City Research Online

City St George's, University of London

Citation: Sivaloganathan, S. (1991). Sketching input for computer aided engineering. (Unpublished Doctoral thesis, City, University of London)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/29258/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

SKETCHING INPUT FOR COMPUTER AIDED ENGINEERING

BY

SANGARAPPILLAI SIVALOGANATHAN
BSc MSc CEng MIMechE MIE (Sri-Lanka)

THESIS

SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

CITY UNIVERSITY

IN

MECHANICAL ENGINEERING

DEPARTMENT OF MECHANICAL ENGINEERING AND AERONAUTICS

CITY UNIVERSITY

NORTHAMPTON SQUARE

LONDON EC1V 0HB

SEPTEMBER 1991

CONTENTS

Chapter 1 INTRODUCTION

1.0	General	12
1.1	Computer Aided Design	12
1.2	Solid Modelling	13
1.3	Sketching input for computer aided engineering	14
1.3.1	Sketching and the design process	16
1.3.2	The proposed system - Why ? and What ?	16
1.4	Structure of the thesis	17

Chapter 2 A SURVEY ON SOLID MODELLING

2.0	General	18
2.1	Explanation of representation schemes	19
2.2	Representation schemes for CAE	19
2.2.1	Pure primitive instancing	20
2.2.2	Spatial occupancy enumeration	20
2.2.3	Cell decomposition	22
2.2.4	Sweep representation	23
2.2.5	Constructive solid geometry	24
2.2.6	Boundary representation B-rep	26
2.3	Analysis of solid modelling schemes	29
2.3.1	Euler operators	32
2.3.1.1	Skeletal primitives MVFS and KVFS	33
2.3.1.2	Local manipulators MEV, KEV, MEF, KEF, KEMR and MEKR	33
2.3.1.3	Global manipulators KFMRH and MFKRH	35
2.3.2	Consequences of Euler operators	35
2.4	Solid modelling re-visited	38
2.5	Sketching input - where ?	39
2.6	Data structures for boundary representation	41
2.6.1	Vertex based boundary model	43
2.6.2	Edged - based boundary models	43
2.6.3	Winged-edge data structure	44
2.7	Storing geometric information	46
2.8	Mantyla's HalfEdge data structure	47
2.9	An example representation	50
2.9.1	Representation of the 'L' block	50
2.9.1.1	List of solid nodes	50
2.9.1.2	List of edge nodes	53

2.9.1.3 List of vertex nodes 55
2.9.2 Representation of the 'L' slider 55
2.10 Inferences 56

Chapter 3 BACKGROUND THEORY

3.0 General 57
3.1 Two dimensional geometry 57
3.2 Regression of Y on X and X on Y 58
3.3 Least square fitting of straight line 60
3.4 Least squares fitting of ellipse 61
3.5 Meeting point of more than two straight lines 62
3.6 Meeting point of more than two curves and straight lines 64
3.7 Detecting a straight line 65
3.7.1 Perpendicular method 65
3.7.2 Moving slope method 66
3.7.3 Generalised conic method 67
3.8 Detecting an ellipse 67
3.8.1 Moving slope method 67
3.8.2 Generalised conic method 68
3.8.3 Points scatter method 68
3.9 Detecting the end of a straight line 68
3.10 Detecting the end of an ellipse or arc 69
3.11 Isometric sketching 70
3.12 Twelve classes of lines in an isometric sketch 73
3.13 Transformation to 3D co-ordinate system 75
3.14 Transformation of a cube - An example 75
3.15 Circles and circular arcs in isometric planes 78
3.15.1 Radius of the circle in XY plane 80
3.15.2 Identification of the slope of the major axis 81

Chapter 4 REQUIREMENT ANALYSIS AND FUNCTIONAL SPECIFICATION

4.0 General 83
4.1 Requirement analysis 83
4.1.1 Sketching process 83
4.1.2 Isometric sketching 84
4.1.3 Analysis 85
4.1.4 The requirements of a sketching input system 85
4.2 System 86
4.2.1 Operation of the digitizer 86

4.3 Statement of the problem 87
4.4 Functional specification 90

Chapter 5 THE PROGRAM

5.0 Introduction 93
5.1 Sketch stage 95
5.1.1 Sketching into the system 96
5.2 Processing the sketch in two dimensions 96
5.2.1 Finding the line segments 96
5.2.2 Finding the terminal points 99
5.2.3 Finding the analytic equations of lines 102
5.2.4 Finding vertices 105
5.2.5 Finding the edges 105
5.3 The merging process 105
5.3.1 Zero distance merging 107
5.3.2 Erase merging 110
5.3.3 Solid merging 110
5.4 Processing in three dimensions 114
5.4.1 Extracting the initial 3 dimensional details 115
5.4.2 Process construction lines 115
5.4.3 Processing of vertices 121
5.4.4 Processing with construction lines 121
5.4.5 Processing for each edge 121
5.4.6 Extracting three dimensional loops 123
5.4.7 Fitting 3D geometry 124
5.5 Program strategy 124

Chapter 6 SAMPLE SESSIONS

6.0 Introduction 126
6.1 Sample 'L' block 126
6.1.1 Processing in two dimensions 128
6.1.2 Merging 128
6.1.3 Processing in three dimensions 129
6.2 The stopper block 129
6.3 Wall fixture 131

CHAPTER 7 DISCUSSION AND CONCLUSION

7.1 Discussion 133
7.2 Conclusions 134
7.3 Areas for future work 134

REFERENCES

133

APPENDIX A

(i) Mantyla's data structures 141

APPENDIX B LISTINGS OF PROGRAMS

(i) Sketch.bas program 145
(ii) Sketch solid data structures 152
(iii) Sketch solid global variables 155
(iv) Memory allocation to vectors and matrices 158
(v) Screen management functions 164
(vi) Memory allocation functions 166
(vii) File handling functions 180
(viii) Graphic output functions 208
(ix) Processing in two dimensions functions 216
(x) Merge facilities functions 241
(xi) Processing in three dimensions functions 264

APPENDIX C OUTPUTS OF L BLOCK SKETCH

(i) The points file 282
(ii) lblock.one file 292
(iii) lblock.two file 295

LIST OF ILLUSTRATIONS

Figure 1.1	Current Design Approaches	15
Figure 2.1	Explanation of Representation	19
Figure 2.2	Illustration of Octree Representation	21
Figure 2.3	Illustration of Cellular Decomposition	22
Figure 2.4	Translational Sweep	23
Figure 2.5	Rotational Sweep	24
Figure 2.6	Illustration of Constructive Solid Geometry	25
Figure 2.7	Illustration of Boundary Representation	27
Figure 2.8	Braid's Boundary Model	28
Figure 2.9	Analysis of Solid Modelling Systems	29
Figure 2.10	Generalised Notion of Solid Modelling Systems	31
Figure 2.11	MEV Operator at Different Conditions	33
Figure 2.12	MEF Operator at Different Conditions	34
Figure 2.13	KEMR Operator at Different Conditions	35
Figure 2.14	Labelling of the 'L' Block	37
Figure 2.15	Representation of Cylinder	38
Figure 2.16	Illustration of Special Edges	39
Figure 2.17	Nine Topological Relationships	42
Figure 2.18	Winged-Edge Data Structure	45
Figure 2.19	HalfEdge Data Structure	48
Figure 2.20	Linked List Structure of Solid Representation	49
Figure 2.21	Supporting List Structures	50
Figure 2.22	Illustration of the Solid Node	51
Figure 2.23	Edge List of the 'L' Block	52
Figure 2.24	Vertex List of the 'L' slider	54
Figure 3.1	Regression of Y on X and X on Y	59
Figure 3.2	Illustration of the Perpendicular Method	65
Figure 3.3	Illustration of Moving Slope Method	66
Figure 3.4	Detection of Ellipse - Moving Slope Method	67
Figure 3.5	Detection of End-point of a Straight line	69
Figure 3.6	Finding the End point of an Ellipse	70
Figure 3.7	Illustration of isometric Cube	71
Figure 3.8	Isometric View of the Cube	71
Figure 3.9	Construction of Isometric Ellipse	72
Figure 3.10	Illustration of Isometric Cylinder	73
Figure 3.11	Co-ordinate System	74
Figure 3.12	Example Cube	76
Figure 3.13	Cube with Circles in the Isometric Planes	79

Figure 3.14	Ellipse on the XY plane	80
Figure 3.15	The Co-ordinate System	81
Figure 4.1	Digitizer Menu	88
Figure 4.2	Nodes used in Two Dimensional Processing	91
Figure 4.3	Five Nodes for Three Dimensional Processing	92
Figure 5.1	Structure of the SKETCH-SOLID Program	94
Figure 5.2	Structure of Sketch	97
Figure 5.3	Structure Diagram of Processing in Two Dimensions	98
Figure 5.4	Extracting the Line Segments	100
Figure 5.5	Finding Terminal Points	101
Figure 5.6	Finding the Analytic Equations of Lines	103
Figure 5.7	Finding Vertices	104
Figure 5.8	Finding the Edges	106
Figure 5.9	Structure of Merge Function	107
Figure 5.10	Zero Distance Merging	108
Figure 5.11	Erase Merging	109
Figure 5.12	Structure of Solid Merge	111
Figure 5.13	Processing in Three Dimensions	112
Figure 5.14	Extracting the Initial Three Dimensional Details	113
Figure 5.15	Illustration of Construction Lines of Type 1	116
Figure 5.16	Illustration of Construction Lines of Type 2	117
Figure 5.17	Structure of Processing Construction Lines	118
Figure 5.18	Structure of Processing Each Vertex	119
Figure 5.19	Structure of Processing With Construction Lines	120
Figure 5.20	Illustration of Non-Isometric Lines With Isometric Processing	122
Figure 5.21	Structure of Extracting 3D Loops	123
Figure 6.1	Simple 'L' Block	126
Figure 6.2	Photograph of the Displayed Sketch	127
Figure 6.3	Photograph of the Displayed Fitted Sketch	127
Figure 6.4	Stopper Block	130
Figure 6.5	Illustration of Wall Fixture	132

ACKNOWLEDGEMENTS

Very special thanks and appreciation are due to professor Alan Jebb, for giving me the privilege of being his student. His vision in engineering design has encouraged me to embark on this project. His frank criticism and constant encouragement has made this project a success.

Additional thanks are due to professor H.P. Wynn for his help in mathematical problems.

Sincere thanks and appreciation are due to my father and late mother for their constant encouragement to get me started in this project. Special appreciation is due to my wife for creating the environment for me to start and continue this research.

Thanks are due to Dr S. Jeyanathan, Dr V. Sivapathasundaram, Mr K. Ranganathan and Mr A. Sivapalan for their financial support during the early days of the research.

DEDICATION

To the employees of the Sri-lankan Cement Industry, for making me look deeper into Engineering, by their constant sufferings

I grant powers to the university librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgements.

Sangarappillai Sivaloganathan

ABSTRACT

The design process often begins with a graphical description of the proposed device or system and sketching is the physical expression of the design engineer's thinking process. Computer Aided Design is a technique in which man and machine are blended into a problem solving team, intimately coupling the best characteristics of each. Solid modelling is developed to act as the common medium between man and the computer. At present it is achieved mainly by designing with volumes and hence does not leave much room for sketching input, the traditional physical expression of the thinking process of the design engineer.

This thesis describes a method of accepting isometric free hand sketching as the input to a solid model. The design engineer is allowed to make a sketch on top of a digitizer indicating (i) visible lines (ii) hidden lines (iii) construction lines (iv) centre lines (v) erased lines and (vi) redundant lines as the input. The computer then process this sketch by identifying the line segments, fitting the best possible lines, removing the erased lines, ignoring the redundant lines and finally merging the hidden lines and visible lines to form the lines in the solid in an interactive manner. The program then uses these lines and the information about the three dimensional origin of the object and produces three dimensional information such as the faces, loops, holes, rings, edges and vertices which are sufficient to build a solid model. This is achieved in the following manner.

The points in the sketch is first written into a file. The computer then reads this file, breaks the group of points into sub-groups belonging to individual line segments, fits the best lines and identify the vertices in two dimensions. These improved lines in two dimensions are then merged to form the lines and vertices in the solid. These lines are then used together with the three dimensional origin (or any other point) to produce the wireframe model in three dimensions. The loops in the wireframe models are then identified and surface equations are fitted to these loops. Finally all the necessary inputs to build a B-rep solid model are produced.

CHAPTER 1
INTRODUCTION

1.0 GENERAL

Computer-Aided Engineering is breaking barriers between the compartments into which the life of an engineering product is traditionally divided. Much of the routine work can now be taken over by computers, leaving the engineer to use his professional expertise in a wider arena, looking at many more if not all possible options, within the limited time and resources allocated for the design phase. This enables him to ensure that the product under development is properly conceived, developed, manufactured and used. To summarise the developments surrounding the computer it can be said that a powerful aid in the name of computer has been invented and methods to exploit its power are developed everyday.

Computer-aided techniques can be classified as follows:

i) Techniques which do not require the abandoning of existing thinking processes and or enhance the existing thinking processes. These are widely accepted and used. Typical examples of this category are drafting packages.

ii) Techniques which do require abandoning of the existing thinking processes and need new ones. These are not used widely when compared to the techniques described above in (i).

The techniques in category (ii) may have valuable aids built into them and all that may be needed is a proper interface between the traditional thinking and the newer thinking. Solid modelling requires 'designing with primitive solids' and abandoning traditional designing with sketches and thus falls within category (ii) above. This thesis describes a method of linking solid modelling with traditional sketching.

1.1 COMPUTER AIDED DESIGN

The following is an excerpt from Beasant and Lui [1] which describes well the use of computer as an aid in design. "**Computer Aided Design (CAD) is a technique in which man and machine (computer) are blended into a problem solving team, intimately coupling the best characteristics of each. The result of this combination works better than either man or machine would work alone, and by using a multi-disciplinary approach it offers the advantage of integrated team work.**" Beasant and Lui [1] rightly argue that the characteristics of the team members, man and machine, affects the design of a CAD system.

The advent of computers in drafting and design started with the 'sketchpad' program

developed by Ivan Sutherland [2] at MIT in the 60's. It made it possible for a man and a computer to communicate through the medium of line drawings which was until then done through written statements which made the use of computers cumbersome. It enabled the user to (i) issue specific commands with a set of push buttons (ii) turn functions on and off with switches (iii) indicate position information through a light pen (iv) rotate and magnify picture parts by turning knobs and (v) observe the drawing on the display. The invention of the use of computer as a sophisticated drafting tool underwent many refinements and has given birth to many drafting packages, with many varieties of application area. More and more construction techniques are introduced everyday into drafting packages enhancing the ones already released for use. The use of computer aided drafting has also had a significant impact on the electronic design automation (EDA) process, where the connections to various components (topology) is the main consideration. Drafting packages provide enormous power in terms of speedy production of drawings, facilities to make enlarged and scaled down drawings and editing a part of the drawing without redrawing the whole of it. Also the use of computer as a drafting tool has resulted in considerable savings in terms of time and cost.

Until the introduction of computers in the design office the role of communication and storage of information in the manufacturing environment has for many years been performed by drawings and nobody has seriously thought of a replacement. Engineers are trained to be competent in reading and understanding drawings. The process of design itself depended on the ability to make drawings. However these two dimensional drawings had their limitations in storing the full information about a three dimensional solid. They needed a human to interpret their three dimensional content. With the introduction of computers to do the job of producing drawings, the limitation imposed on the design process, by the resources required for the production of drawings, is substantially reduced and researchers started to look for a better method of representing the designed artifacts, which would eliminate the shortcomings of drawings. Also it was felt that complete 3D description of the product inside the computer would facilitate automation at various stages of the life cycle of the product.This lead to the use of computers as a symbolic modelling tool instead of its earlier use as sophisticated drafting tool. Methods of storing full 3D descriptions of complex solids were investigated.The need for a complete definition of shapes and the availability of increasingly cheap computing power lead to the birth of the new branch of computer graphics called 'Solid Modelling'.

1.2 SOLID MODELLING

Requicha's [3] description of solid modelling can be treated as a definition. It states **"The term solid modelling encompasses an emerging body of theory, techniques and**

systems focussed on informationally complete representations of solids - representations that permit (at least in principle) any well defined property of any represented object to be calculated automatically". These representations are meant to be understood by computer programs. The goal of the original developers of solid modelling systems was to use them to provide unified descriptions of parts and assemblies in an integrated design and manufacturing context. The data used or entered repeatedly at the synthesis, visualisation and analysis stages of design and also those used during manufacture and use, are identical and could be used in all stages if the information were captured in a suitable, general form at the start, thus saving time and money. The 3D modelling capability has become not only a powerful conceptual design tool and a potential communication medium between design and other functions, on which successful product creation increasingly depends, but also a very bread-and-butter talent for by-passing the need for much prototype making [4]. The first generation of solid modelling systems appeared in the mid 60's to mid 70's. Of these the pioneering work was done by Braid [47] and Baumgart[64]. These systems varied in the range of shapes that they could handle, in the ways the constituent shapes were combined and in their methods of storing information. Chapter 2 describe a survey on these various representation schemes and their merits and demerits.

1.3 SKETCHING INPUT FOR COMPUTER AIDED ENGINEERING

From the time Sutherland [2] developed the program 'sketchpad' the methods of inputting details and commands have changed and become considerably sophisticated. Presently inputs are accepted through the keyboard, mouse and digitizer or tablet driven menus. Drop-down menus and pop-up menus make the system very much user friendly. But none of them really enhanced 'sketching', the process with which the design engineer expands his thinking. Also with all the advantages of a single complete representation offered by the present solid modelling systems, computer is not used as a symbolic modelling tool as widely as a sophisticated drafting tool. This is because present day solid modelling demands reassessment of the current thinking process and needs a new thinking process, typically in terms of primitives and boolean operations. This newer process is essential only for inputting the solids, though the method may also be used in the final representations. The design engineer on the other hand starts his thinking with a pencil and paper and expands it by making sketches of his design. For him " **the design process begins with a graphical description of a proposed device or system to satisfy a human need. He perceives his idea at first not in the perfection of a well-turned english word description, nor in the precision of a mathematical formula, but in some nebulous assembly of building blocks of structure, vaguely beheld. The sketch forms the natural bridge between these vague stirrings of the imagination and the subsequent**

precise statements of the refined detail of the concepts" [5]. His inputs are lines of approximate lengths and not solids. Thus there remains a need for a system that can build a solid model with the design engineer's thinking process. Such a system will permit engineers to use solid modelling from design through manufacture and use.

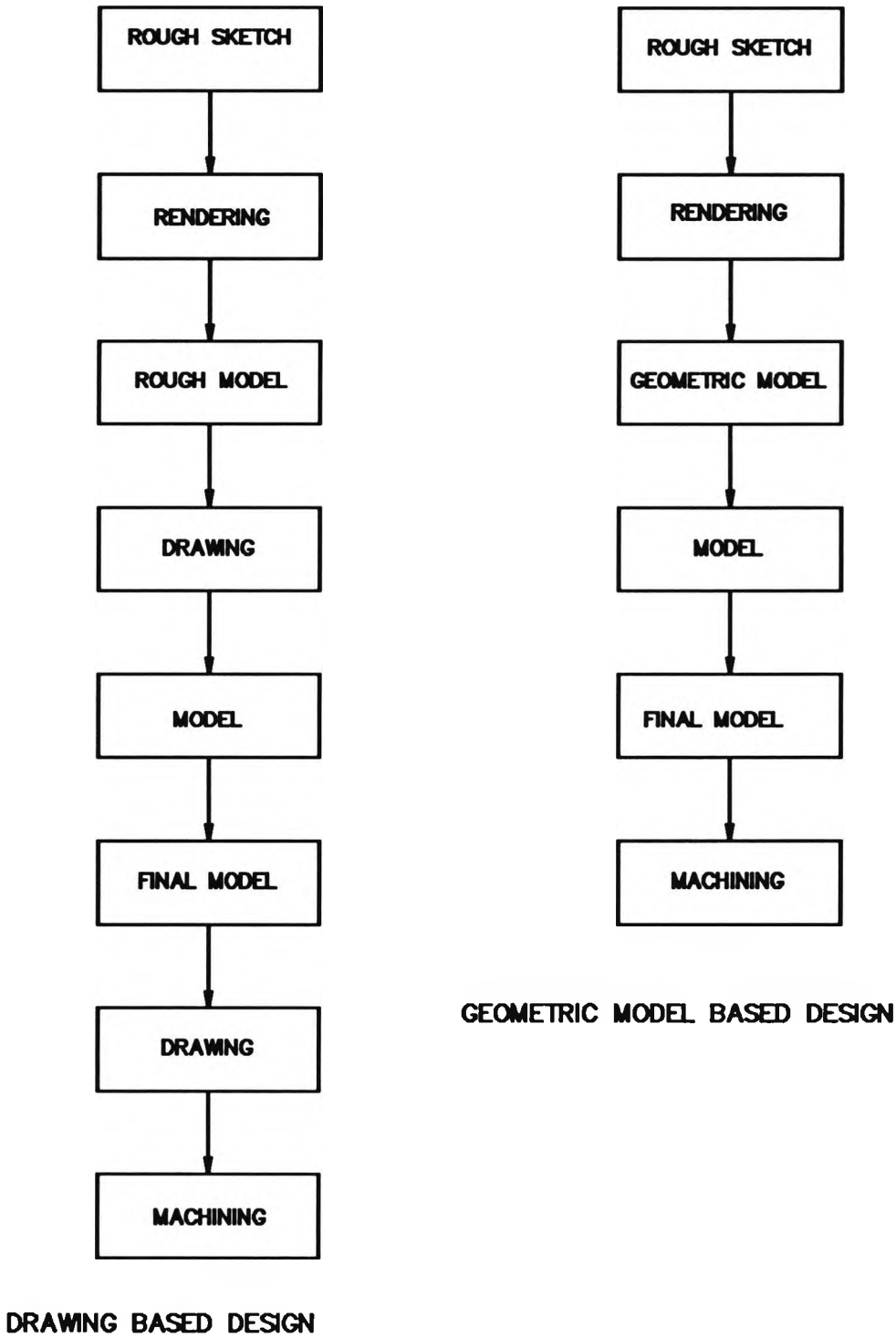

FIGURE 1.1
CURRENT DESIGN APPROACHES

1.3.1 SKETCHING AND THE DESIGN PROCESS

Pictorial sketching is the process of making a three dimensional view of the system or component under consideration. The tools used to construct a sketch in the traditional way are a pencil, paper and a rubber. A good sketch will be (a) accurate in proportion (b) correct in representation and (c) adequate in size. Sketches used by designers can be grouped into two classes as follows:

(i) Incomplete rapidly drawn sketches used for communicating directly with themselves or someone else.

(ii) More complete carefully drawn sketches (including base lines, centre lines and construction lines) prepared for brain storming and further development.

In communicating and extending the thoughts and in the application of the iterative design process, sketching plays a very important part. Traditionally a fundamental gauge of a competent designer is the ease with which he makes a free hand sketch. As the design develops, sketches undergo constant changes due to the emergence of different ideas. An eraser may be in constant use and new starts may be made repeatedly. Sketching should be done as easily and as freely as writing so that the mind is always centred on the idea and not on the sketching.

The points described in the preceding paragraphs establish that sketches are part and parcel of the design process. Further a representation of the three dimensional object is necessary for the man machine team to work efficiently.

1.3.2 THE PROPOSED SYSTEM - WHY AND WHAT

The present practice of building a solid model is for the designer to make his preliminary sketch in paper, then break off and feed the details of the solid into the computer through the solid modelling system. Chiyokura [41] describes the drawing based and geometric model based design systems in the form of a flow chart which is given in figure 1.1. It is evident from this description, that solid modelling is brought to the scene only after the design is fixed. This does not make use of all the potential of the computer in the man machine team and expects a newer thinking from man.

The system described in this thesis attempts to rectify these shortcomings and enables the designer to draw a complete isometric sketch directly into a digitizer connected to a computer. This sketch is written into a file as co-ordinates of points, over which the stylus has passed, separately for the visible lines, hidden lines, centre-lines, construction lines and erased lines. The computer then reads this file of points (i) breaks it into line segments (ii) fits the best possible curve for the segments (iii) identify the loops in the sketch and (iv) interactively builds the solid model. Once this

complete representation is established it could be used to compute geometric properties.

1.4 STRUCTURE OF THE THESIS

Chapter 2 describes a survey on solid modelling. In the process it explains what is solid modelling, the various representation schemes and the application of these schemes. It also extracts the basic features of solid modelling and analyses the various schemes in the light of sketching input. It draws inferences to be used in the development of the software 'SKETCH-SOLID'.

Chapter 3 describes the background theory used to develop the sketching input concept. It broadly falls into three groups namely (a) fitting methods for 2-dimensional lines and curves (b) 2-dimensional and 3-dimensional geometry and (c) co-ordinate transformations from 2-dimensions to 3-dimensions. Also it describes isometric sketching.

Chapter 4 describes the requirement analysis and functional specification of the task set and performed. It destinguishes between the various requirements, needs and wants and finally produces the technical specification of the program 'SKETCH-SOLID'. In the process it describes the data structures used in the program.

Chapter 5 is the explanation of the top level structure diagrams of the programs 'SKETCH-SOLID' together with their associated data structures and functions. It explains the algorithms of these programs and how they are translated into data structures and functions.

Chapter 6 presents and discusses sample sessions explaining the actions of the program at various stages of execution.

Chapter 7 presents discussion, conclusion and highlights areas for future work.

CHAPTER 2

A SURVEY ON SOLID MODELLING

2.0 GENERAL

Engineers build models to observe, analyse, understand and explain an object in an easy way. Physical models are expensive to produce and are limited in scope and traditionally engineering drawings, with rules and conventions for representing dimensions, tolerances, surface finishes, materials and the like played the role of a general purpose model. With one or several of the two dimensional views it conveyed the three dimensional information to a trained engineer. With the introduction of computer as a member of the design team, a model that could be understood by man and computer was needed. This computer model was expected to contain data stored in computer files, which can be used to perform the tasks which are traditionally performed by drawings and to contain additional facilities which make the best use of the computing power to facilitate the design process.

Thus the fundamental philosophy of 'Solid Modelling' is to provide unified description of parts and assemblies in an integrated, design, manufacture and use environment. These descriptions can then be accessed by programs, to read, interpret and compute other geometric properties. Research on 'Solid Modelling' originated from several stimuli. To quote from Jared [6] "**One, following from graphics and description of complex surfaces, was the need for a means of describing complex objects in order to generate realistic images of solid objects for visualisation and simulation purposes. Another was the desire to develop an 'aid to think in 3D' using the approach of building up complex solids by the combination of simple ones such as the rectangular blocks and cylinders**". Whatever the origin of the solid modelling system it has certain fundamental features. They transform the information about the solid they represent into data, store and retrieve them, perform operations on them and finally transform them back to processed information which could be understood by humans. Thus there are four requirements made on the solid modelling system. They are

1. An input system
2. A method of translating the input into representations
3. A set of algorithms to use these representations to compute properties
4. A set of algorithms to produce required outputs to aid man, the other member of the CAD team

Without these four elements the solid modelling system would be of little use.

2.1 EXPLANATION OF REPRESENTATION SCHEMES

Requicha [3] organises representations using the theory of functions as illustrated in figure 2.1.

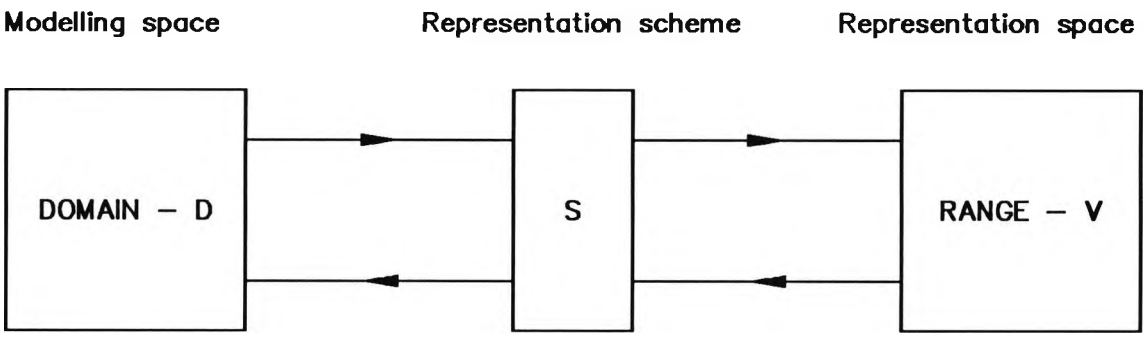

FIGURE 2.1
EXPLANATION OF REPRESENTATION

The modelling space 'M' contains all the solids that are representable and the transformed or mapped space 'R' contains all the representations. The representation 'S' is defined as the relation S:M→R. The domain 'D' of 'S' is the set of elements of 'M' that may be represented via the representation scheme of 'S'. The range 'V' of 'S' is the set of syntactically correct representations of the images of the elements of 'D'. The members of 'V' are valid representations which is an important requirement of the representation scheme, if it is to be used in automation at various stages of the life cycle of the product. If the representation 'r' in 'V' corresponds to a single object in the domain 'D' the representation is unambiguous. The representation is unique if the corresponding object does not admit representations other than 'r' in the scheme. Thus an unambiguous and unique representation scheme establishes a one-to-one correspondence between its domain and range. Also a representation is invalid if it does not correspond to any solid, a valid representation is ambiguous if it corresponds to several solids and a solid has non-unique representations if it can be represented in several ways in the scheme. Thus it can be seen that representations which are unambiguous and unique are highly desirable because they are one-to-one mappings.

2.2 REPRESENTATION SCHEMES FOR CAE

Six representation schemes are identified by Requicha [3] as unambiguous and suitable for computer aided engineering. They are

1 Pure primitive instancing
2 Spatial occupancy enumeration
3 Cell decomposition
4 Sweep representation
5 Constructive Solid Geometry or CSG

These six representation schemes will be described in the following sub-sections.

2.2.1 PURE PRIMITIVE INSTANCING

The 'Group Technology' concept in manufacturing identified objects with similar features as members of a family and each member is identified by specifying few parameters. In a similar fashion in pure primitive instancing objects are represented as families of solids containing common features and individual members are identified by specifying parameters. Each object family is called 'generic primitive' and the individual objects are called 'primitive instances'. Each primitive instance is represented by a fixed length tuple and therefore is easy to use. Most of the information is built into the system and thus the building is difficult and resource consuming and needs specialist knowledge of the family. A family of industrial blowers or gear wheels are typical examples of this scheme. These schemes are suitable for stand alone CAD packages such as, one for the design of gear wheels.

Pure primitive instancing schemes are unambiguous, unique and easy to use. Primitive instancing is used as auxiliary representations in multiple representation schemes, in order to ease the description of often needed parts. This concept may be developed further and libraries of specialist components such as cams, gears, aerofoils and the like could be developed in association with other schemes.

2.2.2 SPATIAL OCCUPANCY ENUMERATION

In spatial occupancy enumeration schemes, three dimensional space is divided up on a regular grid pattern such as blocks, and those which fall inside the volume occupied by an object are marked [6]. Each cell may be represented by the coordinates of a single point (a three tuple), such as the centroid or the body centre. A scanning order is imposed and the corresponding ordered sets of three tuples called 'spatial arrays' are established for the objects represented in this scheme.

Spatial arrays are unambiguous and unique but are potentially bulky. A variation on this representation is the 'Octree Decomposition'. In octree decomposition, the subdivision of the heterogeneous cells continue recursively into eight sub-cubes until a maximum resolution is reached while the decomposition of homogeneous cells cease. The testing of the cells for homogenity is called the leaf node criteria (since a homogeneous cell is a leaf in the tree) and this kind of decomposition is said to be structured in a hierarchical manner. Samet and Weber [7] illustrates this with a good

example which is reproduced in figure 2.2. It illustrates (a) the example 3D object (b) its octree decomposition and (c) its tree representation. In an algebraic notation this could be expressed as (1 1 1 1 (1 1 1 1 0 0 0 0) 0 1 1) where '1' represents a filled cell and '0' represents an empty cell.

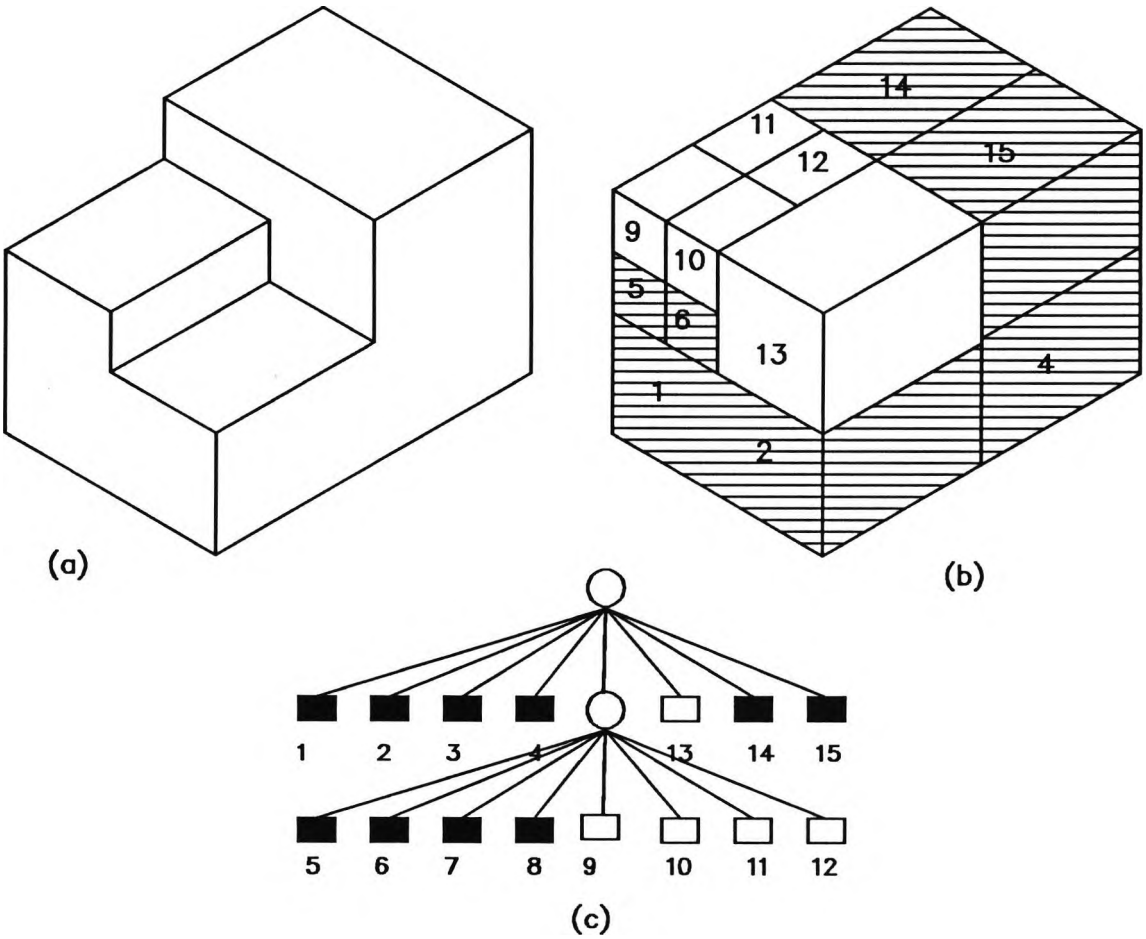

FIGURE 2.2
ILLUSTRATION OF OCTREE DECOMPOSITION

The advantages and disadvantages of the octree representations could be enumerated as follows [8]:

(a) Any arbitrary shape objects, convex and concave with interior holes can be represented to the precision of the smallest cell

(b) Geometrical properties such as, surface area, centre of mass and interference are easily calculated at different levels of precision.

(c) Because of the spatial sorting and uniformity of the representation, operations on octree are efficient.

On the disadvantages it could be said that

(a) The bounding surface is an approximation by square polygons
(b) Objects with complex details require a larger number of cells
(c) Regardless of the number of sub-divisions it is an approximation.

Octree is a widely used representation and has several algorithms and applications using it [9-11]. The ways and means of storing the tree structure (the data structures) and their use are described in many references [7,9,11,12].

The octree encoding while having its advantages and limitations is suitable only for a known solid. In the design by sketching situation the solid will be known only at the end of the design process. This means the participation of the computer can begin only at the end of the design process. Thus octree encoding is not an ideal candidate for the design by sketching situation. However it is a valuable auxiliary representation for B-rep and CSG representations.

2.2.3 CELL DECOMPOSITION

Cell decomposition is similar in concept to spatial occupancy enumeration above, but, unlike the subdivision of the space in occupancy enumeration, in this method the solid itself is subdivided into disjoined polyhedra which may be of arbitrary size and shape.

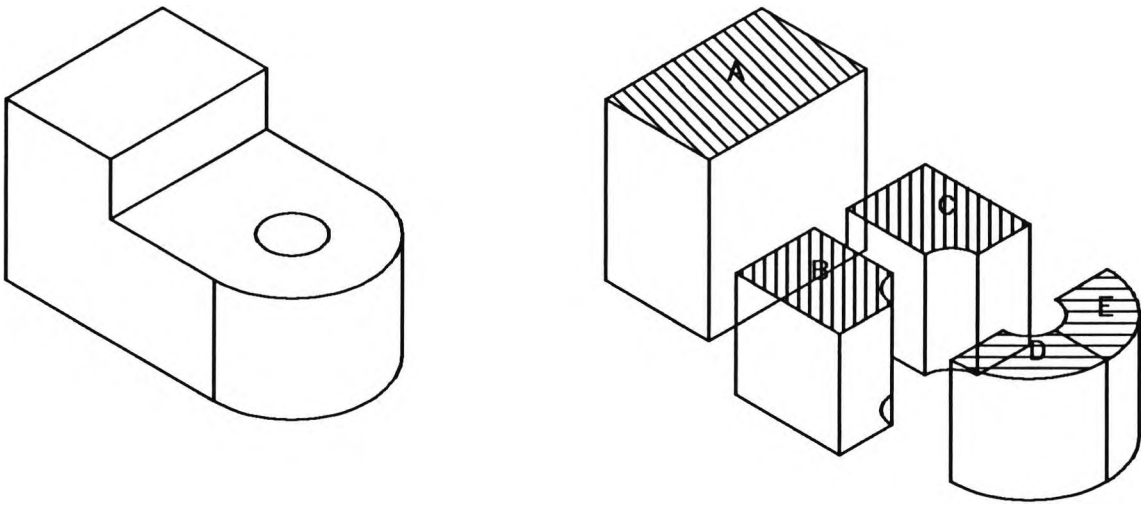

FIGURE 2.3
ILLUSTRATION OF CELLULAR DECOMPOSITION

A solid is represented by decomposing it into cells and representing each cell in the

decomposition. This object space representation is very useful in the identification of the visible subset of the object using the notion of bounding object. When determining whether or not an object is visible, it is easy to surround the object with a bounding box or sphere and say that the object is not visible if the surrounding box or sphere is not visible. Figure 2.3 illustrates the cellular decomposition.

2.2.4 SWEEP REPRESENTATION

Sweep representations are constituted with two kinds of entities namely 'moving object' and 'trajectory'. For example the cylinder in figure 2.4 can be represented by the circle 'A' and the line 'B' where 'A' is the 'moving object' and 'B' is the 'trajectory'. This representation is a translational sweep representation since the moving object translates along the trajectory. The same object could be formed by a rotational sweep representation as shown in figure 2.5.

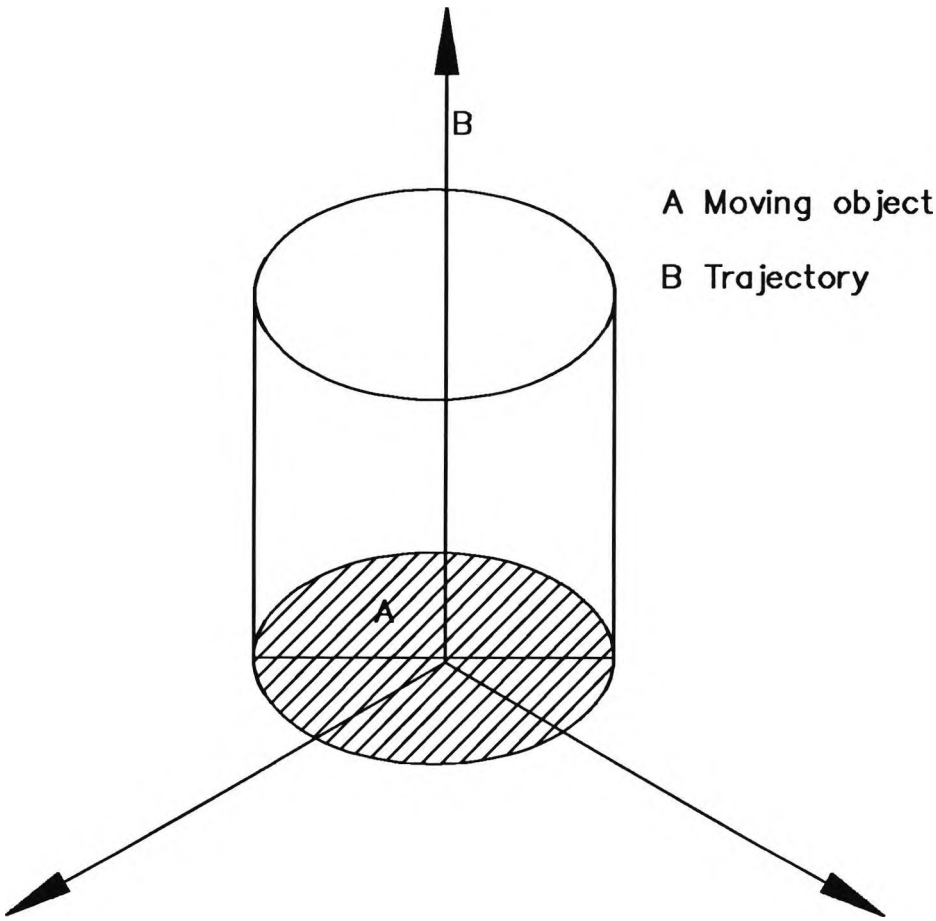

FIGURE 2.4
TRANSLATIONAL SWEEP

The sweep representation method though is a good scheme in its own right is not used in contemporary solid modelling systems as a representation scheme. Instead it is

used as a method of inputting solids, in particular solids with sculptured surfaces.

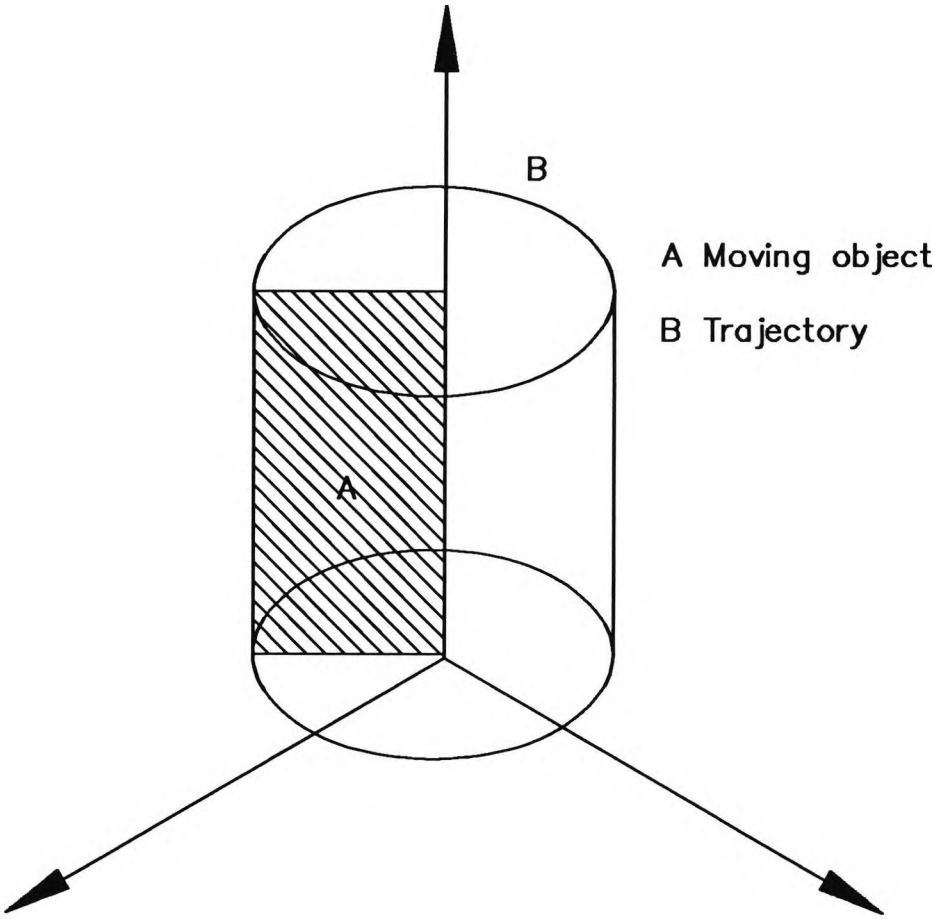

FIGURE 2.5
ROTATIONAL SWEEP

2.2.5 CONSTRUCTIVE SOLID GEOMETRY

The philosophy of CSG is developed from the concept that a complicated solid can be represented as various ordered additions and subtractions of simpler solids by means of the boolean operations, union, intersection and difference. CSG represents solids as a collection of primitives such as blocks, cylinders, cones and spheres. The primitives are stored in a binary tree together with boolean set operations defining the way the primitives are combined. Each node represents an intermediate solid, which is a combination of primitive and intermediate solids lower in the tree. The root node represents the entire solid [7].

Consider the 'L' slider shown in figure 2.6 (a). This could be built by differencing the

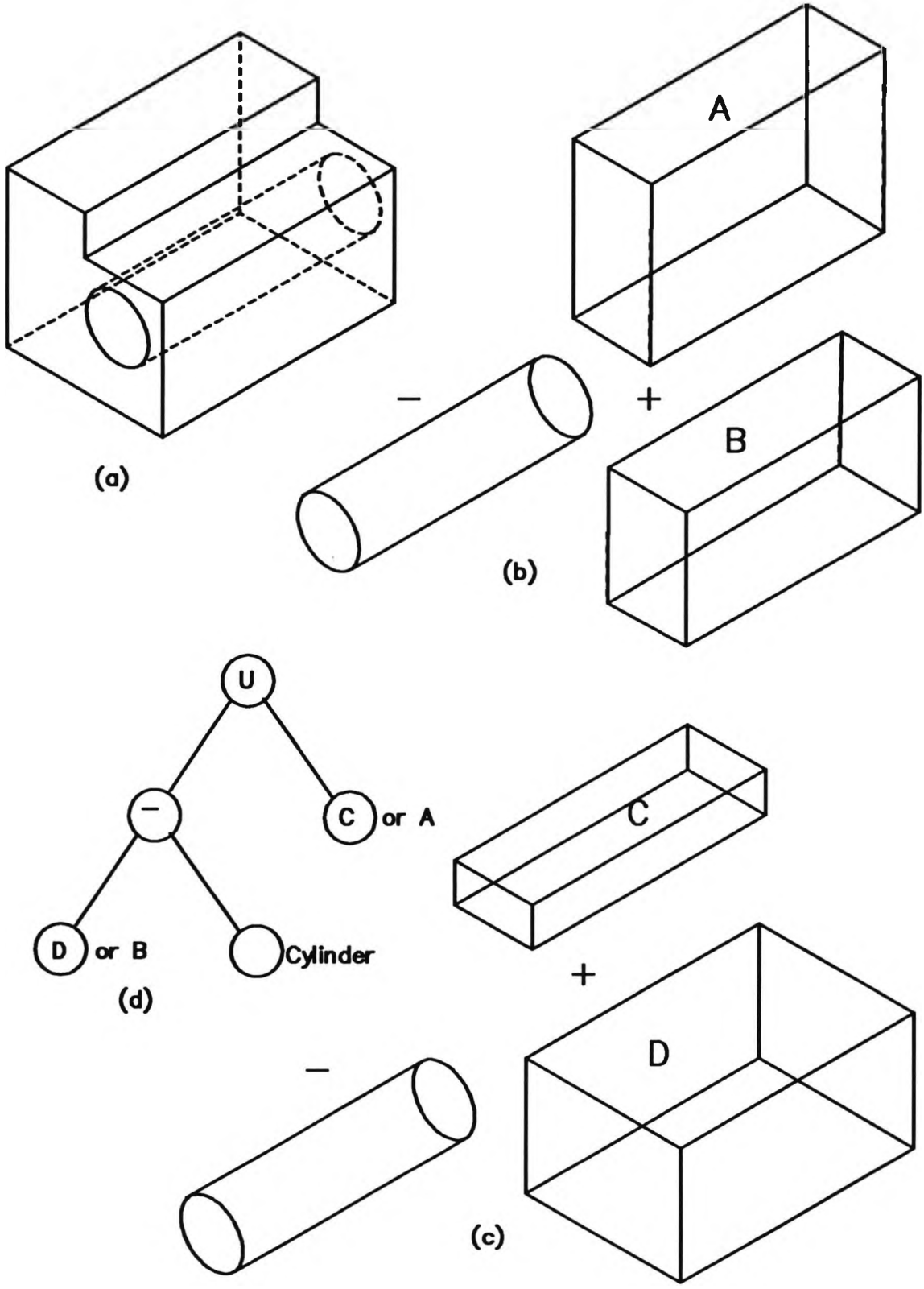

FIGURE 2.6

ILLUSTRATION OF CONSTRUCTIVE SOLID GEOMETRY

cylinder from block B and adding it to block A as shown in figure 2.6 (b). Alternately it could be produced by subtracting the cylinder from block D and adding it to block C as shown in figure 2.6 (c). Thus it can be seen that the same object could be represented by more than one way in CSG. This means CSG representation is not unique. Its domain depends on the primitives the system can handle and on the operations available. When the primitives are valid, the CSG scheme guarantees the validity of the representation. CSG representaions are not efficient sources of geometric data for producing line drawings of objects. However CSG has many algorithms developed for different applications and there is a number of references explaining it [13 - 30]. Figure 2.6 (d) shows the CSG tree or the construction tree of the ' L' slider. The CSG scheme is the most compact of all known class of commonly machined parts [31].

When some operation such as the calculation of mass has to be performed there should be an algorithm to evaluate the tree. Such an algorithm, called the geometry property function, solves a series of problems for the primitives and the intermediate solids until the problem is solved for the entire tree. This indicates that for each of the various applications there should be an algorithm traversing the tree. In CSG described so far there still remains the need for modelling the building blocks. Sometimes they are held in boundary representations. Alternately they may be defined as boolean combination of the directed surfaces in which their faces lie. The CSG description whilst compact and easily deduced from the user's input, is not convenient, for producing pictures. It must first be transformed into a boundary model, a model that is logically equivalent to the surface 3D wireframe model. For this reason some solid modellers have both CSG and boundary representations.

In short CSG defines a few primitives and use their boolean combinations to build the complex solid, storing the history or path of building the final model in the form of the CSG tree. Special functions are then used to traverse the CSG tree and compute the properties. This method though has its advantages is not suitable for the sketching input.

2.2.6 BOUNDARY REPRESENTATION B-rep

The philosophy of boundary representation is developed from the concept that a body could be represented by a manifold which is a surface that divides the space into exactly two regions, inside and outside. The manifold could then be broken into pieces with boundaries and thus could be described as a union of bounded surfaces. Thus in boundary representation, solid boundaries are represented as union of faces, with each face represented in terms of its boundary (usually a union of edges) together with the data defining the surfaces in which the face lies. Boundary representation is akin to the wireframe model which represents an object by the edge curves and their

end points on their surfaces. In B-rep method the 'L' slider of figure 2.6 could be represented by the faces shown in figure 2.7. Boolean operations are not part of the representation of B-rep model, but they are often employed as one of the means of creating or manipulating a model. The effect of a boolean operation on a CSG model is an addition to the CSG data structure. But since B-rep systems require explicit

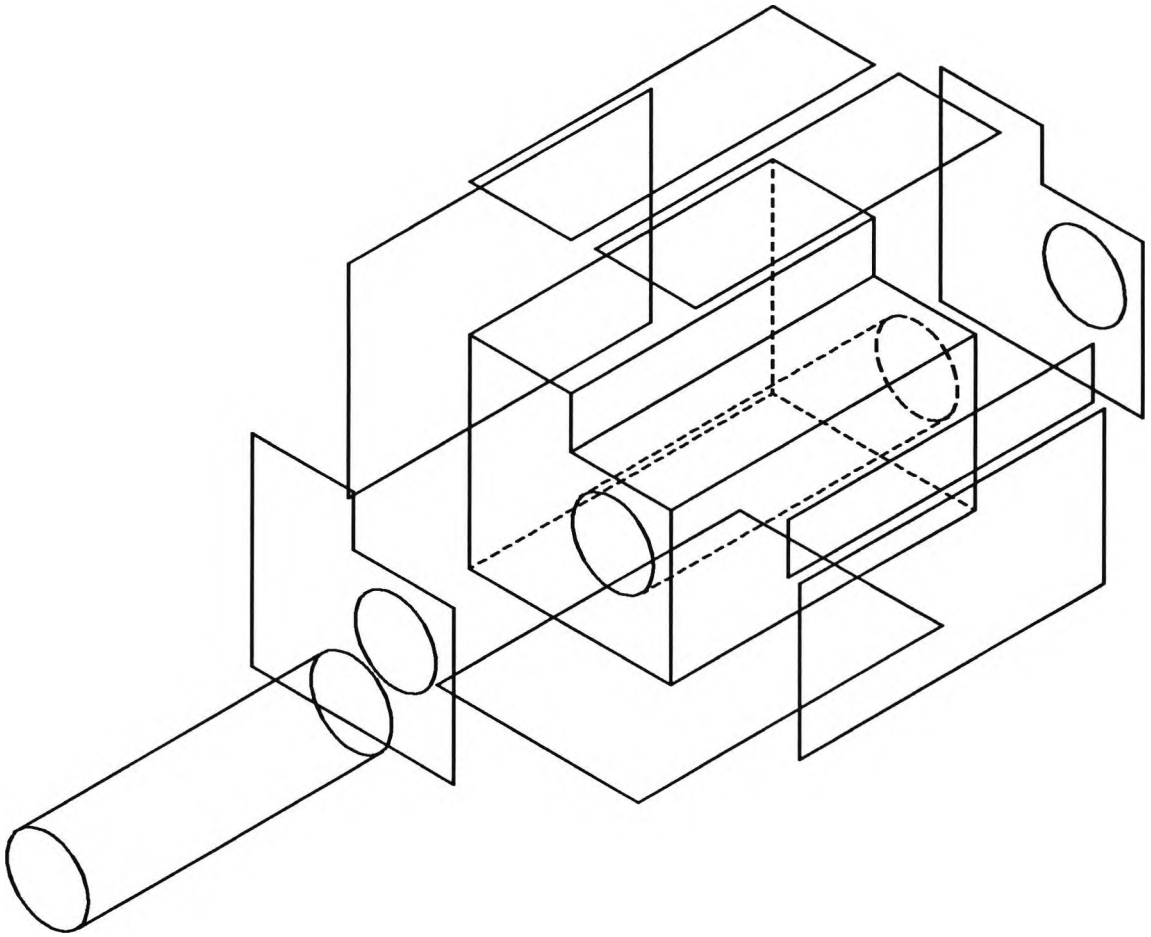

FIGURE 2.7
ILLUSTRATION OF BOUNDARY REPRESENTATION

representations of the boundaries of the solid, they must calculate a new boundary that results from the operation. The information stored in a B-rep model is of two

kinds [32]. They are (i) the links which store the topological information about the connection between faces, edges and vertices and (ii) the real numbers required to specify surface and curve equations and coordinates of the vertices. These informations are called 'topological information' and 'geometrical information' respectively.

An assembly in B-rep model could be represented in the following way. An assembly is constituted of instances. Instances are positioned bodies and hence have a transformation matrix and a body associated with each of them. The body is constituted of shells and shells are constituted by faces. Faces contain one or more loops in each of them. The loops are made of edges and edges are defined by vertices. Figure 2.8 shows Braid's [32] boundary model indicating the topological and geometrical information.

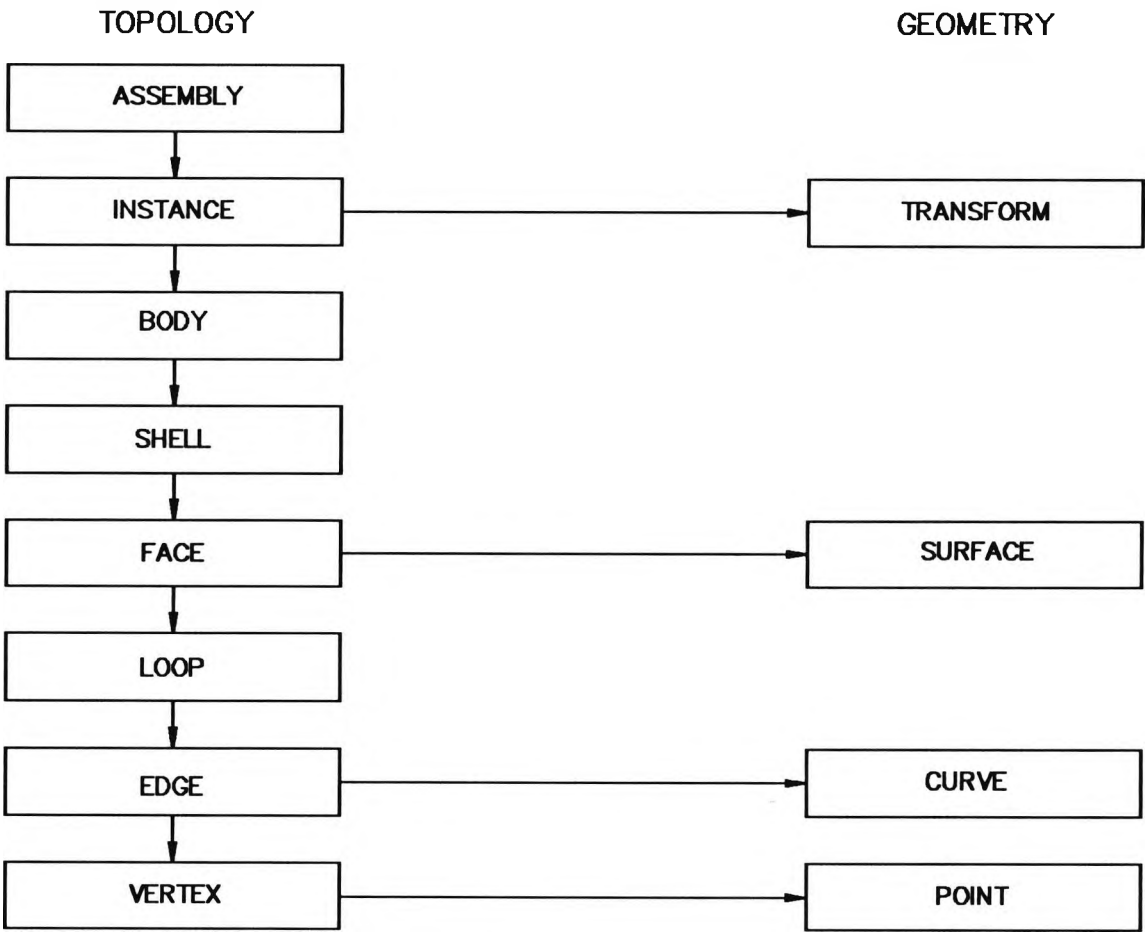

FIGURE 2.8
BRAID'S BOUNDARY MODEL

The B-rep model is more close to the line drawings than any other system. As a result there are many research works and references associated with it [31 - 50]. This method

is more suitable for the sketching input than any one of the other models. It is easy to tailor this model to suit the thinking process of the engineer.

2.3 ANALYSIS OF SOLID MODELLING SCHEMES

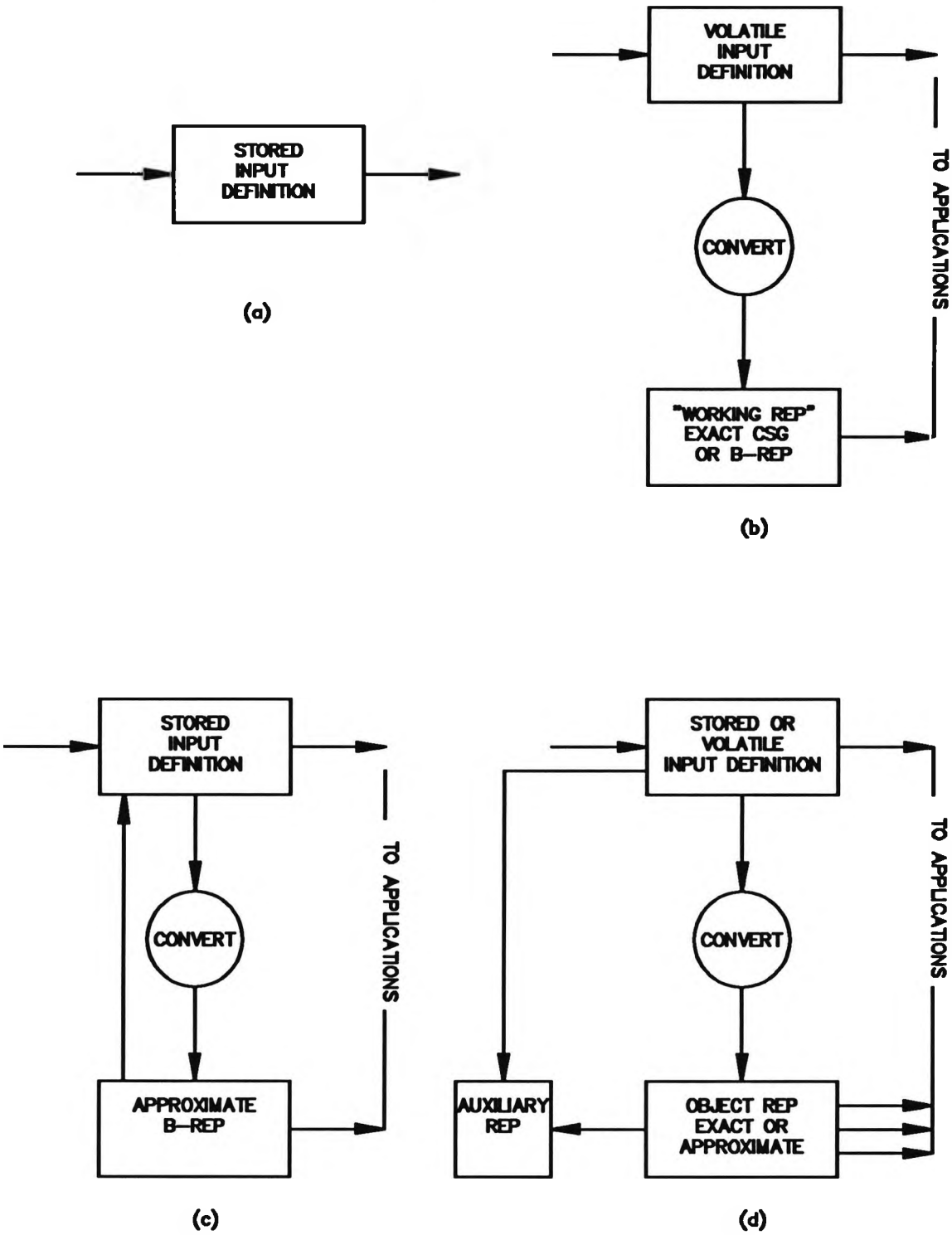

FIGURE 2.9

ANALYSIS OF SOLID MODELLING SYSTEMS

It is seen at the beginning of this chapter that there are four requirements on solid modelling systems namely (i) an input system (ii) a method of translating these inputs into representations (iii) a set of algorithms using these representations to compute geometric properties and (iv) a set of algorithms to produce outputs. As regard to requirement (ii) above six representation schemes were analysed and the B-rep method is found to be more suitable for sketching input. Requicha and Voelcker [51], Baer et al [45] and Braid [44] all looked at the contemporary solid modelling systems at various time points. Almost all of them used primitives and boolean operations as their input method. Thus it can now be said that a solid modeller as now is a 3D data structure synthesiser which transforms the user description of a complex solid into an internal representation with a set of geometrical algorithms, performing operations on simpler solids such as blocks, cylinders and cones. In their analysis in 1982 Requicha and Voelcker [51] concluded that all system representations fell into two categories namely (i) B-rep and (ii) dual (B-rep and CSG). In a later evaluation in 1983 the same authors [52] looked at the then contemporary systems and concluded that there are many auxiliary representations. This is due to the increasing demands of the geometric algorithms and the increasing availability of the memory of the computer. This lead to the analysis of the input and the system as a whole. They grouped this input definitions and representations into four classes. They are

(i) Stored input definitions
(ii) Volatile input definitions
(iii) Stored input definitions with approximate representations
(iv) Stored or volatile input definitions together with auxiliary representations

These methods are illustrated in a digram, which is reproduced here as figure 2.9. Figure (a) here is the simplest method which stores the inputs to be processed by the geometry property functions. Figure (b) represents the first version of a serious system where the inputs are translated into some useful representation and then the inputs are discarded. Figure (c) represents a particular application of approximate B-rep as the main model of representation and any fine details are obtained from the inputs which are stored. In figure (d) an exact or approximate representation together with an auxiliary representation enhancing the algorithms is represented. These lead to the generalised notion of a solid modelling system as illustrated by figure 2.10. This suggests two main developments. They are (i) the number of auxiliary representations increases as the number of applications increases and (ii) as computer technology advances more and more memory becomes available at relatively low cost and removes the necessity for compactness at the expense of runtime efficiency.

These suggest that the current trend can be described in the following way.
(a) The input is only to make the computer understand the solid in development.

(b) Once the input is understood it can be stored as a representation or can be ignored.

(c) Further representations are developed by the computer to store the solid in a way understandable and quickly accessible by some or all of the algorithms.

(d) It is common practice to have many representations for one solid to enhance the runtime efficiency.

These facts suggest that sketching input can coexist with any other input system and could form a representation in its own right. Additional representations such as the B-rep and the octree representation could be developed at a later stage and can form the main or auxiliary representation schemes. In this thesis the B-rep model is made as the only representation and thus makes the sketching a volatile input.

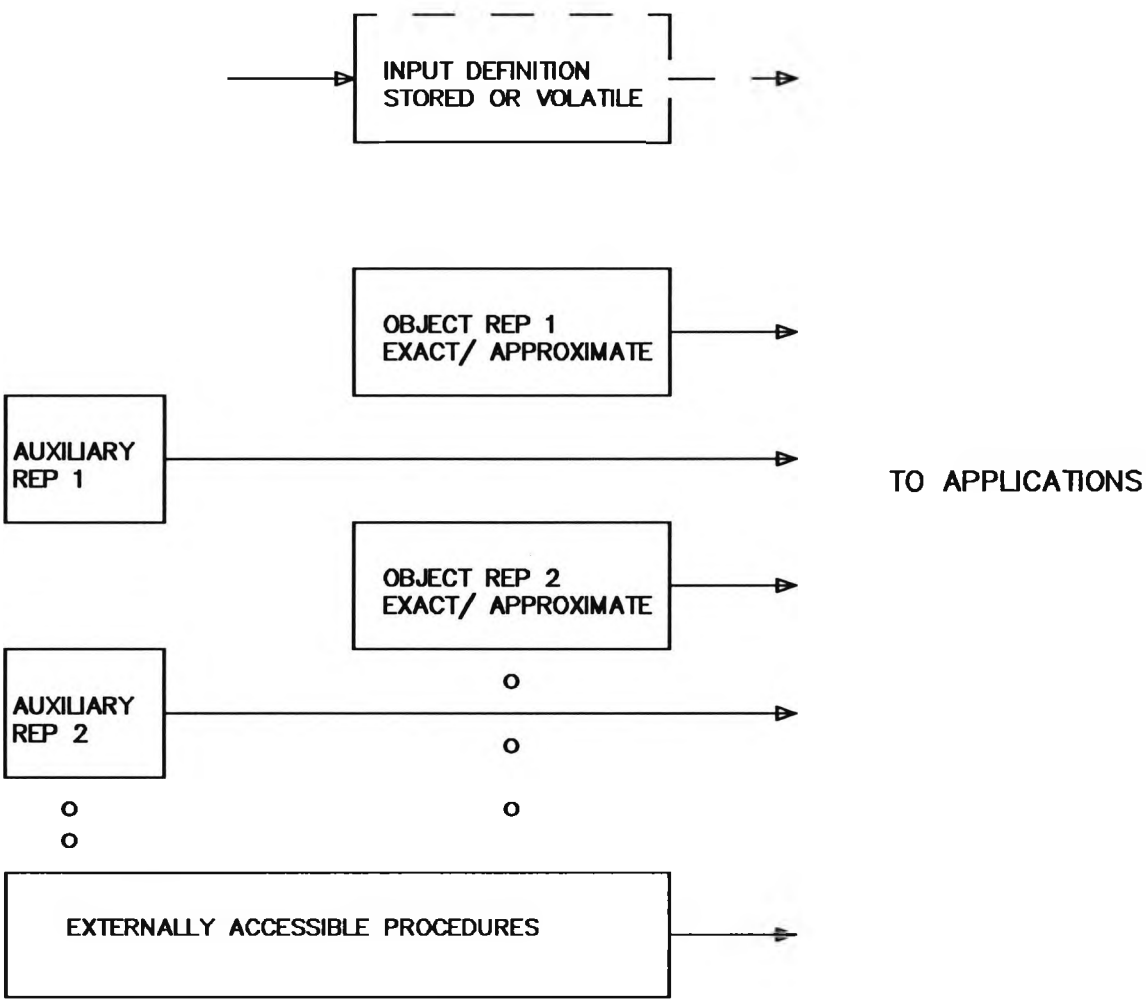

FIGURE 2.10

GENERALISED NOTION OF SOLID MODELLING SYSTEMS

As mentioned earlier the major advantage of CSG is that it guarantees that valid models are always produced and ensuring validity is a serious issue for the B-rep model. However Mantyla [42] shows that the same type of guarantee could be provided for boundary representation modellers if the underlying modelling process is based on the use of Euler operators.

2.3.1 EULER OPERATORS

An object can at most have six topological constituents namely shells (s), holes (h), faces (f), rings or internal loops (r), edges (e) and vertices (v). All physical objects obey the Euler-Poincare formula

$$v-e+f = 2(s-h) + r.$$

The problem of creating a non-realizable or invalid object in boundary models is circumvented in the following way.

In an algebraic analysis solid modelling can be represented by a six dimensional space with v, e, f, h, r and s as the axes. In the development of a boundary model operations are performed to create or add one or more of these six coordinates and any solid is a point P (v,e,f,h,r,s) in this space. The Euler-Poincare formula represents a five dimensional hyper plane of this space and all points in this plane represents valid models. This hyper plane can be spanned by five, six-vectors V_i lying on the hyper-plane and are linearly independent. Once these V_i's are established all points of the hyperplane can be expressed as linear combinations of the V_i's. This means that if a solid model is built with operations representing these V_i's or base vectors, it will be valid. Operations denoting these base vectors are called Euler operations and operators performing them are called Euler operators. There can of course be an infinite number of such five, six-vectors but Mantyla [42] used the operators MVFS, MEV, MEF, KEMR and KFMRH which are described in the following sub-sections. Their names are coined using the historical convention using the names given below.

Operations			Entities		
M	-	Make	V	-	Vertex
K	-	Kill	E	-	Edge
S	-	Split	F	-	Face
J	-	Join	S	-	Solid
			H	-	Hole
			R	-	Ring

Three basic type of manipulative operations are used in creating the five Euler operators mentioned above. They are (i) 'Prototype' primitives that create skeletal models (ii) Local topological operations that sub-divide the sequences of a face or a vertex and (iii) Global topological operations that implement a connected sum of two polygons.

2.3.1.1 SKELETAL PRIMITIVES MVFS AND KVFS

The operator MVFS creates from scratch an instance of the data structure of a solid that has just one face and one vertex. Hence the new face has one empty loop with no edges at all. KVFS the inverse of MVFS destroys the skeletal instance of the data structure identical to that created by MVFS. These operators are called the skeletal operators because they deal with the basic solid from the scratch. Every time a solid is created or destroyed these operators are used.

2.3.1.2 LOCAL MANIPULATORS MEV, KEV, MEF, KEF, KEMR AND MEKR

These operators are called local because they confine to a loop or edge rather than dealing with the solid as a whole. MEV subdivides the cycle of edges of a vertex into two cycles by 'splitting' a vertex into two vertices, joined with a new edge. The net effect of this is to add one vertex and one edge to the data structure, as suggested by the name of the operator.Mantyla [42] included the application of this operator to the cases of (i) 'lone' vertices, by considering the result of subdividing a vertex with no edges at all, as consisting of two vertices joined with an edge and (ii)a vertex joined with the old vertex by means of a new edge.

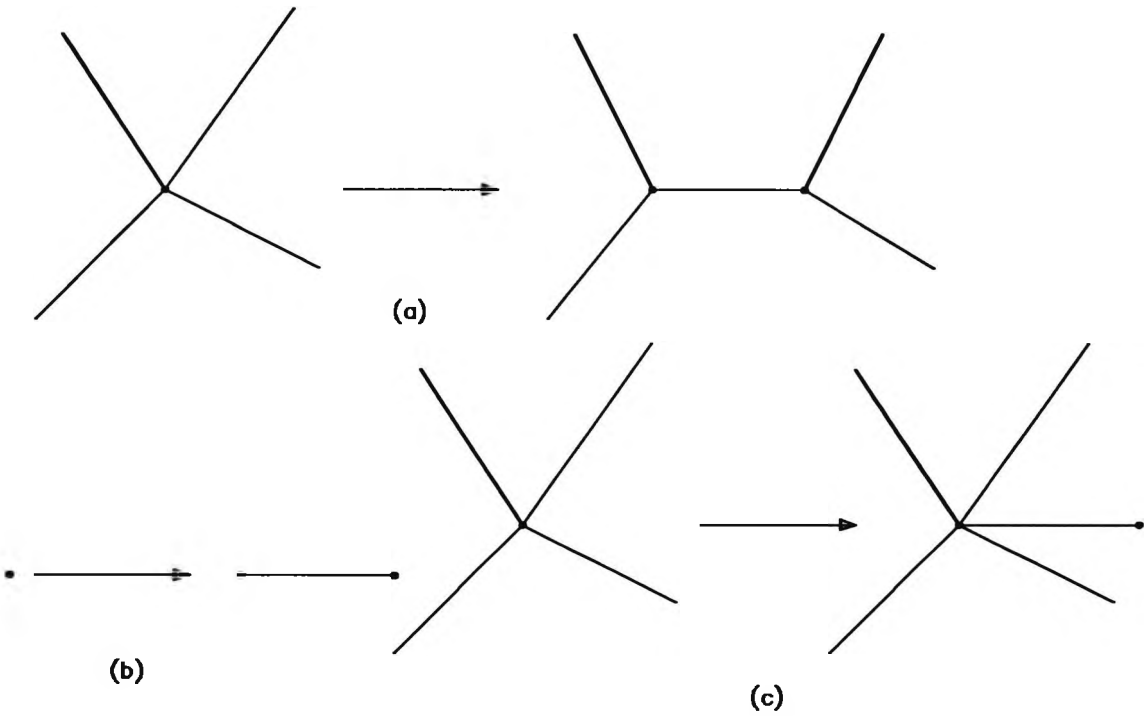

FIGURE 2.11

MEV OPERATOR AT DIFFERENT CONDITIONS

Figure 2.11 shows the situations handled by this operator. The inverse operator KEV is capable of undoing any of the three cases of figure 2.11. Given an edge connecting two distinct vertices KEV can remove the edge, collapse the vertices into one and merge their edge cycles.

In a similar fashion the operator MEF subdivides a loop by joining two vertices with a new edge. Its net effect is to add one new face and edge to the data structure. Figure 2.12 shows the three cases supported by the operator MEF.

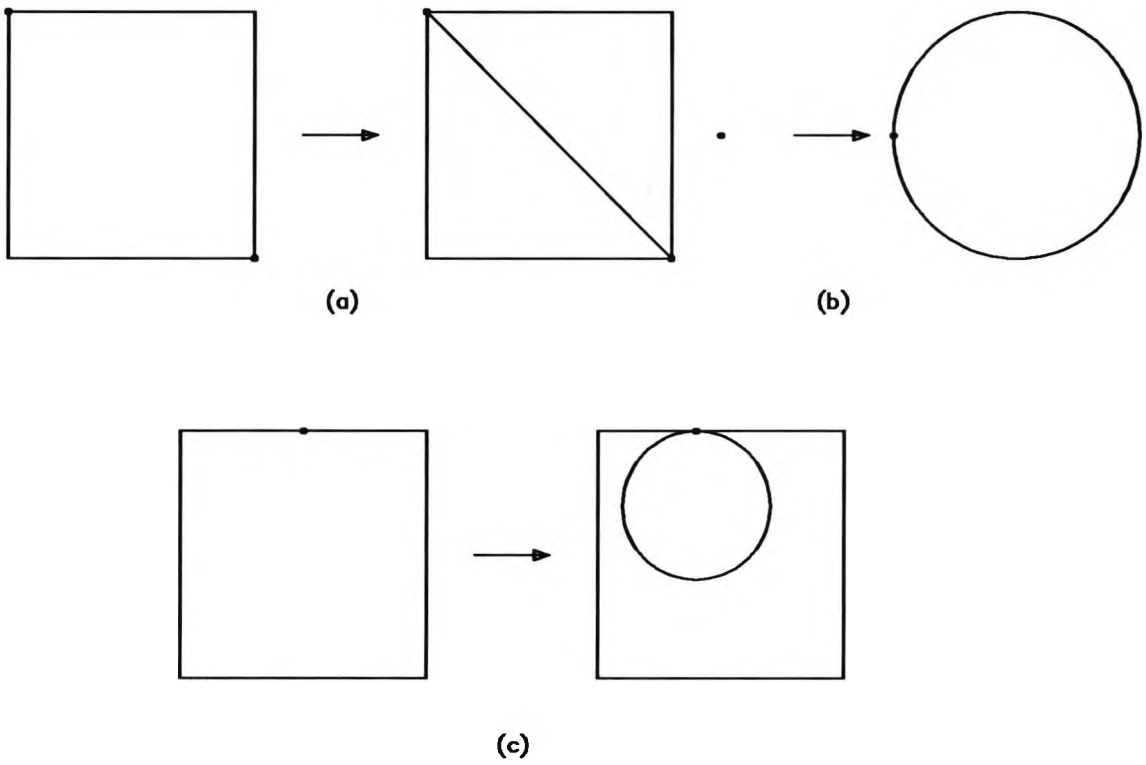

FIGURE 2.12
MEF OPERATOR AT DIFFERENT CONDITIONS

The inverse operator KEF can destroy the effect of MEF in each of the cases in figure 2.12. Given an edge adjacent to two distinct faces, KEF is capable of removing the edge and joining the two faces into one whose bounding loop is the result of merging the two original boundaries.

The operator KEMR splits a loop into two new ones by removing an edge that appears twice in it. It divides a connected bounding curve of a face into two bounding curves,

and has the net effect of removing one edge and adding one ring to the data structure.

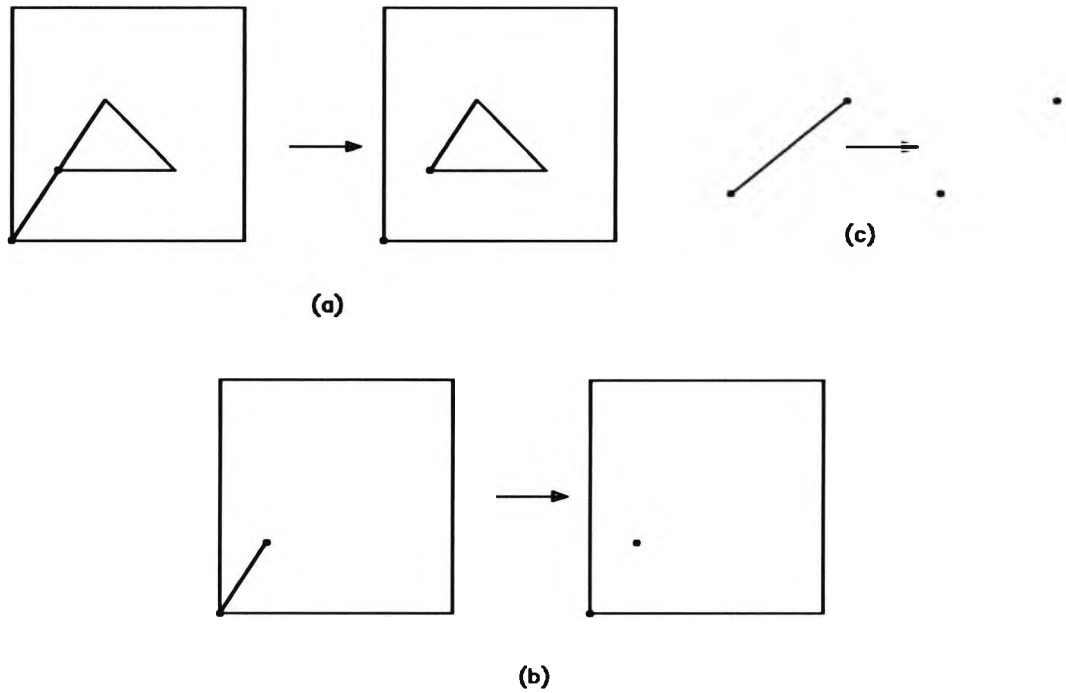

FIGURE 2.13
KEMR OPERATOR AT DIFFERENT CONDITIONS

Figure 2.13 illustrates the three cases of the operation by this operator. The inverse operator MEKR can merge two loops of a face by joining one vertex of each with a new edge.

2.3.1.3 GLOBAL MANIPULATORS KFMRH AND MFKRH

This operator is meant to modify the global topological properties such as dividing a solid into two or adding a hole. Given two loops say f_1 and f_2, KFMRH joins them into one face by transforming the bounding loop of f_2 to a ring of f_1. Hence its net effect is to remove one face f_2, and add one ring instead. KFMRH has no effect on the local arrangement of edges and vertices of its arguments. It is truly a global manipulation. KFMRH creates a hole only when the argument faces belong to the same shell. When the faces belong to different shell it combines them into one shell. The inverse operator MFKRH is capable of reversing the effects of KFMRH. It modifies the ring of a face into the bounding loop of a new face.

2.3.2 CONSEQUENCES OF EULER OPERATORS

As mentioned earlier the Euler operators are represented by the base vectors and

the corresponding base vectors are as follows:

EULER OPERATOR	BASE VECTOR
	v e f h r s
MEV	1 1 0 0 0 0
MEF	0 1 1 0 0 0
MVFS	1 0 1 0 0 1
KEMR	0 -1 0 0 1 0
KFMRH	0 0 -1 1 1 0

The inverse operators are

KEV	-1 -1 0 0 0 0
KEF	0 -1 -1 0 0 0
KVFS	-1 0 -1 0 0 -1
MEKR	0 1 0 0 -1 0
MFKRH	0 0 1 -1 -1 0

These base vectors indicate the addition (1) or deletion (-1) of the element representing the column. Mantyla created a matrix which is termed in this thesis an Euler matrix using these five vectors and the Euler-Poincare formula. The matrix has an inverse and it is used to calculate the number of each Euler operation represented by the rows, necessary to produce an object with the number of elements represented by the vector P(v,e,f,h,r,s), a point in the hyperplane. These numbers of operations required are called the Euler coordinates. Thus

$$P\ (\text{Euler matrix})^{-1} = (\text{Euler coordinate vector})^{T}$$

Thus for instance, an object with 16 vertices, 24 edges, 10 faces, 1 hole, 2 rings, and 1 shell could be represented by a point P(16 24 10 1 2 1) in the hyperplane.

$$P\ (\text{Euler matrix})^{-1} = (15\ 10\ 1\ 1\ 1\ 0)^{T}$$

This means that to model this solid a total of 15 MEV's, 10 MEF's, 1 MVFS, 1 KEMR and 1 KFMRH are needed. The last component taking the value 0 ensures that the solid satisfies the Euler- Poincare equation. If the values are negative their inverse operation should be used.

This is a useful result of Euler operators. In this research, the sketch after perfection is transformed into a vertex model in three dimensions and the vertex model is then transformed into an edge based boundary model. Euler operators could then be used to perform this transformation so that validity is ensured. Consider now the example of the 'L' slider and let the labelling of the edges, faces and vertices be as shown in figure 2.14.

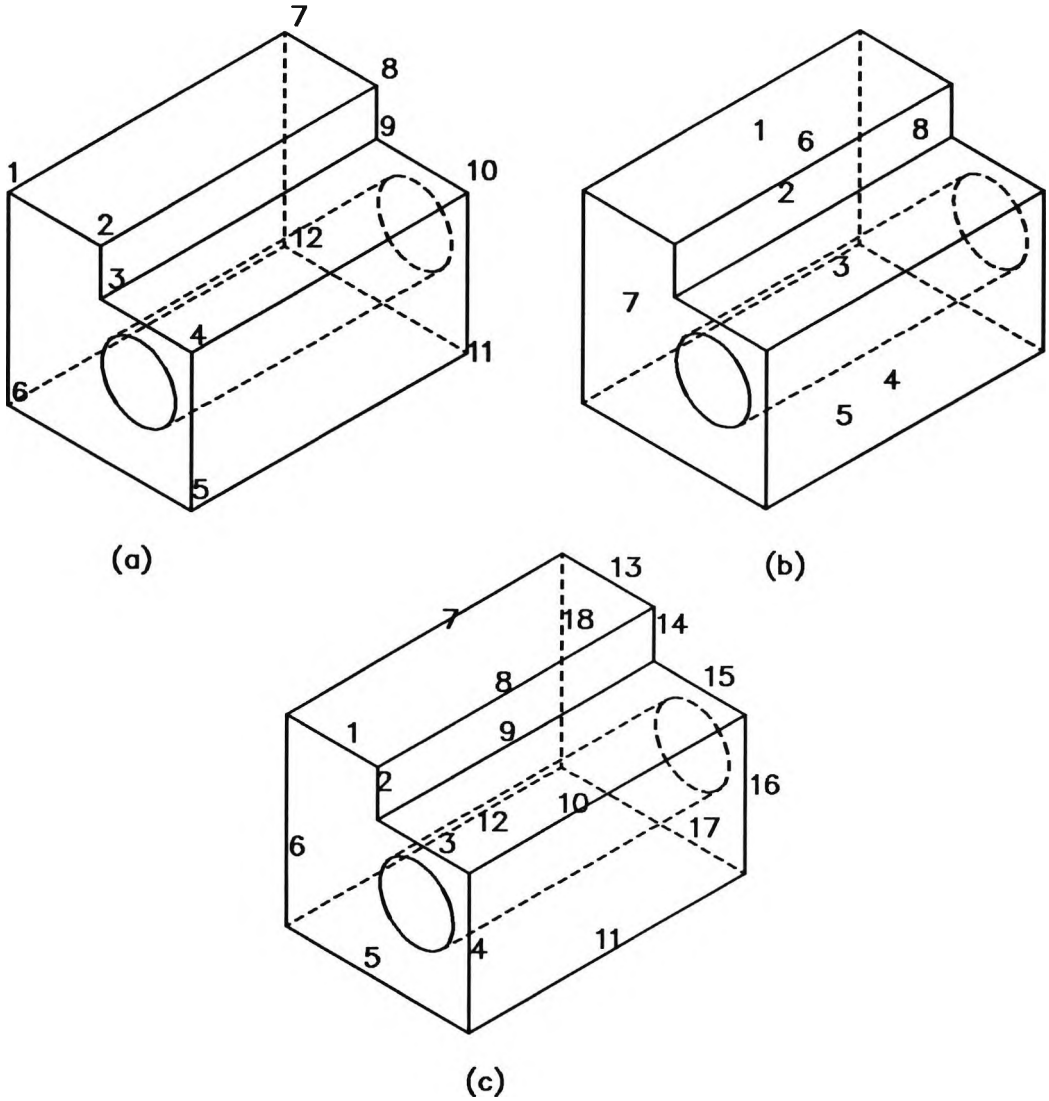

FIGURE 2.14
LABELLING OF THE 'L' BLOCK

Applying the Euler formula depends on the way the cylinder is represented. The cylinder could be represented as two ellipses one connecting line and one curved surface as shown in figure 2.15(a). Alternately it could be represented as four elliptical arcs, two straight edges, two curved surfaces and four vertices as shown in figure 2.15(b). For case (a) $v = 14$, $e = 21$, $f = 9$, $h = 1$, $r = 2$ and $s = 1$ which makes the Euler formula $14 - 21 + 9 = 2(1-1) + 2$ which is true. For case (b) $v = 16$, $e = 24$, $f = 10$, $h = 1$, $r = 2$ and $s = 1$ which makes the Euler formula $16 - 24 + 10 = 2(1 - 1) + 2$ which is also true. This leaves the representation of the cylinder to the choice of the application.

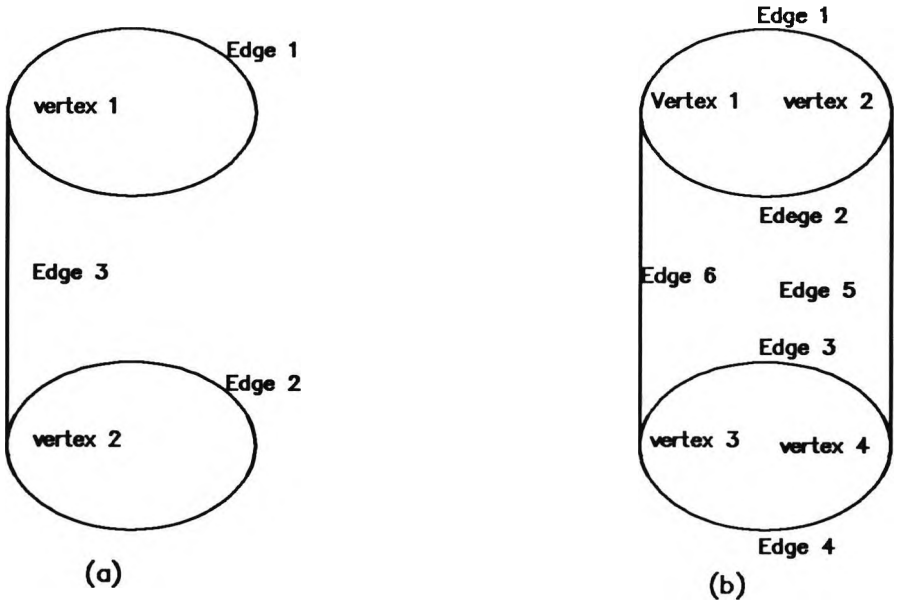

FIGURE 2.15
REPRESENTATION OF CYLINDER

2.4 SOLID MODELLING REVISITED

The preceding sections described the historical development of solid modelling. There, solid modelling is seen as the branch of geometric modelling that emphasises the general applicability of models, and insists on creating 'Complete' representations of physical solid objects. The representations should be adequate for answering arbitary geometric questions algorithmically. Also they revealed the following observations.

(i) Initially objects are described by the user using the description language available in the solid modeller. Once entered these object descriptions are translated to create the actual internal representations stored by the modeller.
(ii) The representation between the description language and the internal representation need not be a direct one. Internal representations can employ modelling concepts different to the original description.
(iii) A solid modeller may well include several description languages intended for different kinds of users and applications.
(iv) None of the three major approaches to solid modelling namely (a) decomposition models (b) constructive solid geometry models and (c) boundary models is far superior to others in all respects.
(v) Decomposition models are superior to others as sources of data for numerical

algorithms even when they are approximate. Constructive models are the most concise of the three major approaches. Boundary models are directly useful for graphical applications.

(vi) A hybrid modeller is capable of supporting several co-existing solid representations and tries to pick the most suitable of them for each task. A hybrid model can well include several types of boundary models and conversion algorithms form a fundamental part of the hybrid modeller.

From these observations it could be said that though, most of the solid modelling systems have the primitives and their boolean combinations as the method of input, they can have 'sketching input' as an additional input method and there could be multiple representations. One of the representations may be the sketch after the improvements. This would facilitate reproduction of the graphic outputs.

2.4.1 SKETCHING INPUT - WHERE?

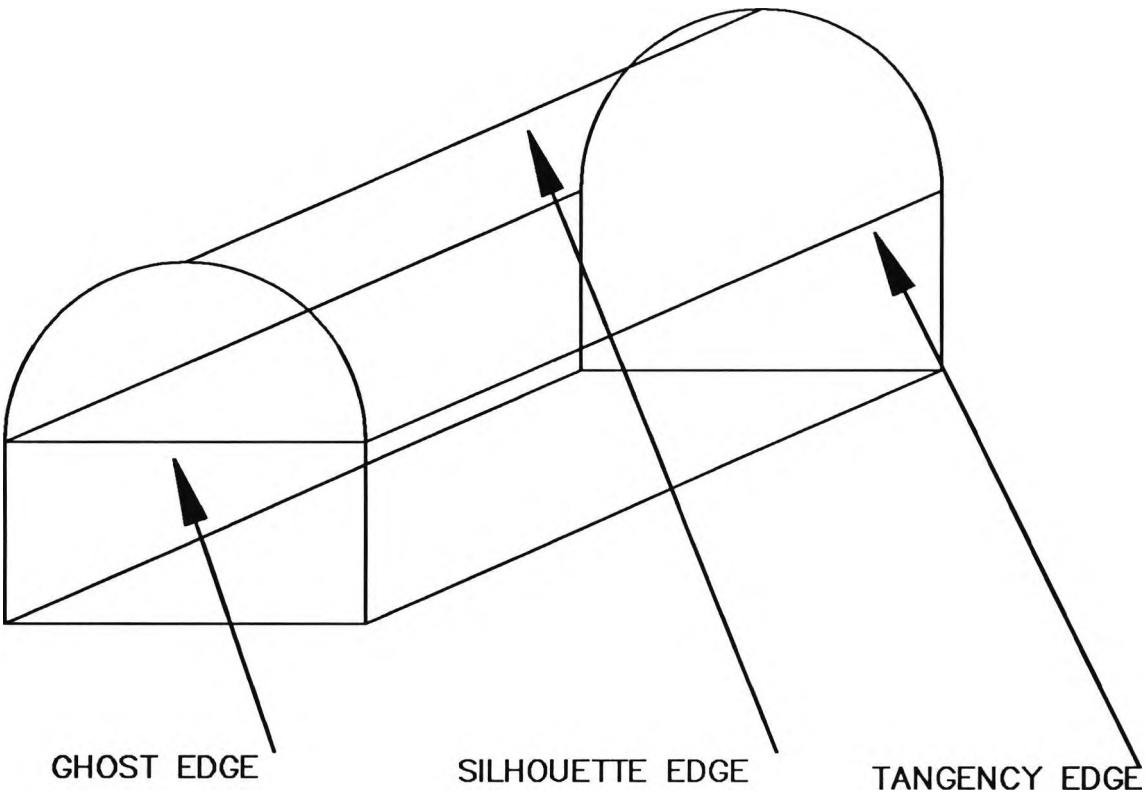

FIGURE 2.16
ILLUSTRATION OF SPECIAL EDGES

Having thus seen that sketching input could be used as one of the multiple representations it is proper to look at the contemporary solid modelling systems to find out

the state of the art with regard to sketching input. Several researchers [56-62] have contributed towards this direcion. The following paragraphs enumerate the salient points of these publications.

A wireframe model of a solid could be described as the lists of vertices and edges in the boundary of the solid without the surface description and the loop organisation. Laquette [56] describes three special edges namely (i) ghost edge (ii) tangency edge and (iii) silhouette edge. A ghost edge is an edge drawn on a surface with two adjacent faces lying on the same surface. The two faces can be merged into one by eliminating the ghost edge. Normally ghost edges are not allowed in wireframe models. A tangency edge separates two faces on two different surfaces, but the surfaces have a first order continuity along the tangency edge. Traditional drafting views do not draw these edges. A silhouette edge is a notion associated with a view of the solid. It is the locus on the surface, of points where the surface normal is perpendicular to the projection direction. The projection of the silhouette edge is the outline of the surface. Laquette [56] illustrates these edges with a figure which is reproduced here as figure 2.16.

It is seen in section 2.3.2 that representing the cylinder as 2 vertices and 3 edges or 4 vertces and 6 edges is a matter for the application. This is because of the varying number of silhouette edges introduced. As seen in the definitions by Laquette silhouette edges define the surfaces and hence are decided according to convenience, though they are important in the definition of the solid by sketching. In a similar analysis it could be seen that tangency edges are important in the definition of the solid by sketching input, since they form the boundary of a face. Thus it could be said that drawing tangency edges in sketching input must be made compulsory. Also the definition of the silhouette edges must be tailored and made compulsory to suit the sketching environment.

Fukui [57] proposed a sketching input system to a boundary solid with planar faces in the following way. The user draws projection lines of a 3D polyhedron on a 2D plane face by face. Then the system projects them inversely into 3D space immediately after each face is drawn and constructs a data structure with a planar face equation. The method connects these faces by the 'adjacent edge' principle and builds the solid model.

Hodes [58] describes a method of processing line drawings. Processing here means the identification of lines and vertices in a drawing. The input to the process is a line drawing represented by a pattern of marked points in a 100 x 100 square array. The actions performed in this method are as follows:
(i) Fit straight line of standard length.
(ii) Advance in the best direction and monitor for the number of points in the line.
(iii) Look for conditions of a vertex by looking at both sides for intersecting lines.

(iv) Continue following a line until a vertex is hit.
(v) Repeat (i) to (iv) until all the lines are exhausted.

The paper does not explain in detail whether this is used for identifying polygons or curves. Also it was too brief to explain the underlying theory and methodology. However it brought to the surface the 'change-in-slope' concept to identify the vertex.

Guzman [59] explains how a two dimensional picture of a 3 dimensional scene could be transformed into solid representation. Murase and Wakahara [60] explains a matching method for the hand sketched figure recognition. This may be very useful where the figures to be processed are known in advance. However this is not suitable for sketching input for solid modelling. Richards and Onwubolu [61] describe a method of automatic interpretation of engineering drawings for 3D surface representation. The steps in this program may be listed as follows:

(i) Establish the number of entities (lines, arcs) in each view
ii) Considering the reference line to lie along with the axis establish the x,y,z coordinates of the vertices
(iii) Match the entities to identify the substructures.
(iv) Formulate Bezier net for the identified substructures.

Hwang and Ullman [62] describes a system where the sketch made on a digitizer tablet is accepted as an input to the solid model. In this system, the sketching action is considered as a sequence of strokes, where each stroke is from initial contact of the drawing device with the surface (tablet) until it is removed. These strokes are then matched to the primitives namely line, arc, circle, ellipse, segmented line or polyline. An expert system is then employed to build the solid model. This though is a positive step in sketching input, still restricts the user.

2.6 DATA STRUCTURES FOR BOUNDARY REPRESENTATION

There are three topological elements namely (i) vertex (ii) edge and (iii) face and for the geometric data any one type is sufficient to represent the solid and any others could be derived from the one stored. A boundary data structure could be thought of as a set of adjacency relationships among topological elements. Baer et al [45] points out there are nine such relationships possible and they showed these relationships with a figure which is given here as figure 2.17.

To connect three entities at least two relationships are necessary. Mathematically there are $9C_2 = 36$ ways of selecting these two relationships. But it is not necessary to select only two relationships. One can decide to store any number of relationships between three and nine to store the topology of the solid. Thus in general there should

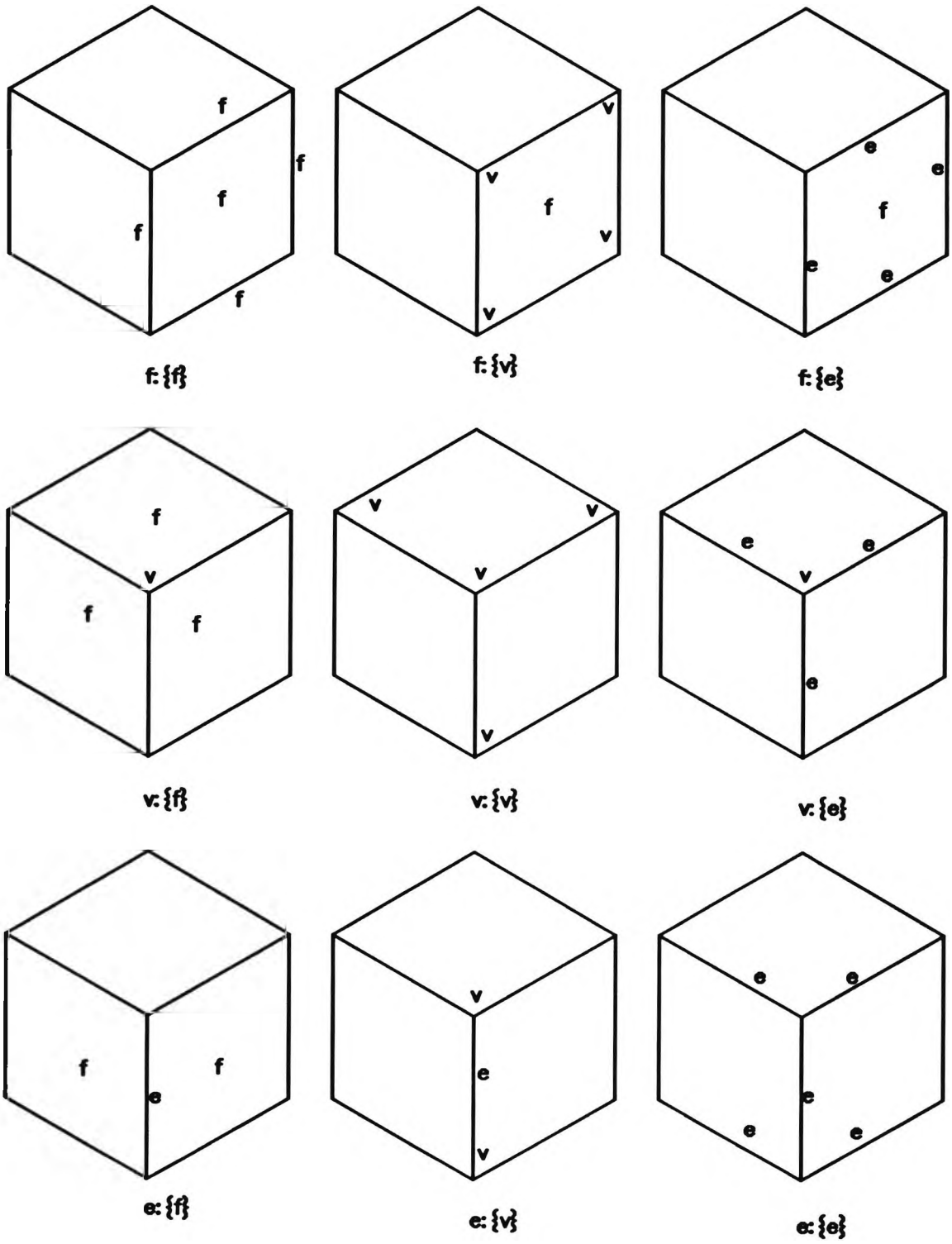

FIGURE 2.17
NINE TOPOLOGICAL RELATIONSHIPS

be

$$9C_2 + 9C_3 + 9C_4 + 9C_5 + 9C_6 + 9C_7 + 9C_8 + 9C_9 = 502$$

possible combinations of which some will be invalid due to disconnectedness. Woo [63] analysed these combinations and concludes that the symmetric structure is the best and the winged edge structure is the next best. Weiler [39] establishes four such data structures including the 'Winged-Edge' and 'HalfEdge' data structures, suitable for edge based representations. The oldest and complete of these representations is the 'Winged Edge' data structure by Baumgart [64].

2.6.1 VERTEX BASED BOUNDARY MODEL

In a vertex based boundary model the faces are expressed as a list of vertices in a consistent order (clockwise or anticlockwise). This consistency in order is useful in many algorithms and is used in the identification of all the loops in the solid sketched. In these representations most of the information is left implicit (to be computed whenever needed). For the 'L' block shown in figure 2.14 the vertex model has the following two lists.

Vertex list

Vertex	Co-ordinates
V_1	$x_1\ y_1\ z_1$
V_2	$x_2\ y_2\ z_2$
V_3	$x_3\ y_3\ z_3$
V_4	$x_4\ y_4\ z_4$
V_5	$x_5\ y_5\ z_5$
V_6	$x_6\ y_6\ z_6$
V_7	$x_7\ y_7\ z_7$
V_8	$x_8\ y_8\ z_8$
V_9	$x_9\ y_9\ z_9$
V_{10}	$x_{10}\ y_{10}\ z_{10}$
V_{11}	$x_{11}\ y_{11}\ z_{11}$
V_{12}	$x_{12}\ y_{12}\ z_{12}$

Face list

Face	Vertices
F_1	$V_1\ V_7\ V_8\ V_2$
F_2	$V_2\ V_8\ V_9\ V_3$
F_3	$V_3\ V_9\ V_{10}\ V_4$
F_4	$V_4\ V_{10}\ V_{11}\ V_5$
F_5	$V_6\ V_{12}\ V_{11}\ V_5$
F_6	$V_1\ V_7\ V_{12}\ V_6$
F_7	$V_1\ V_2\ V_3\ V_4\ V_5\ V_6$
F_8	$V_7\ V_8\ V_9\ V_{10}\ V_{11}\ V_{12}$

2.6.2 EDGE-BASED BOUNDARY MODELS

An edge based boundary model represents a face boundary in terms of a closing sequence of edges. The data structure imposes an orientation for each edge. Edge E_1 is considered to be oriented positively from vertex V_1 to vertex V_2. Again faces are oriented i.e. their edges are listed clockwise as viewed from the outside of the block. Each edge occurs in exactly two faces once in the positive and once in the negative orientation. An edge based boundary model is given below for the 'L' block in figure 2.14. It has three lists (a) Vertices and co-ordinates (b) edge details and (c) face

details. In addition to this the equations of the surfaces could be given for each solid.

Vertex	Co-ordinates	Edge	Vertices	Face	Vertices
V_1	$x_1\ y_1\ z_1$	E_1	$V_1\ V_2$	F_1	$E_1\ E_7\ E_{13}\ E_8$
V_2	$x_2\ y_2\ z_2$	E_2	$V_2\ V_3$	F_2	$E_2\ E_8\ E_{14}\ E_9$
V_3	$x_3\ y_3\ z_3$	E_3	$V_3\ V_4$	F_3	$E_3\ E_9\ E_{15}\ E_{10}$
V_4	$x_4\ y_4\ z_4$	E_4	$V_4\ V_5$	F_4	$E_4\ E_{10}\ E_{16}\ E_{11}$
V_5	$x_5\ y_5\ z_5$	E_5	$V_5\ V_6$	F_5	$E_5\ E_{12}\ E_{17}\ E_{11}$
V_6	$x_6\ y_6\ z_6$	E_6	$V_6\ V_1$	F_6	$E_6\ E_7\ E_{18}\ E_{12}$
V_7	$x_7\ y_7\ z_7$	E_7	$V_1\ V_7$	F_7	$E_6\ E_1\ E_2\ E_3\ E_4\ E_5$
V_8	$x_8\ y_8\ z_8$	E_8	$V_2\ V_8$	F_8	$E_{18}\ V_{13}\ E_{14}\ E_{15}\ E_{16}\ E_{17}$
V_9	$x_9\ y_9\ z_9$	E_9	$V_3\ V_9$		
V_{10}	$x_{10}\ y_{10}\ z_{10}$	E_{10}	$V_4\ V_{10}$		
		E_{11}	$V_5\ V_{11}$		
		E_{12}	$V_6\ V_{12}$		
		E_{13}	$V_7\ V_8$		
		E_{14}	$V_8\ V_9$		
		E_{15}	$V_9\ V_{10}$		
		E_{16}	$V_{10}\ V_{11}$		
		E_{17}	$V_{11}\ V_{12}$		
		E_{18}	$V_{12}\ V_7$		

These two models indicate that all the models have some relationships to all the three entities vertex, edge and face. Inclusion of explicit nodes for each of these connected entities gave birth to the winged-edge data structure.

2.6.3 WINGED-EDGE DATA STRUCTURE

Winged edge data structure makes use of the following properties.
(i) Because an edge appears in exactly two faces there are two other edges after this edge, one in each of these faces which are called the next edges.
(ii) Because of the same reason as above in (i) there are two other edges before this edge, one in each of them which are called the previous edges.
(iii) Because of the consistent orientation of the faces, the edges occur once in positive and once in negative orientation.
(iv) Because the information included in the data structure, each face could be identified by any one edge in it which is called the first edge [42].
(v) Again because of the information included in the data structure, the vertices could be identified by any one of the three edges forming it.

The orientation and information coupled together gives (i) next clockwise edges 'ncw' (ii) previous clockwise edges 'pcw' (iii) next counter clockwise edges 'nccw' (iv) previous counterclockwise edges 'pccw' (v) clockwise face 'fcw' (vi) counterclockwise

face 'fccw' (vii) starting vertex 'vstart' and (viii) ending vertex 'vend' as the elements of the data structure. Each of these elements is represented by a node in the winged-edge data structure. Figure 2.18 illustrates the winged-edge data structure.

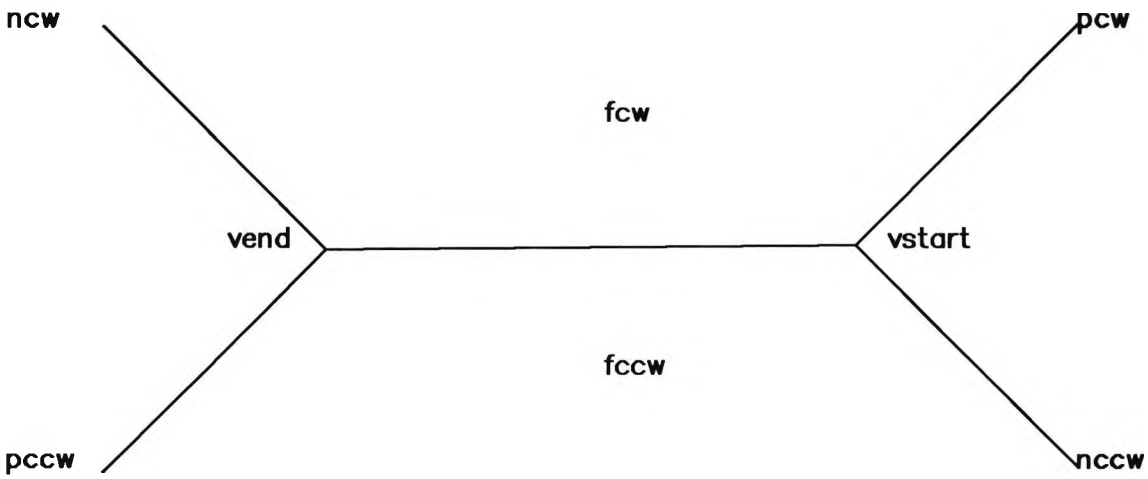

FIGURE 2.18

WINGED-EDGE DATA STRUCTURE

The complete winged edge data structure for the 'L' block of figure 2.14 is given below.

edge	vstart	vend	fcw	fccw	ncw	pcw	nccw	pccw
1	1	2	7	1	2	6	7	8
2	2	3	7	2	3	1	8	9
3	3	4	7	3	4	2	9	10
4	4	5	7	4	5	3	10	11
5	5	6	7	5	6	4	11	12
6	6	1	7	6	1	5	12	7
7	1	7	1	6	13	1	6	18
8	2	8	2	1	14	2	1	13
9	3	9	3	2	15	3	2	14
10	4	10	4	3	16	4	3	15
11	5	11	5	4	17	5	4	16
12	6	12	6	5	18	6	5	17
13	7	8	1	8	8	7	18	14
14	8	9	2	8	9	8	13	15
15	9	10	3	8	10	9	14	16
16	10	11	4	8	11	10	15	17
17	11	12	5	8	12	11	16	18
18	12	7	6	8	7	12	17	13

Vertex	Co-ordinates	first edge
V_1	$x_1\ y_1\ z_1$	E_1
V_2	$x_2\ y_2\ z_2$	E_2
V_3	$x_3\ y_3\ z_3$	E_3
V_4	$x_4\ y_4\ z_4$	E_4
V_5	$x_5\ y_5\ z_5$	E_5
V_6	$x_6\ y_6\ z_6$	E_6
V_7	$x_7\ y_7\ z_7$	E_7
V_8	$x_8\ y_8\ z_8$	E_8
V_9	$x_9\ y_9\ z_9$	E_9
V_{10}	$x_{10}\ y_{10}\ z_{10}$	E_{10}
V_{11}	$x_{11}\ y_{11}\ z_{11}$	E_{11}
V_{12}	$x_{12}\ y_{12}\ z_{12}$	E_{12}

In addition to this there is a list of faces which are represented by their first edges. This winged-edge structure described above deals with faces that do not have rings resulting from holes. But almost all mechanical components have rings in some of their faces. Therefore it becomes necessary to incorporate them. Braid et al [32] suggests a method to incorporate them as explained in figure 2.8. They use winged-edge data structure to represent the loop and a list of loops to represent a face. A similar approach is taken up by Chiyokura [41] and Mantyla [42].

2.7 STORING GEOMETRIC INFORMATION

It has been seen earlier that the constituents of a boundary model are (i) the topological connections of the three constituents edge, vertex and face (ii) co-ordinates of the vertices and (iii) the analytic equations of the various edges and faces. This section describes the method of storing the analytic equations described in (iii) above.

The class of surfaces covered in this research includes planes, cylinders, cones and spheres which cover the majority of those encountered in manufactured products. The approach of storing these information following Mantyla [42] is described below. Every surface is represented by (i) a tuple of parameters that describe it in some orientation and (ii) a transformation matrix that positions it at its correct position. Thus for the plane the tuple is [a b c d] representing the plane equation

$$[a\ b\ c\ d][x\ y\ z\ 1]^T = 0.$$

There is no necessity for a transformation matrix for the planes. Cylinders are represented as if their axes were the z-axis and their bottom faces lie on the 'xy' plane.

Parameters needed are the radius and the height of the cylinder. Cones could be stored in a similar manner to the cylinder. However to include the situation where the cone has the bottom and top radii greater than zero an additional parameter, the top radius, is included. Spheres are represented as if they were centred at the origin and the only parameter required is the radius.

Two kinds of curves are accommodated in this research. They are the straight lines and circular arcs. The straight lines need not have an explicit representation as the co-ordinates at the end points could readily supply them. As for the arcs, the radius, centre, starting angle, finishing angle and the plane in which they are lying should be known. However the parameters are assumed to be in the xy plane and the plane normal is assumed to be coinciding with the z-axis. This information, though not necessary for sketching input, may be necessary for sweep inputs.

From the preceding analysis in this and the previous sections the survey converged into identifying a data structure which could accept 'sketching input' as one of the methods of input in addition to the others already built into it. The 'half edge' data structure is found to be a suitable one and is described in the following section.

2.8 MANTYLA'S HALFEDGE DATA STRUCTURE

Mantyla [40,42] developed the half edge data structure for the solid modeller GWB. Many of its features are similar to the winged-edge data structure. It has a five level hierarchical structure consisting of nodes of type Solid, Face, Loop, HalfEdge, Edge and Vertex. The entire solid is represented by doubly linked lists of these nodes.

The Solid node forms the root node and is a member of the assembly. Thus an assembly is a linked list of Solid nodes. It gives access to faces, edges and vertices of the model through pointers to the three nodes namely, Faces, Edges and Vertices.

The Face node represents the surface. In a surface one loop represents the outer boundary while others represent holes. The nodes that represent (i) the outer loop and (ii) the doubly linked list of all loops of the face are the important pointers in this node. In addition to this there are pointers to the (i) Solid node it belongs to (ii) Surf node describing the geometry and (iii) Face nodes describing the previous and next faces of the solid.

The Loop node describes one connected boundary. It has a pointer to its present face, a pointer to one of the half edges that form the boundary and pointers to the next and previous loops of the face.

The HalfEdge node describes one line segment of a loop. The half edge concept is derived from the fact that all the edges will be included in the loops twice, once in the positive orientation and once in the negative orientation, in a orderly (clockwise

or anticlockwise) arranged list of loops. It consist of a pointer to its parent loop and a pointer to the starting vertex of the line segment in the direction of the loop. It has pointers to the previous and the next HalfEdge nodes of the loop and to the Edge node the HalfEdge belongs to.

The Edge node associates two half edges, which when combined form the full edge (i.e. in the positive and negative orientation), with each other. It consists of pointers to the left and the right half edges as shown in figure 2.19. The doubly linked list of edges is realised by means of pointers to the next and previous edges.

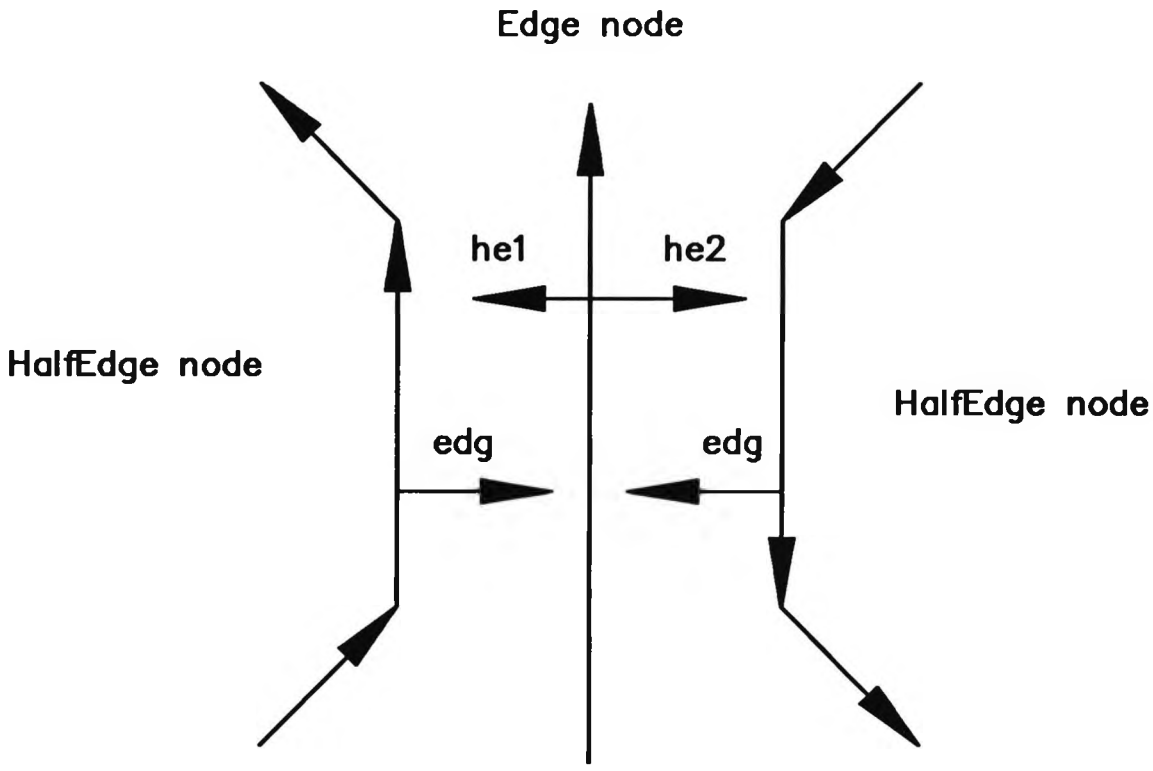

FIGURE 2.19
HALFEDGE DATA STRUCTURE

The Vertex node contains a vector of four floating point numbers representing the homogeneous coordinates of a vertex to pointers to the next and previous vertex realise a doubly linked list of vertices of a solid. Each Vertex node also includes the pointer to one of the half edges emanating from that particular vertex.

The structure is very much complicated and is explained by the figure 2.20. Figure

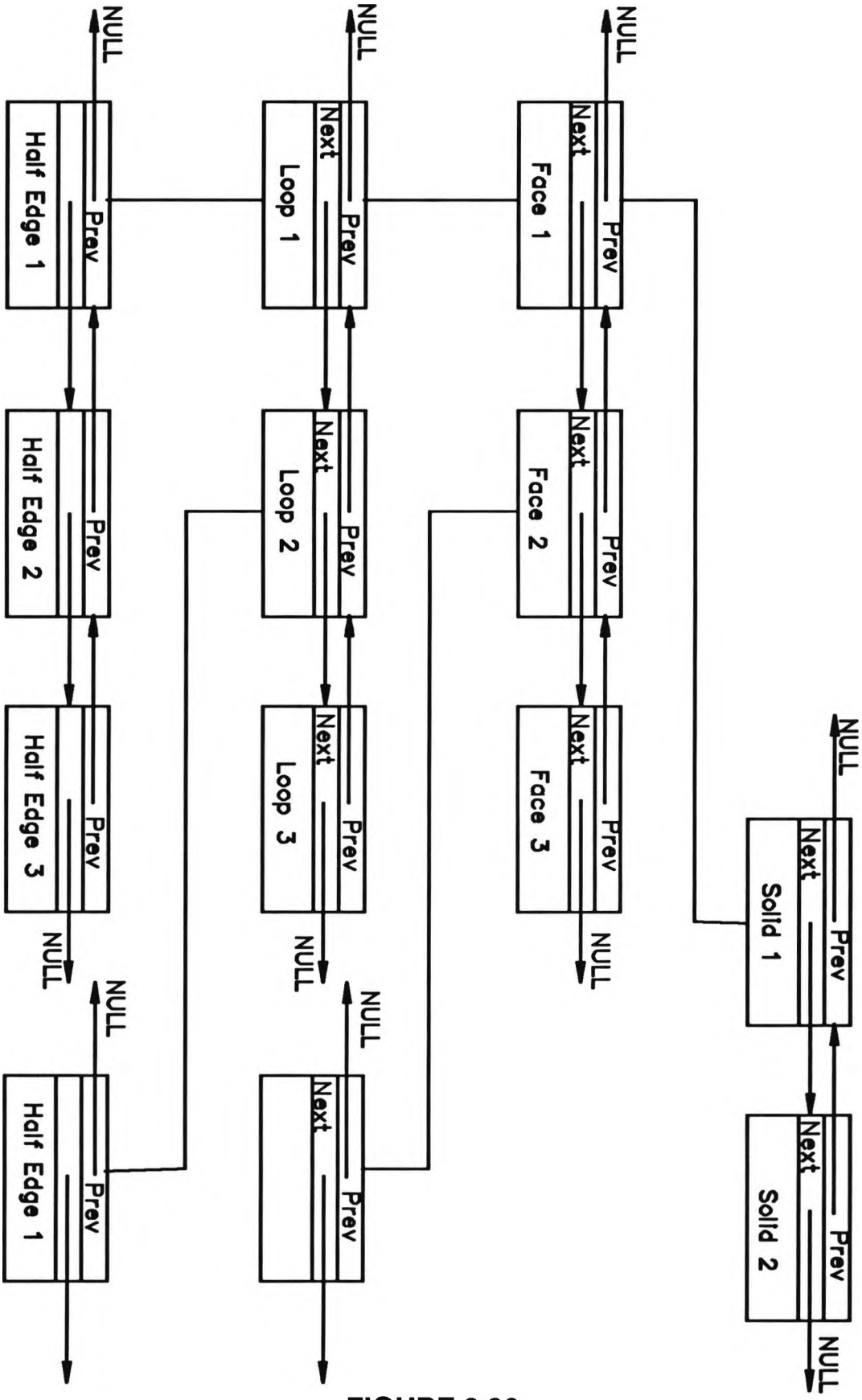

FIGURE 2.20

LINKED LIST STRUCTURE OF SOLID REPRESENTATION

2.21 explains the supporting lists. The 'C' implementation of the halfedge data structure is given in appendix.

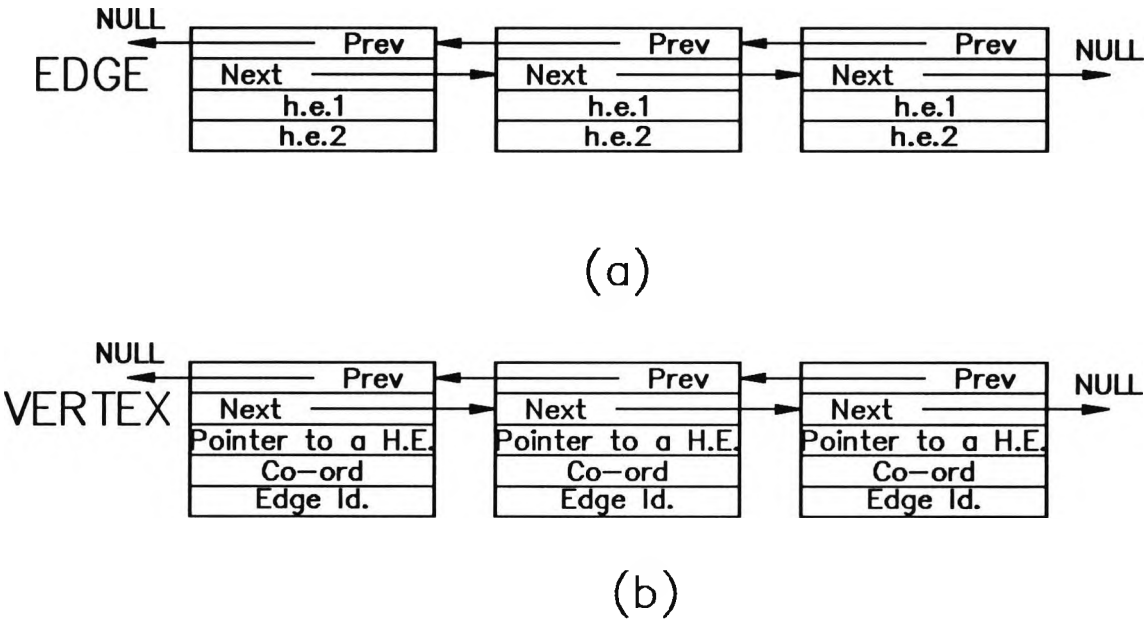

FIGURE 2.21
SUPPORTING LIST STRUCTURES

2.9 AN EXAMPLE REPRESENTATION

It was established in the previous section that there are three doubly linked lists which keep all the data representation of a solid model. They are (i) list of 'Solid' nodes (ii) list of 'Edge' nodes (iii) list of 'Vertex' nodes. Consider the 'L' slider of figure 2.14. Section 2.9.1 describes its representation assuming that the cylindrical hole is not present while section 2.9.2 indicates how the presence of the cylindrical hole is represented.

2.9.1 REPRESENTATION OF THE 'L' BLOCK

In this section the arrangement of data of the 'L' block is described with the assumption that the cylinder is not present. It has four parts namely (i) list of Solid' nodes (ii) list of 'Face' nodes belonging to a solid (iii) list of 'Edge' nodes belonging to an assembly (iv) list of 'Vertex' nodes belonging to a solid.

2.9.1.1 LIST OF SOLID NODES

The 'L' block is made of only one solid and since there is only one solid there can only be one 'Solid' node. Therefore the previous and next 'Solid' nodes should point to

NULL.The solid identifier is '1'. Thus the three requirements to complete this node are

(i) pointer to the start of the list of 'Face' nodes
(ii) pointer to the start of the list of 'Edge' nodes
(iii) pointer to the start of the list of '3Vertex' nodes

Figure 2.22 illustrates this node.

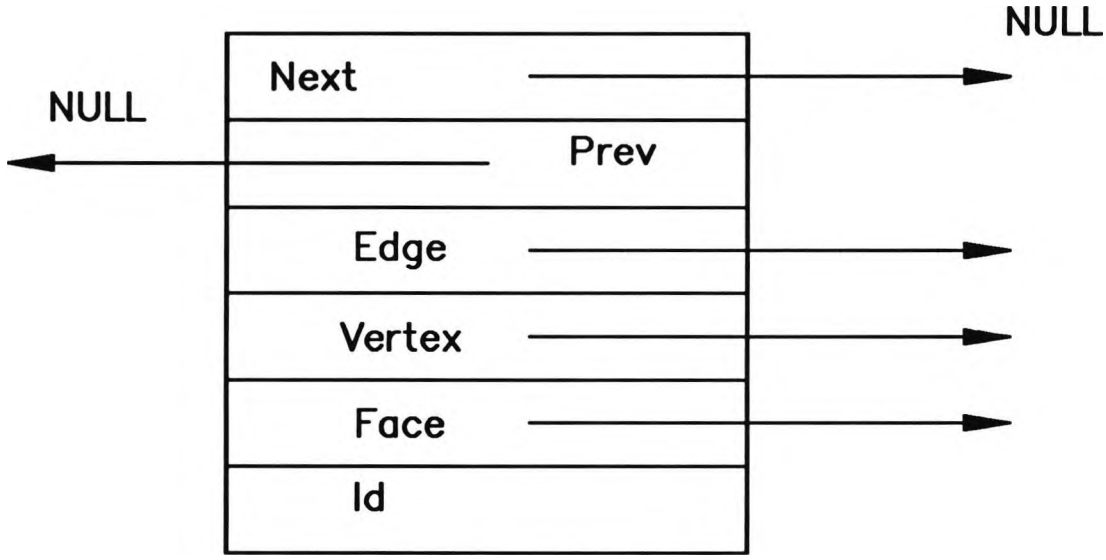

FIGURE 2.22
ILLUSTRATION OF THE SOLID NODE

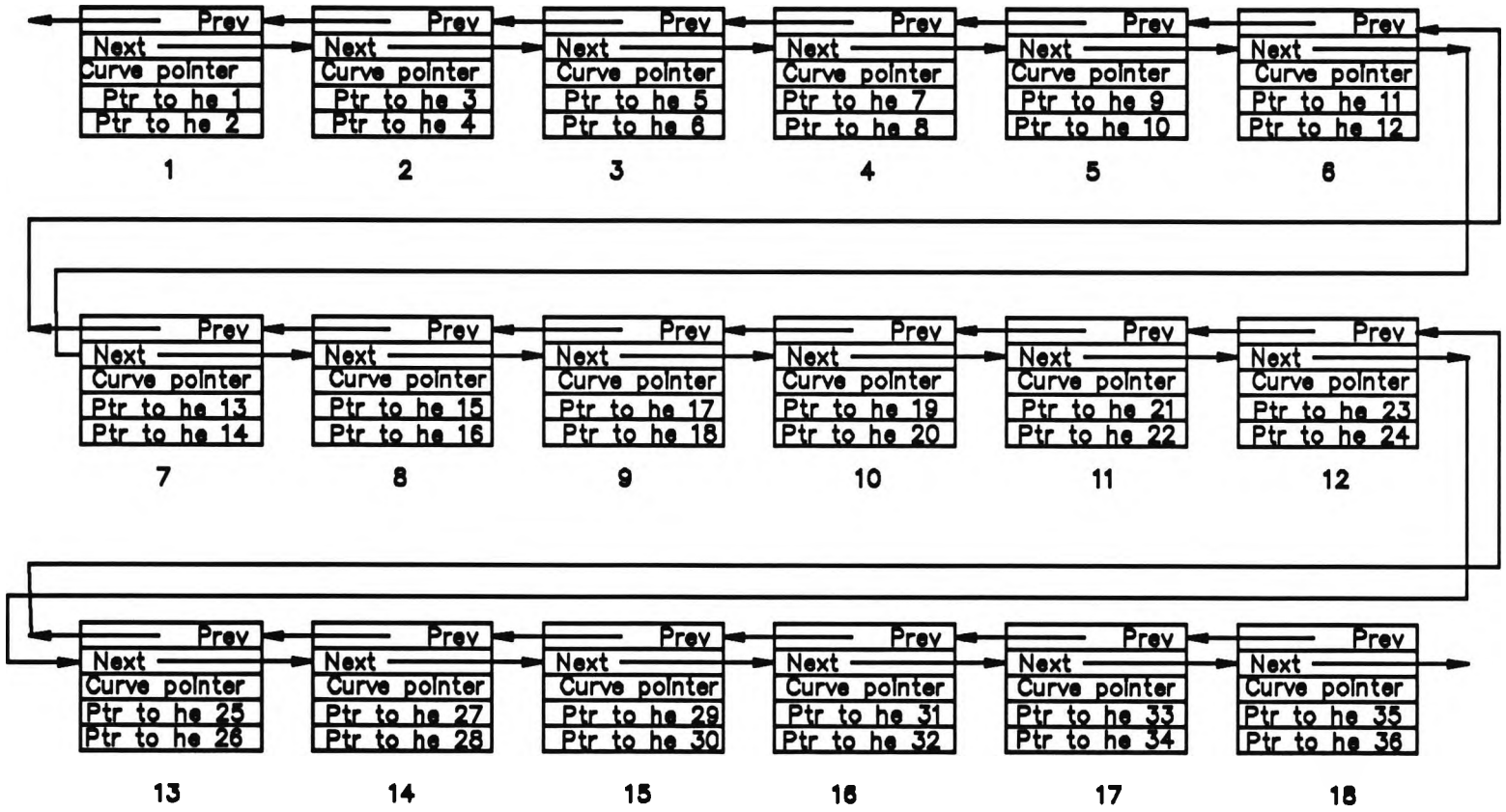

FIGURE 2.23

EDGE LIST OF THE 'L' BLOCK

2.9.1.2 LIST OF EDGE NODES

There are 18 edges in the 'L' block of figure 2.14. This means there should be 36 half edges. They are

Half edge No	Half edge	Start edge	Edge No	Edge details
1	1 2	1	1	1 2
2	2 1	2	1	1 2
3	2 3	2	2	2 3
4	3 2	3	2	2 3
5	3 4	3	3	3 4
6	4 3	4	3	3 4
7	4 5	4	4	4 5
8	5 4	5	4	4 5
9	5 6	5	5	5 6
10	6 5	6	5	5 6
11	6 1	6	6	6 1
12	1 6	1	6	6 1
13	1 7	1	7	1 7
14	7 1	7	7	1 7
15	2 8	2	8	2 8
16	8 2	8	8	2 8
17	3 9	3	9	3 9
18	9 3	9	9	3 9
19	4 10	4	10	4 10
20	10 4	10	10	4 10
21	5 11	5	11	5 11
22	11 5	11	11	5 11
23	6 12	6	12	6 12
24	12 6	12	12	6 12
25	7 8	7	13	7 8
26	8 7	8	13	7 8
27	8 9	8	14	8 9
28	9 8	9	14	8 9
29	9 10	9	15	9 10
30	10 9	10	15	9 10
31	10 11	10	16	10 11
32	11 10	11	16	10 11
33	11 12	11	17	11 12
34	12 11	12	17	11 12
35	7 12	7	18	7 12
36	12 7	12	18	7 12

In the case of the 'L' block all the edges are straight and hence no curve equation is presented. Also all the edges appear in exactly two faces or two times. Hence the curve information is a node of type 'Line' and will have 'times_used' = 2 and 'curve_type' as a constant representing straight line (say 4). The list of 'Edge' nodes in the 'L' block is schematically shown in figure 2.23.

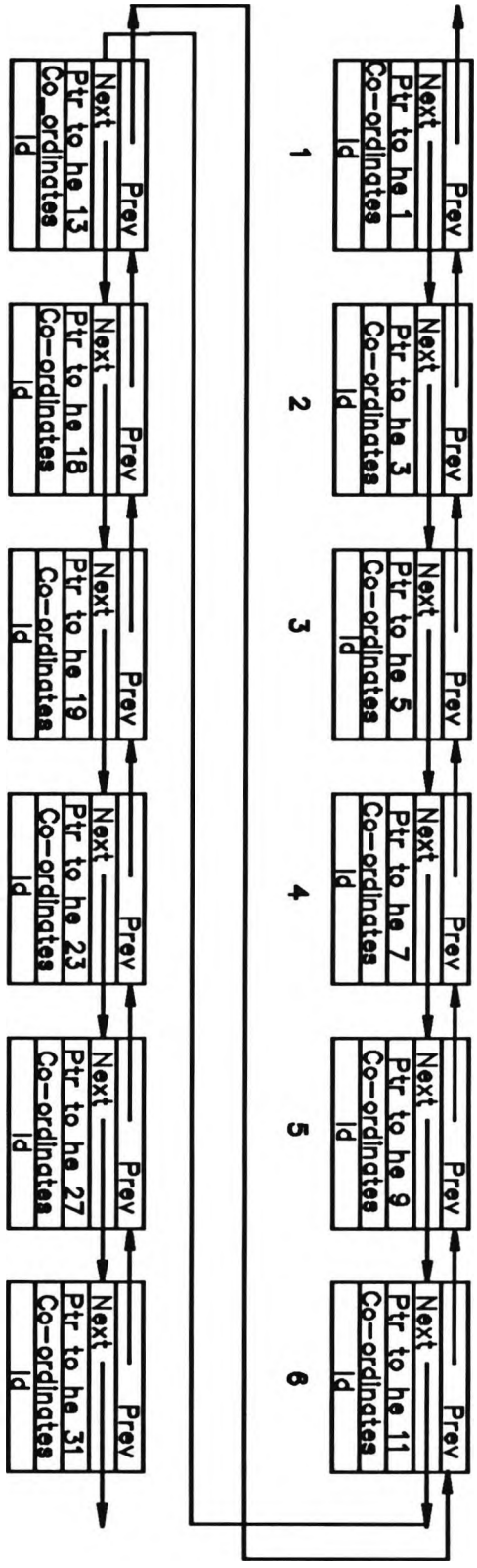

FIGURE 2.24
VERTEX LIST OF THE 'L' SLIDER

2.9.1.3 LIST OF VERTEX NODES

There are 12 vertices in the 'L' block and hence there should be 12 nodes in the list. One half edge is given here in each node to identify other edges emanating from this node. The list is given in figure 2.24.

2.9.1.4 LIST OF FACE NODES

There are 8 faces in the 'L' block. In addition to this there is a cylindrical hole in the 'L' slider. This means that there should be 8 'Face' nodes in the list of faces each node representing a face. For the 8 faces of the 'L' block assuming that there is no hole, there are no rings. This means that the outer loop and the list of loops, point to the boundary of the face. A plane node is present for each of the face representing the equation of the plane. The loops (faces in this case) arranged in the clockwise sense are as follows:

1 7 8 2
2 8 9 3
3 9 10 4
4 10 11 5
5 11 12 6
6 12 1 7
1 2 3 4 5 6
7 12 11 10 9 8

Putting all the lists together in the 'Solid' node it gives the hierarchical structure.

2.9.2 REPRESENTATION OF THE 'L' SLIDER

The 'L' slider is formed by the inclusion of the cylindrical hole in the 'L' block. In effect it makes three changes. They are
(i) addition of a circular ring in face 7
(ii) addition of a circular ring in face 8 and
(iii) addition of a cylindrical face.
Changes (i) and (ii) adds an internal loop to the 'Face' nodes described in 2.9.1 above.

The cylindrical face is represented by the node cylinder. It is assumed to be created, with its axis parallel to the z axis and the face lying in the xy plane, and is rotated and translated to reach the present position. Thus there is a transformation matrix associated with it.

2.10 INFERENCES

The following inferences could be drawn from this survey.

(i) 'Sketching input' can be accepted as an input method for 'solid modelling' with boundary representation

(ii) After the sketch is processed i.e. the descrepancies due to the imperfection of the hand drawing are corrected, it could have a data structure that could permit linking to a solid modeller.

(iii) Euler operators should be used for the building of the solid model

(iv) 'Halfedge' data structure is a suitable candidate for the B-rep solid modeller in the 'sketching input' environment.

CHAPTER 3
BACKGROUND THEORY

3.0 GENERAL

Several special techniques are used in this thesis. They can be summarised as fitting methods for 2-dimensional lines and curves, 2-dimensional geometry and transformations. There is a statistical flavour to some of the material but any deeper statistical theory is omitted. The methods largely involve simple means and smoothing or least squares.

3.1 TWO-DIMENSIONAL GEOMETRY

This section contains a discussion of 2-dimensional geometry. The vector notation is used when it is convenient. A generic vector being written as a column vector

$$\tilde{X} = \begin{bmatrix} x \\ y \end{bmatrix} = (x\ y)^T$$

The Euclidean distance between two points P: $(x_1\ y_1)^T = \tilde{X}$, Q: $(x_2\ y_2)^T = \tilde{Y}$ is PQ where

$$PQ^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$ which is the two norm of $(\tilde{X} - \tilde{Y})$.

A line in 2-dimensions is written

$$\tilde{A}^T \tilde{X} = ax + by = c \qquad (3.1)$$

and as c changes parallel lines are obtained. The special case when c = 0 gives a line through the origin O: $(0\ 0)^T$ perpendicular to the vector

$$\tilde{A} = \begin{bmatrix} a \\ b \end{bmatrix}.$$

Indeed if $\sqrt{(a^2 + b^2)} = 1$ so that $\tilde{A}$ is a unit vector, then $|\ c\ |$ is the perpendicular distance of the line from O. In general this distance is $\left| \dfrac{c}{\sqrt{(a^2 + b^2)}} \right|$. The vector formulation is convenient for a number of different computations. The distance of a point $Z = (z_1\ z_2)^T$ from the line (3.1) can easily be found in the following way. Since $\tilde{A}$ is perpendicular to the line, taking $\tilde{X}$ to be the foot of the perpendicular from $\tilde{Z}$ to the line $\tilde{X} - \tilde{Z} = k\tilde{A}$ for some k.

But then

$$\tilde{A}^T \tilde{X} - \tilde{A}^T \tilde{Z} = k\tilde{A}^T \tilde{A}.$$

Giving

$$c - \tilde{A}^T \tilde{Z} = k\tilde{A}^T \tilde{A} = k(a^2 + b^2).$$

Thus $$k = \left[\frac{c - \tilde{A}^T \tilde{Z}}{(a^2 + b^2)} \right].$$

But the two norm of $(\tilde{X} - \tilde{Z})$ is $k^2(a^2 + b^2)$. Thus the square of the perpendicular

distance is $\frac{[c - \tilde{A}^T \tilde{Z}]^2}{(a^2 + b^2)}$.

The distance between two parallel lines $\tilde{A}^T \tilde{X} = c_1$ and $\tilde{A}^T \tilde{X} = c_2$ is just

$$\frac{c_1 - c_2}{\sqrt{(a^2 + b^2)}}.$$

The division of a line given by its end points, by a given point is handled in the following way. Consider the point P:(x,y) which divides the line connecting A:(x_1,y_1) and B:(x_2,y_2) at a ratio m_1:m_2. Then it can be shown that

$$x = \frac{m_1 x_2 + m_2 x_1}{m_1 + m_2}$$

$$y = \frac{m_1 y_2 + m_2 y_1}{m_1 + m_2}$$

If $m_1 = 1$ and $m_2 = \lambda$ then

$$x = \frac{x_2 + \lambda x_1}{1 + \lambda}$$

$$y = \frac{y_2 + \lambda y_1}{1 + \lambda}$$

From these equations

$$\lambda = \frac{x_2 - x}{x - x_1} = \frac{y_2 - y}{y - y_1}$$

If λ is positive P is between A and B. If λ is negative P is outside AB.

3.2 REGRESSION OF Y ON X AND X ON Y

Regression in simple terms is the development of an equation connecting two variables say x and y. One of these variables is dependant while the other is independant. The dependant variable is said to be regressed on the independant variable. The regression of the y co-ordinate on x co-ordinate of the points gives an equation $y = ax + b$. In the regression of y on x the values of the x co-ordinates are assumed to be correct and the error or deviation is associated with the y co-ordinate as shown in figure 3.1 (a). In a similar way the regression of x co-ordinate on y co-ordinate gives an equation $x = cy + d$. Here the errors are associated with the x co-ordinates and the y co-ordinates are assumed to be correct as shown in figure 3.1 (b). These two regression lines are different with different slopes and different intercepts. It is later shown in section 3.3 that one condition for the least squares solution of y on x is

$$-\sum_i y_i + a\sum_i x_i + n\,b = 0$$

$$-\frac{1}{n}\sum_i y_i + a\frac{1}{n}\sum_i x_i + b = 0$$

$$-\overline{Y} + a\overline{X} + b = 0$$

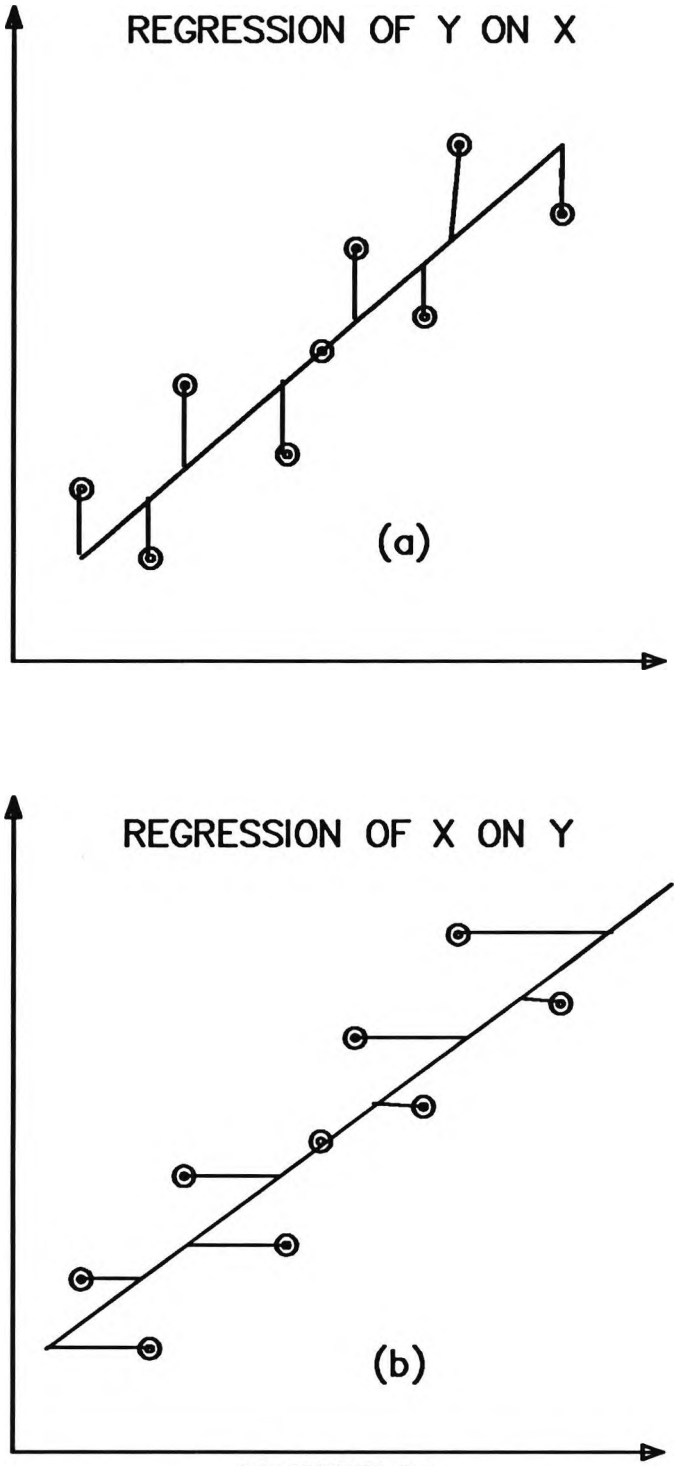

FIGURE 3.1

REGRESSION OF Y ON X AND X ON Y

This means that both lines pass through the mean point.

In a situation where one variable is dependant on the other, the practice is to regress the dependant variable on the independant variable. However in a sketching situation both y and x co-ordinates are independent. This leaves the problem of choosing

between the lines y on x and x on y. In this thesis the line y on x is used to identify the initial slope and wherever possible, the mean point is used to compute the intercept of the corrected line, thus minimising the confusion of the selection.

3.3 LEAST SQUARES FITTING OF STRAIGHT LINE

Consider the situation as shown in figure 3.1 (a). The equation of the fitted line would have satisfied (x_1,y_1) if the fitting was perfect. But because there is an error e_1 present in the fitting the point (x_1,y_1) is above the line as shown in figure 3.1 (a). Let there be such an error associated with each of the 'n' points to be fitted and let them be equal to $e_1, e_2, - - - e_n$ in quantity.

Also let

$$e_1^2 + e_2^2 + - - - - + e_n^2 = e^2 = S$$

The method of least squares fits the line such that the value of e^2 is a minimum. Francis [65] deals with this problem very well. In general if the equation of the line is y = ax + b and the point is (x_i, y_i) then the error $e_i = y_i - (a x_i + b)$

Let $S = \sum e_i^2 = \sum (y_i - a x_i - b)^2$

The least squares solution for a and b is when S is a minimum and to obtain this minimum, the equations $\frac{\partial S}{\partial a} = 0$ and $\frac{\partial S}{\partial b} = 0$ should be solved.

Let $\sum_i$ to represent $\sum_{i=1}^{n}$ in this thesis.

Now $$\frac{\partial S}{\partial a} = \frac{\partial}{\partial a}[\sum_i (y_i - a x_i - b)^2]$$

ie. $$\frac{\partial S}{\partial a} = \sum_i \{2 * (y_i - a x_i - b) * (-x_i)\}$$

Thus $\frac{\partial S}{\partial a} = 0$ gives $$-\sum_i \mathbf{x_i y_i} + \mathbf{a} \sum_i \mathbf{x_i^2} + \mathbf{b} \sum_i \mathbf{x_i} = \mathbf{0}$$

Similarly $$\frac{\partial S}{\partial b} = \frac{\partial}{\partial b}[\sum_i (y_i - a x_i - b)^2]$$

i.e. $$\frac{\partial S}{\partial b} = \sum_i \{2 * (y_i - a x_i - b) * (-1)\}$$

Thus $\frac{\partial S}{\partial b} = 0$ gives $$-\sum_i \mathbf{y_i} + \mathbf{a} \sum_i \mathbf{x_i} + \mathbf{nb} = \mathbf{0}$$

If sumx $= \sum_i x_i$, sumy $= \sum_i y_i$, sumxsqrd $= \sum_i x_i^2$ and sumxy $= \sum_i x_i y_i$ then

$$b = \frac{(sumx*sumxy - sumxsqrd*sumy)}{(sumx*sumx - n*sumxsqrd)}$$

$$a = \frac{(sumy - n*b)}{sumx}$$

Thus the least squares solution to a straight line could be obtained using all the points supposed to be on that straight line. However this solution will fail when the line is vertical. This is because the denominator becomes very small allowing the theoretical infinite intercept on the y axis. In these circumstances the equation to be fitted should take the form x = c and c is given by $\frac{1}{n}\sum_i x_i$

3.4 LEAST SQUARES FITTING OF ELLIPSE

Several research workers have investigated the problem of fitting ellipses, using different parameters for optimising. Numerous references deal with the fitting of circles and ellipses [66 - 76]. Angell and Barber [68] illustrate a method to fit an ellipse using the least squares algorithm.

Consider the general conic section $F(x,y) = x^2 + hxy + by^2 + fx + gy + c$.

If $M = \sum_i F^2(x_i, y_i) - \frac{1}{n}[\sum_i F(x_i, y_i)]^2$ is defined as the measure of goodness of fit, the method suggests that M will be a minimum when the fitness is good.
Re-writing the equation for M

$$M = \sum_i (x_i^2 + hx_iy_i + by_i^2 + fx_i + gy_i + c)^2 - \frac{1}{n}(\sum_i x_i^2 + hx_iy_i + by_i^2 + fx_i + gy_i + c)^2$$

This when rearranged will result in the following :

$$\begin{aligned}
M = \; & [\sum_i x_i^4 - \frac{1}{n}(\sum_i x_i^2)^2] + h^2[\sum_i x_i^2y_i^2 - \frac{1}{n}(\sum_i x_iy_i)^2] \\
& + b^2[\sum_i y_i^4 - \frac{1}{n}(\sum_i y_i^2)^2] + f^2[\sum_i x_i^2 - \frac{1}{n}(\sum_i x_i)^2] \\
& + g^2[\sum_i y_i^2 - \frac{1}{n}(\sum_i y_i)^2] + 2h[\sum_i x_i^3y_i - \frac{1}{n}\sum_i x_i^2 \sum_i x_iy_i] \\
& + 2b[\sum_i x_i^2y_i^2 - \frac{1}{n}(\sum_i x_i^2 \sum_i y_i^2)] + 2f[\sum_i x_i^3 - \frac{1}{n}\sum_i x_i^2 \sum_i x_i] \\
& + 2g[\sum_i x_i^2y_i - \frac{1}{n}(\sum_i x_i^2 \sum_i y_i)] + 2hb[\sum_i x_iy_i^3 - \frac{1}{n}\sum_i x_iy_i \sum_i y_i^2] \\
& + 2hf[\sum_i x_i^2y_i - \frac{1}{n}(\sum_i x_iy_i \sum_i x_i)] + 2hg[\sum_i x_iy_i^2 - \frac{1}{n}\sum_i x_iy_i \sum_i y_i] \\
& + 2bf[\sum_i x_iy_i^2 - \frac{1}{n}(\sum_i y_i^2 \sum_i x_i)] + 2bg[\sum_i y_i^3 - \frac{1}{n}\sum_i y_i^2 \sum_i y_i]
\end{aligned}$$

$$+ 2fg\,[\sum_i x_i y_i - \frac{1}{n}(\sum_i x_i \sum_i y_i)]$$

Angell and Barber rewrite this equation in the following way

$$\begin{aligned} M = \; & C_{11} + C_{hh}h^2 + C_{bb}b^2 + C_{ff}f^2 + C_{gg}g^2 + C_{1h}.2h + C_{1b}.2b \\ & + C_{1f}.2f + C_{1g}.2g + C_{hb}.2hb + C_{hf}.2hf + C_{hg}.2hg + C_{bf}.2bf \\ & + C_{bg}.2bg + C_{fg}.2fg \end{aligned}$$

Now they differentiate M with respect to h, b, f and g and equating each result to zero they get four equations. They are:

$$C_{1h} + C_{hh}h + C_{hb}b + C_{hf}f + C_{hg}g = 0$$
$$C_{1b} + C_{hb}h + C_{bb}b + C_{bf}f + C_{bg}g = 0$$
$$C_{1f} + C_{hf}h + C_{bf}b + C_{ff}f + C_{fg}g = 0$$
$$C_{1g} + C_{hg}h + C_{bg}b + C_{fg}f + C_{gg}g = 0$$

These four equations can be solved by Gaussian elimination or by another numerical technique and the solutions for h, b, f and g can be found. To find the value of c these values should be substituted in the following equation.

$$\sum_i (x_i^2 + hx_i y_i + by_i^2 + fx_i + gy_i + c) = 0$$

Thus $$c = \frac{1}{n}\,[\sum_i x_i^2 + h\sum_i x_i y_i + b\sum_i y_i^2 + f\sum_i x_i + g\sum_i y_i]$$

If these coefficients are computed and stored in a 4x5 matrix then this array could be passed as a parameter to a program which gives the Gaussian solution to the set of equations. In this research the coefficients are expressed by the following coefficient matrix.

$$\text{Coefficient matrix} = \begin{bmatrix} C_{hh} & C_{hb} & C_{hf} & C_{hg} & C_{1h} \\ C_{hb} & C_{bb} & C_{bf} & C_{bg} & C_{1b} \\ C_{hf} & C_{bf} & C_{ff} & C_{fg} & C_{1f} \\ C_{hg} & C_{bg} & C_{fg} & C_{gg} & C_{1g} \end{bmatrix}$$

3.5 MEETING POINT OF MORE THAN TWO STRAIGHT LINES

When two straight lines meet at a point and if their equations are known then there is only a unique solution to those equations which is the point of intersection. However when more than two lines meet in a point and their equations have some error, finding a solution to the point becomes harder. This is the situation when more than 2 hand sketched lines meet at a vertex. A least squares method is described in this section to circumvent this problem. Let the equations of the line be

$$\tilde{A}_1^T \tilde{X} = c_1 \quad i.e. \quad a_1 x + b_1 y - c_1 = 0$$
$$\tilde{A}_2^T \tilde{X} = c_2 \quad i.e. \quad a_2 x + b_2 y - c_2 = 0$$
$$- \quad - \quad -$$
$$\tilde{A}_n^T \tilde{X} = c_n \quad i.e. \quad a_n x + b_n y - c_n = 0$$

Let $a_i^2 + b_i^2 = 1$ for all values of i. Then c_i is the perpendicular distance of the line from the origin. If the point (x_1, y_1) has to lie on these equations it should satisfy them. Let there be errors e_1, e_2, e_3 - - - - - e_n when the point is substituted in these equations. In general these errors could be written as follows:

$$\tilde{A}_i^T \tilde{X} - c_i = e_i$$

and

$$S = e_1^2 + e_2^2 + \; - \; - \; + e_n^2 .$$

Referring the theory in section 3.1 it could be seen that $| e_i |$ is the perpendicular distance of $\tilde{X}$ from the straight line $\tilde{A}_i^T \tilde{X} = C_i$.

The least squares solution seeks to minimise this S by the choice of $\tilde{X}$. These equations could be written in matrix form as $\tilde{A}_i^T \tilde{X} - \tilde{C} = \tilde{E}$. Then S is the 2 norm of $(\tilde{A}_i^T \tilde{X} - \tilde{C})$. Minimising S is a standard least squares problem which has solution $\tilde{X}^* = (\tilde{A}^T \tilde{A})^{-1} \tilde{A} \tilde{C}$. Note that

$$\tilde{A}^T \tilde{A} = \begin{bmatrix} \sum a_i^2 & \sum a_i b_i \\ \sum a_i b_i & \sum b_i^2 \end{bmatrix} \text{ and } \left(\tilde{A}^T \tilde{A}\right)^{-1} = \frac{1}{D} \begin{bmatrix} \sum b_i^2 & -\sum a_i b_i \\ -\sum a_i b_i & \sum a_i^2 \end{bmatrix}$$

$$\tilde{A} \tilde{C} = \begin{bmatrix} \sum a_i c_i \\ \sum b_i c_i \end{bmatrix} \text{ and } D = \sum a_i^2 \sum b_i^2 - (\sum a_i b_i)^2$$

This gives the solution

$$x^* = \frac{1}{D} \left(\sum b_i^2 \sum a_i c_i - \sum a_i b_i \sum b_i c_i \right)$$

and $$y^* = \frac{1}{D} \left(-\sum a_i b_i \sum a_i c_i + \sum a_i^2 \sum b_i c_i \right)$$

An interesting special case is when $\sum a_i b_i = 0$ when

$$x^* = \frac{\sum a_i c_i}{\sum a_i^2}$$

$$y^* = \frac{\sum b_i c_i}{\sum b_i^2}$$

3.6 MEETING POINT OF MORE THAN TWO CURVES AND STRAIGHT LINES

Consider the situation where there are 'a' number of arcs and 'b' number of straight lines meet. Then if an approximate solution (x_1, y_1) to these 'a + b' equations are known (similar to a terminal point to be introduced in chapter 4) then the following could be written.

$$
\begin{aligned}
&x_1^2 + h_1 x_1 y_1 + b_1 y_1^2 + f_1 x_1 + g_1 y_1 + c_1 = e_1 \\
&x_1^2 + h_2 x_1 y_1 + b_2 y_1^2 + f_2 x_1 + g_2 y_1 + c_2 = e_2 \\
&- \quad - \quad - \quad - \quad - \quad - \quad - \\
&x_1^2 + h_a x_1 y_1 + b_a y_1^2 + f_a x_1 + g_a y_1 + c_a = e_a \\
&p_1 y_1 + q_1 x_1 + r_1 = E_1 \\
&p_2 y_1 + q_2 x_1 + r_2 = E_2 \\
&- \quad - \quad - \quad - \\
&p_b y_1 + q_b x_1 + r_b = E_b
\end{aligned}
$$

where $F(x,y) = x^2 + hxy + by^2 + fx + gy + c = 0$ denotes a curve and $p\,y + q\,x + r = 0$ denotes a straight line. The quantities $e_1, e_2 - - - e_a$, $E_1 - - - E_b$ denote errors. The task is then to find (x_1, y_1) such that the sum of the squares of the errors are minimum. The standard method suggests that solution to the equations $\frac{\partial S}{\partial x_1} = 0$ and $\frac{\partial S}{\partial y_1} = 0$ where $S = e_1^2 + e_2^2 + - - + e_a^2 + E_1^2 + - - + E_b^2$, yields the required x_1 and y_1. But the equations are complex and finding a solution is extremely difficult. The following method is suggested to overcome this problem.

Let $s(x_1) = \frac{\partial S}{\partial x_1}$ and $s(y_1) = \frac{\partial S}{\partial y_1}$

In the first stage find 'x_1'such that $s(x_1)$ is zero, keeping y_1 as constant at the initial value. In the second stage find 'y_1' such that $s(y_1)$ is zero keeping x_1 as constant at the value found in the first stage.

Let $e^2 = \sum_i \left[x_1^2 + h x_1 y_1 + b y_1^2 + f x_1 + g y_1 + c\right]^2$

and $E^2 = \sum_i \left[p y_1 + q x_1 + r\right]^2$ for which $\frac{\partial}{\partial x_1}$ and $\frac{\partial}{\partial y_1}$ could be found easily. Then

$S = e^2 + E^2$ and hence $\frac{\partial S}{\partial x_1} = \frac{\partial e^2}{\partial x_1} + \frac{\partial E^2}{\partial x_1} = s(x_1)$ and $\frac{\partial S}{\partial y_1} = \frac{\partial e^2}{\partial y_1} + \frac{\partial E^2}{\partial y_1} = s(y_1)$

Now from Newton-Raphson method $x1_{(n+1)} = x1_n - \frac{s(x1_n)}{s'(x1_n)}$ and $y1_{(n+1)} = y1_n - \frac{s(y1_n)}{s'(y1_n)}$. Thus starting with the approximate values for x_1 and y_1 values for x_1 and y_1 to the required accuracy could be obtained iteratively.

3.7 DETECTING A STRAIGHT LINE

In the design by sketching situation, the designer should be left free to concentrate on the design, using his professional expertise to properly conceive and develop the product while the computer should take up the sketch and recognise it. In the process of recognising it should detect straight lines and arcs. The strategy of detecting a straight line is important in pattern recognition and several methods are employed by researchers to achieve this objective. In the sketching environment the following methods are explored.

(a) Perpendicular distance method
(b) Moving slope method
(c) Generalised conic method

3.7.1 PERPENDICULAR METHOD

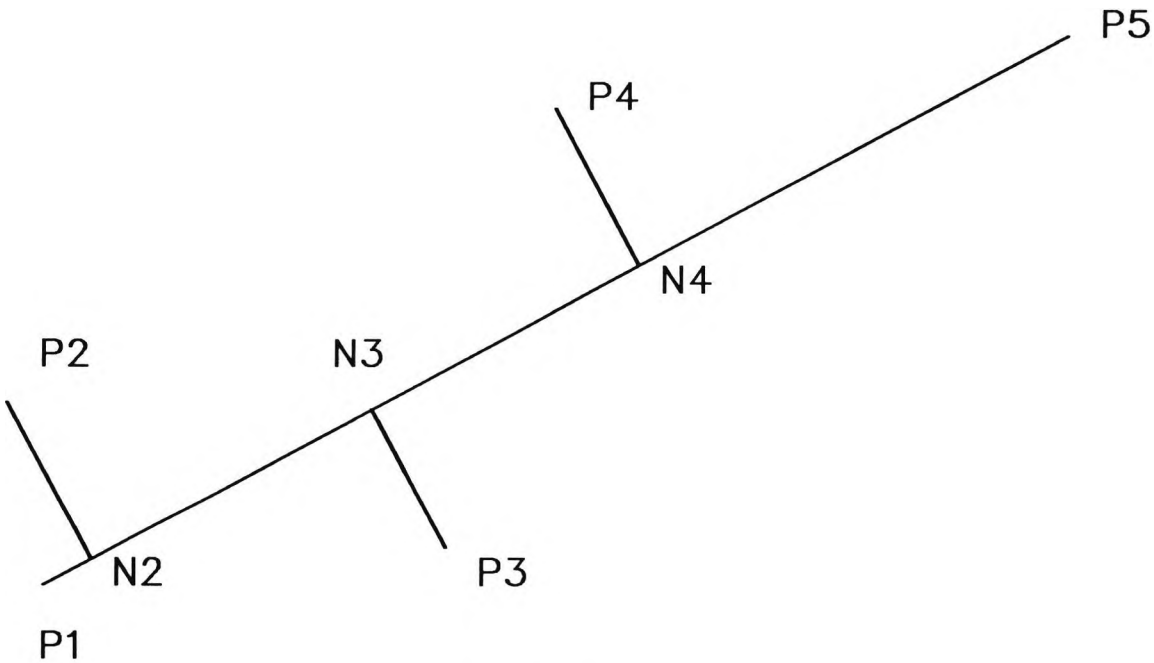

FIGURE 3.2

ILLUSTRATION OF THE PERPENDICULAR METHOD

This method is based on the fact that the points belonging to a sketched straight line will be within a specified small distance from the mean line fitted with these points.

Consider the situation in figure 3.2 describing the scatter of five consecutive points and the line connecting the first and the fifth point. This theory then says that P_2N_2, P_3N_3 and P_4N_4 should be within a specified tolerance (say 2mm). Thus the test constitute the following steps.

(a) Find the maximum perpendicular distance

(b) Check whether it is greater than the tolerance and if it is not then P_1, P_2, P_3, P_4 and P_5 are in a straight line.

3.7.2 MOVING SLOPE METHOD

Consider the situation where the points P_1, P_2, P_3, P_4 and P_5 are in a sketched straight line. Let PQ represent the average line as shown in figure 3.3. Because the length of P_1P_2 is small, small deviations of P_2 from PQ (say 2 mm) could make the angle P_2P_1Q big giving the impression that they do not belong to a straight line. Therefore some kind of smoothing becomes necessary. One such smoothing is the

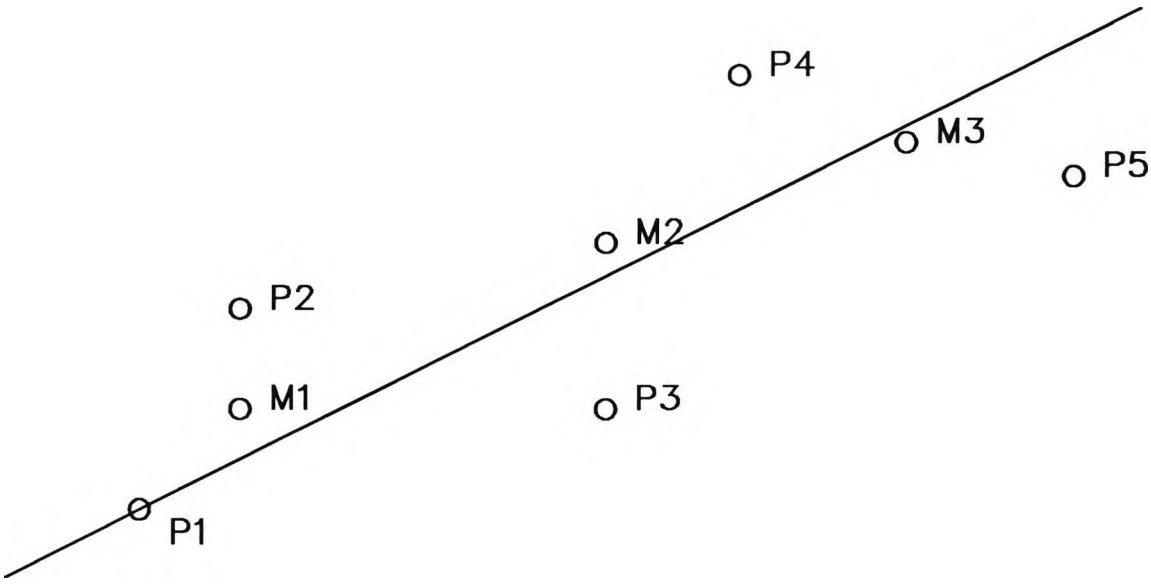

FIGURE 3.3

ILLUSTARION OF MOVING SLOPE METHOD

moving average of the last two or three points. The slope of the line connecting the current point and the smoothed point is termed the 'moving slope' in this thesis. If $P_1 = (x_1,y_1)$ $P_2 = (x_2,y_2)$ and $P_3 = (x_3,y_3)$ then the moving point is

$\frac{x_1 + x_2 + x_3}{3}, \frac{y_1 + y_2 + y_3}{3}$ and if $P_4 = (x_4, y_4)$ the moving slope is $\frac{3y_4 - (y_1 + y_2 + y_3)}{3x_4 - (x_1 + x_2 + x_3)}$.

The condition for a straight line is that the moving slope should not vary outside a specified tolerance i.e. the change in moving slope should be very small.

3.7.3 GENERALISED CONIC METHOD

In this method the points P_1, P_2, P_3, P_4 and P_5 are used to fit an equation of the form $x^2 + hxy + by^2 + fx + gy + c = 0$ using the method of least squares as described in section 3.4. If $c >> 0$, $\frac{h}{c} = 0$, $\frac{b}{c} = 0$ and $\frac{f}{c} \neq 0$ or $\frac{g}{c} \neq 0$ then the points P_1, P_2, P_3, P_4 and P_5 lie in a straight line. Similar conditions could be developed when c is equal to zero.

3.8 DETECTING AN ELLIPSE

Detecting an ellipse is another area of importance and several researchers have worked in this area [68, 69, 74, 75,76]. In the sketching environment the following methods are explored.

(a) Moving slope method
(b) Generalised conic method
(c) Points scatter method

3.8.1 MOVING SLOPE METHOD

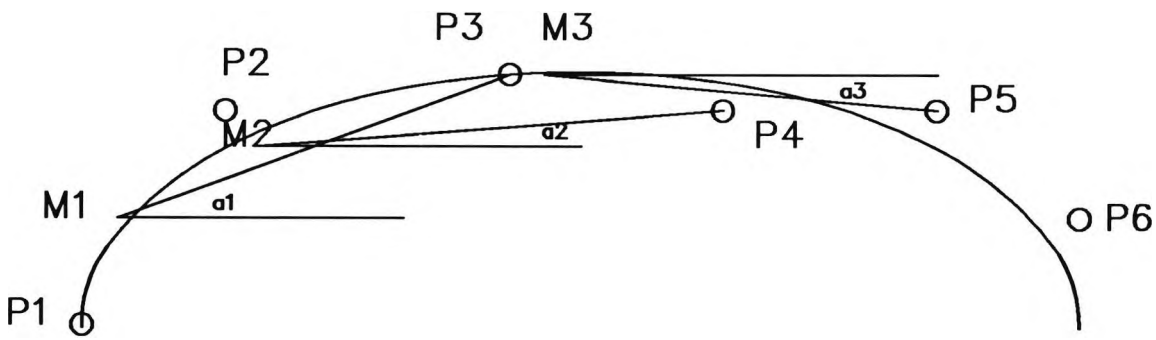

FIGURE 3.4

DETECTION OF ELLIPSE - MOVING SLOPE METHOD

When an ellipse is sketched it is drawn as one whole ellipse in the clockwise or counter clockwise direction or as combination of arcs in the clockwise or counter clockwise direction. This means that the slope changes should be either positive or negative for any particular arc depending on the direction of drawing. Consider the situation described in figure 3.4. As described in section 3.7.2 a smoothing is applied on the past three points leaving M_1, M_2 and M_3 as the smoothed points. Then the moving slopes are a_1, a_2 and a_3. The direction of this elliptical arc is clockwise and hence 'a' is positive (assuming clockwise is positive) and in general 'a' should be greater than 0 for the point to be in the ellipse. This property is used to detect ellipses or elliptical arcs i.e. the points are taken to be in an ellipse as long as the slope change is always having the same sign.

3.8.2 GENERALISED CONIC METHOD

As described in section 3.4 this method employs the method of least squares to fit an equation of the form $x^2 + hxy + by^2 + fx + gy + c = 0$. If it is an ellipse $\frac{1}{c}$ and $\frac{b}{c}$ will not be equal to 0. This is the criterion used in this method to identify the ellipse.

3.8.3 POINTS SCATTER METHOD

Consider an arc whose starting point and finishing point are connected by a straight line. The entire arc falls to one side of this line and this property could be used to identify whether the line under consideration is an arc or not. This is the criterion used in the scatter method.

3.9 DETECTING THE END OF A STRAIGHT LINE

This section describes the situation where the line is identified as a straight line but its end point is not known. Consider the situation described in figure 3.5 (a). The slope of AB could be found by a least squares fitting of the points falling within the first 10 mm (the minimum length of the sketched line, or the diameter of the hole). Then the moving slope could be used to identify whether a point is in the line AB or not. The points upto C in figure 3.5(a) will be passing the test and therefore will be stored in the line segment. When the point P is taken up, the test fails indicating that P is not in line AB. If P belongs to another line as in figure 3.5 (a) then C becomes the first point in the new line. Thus C will be stored as the last point in 'line 1' and first point in 'line 2'. Now consider the situation in figure 3.5 (b). Here again the points up to C will be falling into the line AB. But the difference is that the 'line 2' should start with point P and not point C. This necessitates a second test called the distance test. The distance test states that the distance between the last point and the current point should be less than 3 mm if they belong to the same line segment. This means CP should be less than 3 mm if they are in the same line meaning that 'line 1' and 'line 2' are two continuous lines. If CP is more than 3 mm, they are two distinct lines. This

distance test also allows the identification of the lines shown in figure 3.5(c). Thus the test for a point to be in a straight line constitutes

(a) the moving slope should be constant

(b) the distance between the current point and the last point should be less than 3 or 4 millimeters.

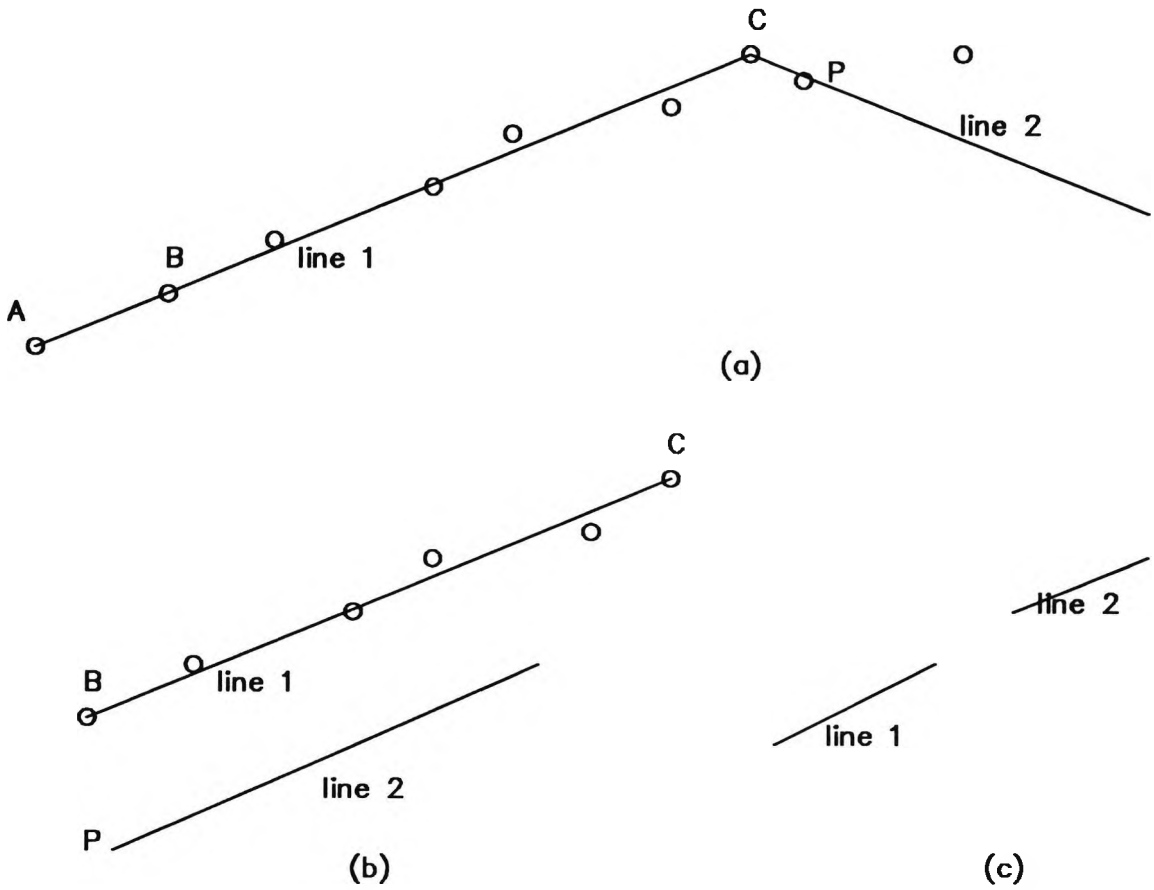

FIGURE 3.5

DETECTION OF ENDPOINT OF A STRAIGHT LINE

If these two conditions are met the point is in the straight line under consideration. When a point fails to meet one of these conditions the last point is identified as the end point of the line. If the distance is less than 3 mm then the lines are continuous and they are distinct otherwise. The starting point of the next line (line 2) is chosen to be C or P in accordance with whether the lines are distinct or continuous.

3.10 DETECTING THE END OF AN ELLIPSE OR ARC

This section describes the situation where the line is identified as an elliptical arc but its end point is not known. Consider the situation described in figure 3.6 (a). The moving slope change will always be less than 0 until $i = 6$ where the point P_6 is in line 2. Thus the point P_5 is identified as the end point by the slope test. In fact this is the

only instance when the slope test could be used satisfactorily to identify the end point in a continuous line situation. Consider the arc combination shown in figure 3.6 (b). Though line 1 and line 2 meets at point P_5 the slope change continues to be negative and hence the slope test fails to identify line 2. Consider now the combination of an arc and straight line as shown in figure 3.6 (c). Here the slope test could indicate at some point that it has entered a straight line. But it will not precisely identify the merging point. In these circumstances the distance test comes to the aid. If the 'line 2' in figure 3.6 (b) and 3.6 (c) could be started from the other ends they could be identified as different lines by the distance test and their end point could be identified by the slope test. This imposes a constraint on the user.

Thus the end point of a distinct elliptical arc could be identified by the combination of (a) slope test and (b) distance test.

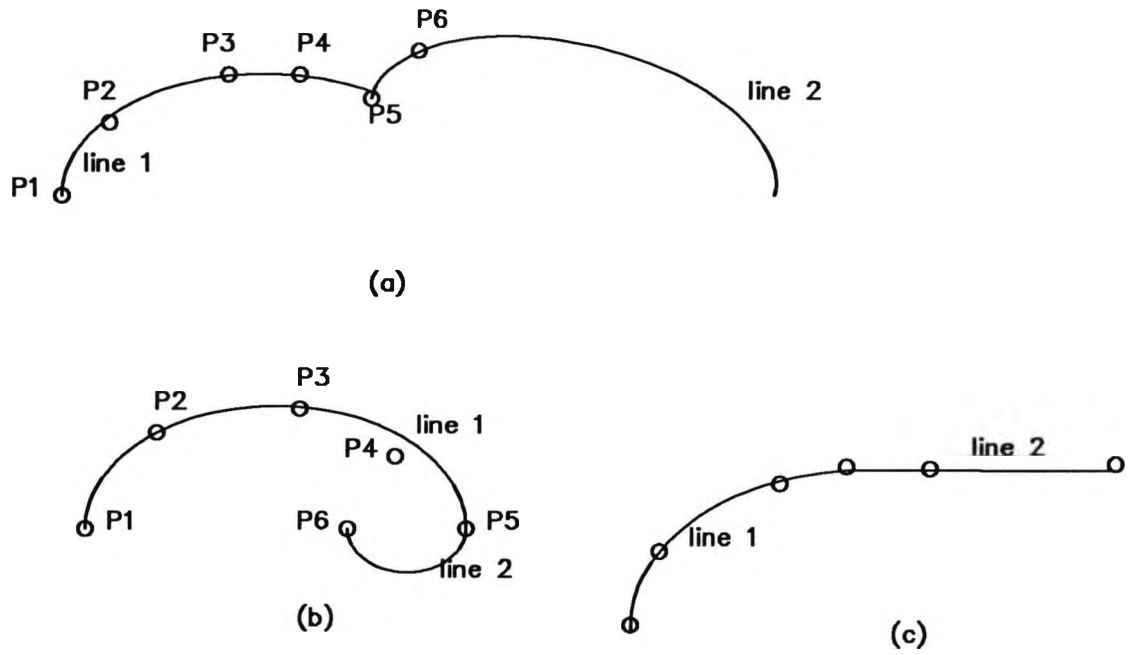

FIGURE 3.6

FINDING THE END POINT OF AN ELLIPSE

3.11 ISOMETRIC SKETCHING

Consider a unit cube ABCDEFGH with the axis system as shown in figure 3.7. The Z axis is perpendicular to the plane of the paper and in the positive direction coming out of the paper as shown.Now first give a rotation of 45 degrees to the cube about the Y axis and then give a rotation about the X axis until AG coincides with the Z axis. This rotation about the X axis is 35.264 degrees. At this stage the cube when

viewed in a direction parallel to the Z axis will look like what is shown in figure 3.8.

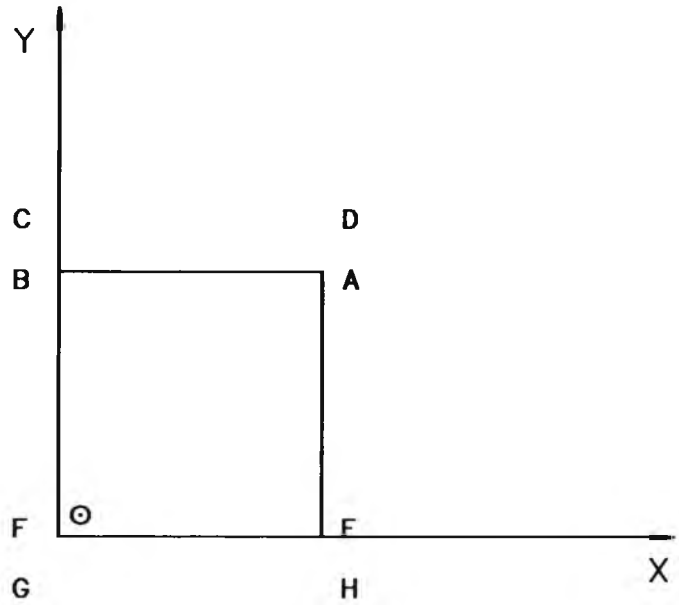

FIGURE 3.7

ILLUSTRATION OF ISOMETRIC CUBE

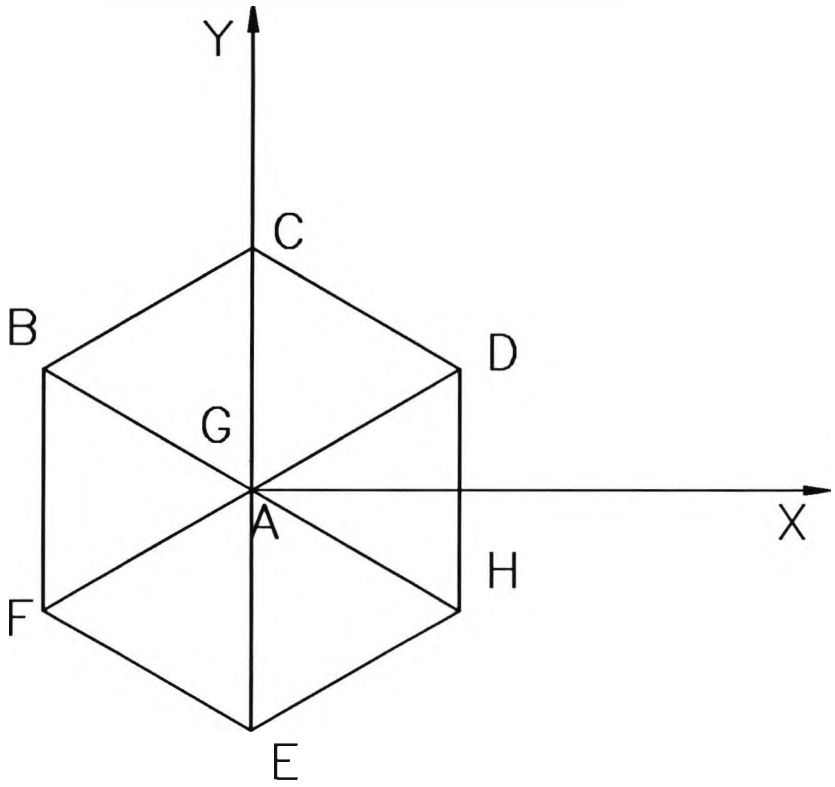

FIGURE 3.8

ISOMETRIC VIEW OF THE CUBE

This view from a point along the Z axis is called the isometric view of the cube [77]. Thus isometric view can be considered as a special case of rotation. An edge (in this case AE) is chosen to be directly in the line of sight and also this edge is drawn vertically, it is in fact tilted towards the viewer. This means the lines are foreshortened. All the lateral lines are at 30 degrees to the horizontal and these too are foreshortened in the same way as the vertical lines. Because isometric projection gives the same foreshortening to all the lines any figure can be drawn to its true size or to a suitable linear scale. Because of this property design engineers favour isometric projection to any other pictorial projection when dealing with three dimensional work. It is for this reason isometric sketching is chosen as the first stage to be implemented in the sketching input system proposed in this thesis.

The lines GC, GH and GD are taken to represent the Y, X and Z axes of the cube and in the isometric view they are called the isometric axes. Any line parallel to the axes are called the isometric lines and any planes parallel to these axes are called the isometric planes.

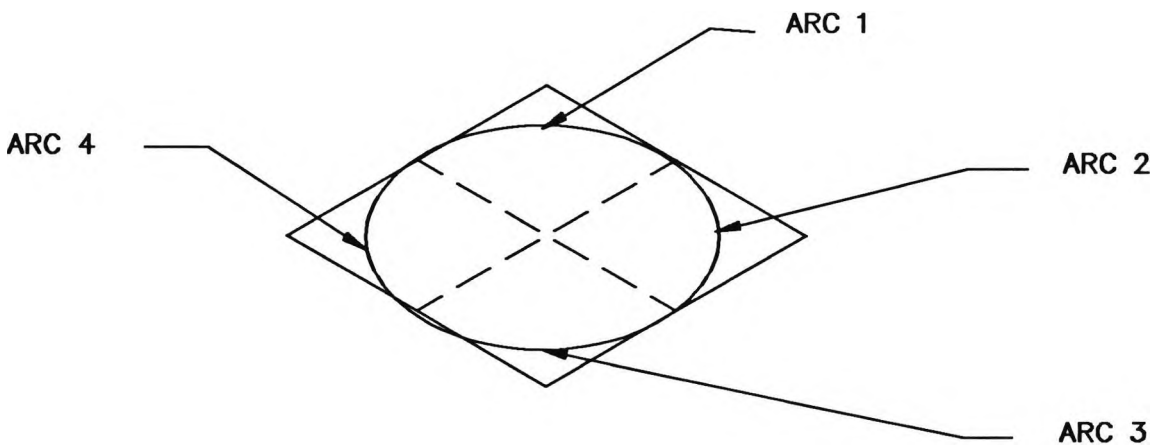

FIGURE 3.9

CONSTRUCTION OF ISOMETRIC ELLIPSE

Isometric sketching starts with the positioning of the three isometric axes and the origin representing the three mutually perpendicular axes and their meeting point in the object. One as seen earlier is vertical and the other two are inclined at 30 degrees to the horizontal. Luzador [78] explains how isometric sketching is performed. If the

object is of simple rectangular form it may be sketched by drawing an enclosing isometric box on the surface of which the orthographic views may be sketched. Projecting cylindrical features are enclosed in isometric prisms and circles are sketched within isometric squares. In sketching an ellipse to represent a circle pictorially, an enclosing rhombus, which is the isometric square, is drawn with sides equal to the diameter of the circle. The ellipse is constructed as a combination of four arcs. Figure 3.9 shows the construction of such an ellipse. Consider the isometric view of a cylinder as shown in figure 3.10. The isometric view will not show lines 1, 2, 3 and 4 but will show a curve inside the ellipse. However in a sketching situation it is customary to indicate a cylinder by the silhouette edges (refer section 2.4.1 and figure 2.16) which are indicated by line 1 and line 2. However line 3 and line 4 are easy to locate and sketch and also are much more precise than line 1 and line 2. Since it is a matter of convention to include the silhouette edges and the tangency edges (refer section 2.4.1) it is decided in this research to include line 3 and line 4 to indicate the cylinder and tangency edges wherever necessary.

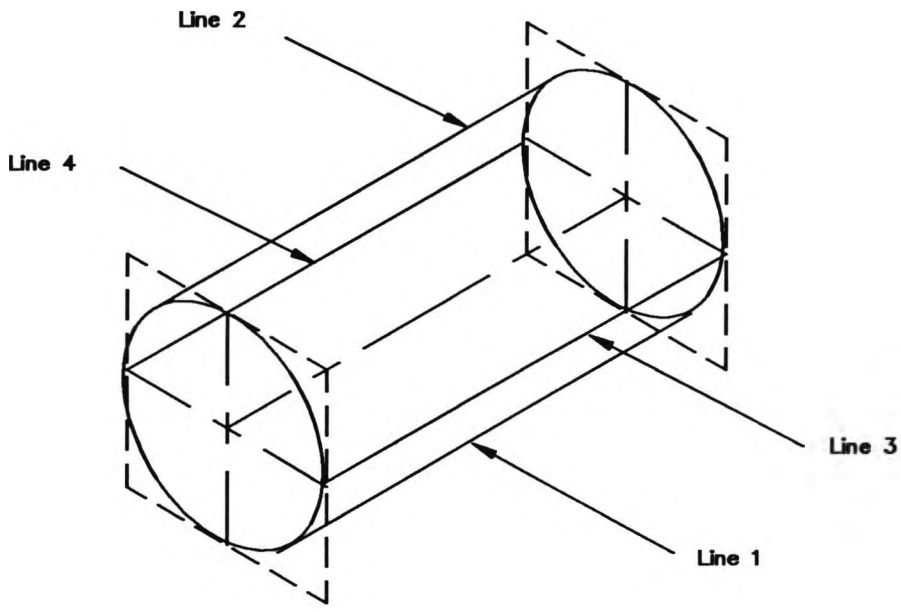

FIGURE 3.10

ILLUSTRATION OF ISOMETRIC CYLINDER

3.12 TWELVE CLASSES OF LINES IN AN ISOMETRIC SKETCH

In an isometric body made up of straight edges and cylindrical surfaces the following twelve classes of lines can be present. They are

1 Vertical isometric lines
2 30 degrees isometric lines
3 330 degrees isometric lines

4 Non-isometric straight lines
5 Isometric ellipses in XY plane
6 Isometric ellipses in YZ plane
7 Isometric ellipses in ZX plane
8 Non isometric ellipses
9 Isometric elliptical arcs in XY plane
10 Isometric elliptical arcs in YZ plane
11 Isometric elliptical arcs in ZX plane
12 Non isometric elliptical arcs

Consider the co-ordinate system shown in figure 3.11 representing the co-ordinate system used in this research, where x, y and z are the three dimensional axes of the isometric sketch while DX and DY are the horizontal and vertical axes of the sketch paper. Then by considering the slopes of the straight lines one can decide upon one of the four classes, each line fall into. For the ellipses and elliptical arcs the slopes of the major axis will indicate in which plane they are situated and the start and finish angle will indicate whether they are complete ellipses or arcs.

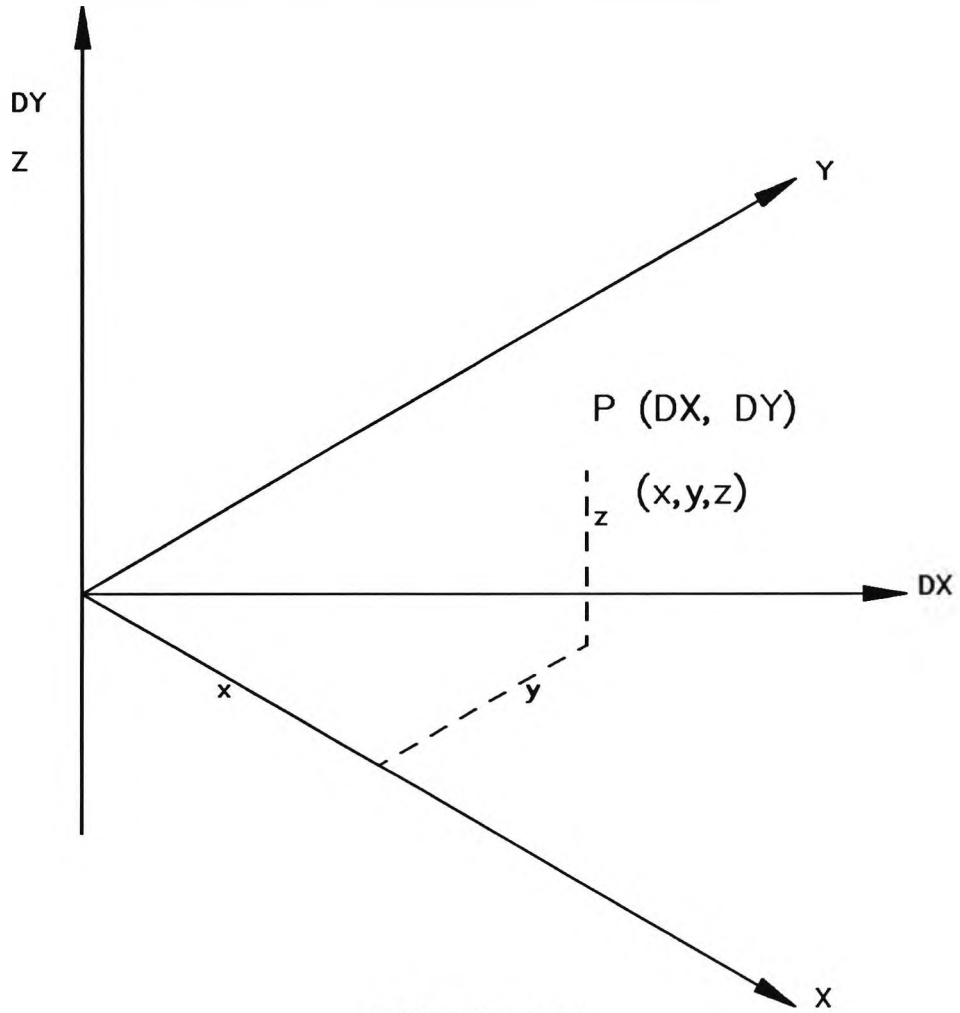

FIGURE 3.11

COORDINATE SYSTEM

Thus by analysing the slopes of the major axes and start and finish angles of ellipses it can be decided to which of the eight classes each curve falls into.

3.13 TRANSFORMATION TO 3D CO-ORDINATE SYSTEM

Consider the point P in isometric and digitizer axes as shown in figure 3.11. Its co-ordinates in 3D are (x y z) and in 2D are (DX, DY).

$$DX = x \text{ Cos } 30 + y \text{ Cos } 30$$
$$DY = -x \text{ Sin } 30 + y \text{ Sin } 30 + z$$

Since only DX and DY are known and there are three unknowns x, y and z another equation is required to get a unique solution for x, y and z. Here the information on the line class is used. Now consider the straight line connecting two points say P_1: (x_1 y_1 z_1) and P_2: (x_2 y_2 z_2). Then

$$DX_1 = x_1 \text{ Cos } 30 + y_1 \text{ Cos } 30$$
$$DX_2 = x_2 \text{ Cos } 30 + y_2 \text{ Cos } 30$$
$$DY_1 = -x_1 \text{ Sin } 30 + y_1 \text{ Sin } 30 + z_1$$
$$DY_2 = -x_2 \text{ Sin } 30 + y_2 \text{ Sin } 30 + z_2$$

This means another two equations are needed for a unique solution. If the line is (a) of type 1 then it is a vertical line and hence $x_1 = x_2$ (b) of type 2 then it is a 30 degrees isometric line and $z_1 = z_2$ and (c) of type 3 it is a 330 degrees isometric line and $y_1 = y_2$. Thus isometric lines get one extra equation. If now P_1 is known then there are three equations connecting x_2 y_2 and z_2 and hence would result in a unique solution. This point P_1 is the origin obtained during the sketching initialisation. Isometric lines emanating from this point terminates in vertices which could now be obtained in 3 dimensions using the method above. Continuing in this manner all connected lines and their corresponding vertices could be calculated. In the case of non-isometric lines it is customary to construct them from isometric lines and the construction lines could be used to identify the 3D co-ordinates. Whenever there is a loop unconnected it forms a hole and the 3-dimensional transformation needs an extra point in 3-dimension. Thus it could be seen that a vertex model could be formed from, the sketch and one point per hole in 3-dimensions.

3.14 TRANSFORMATION OF A CUBE - AN EXAMPLE

Figure 3.12 shows the isometric sketch of a cube with 100 units long edges. From a scaled drawing with each edge having a length of 100 units the following co-ordinates were obtained in two dimensions. Table below shows the co-ordinates in two dimen-

sions and three dimensions.

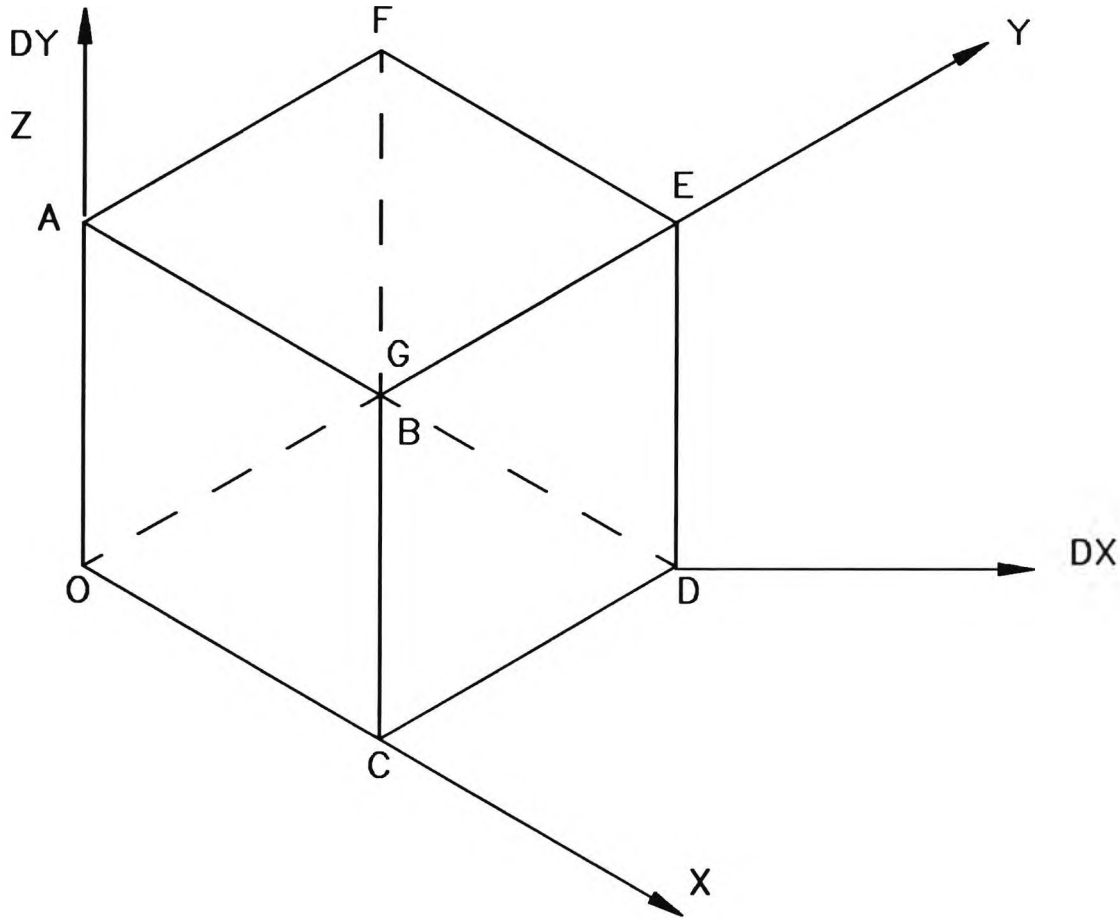

FIGURE 3.12

EXAMPLE CUBE

CO-ORDINATES IN THREE DIMENSIONS

VERTEX	X CO-ORD	Y CO-ORD	Z CO-ORD
A	0	0	0
B	100	0	100
C	100	0	0
O	0	0	0
D	100	100	0
E	100	100	100
F	0	100	100
G	0	100	0

CO-ORDINATES IN TWO DIMENSIONS

VERTEX	X CO-ORD	Y CO-ORD
A	0	100
B	87	50
C	87	-50
O	0	0
D	172	0
E	172	100
F	85	150
G	85	50

EDGE OA

Consider points O and A. Writing down the transformation equations for the point A will give

$$DX_a = x_a \text{ Cos } 30 + y_a \text{ Cos } 30$$

$$DY_a = -x_a \text{ Sin } 30 + y_a \text{ Sin } 30 + z_a$$

Since the line is of type 1 i.e. vertical isometric

$$x_o = x_a$$

But $x_o = 0$

Then $x_a = 0$

Since $DX_a = 0$ and $x_a = 0$, $y_a = 0$. Substituting these in DY_a gives $z_a = 100$.

EDGE AB

Considering points A and B the transformation equations could be written as

$$DX_b = x_b \text{ Cos } 30 + y_b \text{ Cos } 30$$

$$DY_b = -x_b \text{ Sin } 30 + y_b \text{ Sin } 30 + z_b$$

Since the line is of type 3 $y_a = y_b$

But $y_a = 0$

Hence from the transformation equations $x_b = 100$ and $z_b = 100$.

EDGE BC

Considering points B and C the transformation equations could be written as

$$DX_c = x_c \text{ Cos } 30 + y_c \text{ Cos } 30$$

$$DY_c = -x_c \text{ Sin } 30 + y_c \text{ Sin } 30 + z_c$$

But since the line is of type 1

$$x_b = x_c$$

But $x_b = 100$ and hence $x_c = 100$.

Now solving the transformation equations give $y_c = 0$ and $z_c = 0$.

EDGE CD

Considering points C and D the transformation equations could be written as

$DX_d = x_d \text{ Cos } 30 + y_d \text{ Cos } 30$

$DY_d = -x_d \text{ Sin } 30 + y_d \text{ Sin } 30 + z_d$

But since the line is of type 2 $z_c = z_d$.

But $z_c = 0$ and hence $z_d = 0$.

Now solving the transformation equations give $x_d = 100$ and $y_d = 100$.

EDGE DE

Considering the points D and E the transformation equations could be written as

$DX_e = x_e \text{ Cos } 30 + y_e \text{ Cos } 30$

$DY_e = -x_e \text{ Sin } 30 + y_e \text{ Sin } 30 + z_e$

But since the line is of type 1 $x_e = x_d$

But $x_d = 100$ and hence $x_e = 100$.

Now solving the transformation equations give $y_e = 100$ and $z_e = 100$.

EDGE EF

Considering the points E and F the transformation equations could be written as

$DX_f = x_f \text{ Cos } 30 + y_f \text{ Cos } 30$

$DY_f = -x_f \text{ Sin } 30 + y_f \text{ Sin } 30 + z_f$

Since the line is of type 3 $y_e = y_f$.

But $y_e = 100$ and hence $y_f = 100$.

Now solving the transformation equations give $x_f = 0$ and $z_f = 100$

EDGE FG

Considering the points F and G the transformation equations could be written as

$DX_g = x_g \text{ Cos } 30 + y_g \text{ Cos } 30$

$DY_g = -x_g \text{ Sin } 30 + y_g \text{ Sin } 30 + z_g$

Since the line is of type 1 $x_f = x_g$.

But $x_f = 0$ and hence $x_g = 0$.

Now solving the transformation equations give $y_g = 100$ and $z_g = 0$.

3.15 CIRCLES AND CIRCULAR ARCS IN ISOMETRIC PLANES

Circles in isometric planes of the actual object are depicted as ellipses in the isometric sketch. Therefore isometric ellipses should be transformed into circles and isometric arcs should be transformed into circular arcs. Consider figure 3.13 illustrating a cube with circles on its face. The bounding squares of the circles which are faces of the cube become rhombuses in the isometric representation, while the circles become ellipses.

However the points P_1, P_2 - - - P_8 remain to be the meeting points of the tangents. Thus if P_1, P_2 - - - - P_8 could be transformed into 3D representation, they will correspond to the meeting points of the tangents and the edges of the cube. Also the mid points of P_1P_3, P_3P_6 and P_4P_8 will give the centres of the circles. Thus dealing with the isometric ellipses becomes transformation of the meeting points and using these values to fix the diameter of the circle. These tangents are actually some construction lines. Still the transformation requires the finding of the 3D co-ordinates of these critical points P_1 to P_8 in figure 3.13. The slope of the major axis will indicate whether the circle is in the 'xy', 'yz' or 'zx' plane. The major axis is horizontal in the 'xy' plane, inclined at 45 degrees in the 'yz' plane and inclined at 135 degrees in the 'zx' plane. Section 3.15.1 explains how to find the slope of the major axis.

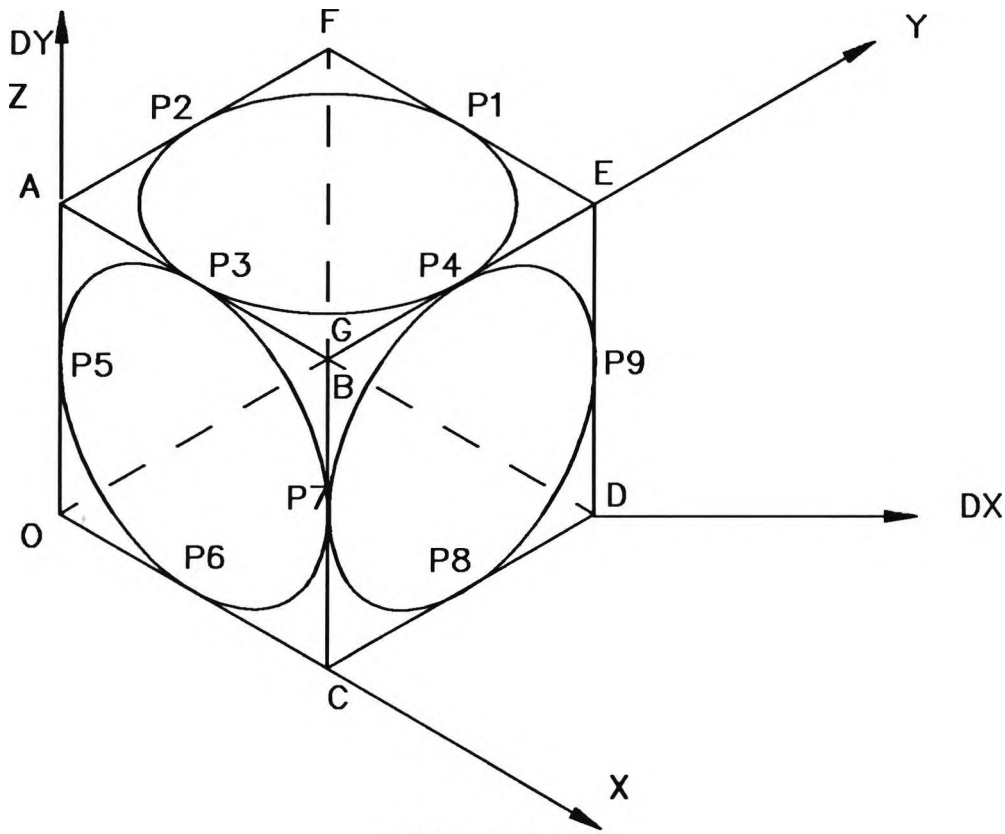

FIGURE 3.13

CUBE WITH CIRCLES IN THE ISOMETRIC PLANES

Dealing with isometric circular arcs is not as easy as the full ellipses. But it is made easier by the fact that it would be connected to some other edge and hence the three dimensional coordinate of the starting point and the ending point would be known. This therefore means that only the radius of the arc need to be found. To find this a generalised ellipse is fitted for the set of points as discussed earlier in section 3.4 and

the values for the semi major and minor axes are obtained. It could be shown that the radius of the circle is $\sqrt{2}$ times the semi minor axis in the following sub-section.

3.15.1 RADIUS OF THE CIRCLE IN XY PLANE

The circle in the xy plane is represented as an ellipse with horizontal major axis. In a similar way circles in the other isometric planes are also represented as ellipses. It could be shown that these ellipses have the same eccentricity. Hence it would be enough to find the relationship between the semi-major or semi-minor axis of the ellipse and the parent circle. Consider figure 3.14 where ABCD is a rhombus of unit size. Let the equation of the ellipse be $\frac{x^2}{l^2} + \frac{y^2}{m^2} = 1$. Then the tangent at $P_1(x_1\ y_1)$ is given by $\frac{x\,x_1}{l^2} + \frac{y\,y_1}{b^2} = 1$.

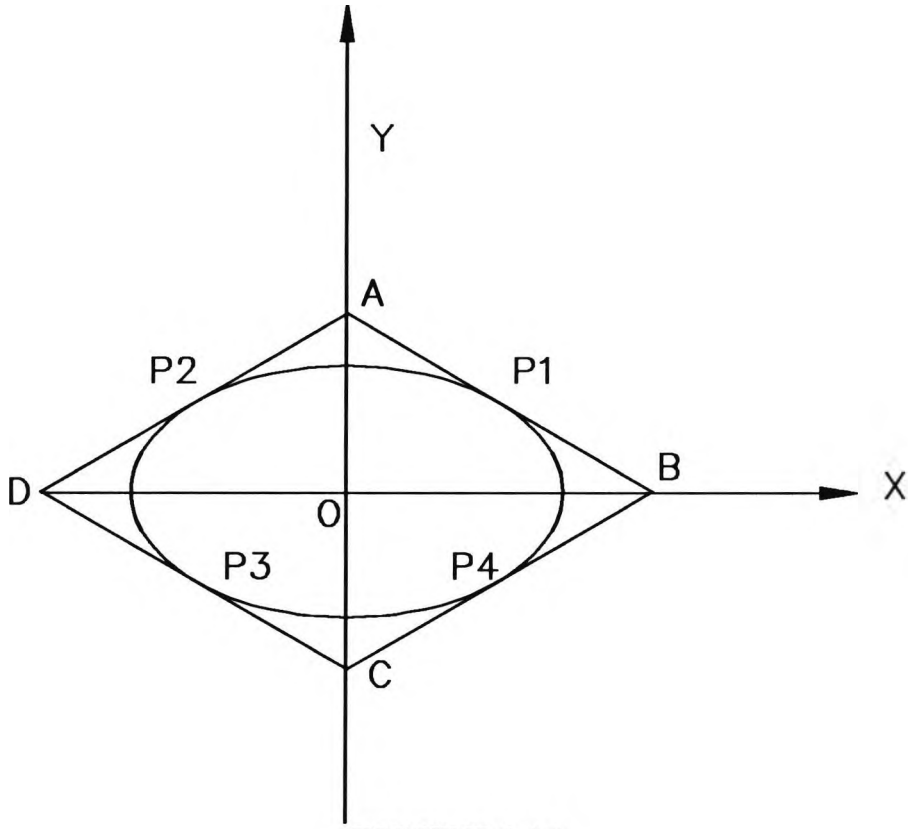

FIGURE 3.14

ELLIPSE ON XY PLANE

If P_1 is the mid point of AB then considering triangle AOB gives $x_1 = \frac{OB}{2} = \frac{AB\,Cos\,30}{2} = \frac{\sqrt{3}}{4}$. Similarly $y_1 = \frac{OA}{2} = \frac{AB\,Sin\,30}{2} = \frac{1}{4}$. Therefore tangent AB is $\frac{\sqrt{3}\,x}{4\,l^2} + \frac{y}{4\,m^2} = 1$. Since A(0 $\frac{1}{2}$) (ABC is isosceles) lies on this line

$\frac{1}{8m^2} = 1$ and hence m $= \frac{1}{2\sqrt{2}}$. Similarly it can be shown that $l = \frac{\sqrt{3}}{2\sqrt{2}}$.

In general the semi-major and semi-minor axes would be $\frac{\sqrt{3}}{2\sqrt{2}}$ times and $\frac{1}{2\sqrt{2}}$ times the diameter of the circle respectively.

The equation of the fitted ellipse with its axis system parallel to the reference axis system would be $x^2 + by^2 + fx + gy + c = 0$. It could be reduced to the standard form as $(x + \frac{f}{2})^2 + (y + \frac{g}{2\sqrt{b}})^2 = \frac{f^2 b + g^2 - 4bc}{4b}$. The centre therefore is $\left(\frac{-f}{2} \quad \frac{-g}{2\sqrt{b}}\right)$ and the radius is $\sqrt{2}\,\frac{4b^{1.5}}{f^2 b + g^2 - 4bc}$

3.15.2 IDENTIFICATION OF THE SLOPE OF THE MAJOR AXIS

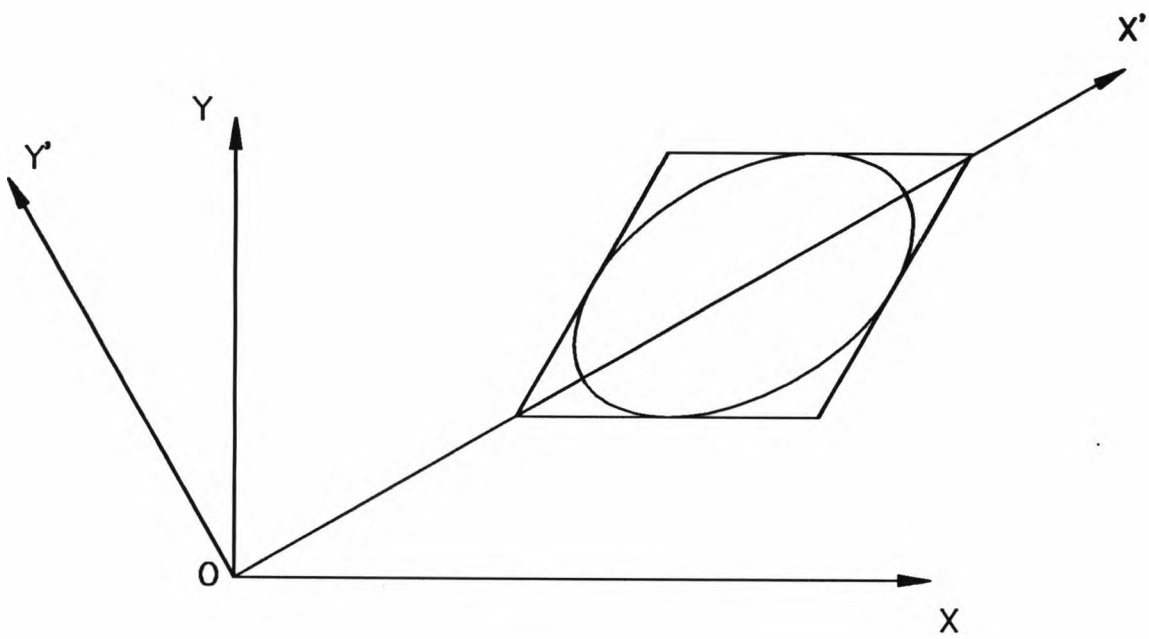

FIGURE 3.15

THE COORDINATE SYSTEMS

Identification of the slope of the major axis is achieved by considering the equation of the ellipse fitted, with respect to a system which is rotated and translated. This is well explained by Askwith [79] and Grieve [80]. Consider figure 3.15 indicating the xy and x'y' axis systems. If θ is the angle x'ox then

$$x = x' Cos\,\theta - y'\,Sin\,\theta$$
$$y = x'\,Sin\,\theta + y'\,Cos\,\theta$$

Now consider the equation

$$x^2 + hxy + by^2 + fx + gy + c = 0$$

If x and y are now replaced by the corresponding terms in x' and y'

$$x^2 = \left(x' Cos\,\theta - y'\,Sin\,\theta\right)^2$$
$$b\,y^2 = b\left(x'\,Sin\,\theta + y'\,Cos\,\theta\right)^2$$
$$hxy = h\left(x'\,Sin\,\theta + y'\,Cos\,\theta\right)\left(x'\,Cos\,\theta - y'\,Sin\,\theta\right)$$
$$gx = g\,(x'\,Cos\,\theta - y'\,Sin\,\theta)$$
$$fy = f\,(x'\,Cos\,\theta + y' Sin\,\theta)$$
$$c = c$$

If now the new equation is written as $x'^2 + h'x'y' + b'y'^2 + f'x' + g'y' + c' = 0$ then the coefficients should be as follows:

$$1 = Cos^2\theta + b^2\,Sin^2\theta + h\,Cos\,\theta\,Sin\,\theta$$
$$b' = b^2\,Cos^2\theta - h\,Sin\,\theta\,Cos\,\theta + Sin^2\theta$$
$$h' = (b - 1)\,Sin\,2\theta + h\,Cos\,2\theta$$
$$g' = g\,Cos\,\theta + f\,Sin\,\theta$$
$$f' = -g\,Sin\,\theta + f\,Cos\,\theta$$
$$c' = c$$

If the axis system x'y' and the axes of the ellipse are parallel then the x'y' term should vanish i.e. h' = 0. This means $Cot\,2\theta = \frac{1-b}{h}$. This θ is the angle, the major axis of the ellipse makes with the original x axis and thus forms the slope of the major axis of the ellipse. The centre of the ellipse could now be found by reducing the new equation to the form $\frac{x'^2}{a^2} + \frac{y'^2}{b^2} = 1$. From this the following could be said about the isometric ellipses.

(a) xy ellipses will have 'h' = 0

(b) yz and zx ellipses will have 'b' = 1 i.e. x^2 and y^2 terms will have the same co-efficients.

Once θ is known the co-efficients could be found for b', g', f' and c'. Now the equation of the ellipse reduces to the familiar $x'^2 + b'y'^2 + f'x' + g'y' + c' = 0$.

CHAPTER 4

REQUIREMENT ANALYSIS AND FUNCTIONAL SPECIFICATION

4.0 GENERAL

Chapter 2 established the requirements of a solid modelling system suitable for sketching input while chapter 3 discussed the necessary background theory. This chapter analyses the requirements made on the software, its implementation details and functional specification. Requirement analysis looks at the desirable features of the sketching input system from the point of view of the user, the design engineer. These requirements are then analysed in stages together with hardware and theoretical considerations by defining the problem. In the process of establishing the statement of the problem, activities and milestones are esablished. Finally the functional specification is drawn. Functional specification here means the requirements set and met in the production of the sketching input system.

4.1 REQUIREMENT ANALYSIS

In this section the requirements of the software is analysed from the point of view of the user, the design engineer. Sketching is the fundamental part of design and subsequent drafting and as a result almost all books in engineering graphics describe how sketching should be done [81-85]. The degree of perfection required in a given sketch depends upon its use. Sketches hurriedly made to supplement oral description may be rough and incomplete. On the other hand if sketch is the medium of conveying important and precise information to engineers, it should be created as carefully as possible. The term 'free-hand sketch' is too often understood to mean a crude or sloppy free hand drawing in which no particular effort has been made. On the contrary, a free hand sketch should be made with care. The sketches considered here are those made with lot of care. The following subsection 4.1.1 summarises the method of sketching to facilitate the decision on the requirements of a sketching input system.

4.1.1 SKETCHING PROCESS

The basic components of a sketch are straight lines, circles and arcs, and ellipses. Experience in making the sketch provide the following tips in drawing different types of lines[77].

To make the straight lines well

(a) Hold pencil naturally about 1.5 inches back from the point and approximately at right angles to the line to be drawn.

(b) Draw horizontal lines from left to right with a free and easy wrist and arm movement.

(c) Draw vertical lines downward with finger and wrist movement.
(d) Inclined lines may be made to conform in direction to horizontal or vertical lines by shifting position with respect to the paper or turning the paper slightly.
(e) In sketching long lines mark the ends of the lines with light dots and move the pencil back and forth between the dots in long sweeps, keeping the eye always on the point towards which the pencil is moving.
(f) An easy method of blocking in horizontal or vertical lines is to hold the hand and pencil rigidly and glide the finger tips along the edge of the pad or board. Another method is to mark the distance on the edge of a card or a strip of paper and transfer this distance at intervals, and then draw the finals through these points.

To sketch circles and arcs well
(a) First sketch lightly the enclosing square.
(b) Mark the mid-points of the sides, draw light arcs tangents to the sides of the square.
(c) Draw the final circle or arc overlapping the light arcs in (b).
(d) An excellent method, particularly for large circles is to mark the estimated radius on the edge of a card (trammel) or scrap paper to set off from the centre as many points as desired and to sketch the final circles through these points.
(e) In sketching tangent arcs, always ensure that the point of tangency is well approximated.

To sketch ellipses and elliptical arcs well
(a) Sketch lightly the enclosing rectangle
(b) Mark the mid-points to the rectangle
(c) Draw light tangent arcs
(d) Complete the ellipse or arc
(e) An alternate trammel method may be useful in sketching large ellipses.

The most important rule in free-hand sketching is to keep the sketch in proportion. To ensure this first the relative proportions of the height to the width must be carefully established. Then as progress is made towards the medium sized areas comparison must be made constantly. A useful tip in this connection is blocking where the paper is divided into rectangular blocks to accomodate sections of the actual sketch.

4.1.2 ISOMETRIC SKETCHING

Isometric sketching as described in section 3.11 relies on the fact that the x and y axes are inclined at 30° to the horizontal while the z axis remains vertical. Straight lines parallel to the axes called the isometric lines are easy to sketch. However lines on an object, located by angles called non-isometric lines are difficult to sketch. This is because angles cannot be laid off directly on an isometric drawing as they do not appear in their true sizes. Lines positioned by angles are drawn by fixing their ends by ordinates, which are isometric lines[81]. Blocking or boxing is a standard technique

used in isometric sketching.

When objects having cylindrical or conical shapes, are placed in the isometric or other oblique positions, the circles will be viewed at an angle and will appear as ellipses. The most important rule in sketching isometric ellipses is: The major axis of the ellipse is always at right angles to the centre line of the cylinder and the minor axis is at right angles to the major axis and coincides with the centre line. To sketch a good ellipse follow the steps detailed in section 4.1.1. An isometric paper is a paper with lines at equal spacing parallel to the x, y, and z axes in an isometric projection. This is very useful to keep the sketch in proportion.

4.1.3 ANALYSIS

From the preceding sections the following conclusions could be drawn.
(a) Even though a free-hand drawing is not laid out to an exact size or scale, the finished drawing must show the relative proportions of the illustrated material. This can be achieved first by visualizing a drawing of correct proportions of the illustrated material to fit the available working space of the drawing paper.
(b) Proportions of the details of any sketch can be controlled by the construction of skeleton boxes whose dimensions are equal to the overall dimensions of the proposed view. The basic outlines of the drawing are constructed within the boxes and are checked for proper proportion and position before adding smaller details. This means there will be some redundant lines used only for proportioning.
(c) When constructing ellipses and long straight lines curves and points are drawn which are overdrawn in the finished sketch thus making them redundant. This and the redundant lines in (b) above indicate the need for a kind of data which need not be stored in the computer in the proposed sketching input system.
(d) With free-hand sketching, pencils with different hardness gradings are used to create different types of lines. Strong, bold, free-hand lines are used to indicate the edges in the object while faint lines are used to indicate the centre lines and construction lines.

4.1.4 THE REQUIREMENTS OF A SKETCHING INPUT SYSTEM

(i) For the sketching input system proposed the inputs are free-hand isometric sketches made up of four classes of lines namely

(a) centre lines
(b) visible lines
(c) hidden lines
(d) construction lines

Therefore the first requirement is the provision for the entry of these lines. It is however is not necessary to have all these lines in all the outputs, and often the visible lines alone are sufficient to illustrate the sketch. Therefore it is necessary to store

these lines separately in the computer memory so that they could be accessed separately and independantly.

(ii) In a traditional sketching environment the visible lines and hidden lines are sketched with pencils of differing hardness. Selecting from the menu should resemble this and must be simple like this.

(iii) As seen earlier sketching is the physical expression of the design activity and the designer should be left free to carry on with the design process. This means there should be minimum of interaction (or distraction). To effect this requirement

(a) The menus must be kept to a minimum

(b) No input should be needed once the sketching commences i.e. there should be no necessity to indicate the start and end of a line segment.

(iv) One of the main requirement of sketching is the ability to erase whole or part of a line. This process of erasing a line should be simple and easy.

(v) When a sketch is made it is often made to rough sizes and hence subsequent extension is an essential feature for the sketching input system.

(vi) Free hand sketching invariably has overstriking of lines and provision should be made to accommodate them.

(vii) The user may at times start the sketch from both ends of a straight line or circle to meet at an interim point. There should be provision to accept them.

(viii) The program should identify the line segments automatically whether the sketching incorporates two consecutive or discrete lines.

(ix) It should automatically identify the vertices.

(x)The program should automatically recognise the straight lines, ellipses and arcs and process them accordingly.

4.2 SYSTEM

The system in which the sketching input software was developed, comprised the following.

(a) An IBM PS/2 model 60 computer
(b) MS DOS Version 3.3 Operating system
(c) CALCOMP 2000 series digitizer

4.2.1 OPERATION OF THE DIGITIZER

The 'CALCOMP' 2000 series digitizer converts graphic information into a digital

form that is suitable for entry into the computer. By simply actuating the transducer over any position on a map, diagram, menu or other graphic presentation the co-ordinates of that position are transformed into their digital equivalent. The digitizer has a tablet and a stylus. The tablet contains a grid of wires positioned in the horizontal (x) and vertical (y) directions. These are used to locate and identify co-ordinate points relative to the tablet origin, at the lower left corner of the tablet. The stylus contains a switch and a replaceable ballpoint cartridge. The stylus is used to select points to be digitized. Pressing the stylus down lightly against the tablet surface initiates the digitizing of data. The stylus is similar to a pen and the action of actuating is similar to sketching. The data transmission parameters are set by the three switch banks at the back of the tablet. They set the baud rate, data bits, stop bits, parity, mode of operation, and the number of points transmitted per second. The operating mode could be selected on a permanant basis by the switches or alternately it could be controlled by the program. To digitize selected points

(a) Select the operating mode (may be preselected by the switches)

(b) Place material to be digitized on the active surface of the tablet

(c) Position the stylus on the point and

(d) press the stylus down.

The digitizer will locate and identify the digital x,y co- ordinates of the point relative to the origin at the lower left corner of the tablet.

In the point mode, actuating the transducer by depressing the stylus on the tablet while in the active area of the tablet causes one x,y co-ordinate pair to be output in the appropriate format. In the track mode, x,y co-ordinate pairs will be output continuously at the selected sampling rate as long as, and only when, the transducer is activated while in the active area of the tablet surface. This is the opearting mode suitable for sketching input. The output from digitizer to the computer is (x y 1) with a carriage return. It is connected to the RS 232 communication port 1 in the PS/2 computer.

4.3 STATEMENT OF THE PROBLEM

It is evident from section 4.2, the sketched lines whether they are straight lines or curves or, object lines or other lines they all would be transmitted to the computer as series of points, each point being represented by its x and y co-ordinates. It therefore becomes necessary to have a menu to differentiate them. This menu could be displayed in the screen requiring an interaction through the keyboard or one in the digitizer itself requiring the activation of a point in the specified menu area. Choosing an item from the digitizer menu resembled the picking of a pencil and hence it was decided to have the menu in the digitizer. Erasing a line wholly or partly is the important requirement in sketching. The first option considered was cris cross cutting which was found to be inefficient and computationally expensive. To indicate a line or a portion of a line there are no recognised lines, during the early stages of sketching.

After careful consideration it was decided to include erased lines as a line type such as centre lines for subsequent processing. This enables the designer to select the 'erase line' option from the menu and run the stylus over the part of the line to be erased. The points transmitted then are stored as erased points, fitted as erased lines and are removed from the final list of lines. In the analysis of requirements in section 4.1 it was found that the certain amount of redundant lines which need not to be stored could be present in a sketching situation. The menu therefore was designed to accomodate this facility. Thus it was decided that the menu should have

(1) Visible lines
(2) Hidden lines
(3) Centre lines
(4) Construction lines
(5) Erase a line
(6) Redundant line
(7) End

Figure 4.1 shows the digitizer menu used in this implementation.

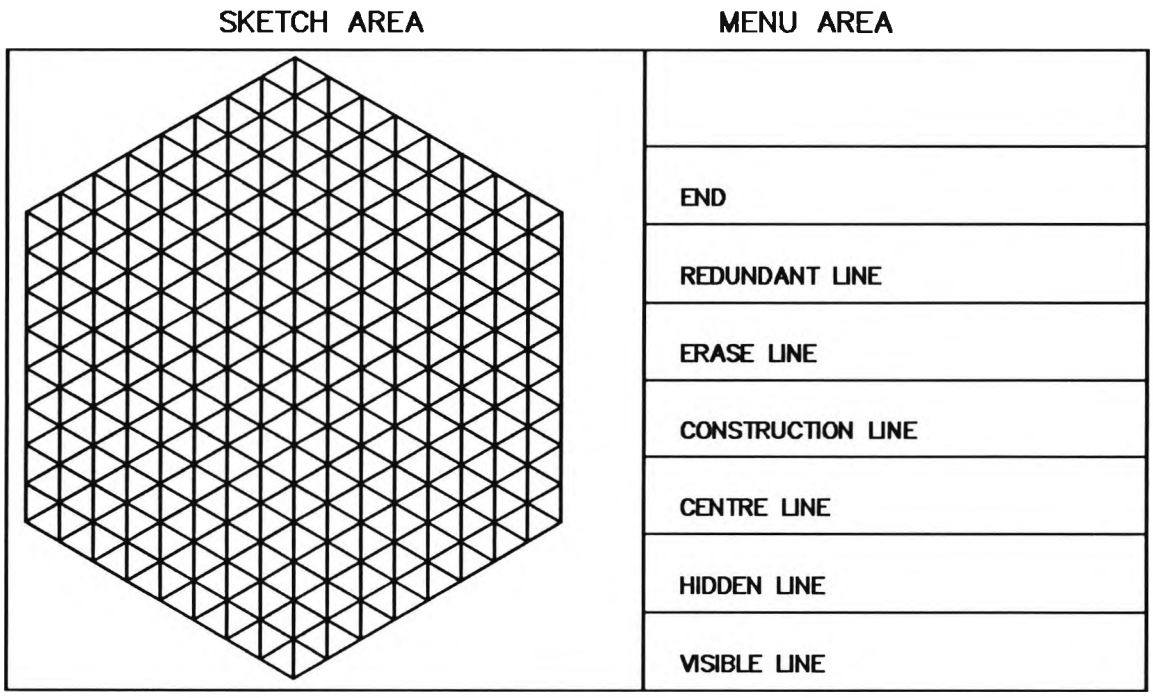

FIGURE 4.1

DIGITIZER MENU

The requirements of facilities to accomodate overstriking, subsequent extensions and starting from both ends are essential features for a good sketching input system. Since

the lines are transmitted as series of points, subsequent addition of points to a segment proved difficult. An acceptable solution to this problem is to fit these points as two or more different lines and do a "zero distance merging". In the zero distance merging, the equations of two lines are checked and if it is found to be same the end points are compared. If one end point in both the lines overlapped then the overlapping vertex is removed together with those two edges and a new edge, with the other two ends of the original edges, is introduced. The difficulty with this approach was the overlapping of different edges in an isometric view. To accommodate this an interactive merge becomes necessary. A similar approach could be used for dealing with the erased lines.

The purpose of this research is to develop a module which could be "bolted on" to some existing solid modelling system. Most of these are implemented in workstations running under different operating systems. System dependent or device dependent portions of the software should therefore be kept separate and to a minimum. The communication part between the digitizer and the computer are system and device dependent and often device drivers are used to facilitate communication between the digitizer and the computer. It was therefore decided to separate the process of inputting the sketch from the processing of the sketch. The points transmitted are written into an ASCII file, which could be read for the subsequent processing. This processing part consists of the following:
(a) Breaking the conglomerate of points in each of the five classes of line, into groups belonging to individual line segments
(b) Fitting the best possible curves for each of these line segments
(c) Identify the end points of these line segments, called terminal points, which are close enough to be considered as vertices
(d) Establishing the vertices
(e) Establishing the edges after, performing zero edge merging and erased line merging
(g) Using the origin in 3D, establishing the 3D co-ordinates of the connected graph
(h) Establishing the presence of holes and obtain the connection between the holes and the bodies from the designer
(i) Identifying all clockwise loops in the body
(j) Establishing the solid model

The backround theory for these steps listed above is given in chapter 3. The software developed, "SKETCH-SOLID", could be conveniently divided into three phases. In the first phase the sketch made on the digitizer is transformed into a series of points in the five classes of lines and written into a file. In the second phase the sketch is processed in two dimensions to establish the vertices and edges in two dimensions. In phase 3 this processed sketch and the information about the three dimensional origin is used to transform it first into a vertex model and finally into a solid model.

4.4 FUNCTIONAL SPECIFICATION

Whatever activity a software is expected to perform the software will have two basic constituents. They are the data structures to store them in the RAM and in the back up memory and the instructions which make the computer to carry out operations in these data structures. This section describes the data structures and the operations to be performed on them in the software "sketch solid".

(a) The points transmitted in the sketching stage are stored in five different arrays for the five different classes of lines together with the number of points in each array. Once the sketching is completed they are written into a file with an extension ".pnt" where the first entry is the co-ordinates in three dimensions and the next five entries give the number of points in each of the class which are followed by the x and y coordinates of the points. Each point occupies a separate line.

(b) The processing in two dimensions part asks for the name of the file and checks whether a file in that name exist otherwise it warns the designer about it. It then allocates memory for the arrays to accommodate the specified number of points in each of the five classes and reads the points into these dynamic arrays.

(c) In the processing in two dimensions stage five linked lists are established in the following way, for each of the five line types:

(i) **Lineseg** - a linked list having (ideally) "the number of edges" nodes, each having the details of one particular line segment. The details included are the row number of the starting point in the particular array (described in (b) above) the number of points in that line segment, the co-ordinates of the starting point and the finishing point of that line segment and an indicator to say whether it is a straight line or curve in the +ve or -ve direction.

(ii) **Geom_edge2d** - a linked list having the "number of edges" nodes, each corresponding to the nodes in (i) above and having the details of the equation of the curve fitted.

(iii) **Edge2d** - a linked list having the "number of edges" nodes, each corresponding to the nodes in (i) and (ii) above and having the starting vertex number, the finishing vertex number and the line type, one out of the twelve types described earlier in chapter 3.

(iv) **Termpoints** - a linked list having (ideally) "the number of vertices" nodes each having the coordinates of an end point in Lineseg nodes ((i) above) and the numbers of the Linesegs which have one end point in the vicinity of this point.

(v) **Vert2d** - a linked list having corresponding nodes to the list "Termpoint" having the finalised co-ordinates of the vertices.

These lists of the five classes of lines are then merged to form the final two dimensional model. The five nodes are schematicallyshown in figure 4.2.

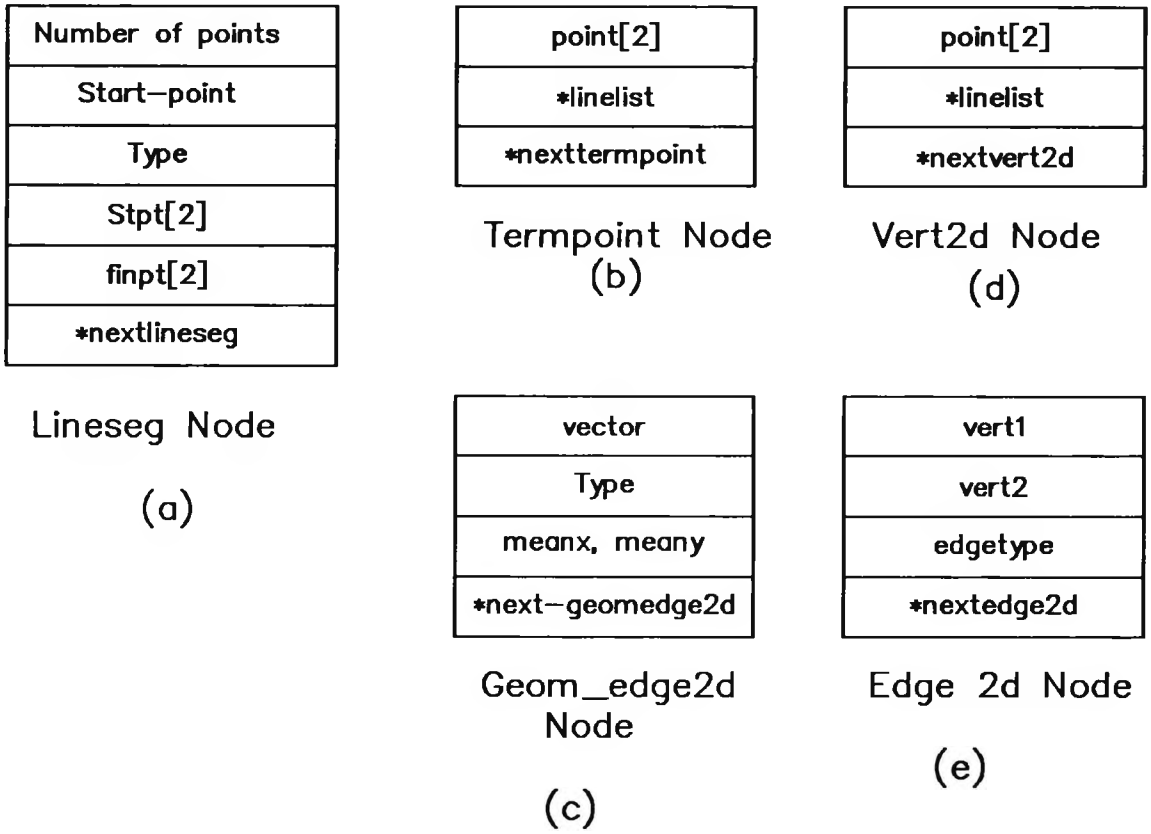

FIGURE 4.2

NODES USED IN TWO DIMENSIONAL PROCESSING

(d) These linked lists should then go through an interactive merge facility which merges the linesegments belonging to the same line in an interactive manner.

(e) The output linked lists are then passed through the erase merge system to remove the lines that are erased.

(f) The final set of visible lines and the hidden lines are then merged together to form the list of lines in the solid.

(g) In the processing in 3 dimensions stage again five linked lists are established. In this process, first the 3 dimensional co-ordinates of the vertices are established as described in chapter 3. During this process details of holes are also established.

(h) Using these 3 dimensional co-ordinates and the connectivity details given in the linked list edge2d, all clockwise loops are established.

(i) Using the details of holes and the loop details equation of faces are established.

(h) The details of the five linked lists established are as follows:

(i) List of vertices
(ii) List of edges
(iii) List of faces
(iv) List of loops
(v) List of rings

The details of the constituent nodes are given in figure 4.3.

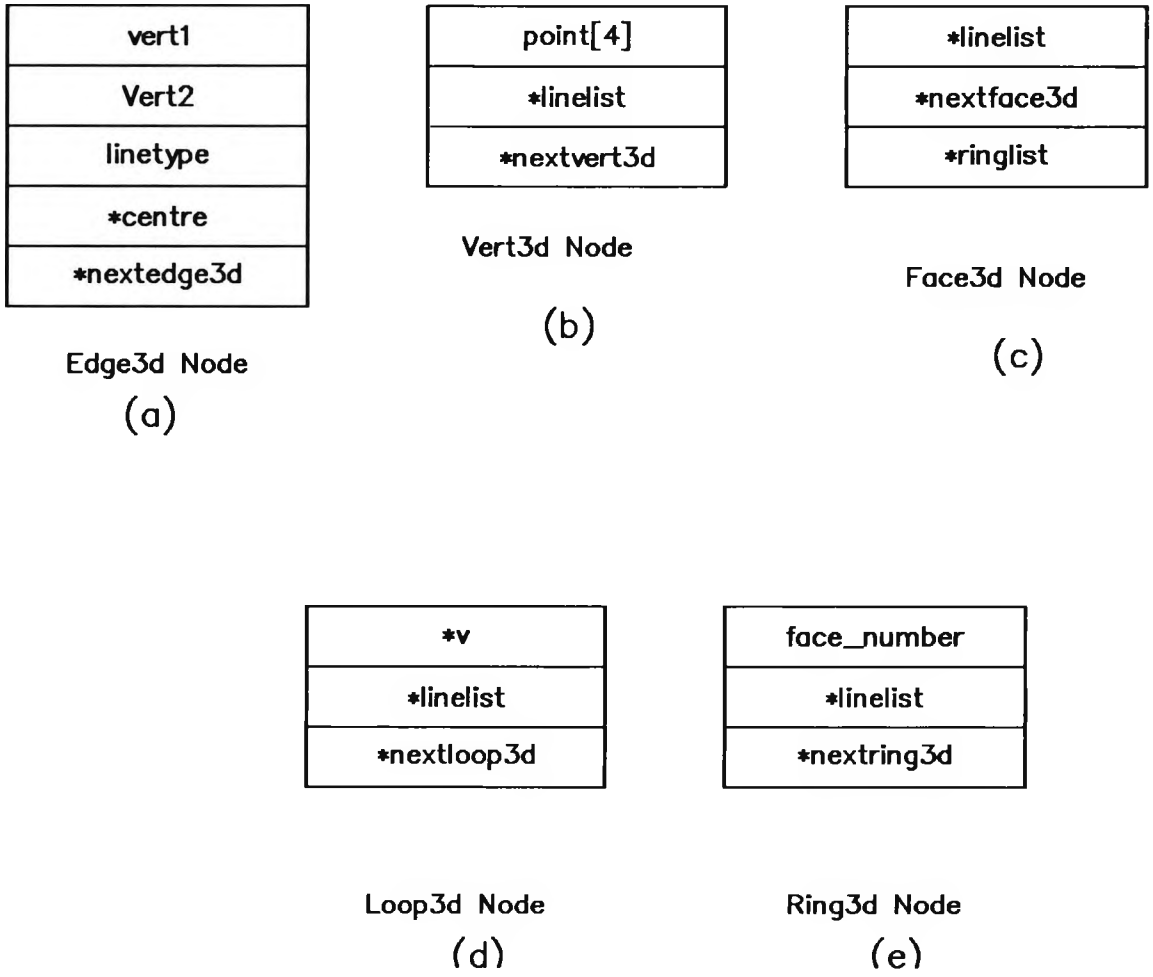

FIGURE 4.3

FIVE NODES FOR THREE DIMENSIONAL PROCESSING

CHAPTER 5

THE PROGRAM

5.0 INTRODUCTION

The program meeting the functional specification established in chapter 4 could have a menu having the following items:

(a) Sketching
(b) Processing in two dimensions
(c) Merging of the processed lines in (b) above to obtain lines in solid
(d) Processing in three dimensions and
(e) Building the solid model

In the sketch part the program accepts the sketch as a series of points and classifies and stores them in a file. In the 'processing in two dimensions' stage this file is read and the points are broken into subgroups belonging to individual line segments. These individual group of points belonging to the line segments are then fitted with analytic equations and the vertices are identified. In the merging part the lines fitted are first checked and merged into one line wherever appropriate. The erased lines which are stored separately are then removed. Finally the lines belonging to the solid are established by combining the visible lines and hidden lines. In the 'processing in three dimensions' stage the three dimensional origin in the solid (or any other point in the solid) is used to find the three dimensional co-ordinates of the vertices and other important points. These three dimensional co-ordinates and the first clockwise loop obtained from the user are then used to obtain all clockwise loops in the solid. If there are holes in the body they will be identified at this stage and extra information would be needed to process them. In this implementation for isometric sketching input, this is achieved through construction lines. These information about the loops would then be used to fit the equations of faces and rings. At this stage it will be possible to check whether the solid under development is satisfying the Euler-Poincare formula. If it is not satisfying the user should go to the sketching stage and add or delete edges. In the final part of the program the solid model could be built in such a manner that could be used by some existing solid modeller. The structure diagram of the program is given in figure 5.1. The structure allows the user to go backwards and forwards between the five stages namely

(a) Sketching
(b) Processing in two dimensions
(c) Merging
(d) Processing in three dimensions and
(e) Building the solid model

until he is satisfied of the model he has built. He could then exit the system by selecting exit. A solid model is complete when it passes all the five stages mentioned above.

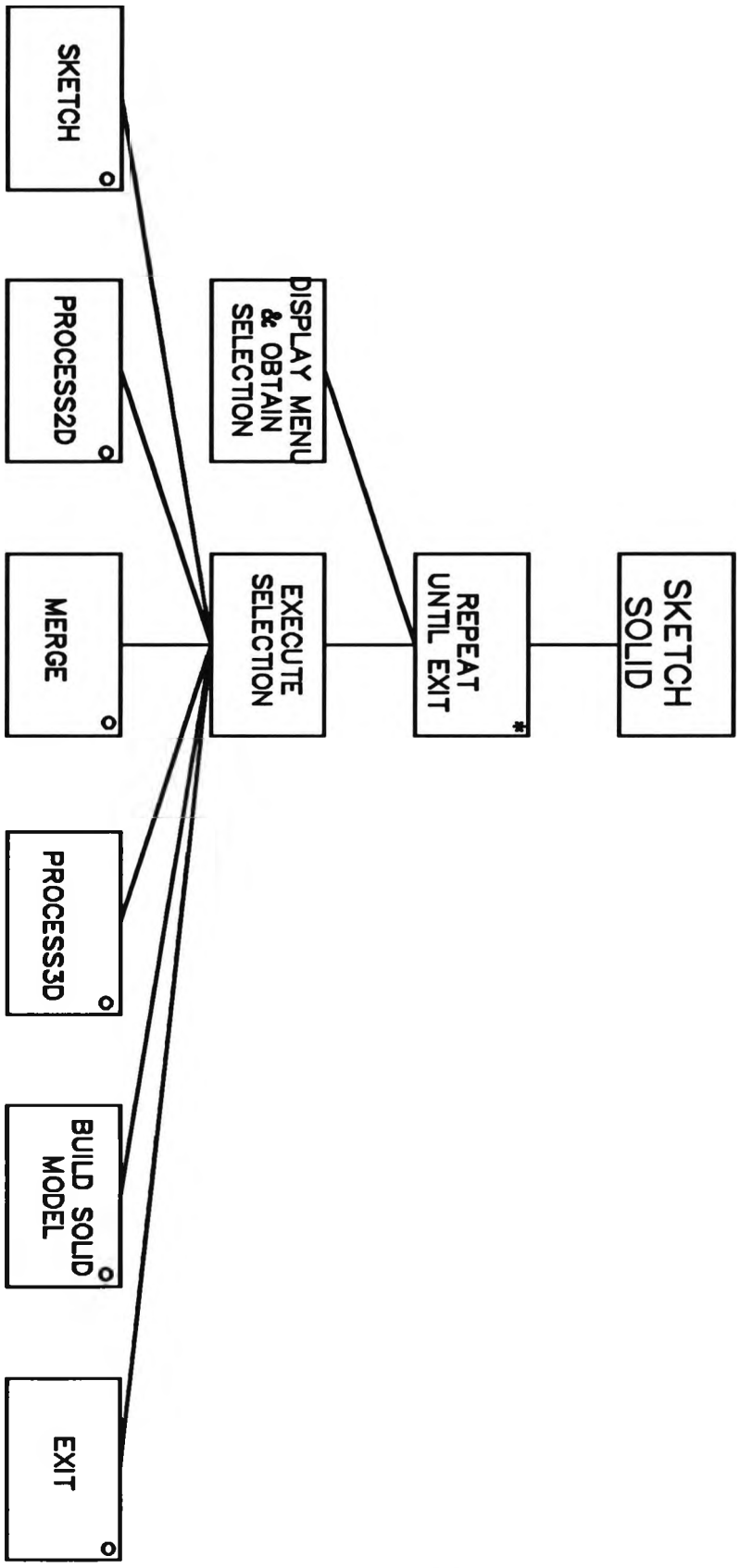

FIGURE 5.1

STRUCTURE OF THE SKETCH - SOLID PROGRAM

At the end of each stage a file is created and items are written onto it. In the sketch stage the name of the sketch is obtained and an extension '.pnt' is catenated to it and a file in this name (with the extension .pnt) is opened to write the points in the sketch. At the end of processing in two dimensions a file with the same filename and an extension of '.one' is created to contain the details of the findings. Similarly at the end of merging a file with the extension '.two' and at the end of three dimensional processing a file with the extension '.thr' are created.

5.1 SKETCH STAGE

The sketch part of the program is written to accept and store the sketching input from the user. Once the input is complete, the stored sketch is accessed by the 'processing in two dimensions' stage. If at a later stage the user realises that he needs to add more lines or remove some lines he could come back to this stage and the old sketch could be appended. As seen earlier the sketch made on the paper placed on the digitizer is transmitted to the computer as co-ordinates of points in the line segment sketched. The program accepts these points in six different classes in accordance with the digitizer menu (see figure 4.1). They are

(a) Points in visible lines
(b) Points in hidden lines
(c) Points in centre lines
(d) Points in construction lines
(e) Points in erased lines
(f) Points in redundant lines

Whatever the class of line the point belongs to, when a point is received it is first checked to find out whether it comes from the menu area or sketch area as shown in figure 4.1. When a point comes from the menu area it sets the current value of the menu selection which remains the same for all the processing that takes place until it is changed by another point coming from the menu area. Thus if the menu selection is set as visible lines and 200 points are sent after that, all those 200 points would be stored in the visible line class.

When the point received is from the sketch area the program first checks whether it is a redundant line or not. If it is a redundant line the point is ignored. All the points following this are also ignored until the menu selection is changed. When the menu selection is not redundant line the point is checked to ensure whether it is at least 15 units (1.5mm) away from the last point received in the sketch area. If it is, then the point is stored in its respective class.

This part of the program is written in IBM BASIC language in this implementation. This allowed an easy way of creating a sketching input without any device driver or other system dependant software involvement. The points are stored in five different

arrays namely VISLINES, HIDLINES, CENTLINES, CONSTLINES and ERASLINES. The array VISLINES can accommodate 5000 points while all other arrays could accommodate 1000 points each. At the begining of the stage the computer will ask the user for the filename and whether it is a new sketch or old. If it is old the file would be read into the arrays before accepting any new points. New lines could then be added to the sketch. If it is a new sketch the program asks the user to indicate the three dimensional origin in the sketch which is assumed to be the two dimensional origin as well (see section 3.13) during the three dimensional transformation. All these steps are collectively called initialisation in this thesis.

5.1.1 SKETCHING INTO THE SYSTEM

Once the initialisation is complete a message will appear asking the user to start or proceed sketching using the digitizer menu. Touch one of the seven items on the menu (similar to picking a pencil) and proceed with sketching. A message will appear on the screen telling the class of the line being sketched. Each time a point is received, distance checked and found acceptable, the point count in that class is incremented and the point is stored in the respective array. The process of receiving a point, checking the distance and storing the point if found suitable, continues until a point from the menu area is received. This could mean the selection of a new class of line or ending the sketching process. If a new class of line is selected the process described above would be repeated and the points would be stored in a different appropriate array. When 'END' is selected the points in the five different arrays and the three dimensional origin are written into the file with the extension '.pnt'. Figure 5.2 shows the structure diagram of the sketching process.

5.2 PROCESSING THE SKETCH IN TWO DIMENSIONS

The processing in two dimensions involves the generation of the five linked lists of nodes discussed in figure 4.2 for each class of line. A function (sub-program) in 'C' is written to implement this process. Figure 5.3 shows the structure diagram of this function. It has five algorithms implemented inside it. They are

(i) Finding the line segments
(ii) Finding the terminal points
(iii) Finding the geometry
(iv) Finding the vertices and
(v) Finding the edges

In the implementation of these algorithms several other functions are written and used.

5.2.1 FINDING THE LINE SEGMENTS

This algorithm takes an array of points and break them into groups belonging to individual line segments. The basic assumptions are that there is no line with a length

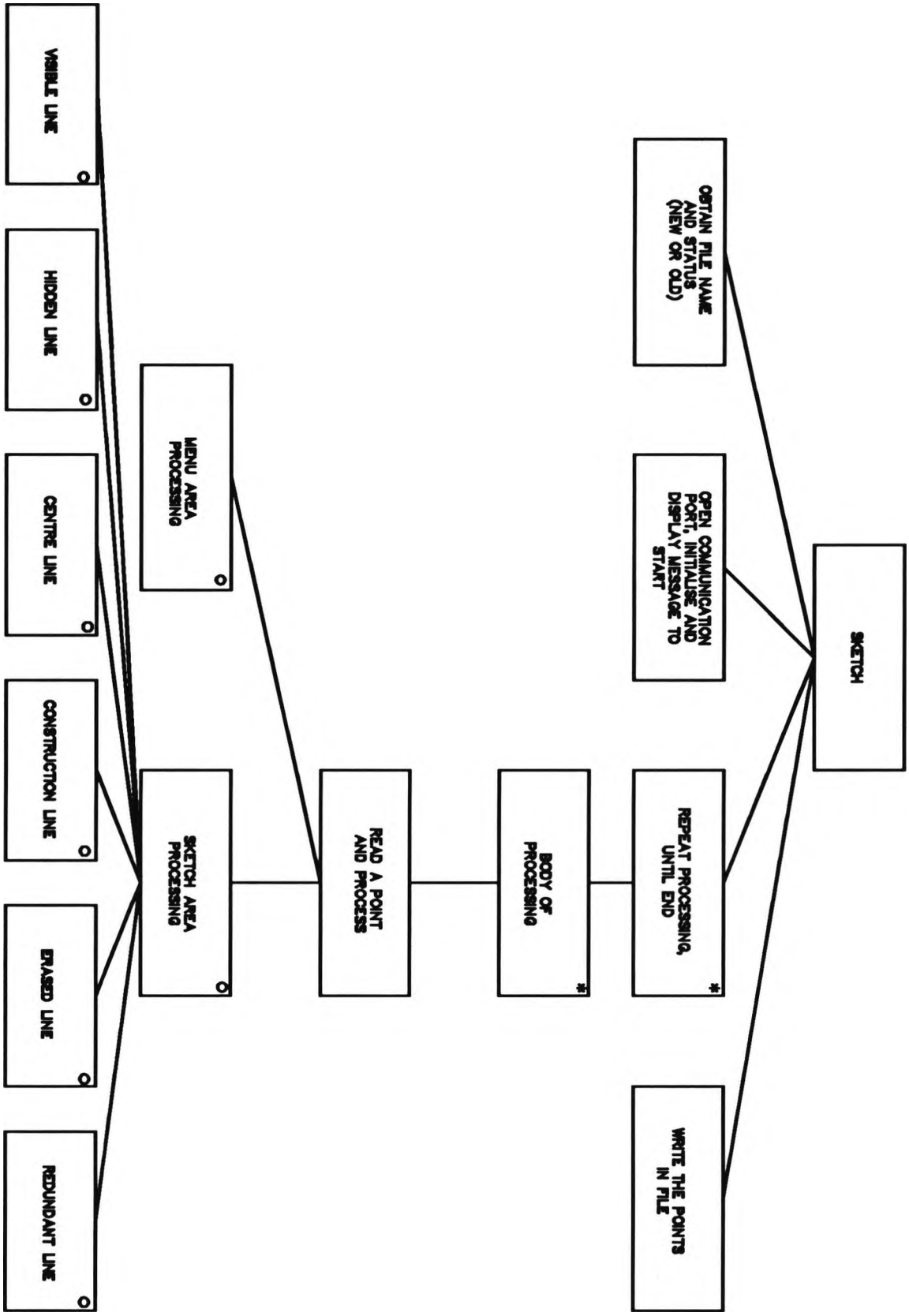

FIGURE 5.2

STRUCTURE OF SKETCH

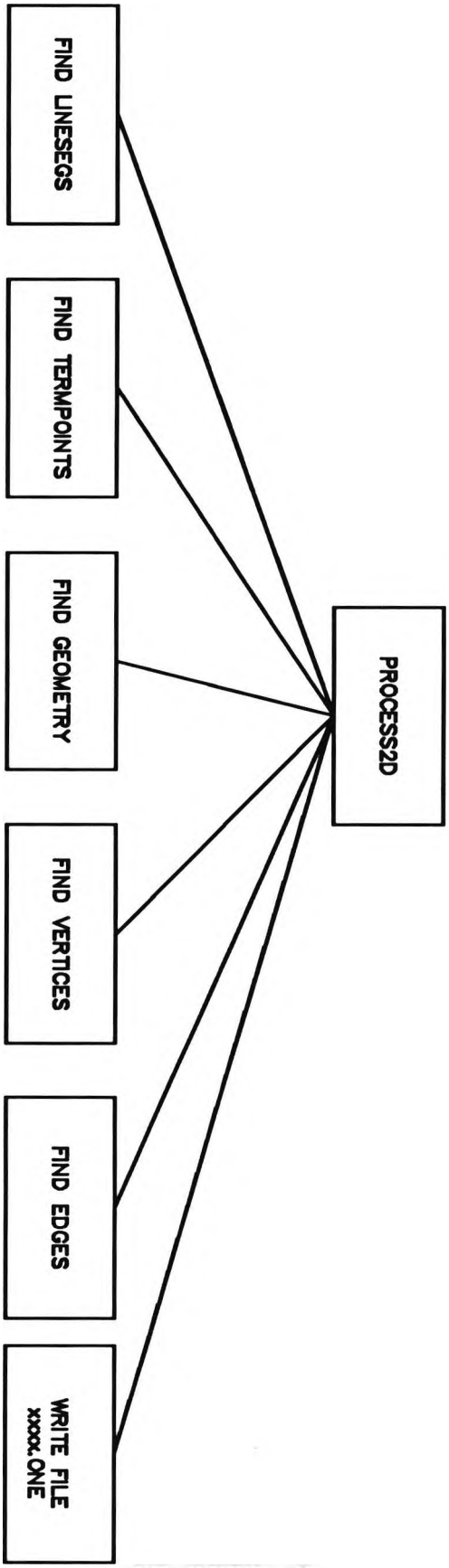

FIGURE 5.3

STRUCTURE DIAGRAM OF PROCESSING IN TWO DIMENSIONS

less than 150 untis or 1.5 cm in length or no circle or ellipse with diameter less than 1.5 cm. It forms a linked list of 'Lineseg' nodes (see figure 4.2) and functions 'get_lineseg' to allocate memory for a 'Lineseg' node and 'back_of_lineseg' to put a new node at the end of a list, are written to support this algorithm. The steps in the algorithm are as follows:

(a) Set the processed points called 'parsed_points' to be = 0

(b) Set the variable 'finished' to be 0

(c) Read the file of points

(d) Repeat the following steps as long as 'finished' is 0

(i) Take the number of points in the first 150 units starting from the point 'parsed_points'

(ii) Connect the first and last point in the group found in (i) above and find whether all points in the group fall to one or both side of the line

(iii) If they fall to both side then it is a straight line

(iv) If it falls to one side do confirmatory test to ensure that it is a curve

(v) After ensuring that the line is either a curve or straight line find the end point of the curve or straight line

(vi) If the end point is equal to the number of points in the array then set finished = 1

(vii) If finished is not equal to 1 and if the distance from the end point and the next point is greater than 150 units, the minimum length of a line, then the next line starts from the next point (two non continuous lines) and CONTINUITY is set at 0. If not, they are two continuous lines and CONTINUITY is set at 1 with the starting point of the next line being the end point of the last line

(viii) Set the 'parsed_points' to be equal to the endpoint or endpoint + 1 according to the value of continuity

(ix) Allocate memory to a new Lineseg node and store the values of the fields

(x) Put the new node at the back of the list

(e) The algorithm at the end returns a linked list of Lineseg nodes each node representing a line segment. Figure 5.4 shows its structure diagram.

5.2.2 FINDING THE TERMINAL POINTS

The points representing any line in the sketch are only selected ones and do not form an exhaustive list of points in the line. This means that the end point of a represented line need not to be a vertex in the sketched solid. It could instead be a point close to it. Such a point close to the vertex and representing an end point of a line is called a terminal point in this thesis. A terminal point is characterised by the following:

(i) It lies in the close vicinity of a vertex

(ii) It lies in the close vicinity of the end points of lines emanating from that vertex

(iii) There is only one vertex for which the terminal point is very close and

(iv) Every terminal point could be approximated to one and only one vertex and there can only be the same number of vertices and terminal points in a sketch

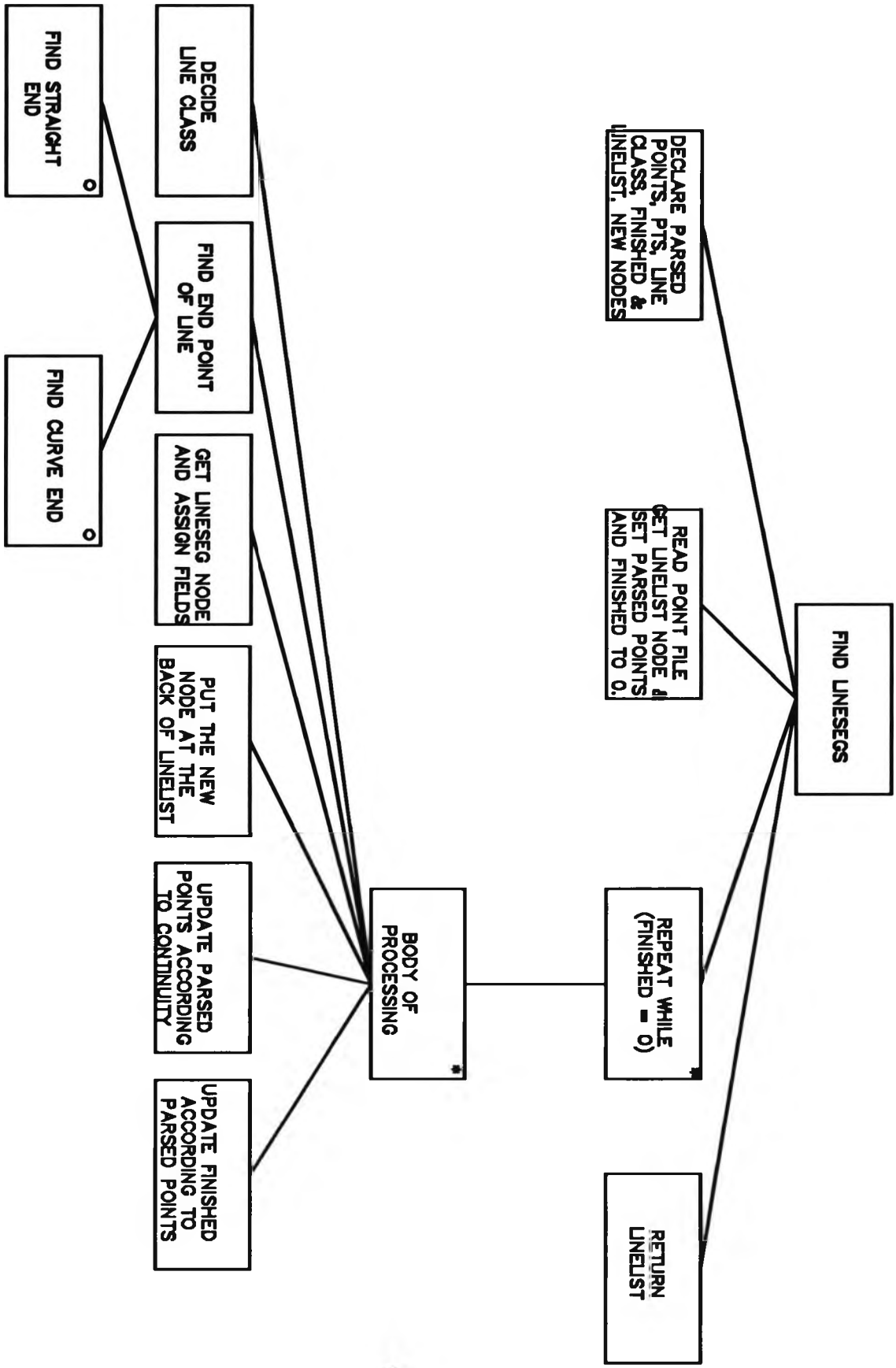

FIGURE 5.4

EXTRACTING THE LINE SEGMENTS

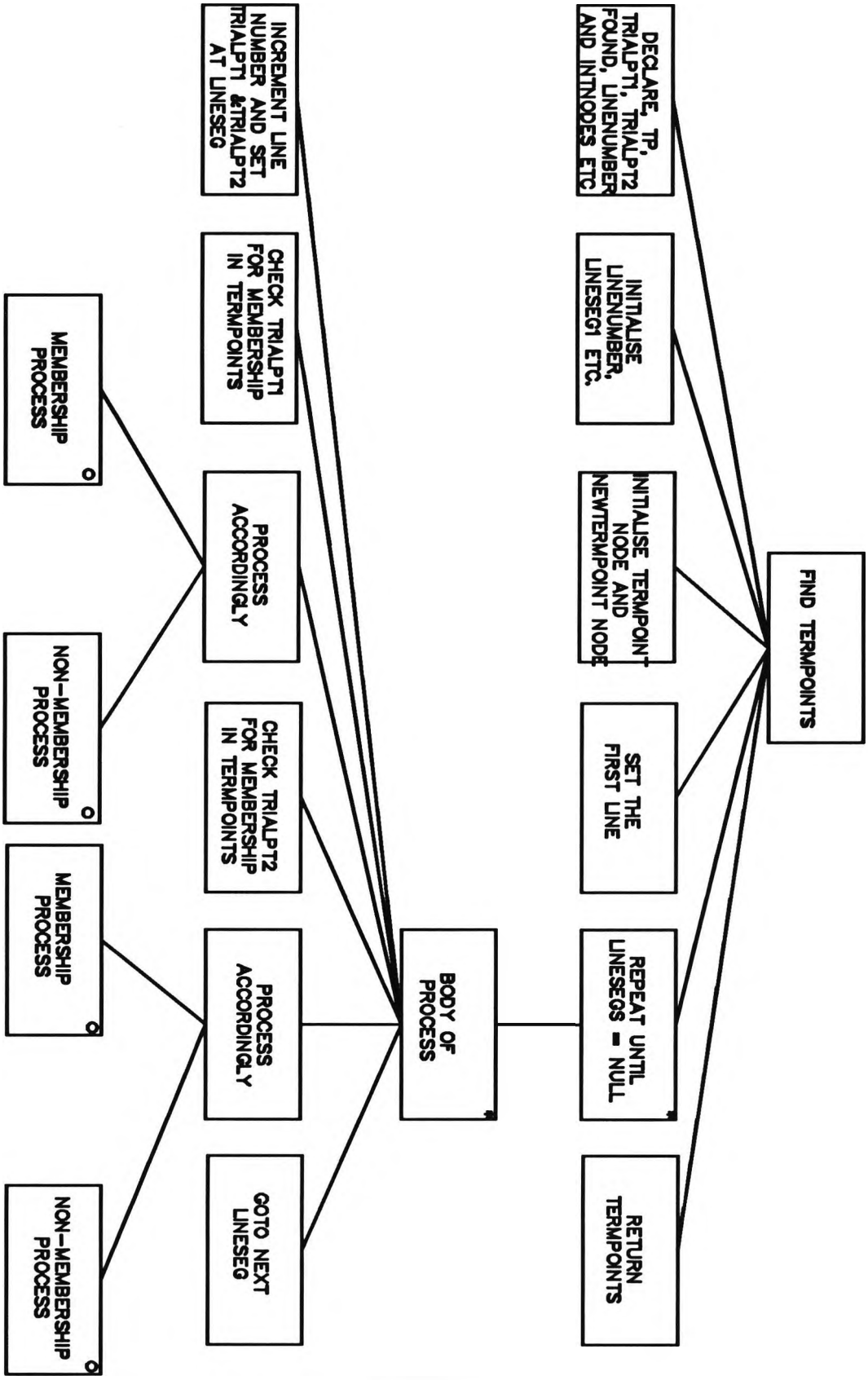

FIGURE 5.5

FINDING TERMINAL POINTS

This algorithm finds all terminal points in the given linked list of 'Lineseg' nodes. It forms a linked list of 'Termpoint' nodes (see figure 4.2) and functions 'get_termpoint' and 'back_of_termpoint' are written in support of this algorithm.The steps in the algorithm are as follows:

(a) Set a list of 'Termpoint' node
(b) Take the first line segment from the list and its starting point and finishing point. Get a new 'Termpoint' node and fill the fields with details of the starting point. Put this node at the back of the list created in (a) above. Get a new 'Termpoint' node and fill the fields with details of the finishing point. Put this node at the back of the list created in (a) above.
(c) Repeat the following to all nodes in the linked list of 'Lineseg' nodes.
(i) Get the correct line number.
(ii) Set starting and finishing points of the 'Lineseg' node as trial point 1 and trial point 2.
(iii) Check whether trial point 1 is already represented in the list of terminal points.
(iv) If it is already represented add the line number to the linelist.
(v) If it is not represented create a new 'Termpoint' node and fill the fields. Put the new node at the back of the list of 'Termpoint' nodes created at (a) above.
(vi) Repeat steps (iii), (iv) and (v) for trial point 2.
(d) The algorithm at the end returns a linked list of 'Termpoint' nodes. Figure 5.5 shows the structure diagram of this algorithm.

5.2.3 FINDING THE ANALYTICAL EQUATION OF LINES

This algorithm fits analytical equations to the line segments given by a linked list of 'Lineseg' nodes and the array of points which is referenced by each of these nodes. For each of the node in the list of 'Lineseg' nodes an equation has to be fitted. The steps in the algorithm are as follows:

(a) Set a list of 'Geom_edge2d' node
(b) Repeat the following for each node in the list of 'Lineseg' nodes
(i) Depending on the linetype in the Lineseg node invoke the straight line fitting or ellipse fitting function.
(ii) If it is a straight line then depending on the slope whether it is a vertical line or not, invoke the appropriate fitting algorithm.
(iii) Get a new 'Geom-Edge' node and assign the newly fitted vector to the node. Put this new node at the back of the list created in (a) above.
(c) The end product at the end of this algorithm is a list of 'Geom_edge2d' nodes. Figure 5.6 shows the structure diagram of this algorithm.

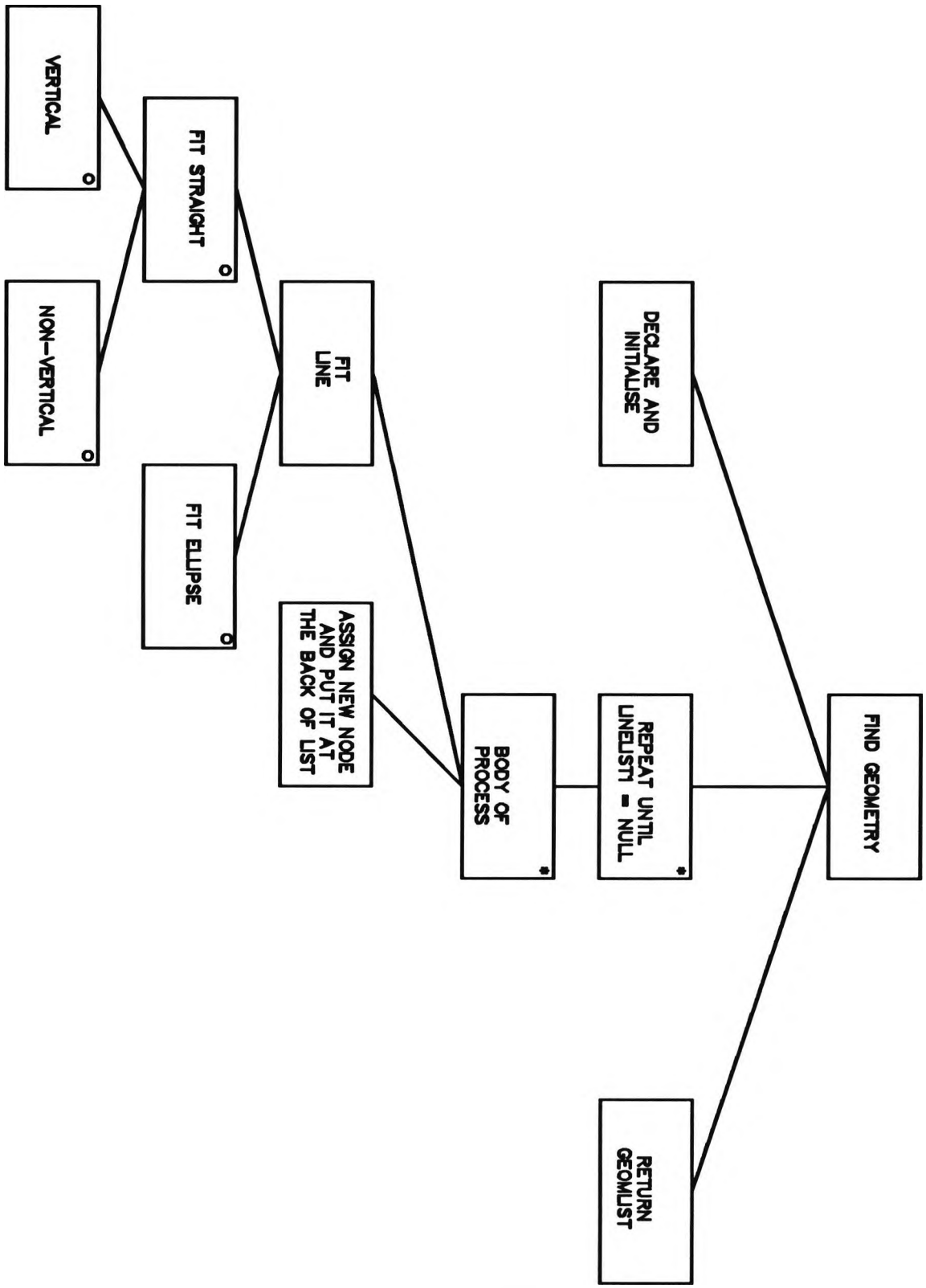

FIGURE 5.6

FINDING THE ANALYTIC EQUATIONS OF LINES

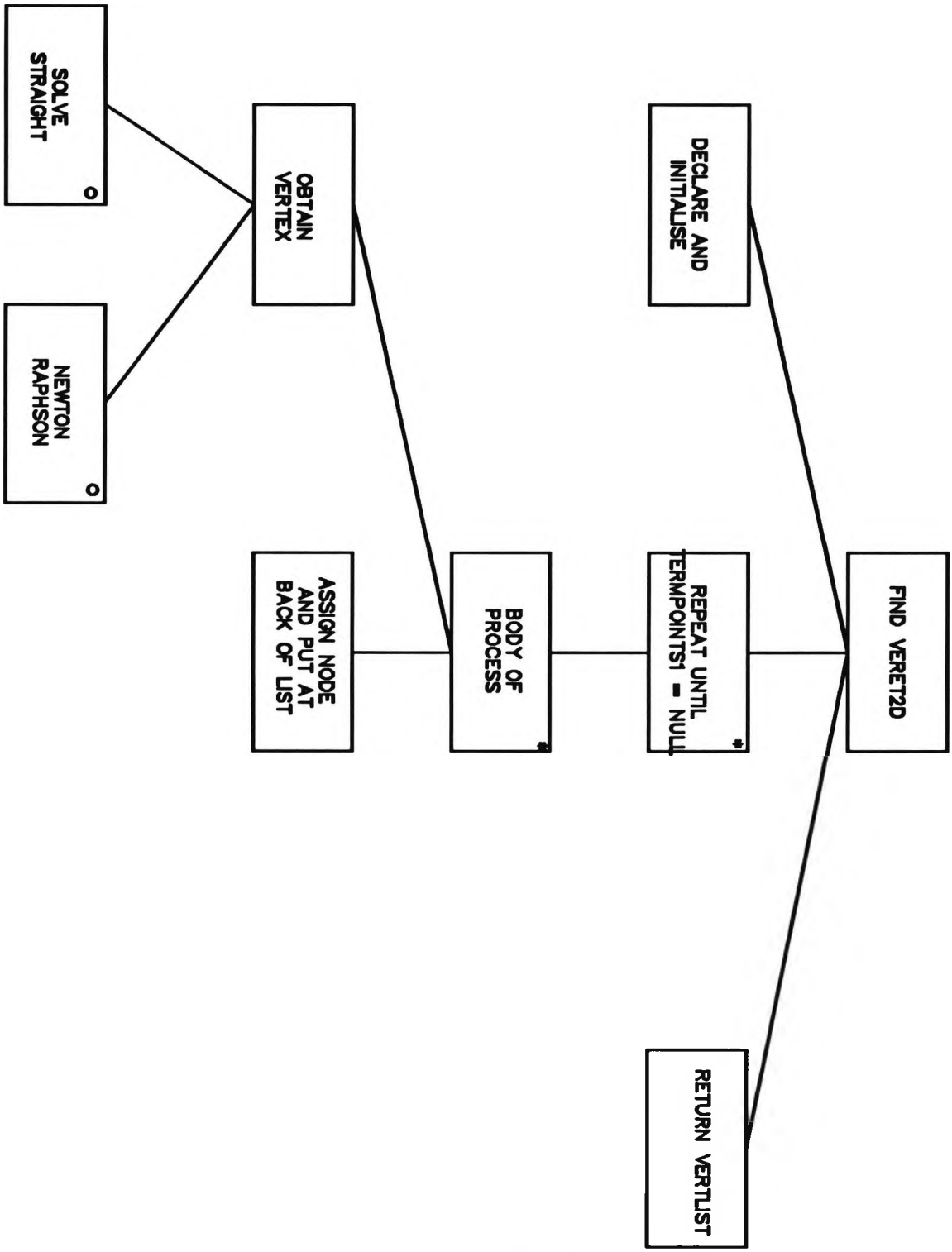

FIGURE 5.7

FINDING VERTICES

5.2.4 FINDING VERTICES

This algorithm finds out a solution to all the equations representing the lines emanating from a vertex or its corresponding terminal point. It is generally an overdetermined system of equations and the lines involved may be all straight lines or a mixture of straight lines and ellipses. The theory of finding solutions is discussed in detail in sections 3.5 and 3.6. The algorithm is consisted of the following steps:

(a) Set a list of 'Vert2d' nodes.

(b) Repeat the following steps for each node in the list of 'Termpoint' nodes.

(i) Check whether there are any curves in the linelist of the 'Termpoint' node.

(ii) If there are no curves solve the set of equations according to the least squares algorithm described in section 3.5.

(iii) If there is at least one curve use the Newton-Raphson algorithm described in section 3.6.

(iv) Get a new 'Vert2d' node and assign the fields with the co- ordinates found in (ii) or (iii) and the corresponding linelist. Put the new node at the back of the list in (a) above.

(e) The algorithm will yield a linked list of 'Vert2d' nodes. The structure diagram of the algorithm is given in figure 5.7.

5.2.5 FINDING THE EDGES

This algorithm finds the starting vertex number and the finishing vertex number and the line type in accordance to the 12 classes described in section 3.12. This has a one to one relationship with the nodes in the linked lists of 'Lineseg' and 'Geom_edge2d' nodes. The algorithm has the following steps:

(a) Set a list of 'Vert2d' nodes.

(b) Repeat the following steps with each node in the linked list of 'Lineseg' nodes.

(i) Assign the end points as trial point 1 and trial point 2.

(ii) Get the vertex 1 to correspond to trial point 1.

(iii) Get a 'Vert2d' node and assign vertex 1.

(iv) Get the vertex 2 corresponding to trial point 2 and assign vertex 2.

(v) Decide the linetype and assign the linetype.

(c) This algorithm will return a linked list of 'Edge2d' nodes. Figure 5.8 shows its structure diagram.

5.3 THE MERGING PROCESS

This is the process which allows for the merging of lines started from both ends to meet at an intermediate point and overstruck lines, the merging of visible and hidden lines, removal of the erased lines and the formation of the lines in the solid. A function called 'merge2d' is written to perform all these functions. It has three major constituent algorithms.

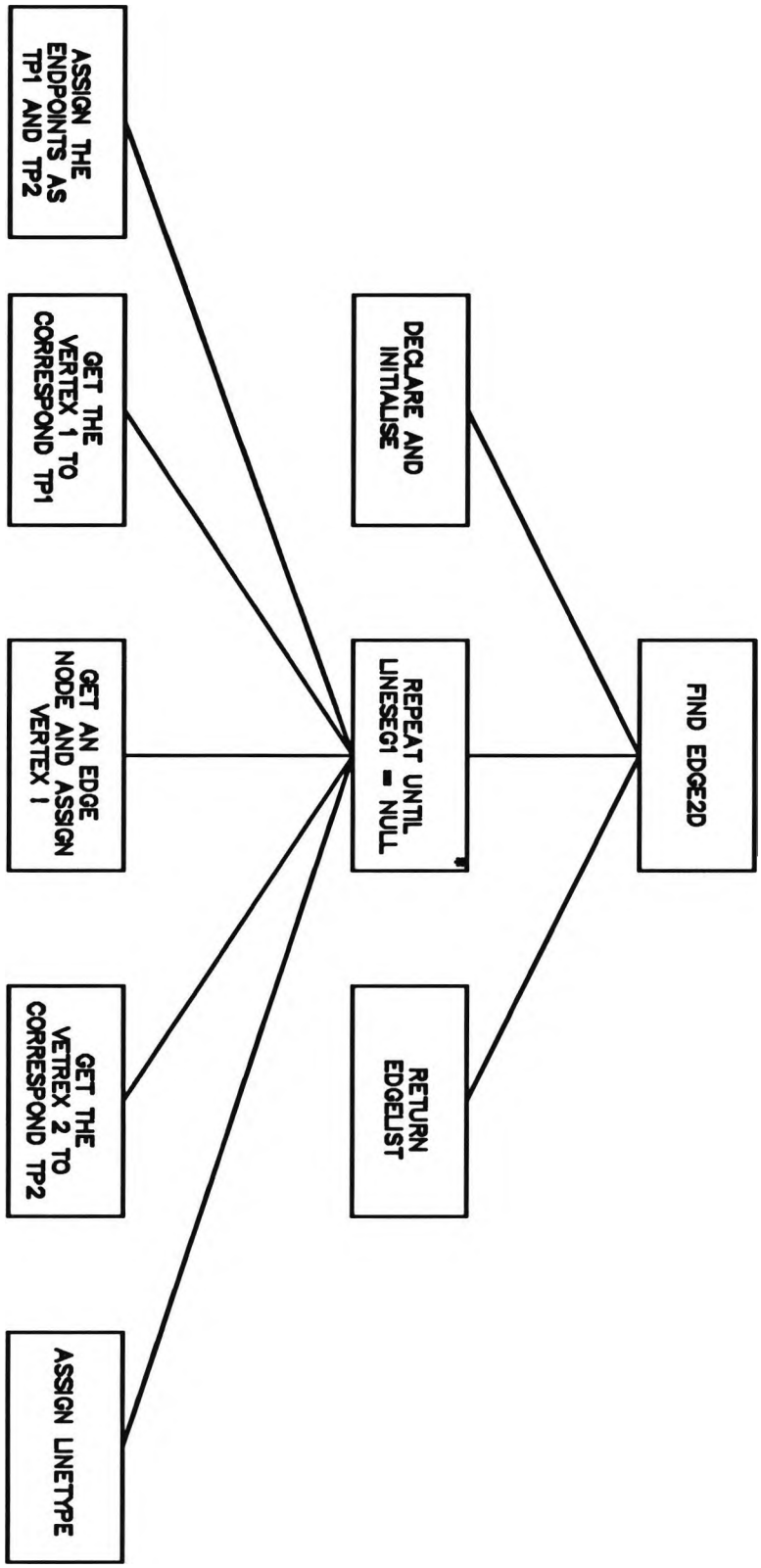

FIGURE 5.8

FINDING THE EDGES

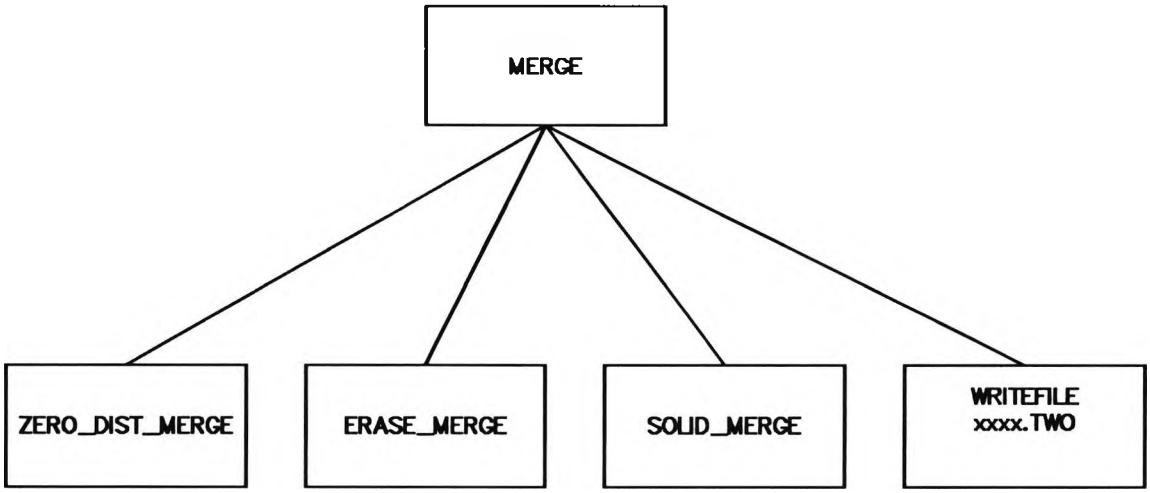

FIGURE 5.9

STRUCTURE OF MERGE FUNCTION

They are
(a) Zero distance merging
(b) Erase merging and
(c) Solid merging
The structure diagram of this function is shown in figure 5.9.

5.3.1 ZERO DISTANCE MERGING

In a sketching situation the user sometimes starts the drawing of a line from both ends to meet at an intermediate point. In these circumstances the computer in this program will identify them as two lines, each starting from the end point and finishing at the intermediate point. To rectify this shortcoming 'zero distance merging' algorithm was developed. The steps in this algorithm are as follows:

(a) Set the second list of 'Geom_edge2d' node which starts from the next node of the first list of 'Geom_edge2d' nodes.
(b) For all nodes in the first list of 'Geom_edge2d' nodes repeat the following:
(i) Check whether the equation of the first list is the same with the equation of the second node and if same return its position (rank) in the list.
(ii) Inform the user that two edges are conformable for merging and obtain his consent for merging.
(iii) Obtain the edge to be removed, the vertices to be removed and the type of merge. The types of merging are (1) Removal of overstruck straight line (2) Accommodation of extended line (3) Removal of overstruck curve and (4) Accommodation of ex-

tended curve.

(iv) Perform the merging according to the type.

(v) Update the second list and the first list.

(c) The algorithm will result in a new set of linked lists of three out of the five nodes, representing the solid.

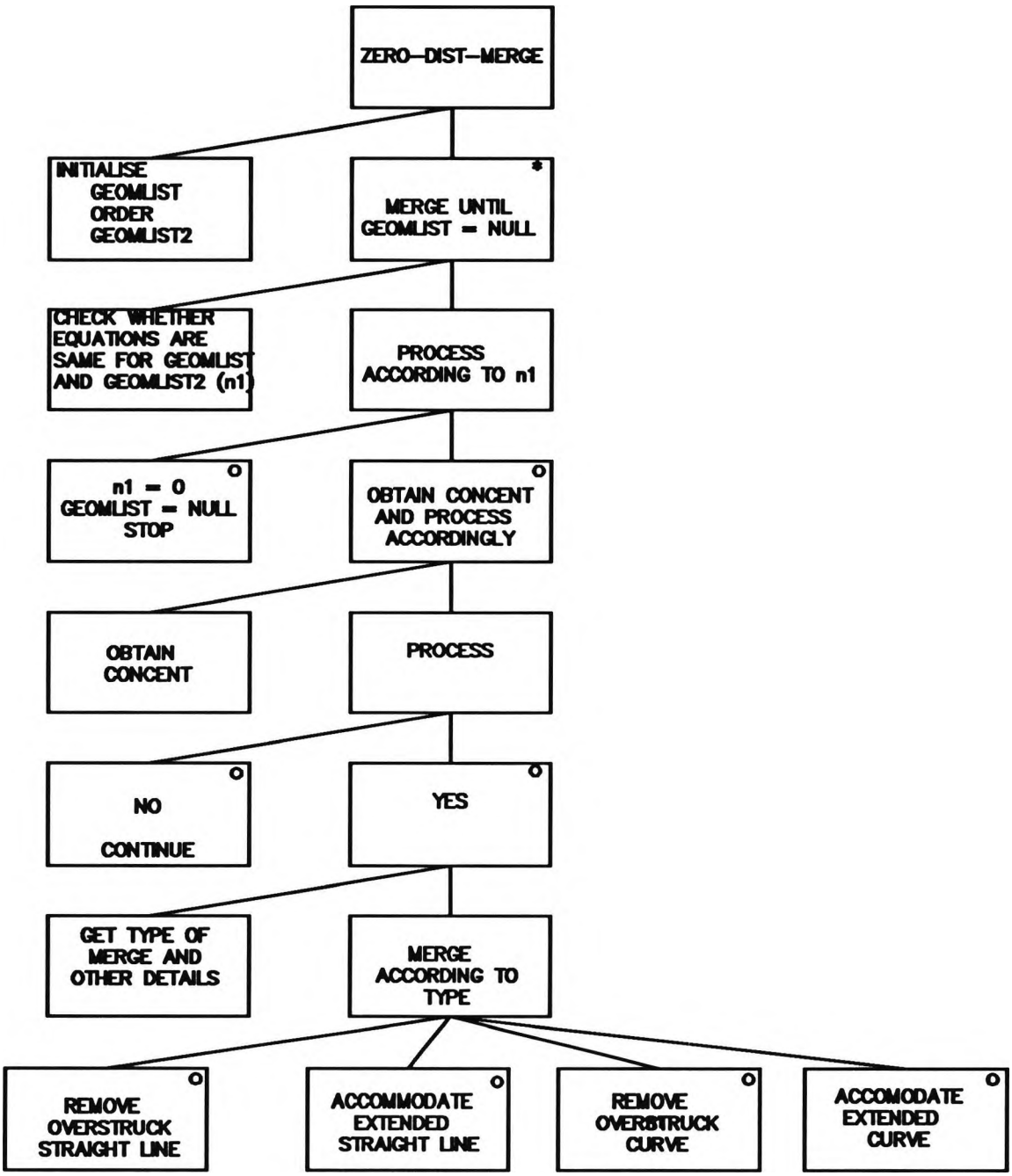

FIGURE 5.10

ZERO DISTANCE MERGING

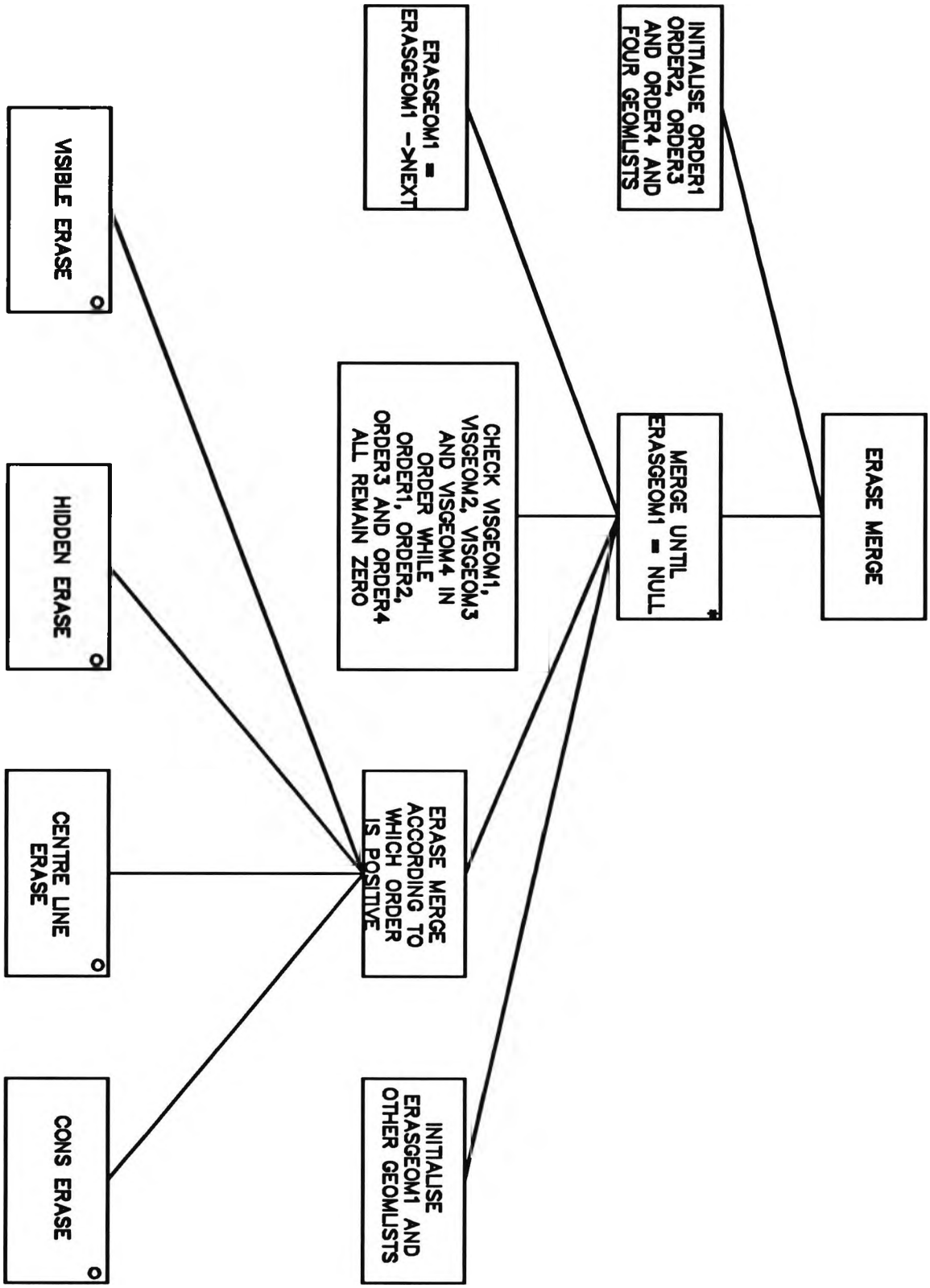

FIGURE 5.11

ERASE MERGING

In the process of implementing this algorithm functions to kill the five types of nodes are developed. They are 'kill_lineseg', 'kill_geomedge', 'kill_termpoint', 'kill_vert2d' and 'kill_edge2d'. Figure 5.10 illustrates the structure of the algorithm.

5.3.2 ERASE MERGING

This algorithm is developed to find the corresponding edge for each edge in the erased list, in any one of the other four lists representing the visible lines, hidden lines, centre lines and construction lines. The algorithm then removes the identified edge. The process is repeated until the entire list of erased edges is exhausted. The steps in the algorithm are as follows:

(a) Set four lists of 'Geom_edge2d' nodes representing those of the visible, hidden, centre and construction lines. Also set four integers which represent the position (rank) of nodes in each of these four lists. Set these integers to take the initial value of zero. This process in step (a) is called initialisation in the following step (b).

(b) For each node in the list of 'Geom_edge2d' nodes representing the erased lines do the following:

(i) Check the lists representing the visible, hidden, centre and construction lines in order, to find an edge with the same equation and is overlapping with the erased edge under consideration.
(ii) Set the position of the overlapping node in the variable set in initialisation for the particular class of line.
(iii) Depending on the value of the variable set in (ii) and on which variable is set perform the erasure.
(iv) Set the erased list and re-initialise.
(c) The algorithm will produce the final list of edges and vertices in the visible, hidden and construction classes which are in the sketched solid. Figure 5.11 shows the structure diagram of this algorithm.

5.3.3 SOLID MERGING

This algorithm is developed to establish the final lists of edges and vertices present in the sketched solid. It is achieved by combining the hidden edges and visible edges together. The steps in the algorithm are as follows:

(a) Set a list of 'Vert2d' nodes to represent the edges in the solid and copy all visible edges onto it.
(b) Set a list of 'Edge2d' nodes to represent the edges in the solid and copy all visible vertices onto it.
(c) Get the number of edges and number of vertices.
(d) For each edge in the list of 'Edge2d' nodes representing the hidden lines repeat the following:

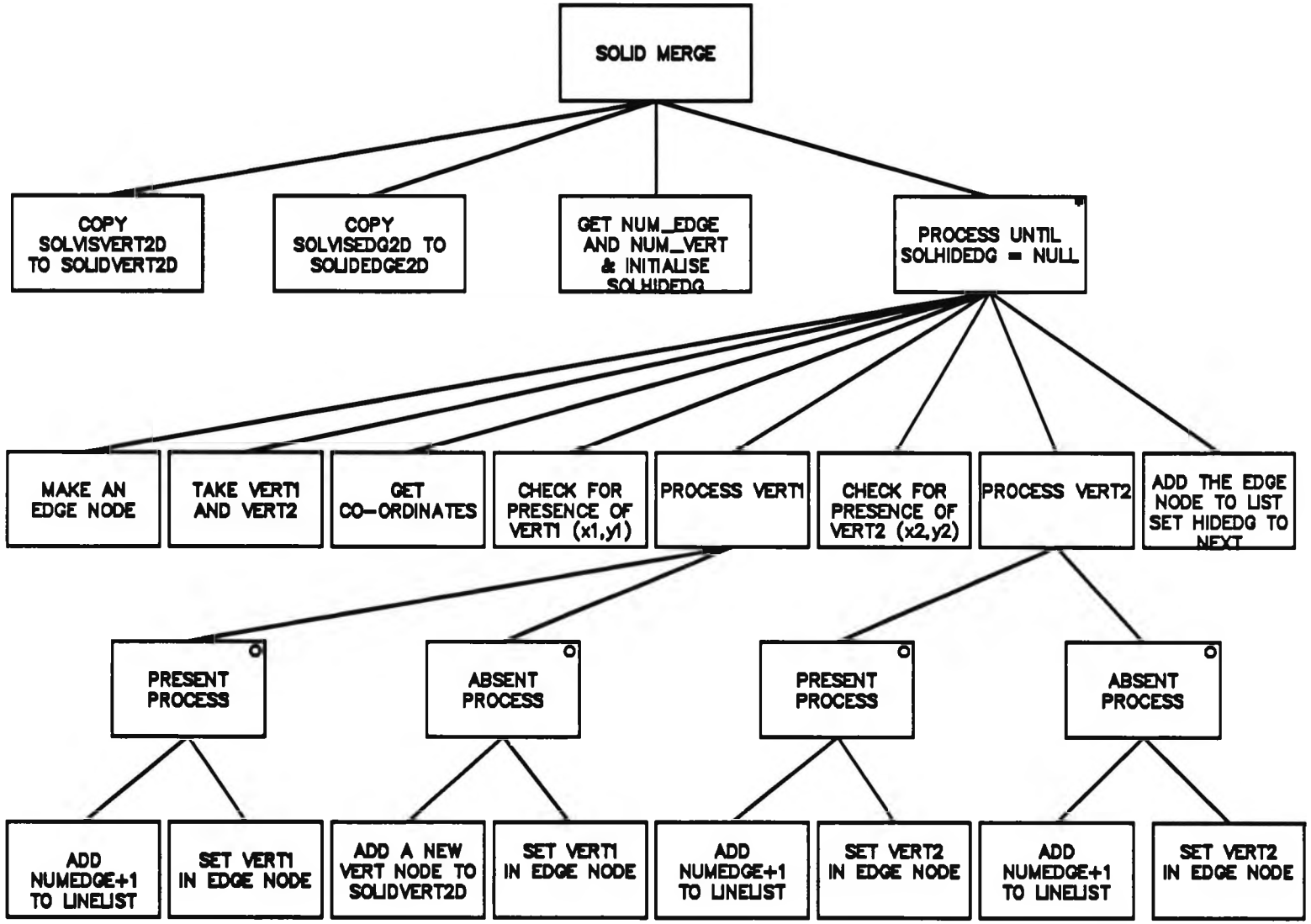

FIGURE 5.12

STRUCTURE OF SOLID MERGE

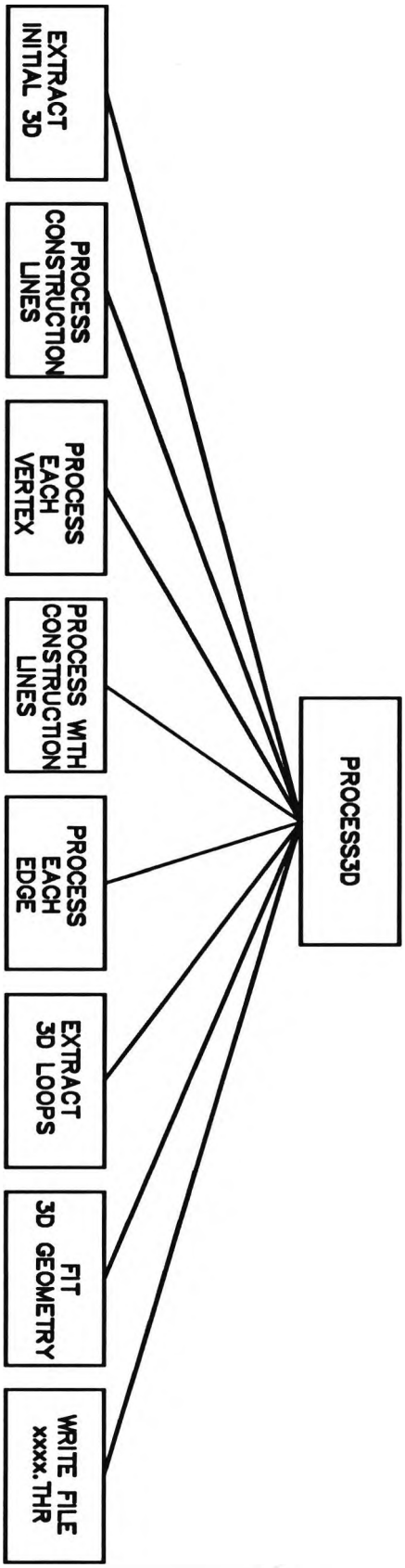

FIGURE 5.13

PROCESSING IN THREE DIMENSIONS

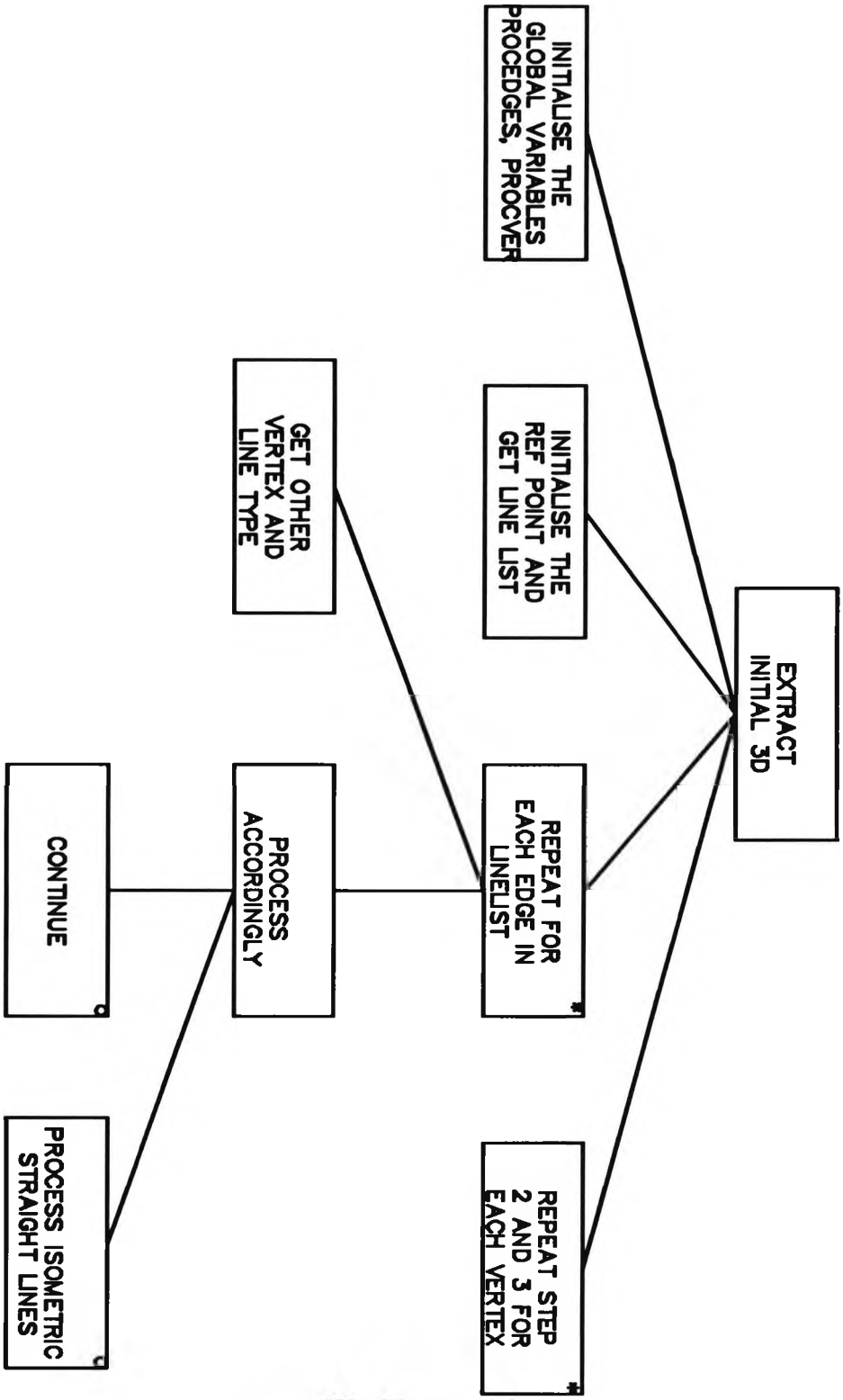

FIGURE 5.14

EXTRACTING THE INITIAL THREE DIMENSIONAL DETAILS

(i) Get an 'Edge2d' node.
(ii) Get the vertex 1 and vertex 2 from the 'Edge2d' node from the hidden line list.
(iii) Get the co-ordinates of vertex 1 and vertex 2.
(iv) Check whether vertex 1 is present in the vertex list of the solid.
(v) If present add the integer (number of edges + 1) to the line list of that vertex.
(vi) Set that vertex number in the edge node obtained in (a) above.
(vii) If absent add a new vertex node to the list of vertices in the solid.
(viii) Set the vertex number in the edge node obtained in (a) above.
(ix) Repeat steps (iv) to (viii) for vertex 2.
(e) Perform a zero distance merge to merge the edges which are partially hidden and partially visible.
(f) The algorithm produces the list of edges and vertices in the solid sketched ready for three dimensional transformation. Figure 5.12 shows the structure diagram of this program.

5.4 PROCESSING IN THREE DIMENSIONS

The processing in three dimensions involves the generation of the five linked lists namely (i) list of Edge3d nodes (ii) list of Vert3d nodes (iii) list of Loop3d nodes (iv) list of Face3d nodes and (v) list of Ring3d nodes. These nodes are shown in figure 4.3. These lists together, give all the details necessary to build the solid model. The details obtained here are as follows:
(i) Co-ordinates of all the vertices in three dimensions
(ii) Co-ordinates of the centres of circular elliptical holes in three dimensions
(iii) Details of edges
(iv) Details of loops
(v) Details of faces
(vi) Details of rings

A function called 'process3d' is written to implement this process. Figure 5.13 shows the structure diagram of this function. In the first stage it converts the isometric lines and curves to three dimensions. Then it converts the construction lines to three dimensions. It now runs through the vertex list in two dimensions and extracts the unprocessed vertices. These unprocessed vertices are then processed using the construction lines. At the end of this, the co-ordinates of all vertices in three dimensions would be known. The function now runs through the lists of edges to ensure that all edges in the solid drawn are processed. Once everything is transformed to three dimensions, the function extracts all clockwise loops in the solid. In order to do this it obtains the first clockwise loop from the user for each connected set of edges. It uses the fact that each edge appears in exactly two loops once in the positive and once in the negative direction (see section 2.6.3). Once the loops are obtained and the tangency edges and silhouette edges are established, the surface equations of the

planar faces could be fitted and the rings could be established.

At this stage all elements in the Euler-Poincare formula for the solid sketched would be known and the solid could be checked for validity. Any addition or deletion could then be made to make the solid valid. Once the solid's validity is assured then the Euler co-ordinates could be calculated by using the Euler matrix (see section 2.3.2). The solid model could then be built with the appropriate Euler operators.

5.4.1 EXTRACTING THE INITIAL THREE DIMENSIONAL DETAILS

In this process the reference point (normally the three dimensional origin) and the isometric lines are used to extract the three dimensional co-ordinates of all the possible vertices. It processes the line types 1, 2, 3 (see section 3.12). The algorithm has the following steps.

(i) Create a 'Vert3d' list and 'Edge3d' list with the same number of vertices and edges in the sketch.

(ii) Allocate memory for two vectors 'processed edges' and 'processed vertices' to accommodate integer elements which are the vertex numbers and edge numbers.

(iii) The number of elements in each vector elements in each vector is the number of edges and number of vertices respectively.

(iv) Set the number of processed edges PROCEDGE and processed vertices PROC-VERT as zero.

(v) Initialise the reference point and obtain the number of lines emanating from the reference point.

(vi) Repeat the following for each edge emanating from the reference vertex.

(a) Get the other vertex and line type.

(b) Process for the straight line, if the line type is 1, 2 or 3.

(c) Go to the next edge emanating from the vertex.

(d) Include the vertex processed in the 'processed vertex' vector and increment the count PROCVERT.

(e) Include the edge processed in the 'processed edge' vector and increment the count

(vii) Now go to the second element in the vector representing the processed vertices and repeat steps (i) to (vii).

The end result of this initial extraction are a partially filled 'Vert3d' and 'Edge3d' lists and the two vectors representing their numbers. If the number of processed vertices is equal to the number of vertices in the edge then the transformation is complete. The structure diagram of this function is given in figure 5.15.

5.4.2 PROCESS CONSTRUCTION LINES

There are two occasions in which construction lines are drawn in an isometric sketch. In the first instance it is drawn to locate a point with respect to an isometric point.

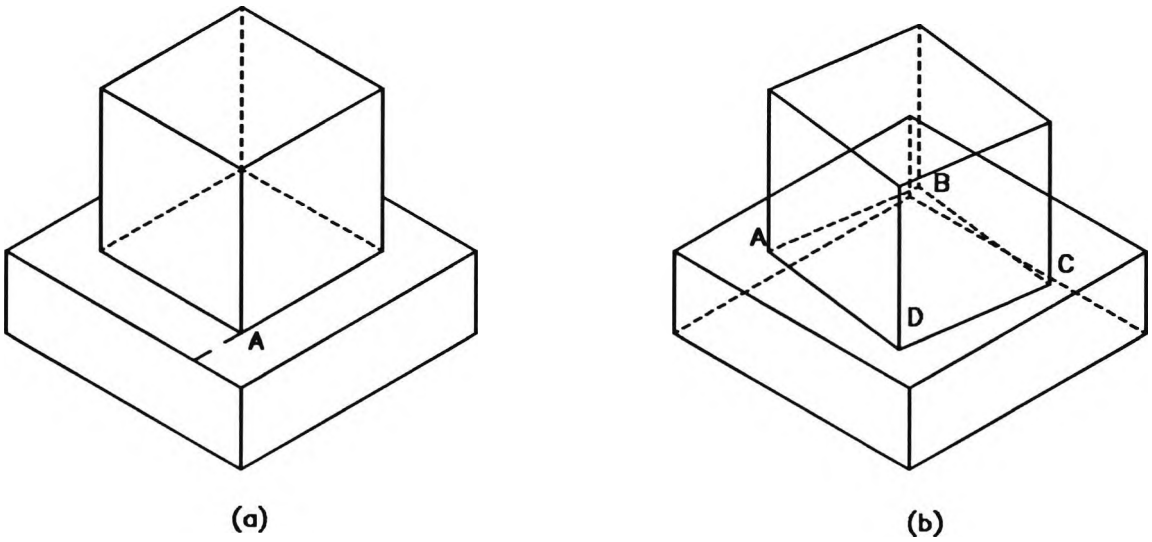

FIGURE 5.15

ILLUSTRATION OF CONSTRUCTION LINES OF TYPE 1

This is illustrated in figure 5.15. In (a) the point A is fixed by a construction line and this is adequate since all the lines involved are isometric lines. The situation is different in (b) where all the points A, B, C and D need a construction line since all the lines involved are non-isometric. The construction lines involved in this case are all isometric straight lines. This is due to the fact that only isometric lines retain the true length.

In the second occassion construction lines are drawn to 'box' the ellipses and arcs as illustrated in figure 5.16. Here again the construction lines are all isometric straight lines. Thus processing of construction lines in these cases is essentially the processing of lines of types 1, 2, and 3. But when non-isometric ellipses are boxed there can be non-isometric construction lines. In any case processing construction lines involves with the extraction of the three dimensional co-ordinates of the ends of the construction lines. When dealing with isometric construction lines if one end point is known the other end point could be calculated while non-isometric lines would need extra construction lines for each end.

The algorithm has the following steps:

(i) Create the two lists CONSEDGE3D and CONSVERT3D of nodes Edge3d and Vert3d.

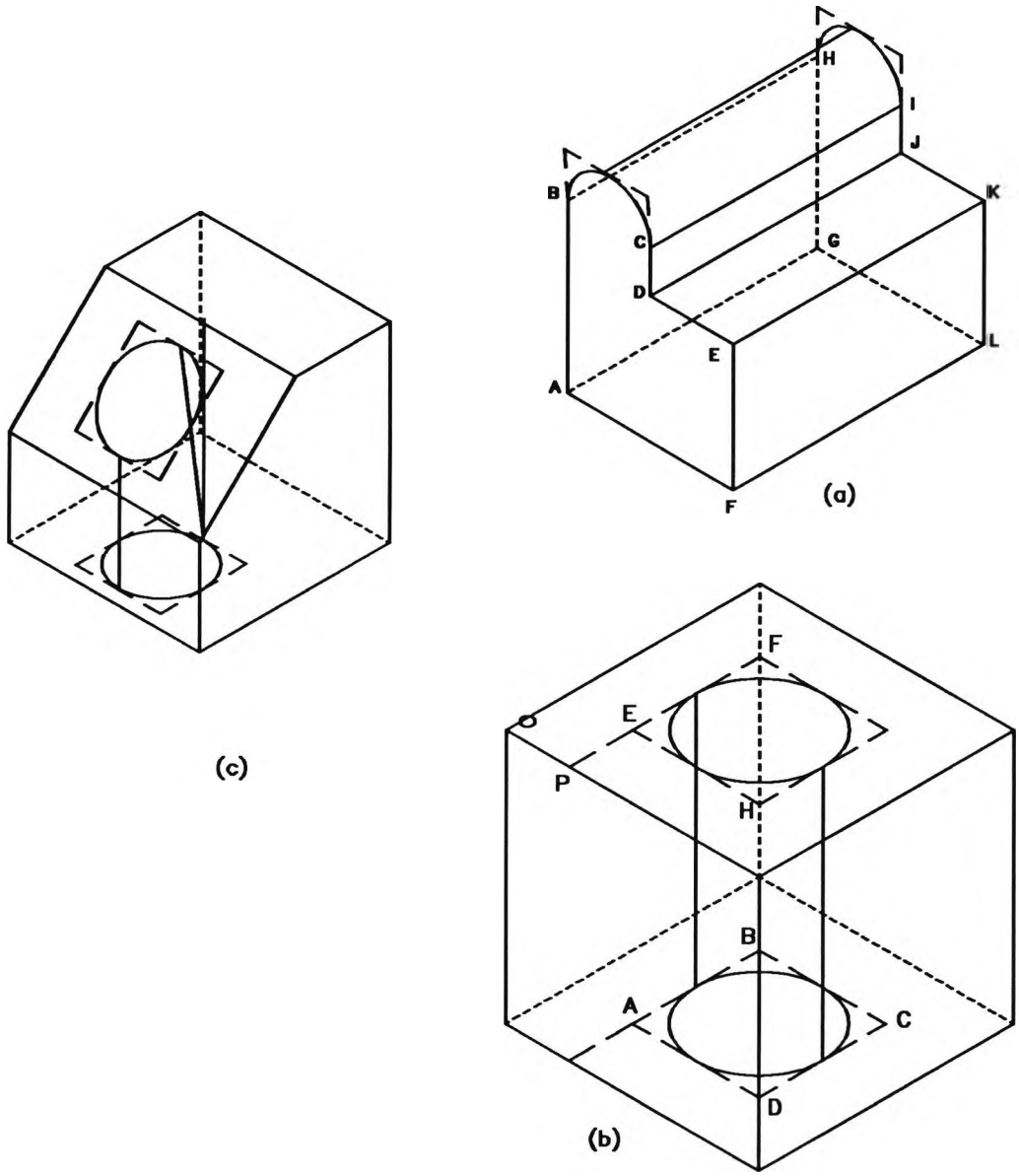

FIGURE 5.16

ILLUSTRATION OF CONSTRUCTION LINES OF TYPE 2

(ii) Initialise the lists consedge2d1, consvert2d1 and consgeom1 representing the edge, vertex and edge geometry lists with pointers pointing to the first line.
(iii) Repeat the following until consedge1 = NULL.
(a) if the linetype is 4 go to the next line through step (i)
(b) If the line type is 1, 2 or 3 do the following
(c) Check whether the 3D co-ordinates of one end is known. If known process the other point.
(d) If both ends are not known check which end is very close to an isometric visible line

(e) Obtain the intersecting 2D point (normally one end of the construction line) and convert it to 3D.
(f) Use this point to obtain the other point.
(g) If (e) and (f) are not possible continue.
(h) Store the values and go to the next edge
(iv) At the end of the processing, the end vertices of all the construction lines, whether isometric or not, will be available. Use them to fill the unfilled node fields in the lists CONSEDGE3D AND CONSVERT3D.

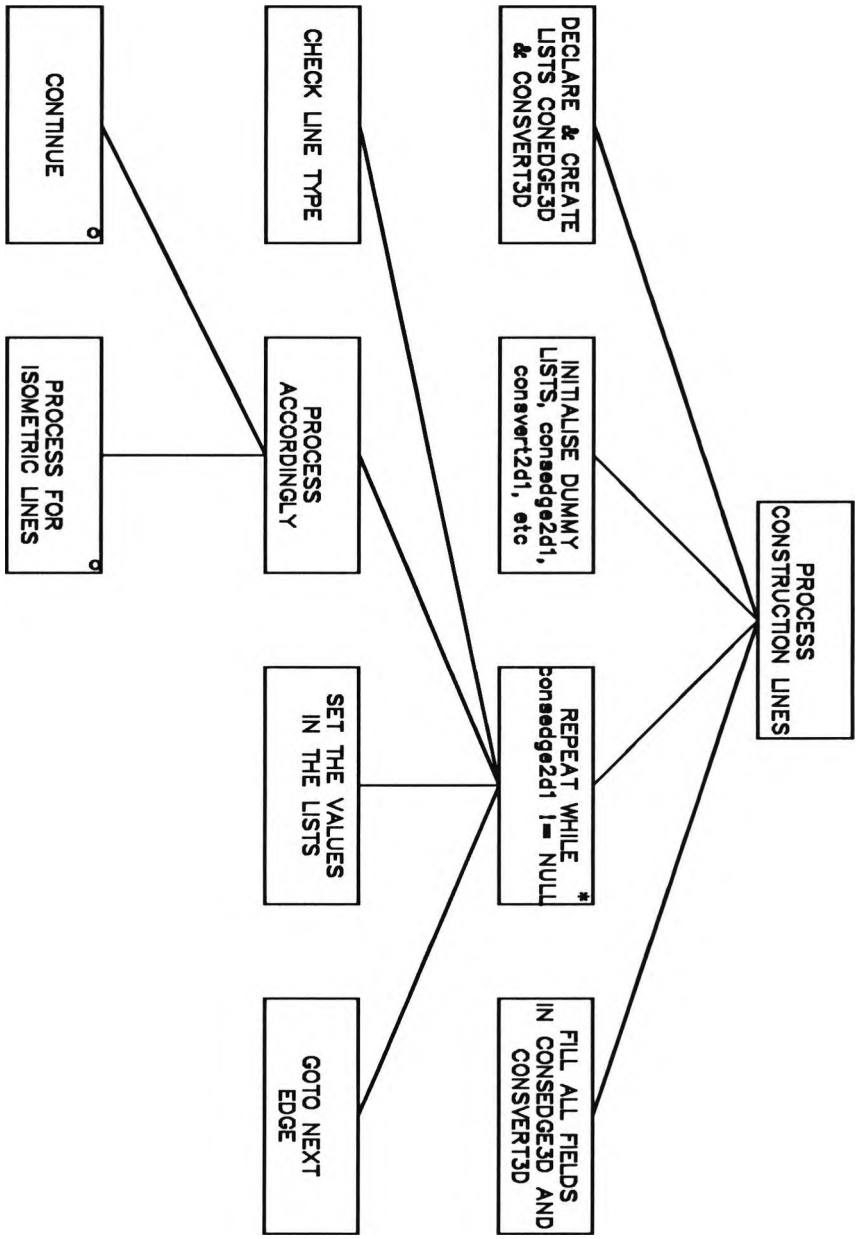

FIGURE 5.17

STRUCTURE OF PROCESSING CONSTRUCTION LINES

The end product of this algorithm is the two global lists CONSEDGE3D and CONSVERT3D containing the three dimensional details of the construction lines. Figure 5.17 shows the structure of the program.

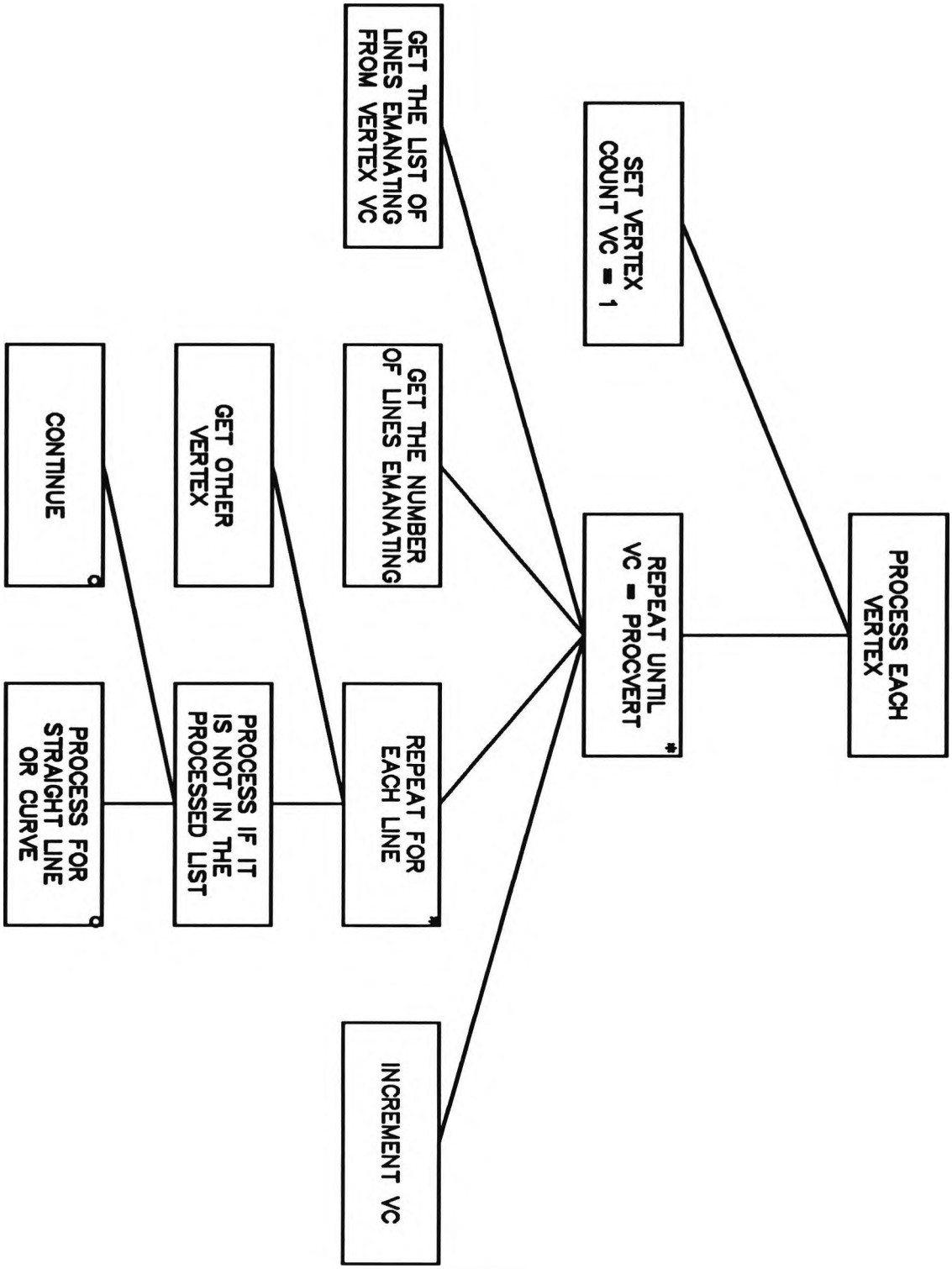

FIGURE 5.18

STRUCTURE OF PROCESSING EACH VERTEX

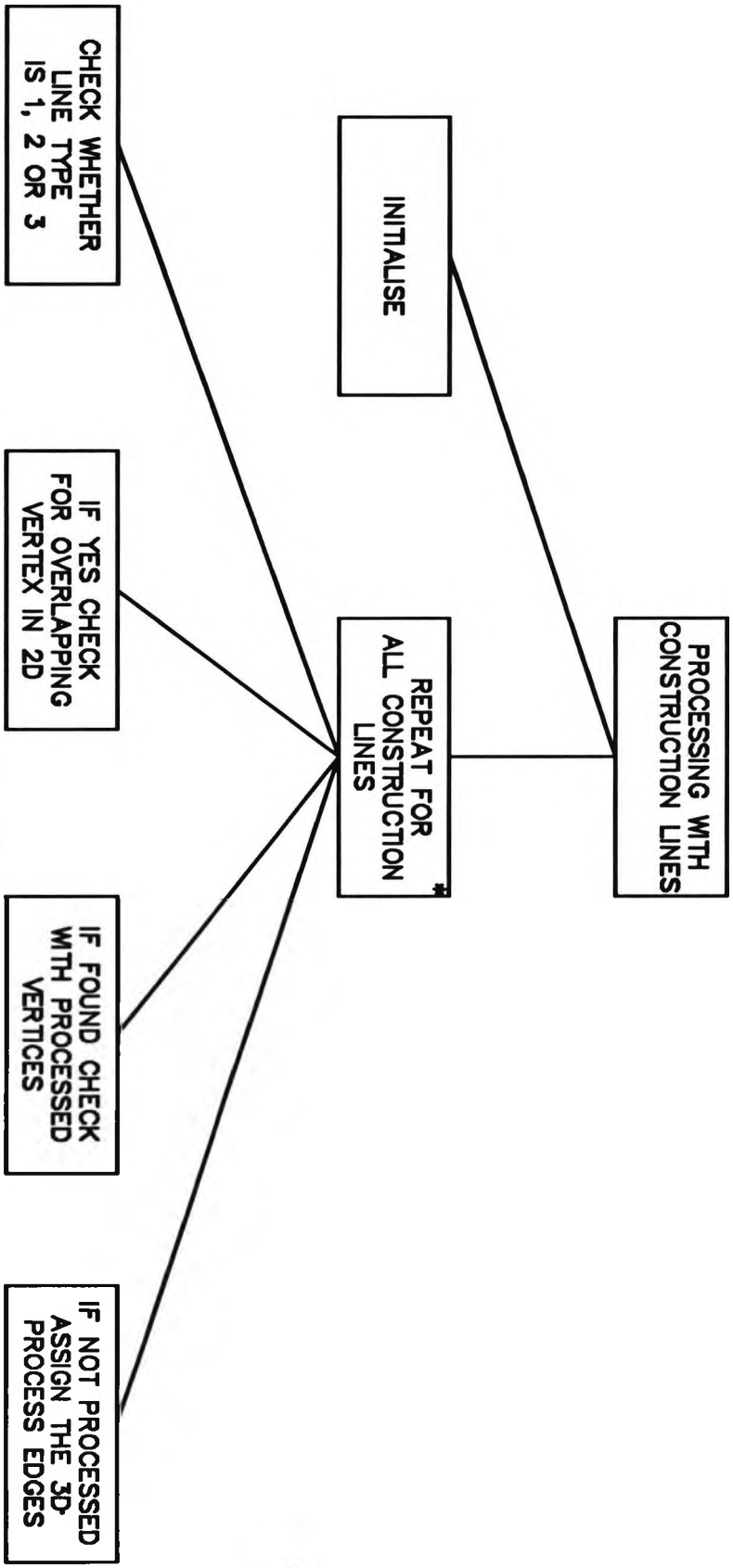

FIGURE 5.19

STRUCTURE OF PROCESSING WITH CONSTRUCTION LINES

5.4.3 PROCESSING OF VERTICES

In this process all the vertices in the vector containing the 'processed vertices' (vertices which are transformed to 3D) are taken one by one and the lines emanating from them are checked. If the lines emanating are of types 1, 2 or 3 and have the other end not transformed to 3D, the transformation is performed. In this implementation this will not produce any extra information but this could be used to process smaller arcs. The algorithm consists of the following steps:

(i) Set the vertex count vc = 1
(ii) Repeat the following until vc becomes equal to PROCVERT (the number of processed vertices in the vector)
(a) Get the list of lines emanating from the vertex vc
(b) Get the number of lines emanating
(c) Repeat for each line the following
(d) Get the other vertex
(e) Check whether it is in the processed list
(f) Process it if the line type is 1, 2 or 3 and the vertex is not already processed
(g) Go to the next vertex

Figure 5.18 illustrates this algorithm.

5.4.4 PROCESSING WITH CONSTRUCTION LINES

Construction lines fall into two kinds with respect to processing. They are (i) Construction lines to assist the construction of the lines in the body (ii) Construction lines to assist the construction of non-isometric construction lines. In this algorithm lines of type 1, 2 or 3 only, are processed in the first pass. Once the isometric lines are processed the non-isometric lines could be transformed by checking each individual edge as described in section 5.4.5 below.

Each line in the list of construction lines is taken individually and is first checked whether it is isometric or not. If they are isometric then it is processed in the following way. The end points in two dimensions are compared with the vertices of the solid in two dimensions. The matching vertex is given the three dimensional co-ordinates of the construction line, if the vertex is not already transformed into three dimensions. This is followed by the processing of all connected lines in the solid, with direct or indirect connections to that vertex. At the end of this process all the vertices in the solid will be known in three dimensions. Figure 5.19 illustrates this algorithm.

5.4.5 PROCESSING FOR EACH EDGE

At the end of processing with construction lines all vertices in three dimensions would be known. This algorithm uses these details to (i) fill the fields of non-isometric edges

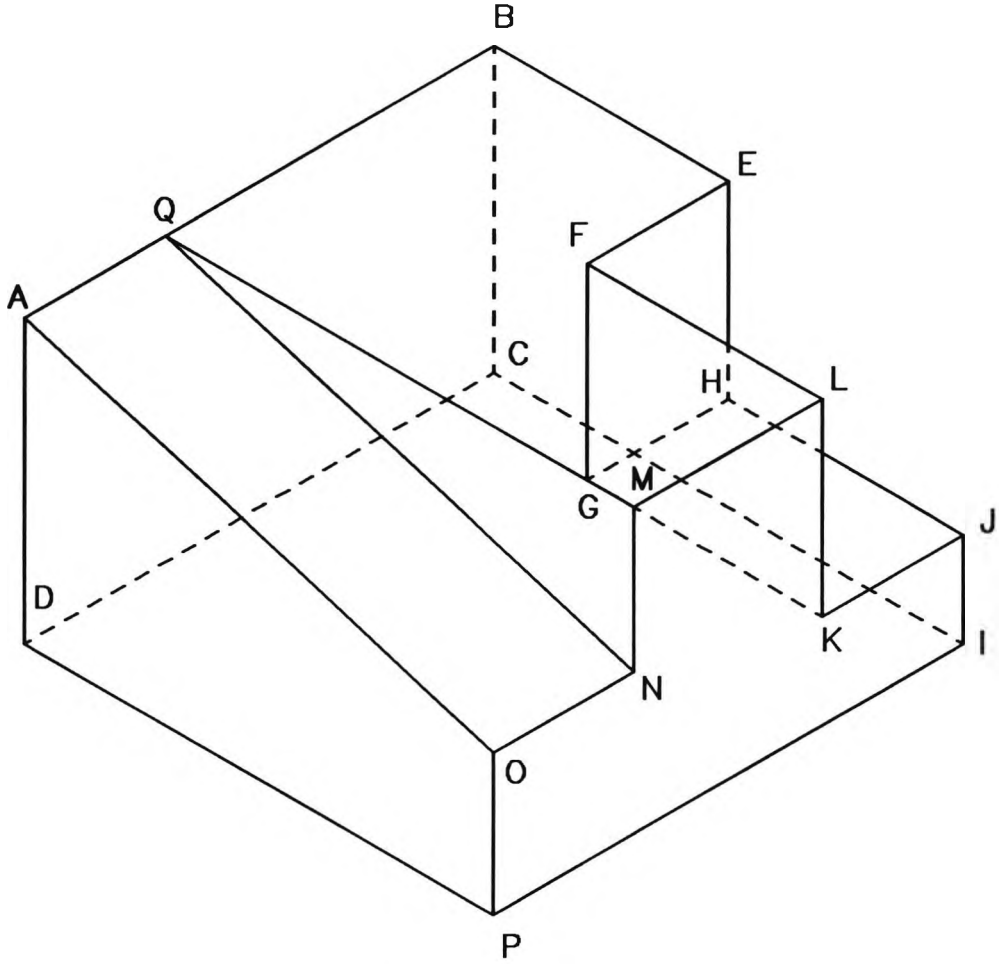

FIGURE 5.20

ILLUSTRATION OF NON-ISOMETRIC LINES WITH ISOMETRIC PROCESSING

in the edge list in three dimensions and (ii) to process arcs and ellipses. Consider figure 5.20. The line NQ is a non-isometric line connecting vertices which would have been transformed to three dimensions by the isometric lines emanating from them. This is a typical circumstance of type (i) described above.

In the case of arcs meeting straight lines their end points would have been transformed to three dimensions by now and only their centres should be transformed. In a similar way for full ellipses their boxes made of construction lines together with the tangential and silhoutte edges of the emerging solid would be known in three dimensions. These are examples of situation described in type(ii) above. This algorithm is written to look after these requirements.

5.4.6 EXTRACTING THREE DIMENSIONAL LOOPS

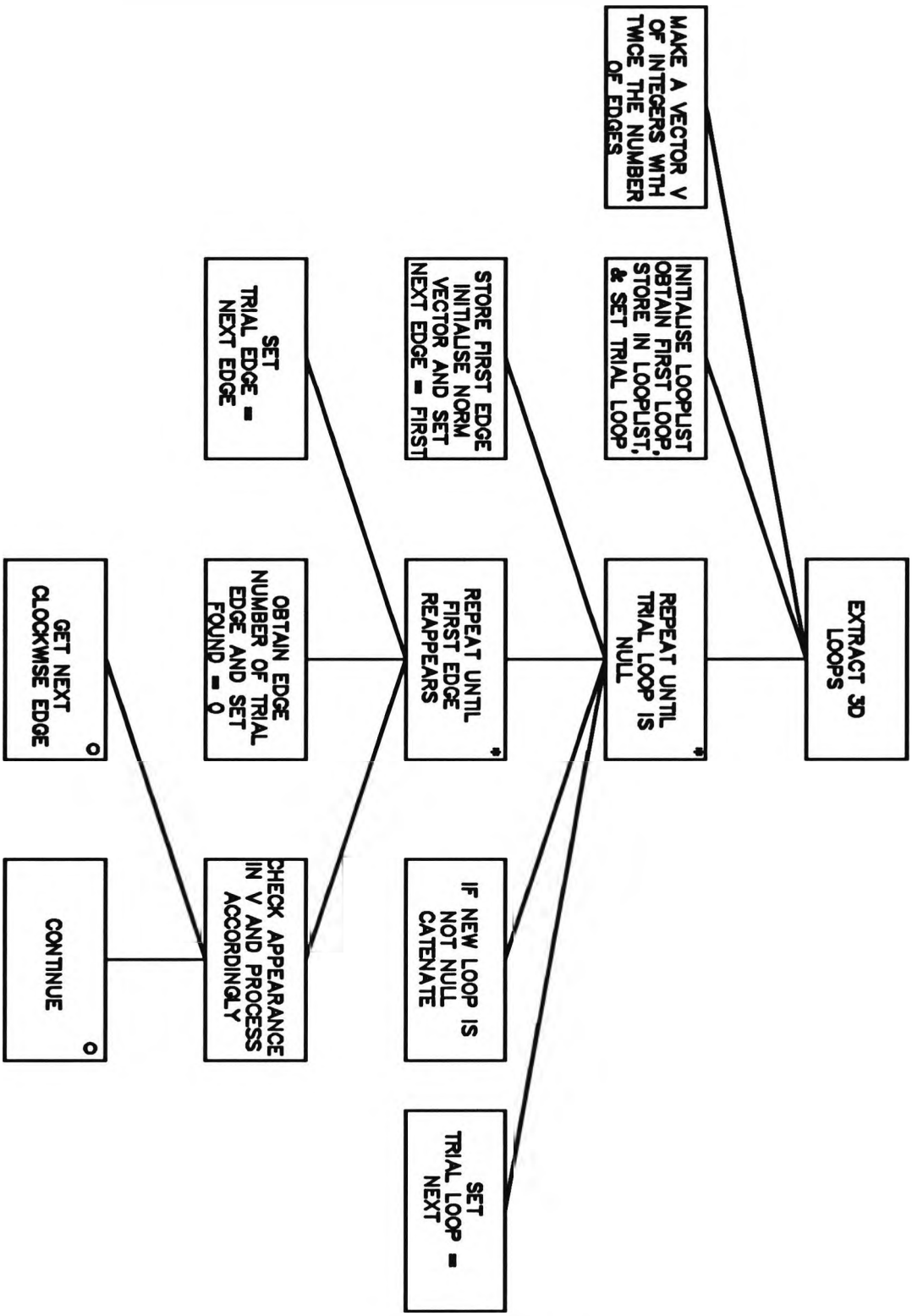

FIGURE 5.21

STRUCTURE OF EXTRACTING 3D LOOPS

This algorithm uses the fact that in a consistently ordered setup of faces each edge would appear in exactly two faces, once in the positive and once in the negative orientation (See section 2.6.2). Thus if there are 'e' number of edges there would be '2e' number of appearances of these edges in the complete list of faces. The algorithm has the following steps:

(i) Make a vector of integers with twice the number of edges, elements.
(ii) Initilise the list of 'Loop3d' nodes and obtain the first clockwise loop from the user
(iii) Store this loop as the first loop in the list and set it as the trial loop.
(iv) Repeat the following until the trial loop is NULL.
(a) Store the first edge in the loop.
(b) Compute the normal vector of the plane consisting the first edge and the next edge in the negative sense of the first edge.
(c) Go to the vertex of the next edge referred in (b) above and obtain the list of edges emanating from it.
(d) Perform a test on the cross products and obtain the next clockwise edge.
(e) Store this edge in the line list of the loop
(f) Continue (c) to (e) until first edge reappears.

Figure 5.21 illustrates this algorithm.

5.4.7 FITTING 3D GEOMETRY

In this implementation holes on curved surfaces are not accommodated. Therefore fitting 3D geometry part consists of only two activities (i) identifying the planar loops and fit their equations and (ii) identifying their inner loops or rings. A loop with no arcs and only straight lines always form a plane. An arc with a tangency edge or silhoutte edge form only a curved surface. These facts are used to decide whether a surface is a plane or not. Plane equations are fitted using the least squares algorithm. Once the equations are fitted inner loops are identified by first comparing the equations and then the enclosed area. Loops with the same equation and smaller area are identified as rings.

5.5 PROGRAM STRATEGY

The program has two parts one written in IBM BASIC and the other written in 'C'. The part written in BASIC accepts the sketch and write it to a file. The part written in 'C' reads this file and process the sketch. Two header files are created for inclusion in this part. They are (i) sketstr.h which defines all the data structures and (ii) globsket.h which declares all the global variables. The program is written in seven files. They are (i) proc2d.c (ii) merge.c (iii) proc3d.c (iv) allocate.c (v) filehand.c (vi) graphout.c and (vii) ss.c. The program in file ss.c is the main driver and all other files contains functions called by other functions. Proc2d.c contains functions in the

processing in two dimensions, merge.c contains functions used during merging and so on. The interim files xxxx.one, xxxx.two and xxxx.thr contain information in ASCII so that they could be accessed by the user or other programs. Samples of these files are given in appendix A.

CHAPTER 6

SAMPLES SESSIONS

6.0 INTRODUCTION

The preceding chapters explained the background, the drawing up of the functional specifications and the development of the program modules to realise the sketching input system. This chapter presents three samples explaining how the processing takes place at various conditions.

6.1 SAMPLE 'L' BLOCK

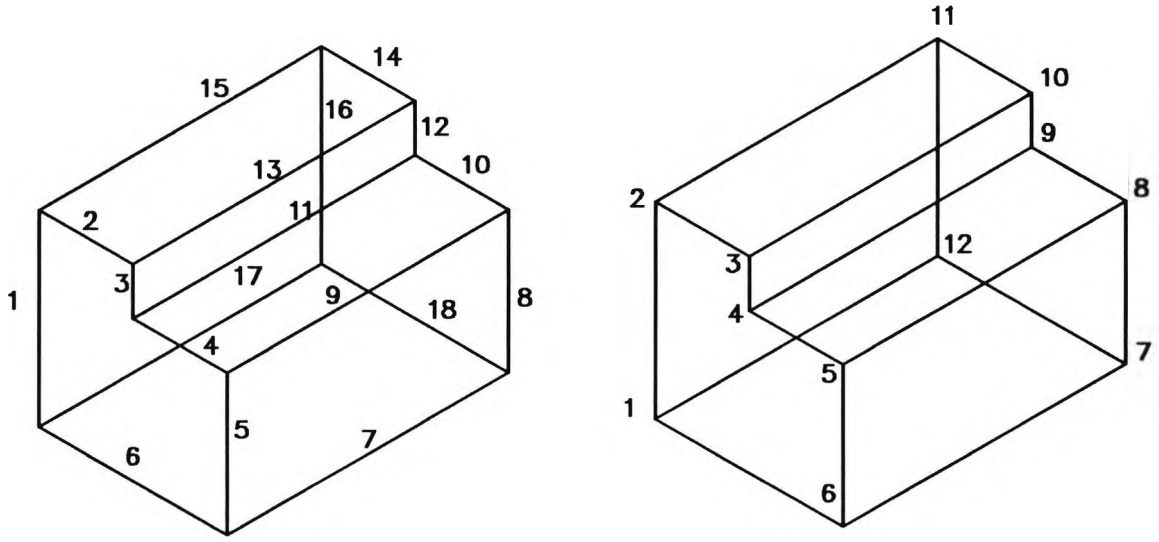

FIGURE 6.1

SIMPLE 'L' BLOCK

Consider figure 6.1 showing the simple 'L' block. The lines are labelled in the order in which they are drawn. Thus line 1 is the first one to draw and line 2 followed it and so on. Lines 1, 2, 3, 4, 5 are drawn as continuous lines. Line 6 is drawn from vertex 1 to vertex 6. The hidden lines and the visible lines in this example are drawn as visible lines. This example is chosen to explain the first working system that was developed during this research. The results referred here are given in the appendix. In the first stage i.e. the sketching part the points in the 18 edges are written to a file called 'lblock.pnt'. The first entry in the file is the three dimensional origin of the block which is vertex 1. It is followed by the number of points in the visible lines which in this case is 389. The next three entries give the points in the hidden lines (zero in this case),

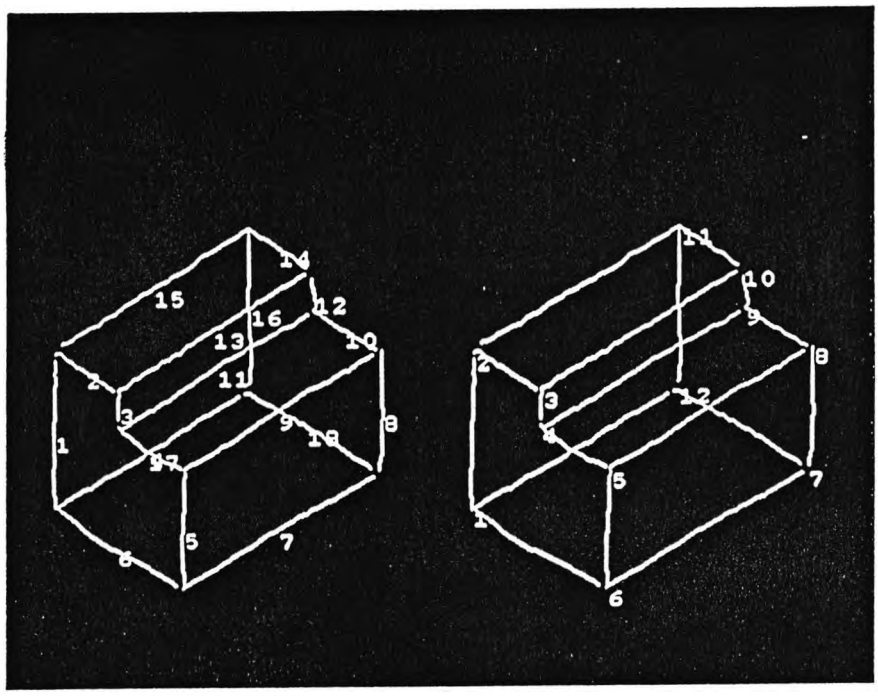

FIGURE 6.2

PHOTOGRAPH OF THE DISPLAYED SKETCH

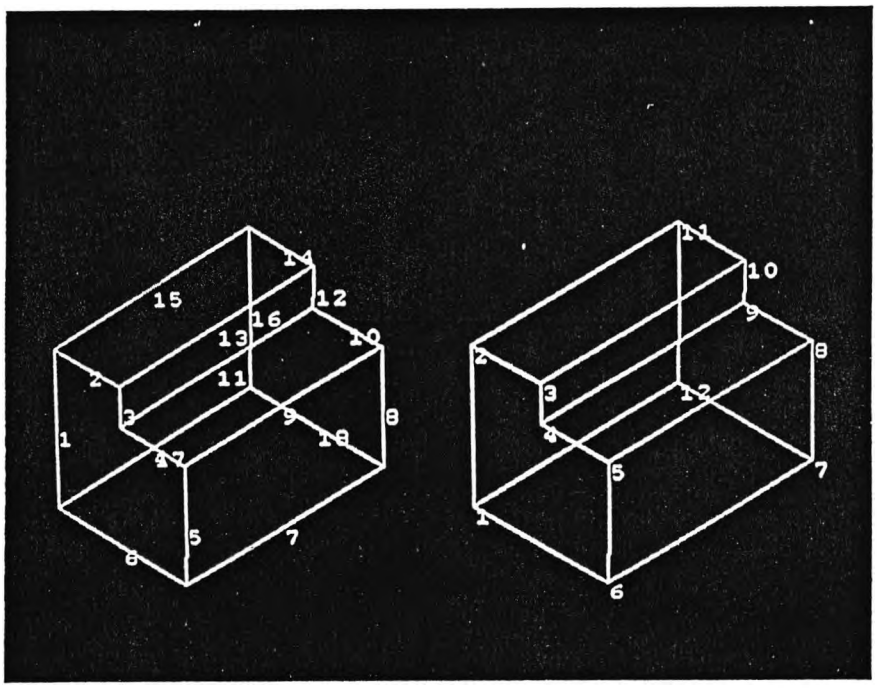

FIGURE 6.3

PHOTOGRAPH OF THE DISPLAYED FITTED SKETCH

centre lines(zero in this case), construction lines(zero in this case) and erased lines(zero in this case). From the seventh entry the 389 points in the visible line are written into the file. Had there been any points from hidden lines they would have followed these visible lines and in a similar fashion the centre lines would have followed the hidden lines, the construction lines would have followed the centre lines and the erased lines would have followed the construction lines. Thus when the "processing in two-dimensions part" access these files it will know whether there are hidden lines, centre lines, construction lines and erased lines in the sketch and the number of points in each category. The entries in this file 'lblock.pnt' is given in appendix C.

6.1.1 PROCESSING IN TWO DIMENSIONS

The first part of this function breaks the 389 points into eighteen groups belonging to the eighteen lines drawn. The details of these groups are stored in eighteen Line_seg nodes which form the linked list 'VISLINESEG'. Once the points belonging to the different line segments are established the sketch could be reproduced. The copies of the screen display of the sketch is given in figure 6.2. The next task is to find the terminal points. These terminal points are points which mark the end of a line segment in the linked list and is close to other terminal points of line segments, which emanate from the vertex, represented by the particular terminal point. Thus for the twelve vertices in the sketch twelve terminal points are established. Looking at the sketch one can clearly see the difference between the terminal points and the vertex. For example look at Vertex 7 in figure 6.2. The edges here are not meeting in the vertex but are terminating at points in the vicinity of vertex 7. The next task is to fit the geometry for the analytic equations of the different line segments.

Eighteen equations are fitted to the eighteen line segments and the details of these equations are kept in the linked list eighteen 'Geom_edge2d' nodes. The next stage of two dimensional processing is to identify the vertices. These are identified by solving the equations fitted to each line segment meeting at any particular vertex. For example vertex 7 is obtained by the least squares solution of 7, 8 and 18. Once these are established the edges in the sketch could be precisely defined and the improved sketch could be drawn. Figure 6.3 shows the copy of the display of the improved 'L' block. The details extracted in these two dimensional processing is written to the file 'lblock.one'.

6.1.2 MERGING

Since there are no hidden lines or erased lines this part of the program does not do any work in this example. However it reads the file lblock.pnt and creates the file containing the details of the vertices, edges and their geometry. These informations are containd in the file 'lblock.two' (see appendix C).

6.1.3 PROCESSING IN THREE DIMENSIONS

In the processing in three dimensions the first part is to identify the 6 loops. Once these are identified equations are fitted to them. The loops then undergo a test to find out whether any two or more of them have the same equation. In this case there are no such faces and therefore no rings are in the solid. If there were rings they would have been identified by comparing their areas. Once the presence or absence of the rings are established then the faces are established. At this stage all the elements of the Euler-Poincare formula are known and validity check is effected. The details upto this point are written in file 'lblock.thr' given in appendix.

To build the solid model from this data the following could be done.
(i) Use the Euler matrix and obtain the Euler co-ordinates (chapter 2)
(ii) Perform the operations as specified by the Euler co-ordinates using the operations available in the solid modeller used

This is the simplest example which is similar to the model that would be developed after merging.

6.2 THE STOPPER BLOCK

This is a hypothetical example of medium complexity chosen to explain the merging facility necessary. In this example there are hidden lines, partially hidden lines and non-isometric lines together with the visible lines. As in the case of the 'L' block the points were accepted by the program and are stored in two arrays namely the VISPOINTS and HIDPOINTS. They are then written to the file. This time the number of hidden points would not be zero. The processing program then read this file of points and breaks it into 21 line segments in the visible line class and 7 line segments in the hidden line class as illustrated in figure 6.4 (b). Fitting the equations and obtaining the co-ordinates of the vertices is a straight forward matter. At this stage the merge function takes over. In the first part no two lines are to be merged because of drawing from both ends or subsequent extensions (i.e. no zero distance merging). In the next stage the hidden lines and visible lines are merged together to form the lines and vertices in the solid. This need the creation of the vertices C and H. The lines MQ and KG, would be identified as lines having the same equation. In a similar fashion the line pairs ER and RH, and HS and SJ also would be identified as lines with same equations. The line pairsER and RH, and HS and SJ should undergo a zero distance merging after the merging of the hidden and visible lines. The lines QM and GK are distinct lines and should not merge even though they have the same equation. This means it is necessary to obtain some input from the user. This of course is done after the user has finished his sketching input. The merging facility in this program is capable of handling similar situations.

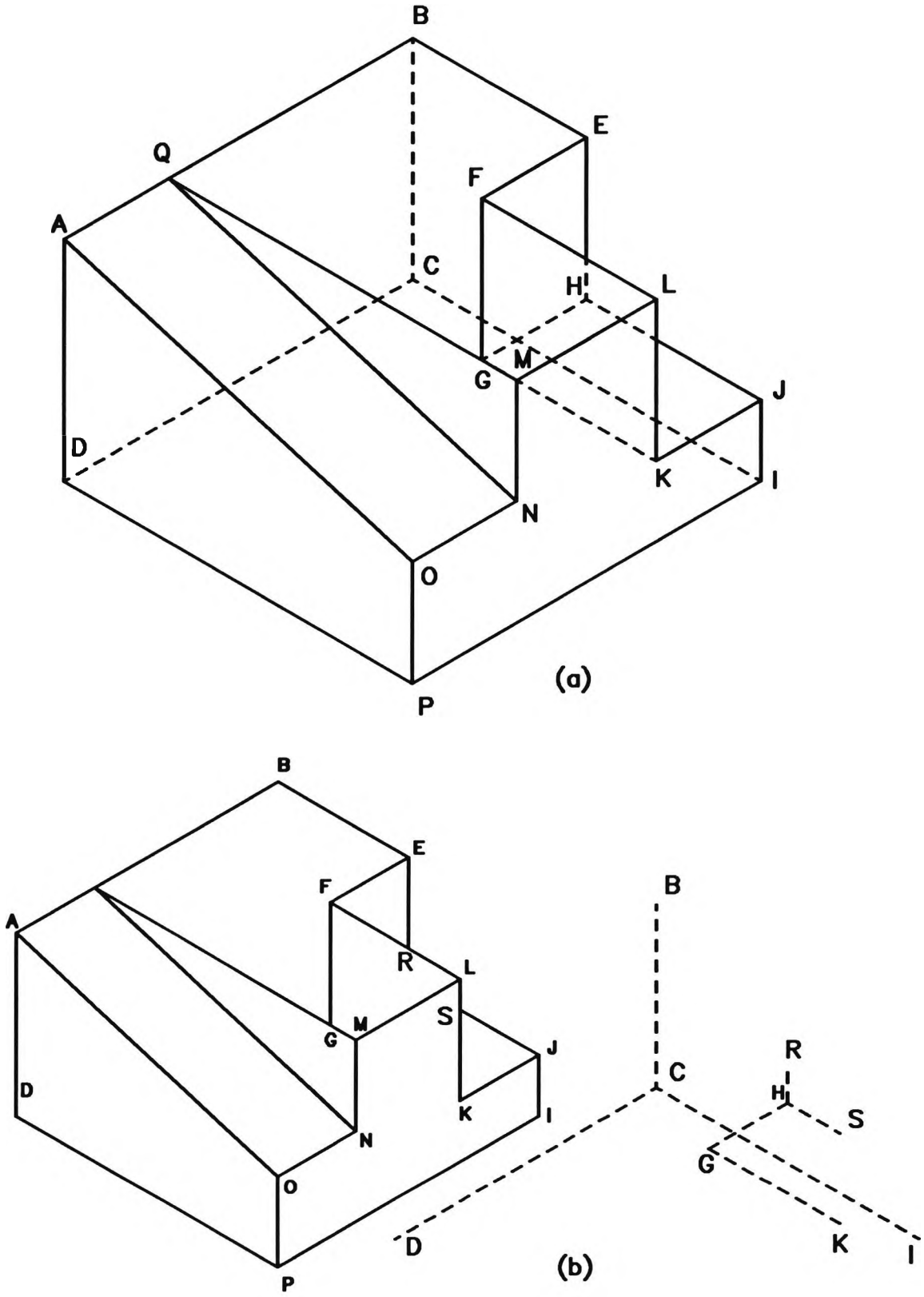

FIGURE 6.4

STOPPER BLOCK

6.3 WALL FIXTURE

This is again a hypothetical example chosen to explain the use of construction lines in three dimensional processing. Figure 6.5 (a) shows the 'wall fixture'. In the first stage the base is sketched as shown in figure 6.5 (b). Then the rectangular block is sketched as shown in figure 6.5 (c). Here the first type of construction line comes into the scene. Vertex H is fixed by this construction line. Figure 6.5 (d) shows the inclusion of the cylindrical hole and the use of construction lines to construct the ellipses. The lines QR and ST together with the ellipses represent the cylinder (see section 2.3.2). The program uses the construction line x and y to fix the construction lines surrounding the ellipses and the co-ordinates of the points Q, R, S and T. These points are then used to fix the centre and radius.

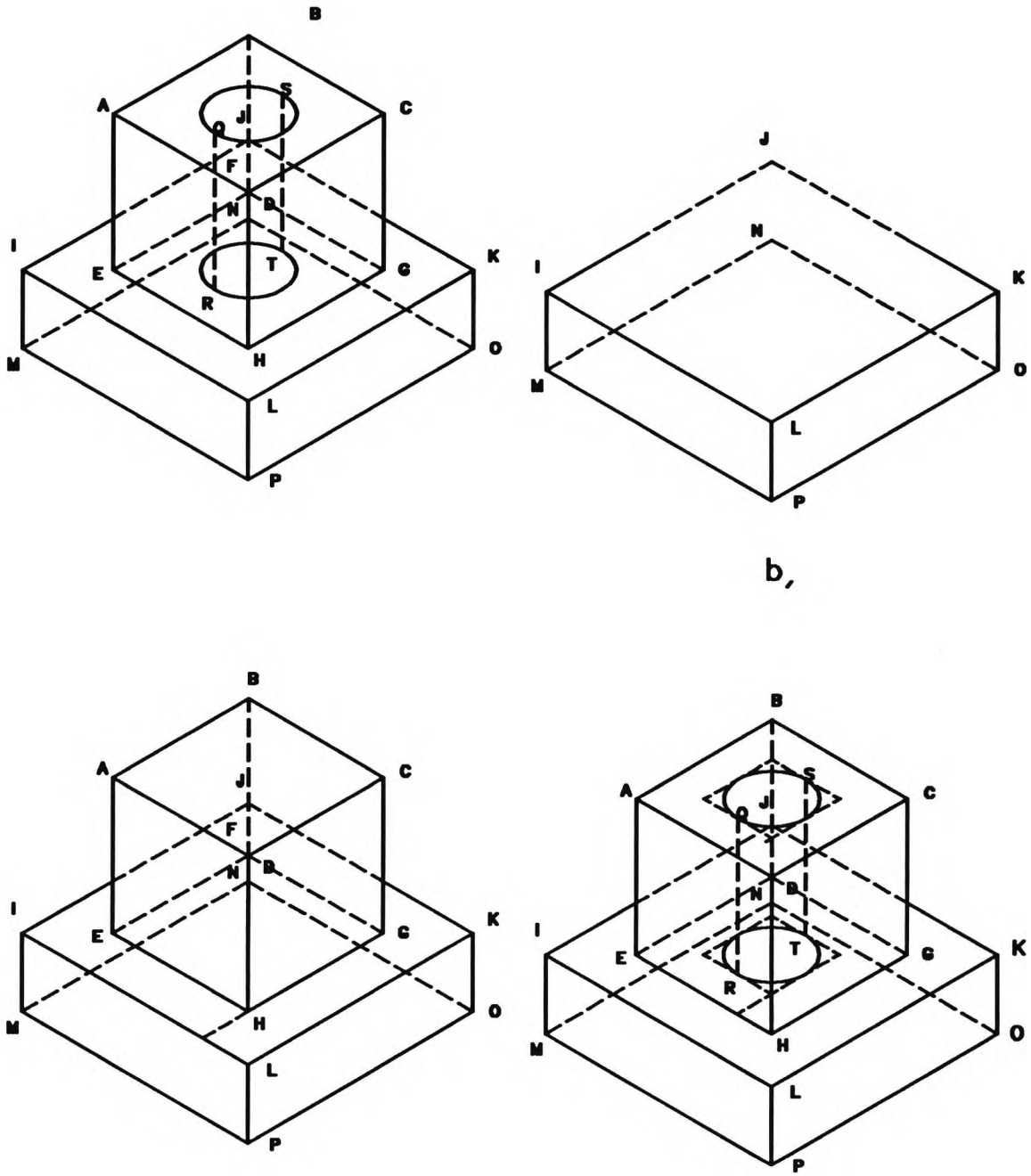

FIGURE 6.5

ILLUSTRATION OF WALL FIXTURE

CHAPTER 7
DISCUSSION AND CONCLUSION

7.0 DISCUSSION

The preceding chapters described the development of a sketching input system, where chapter 1 set the scene by telling why such a system is needed, what is actually needed and how it could be achieved, in a summary form. Chapter 2 described a survey on solid modelling and extracted features of solid modelling that have major influence on the sketching input system. During this process examples are worked and presented by the author to demonstrate the important features. For instance the edge based boundary models in section 2.6.2, the winged-edge structure in section 2.6.3, figure 2.23 representing the full edge list of the 'L' block, list of edge nodes in section 2.9.1.2 illustrating the exactly two representations of each edge and the full list of vertex nodes in figure 2.24 are examples worked out by the author. These examples are all worked out for the same 'L' block to make the understanding easier. The literature on complete examples were limited and therefore it was decided to present these examples in complete form. Chapter 3 described all the theory needed to understand the work described in this thesis. It consists of classical theory as well as the methods developed by the author to solve specific problems using these classical theory. For instance the meeting point of more than two straight lines described in section 3.5, the meeting point of more than two curves and straight lines described in section 3.6 and the twelve classes of isometric lines described on section 3.12 are applications developed by the author. Chapter 4 describes the drawing up of the functional specification in the light of the survey and the theory described earlier. Chapter 5 describes the program in detail. It clearly demonstrates the three stages of processing of the sketch namely

(i) Processing in three dimensions

(ii) Merging of lines in two dimensions and

(iii) Processing in three dimensions

In the 'processing in two dimensions' part the sketch is broken into line segments and lines are fitted to the various sections. In the 'merging in two dimensions' part the lines in the solid are extracted after (i) merging the different segments of the same line (ii) removing the erased lines and (iii) finally by merging the visible and hidden lines together. In the 'processing in three dimensions part' the vertices are first transformed into three dimensions. This is done according to the method described in section 3.13. It is well known that sketching straight lines is much easier than curves and this is why isometric sketching uses straight construction lines heavily (the method of boxing). In the same way processing straight lines is much easier than dealing with

curves. It is because of this reason the use of straight lines to do the transformation is advocated wherever possible in this thesis.

There are two other areas which are necessary to make the sketching input system complete. They are (i) the device dependant sketching part with the involvement of the digitizer and the communication port and (ii) the solid modeller which is going to use this sketching input system. The software was developed using an IBM PS/2 and a CALCOMP 2000 series digitizer. The digitizer area is very small and making a complicated sketch is difficult. The program originally was intended for coupling with Mantyla's [43] 'GWB' solid modeller. Because of the shortcomings in the part of these decisions it was decided to keep the sketching part separately and minimum effort was spent on it. Coupling to an existing solid modeller depends very much on the solid modeller and therefore was not included as part of this work. However the stepwise construction of the solid model by the use of the Euler operators by Mantyla [43] is very much close to this work.

The work was carried out in stages dived along the lines described earlier. First part was to give a simple core structure for the sketching input system which could accept the sketch with minimum of interaction from the user. This needed the automatic identification of different line segments and their class (straight line or curve). Though this appears to be a pattern recognition problem, it is very much different to it. When a line is sketched two kinds of data are generated (i) the explicit co-ordinates of the points and (ii) the implicit order of points (i.e. the points in a line are transmitted in an order from one end to the other). These explicit and implicit data are used in the processing. The first system was very much restrictive without the merge facility (i.e. facilities to erase, facilities to draw a line from both ends etc). The centre lines are not used to any transformation of the sketch. However it was included to accommodate the needs of the solid modelling system (for sweep constructions at a later stage). The processing in three dimensions part is a work in progress and would be complete only after coupling the sketching input system to a solid modeller.

7.2 CONCLUSIONS

A core system for Sketching Input for Computer Aided Engineering, which could accept isometric sketching inputs and develop them as three dimensional models, is developed. It has all the data structures that could accommodate further additions without demanding major re-structuring.

7.3 AREAS FOR FUTURE WORK

(i) The immediate necessity is to couple this to an existing solid modeller.

(ii) It is necessary to go for bigger digitizer and a device driver if necessary.

(iii) It will be for the advantage of the user to see his sketch as he makes it. A

concurrent processing approach in a multitasking environment is another area that needs attention.

(iv) Sketching on its own does not meet the requirements of the applications. The designer should have the facility to sketch a device or system he is developing while having access to libraries of solid models of standard parts such as the gear wheels. This will enable him to use the standard parts wherever appropriate while using his ingenuity or professional skill to develop the product. Thus developing of libraries of primitives is another area that needs researching.

REFERENCES

1 Beasant C.B. and Lui C.W.K.; *Computer Aided Design and Manufacture.* third edition, Ellis Horwood, West Sussex, 1986.

2 Sutherland I.E.; "SKETCHPAD: A man machine graphical communication system," *AFIPS Conference proceedings.*, Vol 12, Spring Joint Computer Conference 1963.

3 Requicha A.G.; "Representation for Rigid Solids: Theory, Methods and Systems," *ACM Computing Surveys.*, Vol 12, No 4, 1980, pp 437- 464.

4 "Solid Modelling: Necessity not Luxury," *Engineering Digest.*, March 1985.

5 Coon S.A.; "An outline for the Requirements for a Computer-Aided design System," *AFIPS Conference Proceedings.*, Vol 23, Spring Joint Computer Conference 1963.

6 Jared, "Solid Modelling," *Colloquium on Design Databases.*, Organised by IEE, Savoy Place, London, March 1986.

7 Samet H. and Webber R.E.; "Data Structures: Heirarchical Data Structures and algorithms for Computer Graphics Part I: Fundamentals," *IEEE* Computer Graphics and Applications., May 1988.

8 Carlbom I. and Chakravarthy I.; "A hierarchical Data Structure for Representing the Spatial Decomposition of 3D objects," *IEEE Computer* Graphics and Applications., April 1985.

9 Kunii T.L., Satoh T. and Yamugachi K.; "Generation of Topological Boundary Representations from Octree Encoding," *IEEE Computer* Graphics and Applications., March 1985.

10 Yamaguchi K. et al., "Octree Related Data Structures and Algorithms," *IEEE Computer Graphics and Applications.*, Vol 4, No 1, January 1984.

11 Yamaguchi K. and Kunii T.L.; "Computer Integrated Manufacturing of Surfaces using Octree Encoding," *IEEE Computer Graphics and Applications.*, Vol 4, No 1, January 1984.

12 Samet H., Weber R.E.; "Data Structures: Hierachical Data Structures for Computer Graphics Part II: Applications," *IEEE Computer Graphics and* Applications., May 1988.

13 Wyvill G., Kunii T.L. and Shirai Y.; "Space division for ray tracing in CSG," *IEEE Computer Graphics and Applications.*, April 1986.

14 Verroust A.; "Visualisation algorithms for CSG polyhedral solids," *Computer Aided Design.*, Vol 19, No 10 1987.

15 Rossignac J.R. and Requicha A.A.G.; " Depth buffering display techniques for Constructive Solid Geometry," *IEEE Computer Graphics and Applications.*, September 1986.

16 Lee Y.C. and Fu K.S.; "Machine understanding of CSG: Extraction and Unification of manufacturing features," *IEEE Computer Graphics and*

Applications., January 1982

17 Toriya H. et al.; "Undo and redo operations for solid modelling," *IEEE Computer Graphics and Applications*., Vol 6, No 4, April 1986.

18 Boyse W.J. and Gilchrist E.J.; "GMsolid: Interactive modelling for design and analysis of solids," *IEEE Computer Graphics and Applications*., March 1982.

19 Tilove B.R.; "Extending solid modelling systems for mechanism design and kinematic simulation," *IEEE Computer Graphics and Applications.*, May/June 1983.

20 Sarrago FR.; "Computation of surface areas in GMSolid," *IEEE* Computer *Graphics and Applications*., March 1982.

21 Saeed S.E., Pennington A.De. and Dodsworth J.R.; "Offsetting in geometric modelling," *Computer Aided Design.*, Vol 20, No 2, March 1988.

22 Wu M.C., Bajaj C.L. and Liu C.R.; "Face area evaluation algorithm for solids," *Computer Aided Design*., Vol 20, No 2, March 1988.

23 Yuen M.M.F., Tan S.T. and Yu K.M.; "Scheme for automatic dimensioning of CSG defined parts," *Computer Aided Design*., Vol 20, No 3, April 1988.

24 Bronsvoot W.F.; "Boundary evaluation and direct display of CSG models," *Computer Aided Design*., Vol 20, No 7, 1988.

25 Verroust A.; "Visualisation algorithm for CSG polyhedral solids," *Computer Aided Design*., Vol 19, No 10, 1987.

26 Bronsvoot W.F.; "Techniques for reducing boolean evaluation time in CSG scan line algorithms," *Computer Aided Design*., Vol 18, No 10, 1986.

27 Chiyokura H. and Kimura H.; "A method of representing the solid design process," *IEEE Computer Graphics and Applications.*, April 1985.

28 Lee T.Y. and Requicha A.A.G.; "Algorithms for computing the volume and other integral properties of solids I: Known methods and open issues," *Communications of the ACM*., Vol 25, No 9, 1982.

29 Lee T.Y. and Requicha A.A.G.; "Algorithms for computing the volume and other integral properties of solids II: A family of algorithms based on representation conversion and cellular approximation," *Communications of the ACM*.,Vol 25, No 9, 1982.

30 Okino N., Kakazu Y. and Morimoto M.; "Extended depth buffer algorithms for hidden surface visualization," *IEEE Computer Graphics and Applications*., May 1984.

31 Hilyard R.; "The Build group of solid modellers," *IEEE Computer Graphics and Applications*., September 1982.

32 Braid I.C.; "Geometric modelling," *Advances in computer graphics I*. Edited by Endrie C., Grave M. and Lillehagen F.; Springer Verlag 1986.

33 Pratt M.J.; "Interactive geometric modelling for integrated CAD/CAM," *Advances in Computer Graphics I*. Edited by Endrie C., Graves M. and Lillehagen; Springer Verlag 1986.

34 Kalay E.W., "Worldview: An integrated modelling/drafting system,"

IEEE Computer Graphics and Applications., February 1987.

35 Peterson D.P.; "Boundary to constructive solid geometry mapping," *Computer Aided Design.*, Vol 18, No 1, 1986.

36 Lee G. and Gossard C.D.; "A hierachical data structure for a representing assemblies Part I," *Computer Aided Design.*, Vol 17, No 1, 1985.

37 Roy V. and Liu C.R.; "Establishment of functional relationship between product components in assembly database," *Computer Aided Design.*, Vol 20, No 10, Dec 1988.

38 Wilson P.R. et al., "Interface for data transfer between solid modelling systems," *IEEE Computer Graphics and Applications.*, Jan 1985.

39 Weiler K., "Edge based data structures for solid modelling in curved surface environments," *IEEE Computer Graphics and Applications*s., January 1985.

40 Mantyla M. and Sulonen R.; "GWB: A solid modeler with Euler operators," *IEEE Computer Graphics and Applications.*, September 1982.

41 Chiyokura H.; "An Introduction to Solid Modelling," 1988.

42 Mantyla M.; *An introduction to Solid Modelling.* Computer Science Press, Maryland, 1988.

43 Wilson P.R.; "Euler formulas and geometric modelling," *IEEE Computer Graphics and Applications.*, August 1985.

44 Braid I.C.; "Six systems for shape design and representation," *CAD group document No 87.*, University of Cambridge 1975.

45 Baer A., Eastman C. and Hanrion M.; "Geometric modelling: A survey," *Computer Aided Design.*, September 1979.

46 Braid I.C., Hillyard R.C. and Stroud I.A.; *Stepwise construction of polyhedra in geometric modelling.* Computer Laboratory, University of Cambridge.

47 Braid I.C.; *Designing with Volumes.* PhD thesis, Darwin College, Cambridge, 1973.

48 Joshi S. and Chang T.C.; "Graph-based heurestics for recognition of machined features from a 3D solid model," *Computer Aided Design.,* Vol 20, No 2, 1988.

49 Holstrom L. and Laako T.; " Rounding facility for solid modelling of mechanical parts," *Computer Aided Design.*, Vol 20, No 10, 1988.

50 Ala S.R.; "Design methodologies of boundary data structures," *ACM Symposium on Solid modelling foundations & CAD/CAM applications.*, Austin, USA, June 1991.

51 Requicha A.A.G. and Voelcker H.B.; " Solid modelling: A historical summary and contemporary assesment," *IEEE Computer Graphics and Applications.*, March 1982.

52 Requicha A.A.G. and Voelcker H.B.; "Solid modelling: Current status and research directions," *IEEE Computer Graphics and Applications.,* October,1983.

53 Wilson P.R.; "Euler formulas and geometric modelling," *IEEE Computer*

Graphics And Applications., August 1985.

54 Mantyla M.; "A note on the modelling space of Euler Operators," *Computer vision, graphics and image processing*., Vol 26, No 1, April 1984.

55 Eastman C.M. and Weiler K.; "Geometric modelling using the Euler operators," *Proc first Annual Conference on Computer Graphics and CAD/CAM systems.,* MIT press, Cambridge, Mass., Apr 1979.

56 Lequette R.; "Automatic construction of curvlinear solids from wireframe views," *Computer Aided Design*., Vol 20, No 4, May 1988.

57 Fukui Y.; "Input method of boundary solid by sketching," *Computer Aided Design*., Vol 20, No 8, Oct 1988.

58 Hodes l.; *Machine processing of line drawings*. MIT, Lincoln Laboratory Report no 54G-0028, 1961.

59 Guzman A.; "Decomposition of a visual scene into three dimensional bodies," *AFIPS: Fall Joint Computer Conference*., Vol 33, Part I, 1968.

60 Murase H. and Wakahara T.; "Online hand sketched figure recognition," *Pattern Recognition*., Vol 19, No 2, 1986.

61 Richards T.H. and Onwubolu G.C.; "Automatic interpretation of engineering drawings for 3D surface representation in CAD," *Computer Aided Design*., Vol 18, No 3, April 1986.

62 Hwang T.S. and Ullman D.G.; "The design capture system: Capturing back -of-the envelope sketches," *International conference on engineering design*., ICED 1990.

63 Woo T.C.; "A combinatorial analysis of boundary data schemata," *IEEE* Computer Graphics and Applications., March 1985.

64 Baumgart B.G.; *Winged edge polyhedron representation*. Stanford Artificial Intelligence Project, Memo AIM-179, October 1972.

65 Francis A.; *Advanced level statistics, an integrated course*. second edition, Stanley Thornes Ltd, Cheltnam,UK.

66 Thomas S.M and Chan Y.T.; "A simple approach for the estimation of circular arc centre and its radius," *Computer vision, Graphics and Image* Processing., Vol 45,1989.

67 Berman M.; "Large sample bias in least squares estimators of a circular arc centre and radius," *Computer vision, Graphics and Image Processing*., Vol 45, 1989.

68 Crouch E.G.A.; "Averbury circle: its geometry and metrology," *Science and* archeology., No 22, 1980, pp 32-34.

69 Angell I. and Barber J.; "An algorithm for fitting circles and ellipses to megalithic stone rings," *Science and archeology*., No 20, 1977, pp 11-16.

70 Smith R.W.; "Computer processing of line images: A survey," *Pattern Recognition*., Vol 20, No 1, Jan 1987.

71 Mantas J.; "Methodologies in pattern recognition and image analysis - A brief survey," *Pattern Recognition.,* Vol 20, No 1, Jan 1987.

72 Basu S. and Fu K. S.; "Image segmentation by syntatic method," *Pattern Recognotion.,* Vol 19, No 2, 1986.

73 Landu U. M.; "Estimation of a Circular Arc Centre and its Radius," *Computer vision, Graphics and Image Processing.*, Vol 38, 1987, pp 317-326.

74 Nagata T., Tamura H. and Ishibashi K.; "Detection of an Ellipse by use of a Recursive Least-Squares Estimator.," *Journal of Robotic Systems.*, John Wiley & Sons inc, 1985, pp 163-177.

75 Mardia K.V. and Holmes D.; "A Statistical Analysis of Megalithic Data under Elliptic Pattern," *Journal of Royal Statistical Society.*, Vol 143, Part 3, 1980.

76 Nagata T., Tamura H. and Ishibashi K.; "Detection of an Ellipse by the use of a recursive least squares estimator," *Journal of Robotic Systems.*, Vol 2, No2, 1985

77 Dewy B.R.; *Computer Graphics for Engineers.* Harper & Row publishers, NewYork, 1988.

78 Luzador W.J.; *Fundamentals of Engineering Drawing for design, product development and numerical control.* Eighth Edition, Prentice Hall Inc., Englewood Cliffs, NJ, 1981.

79 Askwith E.H.; *The Analytical Geometry of the Conic Sections.* A&C Black Ltd London 1935.

80 Grieve B.; *Analytical Geometry, Part 1.* Bell and Sons Ltd, London 1926.

81 Levens A. and Chalk W.; *Graphics in Engineering Design.* John Weily & Sons 1980.

82 Gibby J.C.; *Technical Illustration Procedure and Practice.* Third Edition American Technical Society, 1970.

83 Giesecke F.E.; *Engineering Graphics.* Fourth Edition, Macmillan Publishing Company, 1987.

84 Hoelscher P.R. and Springer C.H.; *Engineering Drawing and Geometry.* John Weily, 1956.

85 Sutton E.; *Free Hand Sketching.* Stam Press Ltd, England, 1973.

86 Press W.H., Flannery B. P., Teukolsky S. A.and Vetterling W. T.; Numerical Recipies in C. Cambridge University press,1988.

APPENDIX A

MANTYLA'S DATA STRUCTURES

TYPE DEFINITIONS

```
typedef    float    vector [4];
typedef    float    matrix[4][4];
typedef    short    Id;
typedef    struct   solid      Solid;
typedef    struct   face       Face;
typedef    struct   loop       Loop;
typedef    struct   halfedge   HalfEdge;
typedef    struct   vertex     Vertex;
typedef    struct   edge       Edge;
typedef    struct   line       Line;
typedef    struct   arc        Arc;
typedef    struct   plane      Plane;
typedef    struct   cylinder   Cylinder;
typedef    union    curve      Curve;
typedef    union    surf       Surf;
```

'C' STRUCTURES

```
(i) struct solid
        {
        Id        solidno;        /* solid identifier */
        Face      *sfaces;        /* pointer to list of faces */
        Edge      *sedges;        /* pointer to outer loop */
        Vertex    *sverts;        /* pointer to list of vertices */
        Solid     *nexts;         /* pointer to next solid */
        Solid     *prevs;         /* pointer to previous solid */
        };
(ii) struct face
        {
        Id        faceno;         /* face identifier */
        Solid     *fsolid;        /* back pointer to solid */
        Loop      *flout;         /* pointer to outer loop */
        Loop      *floops         /* pointer to list of loops */
        Surf      *fsurf;         /* pointer to the surface information */
        Face      *nextf;         /* pointer to the next face */
        Face      *prevf;         /* pointer to previous face */
        };
```

(iii) struct loop

```
{
HalfEdge    *ledg;      /* pointer to ring of halfedges */
Face        *lface;     /* back pointer to face */
Loop        *nextl;     /* pointer to next loop */
Loop        *prevl;     /* pointer to previous loop */
};
```

(iv) struct plane

```
{
short       surf_type;  /* surface type */
short       times_used;
real        a, b, c, d;
};
```

(v) struct cylinder

```
{
short       surf_type;
short       times_used;
matrix      cy_transf;
real        cy_rad;
real        cy_h;
};
```

(vi) struct cone

```
{
short       surf_type;
short       times_used;
matrix      co_transf;
real        co_top_rad;
real        co_bottom_rad;
real        co_h;
};
```

(vii) struct sphere

```
{
short       surf_type;
short       times_used;
matrix      sph_transf;
real        sph_rad;
};
```

(viii) struct line

```
{
short       curve_type;
short       times_used;
};
```

(ix) struct arc

```
{
short       curve_type;
short       times_used;
real        arc_rad;
real        arc_cx, arc_cy;     /* centre */
real        arc_phi1;           /* start angle */
real        arc_phi2;           /* end angle */
plane       *arc_plane;         /*plane of the arc */
};
```

(x) struct edge

```
{
HalfEdge    *he1;               /* pointer to right half edge */
HalfEdge    *he2;               /* pointer to left half edge */
Curve       *ecurve;            /* curve information */
Edge        *nexte;             /* pointer to next edge */
Edge        *preve;             /* pointer to previous edge */
};
```

(xi) struct halfedge

```
{
Edge        *edg;               /* pointer to parent edge */
Vertex      *vtx;               /* pointer to a starting vertex */
Loop        *wloop;             /* back pointer to the loop */
HalfEdge    *nxt;               /* pointer to next half edge */
HalfEdge    *prv;               /* pointer to previous half edge */
};
```

(xii) struct *vertex

```
{
Id          vertexno;           /* vertex identifier */
HalfEdge    *vedge;             /* pointer to a half edge */
vector      vcoord;             /* vertex coordinates */
Vertex      *nextv;             /* pointer to next vertex */
Vertex      *prevv;             /* pointer to previous vertex */
};
```

(xiii) union surf

```
{
Plane        p;
Cylinder     cy;
Cone         co;
sphere       sph;
};
```

(xiv) union curve

```
{
Line         l;
Arc          a;
};
```

APPENDIX B

LISTINGS OF PROGRAMS

SKETCH.BAS PROGRAM

```
1000 REM                    SKETCH.BAS
1010 REM
1020 REM
1030 REM ---------- DECLARATIONS -------------
1040 REM
1050 SCREEN 9
1060 COLOR 15,1
1070 DIM VISLINES%(5000,2)
1080 DIM HIDLINES%(1000,2)
1090 DIM CENTLINES%(1000,2)
1100 DIM CONSTLINES%(1000,2)
1110 DIM ERASEDLINES%(1000,2)
1120 VISPOINTS = 0
1130 CENTREPOINTS = 0
1140 HIDDENPOINTS = 0
1150 CONSTPOINTS = 0
1160 ERASEPOINTS = 0
1170 REM
1180 REM ------------ FILE DETAILS -------------
1190 REM
1200 CLS
1210 PRINT : PRINT : PRINT
1220 PRINT : PRINT : PRINT
1230 PRINT
1240 PRINT
1250 PRINT TAB(15) "ENTER THE NAME OF THE SKETCH"
1260 PRINT
1270 PRINT
1280 INPUT FILENAME$
1290 FILENAME$ = FILENAME$ + ".PNT"
1300 PRINT : PRINT :PRINT
1310 PRINT TAB(15)   "ENTER WHETHER THE FILE IS OLD OR NEW"
1320 PRINT : PRINT
1330 INPUT OLDORNEW$
1340 IF NOT ((OLDORNEW$ "OLD") XOR (OLDORNEW$"NEW")) GOTO
```

```
1310
1350 REM
1360 REM --------- OPEN FILE FOR WRITING ---------
1370 REM
1380 IF (OLDORNEW$ = "OLD") GOTO 1410
1390 OPEN FILENAME$ FOR OUTPUT ACCESS WRITE AS #2
1400 GOTO 1710
1410 OPEN FILENAME$ FOR INPUT ACCESS READ AS #2
1420 INPUT #2,THREEX%, THREEY%
1430 PRINT THREEX%
1440 INPUT #2,VISPOINTS%
1450 INPUT #2,HIDDENPOINTS%
1460 INPUT #2,CENTREPOINTS%
1470 INPUT #2,CONSTPOINTS%
1480 INPUT #2,ERASEDPOINTS%
1490 IF (VISPOINTS% = 0) GOTO 1530
1500 FOR I% = 1 TO VISPOINTS% STEP 1
1510 INPUT #2,VISLINES%(I%,1),VISLINES%(I%,2)
1520 NEXT
1530 IF (HIDDENPOINTS% = 0) GOTO 1570
1540 FOR I% = 1 TO HIDDENPOINTS% STEP 1
1550 INPUT #2,HIDLINES%(I%,1),HIDLINES%(I%,2)
1560 NEXT
1570 IF (CENTREPOINTS% = 0) GOTO 1610
1580 FOR I% = 1 TO CENTREPOINTS% STEP 1
1590 INPUT #2,CENTLINES%(I%,1),CENTLINES%(I%,2)
1600 NEXT
1610 IF (CENTREPOINTS% = 0) GOTO 1650
1620 FOR I% = 1 TO CONSTPOINTS% STEP 1
1630 INPUT #2,CONSTLINES%(I%,1),CONSTLINES%(I%,2)
1640 NEXT
1650 IF(ERASEDPOINTS = 0) GOTO 1710
1660 FOR I% = 1 TO ERASEDPOINTS% STEP 1
1670 INPUT #2,ERASEDLINES%(I%,1),ERASEDLINES%(I%,2)
1680 NEXT
1690 CLOSE #2
1700 OPEN FILENAME$ FOR OUTPUT ACCESS WRITE AS #2
1710 REM ---------- OPEN COM1 PORT ------------
1720 REM
1730 OPEN "COM1:4800,E,7,1,CS,DS,CD" AS #1
1740 CLS
```

```
1750 PRINT : PRINT : PRINT : PRINT :PRINT
1760 REM
1770 REM ---------- OBTAIN ORIGIN ------------
1780 REM
1790 DIGMENU% = 0
1800 PRINT TAB(15) "INDICATE YOUR  3D  ORIGIN IN THE DIGITIZER"
1810 REM
1820 INPUT #1,THREEX%,THREEY%,N%
1830 REM
1840 REM ---------- START SKETCHING -----------
1850 BEEP
1860 CLS
1870 PRINT : PRINT : PRINT : PRINT :PRINT :PRINT
1880 PRINT TAB(19) "YOU CAN START/PROCEED YOUR SKETCH BY
USING THE"
1890 PRINT: PRINT
1900 PRINT TAB(30) "DIGITIZER MENU"
1910 INPUT #1,X%,Y%,N%
1920 IF (X% 1700) GOTO 1860
1930 INPUT #1,X%,Y%,N%
1940 IF ((Y% 320) AND (Y% )) THEN DIGMENU% = 1
1950 IF ((Y% 520) AND (Y% Ò)) THEN DIGMENU% = 2
1960 IF ((Y% 720) AND (Y% ÿ)) THEN DIGMENU% = 3
1970 IF ((Y% 920) AND (Y% ‘)) THEN DIGMENU% = 4
1980 IF ((Y% 1120) AND (Y% ()) THEN DIGMENU% = 5
1990 IF ((Y% 1520) AND (Y% Ō)) THEN DIGMENU% = 6
2000 ON (DIGMENU%)GOSUB 2050, 2290, 2530, 2770, 2930, 3150
2010 END
2020 REM
2030 REM ----------- VISIBLE LINE PROCESSING --------
2040 REM
2050 CLS
2060 PRINT : PRINT : PRINT : PRINT : PRINT
2070 PRINT TAB(25) "VISIBLE LINE PROCESSING ! "
2080 BEEP
2090 X1% = 0
2100 Y1% = 0
2110 INPUT #1,X%,Y%,N%
2120 IF(X% 1700) AND (Y% ) GOTO 2110
2130 INPUT #1,X%,Y%,N%
2140 IF (X% 1700) THEN GOTO 2250
```

```
2150 DELTAX = X%-X1%
2160 DELTAY = Y%-Y1%
2170 DIST = SQR(DELTAX*DELTAX+DELTAY*DELTAY)
2180 IF (DIST !) GOTO 2130
2190 VISPOINTS%=VISPOINTS%+1
2200 VISLINES%(VISPOINTS%,1)=X%
2210 VISLINES%(VISPOINTS%,2)=Y%
2220 X1%=X%
2230 Y1%=Y%
2240 GOTO 2130
2250 RETURN 1860
2260 REM
2270 REM - - - - - - - - - HIDDEN LINE PROCESSING - - - - - - - - -
2280 REM
2290 CLS
2300 PRINT : PRINT : PRINT : PRINT : PRINT
2310 PRINT TAB(25) "HIDDEN LINE PROCESSING ! "
2320 BEEP
2330 X1%=0
2340 Y2%=0
2350 INPUT #1,X%,Y%,N%
2360 IF(X% 1700) AND (Y% Ò) GOTO 2350
2370 INPUT #1,X%,Y%,N%
2380 IF (X% 1700) THEN GOTO 2490
2390 DELTAX = X%-X1%
2400 DELTAY = Y%-Y1%
2410 DIST = SQR(DELTAX*DELTAX + DELTAY*DELTAY)
2420 IF (DIST 15!) GOTO 2370
2430 HIDDENPOINTS%=HIDDENPOINTS%+1
2440 HIDLINES%(HIDDENPOINTS%,1)=X%
2450 HIDLINES%(HIDDENPOINTS%,2)=Y%
2460 X1%=X%
2470 Y1%=Y%
2480 GOTO 2370
2490 RETURN 1860
2500 REM
2510 REM - - - - - - - - - CENTRE LINE PROCESSING - - - - - - - - -
2520 REM
2530 CLS
2540 PRINT : PRINT : PRINT : PRINT : PRINT
2550 PRINT TAB(25) "CENTRE LINE PROCESSING ! "
```

```
2560 BEEP
2570 X1% = 0
2580 Y1% = 0
2590 INPUT #1,X%,Y%,N%
2600 IF(X% 1700) AND (Y% ÿ) GOTO 2590
2610 INPUT #1,X%,Y%,N%
2620 IF (X% 1700) THEN GOTO 2730
2630 DELTAX = X%-X1%
2640 DELTAY = Y%-Y1%
2650 DIST = SQR(DELTAX*DELTAX + DELTAY*DELTAY)
2660 IF(DIST  15!) GOTO 2610
2670 CENTREPOINTS% = CENTREPOINTS% + 1
2680 CENTLINES%(CENTREPOINTS%,1) = X%
2690 CENTLINES%(CENTREPOINTS%,2) = Y%
2700 X1% = X%
2710 Y1% = Y%
2720 GOTO 2610
2730 RETURN 1860
2740 REM
2750 REM - - - - - - - - - CONSTRUCTION LINE PROCESSING - - - - - -
2760 REM
2770 CLS
2780 PRINT : PRINT : PRINT : PRINT : PRINT
2790 PRINT TAB(25) "CONSTRUCTION LINE PROCESSING !"
2800 BEEP
2810 INPUT #1,X%,Y%,N%
2820 IF(X% 1700) AND (Y% ÿ) GOTO 2810
2830 INPUT #1,X%,Y%,N%
2840 IF (X% 1700) THEN GOTO 2890
2850 CONSTPOINTS% = CONSTPOINTS% + 1
2860 CONSTLINES%(CONSTPOINTS%,1) = X%
2870 CONSTLINES%(CONSTPOINTS%,2) = Y%
2880 GOTO 2830
2890 RETURN 1860
2900 REM
2910 REM - - - - - - - - ERASED LINES - - - - - - - - - - - - - - - -
2920 REM
2930 CLS
2940 PRINT : PRINT : PRINT : PRINT : PRINT
2950 PRINT TAB(25) "ERASED LINE PROCESSING !"
2960 BEEP
```

```
2970 X1% = 0
2980 Y1% = 0
2990 INPUT #1,X%,Y%,N%
3000 IF(X% 1700) AND (Y% ÿ) GOTO 2990
3010 INPUT #1,X%,Y%,N%
3020 IF (X% 1700) THEN GOTO 3110
3030 DELTAX = X%-X1%
3040 DELTAY = Y%-Y1%
3050 DIST = SQR(DELTAX%*DELTAX% + DELTAY%*DELTAY%)
3060 IF(DIST !) GOTO 3010
3070 ERASEDPOINTS% = ERASEDPOINTS% + 1
3080 ERASEDLINES%(ERASEDPOINTS%,1) = X%
3090 ERASEDLINES%(ERASEDPOINTS%,2) = Y%
3100 GOTO 3010
3110 RETURN 1860
3120 REM
3130 REM - - - - - - - - END - - - - - - - - - - - - - - - - - - - -
3140 REM
3150 WRITE #2, THREEX%, THREEY%
3160 WRITE #2, VISPOINTS%
3170 WRITE #2, HIDDENPOINTS%
3180 WRITE #2,CENTREPOINTS%
3190 WRITE #2,CONSTPOINTS%
3200 WRITE #2,ERASEDPOINTS%
3210 IF (VISPOINTS%  0) THEN GOSUB 3270
3220 IF (HIDDENPOINTS%  0) THEN GOSUB 3310
3230 IF (CENTREPOINTS%  0) THEN GOSUB 3350
3240 IF (CONSTPOINTS%  0) THEN GOSUB 3390
3250 IF (ERASEDPOINTS%  0) THEN GOSUB 3430
3260 RETURN 2010
3270 FOR I% = 1 TO VISPOINTS% STEP 1
3280 WRITE #2,VISLINES%(I%,1),VISLINES%(I%,2)
3290 NEXT
3300 RETURN
3310 FOR I% = 1 TO HIDDENPOINTS% STEP 1
3320 WRITE #2,HIDLINES%(I%,1),HIDLINES%(I%,2)
3330 NEXT
3340 RETURN
3350 FOR I% = 1 TO CENTREPOINTS% STEP 1
3360 WRITE #2,CENTLINES%(I%,1),CENTLINES%(I%,2)
3370 NEXT
```

```
3380 RETURN
3390 FOR I% = 1 TO CONSTPOINTS% STEP 1
3400 WRITE #2,CONSTLINES%(I%,1),CONSTLINES%(I%,2)
3410 NEXT
3420 RETURN
3430 FOR I% = 1 TO ERASEDPOINTS% STEP 1
3440 WRITE #2,ERASEDLINES%(I%,1),ERASEDLINES%(I%,2)
3450 NEXT
3460 RETURN
```

SKETCH SOLID - DATA STRUCTURES

```
typedef struct intnode Intnode;
typedef struct lineseg Lineseg;
typedef struct termpoint Termpoint;
typedef struct geom_edge2d Geom_edge2d;
typedef struct vert2d Vert2d;
typedef struct edge2d Edge2d;
typedef struct loop3d Loop3d;
typedef struct edge3d Edge3d;
typedef struct vert3d Vert3d;
typedef struct ring3d Ring3d;
typedef struct face3d Face3d;
struct intnode
        {
        int             a;
        Intnode         *nextintnode;
        };
struct lineseg
        {
        int             num_points;
        int             start_point;
        int             type;
        float           stpt[2], finpt[2];
        Lineseg         *nextlineseg;
        };
struct termpoint
        {
        float point[2];
        Intnode         *linelist;
        Termpoint       *nexttermpoint;
        };
struct geom_edge2d
        {
        float           *v;
        int             type;
        float           meanx, meany;
        Geom_edge2d *nextgeom_edge2d;
        };
```

```
struct vert2d
        {
        float       *point;
        Vert2d      *nextvert2d;
        Intnode     *linelist;
        };
struct edge2d
        {
        int         vert1, vert2;
        int         edgetype;
        Edge2d      *nextedge2d;
        };
struct edge3d
        {
        int         vert1, vert2;
        int         linetype;
        float       *centre;
        Edge3d      *nextedge3d;
        };
struct loop3d
        {
        float       *v;
        Intnode     *linelist;
        Loop3d      *nextloop3d;
        };
struct vert3d
        {
        float       point[4];
        Intnode     *linelist;
        Vert3d      *nextvert3d;
        };
struct face3d
        {
        Ring3d      *ringlist;
        Intnode     *linelist;
        Face3d      *nextface3d;
        };
```

```
struct ring3d
        {
        int         face_number;
        Intnode     *linelist;
        Ring3d      *nextring3d;
        };
```

GLOBAL VARIABLES - SKETCH SOLID

```
/*                          GLOBSKET.H
*/
char        FILENAME[9];
int         NUM_VISPOINTS = 0;
int         NUM_HIDPOINTS;
int         NUM_CENTPOINTS;
int         NUM_CONSPOINTS;
int         NUM_ERASPOINTS;
float       **VISPOINTS;
float       **HIDPOINTS;
float       **CENTPOINTS;
float       **CONSPOINTS;
float       **ERASPOINTS;
float       THREEX, THREEY;
/*------- FILE xxxx.one VARIABLES ----------- */
Lineseg     *VISLINESEG;
Lineseg     *HIDLINESEG;
Lineseg     *CENLINESEG;
Lineseg     *CONSLINESEG;
Lineseg     *ERASLINESEG;
Geom_edge2d         *VISGEOM;
Geom_edge2d         *HIDGEOM;
Geom_edge2d         *CENGEOM;
Geom_edge2d         *CONSGEOM;
Geom_edge2d         *ERASGEOM;
Edge2d      *VISEDGE2D;
Edge2d      *HIDEDGE2D;
Edge2d      *CENEDGE2D;
Edge2d      *CONSEDGE2D;
Edge2d      *ERASEDGE2D;
Termpoint   *VISTERMPOINT;
Termpoint   *HIDTERMPOINT;
Termpoint   *CENTERMPOINT;
Termpoint   *CONSTERMPOINT;
Termpoint   *ERASTERMPOINT;
Vert2d      *VISVERT2D;
Vert2d      *HIDVERT2D;
```

```
Vert2d          *CENVERT2D;
Vert2d          *CONSVERT2D;
Vert2d          *ERASVERT2D;
int             NUMVISEDGE;
int             NUMHIDEDGE;
int             NUMCENEDGE;
/*-------------------------------*/
/*
                        FILE xxxx.two  VARIABLES
*/
Geom_edge2d             *SOLVISGEOM;
Geom_edge2d             *SOLCONSGEOM;
Geom_edge2d             *SOLHIDGEOM;
Geom_edge2d             *SOLCENGEOM;
Geom_edge2d             *SOLGEOM;
Edge2d          *SOLVISEDGE2D;
Edge2d          *SOLCONSEDGE2D;
Edge2d          *SOLHIDEDGE2D;
Edge2d          *SOLCENEDGE2D;
Edge2d          *SOLEDG2D;
Vert2d          *SOLVISVERT2D;
Vert2d          *SOLHIDVERT2D;
Vert2d          *SOLCONSVERT2D;
Vert2d          *SOLCENVERT2D;
Vert2d          *SOLVERT2D;
int             NUMSOLVERTS;
int             NUMSOLEDGES;
int             NUMVISEDGE;
int             NUMVISVERT;
int             NUMCONSEDGES;
int             NUMCONSVERTS;
/*-------------------------------*/
/*
                        FILE xxxx.thr VARIABLES
*/
Loop3d          *LOOPLIST;
Face3d          *FACELIST;
Edge3d          *EDGELIST;
Ring3d          *RINGLIST;
Vert3d          *VERTLIST;
int             NUM_LOOP;
```

```
int         NUM_FACE;
int         NUM_EDGE;
int         NUM_VERT;
int         NUM_RING;
int         NUM_HOLES;
/*-------------------------------*/
/*
            VARIABLES FOR PROC3D.C
*/
int         *PROCEDGES;
int         *PROCVERTS;
int         *PROCCONSEDGE;
int         *PROCCONSVERT;
float       REFPOINT[4];
/*-------------------------------*/
/*
            VARIABLES FOR PROC2D.C
*/
float       MINDIST = 100.0;
float       MEANX, MEANY;
float       STRAIGHTTOLDEG = 15.0;
float       STRAIGHTSLOPE;
int         SAMPLE;
int         CONTINUITY;
int         TOLDIST = 20;
int         LINE_TYPE;
/*-------------------------------*/
```

MEMORY ALLOCATION TO VECTORS AND MATRICES

(From Press et al. [86])

```
/*-------------------------------*/
/*
Allocates a float vector with range 'nl' to 'nh'
*/
float *vector(nl,nh)
int nl,nh;
{
float *v;
int success = 0;
while (success = = 0)
{
    v = (float *) malloc((unsigned)(nh-nl + 1)*sizeof(float));
    success = 1;
    if (!v) success = 0;
}
return v-nl;
}
/*-------------------------------*/
/*
Allocates an int vector with range 'nl' to 'nh'
*/
int *ivector(nl,nh)
int nl, nh;
{
int *v;
int success = 0;
while (success = = 0)
{
    v = (int *) malloc((unsigned)(nh-nl + 1)*sizeof(int));
    success = 1;
    if (!v) success = 0;
}
return v-nl;
}
/*-----------------------------*/
/*
```

```
Allocates a double vector with range 'nl' to 'nh'
*/
double *dvector(nl,nh)
int nl,nh;
{
double *v;
int success = 0;
while (success = = 0)
{
    v = (double *) malloc((unsigned)(nh-nl + 1)*sizeof(int));
    success = 1;
    if(!v) success = 0;
}
return v-nl;
}
/*-------------------------------*/
/*
Allocates a float matrix with range 'nrl' to 'nrh' rows and 'ncl'
to 'nch' columns
*/
float **fmatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
int i;
float **m;
int success = 0;
/*  Allocating pointers to rows              */
while(success = = 0)
{
    m = (float **) malloc((unsigned) (nrh-nrl + 1)*sizeof(float *));
    success = 1;
    if (!m) success = 0;
}
m -= nrl;
/*  Allocating rows and set pointers to them     */
for (i = nrl; i < = nrh; i + +)
{
    m[i] = (float *) malloc ((unsigned) (nch-ncl + 1)*sizeof(float));
    m[i] -= ncl;
}
return m;
```

```
}
/*-------------------------------*/
/*
Allocates an int matrix with range 'nrl' to 'nrh' rows and 'ncl'
to 'nch' columns
*/
int **imatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
int i;
int **m;
int success = 0;
/*   Allocating pointers to rows              */
while(success = = 0)
{
     m = (int **) malloc((unsigned) (nrh-nrl + 1)*sizeof(int *));
     success = 1;
     if (!m) success = 0;
}
m - = nrl;
/*   Allocating rows and set pointers to them     */
for (i = nrl; i < = nrh; i + +)
{
     m[i] = (int *) malloc ((unsigned) (nch-ncl + 1)*sizeof(int));
     m[i] - = ncl;
}
return m;
}
/*-------------------------------*/
/*
Allocates a double matrix with range 'nrl' to 'nrh' rows and 'ncl'
to 'nch' columns
*/
double **dmatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
int i;
double **m;
int success = 0;
/*   Allocating pointers to rows              */
while(success = = 0)
```

```
{
    m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double *));
    success = 1;
    if (!m) success = 0;
}
m -= nrl;
/*  Allocating rows and set pointers to them     */
for (i=nrl; i<=nrh; i++)
{
    m[i]=(double *) malloc ((unsigned) (nch-ncl+1)*sizeof(double));
    m[i] -= ncl;
}
return m;
}
/*-----------------------------*/
/*
Frees a float vector allocated by vector()
*/
void free_vector(v,nl,nh)
float *v;
int nl, nh;
{
free ((char *) (v+nl));
}
/*-----------------------------*/
/*
Frees an int vector allocated by ivector()
*/
void free_ivector(v,nl,nh)
int *v;
int nl,nh;
{
free ((char *) (v+nl));
}
/*-----------------------------*/
/*
Frees a double vector allocated by dvector()
*/
void free_dvector(v,nl,nh)
int *v;
int nl,nh;
```

```
{
free ((char *) (v+nl));
}
/*-----------------------------*/
/*
Frees a float matrix allocated with matrix
*/
void free_matrix(m,nrl,nrh,ncl,nch)
float **m;
int nrl,nrh,ncl,nch;
{
int i;
for (i=nrh; i=nrl; i--)
{
     free ((char *) (m[i]+ncl));
}
free((char *) (m+nrl));
}
/*-----------------------------*/
/*
Frees an int matrix allocated with imatrix
*/
void free_imatrix(m,nrl,nrh,ncl,nch)
int **m;
int nrl,nrh,ncl,nch;
{
int i;
for(i=nrh; i=nrl; i--)
{
     free ((char *) (m[i]+ncl));
}
free((char *)(m+nrl));
}
/*-----------------------------*/
/*
Frees a double matrix allocated with dmatrix
*/
void free_dmatrix(m,nrl,nrh,ncl,nch)
double **m;
int nrl,nrh,ncl,nch;
{
```

```
int i;
for(i = nrh; i = nrl; i--)
{
    free ((char *) (m[i] + ncl));
}
free ((char *) (m + nrl));
}
/*------------------------------*/
```

SCREEN MANAGEMENT FUNCTIONS

```
/* DOSSCRN.C function definitions*/
void setblue();
void setgreen();
void setwhite();
void setred();
void setorange();
void setcyan();
void putcursor (int row, int col);
void clears();
void clearline();
/*-------------------------------*/
void setblue()
    {
    printf("\033[44m");
    printf("\033[37m");
    printf("\033[2J");
    }
/*-------------------------------*/
void setgreen()
    {
     printf("\033[42m");
     printf("\033[31m");
     printf("\033[2J");
    }
/*-------------------------------*/
void setred()
     {
     printf("\033[41m");
     printf("\033[37m");
     printf("\033[2J");
     }
/*-------------------------------*/
```

```
void setwhite()
      {
      printf("\033[47m");
      printf("\033[30m");
      printf("\033[2J");
      }
/*-------------------------------*/
void setorange()
      {
      printf("\033[43m");
      printf("\033[37m");
      printf("\033[2J");
      }
/*-------------------------------*/
void setcyan()
      {
      printf("\033[46m");
      printf("\033[31m");
      printf("\33[2J");
      }
/*-------------------------------*/
void putcursor(int row,int col)
     {
     printf("%c[%d;%dH",27,row,col);
     }
/*-------------------------------*/
void clears()
    {
    printf("\033[2J");
    }
/*-------------------------------*/
void clearline()
      {
      printf("\033[K");
      }
/*-------------------------------*/
```

MEMORY ALLOCATION FUNCTIONS

```
/*                              allocate.c
*/
/*-------------------------------*/
Lineseg *get_lineseg()
{
Lineseg *item;
item = (Lineseg *) malloc(sizeof(Lineseg));
if(item != NULL)
                {
                item->nextlineseg = NULL;
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
return(item);
}
/*-------------------------------*/
Lineseg *back_of_lineseg(Lineseg *new, Lineseg *linesegs)
{
if(linesegs == NULL)
                {
                linesegs = new;
                return(linesegs);
                }
                else
                {
                linesegs->nextlineseg=back_of_lineseg(new, linesegs->nextlineseg);
                return(linesegs);
                }
}
```

```
/*-----------------------------------*/
Lineseg *kill_lineseg(Lineseg *linesegment)
{
char *buffer;
buffer = (char *) linesegment;
free(buffer);
linesegment = NULL;
return(linesegment);
}
/*-----------------------------------*/
Termpoint *get_termpoint()
{
Termpoint *item;
item = (Termpoint *) malloc(sizeof(Termpoint));
if (item != NULL)
        {
        item->nexttermpoint=NULL;
        return(item);
        }
else
        {
        printf("No memory to allocate");
        return(0);
        }
}
/*-----------------------------------*/
Termpoint *back_of_termpoint(Termpoint *new, Termpoint *termpoints)
{
if (termpoints == NULL)
        {
        termpoints=new;
        return(termpoints);
        }
else
        {
        termpoints->nexttermpoint =
                back_of_termpoint(new,termpoints->nexttermpoint);
        return(termpoints);
        }
}
/*-----------------------------------*/
```

```
Termpoint *kill_termpoint(Termpoint *termnode)
{
char *buffer;
buffer = (char *) termnode->linelist;
free(buffer);
buffer = (char *) termnode;
free (buffer);
termnode = NULL;
return(termnode);
}
/*-------------------------------*/
void show_termpoints(Termpoint *termpoints)
{
Termpoint *termpoint1;
Intnode *linelist1;
termpoint1 = termpoints;
termpoints = termpoints->nexttermpoint;
while (termpoints != NULL)
        {
        printf("The point is  %3.2f%s%3.2f\n",
                termpoints->point[0], "   ", termpoints->point[1]);
        linelist1 = termpoints->linelist;
        linelist1 = linelist1->nextintnode;
        printf("The lines emanate are \n");
        while (linelist1 != NULL)
                {
                printf("%4d",linelist1-a);
                linelist1 = linelist1->nextintnode;
                }
        printf("\n");
        termpoints = termpoints->nexttermpoint;
        }
}
/*-------------------------------*/
```

```
Intnode *get_intnode()
{
Intnode *item;
item = (Intnode *) malloc(sizeof(Intnode));
if (item != NULL)
                {
                item->nextintnode = NULL;
                return(item);
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
}
/*-------------------------------*/
Intnode *back_of_intnode(Intnode *new, Intnode *intlist)
{
if (intlist == NULL)
                {
                intlist = new;
                return(intlist);
                }
else
                {
                intlist->nextintnode =
                 back_of_intnode(new, intlist->nextintnode);
                return(intlist);
                }
}
/*-----------------------------------*/
Intnode *kill_intnode(Intnode *intnodetokill)
{
char *buffer;
buffer = (char *) intnodetokill;
free(buffer);
intnodetokill = NULL;
return(intnodetokill);
}
/*-----------------------------------*/
```

```
Geom_edge2d *get_geom_edge2d()
{
Geom_edge2d *item;
item=(Geom_edge2d *) malloc(sizeof(Geom_edge2d));
if (item != NULL)
                {
                item->nextgeom_edge2d = NULL;
                return(item);
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
}
/*-----------------------------------*/
Geom_edge2d *back_of_geom2d(Geom_edge2d *newgeom, Geom_edge2d
*geomlist)
{
if (geomlist == NULL)
                {
                geomlist=newgeom;
                return(geomlist);
                }
else
                {
                geomlist->nextgeom_edge2d =
                  back_of_geom2d(newgeom, geomlist->nextgeom_edge2d);
                return(geomlist);
                }
}
/*-----------------------------------*/
Geom_edge2d *kill_geomedge2d(Geom_edge2d *geomnode)
{
char *buffer;
buffer = (char *) geomnode->v;
free(buffer);
buffer = (char *) geomnode;
free(buffer);
geomnode = NULL;
return(geomnode);
```

```
}
/*-------------------------------*/
show_geom_edge2d(Geom_edge2d *geomlist)
{
int i, j;
Geom_edge2d *geomlist1;
geomlist1 = geomlist;
geomlist = geomlist->nextgeom_edge2d;
while (geomlist != NULL)
            {
            j=3;
            if (geomlist->type == 3) j = 6;
                        {
                        for (i=0; i; i++) printf ("%2.4f%s",
                                    geomlist->v[i], "    ");
                        printf("\n");
                        }
            geomlist = geomlist->nextgeom_edge2d;
            }
geomlist = geomlist1;
}
/*----------------------------------*/
Vert2d *get_vert2d()
{
Vert2d *item;
item = (Vert2d *) malloc(sizeof(Vert2d));
if (item != NULL)
            {
            item->nextvert2d = NULL;
            return(item);
            }
else
    {
    printf("No memory to allocate");
    return(0);
    }
}
/*-----------------------------------*/
```

```
Vert2d *back_of_vert2d(Vert2d *newvert2d, Vert2d *vert2dlist)
{
if (vert2dlist = = NULL)
                {
                vert2dlist = newvert2d;
                return(vert2dlist);
     }
else
     {
     vert2dlist->nextvert2d =
                back_of_vert2d(newvert2d, vert2dlist->nextvert2d);
     return(vert2dlist);
     }
}
/*-----------------------------------*/
Vert2d *kill_vert2d(Vert2d *vertnode)
{
char *buffer;
buffer = (char *) vertnode->point;
free(buffer);
buffer = (char *) vertnode->linelist;
free(buffer);
buffer = (char *) vertnode;
free(buffer);
vertnode = NULL;
return(vertnode);
}
/*-----------------------------------*/
void show_vert2d(Vert2d *vertlist)
{
Vert2d *vertlist1;
Intnode *intlist1;
vertlist1 = vertlist;
vertlist = vertlist->nextvert2d;
while (vertlist != NULL)
                {
     printf("The point is %4.2f%s%4.2f\n", vertlist->point[0],
                "    ", vertlist->point[1]);
                intlist1 = vertlist->linelist;
                intlist1 = intlist1->nextintnode;
                printf("The lines emanating are \n");
```

```
    while (intlist1 ! = NULL)
                        {
                        printf("%4d", intlist1- > a);
                        intlist1 = intlist1- > nextintnode;
                        }
            printf("\n");
    vertlist = vertlist- > nextvert2d;
            }
vertlist = vertlist1;
}
/*-----------------------------------*/
Edge2d *get_edge2d()
{
Edge2d *item;
item = (Edge2d *) malloc(sizeof(Edge2d));
if (item ! = NULL)
            {
            item- > nextedge2d = NULL;
            return(item);
            }
else
            {
            printf("No memory to allocate");
            return(0);
            }
}
/*-----------------------------------*/
Edge2d *back_of_edge2d(Edge2d *newedge, Edge2d *edge2dlist)
{
if (edge2dlist = = NULL)
            {
            edge2dlist = newedge;
            return(edge2dlist);
            }
else
            {
            edge2dlist- > nextedge2d =
                        back_of_edge2d(newedge, edge2dlist- > nextedge2d);
            return(edge2dlist);
            }
}
```

```
/*-----------------------------------*/
Edge2d *kill_edge2d(Edge2d *edgenode)
{
char *buffer;
buffer = (char *) edgenode;
free(buffer);
edgenode = NULL;
return(edgenode);
}
/*-----------------------------------*/
void show_edge2d(Edge2d *edgelist)
{
Edge2d *edgelist1;
edgelist1 = edgelist;
edgelist1 = edgelist1->nextedge2d;
while (edgelist1 != NULL)
        {
        printf("The end vertices of this edge are\n");
        printf("%4d%4d\n", edgelist1->vert1, edgelist1->vert2);
        printf("The edge type is     %d\n", edgelist1->edgetype);
        edgelist1 = edgelist1->nextedge2d;
        }
}
/*-----------------------------------*/
Vert3d *get_vert3d()
{
Vert3d *item;
item = ((Vert3d *) malloc (sizeof (Vert3d)));
if (item != NULL) return (item);
else
        {
        printf("No memory to allocate\n");
        return(0);
        }
}
/*-----------------------------------*/
```

```
Vert3d *back_of_vert3d(Vert3d *newvert, Vert3d *vertlist)
{
if (vertlist = = NULL)
                {
                vertlist = newvert;
                return(vertlist);
                }
else
                {
                vertlist->nextvert3d = back_of_vert3d(newvert,
                                                vertlist->nextvert3d);
                return(vertlist);
                }
}
/*-----------------------------------*/
Vert3d *kill_vert3d(Vert3d *vertnode)
{
char *buffer;
buffer = (char *) vertnode->linelist;
free(buffer);
buffer = (char *) vertnode;
free(buffer);
vertnode = NULL;
return(vertnode);
}
/*-----------------------------------*/
Loop3d *get_loop3d()
{
Loop3d *item;
item = ((Loop3d *) malloc (sizeof(Loop3d)));
if (item != NULL)
                {
                item->nextloop3d = NULL;
                return(item);
                }
else
                {
                printf("No memory to allocate\n");
                return(0);
                }
```

```
}
/*-----------------------------------*/
Loop3d *back_of_loop3d(Loop3d *newloop, Loop3d *looplist)
{
if (looplist = = NULL)
                {
                looplist = newloop;
                return(looplist);
                }
else
                {
                looplist->nextloop3d = back_of_loop3d(newloop,
                                                        looplist->nextloop3d);
                return(looplist);
                }
}
/*-----------------------------------*/
Face3d *get_face3d()
{
Face3d *item;
item = (Face3d *) malloc(sizeof(Face3d));
if(item != NULL)
                {
                item->nextface3d = NULL;
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
return(item);
}
/*-----------------------------*/
Face3d *back_of_face3d(Face3d *new, Face3d *facelist)
{
if(facelist = = NULL)
                {
                facelist = new;
                return(facelist);
                }
                else
```

```
                {
                facelist->nextface3d = back_of_face3d(new, facelist->nextface3d);
                return(facelist);
                }
}
/*-----------------------------------*/
Face3d *kill_face3d(Face3d *facelist)
{
char *buffer;
buffer = (char *) facelist->linelist;
free(buffer);
buffer = (char *) facelist;
free(buffer);
facelist = NULL;
return(facelist);
}
/*-----------------------------------*/
Ring3d *get_ring3d()
{
Ring3d *item;
item = (Ring3d *) malloc(sizeof(Ring3d));
if (item != NULL)
                {
                item->nextring3d=NULL;
                return(item);
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
}
/*-----------------------------------*/
Ring3d *back_of_ring3d(Ring3d *new, Ring3d *ringlist)
{
if (ringlist == NULL)
                {
                ringlist=new;
                return(ringlist);
                }
else
```

```
                {
                ringlist->nextring3d =
                                back_of_ring3d(new, ringlist->nextring3d);
                return(ringlist);
                }
}
/*-------------------------------*/
Ring3d *kill_ringnode(Ring3d *ringnode)
{
char *buffer;
buffer = (char *) ringnode->linelist;
free(buffer);
buffer = (char *) ringnode;
free (buffer);
ringnode = NULL;
return(ringnode);
}
/*-------------------------------*/
Edge3d *get_edge3d()
{
Edge3d *item;
item = (Edge3d *) malloc(sizeof(Edge3d));
if (item != NULL)
                {
                item->nextedge3d = NULL;
                return(item);
                }
else
                {
                printf("No memory to allocate");
                return(0);
                }
}
/*-----------------------------------*/
Edge3d *back_of_edge3d(Edge3d *newedge, Edge3d *edge3dlist)
{
if (edge3dlist == NULL)
                {
                edge3dlist = newedge;
                return(edge3dlist);
                }
```

```
else
            {
            edge3dlist->nextedge3d =
                        back_of_edge3d(newedge, edge3dlist->nextedge3d);
            return(edge3dlist);
            }
}
/*-----------------------------------*/
Edge3d *kill_edge3d(Edge3d *edgenode)
{
char *buffer;
buffer = (char *) edgenode;
free(buffer);
edgenode = NULL;
return(edgenode);
}
/*-----------------------------------*/
void show_edge3d(Edge3d *edgelist)
{
Edge3d *edgelist1;
edgelist1 = edgelist;
edgelist1 = edgelist1->nextedge3d;
while (edgelist1 != NULL)
            {
            printf("The end vertices of this edge are\n");
            printf("%4d%4d\n", edgelist1->vert1, edgelist1->vert2);
            printf("The edge type is     %d\n", edgelist1->linetype);
            edgelist1 = edgelist1->nextedge3d;
            }
}
/*-----------------------------------*/
```

FILE HANDLING FUNCTIONS

```
/*                              filehand.c
/*--------------------------------*/
void   copy_string (string1, string2)
     char *string1;
     char *string2;
              {
          while (*string2++ = *string1++)
              ;
              }
/*--------------------------------*/
/*              append_string (string1, string2)
*
*     This function appends the contents of string1 to string2.
*/
void   append_string (string1, string2)
     char *string1;
     char *string2;
     {
          while (*string2)
          string2++;
          while (*string2++ = *string1++)
          ;
     }
/*--------------------------------*/
float *read_coord(char line[])
{
int i, j;
char c, xcoord[6], ycoord[6];
float v[2];
i=0;
c=line[i];
while (c != ',')
              {
              xcoord[i]=c;
              i++;
              c=line[i];
              }
xcoord[i]='\0';
```

```
i++;
j=0;
c=line[i];
while (c != '\0')
                {
                ycoord[j]=c;
                i++;
                j++;
                c=line[i];
                }
ycoord[j]='\0';
v[0]=atof(xcoord);
v[1]=atof(ycoord);
return(v);
}
/*-------------------------------*/
void read_point_file()
{
int i, j;
char c, line[12], line1[6], count, xcoord[6], ycoord[6];
char filename[13];
FILE *notes;
float *v;
v=vector(0,1);
copy_string(FILENAME, filename);
append_string(".pnt", filename);
notes = fopen(filename, "r");
fgets(line, 12, notes);
v = read_coord(line);
THREEX = v[0];
THREEY = v[1];
fgets (line, 12, notes);
NUM_VISPOINTS = atoi(line);
fgets(line, 12, notes);
NUM_HIDPOINTS = atoi(line);
fgets(line, 12, notes);
NUM_CENTPOINTS = atoi(line);
fgets(line, 12, notes);
NUM_CONSPOINTS = atoi(line);
fgets(line, 12, notes);
NUM_ERASPOINTS = atoi(line);
```

```
VISPOINTS = fmatrix(0,(NUM_VISPOINTS-1), 0, 1);
for (i=0; i<NUM_VISPOINTS; i++)
        {
        fgets(line, 12, notes);
        v=read_coord(line);
        VISPOINTS[i][0]=v[0];
        VISPOINTS[i][1]=v[1];
        }
HIDPOINTS = fmatrix(0,(NUM_HIDPOINTS-1), 0, 1);
for (i=0; i<NUM_HIDPOINTS; i++)
        {
        fgets(line, 12, notes);
        v=read_coord(line);
        HIDPOINTS[i][0] = v[0];
        HIDPOINTS[i][1] = v[1];
        }
CENTPOINTS = fmatrix(0,(NUM_CENTPOINTS-1),0,1);
for (i=0; i<NUM_CENTPOINTS; i++)
        {
        fgets(line, 12, notes);
        v=read_coord(line);
        CENTPOINTS[i][0] = v[0];
        CENTPOINTS[i][1] = v[1];
        }
CONSPOINTS = fmatrix(0, (NUM_CONSPOINTS-1), 0, 1);
for (i=0; i<NUM_CONSPOINTS; i++)
        fgets(line, 12, notes);
        v=read_coord(line);
        CONSPOINTS[i][0] = v[0];
        CONSPOINTS[i][1] = v[1];
        }
ERASPOINTS = fmatrix(0, (NUM_ERASPOINTS-1), 0, 1);
for (i=0; i<NUM_ERASPOINTS; i++)
        {
        fgets(line, 12, notes);
        v=read_coord(line);
        ERASPOINTS[i][0] = v[0];
        ERASPOINTS[i][1] = v[1];
        }
}
/*-------------------------------*/
```

```
void write_file_one1(Lineseg *linesegs, Geom_edge2d *geomlist,
                     Edge2d *edgelist, Termpoint *termpoints,
                     Vert2d *vertlist, FILE *notes)
{
Lineseg *lineseg1;
Geom_edge2d *geomlist1;
Intnode *linelist1;
Edge2d *edgelist1;
Termpoint *termpoints1;
Vert2d *vertlist1;
char filename[13];
int i=0,j=0;
char str = ' ';
lineseg1=linesegs->nextlineseg;
vertlist1=vertlist->nextvert2d;
while (lineseg1 != NULL)
        {
        i++;
        lineseg1 = lineseg1->nextlineseg;
        }
while (vertlist1 != NULL)
        {
        j++;
        vertlist1 = vertlist1->nextvert2d;
        }
fprintf(notes,"%d\n",i);
fprintf(notes,"%d\n",j);
lineseg1=linesegs->nextlineseg;
vertlist1=vertlist->nextvert2d;
while (lineseg1 != NULL)
        {
        fprintf(notes,"%d%s%d%s%d%s%4.2f%s%4.2f%s%4.2f%s%4.2f\n",
                lineseg1->num_points, str,
                lineseg1->start_point, str,
                lineseg1->type, str,
                lineseg1->stpt[0], str, lineseg1->stpt[1],str,
                lineseg1->finpt[0], str, lineseg1->finpt[1]);
        lineseg1=lineseg1->nextlineseg;
        }
lineseg1=linesegs->nextlineseg;
while (geomlist1 != NULL)
```

```
                {
                if (lineseg1->type == 1)
                        {
                        fprintf(notes, "%4.2f%s%4.2f%s%4.2f%s%4.2f%s%4,2f\n",
                                geomlist1->v[0], str,
                                geomlist1->v[1], str,
                                geomlist1->v[2], str,
                                geomlist1->meanx, str,
                                geomlist1->meany);
                        lineseg1 = lineseg1->nextlineseg;
                        geomlist1 = geomlist1->nextgeom_edge2d;
                        }
                else
                        {
                        fprintf(notes, "%4.2f%s%4.2f%s%f%s%f%s%f%s
                                %f%s%f%s%f\n",
                        geomlist1->v[0],str,
                        geomlist1->v[1],str,
                        geomlist1->v[2],str,
                        geomlist1->v[3],str,
                        geomlist1->v[4],str,
                        geomlist1->v[5],str,
                        geomlist1->meanx,str,
                        geomlist1->meany);
                        lineseg1 = lineseg1->nextlineseg;
                        geomlist1 = geomlist1->nextgeom_edge2d;
                        }
                }
edgelist1 = edgelist->nextedge2d;
while (edgelist1 != NULL)
                {
                fprintf(notes,"%4.2f%s%4.2f%s%4.2f\n",
                        edgelist1->vert1, str,
                        edgelist1->vert2, str,
                        edgelist1->edgetype);
                edgelist1 = edgelist1->nextedge2d;
                }
termpoints1 = termpoints->nexttermpoint;
while (termpoints1 != NULL)
                {
                fprintf(notes,"%4.2f%s%4.2f%s", termpoints1->point[0], str,
```

```
                    termpoints1->point[1], str);
            linelist1 = termpoints1->linelist->nextintnode;
            while (linelist1 != NULL)
                    {
                    fprintf(notes,"%d%s",linelist1->a, str);
                    linelist1 = linelist1->nextintnode;
                    }
            fprintf(notes,"\n");
            termpoints1 = termpoints1->nexttermpoint;
            }
vertlist1 = vertlist->nextvert2d;
while (vertlist1 != NULL)
            {
            fprintf(notes,"%4.2f%s%4.2f%s", vertlist1->point[0],
                        vertlist1->point[1]);
            linelist1 = vertlist1->linelist->nextintnode;
            while (linelist1 != NULL)
                    {
                    fprintf(notes,"%d%s", linelist1->a, str);
                    linelist1 = linelist1->nextintnode;
                    }
            fprintf(notes,"\n");
            vertlist1 = vertlist1->nextvert2d;
            }
}
/*--------------------------------*/
void write_file_one()
{
FILE *notes;
char filename[13];
char str = ' ';
copy_string(FILENAME, filename);
append_string(".one",filename);
notes = fopen(filename,"w");
fprintf(notes,"%d%s%d\n", THREEX, str, THREEY);
fprintf(notes,"%d\n",NUM_VISPOINTS);
fprintf(notes,"%d\n",NUM_HIDPOINTS);
fprintf(notes,"%d\n",NUM_CENTPOINTS);
fprintf(notes,"%d\n",NUM_CONSPOINTS);
fprintf(notes,"%d\n",NUM_ERASPOINTS);
write_file_one1(VISLINESEG, VISGEOM, VISEDGE2D, VISTERMPOINT,
```

```
                                        VISVERT2D,notes);
if (NUM_HIDPOINTS  0)
write_file_one1(HIDLINESEG, HIDGEOM, HIDEDGE2D, HIDTERMPOINT,
                                        HIDVERT2D, notes);
if (NUM_CENTPOINTS  0)
write_file_one1(CENLINESEG, CENGEOM, CENEDGE2D,
CENTERMPOINT, CENVERT2D, notes);
if (NUM_CONSPOINTS  0)
write_file_one1(CONSLINESEG, CONSGEOM, CONSEDGE2D, CON-
STERMPOINT, CONSVERT2D, notes);
if (NUM_ERASPOINTS  0)
write_file_one1(ERASLINESEG, ERASGEOM, ERASEDGE2D,
ERASTERMPOINT, ERASVERT2D, notes);
fclose(notes);
}
/*--------------------------------*/
void write_file_two1( Geom_edge2d *geomlist, Edge2d *edgelist,
                        Vert2d *vertlist, FILE *notes)
{
Geom_edge2d *geomlist1;
Intnode *linelist1;
Edge2d *edgelist1;
Vert2d *vertlist1;
int i=0,j=0;
char str = ' ';
edgelist1=edgelist->nextedge2d;
vertlist1=vertlist->nextvert2d;
while (edgelist1 != NULL)
        {
        i++;
        edgelist1 = edgelist1->nextedge2d;
        }
while (vertlist1 != NULL)
        {
        j++;
        vertlist1 = vertlist1->nextvert2d;
        }
fprintf(notes,"%d\n",i);
fprintf(notes,"%d\n",j);
edgelist1=edgelist->nextedge2d;
vertlist1=vertlist->nextvert2d;
```

```
geomlist1 = geomlist->nextgeom_edge2d;
while (edgelist1 != NULL)
        {
        fprintf(notes,"%d%s%d%s%d\n",
                edgelist1->vert1,str,
                edgelist1->vert2,str,
                edgelist1->edgetype);
        edgelist1 = edgelist1->nextedge2d;
        }
edgelist1 = edgelist->nextedge2d;
while (geomlist1 != NULL)
        {
        if (abs(edgelist1->edgetype) 5)
                {
                fprintf(notes, "%f%s%f%s%f%s%f%s%f\n",
                        geomlist1->v[0], str,
                        geomlist1->v[1], str,
                        geomlist1->v[2], str,
                        geomlist1->meanx, str,
                        geomlist1->meany);
                edgelist1 = edgelist1->nextedge2d;
                geomlist1 = geomlist1->nextgeom_edge2d;
                }
        else
                {
                fprintf(notes, "%f%s%f%s%f%s%f%s%f%s%f
                %s%f%s%f\n",
                geomlist1->v[0],str,
                geomlist1->v[1],str,
                geomlist1->v[2],str,
                geomlist1->v[3],str,
                geomlist1->v[4],str,
                geomlist1->v[5],str,
                geomlist1->meanx,str,
                geomlist1->meany);
                edgelist1 = edgelist1->nextedge2d;
                geomlist1 = geomlist1->nextgeom_edge2d;
                }
        }
vertlist1 = vertlist->nextvert2d;
while (vertlist1 != NULL)
```

```
            {
            fprintf(notes,"%f%s%f%s", vertlist1->point[0], str,
                        vertlist1->point[1]);
            linelist1 = vertlist1->linelist->nextintnode;
            while (linelist1 != NULL)
                        {
                        fprintf(notes,"%d%s", linelist1->a, str);
                        linelist1 = linelist1->nextintnode;
                        }
            fprintf(notes,"\n");
            vertlist1 = vertlist1->nextvert2d;
            }
}
/*--------------------------------*/
void write_file_two()
{
FILE *notes;
char filename[13];
char str = ' ';
copy_string(FILENAME, filename);
append_string(".two",filename);
notes = fopen(filename,"w");
fprintf(notes,"%d%s%d\n", THREEX, str, THREEY);
fprintf(notes,"%d\n",NUM_VISPOINTS);
fprintf(notes,"%d\n",NUM_HIDPOINTS);
fprintf(notes,"%d\n",NUM_CENTPOINTS);
fprintf(notes,"%d\n",NUM_CONSPOINTS);
fprintf(notes,"%d\n",NUM_ERASPOINTS);
write_file_two1(SOLVISGEOM, SOLVISEDGE2D, SOLVISVERT2D, notes);
if (NUM_HIDPOINTS 0)
write_file_two1(SOLHIDGEOM, SOLHIDEDGE2D, SOLHIDVERT2D, notes);
if (NUM_CENTPOINTS 0)
write_file_two1(SOLCENGEOM, SOLCENEDGE2D, SOLCEN-
VERT2D,notes);
if (NUM_CONSPOINTS 0)
write_file_two1(SOLCONSGEOM, SOLCONSEDGE2D, SOLCONSVERT2D,
notes);
write_file_two1(SOLGEOM, SOLEDG2D, SOLVERT2D, notes);
fclose(notes);
}
/*--------------------------------*/
```

```
void write_file_thr( Loop3d *looplist, Face3d *facelist,
                     Edge3d *edgelist, Ring3d *ringlist,
                     Vert3d *vertlist)
{
Intnode *linelist1;
Edge3d *edgelist1;
Loop3d *looplist1;
Face3d *facelist1;
Ring3d *ringlist1;
Vert3d *vertlist1;
char filename[13], str = ' ';
int i=0,j=0, k=0, l=0, m=-1;
FILE *notes;
copy_string(FILENAME,filename);
append_string(".thr",filename);
notes = fopen(filename, "w");
edgelist1=edgelist->nextedge3d;
vertlist1=vertlist->nextvert3d;
facelist1 = facelist->nextface3d;
looplist1 = looplist;
ringlist1 = ringlist;
while (edgelist1 != NULL)
        {
        i++;
        edgelist1 = edgelist1->nextedge3d;
        }
while (vertlist1 != NULL)
        {
        j++;
        vertlist1 = vertlist1->nextvert3d;
        }
while (facelist1 != NULL)
        {
        k++;
        facelist1 = facelist1->nextface3d;
        }
while (looplist1 != NULL)
        {
        looplist1 = looplist->nextloop3d;
        l++;
        }
```

```
while (ringlist1 != NULL)
            {
            ringlist1 = ringlist1->nextring3d;
            m++;
            }
if (m == -1) m = 0;
fprintf(notes,"%d\n",i);
fprintf(notes,"%d\n",j);
fprintf(notes, "%d\n",k);
fprintf(notes,"%d\n",l);
fprintf(notes,"%d\n",m);
edgelist1=edgelist->nextedge3d;
vertlist1=vertlist->nextvert3d;
while (edgelist1 != NULL)
            {
            fprintf(notes,"%d%s%d%s%d",
                        edgelist1->vert1,str,
                        edgelist1->vert2,str,
                        edgelist1->linetype);
            if (edgelist1->linetype < 4)
                        {
                        fprintf(notes,"%s%f%s%f%s%f",str,edgelist1->centre[0],
                        edgelist1->centre[1],str, edgelist1->centre[2]);
                        }
            fprintf(notes,"\n");
            edgelist1 = edgelist1->nextedge3d;
            }
edgelist1 = edgelist->nextedge3d;
while (vertlist1 != NULL)
            {
            fprintf(notes,"%f%s%f%s%f%s", vertlist1->point[0], str,
                                    vertlist1->point[1],str,
                                    vertlist1->point[2],str);
            linelist1 = vertlist1->linelist->nextintnode;
            while (linelist1 != NULL)
                        {
                        fprintf(notes,"%d%s", linelist1->a, str);
                        linelist1 = linelist1->nextintnode;
                        }
            fprintf(notes,"\n");
            vertlist1 = vertlist1->nextvert3d;
```

```
                }
while(facelist1 != NULL)
                {
                linelist1 = facelist1->linelist->nextintnode;
    while (linelist1 != NULL)
                        {
                        fprintf(notes, "%d%s", linelist1->a, str);
                        linelist1 = linelist1->nextintnode;
                        }
                fprintf(notes,"\n");
                facelist1 = facelist1->nextface3d;
                }
while(looplist1 != NULL)
                {
                linelist1 = looplist1->linelist->nextintnode;
                while (linelist1 != NULL)
                        {
                        fprintf(notes, "%d%s", linelist1->a, str);
                        linelist1 = linelist1->nextintnode;
                        }
                fprintf(notes,"\n");
                looplist1 = looplist1->nextloop3d;
                }
while(ringlist1 != NULL)
                {
                linelist1 = looplist1->linelist->nextintnode;
                while (linelist1 != NULL)
                        {
                        fprintf(notes, "%d%s", linelist1->a, str);
                        linelist1 = linelist1->nextintnode;
                        }
                fprintf(notes, "%\n");
                looplist1 = looplist1->nextloop3d;
                }
fclose(notes);
}
/*-------------------------------*/
Lineseg *read_lineseg(char line[])
{
Lineseg *new;
char c, str[6];
```

```
int i=0,j,k=0;
new = get_lineseg();
c = line[k];
for (j=0; j; j++)
        {
        while (c != ' ')
                {
                str[i] = c;
                i++;
                c = line[k];
                k++;
                }
        str[i] = '\0';
        if (j == 0) new->num_points = atoi(str);
        if (j == 1) new->start_point = atoi(str);
        if (j == 2) new->type = atoi(str);
        if (j == 3) new->stpt[0] = atof(str);
        if (j == 4) new->stpt[1] = atof(str);
        if (j == 5) new->finpt[0] = atof(str);
        if (j == 6) new->finpt[1] = atof(str);
        i=0;
        }
return(new);
}
/*-------------------------------*/
Geom_edge2d *read_geomedge(char line[], int type)
{
Geom_edge2d *new;
char c, str[6];
int i=0,j,k=0, num_entry;
num_entry = 6;
if(abs(type) 4) num_entry = 9;
new = get_geom_edge2d();
c = line[k];
for (j=0; j<num_entry; j++)
        {
        while (c != ' ')
                {
                str[i] = c;
                i++;
                c = line[k];
```

```
                    k++;
                    }
            str[i] = '\0';
            if(abs(type) 5)
            {
            if (j == 0) new->v[0] = atof(str);
            if (j == 1) new->v[1] = atof(str);
            if (j == 2) new->v[2] = atof(str);
    if (j == 3) new->type = atoi(str);
            if (j == 4) new->meanx = atof(str);
            if (j == 5) new->meany = atof(str);
            i=0;
            }
            else
            {
            if (j == 0) new->v[0] = atof(str);
            if (j == 1) new->v[1] = atof(str);
            if (j == 2) new->v[2] = atof(str);
            if (j == 3) new->v[3] = atof(str);
            if (j == 4) new->v[4] = atof(str);
            if (j == 5) new->v[5] = atof(str);
            if (j == 6) new->type = atoi(str);
            if (j == 7) new->meanx = atof(str);
            if (j == 8) new->meany = atof(str);
  }
            }
return(new);
}
/*-------------------------------*/
Intnode *read_linelist(char line[], int stpt)
{
Intnode *linelist, *new;
int i, j, k;
char str[6], c;
linelist = get_intnode();
i = stpt-1;
k=0;
c = line[i];
while (c != '\0')
            {
            while (c != ' ')
```

```
                        {
                        str[k] = c;
                        k++;
                        i++;
                        c = line[i];
                        }
            str[i] ='\0';
            new = get_intnode();
            new->a = atoi(str);
            linelist = back_of_intnode(new, linelist);
            i++;
            c = line[i];
            }
return(linelist);
}
/*-------------------------------*/
Termpoint *read_termpoint(char line[])
{
Termpoint *new;
Intnode *linelist;
int i=0,j,k=0;
char c, str[6];
new = get_termpoint();
c = line[k];
for(j=0; j; j++)
            {
            while (c != ' ')
                        {
                        str[i] = c;
                        i++;
                        k++;
                        c = line[k];
                        }
            str[i] = '\0';
            if (j == 0) new->point[0] = atof(str);
            if (j == 1) new->point[1] = atof(str);
            i = 0;
            k++;
            }
linelist = read_linelist(line, k-1);
new->linelist = linelist;
```

```
return(new);
}
/*-------------------------------*/
Edge2d *read_edge2d(char line[])
{
Edge2d *new;
char c, str[6];
int i=0, j, k=0;
for (j=0; j; j++)
        {
        while(c != ' ')
                {
                str[i] = c;
                i++;
                k++;
                c = line[k];
                }
        str[i] = '\0';
        if (j == 0) new->vert1 = atoi(str);
        if (j == 1) new->vert2 = atoi(str);
        if (j == 2) new->edgetype = atoi(str);
        i = 0;
        k++;
        }
return(new);
}
/*-------------------------------*/
Vert2d *read_vertex2d(char line[])
{
Vert2d *new;
Intnode *linelist;
int i=0,j,k=0;
char c, str[6];
new = get_vert2d();
c = line[k];
for(j=0; j; j++)
        {
        while (c != ' ')
                {
                str[i] = c;
                i++;
```

```
                    k++;
                    c = line[k];
                    }
            str[i] = '\0';
            if (j == 0) new->point[0] = atof(str);
            if (j == 1) new->point[1] = atof(str);
            i = 0;
            k++;
            }
linelist = read_linelist(line, k-1);
new->linelist = linelist;
return(new);
}
/*-------------------------------*/
void initialise_nodes()
{
VISLINESEG = get_lineseg();
HIDLINESEG = get_lineseg();
CENLINESEG = get_lineseg();
CONSLINESEG = get_lineseg();
ERASLINESEG = get_lineseg();
VISGEOM = get_geom_edge2d();
HIDGEOM = get_geom_edge2d();
CENGEOM = get_geom_edge2d();
CONSGEOM = get_geom_edge2d();
ERASGEOM = get_geom_edge2d();
VISEDGE2D = get_edge2d();
HIDEDGE2D = get_edge2d();
CENEDGE2D = get_edge2d();
CONSEDGE2D = get_edge2d();
ERASEDGE2D = get_edge2d();
VISTERMPOINT = get_termpoint();
HIDTERMPOINT = get_termpoint();
CENTERMPOINT = get_termpoint();
CONSTERMPOINT = get_termpoint();
ERASTERMPOINT = get_termpoint();
VISVERT2D = get_vert2d();
HIDVERT2D = get_vert2d();
CENVERT2D = get_vert2d();
CONSVERT2D = get_vert2d();
ERASVERT2D = get_vert2d();
```

```
}
/*-------------------------------*/
void read_file_one()
{
int i, j, type;
char c, line[63], line1[6], count, xcoord[6], ycoord[6];
char filename[13];
int numedge, numvert;
Geom_edge2d *newgeom;
Lineseg *newlineseg, *lineseg1;
Vert2d *newvert;
Edge2d *newedge;
Termpoint *newtermpoint;
FILE *notes;
float *v;
initialise_nodes();
v=vector(0,1);
copy_string(FILENAME, filename);
append_string(".one", filename);
notes = fopen(filename, "r");
fgets(line, 63, notes);
v = read_coord(line);
THREEX = v[0];
THREEY = v[1];
fgets (line, 63, notes);
NUM_VISPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_HIDPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_CENTPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_CONSPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_ERASPOINTS = atoi(line);
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
```

```
                newlineseg = read_lineseg(line);
                VISLINESEG = back_of_lineseg(newlineseg, VISLINESEG);
                }
lineseg1 = VISLINESEG->nextlineseg;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = lineseg1->type;
                if(type != 1) type = 5;
                lineseg1 = lineseg1->nextlineseg;
                newgeom = read_geomedge(line, type);
                VISGEOM = back_of_geom2d(newgeom, VISGEOM);
                }
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                VISEDGE2D = back_of_edge2d(newedge, VISEDGE2D);
                }
for (i=0; i<numverts; i++)
                {
                fgets(line, 63, notes);
                newtermpoint = read_termpoint(line);
                VISTERMPOINT = back_of_termpoint(newtermpoint,
                VISTERMPOINT);
                }
for (i=0; i<numverts; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                VISVERT2D = back_of_vert2d(newvert, VISVERT2D);
                }
if (NUM_HIDPOINTS > 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
```

```
                newlineseg = read_lineseg(line);
                HIDLINESEG = back_of_lineseg(newlineseg, HIDLINESEG);
                }
lineseg1 = HIDLINESEG->nextlineseg;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = lineseg1->type;
                if(type != 1) type = 5;
                lineseg1 = lineseg1->nextlineseg;
                newgeom = read_geomedge(line, type);
                HIDGEOM = back_of_geom2d(newgeom, HIDGEOM);
                }
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                HIDEDGE2D = back_of_edge2d(newedge, HIDEDGE2D);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newtermpoint = read_termpoint(line);
                HIDTERMPOINT = back_of_termpoint(newtermpoint,
                HIDTERMPOINT);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                HIDVERT2D = back_of_vert2d(newvert, HIDVERT2D);
                }
                }
if(NUM_CENTPOINTS > 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
                {
```

```
                fgets(line, 63, notes);
                newlineseg = read_lineseg(line);
                CENLINESEG = back_of_lineseg(newlineseg, CENLINESEG);
                }
lineseg1 = CENLINESEG->nextlineseg;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = lineseg1->type;
                if(type != 1) type = 5;
                lineseg1 = lineseg1->nextlineseg;
                newgeom = read_geomedge(line, type);
                CENGEOM = back_of_geom2d(newgeom, CENGEOM);
                }
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                CENEDGE2D = back_of_edge2d(newedge, CENEDGE2D);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newtermpoint = read_termpoint(line);
                CENTERMPOINT = back_of_termpoint(newtermpoint, CENTERMPOIN
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                CENVERT2D = back_of_vert2d(newvert, CENVERT2D);
                }
  }
if (NUM_CONSPOINTS 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
                {
```

```
                fgets(line, 63, notes);
                newlineseg = read_lineseg(line);
                CONSLINESEG = back_of_lineseg(newlineseg, CONSLINESEG);
                }
lineseg1 = CONSLINESEG->nextlineseg;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = lineseg1->type;
                if(type != 1) type = 5;
                lineseg1 = lineseg1->nextlineseg;
                newgeom = read_geomedge(line, type);
                CONSGEOM = back_of_geom2d(newgeom, CONSGEOM);
                }
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                CONSEDGE2D = back_of_edge2d(newedge, CONSEDGE2D);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newtermpoint = read_termpoint(line);
                CONSTERMPOINT = back_of_termpoint(newtermpoint, CONSTERMPO
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                CONSVERT2D = back_of_vert2d(newvert, CONSVERT2D);
                }
                }
if (NUM_ERASPOINTS > 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
                {
```

```
                fgets(line, 63, notes);
                newlineseg = read_lineseg(line);
                ERASLINESEG = back_of_lineseg(newlineseg, ERASLINESEG);
                }
lineseg1 = ERASLINESEG->nextlineseg;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = lineseg1->type;
                if(type != 1) type = 5;
                lineseg1 = lineseg1->nextlineseg;
                newgeom = read_geomedge(line, type);
                ERASGEOM = back_of_geom2d(newgeom, ERASGEOM);
                }
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                ERASEDGE2D = back_of_edge2d(newedge,ERASEDGE2D);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newtermpoint = read_termpoint(line);
                ERASTERMPOINT = back_of_termpoint(newtermpoint, ERASTERMPO
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                ERASVERT2D = back_of_vert2d(newvert, ERASVERT2D);
                }
                }
}
/*-----------------------------------*/
void initialise_solidnodes()
{
SOLVISGEOM = get_geom_edge2d();
SOLHIDGEOM = get_geom_edge2d();
SOLCONSGEOM = get_geom_edge2d();
SOLVISEDGE2D = get_edge2d();
```

```
SOLHIDEDGE2D = get_edge2d();
SOLCONSEDGE2D = get_edge2d();
SOLVISVERT2D = get_vert2d();
SOLHIDVERT2D = get_vert2d();
SOLCONSVERT2D = get_vert2d();
}
/*-------------------------------*/
void read_file_two()
{
int i, j, type;
char c, line[63], line1[6], count, xcoord[6], ycoord[6];
char filename[13];
int numedge, numvert;
Geom_edge2d *newgeom;
Vert2d *newvert;
Edge2d *newedge, *edgelist;
Termpoint *newtermpoint;
FILE *notes;
float *v;
initialise_nodes();
v=vector(0,1);
copy_string(FILENAME, filename);
append_string(".one", filename);
notes = fopen(filename, "r");
fgets(line, 63, notes);
v = read_coord(line);
THREEX = v[0];
THREEY = v[1];
fgets (line, 63, notes);
NUM_VISPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_HIDPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_CENTPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_CONSPOINTS = atoi(line);
fgets(line, 63, notes);
NUM_ERASPOINTS = atoi(line);
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
```

```
numvert = atoi(line);
for(i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        newedge = read_edge2d(line);
        SOLVISEDGE2D = back_of_edge2d(newedge, SOLVISEDGE2D);
        }
edgelist = SOLVISEDGE2D->nextedge2d;
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        type = edgelist->edgetype;
        edgelist = edgelist->nextedge2d;
        newgeom = read_geomedge(line, type);
        SOLVISGEOM = back_of_geom2d(newgeom, SOLVISGEOM);
        }
for (i=0; i<numvert; i++)
        {
        fgets(line, 63, notes);
        newvert = read_vertex2d(line);
        SOLVISVERT2D = back_of_vert2d(newvert, SOLVISVERT2D);
        }
if (NUM_HIDPOINTS 0)
        {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for(i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        newedge = read_edge2d(line);
        SOLHIDEDGE2D = back_of_edge2d(newedge, SOLHIDEDGE2D);
        }
edgelist = HIDEDGE2D-nextedge2d;
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        type = edgelist->edgetype;
        edgelist = edgelist->nextedge2d;
        newgeom = read_geomedge(line, type);
```

```
                SOLHIDGEOM = back_of_geom2d(newgeom, SOLHIDGEOM);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                SOLHIDVERT2D = back_of_vert2d(newvert, SOLHIDVERT2D);
                }
                }
if(NUM_CENTPOINTS > 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                newedge = read_edge2d(line);
                SOLCENEDGE2D = back_of_edge2d(newedge, SOLCENEDGE2D);
                }
edgelist = SOLCENEDGE2D->nextedge2d;
for (i=0; i<numedge; i++)
                {
                fgets(line, 63, notes);
                type = edgelist->edgetype;
                edgelist = edgelist->nextedge2d;
                newgeom = read_geomedge(line, type);
                SOLCENGEOM = back_of_geom2d(newgeom, SOLCENGEOM);
                }
for (i=0; i<numvert; i++)
                {
                fgets(line, 63, notes);
                newvert = read_vertex2d(line);
                SOLCENVERT2D = back_of_vert2d(newvert, SOLCENVERT2D);
                }
                }
if (NUM_CONSPOINTS > 0)
                {
fgets(line,63,notes);
numedge = atoi(line);
```

```
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        newedge = read_edge2d(line);
        SOLCONSEDGE2D = back_of_edge2d(newedge,
        SOLCONSEDGE2D);
        }
edgelist = SOLCONSEDGE2D-nextedge2d;
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        type = edgelist->edgetype;
        edgelist = edgelist->nextedge2d;
        newgeom = read_geomedge(line, type);
        SOLCONSGEOM = back_of_geom2d(newgeom, SOLCONSGEOM);
        }
for (i=0; i<numvert; i++)
        {
        fgets(line, 63, notes);
        newvert = read_vertex2d(line);
        SOLCONSVERT2D = back_of_vert2d(newvert, SOLCONSVERT2D);
        }
        }
fgets(line,63,notes);
numedge = atoi(line);
fgets(line,63,notes);
numvert = atoi(line);
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        newedge = read_edge2d(line);
        SOLEDG2D = back_of_edge2d(newedge, SOLEDG2D);
        }
edgelist = SOLEDG2D->nextedge2d;
for (i=0; i<numedge; i++)
        {
        fgets(line, 63, notes);
        type = edgelist->edgetype;
        edgelist = edgelist->nextedge2d;
```

```
            newgeom = read_geomedge(line, type);
            SOLGEOM = back_of_geom2d(newgeom, SOLGEOM);
            }
for (i=0; i<numvert; i++)
            {
            fgets(line, 63, notes);
            newvert = read_vertex2d(line);
            SOLVERT2D = back_of_vert2d(newvert, SOLVERT2D);
            }
}
/*-----------------------------------*/
```

GRAPHIC OUTPUT FUNCTIONS

```
/*                GRAPHOUT.C
*
  This program contains functions to output the sketch
*/
/*-----------------------------*/
set_video_mode()
  {
  _setvideomode(_VRES16COLOR);
  _remappalette(1,_BRIGHTWHITE);
  _setbkcolor(_BLUE);
  _setwindow(1,0,0,639,479);
  _setcolor(1);
  }
/*-----------------------------*/
set_dashed_line()
  {
  _setlinestyle(0xF0F0);
  }
/*-----------------------------*/
set_centre_line()
  {
  _setlinestyle(0xF18F);
  }
/*-----------------------------*/
set_solid_line()
  {
  _setlinestyle(0xFFFF);
  }
/*-----------------------------*/
write_linenumber()
  {
  static unsigned char list[20];
  strcpy(list,"t'courier'");
  _registerfonts("*.FON");
  _setfont(list);
  }
/*-----------------------------*/
```

```
draw_VISLINES(Lineseg *VISLINESEG, int linattr, float *VISPOINTS[2],
                                Termpoint *VISTERMPOINT)
  {
  int numpts, startpt,i,j;
  float x1, y1, x2, y2, sumx, sumy, meanx, meany;
  Lineseg *linesegment1;
  Termpoint *termpoint1;
  char linenumber[3];
  termpoint1 = VISTERMPOINT;
  linesegment1 = VISLINESEG;
  if (linattr = = 1) set_solid_line();
  if (linattr = = 2) set_dashed_line();
  if (linattr = = 3) set_centre_line();
  write_linenumber();
  j = 0;
  linesegment1 = linesegment1- > nextlineseg;
  while (linesegment1 ! = NULL)
              {
              j + +;
              numpts = linesegment1- > num_points;
              startpt = linesegment1- > start_point;
              x1 = VISPOINTS[startpt-1][0]*.2;
              y1 = VISPOINTS[startpt-1][1]*.2;
              sumx = x1;
              sumy = y1;
              _moveto_w(x1,y1);
              for(i = 1; i < numpoints; i + +)
                          {
                          x2 = VISPOINTS[startpt-1 + i][0]*.2;
                          y2 = VISPOINTS[startpt-1 + i][1]*.2;
                          sumx = sumx + x2;
                          sumy = sumy + y2;
                          _lineto_w(x2,y2);
                          }
              meanx = sumx/numpts;
              meany = sumy/numpts;
              _moveto_w(meanx, meany);
              ultoa(j, linenumber, 10);
              _outgtext(linenumber);
              linesegment1 = linesegment1- > nextlineseg;
```

```
            }
  j=0;
  linesegment1=VISLINESEG->nextlineseg;
  while (linesegment1 != NULL)
            {
            j++;
            numpts = linesegment1->num_points;
            startpt = linesegment1->start_point;
            x1=VISPOINTS[startpt-1][0]*.2+300.0;
            y1=VISPOINTS[startpt-1][1]*.2;
            _moveto_w(x1,y1);
            for(i=1; i<numpoints; i++)
                        {
                        x2 = VISPOINTS[startpt-1+i][0]*.2+300.0;
                        y2 = VISPOINTS[startpt-1+i][1]*.2;
                        _lineto_w(x2,y2);
                        }
            linesegment1=linesegment1->nextlineseg;
            }
  j=0;
  while(termpoint1 != NULL)
            {
            j++;
            termpoint1 = termpoint1->nexttermpoint;
            meanx=termpoint1->point[0]*.2 +300;
            meany=termpoint1->point[1]*.2;
            _moveto_w(meanx, meany);
            ultoa(j, linenumber, 10);
            _outgtext(linenumber);
            }
  }
/*-----------------------------*/
draw_VISEDGE2D(Edge2d *VISEDGE2D, int linattr, Vert2d *VISVERT2D)
  {
  int n1, n2, i, j, k;
  float x1, y1, x2, y2, meanx, meany;
  Edge2d *edgelist1;
  Vert2d *vertlist1;
  char linenumber[3];
  edgelist1 = VISEDGE2D;
  if (linattr==1) set_solid_line();
```

```
if (linattr= =2) set_dashed_line();
if (linattr= =3) set_centre_line();
write_linenumber();
j=0;
edgelist1 = edgelist1->nextedge2d;
while (edgelist1 != NULL)
        {
        j++;
        k = 0;
        vertlist1 = VISVERT2D->nextvert2d;
        while (vertlist1 != NULL)
                {
                k++;
                if (edgelist1->vert1 = = k)
                        {
                        x1 = vertlist1->point[0]*0.2;
                        y1 = vertlist1->point[1]*0.2;
                        n1 = k;
                        }
                if (edgelist1-vert2 = = k)
                        {
                        x2 = vertlist1->point[0]*0.2;
                        y2 = vertlist1->point[1]*0.2;
                        n2 = k;
                        }
                vertlist1 = vertlist1->nextvert2d;
                }
                _moveto_w(x1, y1);
                _lineto_w(x2,y2);
                meanx=((x1+x2)/2);
                meany=((y1+y2)/2);
                _moveto_w(meanx, meany);
                ultoa(j, linenumber, 10);
                _outgtext(linenumber);
                vertlist1 = vertlist1->nextvert2d;
                _moveto_w(x1+300, y1);
                _lineto_w(x2+300,y2);
                ultoa(n1, linenumber, 10);
                _moveto_w(x1+300, y1);
                _outgtext(linenumber);
                ultoa(n2, linenumber, 10);
```

```
                    _moveto_w(x2 + 300, y2);
                    _outgtext(linenumber);
            edgelist1 = edgelist1->nextedge2d;
}
}
/*-----------------------------*/
draw_HIDLINES(Lineseg *HIDLINESEG, int linattr, float *HIDPOINTS[2],
                    Termpoint *HIDTERMPOINT)
  {
  int numpts, startpt,i,j;
  float x1, y1, x2, y2, sumx, sumy, meanx, meany;
  Lineseg *linesegment1;
  Termpoint *termpoint1;
  char linenumber[3];
  termpoint1 = HIDTERMPOINT;
  linesegment1 = HIDLINESEG;
  if (linattr = = 1) set_solid_line();
  if (linattr = = 2) set_dashed_line();
  if (linattr = = 3) set_centre_line();
  write_linenumber();
  j = 0;
  linesegment1 = linesegment1->nextlineseg;
  while (linesegment1 ! = NULL)
          {
          j + +;
          numpts = linesegment1->num_points;
          startpt = linesegment1->start_point;
          x1 = HIDPOINTS[startpt-1][0]*.2;
          y1 = HIDPOINTS[startpt-1][1]*.2;
          sumx = x1;
          sumy = y1;
          _moveto_w(x1,y1);
          for(i = 1; i < numpoints; i + +)
                    {
                    x2 = HIDPOINTS[startpt-1 + i][0]*.2;
                    y2 = HIDPOINTS[startpt-1 + i][1]*.2;
                    sumx = sumx + x2;
                    sumy = sumy + y2;
                    _lineto_w(x2,y2);
                    }
          meanx = sumx/numpts;
```

```
                meany = sumy/numpts;
                _moveto_w(meanx, meany);
                ultoa(j, linenumber, 10);
                _outgtext(linenumber);
                linesegment1 = linesegment1->nextlineseg;
                }
    j = 0;
    linesegment1 = HIDLINESEG->nextlineseg;
    while (linesegment1 != NULL)
                {
                j++;
                numpts = linesegment1->num_points;
                startpt = linesegment1->start_point;
                x1 = HIDPOINTS[startpt-1][0]*.2 + 300.0;
                y1 = HIDPOINTS[startpt-1][1]*.2;
                _moveto_w(x1,y1);
                for(i = 1; i < numpoints; i++)
                            {
                            x2 = HIDPOINTS[startpt-1 + i][0]*.2 + 300.0;
                            y2 = HIDPOINTS[startpt-1 + i][1]*.2;
                            _lineto_w(x2,y2);
                            }
                linesegment1 = linesegment1->nextlineseg;
                }
    j = 0;
    while(termpoint1 != NULL)
                {
                j++;
                termpoint1 = termpoint1-nexttermpoint;
                meanx = termpoint1-point[0]*.2 + 300;
                meany = termpoint1-point[1]*.2;
                _moveto_w(meanx, meany);
                ultoa(j, linenumber, 10);
                _outgtext(linenumber);
                }
    }
/*-----------------------------*/
draw_HIDEDGE2D(Edge2d *HIDEDGE2D, int linattr, Vert2d *HIDVERT2D)
    {
    int n1, n2, i, j, k;
    float x1, y1, x2, y2, meanx, meany;
```

```
Edge2d *edgelist1;
Vert2d *vertlist1;
char linenumber[3];
edgelist1 = HIDEDGE2D;
if (linattr= =1) set_solid_line();
if (linattr= =2) set_dashed_line();
if (linattr= =3) set_centre_line();
write_linenumber();
j=0;
edgelist1 = edgelist1-nextedge2d;
while (edgelist1 != NULL)
        {
        j++;
        k = 0;
        vertlist1 = HIDVERT2D-nextvert2d;
        while (vertlist1 != NULL)
                {
                k++;
                if (edgelist1-vert1 = = k)
                        {
                        x1 = vertlist1-point[0]*0.2;
                        y1 = vertlist1-point[1]*0.2;
                        n1 = k;
                        }
                if (edgelist1-vert2 = = k)
                        {
                        x2 = vertlist1-point[0]*0.2;
                        y2 = vertlist1-point[1]*0.2;
                        n2 = k;
                        }
                vertlist1 = vertlist1-nextvert2d;
                }
                _moveto_w(x1, y1);
                _lineto_w(x2,y2);
                meanx=((x1+x2)/2);
                meany=((y1+y2)/2);
                _moveto_w(meanx, meany);
                ultoa(j, linenumber, 10);
                _outgtext(linenumber);
                vertlist1 = vertlist1-nextvert2d;
                _moveto_w(x1+300, y1);
```

```
                    _lineto_w(x2 + 300,y2);
                    ultoa(n1, linenumber, 10);
                    _moveto_w(x1 + 300, y1);
                    _outgtext(linenumber);
                    ultoa(n2, linenumber, 10);
                    _moveto_w(x2 + 300, y2);
                    _outgtext(linenumber);
            edgelist1 = edgelist1->nextedge2d;
}
}
/*------------------------------*/
```

PROCESSING IN TWO DIMENSIONS FUNCTIONS

```
/*          PROC2D.C
*/
/*-------------------------------*/
float *gauss(float *m[5])
{
float *v, temp[5], mult, s;
int i, j, k, l, t;
for (k=0; k<3; k++)
        {
        for (i=k+1; i<4; i++)
        {
        t=k+1;
        while (fabs(m[k][k])  0.01)
                {
                for (l=0; l; l++)
                        {
                        temp[l] = m[k][l];
                        m[k][l] = m[t][l];
                        m[t][l] = temp[l];
                        }
                t++;
                }
        mult = m[i][k]/m[k][k];
        m[i][k] = 0;
        for(j = k+1; j<5; j++)
                        {
                        m[i][j] = m[i][j] - mult*m[k][j];
                        }
        }
        }
v = vector(0,3);
v[3] = m[3][4]/m[3][3];
i=2;
while(!(i  < 0))
        {
        s=0;
        for (j=i+1; j<4; j++)
                {
```

```
                        s = s + m[i][j]*v[j];
                        }
            v[i] = (m[i][4] - s)/m[i][i];
            i--;
            }
return(v);
}
/*-------------------------------*/
float perp_distance(float x1, float y1, float x2, float y2,
                                        float x, float y)
{
float r,deltax,deltay,d;
deltax = x2-x1;
deltay = y2-y1;
r = sqrt ((deltax*deltax) + (deltay*deltay));
d = (-x*deltay + y*deltax -(x2*y1 - x1*y2))/r;
return(d);
}
/*-------------------------------*/
/*                        findslope(float x1, float y1, float x2, float y2)
*
This function finds the slope of the line connecting the points (x1,y1)
and (x2,y2) in degrees ranging from -180 to 180. The return value is
the angle.
*
*/
float findslope(float x1, float y1, float x2, float y2)
{
float angle;
if(fabs(x2-x1))
            {
            angle = 90.0;
            if((y2-y1)) angle = -90;
            }
else
            {
    angle = atan2((y2-y1),(x2-x1));
            angle = 180*7*angle/22;
            }
return(angle);
}
```

```
/*-------------------------------*/
float *line_equation(float x1, float y1, float x2, float y2)
{
float *v;
v=vector(0,2);
if (fabs(x2 - x1)  25.0)
                {
                v[0]=0;
                v[1]=1;
                v[2]=-(x1+x2)/2;
                return(v);
                }
else
                {
                v[0]=1;
                v[1]=(y2-y1)/(x1-x2);
                v[2]=(x2*y1 - x1*y2)/(x1-x2);
                return(v);
                }
}
/*--------------------------------*/
int side_of_line(float v[3], float x, float y)
{
int side=1;
float value;
value = v[0]*y+v[1]*x+v[2];
if (value < 0) side = -1;
return(side);
}
/*------------------------------ */
/*        distance(float x1, float y1, float x2, float y2)
*
This function finds the distance between points (x1,y1) and (x2,y2)
The return value is the distance.
*
*/
float distance(float x1, float y1, float x2, float y2)
{
float d;
d = sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
return(d);
```

```
}
/*-----------------------------------*/
/*                    test_line_end(int stpt, float *points[2])
*
This function returns the row number of the end point of the test line
(of length less than 10 mm).
*
*/
int test_line_end(int stpt, float *points[2])
{
int i;
float d, x1, y1, x2, y2;
x1 = points[stpt-1][0];
y1 = points[stpt-1][1];
d = 0;
i = 0;
while (d  MINDIST)
        {
        i + +;
        x2 = points[stpt + i-1][0];
        y2 = points[stpt + i-1][1];
        d = distance(x1, y1, x2, y2);
        }
return(stpt + i-1);
}
/*-----------------------------------*/
float find_max_testdistance(int stpt, float *points[2])
{
float d, x1 ,y1, x2, y2, x, y, temp;
int n, i;
x1 = points[stpt-1][0];
y1 = points[stpt-1][1];
n = test_line_end(stpt,points);
x2 = points[n-1][0];
y2 = points[n-1][1];
d = 0;
for (i = 0; i < (n-stpt-2); i + +)
        {
        temp = perp_distance(x1,y1,x2,y2, points [stpt-i][0],
                                points[stpt-i][1]);
        if (d  fabs(temp)) d = fabs(temp);
```

```
                }
return(d);
}
/*-----------------------------------*/
/*                    find_straight_end(int r, float *points[2])
*
This is a function finds the row number of the end point of a straight
line segment in the array points. The return value is the row number.
It also sets the value of the variable CONTINUITY as 0 or 1 according
to whether the next line starts fresh or continues from this line.
*
*/
int find_straight_end(int r, float *points[2])
{
int  pointcount;
float d, cur_slope, sum_slope, mean_slope;
float stpt[2], cur_pt[2], meanx, meany;
int found = 0;
SAMPLE = 6;
pointcount = r + SAMPLE;
stpt[0] = points[r-1][0];
stpt[1] = points[r-1][1];
cur_pt[0] = points[r + SAMPLE-1][0];
cur_pt[1] = points[r + SAMPLE-1][1];
cur_slope = findslope(stpt[0], stpt[1], cur_pt[0], cur_pt[1]);
sum_slope = cur_slope*(SAMPLE-1);
while (found = = 0)
                {
                cur_pt[0] = points[pointcount][0];
                cur_pt[1] = points[pointcount][1];
                meanx = 0.5*(points[pointcount-1][0] + points[pointcount-2][0]);
                meany = 0.5*(points[pointcount-1][1] + points[pointcount-2][1]);
                cur_slope = findslope(meanx, meany, cur_pt[0], cur_pt[1]);
                sum_slope = sum_slope + cur_slope;
                mean_slope = sum_slope/(pointcount - r);
                if(fabs(cur_slope - mean_slope)  STRAIGHTTOLDEG)
                                {
                                found = 1;
          d = distance(points[pointcount-1][0],
                                                points[pointcount-1][1], cur_pt[0],
                cur_pt[1]);
```

```
                    if(d > MINDIST) CONTINUITY = 0;
                    if(d < = MINDIST) CONTINUITY = 1;
                    }
          else pointcount + +;
          }
LINE_TYPE = 1;
return(pointcount);
}
/*-----------------------------------*/
/*                    find_curve_end(int r, float *points[2])
*
This function finds the row number of the end point of a curve in the
array points. The return value is the row number. It also sets the
value of the variable CONTINUITY.
*
*/
int find_curve_end(int r, float *points[2])
{
int  pointcount, change_count, sense = 1;
float stpt[2], finpt[2], cur_pt[2];
int found = 0;
float meanx, meany, d, cur_slope, slope_change, sum_change,mean_change;
float last_slope;
pointcount = r + 2;
cur_pt[0] = points[r + 1][0];
cur_pt[1] = points[r + 1][1];
meanx = 0.5*(points[r-1][0] + points[r][0]);
meany = 0.5*(points[r-1][1] + points[r][1]);
last_slope = findslope(meanx, meany, cur_pt[0],cur_pt[1]);
change_count = 0;
sum_change = 0;
while (found = = 0)
          {
          cur_pt[0] = points[pointcount][0];
    cur_pt[1] = points[pointcount][1];
          meanx = 0.5*(points[pointcount-1][0] + points[pointcount-2][0]);
          meany = 0.5*(points[pointcount-1][1] + points[pointcount-2][1]);
          cur_slope = findslope(meanx, meany, cur_pt[0], cur_pt[1]);
          slope_change = cur_slope - last_slope;
          last_slope = cur_slope;
          change_count + +;
```

```
            if(change_count = = 1)
                        {
                        if(slope_change < 0) sense = -1;
                        if(slope_change > 0) sense = 1;
                        }
            else
                        {
                        if ((sense * slope_change) < 0) found = 1;
                        if (fabs(slope_change) < 0.10) found = 1;
                        }
            if (found = = 1)
                        {
                        d = distance(points[pointcount-1][0],
              points[pointcount-1][1],
                                    cur_pt[0], cur_pt[1]);
                        if (d  MINDIST) CONTINUITY = 0;
                        if (d  MINDIST) CONTINUITY = 1;
                        }
            pointcount + +;
            }
LINE_TYPE = 2*sense;
return(pointcount - 1);
}
/*-----------------------------------*/
int decide_lineclass_side(int stpt, float *points[2])
{
int i, lineclass = 1, side, oneside = 1, n,side1,n1;
float maxdist,*v, x1,y1,x2,y2,x,y;
n = test_line_end(stpt, points);
SAMPLE = 6;
if ((n-stpt) < 6) SAMPLE = n-stpt;
x1 = points[stpt-1][0];
y1 = points[stpt-1][1];
x2 = points[stpt + SAMPLE-1][0];
y2 = points[stpt + SAMPLE-1][1];
v = line_equation(x1,y1,x2,y2);
            x = points[stpt][0];
            y = points[stpt][1];
            side1 = side_of_line(v,x,y);
for(i = 1; i < SAMPLE-2; i + +)
            {
```

```
                x = points[stpt+i][0];
                y = points[stpt+i][1];
                side = side_of_line(v,x,y);
                if (side1*side  0) oneside = 0;
                }
if (oneside = = 1)
                {
                n1 = find_curve_end(stpt,points);
                if(n_1)
                                {
                                maxdist = distance(points[n1-1][0], points[n1-1][1],
                                                points[n-1][0], points[n-1][1]);
                                if(maxdist  MINDIST) lineclass = LINE_TYPE;
                                }
                }
return(lineclass);
}
/*-------------------------------*/
/*                              decide_lineclass_slope(int r, float *points)
*
This function decides whether a line starting from the rth point in
the array points is a curve or a straight line or a curve. It returns
1 if it is a straight line, -2 if it is a clockwise curve and +2 if
it is an anti-clockwise curve. It uses the slope change method.
*
*/
int decide_lineclass_slope(int r, float *points[2])
{
int i, line_class, slopecount;
float **sample_points, *sample_slopes;
float meanx, meany, mean_slope_change, ssc;
float *sample_slope_changes, ss=0;
i = test_line_end(r, points);
SAMPLE = 6;
if ((i-r)  6) SAMPLE = (i-r);
sample_points=fmatrix(0,(SAMPLE-1),0,1);
sample_slopes=vector(0,(SAMPLE-2));
sample_slope_changes=vector(0,(SAMPLE-3));
for(i=0; i<SAMPLE; i++)
                {
                sample_points[i][0]=points[r-1][0];
```

```
                sample_points[i][1] = points[r-1][1];
                r++;
                }
for(i=0; i<(SAMPLE-1); i++)
                {
                if (i)
                                {
                                meanx = sample_points[i][0];
                                meany = sample_points[i][1];
                                }
                else
                                {
                                meanx = (sample_points[i][0] + sample_points[i-1][0]
                                                + sample_points[i-2][0])/3;
                                meany = (sample_points[i][1] + sample_points[i-1][1]
                                                + sample_points[i-2][1])/3;
                                }
                sample_slopes[i] = findslope(meanx, meany,
                                                sample_points[i+1][0], sample_points[i+1][1]);
                ss = ss + sample_slopes[i];
                }
ssc = 0.0;
slopecount = 0;
for(i=0; i<(SAMPLE-2); i++)
                {
                sample_slope_changes[i] = sample_slopes[i+1] -sample_slopes[i];
                if(fabs(sample_slope_changes[i])œ)
                                {
                                ssc = ssc + fabs(sample_slope_changes[i]);
                                slopecount++;
                                }
                }
mean_slope_change = ssc/slopecount;
if (fabs(mean_slope_change) < STRAIGHTTOLDEG)
                {
                line_class = 1;
                STRAIGHTSLOPE = ss/(SAMPLE-1);
                }
else
                {
        if (mean_slope_change < 0) line_class = -2;
```

```
                if (mean_slope_change > 0) line_class = 2;
                }
LINE_TYPE = line_class;
SAMPLE = 6;
return(line_class);
}
/*-------------------------------*/
/*                              parse(float *points[2], int num_points)
*
This function breaks the array of points into line segments. The
details of each segment are stored in a Lineseg node and all the nodes
are connected into a linked list. The function returns this linked list.
*
*/
Lineseg *parse(float *points[2], int num_points)
{
int parsed_points, pts, line_class, finished = 0, i;
Lineseg *new, *linelist;
linelist = get_lineseg();
parsed_points = 1;
i = 0;
while (finished = = 0)
                {
                line_class = decide_lineclass_side(parsed_points, points);
                if (line_class = = 1)
                                                pts = find_straight_end(parsed_points, points);
                else
                                                pts = find_curve_end(parsed_points, points);
                new = get_lineseg();
                i + +;
                new-start_point = parsed_points;
                new-stpt[0] = points[parsed_points-1][0];
                new-stpt[1] = points[parsed_points-1][1];
                new-finpt[0] = points[pts-1][0];
                new-finpt[1] = points[pts-1][1];
                new-num_points = pts - parsed_points;
                new-type = LINE_TYPE;
                linelist = back_of_lineseg(new, linelist);
                if (CONTINUITY = = 0) parsed_points = pts + 1;
                if (CONTINUITY = = 1) parsed_points = pts;
                if (parsed_points = num_points) finished = 1;
```

```
                }
return(linelist);
}
/* ----------------------------------*/
/*              termpoint_member
*
This function decides whether a point given by the vector pt is a
member of the list of termpoints. Returns 1 if it is a member and
0 if not.Adds the line number to the list of lines if a member.
*
*/
int termpoint_member(Termpoint *termpoints, int linenumber, float pt[2])
{
int found;
float tp[2];
Termpoint *termpoint1;
Intnode *linelist, *newintnode;
found = 0;
termpoint1 = termpoints;
while ((termpoints != NULL) && (found != 1))
                {
                termpoints = termpoints->nexttermpoint;
                tp[0] = termpoints->point[0];
                tp[1] = termpoints->point[1];
                if(distance(pt[0], pt[1], tp[0], tp[1]) < 25.0)
                                {
                                found = 1;
                                linelist = termpoints->linelist;
                                newintnode = get_intnode();
                                newintnode->a = linenumber;
                                linelist = back_of_intnode(newintnode, linelist);
                                termpoints->linelist = linelist;
                                }
                }
termpoints = termpoint1;
return(found);
}
/* ------------------------------- */
/*              find_termpoint(Lineseg *linesegs)
*
This function finds the terminal points of all the line segments in
```

```
the list of Lineseg nodes, linesegs. It returns a list of Termpoint
nodes, with one node for each Lineseg node.
*
*/
Termpoint *find_termpoint(Lineseg *linesegs)
{
float tp[2], trialpt1[2], trialpt2[2];
int i, found, linenumber;
Intnode *linelist, *newintnode;
Lineseg *lineseg1;
Termpoint *termpoints, *newtermpoint;
linenumber = 1;
lineseg1 = linesegs;
linesegs = linesegs->nextlineseg;
termpoints = get_termpoint();
linelist = get_intnode();
newintnode = get_intnode();
newintnode->a = 1;
linelist = back_of_intnode(newintnode, linelist);
newtermpoint = get_termpoint();
newtermpoint->point[0] = linesegs->stpt[0];
newtermpoint->point[1] = linesegs->stpt[1];
newtermpoint->linelist = linelist;
termpoints = back_of_termpoint(newtermpoint, termpoints);
newtermpoint = get_termpoint();
linelist = get_intnode();
newintnode = get_intnode();
newintnode->a = 1;
linelist = back_of_intnode(newintnode, linelist);
newtermpoint->point[0] = linesegs->finpt[0];
newtermpoint->point[1] = linesegs->finpt[1];
newtermpoint->linelist = linelist;
termpoints = back_of_termpoint(newtermpoint, termpoints);
linesegs = linesegs->nextlineseg;
while (linesegs != NULL)
        {
        linenumber++;
        for (i=0; i<2; i++)
                {
                trialpt1[i] = linesegs->stpt[i];
                trialpt2[i] = linesegs->finpt[i];
```

```
                    }
            found = termpoint_member(termpoints, linenumber, trialpt1);
            if (found = = 0)
                    {
                    linelist = get_intnode();
                    newintnode = get_intnode();
                    newintnode->a = linenumber;
                    linelist = back_of_intnode(newintnode, linelist);
                    newtermpoint = get_termpoint();
                    newtermpoint->point[0] = trialpt1[0];
                    newtermpoint->point[1] = trialpt1[1];
                    newtermpoint->linelist = linelist;
                    termpoints = back_of_termpoint(newtermpoint,
                                                                        termp
                    }
    found = termpoint_member(termpoints, linenumber, trialpt2);
            if (found = = 0)
                    {
                    linelist = get_intnode();
                    newintnode = get_intnode();
                    newintnode->a = linenumber;
                    linelist = back_of_intnode(newintnode, linelist);
                    newtermpoint = get_termpoint();
                    newtermpoint->point[0] = trialpt2[0];
                    newtermpoint->point[1] = trialpt2[1];
                    newtermpoint->linelist = linelist;
                    termpoints = back_of_termpoint(newtermpoint, termpoints);
                    }
            linesegs = linesegs->nextlineseg;
            }
return(termpoints);
}
/*-------------------------------*/
/*                    leastsqr_straight(int np, int sp, float *points[2])
*
This function fits a straight line for the np points starting from
the spth row in the array points. The return value is a vector whose
elements are the co-efficients of the equation -y + ax + b = 0.
*
*/
float *leastsqr_straight(int np, int sp, float *points[2])
```

```
{
int i, j, n;
float *v;
float a, b;
float sumx = 0, sumy = 0, sumxsqrd = 0, sumxy = 0;
float angle;
for (i = 0; i < np; i + +)
                {
                sumx = points[sp-1 + i][0] + sumx;
                sumy = points[sp-1 + i][1] + sumy;
                sumxsqrd = points[sp-1 + i][0]*points[sp-1 + i][0] + sumxsqrd;
                sumxy = points[sp-1 + i][0]*points[sp-1 + i][1] + sumxy;
                }
b = ((sumx * sumxy - sumxsqrd * sumy)/(sumx*sumx - np*sumxsqrd));
a = (sumy - np*b)/sumx;
angle = atan(a)*180*7/22;
if (fabs(30-angle)) angle = 30.0;
if (fabs(30 + angle)) angle = -30.0;
a = tan((angle*22)/(180*7));
b = sumy/np - a*sumx/np;
v = vector(0,2);
v[0] = -1;
v[1] = a;
v[2] = b;
return(v);
}
/*-----------------------------------*/
/*                                fit_vertical(int np, int sp, float *points[2])
*
This function fits a vertical line. It returns a vector with three
elements whose first element is 0, second element is -1 and third elemen
element is the value b in the equation -x + b = 0.
*
*/
float *fit_vertical (int np, int sp, float *points[2])
{
int i, j;
float sumx = 0.0;
float *v;
for (i = 0; i < np; i + +)
                {
```

```
                sumx = sumx + points[sp-1+i][0];
                }
v = vector(0,2);
v[0] = 0;
v[1] = -1;
v[2] = sumx/np;
return(v);
}
/*-----------------------------------*/
float *least_square_ellipse(int np,int sp, float *points[2])
{
int i, j;
float **m, *v;
float sumx=0, sumxsqrd=0, sumxcubed=0;
float sumy=0, sumysqrd=0, sumycubed=0, sumyto4=0;
float sumxy=0, sumxsqrdy=0, sumxcubedy=0;
float sumxsqrdysqrd=0, sumxysqrd=0, sumxycubed=0;
for (i=0; i<np; i++)
                {
                sumx = sumx + points[sp-1+i][0];
                sumxsqrd = sumxsqrd + points[sp-1+i][0]*points[sp-1+i][0];
                sumxcubed = sumxcubed + points[sp-1+i][0]*points[sp-1+i][0]*
                                                points[sp-1+i][0];
                sumy = sumy + points[sp-1+i][1];
                sumysqrd = sumysqrd + points[sp-1+i][1]*points[sp-1+i][1];
                sumycubed = sumycubed + points[sp-1+i][1]*points[sp-1+i][1]*
                                                points[sp-1+i][1];
                sumyto4 = sumyto4 + points[sp-1+i][1]*points[sp-1+i][1]*
                  points[sp-1+i][1]*points[sp-1+i][1];
                sumxy = sumxy + points[sp-1+i][0]*points[sp-1+i][1];
                sumxsqrdy = sumxsqrdy + points[sp-1+i][0]*points[sp-1+i][0]*
                                                points[sp-1+i][1];
                sumxcubedy = sumxcubedy + points[sp-1+i][0]*points[sp-1+i][0]*
                                                 points[sp-1+i][0]*points[sp-1+i]
                sumxsqrdysqrd = sumxsqrdysqrd + points[sp-1+i][0]*
                            points[sp-1+i][0]*points[sp-1+i][1]*points[sp-1+i][1];
                sumxysqrd = sumxysqrd + points[sp-1+i][0]*points[sp-1+i][1]*
                                                points[sp-1+i][1];
                sumxycubed = sumxycubed + points[sp-1+i][0]*points[sp-1+i][1]*
                                                 points[sp-1+i][1]*points[sp-1+i]
                }
```

```
m = fmatrix(0,3,0,4);
m[0][4] = -(sumxcubedy - sumxsqrd*sumxy/np);
m[0][0] = sumxsqrdysqrd - sumxy*sumxy/np;
m[0][1] = sumxycubed - sumxy*sumysqrd/np;
m[0][2] = sumxsqrdy - sumxy*sumx/np;
m[0][3] = sumxysqrd - sumxy*sumy/np;
m[1][4] = -(sumxsqrdysqrd - sumxsqrd*sumysqrd);
m[1][0] = m[0][1];
m[1][0] = sumxsqrdysqrd - sumxsqrd*sumysqrd/np;
m[1][1] = sumyto4 - sumysqrd*sumysqrd/np;
m[1][2] = sumxysqrd -sumysqrd*sumx/np;
m[1][3] = sumycubed - sumysqrd*sumy/np;
m[2][4] = -(sumxcubed - sumxsqrd*sumx/np);
m[2][0] =  m[0][2];
m[2][1] = m[1][2];
m[2][2] = sumxsqrd - sumx*sumx/np;
m[2][3] = sumxy - sumx*sumy/np;
m[3][4] = -(sumxsqrdy - sumxy*sumx/np);
m[3][0] = m[0][3];
m[3][1] = m[1][3];
m[3][2] = m[2][3];
m[3][3] = sumysqrd - sumy*sumy/np;
v = gauss(m);
return(v);
}
/*-----------------------------------*/
/*                    find_geom_edge2d(Lineseg *linelist, float *points[2])
*
This function fits the appropriate curves for each of the line
segment in linelist. It returns a list of the nodes 'Geom_edge2d'.
*
*/
Geom_edge2d *find_geom_edge2d(Lineseg *linelist, float *points[2])
{
Lineseg *linelist1;
int i, j;
float *m;
Geom_edge2d *geomlist, *newgeom;
geomlist = get_geom_edge2d();
linelist1 = linelist;
linelist = linelist->nextlineseg;
```

```
while (linelist != NULL)
            {
            j = 0;
            if(linelist->type==1)
                        {
        if (fabs(linelist->stpt[0] - linelist->finpt[0]) < 20)
                                    {
                                m = fit_vertical(linelist->num_points,
            linelist->start_point, points);
                                    j = 1;
                                    }
                        else
                                    {
                                    m = leastsqr_straight(linelist->num_points,
              linelist->start_point, points);
                                    j = 1;
                                    }
                        }
            else
                        {
                        m = least_square_ellipse(linelist->num_points,
                                    linelist->start_point, points);
                        j = 2;
                        }
            newgeom = get_geom_edge2d();
            newgeom->v = m;
            newgeom->type = j;
            geomlist = back_of_geom2d(newgeom, geomlist);
            linelist = linelist-nextlineseg;
            }
linelist = linelist1;
return(geomlist);
}
/*-----------------------------------*/
/*                        solve_straight(Geom_edge2d *geomlist, Intnode *intlist)
*
This function finds the least square solution of the meeting point
of lines in the geomlist indicated by the intlist. The return value is
a vector whose elements are the co-ordinates of the meeting point.
*
*/
```

```
float *solve_straight(Geom_edge2d *geomlist, Intnode *intlist)
{
int *v1;
int a1, i=0, j, k;
Intnode *intlist1;
float **m, *v;
float l1=0, m1=0, n1=0, l2=0, m2=0, n2=0;
Geom_edge2d *geomlist1;
geomlist1 = geomlist;
intlist = intlist->nextintnode;
intlist1 = intlist;
while (intlist != NULL)
        {
        i++;
        intlist = intlist->nextintnode;
        }
intlist = intlist1;
v1 = ivector(0,(i-1));
m = fmatrix(0,(i-1),0,2);
a1 = i;
i = 0;
while (intlist1 != NULL)
        {
        v1[i] = intlist1->a;
        i++;
        intlist1 = intlist1->nextintnode;
        }
for (i=0; i<a1; i++)
        {
    for (j=0; j<v[i]; j++) geomlist1 = geomlist1->nextgeom_edge2d;
        for (k=0; k; k++)
                {
                m[i][k] = geomlist1->v[k];
                }
        geomlist1 = geomlist;
        }
for (i=0; ia1; i++)
        {
        l1 = l1+m[i][1]*m[i][1];
        l2 = l2+m[i][1]*m[i][0];
        m2 = m2+m[i][0]*m[i][0];
```

```
                n1 = n1+m[i][1]*m[i][2];
                n2 = n2+m[i][0]*m[i][2];
                }
m1 = l2;
v = vector(0,1);
v[0] = ((m1*n2)-(m2*n1))/((l1*m2)-(l2*m1));
v[1] = ((l2*n1)-(l1*n2))/((l1*m2)-(l2*m1));
return(v);
}
/*-----------------------------------*/
/*              find_vert2d(Termpoint *termpoints, Geom_edge2d *geomlist)
*
This function finds the vertices corresponding to each of the
terminal points in the list termpoints using the analytical equations
representing the edges given by the list geomlist.
*
*/
/*
Vert2d *find_vert2d(Termpoint *termpoints, Geom_edge2d *geomlist)
{
Termpoint *termpoints1;
int i, j;
Intnode *linelist;
float *v;
Vert2d *vertlist, *newvert;
termpoints1 = termpoints;
termpoints1 = termpoints1-nexttermpoint;
vertlist = get_vert2d();
while (termpoints1 != NULL)
                {
                linelist = termpoints1->linelist;
                v = solve_straight(geomlist, linelist);
                newvert = get_vert2d();
                newvert->point = v;
                newvert-linelist = termpoints1->linelist;
                vertlist = back_of_vert2d(newvert, vertlist);
                termpoints1 = termpoints1->nexttermpoint;
                }
return(vertlist);
}
/*-----------------------------------*/
```

```
void show_lineseg(Lineseg *linesegs)
{
Lineseg *lineseg1;
lineseg1 = linesegs;
linesegs = linesegs->nextlineseg;
while (linesegs != NULL)
        {
        printf("Number of points  :  %d\n", linesegs->num_points);
        printf("Starting point    :  %d\n", linesegs->start_point);
        switch (linesegs->type)
                {
                case 1 :
                        printf("It is a straight line\n");
                        break;
                case 2 :
                        printf("It is a closed curve\n");
                        break;
                case 3 :
                        printf("It is an arc\n");
                        break;
                }
        printf("The starting point is :  %3.2f%s%3.2f\n",
                linesegs->stpt[0], "    ",linesegs->stpt[1]);
        printf("The finishing point is:  %3.2f%s%3.2f\n",
                linesegs->finpt[0], "    ",linesegs->finpt[1]);
        linesegs = linesegs->nextlineseg;
        }
linesegs = lineseg1;
}
/*-----------------------------------*/
Vert2d *find_vert2d(Termpoint *termpoints, Geom_edge2d *geomlist)
{
Termpoint *termpoints1;
int i, j;
Intnode *linelist;
float *v;
Vert2d *vertlist, *newvert;
termpoints1 = termpoints;
termpoints1 = termpoints1->nexttermpoint;
vertlist = get_vert2d();
while (termpoints1 != NULL)
```

```
                {
                linelist = termpoints1->linelist;
                v = solve_straight(geomlist, linelist);
                newvert = get_vert2d();
                newvert->point = v;
                newvert->linelist = termpoints1->linelist;
                vertlist = back_of_vert2d(newvert, vertlist);
                termpoints1 = termpoints1->nexttermpoint;
                }
return(vertlist);
}
/*-----------------------------------*/
float slope_of_major(float v[6])
{
float angle;
angle = atan2(v[2],(v[0]-v[1]));
angle = 180*7*angle/22;
return(angle);
}
/*-----------------------------------*/
int decide_linetype(Lineseg *lineseg, Geom_edge2d *geomlist)
{
int i, j, k, linetype=0, sense=1, complete=0;
float angle, tp1[2],tp2[2],d;
if (lineseg->type == 1)
                {
                if (geomlist->v[0] == 0) linetype = 1;
                else
                                {
                                angle = atan(geomlist->v[1])*180*7/22;
                                if ((fabs(angle-30)) 1.0) linetype = 2;
                                if ((fabs(angle+30)) 1.0) linetype = 3;
                                if (linetype == 0) linetype = 4;
                                }
                }
else
                {
                if (lineseg->type == -2) sense = -1;
                tp1[0] = lineseg->stpt[0];
                tp1[1] = lineseg->stpt[1];
                tp2[0] = lineseg->finpt[0];
```

```
            tp2[1] = lineseg->finpt[1];
            d = distance(tp1[0],tp1[1],tp2[0],tp2[1]);
            if (d 50) complete = 1;
            angle = slope_of_major(geomlist-v);
            if ((fabs(angle)-7) < 0) angle = 0;
            if (fabs(angle-30) < 7) angle = 30;
            if (fabs(angle -150) < 7) angle = 150;
            if (complete = = 1)
                        {
                        if (angle = = 0) linetype = sense*5;
                        if (angle = = 30) linetype = sense*6;
                        if (angle = = 150) linetype = sense*7;
                        if (linetype = = 0) linetype = sense*8;
                        }
            else
                        {
                        if (angle = = 0) linetype = sense*9;
                        if (angle = = 30) linetype = sense*10;
                        if (angle = = 150) linetype = sense*11;
                        if (linetype = = 0) linetype = sense*12;
                        }
            }
return(linetype);
}
/* ----------------------------------*/
/*          find_edge2d(Lineseg *linesegs, Vert2d *vertlist
                                    Geom_edge2d *geomlist)
*
This function returns the list of nodes edge2d corresponding to each
in the list linesegs using the vertlist.
*
*/
Edge2d *find_edge2d(Lineseg *linesegs, Vert2d *vertlist,
                                    Geom_edge2d *geomlist)
{
float tp1[2], tp2[2], tp[2], d;
int i=0, j, found =0;
Edge2d *edgelist, *newedge;
Geom_edge2d *geomlist1;
Lineseg *lineseg1;
Vert2d *vertlist1;
```

```
lineseg1 = linesegs;
edgelist = get_edge2d();
vertlist1 = vertlist->nextvert2d;
lineseg1 = lineseg1->nextlineseg;
geomlist1 = geomlist->nextgeom_edge2d;
while (lineseg1 != NULL)
        {
        tp1[0] = lineseg1->stpt[0];
        tp1[1] = lineseg1->stpt[1];
        tp2[0] = lineseg1->finpt[0];
        tp2[1] = lineseg1->finpt[1];
        while ((vertlist1 != NULL) && (found != 1))
                {
                i++;
                tp[0] = vertlist1->point[0];
                tp[1] = vertlist1->point[1];
                d = distance(tp[0], tp[1], tp1[0], tp1[1]);
                if (d  50.0) found = 1;
                vertlist1 = vertlist1->nextvert2d;
                }
        newedge = get_edge2d();
        newedge->vert1 = i;
        newedge->edgetype = decide_linetype(lineseg1, geomlist1);
        found = 0;
        i = 0;
        vertlist1 = vertlist->nextvert2d;
        while ((vertlist1 != NULL) && (found != 1))
                {
                i++;
                tp[0] = vertlist1->point[0];
                tp[1] = vertlist1->point[1];
                d = distance(tp[0], tp[1], tp2[0], tp2[1]);
                if (d  50) found = 1;
                vertlist1 = vertlist1->nextvert2d;
                }
        newedge->vert2 = i;
        found =0;
        i = 0;
        vertlist1 = vertlist->nextvert2d;
        edgelist = back_of_edge2d(newedge, edgelist);
        lineseg1 = lineseg1->nextlineseg;
```

```
                geomlist1 = geomlist1->nextgeom_edge2d;
                }
return(edgelist);
}
/*-------------------------------*/
process2d()
{
read_point_file();
VISLINESEG=parse(VISPOINTS, NUM_VISPOINTS);
VISTERMPOINT = find_termpoint(VISLINESEG);
VISGEOM = find_geom_edge2d(VISLINESEG, VISPOINTS);
VISVERT2D = find_vert2d(VISTERMPOINT, VISGEOM);
VISEDGE2D = find_edge2d(VISLINESEG, VISVERT2D, VISGEOM);
if (NUM_HIDPOINTS 0)
  {
   HIDLINESEG =parse(HIDPOINTS, NUM_HIDPOINTS);
   HIDTERMPOINT = find_termpoint(HIDLINESEG);
   HIDGEOM = find_geom_edge2d(HIDLINESEG, HIDPOINTS);
   HIDVERT2D = find_vert2d(HIDTERMPOINT, HIDGEOM);
   HIDEDGE2D = find_edge2d(HIDLINESEG, HIDVERT2D, HIDGEOM);
  }
if (NUM_CENTPOINTS 0)
  {
  CENLINESEG = parse(CENTPOINTS, NUM_CENTPOINTS);
  CENTERMPOINT = find_termpoint(CENLINESEG);
  CENGEOM = find_geom_edge2d(CENLINESEG, CENTPOINTS);
  CENVERT2D = find_vert2d(CENTERMPOINT, CENGEOM);
  CENEDGE2D = find_edge2d(CENLINESEG, CENVERT2D, CENGEOM);
  }
if (NUM_CONSPOINTS 0)
  {
  CONSLINESEG = parse(CONSPOINTS, NUM_CONSPOINTS);
  CONSTERMPOINT = find_termpoint(CONSLINESEG);
  CONSGEOM = find_geom_edge2d(CONSLINESEG, CONSPOINTS);
  CONSVERT2D = find_vert2d(CONSTERMPOINT, CONSGEOM);
  CONSEDGE2D = find_edge2d(CONSLINESEG, CONSVERT2D, CON-
SGEOM);
  }
if (NUM_ERASPOINTS 0)
  {
  ERASLINESEG = parse(ERASPOINTS, NUM_ERASPOINTS);
```

```
  ERASTERMPOINT = find_termpoint(ERASLINESEG);
  ERASGEOM = find_geom_edge2d(ERASLINESEG, ERASPOINTS);
  ERASVERT2D = find_vert2d(ERASTERMPOINT, ERASGEOM);
  ERASEDGE2D = find_edge2d(ERASLINESEG, ERASVERT2D, ERAS-
GEOM);
  }
write_file_one();
}
/*-----------------------------------*/
```

MERGE FACILITIES FUNCTIONS

```
/*                              merge2d.c
-----------------------------------------*/
/*              parallel_dist_lines(float v1[3], float v2[3])
*
This function calclates the perpendicular distance between two
parallel lines given by the equations v1[3] and v2[3]. The return
value is the distance.
*
*/
float parallel_dist_lines(float v1[3], float v2[3])
{
float dist;
dist = fabs((-v1[2]+v2[2])/(sqrt(v1[0]*v1[0] + v1[1]*v1[1])));
return(dist);
}
/*-------------------------------------*/
/*              div_in_or_out(x1,y1,x2,y2,x,y)
*
This function returns a value 1 if the point (x,y) lies between
(x1,y1) and (x2,y2) and returns 0 if it is outside.
*
*/
int div_in_or_out(x1,y1,x2,y2,x,y)
float x1,y1,x2,y2,x,y;
{
float lambda;
int result=0;
lambda = (x2-x)/(x-x1);
if (lambda > 0) result = 1;
return(result);
}
/*-------------------------------------*/
/*              div_in_or_out_arc(order, geomlist, edgelist, vertlist, x1, y1)
*
This function tells whether point (x1,y1) is inside or outside the
arc represented by the orderth node in the list. The result returned
is '0' if it is outside and '1' if it is inside.
```

```
*/
int div_in_or_out_arc(int order, Geom_edge2d *geomlist,
                Edge2d *edgelist, Vert2d *vertlist, float x1, float y1)
{
float xc, yc, x2, y2, x3, y3;
float *v1, *v2;
int i, n1, n2, side1, side2, ans=0, side3, side4;
Edge2d *edgelist1;
Vert2d *vertlist1;
Geom_edge2d *geomlist1;
for(i=0; i=order; i++)
                {
                geomlist1 = geomlist1->nextgeom_edge2d;
                edgelist1 = edgelist1->nextedge2d;
                }
xc = geomlist->meanx;
yc = geomlist->meany;
n1 = edgelist1->vert1;
n2 = edgelist1-vert2;
vertlist1 = vertlist;
for (i=0; i=n1; i++) vertlist1 = vertlist1->nextvert2d;
x2 = vertlist1->point[0];
y2 = vertlist1->point[1];
vertlist1 = vertlist;
for (i=0; i=n2; i++) vertlist1 = vertlist1->nextvert2d;
x3 = vertlist1->point[0];
y3 = vertlist1->point[1];
v1 = line_equation(xc, yc, x2, y2);
v2 = line_equation(xc, yc, x3, y3);
side1 = side_of_line(v1,x1,y1);
side2 = side_of_line(v1,x3,y3);
side3 = side_of_line(v2,x1,y1);
side4 = side_of_line(v2,x2,y2);
if ((side1*side2  0) && (side3*side4  0)) ans = 1;
return(ans);
}
/*-------------------------------*/
/*                        same_equation(geomlist, numedges, lastchecked)
This function checks whether the (lastchecked + 1)th node in the
list has any nodes behind it with the same equation. If there are
none avector with all elements 0 would be returned. If there are
```

```
any the first element of the vector would be the number and the
following elements would give their ranks in geomlist.
*/
int *same_equation(Geom_edge2d *geomlist, int numedges,
                                        int lastchecked)
{
int ans=0, order, type1, type2, i, *v, *v3, geomrank, numsame;
Geom_edge2d *geomlist1, *geomlist2;
float d1,d2,d3,d4,d5,d6,*v1,*v2;
geomlist1 = geomlist->nextgeom_edge2d;
v=ivector(0,numedges-1);
for(i=0; i<numedges; i++) v[i] = 0;
for(i=0; iastchecked; i++)
geomlist1 = geomlist->nextgeom_edge2d;
order = lastchecked;
geomlist2 = geomlist1->nextgeom_edge2d;
order++;
geomrank = order +1;
while (geomlist2 != NULL)
            {
            type1 = geomlist1->type;
            type2 = geomlist2->type;
            if ((abs(type1)) == (abs(type2)))
                        {
                        v1 = geomlist1->v;
                        v2 = geomlist2->v;
                        if ((abs(type1)) < 5)
                                    {
                                    d1 = fabs(v1[0] - v2[0]);
                                    d2 = fabs(v1[1] - v2[1]);
                                    d3 = fabs(v1[2] - v2[0]);
                                    if(((d1) && (d2)) && (d3)) ans=1;
                                    }
                        if(abs(type1) 4)
                                    {
                                    d1 = fabs(v1[0] - v2[0]);
                                    d2 = fabs(v1[1] - v2[1]);
                                    d3 = fabs(v1[2] - v2[2]);
                                    d4 = fabs(v1[3] - v2[3]);
                                    d5 = fabs(v1[4] - v2[4]);
                                    d6 = fabs(v1[5] - v2[5]);
```

```
                                        if((d1) && (d2) && (d3) &&
                                        (d4) && (d5) && (d6)) ans = 1;
                                        }
                        }
                        if(ans = = 1)
                                        {
                                        v[numsame] = geomrank;
                                        numsame + +;
                                        }
            geomlist2 = geomlist2->nextgeom_edge2d;
            geomrank + +;
            }
if(numsame  0)
            {
            v3 = ivector(0,numsame);
            v3[0] = numsame;
            for (i = 0; i < numsame; i + +)
                        {
                        v3[i + 1] = v[i];
                        }
            free_ivector(v,0,numedges);
            return(v3);
            }
else
            {
            return(v);
            }
}
/*-------------------------------*/
/*                        merge_type_test_straight(order, order1,
                                                    edgelist, vertlist)
```

This function tests two edges with the same equations and find what kind of merging is possible. The result is given in a vector of five elements. The first element tells what kind of merging is possible. A '0' represents no merging possible, a '1' represents the removal of an overstruck edge and a '2' represents a later extension or drawing from both ends. Other elements represent

2 Edge to be retained
3 Edge to be removed
4 First vertex to be removed
5 Second vertex to be removed

```
*/
int *merge_type_test_straight(int order, Edge2d *edgelist,
                                        Vert2d *vertlist, int order1)
{
float x1,y1,x2,y2,x3,y3,x4,y4,d1,d2,d3,d4,l1,l2;
Edge2d *edge1, *edge2;
Vert2d *vert1;
int ans, i, n1, n2, n3, n4, test1, test2, test3, test4, *v;
edge1 = edgelist->nextedge2d;
edge2 = edgelist->nextedge2d;
for (i=0; i<order; i++) edge1 = edge1->nextedge2d;
for (i=0; i<order; i++) edge2 = edge2->nextedge2d;
n1 = edge1->vert1;
n2 = edge1->vert2;
vert1 = vertlist->nextvert2d;
for (i=0; i<n1; i++) vert1 = vert1->nextvert2d;
x1 = vert1->point[0];
y1 = vert1->point[1];
vert1 = vertlist->nextvert2d;
for (i=0; i<n2; i++) vert1 = vert1->nextvert2d;
x2 = vert1->point[0];
y2 = vert1->point[1];
n3 = edge2->vert1;
n4 = edge2->vert2;
vert1 = vertlist->nextvert2d;
for (i=0; i<n3; i++) vert1 = vert1->nextvert2d;
x3 = vert1->point[0];
y3 = vert1->point[1];
vert1 = vertlist->nextvert2d;
for (i=0; i 4; i++) vert1 = vert1->nextvert2d;
x4 = vert1->point[0];
y4 = vert1->point[1];
test1 = div_in_or_out(x1,y1,x2,y2,x3,y3);
test2 = div_in_or_out(x1,y1,x2,y2,x4,y4);
v = ivector(0,4);
l1 = distance(x1,y1,x2,y2);
l2 = distance(x3,y3,x4,y4);
for (i=0; i<5; i++) v[i] =0;
if ((test1 == 0) && (test2 == 0) && (l1  l2))
            {
            d1 = distance(x1,y1,x3,y3);
```

```
            d2 = distance(x2,y2,x3,y3);
            d3 = distance(x1,y1,x4,y4);
            d4 = distance(x2,y2,x4,y4);
            if ((fabs(d1)  MINDIST) && (fabs(d2)  MINDIST) &&
      (fabs(d3)  MINDIST) && (fabs(d4)  MINDIST)) return (v);
            }
    if ((l1  l2) && (test1 = = 1) && (test2 = = 1))
            {
            v[0] = 1; /* type 1 merge remove an overstruck edge */
            v[1] = order; /* edge to be retained */
            v[2] = order1;  /* edge to be removed */
            v[3] = n3;      /* vertex to be removed */
            v[4] = n4;      /* vertex to be removed */
            return (v);
            }
if (((test1 = = 1) && (test2 = = 0)) || ((test1 = = 0) && (test2 = = 1)))
            {
            v[0] = 2; /* type 2 merging later extension  or drawing
                                        from both ends */
            v[1] = order;
            v[2] = order1;
            if (test1 = = 1)
                        {
                        v[3] = n3;
                        test3 = div_in_or_out(x1,y1,x4,y4,x3,y3);
                        if (test3 = = 1) v[4] = n2;
                        if (test3 = = 0) v[4] = n1;
                        }
            if (test2 = = 1)
                        {
                        v[3] = n4;
                        test4 = div_in_or_out(x1,y1,x3,y3,x4,y4);
                        if (test4 = = 1) v[4] = n2;
                        if (test4 = = 0) v[4] = n1;
                        }
            return(v);
            }
if ((l1  l2) && (test1 = = 0) && (test2 = = 0))
    {
            v[0] = 1;
            v[1] = order1;
```

```
            v[2] = order;
            v[3] = n1;
            v[4] = n2;
            return(v);
            }
}
/*-----------------------------------*/
/*          int *merge_type_test_arc(order, order1, geomlist, edgelist,
                                        vertlist)
This function tests whether two arcs with the same equation and
finds out the kind of merging possible. The result is given in a
five element vector. The first element tells the kind of merging
possible. A '0' represents no possible merging, a '1' represents
an overstruck edge and a '2' represents drawing from both ends.
*/
int *merge_type_test_arc(int order, int order1, Edge2d *edgelist,
                        Geom_edge2d*geomlist, Vert2d *vertlist)
{
int end1, end2, end3, end4;
int i, n1, n2, n3, n4, *v, side1, side2;
float x1, y1, x2, y2, x3, y3, x4, y4, xc, yc, d1, d2, d3, d4;
float *line1;
Edge2d *edgelist1;
Vert2d *vertlist1;
Geom_edge2d *geomlist1;
edgelist1 = edgelist;
vertlist1 = vertlist1;
geomlist1 = geomlist;
v = ivector(0,4);
for (i=0; i<5; i++) v[i] =0;
for (i=0; i=order1; i++)
            {
            edgelist1 = edgelist1->nextedge2d;
            geomlist1 = geomlist1-nextgeom_edge2d;
            }
n1 = edgelist1->vert1;
n2 = edgelist1->vert2;
xc = geomlist->meanx;
yc = geomlist->meany;
for (i=0; i=n1; i++) vertlist1 = vertlist1->nextvert2d;
x1 = vertlist1->point[0];
```

```
y1 = vertlist1->point[1];
vertlist1 = vertlist;
for (i=0; i=n2; i++) vertlist1 = vertlist1->nextvert2d;
x2 = vertlist1->point[0];
y2 = vertlist1->point[1];
edgelist1 = edgelist;
vertlist1 = vertlist;
for (i=0; i=order; i++) edgelist1 = edgelist1->nextedge2d;
n3 = edgelist1->vert1;
n4 = edgelist1->vert2;
for (i=0; i=n3; i++) vertlist1 = vertlist1->nextvert2d;
x3 = vertlist1->point[0];
y3 = vertlist1->point[1];
vertlist1 = vertlist;
for (i=0; i=n4; i++) vertlist1 = vertlist1->nextvert2d;
x4 = vertlist->point[0];
y4 = vertlist->point[1];
d1 = distance(x1, y1, x3, y3);
d2 = distance(x1, y1, x4, y4);
d3 = distance(x2, y2, x3, y3);
d4 = distance(x2, y2, x4, y4);
if ((d12) || (d22) || (d32) || (d42))
            {
            v[0] = 2;
            v[1] = order;
            v[2] = order1;
            if ((d1  50) || (d2  50))
                        {
                        v[3] = n1;
                        if (d1  50) v[4] = n3;
                        if (d2  50) v[4] = n4;
                        }
            if ((d3  50) || (d4  50))
                        {
                        v[3] = n2;
                        if (d3  50) v[4] = n3;
                        if (d4  50) v[4] = n4;
                        }
            return(v);
            }
end1 = div_in_or_out_arc(order, geomlist, edgelist, vertlist, x1, y1);
```

```
end2 = div_in_or_out_arc(order, geomlist, edgelist, vertlist, x2, y2);
if ((end1 = = 1) && (end2 = = 1))
            {
            v[0] = 1;
            v[1] = order;
            v[2] = order1;
            v[3] = n1;
            v[4] = n2;
            return(v);
            }
if ((end1 = = 1) && (end2 = = 0))
            {
            v[0] = 2;
            v[1] = order;
            v[2] = order1;
            v[3] = n1;
            line1 = line_equation(xc, yc, x1, y1);
            side1 = side_of_line(line1, x2, y2);
            side2 = side_of_line(line1, x3, y3);
            v[4] = n4;
            if (side1*side2 <0) v[4] = n3;
            return(v);
            }
if ((end1 = = 0) && (end2 = = 1))
            {
            v[0] = 2;
            v[1] = order;
            v[2] = order1;
            v[3] = n2;
            line1 = line_equation(xc, yc, x2, y2);
            side1 = side_of_line(line1, x1, y1);
            side2 = side_of_line(line1, x3, y3);
            v[4] = n4;
            if (side1*side2< 0) v[4] = n3;
            return(v);
            }
end3 = div_in_or_out_arc(order1, geomlist, edgelist, vertlist, x3, y3);
end4 = div_in_or_out_arc(order1, geomlist, edgelist, vertlist, x4, y4);
if (((end1 = = 0) && (end2 = = 0)) && ((end3 = = 1) && (end4 = = 1)))
            {
            v[0] = 1;
```

```
                v[1] = order1;
                v[2] = order;
                v[3] = n3;
                v[4] = n4;
                return(v);
                }
}
/*-----------------------------------*/
int members_in_intlist(Intnode *intlist)
{
int n;
Intnode *intlist1;
intlist1 = intlist1->nextintnode;
while (intlist1 != NULL)
                {
                intlist1 = intlist1->nextintnode;
                n++;
                }
return(n);
}
/*-----------------------------------*/
Intnode *remove_node_intlist(Intnode *intlist, int n)
{
Intnode *intlist1, *intlist2;
int i=0, j;
intlist1 = intlist-nextintnode;
while (intlist1->a != n)
                {
                intlist1 = intlist1->nextintnode;
                i++;
                }
intlist1 = intlist-nextintnode;
for(j=1; j<(i-1); j++) intlist1 = intlist1->nextintnode;
intlist2 = intlist1->nextintnode;
intlist1->nextintnode = intlist2->nextintnode;
kill_intnode(intlist2);
return(intlist);
}
/*-----------------------------------*/
/*              zero_dist_merge(geomlist, edgelist, vertlist, numedges)
This function merges the lines which are having the same equations
```

```
and are given permission by the user.
*/
void zero_dist_merge(Geom_edge2d *geomlist, Edge2d *edgelist,
                                        Vert2d *vertlist, int numedges)
{
int *v, n1, n2,i,*v1, order = 1;
char ch;
Intnode *intlist;
Vert2d *vert1, *vert2;
Edge2d *edge1, *edge2;
Geom_edge2d *geomlist1, *geomedge1, *geomedge2;
geomlist1 = geomlist->nextgeom_edge2d;
while (geomlist1 != NULL)
        {
        v1 = same_equation(geomlist1, order, numedges);
        if(v1[0] != 0)
        {
        for (i=0; i<v1[0]; i++)
                {
                v = merge_type_test_straight
                (order, edgelist, vertlist,v1[(i+1)]);
                edge1 = edgelist;
                geomedge1 = geomlist;
                printf("%s%d%s%d%s\n","The edges",order, "and",
                        v1[(i+1)], "are conformable for merging");
        printf("Do you want them to merge? Type y or n\n");
                ch = getch();
                if ((v[0] != 0) && (ch == 'y'))
                {
                for(i=0; i<v[2]; i++)
                        {
                        edge1 = edge1->nextedge2d;
                geomedge1 = geomedge1->nextgeom_edge2d;
                        }
                geomedge2 = geomedge1->nextgeom_edge2d;
        geomedge1->nextgeom_edge2d = geomedge2->nextgeom_edge2d;
                edge2 = edge1->nextedge2d;
                edge1->nextedge2d = edge1->nextedge2d->nextedge2d;
                kill_geomedge2d(geomedge2);
                kill_edge2d(edge2);
                vert1 = vertlist->nextvert2d;
```

```
                    for (i=0; i<v[3]; i++) vert1 = vert1->nextvert2d;
                    intlist = vert1->linelist;
                    n2 = members_in_intlist(intlist);
                    vert1 = vertlist->nextvert2d;
                    if (n2 == 1)
                              {
                    for (i=0; i<v[3]-1; i++) vert1 = vert->nextvert2d;
                              vert2 = vert1->nextvert2d;
                    vert1->nextvert2d = vert1->nextvert2d->nextvert2d
                              kill_vert2d(vert2);
                              }
                    else
                              {
                              intlist = remove_node_intlist(intlist, v[2]);
                              }
                    vert1 = vertlist->nextvert2d;
                    for (i=0; i<v[4]-1; i++) vert1 = vert1->nextvert2d;
                    intlist = vert1->linelist;
                    n2 = members_in_intlist(intlist);
                    if (n2 == 1)
                              {
                    for (i=0; i<v[4]-1; i++) vert1 = vertlist->nextvert2d;
                              vert2 = vert1->nextvert2d;
                              vert1->nextvert2d = vert1->nextvert2d-
                                                  nextvert2d;
                              kill_vert2d(vert2);
                              }
                    else
                              {
                              intlist = remove_node_intlist(intlist, v[2]);
                              }
                    }
                    }
          }
          geomlist1 = geomlist1->nextgeom_edge2d;
          free_ivector(v1,0,v1[0]);
          }
}
/*-----------------------------------*/
void comb_zero_dist_merge()
{
```

```
zero_dist_merge(VISGEOM, VISEDGE2D, VISVERT2D, NUMVISEDGE);
if(NUM_HIDPOINTS 0)
zero_dist_merge(HIDGEOM, HIDEDGE2D, HIDVERT2D, NUMHIDEDGE);
if(NUM_CENTPOINTS 0)
zero_dist_merge(CENGEOM, CENEDGE2D, CENVERT2D, NUM-
CENEDGE);
if(NUM_CONSPOINTS 0)
zero_dist_merge(CONSGEOM, CONSEDGE2D, CONSVERT2D, NUMCON-
SEDGES);
}
/*-----------------------------------*/
/*                    same_equation_toerase(geomlist,
                                          erasgeomlist, lastchecked)
This function checks whether the (lastchecked + 1)th node in the
erasgeomlist has any nodes in the list geomlist with same equation. If
none are present a vector with all elements 0 would be returned. If
any are present the first element of the vector would be the number,
the second give the type with the following elements would give their
ranks in geomlist.
*/
int *same_equation_toerase(Geom_edge2d *geomlist,
                           Geom_edge2d *erasgeomlist,
                           int lastchecked)
{
int numsame = 0, ans=0, order, type1, i, geomrank, type2, *v, *v3, numedges=0;
Geom_edge2d *geomlist1, *erasgeomlist1, geomlist2;
float d1,d2,d3,d4,d5,d6,*v1,*v2;
geomlist1 = geomlist->nextgeom_edge2d;
while(geomlist1 != NULL)
        {
        numedges++;
        geomlist1 = geomlist1->nextgeom_edge2d;
        }
geomlist1 = geomlist->nextgeom_edge2d;
erasgeomlist1 = erasgeomlist->nextgeom_edge2d;
v=ivector(0,numedges-1);
for(i=0; i<numedges; i++) v[i] = 0;
for(i=0; i=lastchecked; i++)
erasgeomlist1 = erasgeomlist1->nextgeom_edge2d;
geomrank = 1;
while (geomlist1 != NULL)
```

```
{
type1 = geomlist1->type;
type2 = erasgeomlist->type;
if ((abs(type1)) = = (abs(type2)))
        {
        v1 = geomlist1->v;
        v2 = erasgeomlist->v;
        if ((abs(type1)) 5)
                {
                d1 = fabs(v1[0] - v2[0]);
                d2 = fabs(v1[1] - v2[1]);
                d3 = fabs(v1[2] - v2[0]);
                if(((d1) && (d2)) && (d3)) ans = 1;
                }
        if(abs(type1) 4)
                {
                d1 = fabs(v1[0] - v2[0]);
                d2 = fabs(v1[1] - v2[1]);
                d3 = fabs(v1[2] - v2[2]);
                d4 = fabs(v1[3] - v2[3]);
                d5 = fabs(v1[4] - v2[4]);
                d6 = fabs(v1[5] - v2[5]);
                if((d1) && (d2) && (d3) &&
                (d4) && (d5) && (d6)) ans = 1;
                }
        }
        if(ans = = 1)
                {
                v[numsame] = geomrank;
                numsame + +;
                }
geomlist1 = geomlist1->nextgeom_edge2d;
geomrank + +;
}
if(numsame 0)
{
v3 = ivector(0,numsame);
v3[0] = numsame;
for (i = 0; i < numsame; i + +)
{
        v3[i + 1] = v[i];
```

```
                }
                free_ivector(v,0,numedges);
                return(v3);
                }
else
                {
                return(v);
                }
}
/*-------------------------------*/
int overlap_test_erase(int order, Edge2d *edgetoerase, Edge2d
                                *edgelist, Vert2d *vertlist, Geom_edge2d *geomlist)
{
float x1,y1,x2,y2,x3,y3,x4,y4;
int d1, d2, d3, d4, ans=0, a1, a2;
Edge2d *edge1, *edge2, *edge3, *edgelist1;
Intnode *intlist;
char c;
Geom_edge2d *geomedge1, *geomedge2;
Vert2d *vert1,*vert2, *vertlist1;
int i, n1, n2, n3, n4, test1;
edge1 = edgetoerase;
edge2 = edgelist->nextedge2d;
for(i=0; i<order; i++) edge2 = edge2->nextedge2d;
n1 = edge1->vert1;
n2 = edge1->vert2;
vert1 = ERASVERT2D-nextvert2d;
for (i=0; i<n1; i++) vert1 = vert1->nextvert2d;
x1 = vert1->point[0];
y1 = vert1->point[1];
vert1 = ERASVERT2D->nextvert2d;
for (i=0; i<n2; i++) vert1 = vert1->nextvert2d;
x2 = vert1->point[0];
y2 = vert1->point[1];
n3 = edge2->vert1;
n4 = edge2->vert2;
vert1 = vertlist->nextvert2d;
for (i=0; i<n3; i++) vert1 = vert1->nextvert2d;
x3 = vert1->point[0];
y3 = vert1->point[1];
vert1 = vertlist->nextvert2d;
```

```
for (i=0; i<n4; i++) vert1 = vert1->nextvert2d;
x4 = vert1->point[0];
y4 = vert1->point[1];
if((fabs(x1-x3) < 20) && (fabs(y1-y3) 20)) d1 = 1;
if((fabs(x1-x4) < 20) && (fabs(y1-y4) 20)) d2 = 1;
if((fabs(x2-x3) < 20) && (fabs(y2-y3) 20)) d3 = 1;
if((fabs(x2-x4) < 20) && (fabs(y2-y3) 20)) d4 = 1;
if((d1 = = 1) && (d4 = = 1)) test1 = 1;
if((d2 = = 1) && (d3 = = 1)) test1 = 1;
if(test1 = = 1)
            {
            printf("The line %d",order," is conformable for eraseing!\n");
            printf("Can it be erased? Y or N\n");
            c = getch();
            if (c = = 'y')
                        {
                        edge2 = edgelist->nextedge2d;
                        geomedge1 = geomlist->nextgeom_edge2d;
                        for(i=0; i=order-1; i++)
                                    {
                                    edge2 = edge2->nextedge2d;
                        geomedge1 = geomedge1->nextgeom_edge2d;
                                    }
                        edge3 = edge2->nextedge2d;
                        geomedge2 = geomedge2->nextgeom_edge2d;
                        edge2->nextedge2d = edge3->nextedge2d;
            geomedge1->nextgeom_edge2d = geomedge2->nextgeom_edge2d;
                        a1 = edge3-vert1;
                        a2 = edge3-vert2;
                        vertlist1 = vertlist;
                        for(i=0; i=a1; i++) vertlist1 = vertlist1->nextvert2d;
                        intlist = vertlist1->linelist;
                        if(members_in_intlist(intlist) = = 1)
                                    {
           vertlist1 = vertlist;
                                    for(i=0; i=a1; i++) vertlist1 =
                                    vertlist1->nextvert2d;
                                    kill_vert2d(vertlist1);
                                    }
                        else
                                    {
```

```
                                        intlist = remove_node_intlist(intlist, a1);
                                        }
                        vertlist1 = vertlist;
                        for(i=0; i=a2; i++) vertlist1 = vertlist1-nextvert2d;
                        intlist = vertlist1->linelist;
                        if(members_in_intlist(intlist) == 1)
                                        {
                                        vertlist1 = vertlist;
                                        for(i=0; i=a2; i++) vertlist1 =
                                        vertlist1 = vertlist1->nextvert2d;
                                        kill_vert2d(vertlist1);
                                        }
                        else
                                        {
            intlist = remove_node_intlist(intlist, a2);
                                        }
                        kill_edge2d(edge3);
                        kill_geomedge2d(geomedge2);
                        ans = 0;
                        }
            }
return(ans);
}
/*-------------------------------*/
void erase_merge()
{
int ans = 0, *v, i, j, numerased=0;
Edge2d *erasedge;
Geom_edge2d *erasgeom;
erasedge = ERASEDGE2D-nextedge2d;
while(erasedge != NULL)
            {
            numerased++;
            erasedge = erasedge->nextedge2d;
            }
erasedge = ERASEDGE2D-nextedge2d;
for (i=1; i=numerased; i++)
            {
            v = same_equation_toerase(VISGEOM, ERASGEOM, i);
            if(v[0] != 0)
                        {
```

```
                for(j=0; j<v[0]; j++)
                        {
                    ans = overlap_test_erase(v[j+1], erasedge,
                VISEDGE2D, VISVERT2D, VISGEOM);
                if(ans == 1) break;
                }
                if(ans == 1) break;
                }
v = same_equation_toerase(HIDGEOM, ERASGEOM, i);
if(v[0] != 0)
                {
                for(j=0; j[0]; j++)
{
                ans = overlap_test_erase(v[j+1], erasedge,
                HIDEDGE2D, HIDVERT2D, HIDGEOM);
                if(ans == 1) break;
                        }
                if(ans == 1) break;
                }
v = same_equation_toerase(CENGEOM, ERASGEOM, i);
if(v[0] != 0)
{
for(j=0; j[0]; j++)
                {
                ans = overlap_test_erase(v[j+1], erasedge,
                CENEDGE2D, CENVERT2D, CENGEOM);
                if(ans == 1) break;
                }
                if(ans == 1) break;
                }
v = same_equation_toerase(CONSGEOM, ERASGEOM, i);
if(v[0] != 0)
                {
                for(j=0; j<v[0]; j++)
                        {
                ans = overlap_test_erase(v[j+1], erasedge,
                CONSEDGE2D, CONSVERT2D, CONSGEOM);
                if(ans == 1) break;
                }
                if(ans == 1) break;
                }
```

```
                erasedge = erasedge-nextedge2d;
                ans = 0;
                }
}
/*-----------------------------------*/
copy_solid()
{
int i, numedge = 0, numvert = 0;
Edge2d *newedge, *visedge2d;
Vert2d *newvert, *visvert2d;
Geom_edge2d *newgeom, *visgeom;
visgeom = VISGEOM->nextgeom_edge2d;
while(visgeom != NULL)
                {
                numedge++;
                visgeom = visgeom->nextgeom_edge2d;
                }
while(visvert2d != NULL)
{
                numvert++;
     visvert2d = visvert2d->nextvert2d;
                }
SOLVERT2D = get_vert2d();
visvert2d = VISVERT2D->nextvert2d;
for(i=0; i<numvert; i++)
                {
                newvert = get_vert2d();
                newvert->point = visvert2d->point;
                newvert->linelist = visvert2d->linelist;
                back_of_vert2d(newvert, SOLVERT2D);
                visvert2d = visvert2d->nextvert2d;
                }
visgeom = VISGEOM->nextgeom_edge2d;
visedge2d = VISEDGE2D>-nextedge2d;
SOLEDG2D = get_edge2d();
SOLGEOM = get_geom_edge2d();
newedge = get_edge2d();
newgeom = get_geom_edge2d();
newedge->vert1 = visedge2d->vert1;
newedge->vert2 = visedge2d->vert2;
newedge->edgetype = visedge2d->edgetype;
```

```
visedge2d = visedge2d->nextedge2d;
SOLEDG2D = back_of_edge2d(newedge, SOLEDG2D);
newgeom->v = visgeom->v;
newgeom->type = visgeom->type;
newgeom->meanx = visgeom->meanx;
newgeom->meany = visgeom->meany;
visgeom = visgeom->nextgeom_edge2d;
SOLGEOM = back_of_geom2d(newgeom, SOLGEOM);
NUMVISEDGE = numedge;
NUMVISVERT = numvert;
}
/*-------------------------------*/
int member_vertlist(Vert2d *vertlist, float x, float y)
{
int ans=0;
float d1, d2;
Vert2d *vertlist1;
vertlist1 = vertlist->nextvert2d;
while(vertlist1 != NULL)
        {
        d1 = fabs(x - vertlist1->point[0]);
        d2 = fabs(y - vertlist1->point[1]);
        if((d1  20) && (d2  20)) ans = 1;
        vertlist1 = vertlist1->nextvert2d;
        }
}
/*-------------------------------*/
solid_merge()
{
Edge2d *hidedge2d, *newedge2d;
Vert2d *hidvert2d, *newvert2d, *solvert2d;
Geom_edge2d *hidgeom, *newgeom;
Intnode *linelist, *newintnode;
int hidedge=0, hidvert=0, n1, n2, n, i, j;
float x1, y1, x2, y2, *v;
hidedge2d = HIDEDGE2D->nextedge2d;
hidvert2d = HIDVERT2D->nextvert2d;
while(hidedge2d != NULL)
        {
        hidedge++;
        hidedge2d = hidedge2d->nextedge2d;
```

```
                }
while(hidvert2d ! = NULL)
                {
                hidvert + +;
                hidvert2d = hidvert2d- > nextvert2d;
                }
hidedge2d = HIDEDGE2D- > nextedge2d;
hidvert2d = HIDVERT2D- > nextvert2d;
hidgeom = HIDGEOM-nextgeom_edge2d;
for(i = 0; ihidedge; i + +)
                {
                newedge2d = get_edge2d();
                n1 = hidedge2d-vert1;
                n2 = hidedge2d- > vert2;
                for(j = 0; j 1; j + +) hidvert2d = hidvert2d- > nextvert2d;
                x1 = hidvert2d- > point[0];
                y1 = hidvert2d- > point[1];
                hidvert2d = HIDVERT2D- > nextvert2d;
                for(j = 0; j < 2; j + +) hidvert2d = hidvert2d- > nextvert2d;
                x2 = hidvert2d- > point[0];
                y2 = hidvert2d- > point[1];
                n = member_vertlist(SOLVERT2D, x1, y1);
                if(n = = 0)
                                {
                                newvert2d = get_vert2d();
                                v = vector(0,1);
                                v[0] = x1;
                                v[1] = y1;
                                newvert2d- > point = v;
                                linelist = get_intnode();
                                newintnode = get_intnode();
                                newintnode- > a = NUMVISEDGE + i;
                                back_of_intnode(newintnode, linelist);
                                newvert2d- > linelist = linelist;
                                back_of_vert2d(newvert2d, SOLVERT2D);
                                NUMSOLVERTS + +;
                                }
                else
                                {
                                solvert2d = SOLVERT2D- > nextvert2d;
                                for(j = 0; j < n j + +) solvert2d = solvert2d-nextvert2d;
```

```
                linelist = solvert2d-linelist;
                newintnode = get_intnode();
                newintnode->a = NUMVISEDGE + i;
                back_of_intnode(newintnode, linelist);
                }
newedge2d = get_edge2d();
newedge2d->vert1 = NUMSOLVERTS;
if(n0) newedge2d->vert1 = n;
n = member_vertlist(SOLVERT2D, x2, y2);
if(n==0)
                {
                newvert2d = get_vert2d();
                v = vector(0,1);
                v[0] = x1;
                v[1] = y1;
                newvert2d->point = v;
                linelist = get_intnode();
                newintnode = get_intnode();
                newintnode->a = NUMVISEDGE+i;
                back_of_intnode(newintnode, linelist);
                newvert2d-linelist = linelist;
                back_of_vert2d(newvert2d, SOLVERT2D);
                NUMSOLVERTS++;
                }
else
                {
                solvert2d = SOLVERT2D->nextvert2d;
                for(j=0; j<n; j++) solvert2d = solvert2d->nextvert2d;
                linelist = solvert2d->linelist;
                newintnode = get_intnode();
                newintnode->a = NUMVISEDGE + i;
                back_of_intnode(newintnode, linelist);
                }
newedge2d->vert2 = NUMSOLVERTS;
if(n0) newedge2d->vert2 = n;
newedge2d->edgetype = hidedge2d->edgetype;
back_of_edge2d(newedge2d, SOLEDG2D);
newgeom = get_geom_edge2d();
newgeom->type = hidgeom->type;
newgeom->v = hidgeom-v;
newgeom->meanx = hidgeom->meanx;
```

```
        newgeom->meany = hidgeom->meany;
        back_of_geom2d(newgeom, SOLGEOM);
        hidedge2d = hidedge2d->nextedge2d;
        hidvert2d = hidvert2d>-nextvert2d;
        hidgeom = hidgeom-nextgeom_edge2d;
        }
}
```

```
/*                              proc3d.c
*/
/*-------------------------------*/
/* This function extracts x2, y2, z2, and returns the homogeneous
co-ordinates in v when the line type is 1, 2, or 3. Given are
x1, y1, z1 and dx1 and dy1, the digitizer co-ordinates together
with the line type */
float *solve_simultaneous(type, dx2, dy2, x1, y1, z1)
float  dx2, dy2, x1, y1, z1;
int type;
{
float x2,y2,z2,*v;
v = vector(0,3);
v[3] = 1;
if (type = =1)
            {
            v[0] = x1;
            v[1] = (dx2 - x1*cos(22/21))/(cos(22/21));
            v[2] = dy2 + (x1 - v[1])/2;
            return(v);
            }
if (type= =2)
            {
            v[2] = z1;
            v[1] = 0.5*(2*(dy2-z1)+2*dx2/(sqrt(3)));
            v[0] = v[1]-2*(dy2-z1);
            return(v);
            }
if (type= =3)
            {
            v[1] = y1;
            v[0] = dx2*2/(sqrt(3)) - y1;
            v[2] = dy2 + (v[0] - v[1])/2;
            return(v);
            }
}
/*-------------------------------*/
/* This function extracts the 3D co-ordinates of the node
vertnode using solve_simultaneous and the rhe co-ordinates of the
other point of the straight line refpoint[3]. The assumption is
that the line is of type 1, 2 or 3. */
```

```
float *process_iso_straight(int type, float refpoint[3],
                                        Vert2d *vertnode)
{
float x1, y1, z1, dx2, dy2, *v;
x1 = refpoint[0];
y1 = refpoint[1];
z1 = refpoint[2];
dx2 = vertnode->point[0];
dy2 = vertnode->point[1];
v = solve_simultaneous(type, dx2, dy2, x1, y1, z1);
return(v);
}
/*-----------------------------*/
int intmember(int v[], int c, int length)
{
int i, result;
result=0;
i=0;
do
    {
    if (v[i] == c) result=1;
    i++;
    }
while((result==0) && (i gth));
return(result);
}
/*-----------------------------*/
/*                    extract_initial_3d(edgelist, refpoint[4], ref,
                                        vertlist)
This function establishes the three dimensional co-ordinates
of all the vertices connected to the origin and the isometric
lines emanating from it. This will establish many of the
co-ordinates in the sketch. It first establishes the three
dimensional lists of vertices of edges and vertices. The origin
is first used to establish the 3D co-ordinates of the connected
vertices. Then the edgelist in 2D is taken in order and checked
with the vector PROCEDGES and any not in that are processed
wherever possible */
extract_initial_3d(Edge2d *edgelist, float refpoint[4], int ref,
                                        Vert2d *vertlist)
{
```

```
int n1, n2, n3, n4, n5, n6, i, j, finvert, l, l1, l2;
int e1 = 0, e2 = 0;
float *v, *v1;
Intnode *lines, *lines1;
Edge2d *edgelist1;
Vert2d *vertlist1, *finvert2d;
Vert3d *vert3d1, *newvert3d;
Edge3d *edge3d1, *newedge3d;
VERTLIST = get_vert3d();
EDGELIST = get_edge3d();
vert3d1 = VERTLIST;
edge3d1 = EDGELIST;
for(i = 0; i < NUMSOLVERTS; i + +)
        {
        newvert3d = get_vert3d();
        VERTLIST = back_of_vert3d(newvert3d, VERTLIST);
        }
for(i = 0; i < NUMSOLEDGES; i + +)
        {
        newedge3d = get_edge3d();
        EDGELIST = back_of_edge3d(newedge3d, EDGELIST);
        }
edgelist1 = edgelist-nextedge2d;
vertlist1 = vertlist-nextvert2d;
PROCEDGES = ivector(0,(NUMSOLEDGES-1));
PROCVERTS = ivector(0,(NUMSOLVERTS-1));
for (i = 0; i < NUMSOLEDGES; i + +) PROCEDGES[i] = 0;
for (i = 0; i < NUMSOLVERTS; i + +) PROCVERTS[i] = 0;
PROCVERTS[e2] = ref;
REFPOINT[0] = 0;
REFPOINT[1] = 0;
REFPOINT[2] = 0;
REFPOINT[3] = 1;
e2 + +;
for(i = 0; i < ref; i + +) vertlist1 = vertlist1-nextvert2d;
lines = vertlist1-linelist;
vert3d1 = VERTLIST-nextvert3d;
vert3d1->point[0] = 0;
vert3d1->point[1] = 0;
vert3d1->point[2] = 0;
vert3d1->point[3] = 1;
```

```
vert3d1->linelist = lines;
lines1 = lines->nextintnode;
vert3d1 = VERTLIST;
while (lines1 != NULL)
            {
            j++;
            lines1 = lines1->nextintnode;
            }
lines1 = lines;
n5 = j;
for (i=0; i<5; i++)
            {
            lines1 = lines1->nextintnode;
            n6 = lines1->a;
            PROCEDGES[e1] = n6;
            e1++;
            for (i=0; i 6; i++) edgelist1 = edgelist1->nextedge2d;
            n1 = edgelist1->vert1;
            n2 = edgelist1->vert2;
            n3 = edgelist1->edgetype;
            finvert = n1;
            if (n1 == ref) finvert = n2;
            for (i=0; i<finvert; i++)
                        {
                        vertlist1 = vertlist1->nextvert2d;
                        }
            finvert2d = vertlist1;
            vertlist1 = vertlist->nextvert2d;
            if (abs(n3) < 4)
                        {
                        v=process_iso_straight(n3, REFPOINT, finvert2d);
                        for (i=0; i<n3; i++) vert3d1 = vert3d1-nextvert3d;
                        vert3d1->point[0] = v[0];
                        vert3d1->point[1] = v[1];
                        vert3d1->point[2] = v[2];
                        vert3d1->point[3] = v[3];
                        vert3d1->linelist = finvert2d->linelist;
                        vert3d1 = VERTLIST;
                        }
            }
v = vector(0,3);
```

```
edgelist1 = edgelist-nextedge2d;
for (i=1; i=NUMSOLEDGES; i++)
        {
        l = intmember(PROCEDGES,i, NUMSOLEDGES);
        edgelist1 = edgelist1->nextedge2d;
        if((l == 0) && (edgelist1->edgetype < 4))
                {
                n1 = edgelist1->vert1;
                n2 = edgelist1->vert2;
                n3 = edgelist1->edgetype;
                }
        l1 = intmember(PROCVERTS, n1, NUMSOLVERTS);
        l2 = intmember(PROCVERTS, n2, NUMSOLVERTS);
        if ((l1 == 1) && (l2 == 0))
        {
        e1++;
        PROCEDGES[e1] = i;
        vert3d1 = VERTLIST;
        vertlist1 = vertlist;
        for (i=0; i=n1; i++)
                {
                vert3d1 = vert3d1->nextvert3d;
                vertlist1 = vertlist1->nextvert2d;
                }
        v[0] = vert3d1->point[0];
        v[1] = vert3d1->point[1];
        v[2] = vert3d1->point[2];
        v[3] = vert3d1->point[3];
        v1=process_iso_straight(n3, v, vertlist1);
        vert3d1 = VERTLIST;
        for(i=0; i 2; i++) vert3d1 = vert3d1->nextvert3d;
        vert3d1->point[0] = v1[0];
        vert3d1->point[1] = v1[1];
        vert3d1->point[2] = v1[2];
        vert3d1->point[3] = v1[3];
        }
        if ((l1 ==0) && (l2 == 1))
        {
        e1++;
        PROCEDGES[e1] = i;
        vert3d1 = VERTLIST;
```

```
            vertlist1 = vertlist;
            for (i=0; i=n2; i++);
                        {
                        vert3d1 = vert3d1->nextvert3d;
                        vertlist1 = vertlist1->nextvert2d;
                        }
            v[0] = vert3d1->point[0];
            v[1] = vert3d1->point[1];
            v[2] = vert3d1->point[2];
            v[3] = vert3d1->point[3];
            v1=process_iso_straight(n3, v, vertlist1);
            vert3d1 = VERTLIST;
            for (i=0; i=n1; i++) vert3d1 = vert3d1->nextvert3d;
            vert3d1->point[0] = v1[0];
            vert3d1->point[1] = v1[1];
            vert3d1->point[2] = v1[2];
            vert3d1->point[3] = v1[3];
            }
            }
}
/*-------------------------------*/
void process_construction()
{
Edge3d *newedge, *consedge3d;
Vert3d *newvert, *consvert3d, *solvert3d1;
Edge2d *consedge2d1, *soledge2d1;
Vert2d *consvert2d1, *solvert2d1;
Geom_edge2d *consgeom2d1, *solgeom;
int i, j=0, k, l ,n, n1, n2, m1, m2, type;
float x1, y1, x2, y2, x3, y3, x4, y4, p1, *v, *v1, *v2, r;
PROCCONSVERT = ivector(0, NUMCONSVERTS-1);
PROCCONSEDGE = ivector(0, NUMCONSEDGES-1);
CONSEDGE3D = get_edge3d();
CONSVERT3D = get_vert3d();
consvert3d = CONSVERT3D;
for(i=0; i<NUMCONSEDGES; i++)
  {
  newedge = get_edge3d();
  CONSEDGE3D = back_of_edge3d(newedge, CONSEDGE3D);
  PROCCONSEDGE[i] = 0;
  }
```

```
for(i=0; i<NUMCONSVERTS; i++)
  {
  newvert = get_vert3d();
  CONSVERT3D = back_of_vert3d(newvert, CONSVERT3D);
  PROCCONSVERT[i] = 0;
  }
consvert2d1 = SOLCONSVERT2D->nextvert2d;
consedge2d1 = SOLCONSEDGE2D->nextedge2d;
consgeom2d1 = SOLCONSGEOM->nextgeom_edge2d;
CONSEDGE3D = CONSEDGE3D->nextedge3d;
solgeom = SOLGEOM;
while(consedge2d1 != NULL)
  {
  n = consedge2d1->edgetype;
  if(abs(n)< 4)
            {
            n1 = consedge2d1->vert1;
            n2 = consedge2d1->vert2;
            m1 = intmember(PROCCONSVERT, n1, NUMCONSVERTS);
            m2 = intmember(PROCCONSVERT, n2, NUMCONSVERTS);
            if((m1 == 1) && (m2 == 1)) break;
            if((m1 == 0) && (m2 == 0))
    {
            for(i=1; i=n1; i++) consvert2d1 = consvert2d1->nextvert2d;
            x1 = consvert2d1->point[0];
            y1 = consvert2d1->point[1];
            consvert2d1 = SOLCONSVERT2D->nextvert2d;
            for(i=1; i=n2; i++) consvert2d1 = consvert2d1->nextvert2d;
            x2 = consvert2d1->point[0];
            y2 = consvert2d1->point[1];
            for(i=0; i<NUMSOLEDGES; i++)
                        {
                        solgeom = solgeom->nextgeom_edge2d;
                        j++;
                        v1 = solgeom->v;
                        p1 = v1[0]*y1 + v1[1]*x1 +v1[2];
                        r = sqrt(v1[0]*v1[0] + v[1]*v[1]);
                        p1 = fabs(p1/r);
                        if((fabs(p1) 20) && (solgeom->type < 4)) break;
                        }
    if(p1 20)
```

```
                    {
                    for(i=0; i=j; i++)
                                {
                                soledge2d1 = soledge2d1->nextedge2d;
                                }
                    k = soledge2d1->vert1;
                    for(i=0; i=k; i++)
                                {
                                solvert2d1 = solvert2d1->nextvert2d;
                                }
                    v = solvert3d1->point;
          for(l=0; l=n1; l++) consvert2d1 = consvert2d1->nextvert2d;
                    v2 = process_iso_straight(type, v, consvert2d1);
                    for(l=0; l=n1; l++)
                                {
                                consvert3d = consvert3d->nextvert3d;
                                }
                    consvert3d-point[0] = v2[0];
                    consvert3d-point[1] = v2[1];
                    consvert3d-point[2] = v2[2];
                    consvert3d-point[3] = v2[3];
                    }
          else
          {
          for(i=0; i<NUMSOLEDGES; i++)
                    {
                    solgeom = solgeom->nextgeom_edge2d;
                    j++;
                    v1 = solgeom-v;
                    p1 = v1[0]*y2 + v1[1]*x2 +v1[2];
                    r = sqrt(v1[0]*v1[0] + v[1]*v[1]);
                    p1 = fabs(p1/r);
                    if((fabs(p1)  20) && (solgeom->type < 4)) break;
                    }
if(p1  20)
                    {
                    for(i=0; i=j; i++)
                                {
                                soledge2d1 = soledge2d1->nextedge2d;
                                }
                    k = soledge2d1->vert1;
```

```
                    for(i=0; i=k; i++)
                                {
                                solvert3d1 = solvert3d1->nextvert3d;
                                }
                    v = solvert3d1->point;
          for(l=0; l=n1; l++) consvert2d1 = consvert2d1->nextvert2d;
                    v2 = process_iso_straight(type, v, consvert2d1);
                    for(l=0; l=n1; l++)
                                {
                                consvert3d = consvert3d->nextvert3d;
                                }
                    consvert3d->point[0] = v2[0];
                    consvert3d->point[1] = v2[1];
                    consvert3d->point[2] = v2[2];
                    consvert3d->point[3] = v2[3];
                    consvert3d->linelist = consvert2d1->linelist;
                    }
          }
          consvert3d = CONSVERT3D;
          }
if((m1 == 1) && (m2 == 0))
          {
          for(l=0; l=n1; l++)
                    consvert3d = consvert3d->nextvert3d;
    v = consvert3d->point;
          consvert2d1 = CONSVERT2D;
          for(l=0; l=n2; l++)
                    consvert2d1 = consvert2d1->nextvert2d;
          type = consedge2d1->edgetype;
          v2 = process_iso_straight(type, v, consvert2d1);
          consvert3d = CONSVERT3D;
          for(l=0; l=n2; l++)
                    consvert3d = consvert3d->nextvert3d;
          consvert3d->point[0] = v2[0];
          consvert3d-point[1] = v2[1];
          consvert3d-point[2] = v2[2];
          consvert3d-point[3] = v2[3];
          consvert3d-linelist = consvert2d1->linelist;
          }
if((m1 == 0) && (m2 == 1))
          {
```

```
            for(l=0; l=n2; l++)
                        consvert3d = consvert3d->nextvert3d;
    v = consvert3d->point;
            consvert2d1 = CONSVERT2D;
            for(l=0; l=n1; l++)
                        consvert2d1 = consvert2d1->nextvert2d;
            type = consedge2d1->edgetype;
            v2 = process_iso_straight(type, v, consvert2d1);
            consvert3d = CONSVERT3D;
            for(l=0; l=n1; l++)
                        consvert3d = consvert3d->nextvert3d;
            consvert3d->point[0] = v2[0];
            consvert3d->point[1] = v2[1];
            consvert3d->point[2] = v2[2];
            consvert3d->point[3] = v2[3];
            consvert3d->linelist = consvert2d1->linelist;
            }
            n1 = CONSEDGE3D->vert1;
            n2 = CONSEDGE3D->vert2;
            CONSEDGE3D = CONSEDGE3D->nextedge3d;
            consedge2d1 = consedge2d1->nextedge2d;
            }
}
}
/*-----------------------------------*/
float *endpoints(Edge3d *edge)
{
int i, n1, n2;
Vert3d *vertlist;
float *endpoints;
endpoints = vector(0,5);
n1 = edge->vert1;
n2 = edge->vert2;
vertlist = VERTLIST;
for(i=0; i=n1; i++)
            vertlist = vertlist-nextvert3d;
endpoints[0] = vertlist-point[0];
endpoints[1] = vertlist-point[1];
endpoints[2] = vertlist-point[2];
vertlist = VERTLIST;
for(i=0; i=n1; i++)
```

```
                vertlist = vertlist-nextvert3d;
endpoints[3] = vertlist-point[0];
endpoints[4] = vertlist-point[1];
endpoints[5] = vertlist-point[2];
return(endpoints);
}
/*-----------------------------*/
float *reverse_position_vector(float v[])
{
float *position;
position = vector(0,2);
position[0] = v[0] - v[3];
position[1] = v[1] - v[4];
position[2] = v[2] - v[5];
return(position);
}
/*-----------------------------*/
float *position_vector(float v[])
{
float *position;
position = vector(0,2);
position[0] = v[3] - v[0];
position[1] = v[4] - v[1];
position[2] = v[5] - v[2];
return(position);
}
/*-------------------------------*/
int equality(float v1[], float v2[], int size)
{
int i, ans = 0;
for(i = 0; i < size; i + +)
{
if(v1[i] - v2[i]  0.01) break;
}
if(i = = size-1) ans = 1;
return(ans);
}
/*-------------------------------*/
float *normal_vector(float v1[], float v2[])
{
float *norm;
```

```
norm = vector(0,2);
norm[0] = v1[1]*v2[2] - v1[2]*v2[1];
norm[1] = v1[2]*v2[0] - v1[0]*v2[2];
norm[2] = v1[0]*v2[1] - v1[1]*v2[0];
return(norm);
}
/*-------------------------------*/
extract_loops()
{
int *v;
Loop3d *looplist, *newloop, *trialloop, *looplist1;
Edge3d *edgelist, *firstedge, *nextedge;
Intnode *linelist, *newintnode, *linelist2;
char c, givenloop[30],str[3];
float *norm1, *norm2, *endpoints1, endpoints2, *pos1, *pos2, *refnorm;
float *revpos, *endpoint;
Vert2d *vert2d;
int i=0, edgecount=0, a, fe, ne, te, j=0, elements, vert, ans;
linelist = get_intnode();
looplist = get_loop3d();
looplist1 = looplist;
newloop = get_loop3d();
v = ivector(0,(2*NUMSOLEDGES -1));
printf("Please enter the first clockwise loop line numbers and space");
gets(givenloop);
c=givenloop[i];
while(c != '\n')
            {
            if(c! = ' ')
            {
            str[j] = c;
            j++;
            }
    else
            {
            str[j] = '\0';
            j=0;
            a = atoi(str);
            newintnode = get_intnode();
            newintnode->a = a;
            back_of_intnode(newintnode, linelist);
```

```
            v[edgecount] = a;
            edgecount++;
            }
            i++;
            c = givenloop[i];
            }
newloop->linelist = linelist;
trialloop = looplist1->nextloop3d;
while(trialloop != NULL)
            {
            linelist = trialloop-linelist;
            elements = members_in_intlist(linelist);
            while((linelist != NULL) && (elements == 3))
            {
            linelist = linelist->nextintnode;
            fe = linelist->a;
            ne = linelist->nextintnode->a;
            edgelist = EDGELIST;
            for(i=0; i=fe; i++) edgelist = edgelist->nextedge3d;
            endpoint = endpoints(edgelist);
            pos1 = position_vector(endpoint);
            edgelist = EDGELIST;
            for(i=0; i=ne; i++) edgelist = edgelist->nextedge3d;
            endpoints1 = endpoints(edgelist);
            pos2 = position_vector(endpoints1);
            refnorm = normal_vector(pos1, pos2);
            ne = fe;
            do
                        {
                        edgelist = EDGELIST;
                        for(i=0; i=ne; i++) edgelist = edgelist->nextedge3d;
        endpoint = endpoints(edgelist);
                        vert = edgelist->vert1;
                        revpos = reverse_position_vector(endpoint);
                        vert2d = SOLVERT2D;
                        for(i=0; i=vert; i++) vert2d = vert2d->nextvert2d;
                        linelist2 = vert2d-linelist;
                        do
                        {
                        linelist2 = linelist2->nextintnode;
                        te = linelist2-a;
```

```
                    edgelist = EDGELIST;
                    for(i=0; i=te; i++) edgelist = edgelist->nextedge3d;
                    endpoint = endpoints(edgelist);
                    pos1 = position_vector(endpoint);
                    norm1 = normal_vector(revpos, pos1);
                    ans = equality(norm1, refnorm, 3);
                    }
                    while (ans == 1);
            ans = 1;
            if (te != fe)
            {
            v[edgecount] = te;
            edgecount++;
            }
            ne = te;
            }
            while(ne == fe);
            linelist = linelist->nextintnode;
            }
trialloop = looplist1->nextloop3d;
}
}
/*-----------------------------*/
int *get_loop_vertexlist(Intnode *linelist)
{
int i, ans=0, vertcount, n=0, n1, n2, *v;
Intnode *linelist1;
Edge3d *edge3d;
linelist1 = linelist-nextintnode;
while(linelist1 != NULL)
            {
            n++;
            linelist1 = linelist1-nextintnode;
            }
v = ivector(0,n-1);
for(i=0; i ; i++) v[i] = 0;
linelist1 = linelist;
while(linelist1 != NULL)
            {
            linelist1 = linelist1-nextintnode;
            edge3d = EDGELIST;
```

```
                for(i=0; i=linelist1-a; i++) edge3d = edge3d-nextedge3d;
                n1 = edge3d-vert1;
                n2 = edge3d-vert2;
                ans = intmember(v, n1, n);
                if(ans==1) v[vertcount] = n1;
                ans = 0;
                vertcount++;
                }
return(v);
}
/*-----------------------------*/
/* This algorithm fits an equation of a plane of the form
[a b c d].[x y z 1] = 0
*/
float *fit_face(Intnode *linelist)
{
float yizi=0, yiziplus1=0, yiplus1zi=0, yiplus1ziplus1=0;
float zixi=0, zixiplus1=0, ziplus1xi=0, ziplus1xiplus1=0;
float xiyi=0, xiyiplus1=0, xiplus1yi=0, xiplus1yiplus1=0;
float sumx=0, sumy=0, sumz=0;
float xav, yav, zav;
float *v;
float x, y, z, x1, y1, z1, x0, y0, z0;
int i, j, n=0, *v1;
Vert3d *vertlist1;
Intnode *linelist1;
v1 = get_loop_vertexlist(linelist);
linelist1 = linelist->nextintnode;
while(linelist1 != NULL)
    {
                n++;
                linelist1 = linelist1->nextintnode;
                }
for(i=0; i<n; i++)
                {
                vertlist1 = VERTLIST;
                for(j=0; j=v[i]; j++) vertlist1 = vertlist1->nextvert3d;
                x1 = vertlist1->point[0];
                y1 = vertlist1->point[1];
                z1 = vertlist1->point[2];
                if(i==0)
```

```
                {
                x0=x1;
                y0=y1;
                z0=z1;
                x = x1;
                y = y1;
                z = z1;
                }
if(i>0)
{
yizi = yizi + y*x;
yiziplus1 = yiziplus1 + y*z1;
yiplus1zi = yiplus1zi + y1*z;
yiplus1ziplus1 = yiplus1ziplus1 + y1*z1;
zixi = zixi + z*x;
zixiplus1 = zixiplus1 + z*x1;
ziplus1xi= ziplus1xi + z1*x;
ziplus1xiplus1 = ziplus1xiplus1 + z1*x1;
xiyi = xiyi + x*y;
xiyiplus1 = xiyiplus1 + x*y1;
xiplus1yi = xiplus1yi + x1*y;
xiplus1yiplus1 = xiplus1yiplus1 + x1*y1;
sumx = sumx + x1;
sumy = sumy + y1;
sumz = sumz + z1;
x = x1;
y = y1;
z = z1;
}
}
if(i==n-1)
{
x1 = x0;
y1 = y0;
z1 = z0;
yizi = yizi + y*x;
yiziplus1 = yiziplus1 + y*z1;
yiplus1zi = yiplus1zi + y1*z;
yiplus1ziplus1 = yiplus1ziplus1 + y1*z1;
zixi = zixi + z*x;
zixiplus1 = zixiplus1 + z*x1;
```

```
            ziplus1xi = ziplus1xi + z1*x;
            ziplus1xiplus1 = ziplus1xiplus1 + z1*x1;
            xiyi = xiyi + x*y;
            xiyiplus1 = xiyiplus1 + x*y1;
            xiplus1yi = xiplus1yi + x1*y;
            xiplus1yiplus1 = xiplus1yiplus1 + x1*y1;
            }
            v = vector(0,3);
            v[0] = yizi + yiziplus1 - yiplus1zi - yiplus1ziplus1;
            v[1] = zixi + zixiplus1 - ziplus1xi - ziplus1xiplus1;
            v[2] = xiyi + xiyiplus1 - xiplus1yi - xiplus1yiplus1;
            xav = sumx/n;
            yav = sumx/n;
            zav = sumx/n;
            v[3] = -(xav*v[0] + yav*v[1] + zav*v[2]);
            return(v);
}
/*-----------------------------------*/
fit_3dgeometry()
{
Loop3d *looplist1, *looplist2;
Intnode *linelist, *linelist2;
int linecount, arccount, ellipsecount, n, type, i;
float *v;
Edge3d *edgelist1;
edgelist1 = EDGELIST;
FACELIST = get_face3d();
looplist1 = LOOPLIST->nextloop3d;
NUM_LOOP =0;
while (looplist1 != NULL)
            {
            NUM_LOOP++;
            linelist = looplist1->linelist;
            linelist = linelist->nextintnode;
            linecount = 0;
            arccount = 0;
            ellipsecount = 0;
            while(linelist != NULL)
                        {
                        n = linelist->a;
                        for(i=0; i=n; i++) edgelist1 = edgelist1->nextedge3d;
```

```
                    type = edgelist1->linetype;
                    if (type < 5) linecount++;
                    if(type > 8) arccount++;
                    linelist = linelist->nextintnode;
                    }
          linelist2 = looplist1->linelist;
          if((linecount < 2) && (arccount < 2))
                    {
                    v = fit_face(linelist2);
                    looplist1->v = v;
                    }
          looplist1 = looplist1->nextloop3d;
          }
}/*-----------------------------*/
```

APPENDIX C

OUTPUTS OF 'L' BLOCK SKETCH

THE POINTS FILE

37,725	The origin in three and two dimensions
389	Number of points in the visible lines
0	Number of points in the hidden lines
0	Number of points in the centre lines
0	Number of points in the construction lines
0	Number of points in the erased lines
34,719	Start of the 389 points
33,743	
31,767	
29,787	
29,810	
30,831	
31,856	
34,882	
35,912	
35,933	
36,956	
37,976	
36,997	
37,1019	
39,1050	
39,1079	
39,1107	
40,1128	
40,1148	
43,1172	
42,1192	
43,1218	
44,1239	
45,1259	
66,1264	
85,1246	
104,1236	
122,1221	
144,1208	

162,1198
180,1184
204,1172
223,1160
243,1146
262,1137
281,1128
276,1107
276,1082
276,1061
277,1034
275,1011
290,991
308,980
323,965
340,953
356,939
376,926
401,916
418,903
442,890
462,883
484,875
504,867
520,852
516,830
514,801
515,772
515,752
513,728
509,701
510,672
508,650
506,630
506,608
507,587
506,563
507,541
506,519
506,498
508,478

509,457
508,436
46,719
67,705
84,690
103,677
119,660
134,644
151,633
171,618
194,606
212,596
237,580
256,573
275,564
297,553
316,546
337,536
357,526
376,514
396,501
416,489
439,476
462,465
481,452
504,441
522,450
546,464
564,476
592,488
611,502
629,512
655,528
677,540
697,555
720,568
752,587
775,599
799,621
826,635
849,646

867,657
888,669
911,682
930,693
960,704
978,720
1006,732
1028,746
1058,757
1081,771
1099,780
1117,790
1136,802
1159,813
1176,825
1193,837
1212,846
1223,1268
1223,1244
1223,1213
1226,1188
1225,1160
1225,1130
1225,1106
1227,1077
1228,1054
1227,1026
1227,1003
1226,973
1225,947
1224,925
1222,903
1217,881
1217,860
1214,839
514,858
540,873
565,884
594,899
613,912
639,927

658,938
683,952
711,967
728,982
750,993
773,1009
796,1024
819,1042
843,1054
865,1067
887,1082
908,1094
924,1106
947,1119
970,1133
998,1144
1018,1156
1043,1166
1065,1182
1089,1193
1110,1202
1131,1217
1152,1231
1172,1241
1194,1255
1214,1262
989,1410
1010,1400
1031,1384
1052,1368
1069,1355
1086,1344
1109,1334
1127,1321
1146,1309
1164,1299
1182,1284
1204,1274
1220,1262
284,1000
314,1017

341,1036
361,1043
383,1058
407,1067
429,1084
463,1098
485,1115
504,1130
528,1143
548,1155
577,1175
594,1186
619,1202
651,1219
669,1235
693,1243
714,1256
737,1267
757,1281
786,1295
812,1305
833,1321
857,1332
874,1343
895,1351
918,1359
935,1376
953,1386
971,1397
988,1408
998,1427
994,1448
990,1468
986,1494
984,1517
982,1538
274,1134
303,1147
330,1168
353,1182
375,1193

393,1203
419,1217
436,1229
464,1242
480,1254
503,1262
531,1287
553,1293
577,1310
611,1328
634,1344
662,1356
679,1372
706,1384
724,1397
747,1410
769,1422
785,1435
812,1446
834,1458
858,1473
875,1486
899,1497
919,1505
938,1519
959,1532
978,1543
763,1684
781,1673
802,1659
823,1650
848,1636
871,1619
892,1605
914,1593
929,1579
944,1565
963,1553
983,1540
761,1684
740,1674

720,1661
690,1647
670,1634
642,1617
621,1606
597,1595
579,1583
555,1562
528,1546
509,1536
483,1524
464,1508
438,1498
419,1491
394,1476
372,1465
352,1450
330,1442
310,1428
292,1416
271,1402
247,1390
225,1378
203,1360
176,1350
150,1337
130,1327
108,1311
89,1302
69,1293
50,1279
759,1680
759,1658
761,1634
760,1610
759,1587
761,1565
761,1545
759,1524
759,1504
757,1480

756,1456
757,1433
757,1413
758,1386
760,1363
761,1340
762,1320
763,1296
763,1271
763,1251
763,1228
761,1204
759,1180
757,1160
757,1138
44,725
64,733
83,749
110,763
128,774
147,783
169,792
188,804
207,816
225,829
252,839
271,849
296,864
314,873
333,887
360,900
376,915
406,929
424,942
449,952
472,972
494,984
512,994
537,1007
558,1025
576,1034

594,1044
619,1055
637,1065
658,1079
680,1089
702,1100
720,1114
738,1126
758,1127
774,1115
795,1101
816,1091
838,1080
857,1072
873,1058
893,1047
915,1038
932,1026
951,1018
969,1006
986,995
1010,984
1027,971
1048,958
1068,945
1088,930
1110,916
1128,901
1149,888
1168,878
1184,865
1206,856

LBLOCK.ONE FILE

37.00 725.00	Three and two dimensional origin
389	Number of points in visible lines
0	Number of points in hidden lines
0	Number of points in centre lines
0	Number of points in construction lines
0	Number of points in erased lines
18	Number of edges
12	Number of vertices
23 1 1 34.00 719.00 45.00 1259.00	First line segment
12 24 1 45.00 1259.00 281.00 1128.00	
6 36 1 281.00 1128.00 290.00 991.00	
12 42 1 290.00 991.00 520.00 852.00	
18 54 1 520.00 852.00 508.00 436.00	
23 73 1 46.00 719.00 504.00 441.00	
32 96 1 504.00 441.00 1212.00 846.00	
17 129 1 1223.00 1268.00 1214.00 839.00	
31 147 1 514.00 858.00 1214.00 1262.00	
12 179 1 989.00 1410.00 1220.00 1262.00	
31 192 1 284.00 1000.00 988.00 1408.00	
6 223 1 988.00 1408.00 982.00 1538.00	
31 230 1 274.00 1134.00 978.00 1543.00	
11 262 1 763.00 1684.00 983.00 1540.00	
32 274 1 761.00 1684.00 50.00 1279.00	
24 307 1 759.00 1680.00 757.00 1138.00	
33 332 1 44.00 725.00 738.00 1126.00	
24 365 1 738.00 1126.00 1206.00 856.00	
0.00 -1.00 36.35 36.35 0.00	Geometry of the first edge
-1.00 -0.58 1291.15 153.33 1202.58	
0.00 -1.00 276.83 276.83 0.00	
-1.00 -0.58 1150.43 392.00 924.00	
0.00 -1.00 510.06 510.06 0.00	
-1.00 -0.58 728.09 257.65 579.26	
-1.00 0.58 150.56 853.28 643.44	
0.00 -1.00 1224.12 1224.12 0.00	
-1.00 0.58 563.87 861.26 1061.35	
-1.00 -0.58 1974.07 1097.42 1340.17	
-1.00 0.58 838.22 641.68 1208.87	
0.00 -1.00 990.00 990.00 0.00	

-1.00 0.58 976.70 625.87 1338.23
-1.00 -0.58 2120.08 866.36 1619.64
-1.00 0.58 1247.07 410.44 1484.16
0.00 -1.00 759.79 759.79 0.00
-1.00 0.58 696.94 381.97 917.58
-1.00 -0.58 1561.04 961.46 1005.67
1 2 1 Firsr edge connecting vertex 1 and 2 of type 1
2 3 3 Second edge connecting vertex 2 and 3 of type 3
3 4 1
4 5 3
5 6 1
1 6 3
6 7 2
8 7 1
5 8 2
9 8 3
4 9 2
9 10 1
3 10 2
11 10 3
11 2 2
11 12 1
1 12 2
12 7 3
34.00 719.00 1 6 17 First terminal point common to lines 1 6 17
45.00 1259.00 1 2 15 Second terminal point common to lines 1 2 15
281.00 1128.00 2 3 13
290.00 991.00 3 4 11
520.00 852.00 4 5 9
508.00 436.00 5 6 7
1212.00 846.00 7 8 18
1223.00 1268.00 8 9 10
989.00 1410.00 10 11 12
982.00 1538.00 12 13 14
763.00 1684.00 14 15 16
757.00 1138.00 16 17 18
32.59 712.51 1 6 17 First vertex common to lines 1 6 17
37.07 1269.11 1 2 15
274.97 1133.93 2 3 13
274.20 994.32 3 4 11
509.13 857.15 4 5 9

506.00 439.32 5 6 7
1222.84 855.80 7 8 18
1222.74 1268.97 8 9 10
987.28 1406.14 10 11 12
989.88 1548.39 12 13 14
758.14 1683.57 14 15 16
755.06 1128.99 16 17 18

LBLOCK.TWO FILE

37.00 725.00	Three and two dimensional origin
389	Number of points in visible lines
0	Number of points in hidden lines
0	Number of points in centre lines
0	Number of points in construction lines
0	Number of points in erased lines
18	Number of edges
12	Number of vertices
1 2 1	Firsr edge connecting vertex 1 and 2 of type 1
2 3 3	Second edge connecting vertex 2 and 3 of type 3
3 4 1	
4 5 3	
5 6 1	
1 6 3	
6 7 2	
8 7 1	
5 8 2	
9 8 3	
4 9 2	
9 10 1	
3 10 2	
11 10 3	
11 2 2	
11 12 1	
1 12 2	
12 7 3	
0.00 -1.00 36.35 36.35 0.00	Geometry of the first edge
-1.00 -0.58 1291.15 153.33 1202.58	
0.00 -1.00 276.83 276.83 0.00	
-1.00 -0.58 1150.43 392.00 924.00	
0.00 -1.00 510.06 510.06 0.00	
-1.00 -0.58 728.09 257.65 579.26	
-1.00 0.58 150.56 853.28 643.44	
0.00 -1.00 1224.12 1224.12 0.00	
-1.00 0.58 563.87 861.26 1061.35	
-1.00 -0.58 1974.07 1097.42 1340.17	
-1.00 0.58 838.22 641.68 1208.87	
0.00 -1.00 990.00 990.00 0.00	

-1.00 0.58 976.70 625.87 1338.23
-1.00 -0.58 2120.08 866.36 1619.64
-1.00 0.58 1247.07 410.44 1484.16
0.00 -1.00 759.79 759.79 0.00
-1.00 0.58 696.94 381.97 917.58
-1.00 -0.58 1561.04 961.46 1005.67
32.59 712.51 1 6 17 First vertex common to lines 1 6 17
37.07 1269.11 1 2 15
274.97 1133.93 2 3 13
274.20 994.32 3 4 11
509.13 857.15 4 5 9
506.00 439.32 5 6 7
1222.84 855.80 7 8 18
1222.74 1268.97 8 9 10
987.28 1406.14 10 11 12
989.88 1548.39 12 13 14
758.14 1683.57 14 15 16
755.06 1128.99 16 17 18