFIP/IFAC Int. Conf. on Programming Languages for Machine Tools*, PROLAMAT'73, Budapest, April 10-13, pp 109-118 and pp. 173-184.

18 **Braid, I.C.**, .Jew Directions in Geometric Modelling", *CAD group Document* No.98, Computer Laboratory, Univ. of Cambridge, U. K., March 1978.

19 **Requicha, A.A.G.**, "Representation for Rigid Solids: Theory, Method and Systems", *ACM Computing Surveys*,

vol.12, No.4, pp. 437-464, (1980).

20 **Requicha, A.A.G.** & **Voelcker, H.B.**, "Solid Modelling: A Historical Summary and Comtemporary Assessement", IEEE *Comp. Gtaphics & Applications*, vol.2 No.4, pp. 9-24, (1982).

21 **Meagher, D.**, "Geometric Modelling using Octtree Encoding", *Comp. Graphics & Image Processing*, vol.19, No.2, pp. 129-147, (1982).

22 **Gargantini, I.**, "Linear Octtrees for Fast Processing of Three-Dimensional Objects", Comp. *Graphics & Image Processing*, vol.20, No.4, pp. 365-375, (1982).

23 **Wijk, J.J. Van**, "Ray Tracing Objects Defined by Sweeping Planar Cubic Splines", *ACM Trans. on Graphics*, vol.3, No.3, pp. 223-237,(1984).

24 **Wijk, J.J. Van**, "Ray Tracing Objects Defining by Sweeping a Sphere", in K. Bo & H. A. Tucker (eds), *Proc. Eurographics'84*, Elsevier Science Publishers BV (North-Holland), Amsterdam, p.73, (1984).

25 **Bronsvoort, W.F.** & **Klok, F.**, "Ray Tracing Generalized Cylinders", *ACM Trans. on Graphics*, vol.4, No.4, pp. 291-303, (1985).

26 **Post, F.H.** & **Klok, F.**, "Deformation of Sweep Objects in Solid Modelling", in: A. A. G. Requicha (ed), *Proc. Eurographics'86*. Elsevier Science Publishers BV (North-Holland), Amsterdam, p.103, (1986).

27 **Baumgart, B.G.**, "A Polyhedron Representation for ComputerVision", in: *AFIPS Proc. Nat. Comp. Conf.44*, p.589, (1975).

28 **Varady, T.** & **Pratt, M.J.**, "Design Techniques for the

Definition of Solid Object with Free-form Geometry", *Computer-Aided Geometric Design* Vol.1, pp.207-225, 1984.

29 **Requicha, A.A.G.** & **Voelcker, H.B.**,"Constructive Solid Geometry", *Document TM-25*, Production Automation Project at the University of Rochester, New York, November 1977.

30 **Requicha, A.A.G.** & **Voelcker, H.B.**,"Geometric Modeling of Mechanical Parts and Processes", *Computer*, 7006 Vol.10, No.12, pp. 48-57, Dec. 1977.

31 **SUTHERLAND, I.** "Three Dimensional Data Input by Tablet", *Proc. IEEE 62*, April 1974.

32 **THORNTON, R.W.**" Interactive Modelling in Three Dimensions through Two Dimensional Windows " *3rd Int. Conf. on Computers in Engineering and Building Design*, (1978).

33 **IDESAWA, M.**" A System to Generate a Solid Figure from Three Views ", *Bull. JSME*, vol.16, Feb 1973, pp.216-225.

34 **IDESAWA, M.**, **SOMA,T.**, **GOTO, E.** & **SHIBATA, S.**, "Automatic Input of Line Drawing and Generation of a Solid Figure from Three-View Data ", *Proceedings of the Int. Joint Computer Symposium*, 1975, pp. 304-311.

35 **LAFUE, G.** " Recognition of Three Dimensional Objects from Orthographic Views ", *Proceedings 3rd Annual Conf. on Computer Graphics, Interactive Techniques and Image Processing*, ACM/SIGGRAPH, July 1976, pp. 103-108.

36 **PREISS, K.** " Algorithms for Automatic Conversion of a 3-View Drawing of a Plane-Faced Part to the 3D Representation ", *Computers in Industry*, vol. 12, 1981, pp. 133-139.

37 **PREISS, K.** & **KAPLANSKY, E.** " Solving CAD / CAM Problems by Heuristic Programming ", *Computers in Mechanical Engineering*, Sept. 1983, pp. 56-60 .

38 **ALDEFELD, B.** " On Automatic Recognition of 3D Structures from 2D Representations ", *Computer-aided Design*, vol. 15, no. 2, March 1983.

39 **MARKOWSKY, G.** & **WESLEY, M.A.**, "Fleshing Out Wire Frames", *IBM J. Res. Develop.*, vol. 24, no.5, sept.1980 pp. 582-597

40 **WESLEY, M.A.** & **MARKOWSKY, G.** "Fleshing Out Projections", *IBM J. Res. Develop.*, vol. 25, no. 6, Nov. 1981, pp. 934-953

41 **SAKURAI, H.** & **GOSSARD, D.C.**, "Solid Model Input through Orthographic Views ", *Computer Graphics J.*, vol. 17, no. 3, July 1983, pp. 243-252

42 **KAINING, G., ZESHENG, T.** & **JIAGUANG, S.**, Reconstruction of 3D Objects from Orthographic Projections", *Computer Graphics Forum*, Vol. 5, No. 4, December 1986

43 **ALDEFELD, B. and RICHTER, H.** "Semiautomatic Three-dimensional Interpretation of Line Drawings" *Computers and Graphics (GB) J.*, vol. 8, no. 4, 1984, pp. 371-380

44 **HO BIN,** "Inputting Constructive Solid Geometry Directly from 2D orthographic Engineering Drawings" *Computer-Aided Design J.*, vol. 18, no.3, April 1986, pp. 147-185

45 "Geometric Modelling Project : Geometric Modelling User Manual 1", Department of Mechanical Engineering, University of Leeds, August 1981.

46 "Perq GKS User Guide: Software Version 1.0", International Computers Ltd., London, U.K., First Edition, 1984.

47 "GKS Reference: Software Version 1.0", International Computers Ltd., London, U.K., First Edition, 1984.

48 "Programming with Domain Graphics Primitives: Software Version 9.0", Apollo Computers Inc., USA, First Edition, 1985.

49 "Programming with 2D Graphics Metafiles Resources: Software Version 9.5", Apollo Computers Inc., USA, First Edition, 5th December 1986.

50 **COOLEY, P.**, "Decision-Making Algorithms in Geometric Modelling", Ph.D. Thesis, Aston University, Birmingham, U.K., 1984.

51 "AutoCAD® Drafting Package: Reference Manual", Autodesk Inc., U.S.A, 1986

52 **KING J.P.**, **ADAMS P.J.**, & **SHEARMAN J.R.**, "MacDraft User's Manual", Innovative Data Design Inc., U.S.A, 1985.

53 "DOGS: User's Manual Level 3.2", Pafec Ltd., Nottingham, U.K., 1984

54 **WILSON, R.J.**,"Introduction to Graph Theory", Longman, U.K., 2nd Edition, 1979.

55 **Requicha, A.A.G.** & **Voelcker, H.B.**, "Solid Modelling: Current Status and Research Directions", IEEE *Comp. Gtaphics & Applications*, vol.3, No.7, pp.25-37, (1983).

56 **MANTYLA, M.**, "Set Operation of GWB", *Comp. Graphics Forum,* Vol. 2 (2/3), pp. 122-134, (1983).

57 **ROBERTS, L.G.**, "Homogeneous Matrix Representation and Manipulation of N-Dimensional Constructs", MS-1405, Lincoln Laboratory, MIT, May 1965.

58 **FORREST, A.R.**, "Coordinates, Transformations, and Visualization Techniques", *Computer Aided Design Group Doc.* No. 23, University of Cambridge, June 1969.

59 **NEWMAN, W.M.** & **SPROULL, R.F.**, "Principles of Interactive Computre Graphics", McGraw-Hill, New York, 2nd Edition, 1979.