The Nottingham Trent University
Library & Information Services

SHORT LOAN COLLECTION

Date	Time	Date	Time
23 MAY			

Please return this item to the Issuing Library.
Fines are payable for late return.

THIS ITEM MAY NOT BE RENEWED

Short Loan Coll May 1996

40 0681771 7

ProQuest Number: 10183061

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10183061

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

"This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent."

Data Structures for The Reconstruction of Engineering Drawings.

M. Waite

ABSTRACT.

An extension to the Region finding algorithm presented in [68] is described which constructs a description of the topology of a drawing with a single pass through the vector file of the drawing. The description constructed by the algorithm is a development of the Modified Wing-edged data structure described in [112]. The data structure presented here has two main components:- a series of directed graphs, each of which describes a set of connected vectors in the drawing; and a Containment hierarchy representing the positional relationships between the series of discrete graphs.

The intended use of the algorithm is as an intermediate stage in constructing three-dimensional representations of objects from digitised three-view engineering projections. A semi-automatic engineering drawing interpreter is presented which is based on the system developed by Aldefeld[2,3,4], but which uses the new algorithm to produce the drawing data structures and so decrease the constraints on the data format of the input drawing. It is envisaged that this new algorithm will enable the interpreter to work with the loosely structured, inaccurate and incomplete drawing data derived from scanned images of drawings.

ERRATA.

The following is a list of corrections to errors which have been found in the text.

Page 14, paragraph 2, line 2 :-
´boundary-representations´ should read ´Wire-frames´.

Page 16, paragraph 1, line 3 :-
´itermediate´ should read ´intermediate´.

Page 24, paragraph 3, line 1 :-
´independant´ should read ´independent´.

Page 33, paragraph 2, line 2 :-
´are´ should read ´is´.

Page 61, the label of the fourth entry in the line list :-
´L#EF´ should read ´L#FE´.

CPHD 5

Data Structures for The Reconstruction of Engineering Drawings.

Martin Waite B.A.(Hons)

Thesis submitted to the Council for National Academic Awards
in partial fulfilment of the requirements for degree of
Doctor of Philosophy.

Trent Polytechnic
Department of Computing
in collaboration with
PAFEC Ltd., Strelley Hall, Nottingham.

June 1989

Abstract 1

Chapter 1 -Interpreting Engineering Drawings 2
1.1 Introduction. 3
1.2 Existing Engineering Drawing Interpreters. 10
1.3 The extent of progress towards fully automatic interpretation. 24
1.4 The extent of progress towards a practically applicable engineering drawing interpreter. 26
1.5 The aims of this project.

Chapter 2 -Interpretation of an engineering drawing - a tutorial. 30
2.1 Uniform thickness objects. 33
2.2 Interpretation of the three-view engineering drawing. 35
2.3 Identifying the projections of uniform thickness objects. 42
2.4 False components may be identified. 44
2.5 An outline of the system which has been developed. 48

Chapter 3 -A Region Analysis Program. 50
3.1 Information requirements of the interpreter. 52
3.2 Data Structures which supply information to the interpreter. 54
3.3 An Example of the representation. 57
3.4 Plane-sweep algorithms. 62
3.5 Extending the Region-finding algorithm enabling it to cope with general vertex types. 69
3.6 An outline of the algorithm. 71
3.7 Extensions required for coping with vertical lines. 93
3.8 Enabling the algorithm to construct an Adjacency Graph. 108
3.9 Constructing the Containment Hierarchy. 129

Chapter 4 -The Reconstruction Process. 152
4.1 Introduction. 153
4.2 The user and the Reconstruction program. 153
4.3 Correspondences between views. 156
4.4 Mapping three-dimensional co-ordinates onto wo-dimensional drawing co-ordinates. 165
4.5 The Reconstruction Procedures. 168
4.6 Constructing solids. 185
4.7 Some examples of reconstruction. 188

Chapter 5 -Conclusion. 193
5.1 An evaluation of the Region Analysis program. 194
5.2 Areas for future development of the Region Analysis program. 195
5.3 An evaluation of the Reconstruction program. 197
5.4 Areas for future development of the Reconstruction program. 197
5.5 Areas for future research in the field of Engineering drawing interpretation. 199

References. 204

Appendices - Pseudo-code outlines of the Region Analysis program.
Appendix A - Version which builds Region-Lists from Vertices. A- 2
Appendix B - Vertex-based version which copes with Vertical Lines. A-13
Appendix C - Edge-based version which builds an Adjacency Graph. A-27
Appendix D - Edge-based version which builds a Containment Hierarchy. A-38

Dedication :

For Margaret and John, my mother and father.

Acknowledgements:

Gratitude is due to my supervisors Dr R.J. Whitrow and Dr P.D. Thomas of Trent Polytechnic for their patience, direction and advice throughout the past four years. Acknowledgements are also due to PAFEC Ltd, Strelley Hall, Nottingham for financing this project, and especially to Dr. Alan Austin of said company for his active participation and support.

I would like to take this opportunity to properly thank Nasser Sherkat and Imelda Whelehan for their invaluable help and encouragement during my time at Nottingham. Sincere thanks also to Graham Lerant, Sally Golding and Winston Golding for their contributions in Bristol. Mention must also be made of Mark Hammes, Graham Hainesworth, Nicola Hainesworth, Lisa Welbourne, Gwen, Margaret and all the other Staff at Trent Polytechnic who helped to make my time there so enjoyable.

Data Structures for The Reconstruction of Engineering Drawings.

M. Waite

ABSTRACT.

An extension to the Region finding algorithm presented in [68] is described which constructs a description of the topology of a drawing with a single pass through the vector file of the drawing. The description constructed by the algorithm is a development of the Modified Wing-edged data structure described in [112]. The data structure presented here has two main components:- a series of directed graphs, each of which describes a set of connected vectors in the drawing; and a Containment hierarchy representing the positional relationships between the series of discrete graphs.

The intended use of the algorithm is as an intermediate stage in constructing three-dimensional representations of objects from digitised three-view engineering projections. A semi-automatic engineering drawing interpreter is presented which is based on the system developed by Aldefeld[2,3,4], but which uses the new algorithm to produce the drawing data structures and so decrease the constraints on the data format of the input drawing. It is envisaged that this new algorithm will enable the interpreter to work with the loosely structured, inaccurate and incomplete drawing data derived from scanned images of drawings.

CHAPTER 1.

1. Interpreting Engineering Drawings.

1.1 Introduction.

Engineering drawings are representations of the design of objects. The engineering drawing contains a stylised graphical representation of one or more related parallel projections of an object or assembly, drawn with regard to the mathematics of projection [17] and also to the accepted rules of draftsmanship [42]. Extra data in the form of text and symbols is woven around the graphical representation giving details of design decisions such as tolerances, materials, and finishes, and also detailing the relationship between separate drawing sheets and versions of drawings[34,43]. Significantly, some text and symbols are used as abbreviations for common entities which should, following the rules of projection, have appeared in the graphic but were omitted for the sake of clarity and time [18,44].

In principle, the representation of engineering drawings is a simple matter. Engineering drawings are made from collections of curve segments and text configured in two dimensional space. All that is required to represent a drawing is to create data entities which describe the types and positions of curves, and the font, size, orientation and location of any text[18]. In theory it is not difficult to write computer programs that can be used to create, store and retrieve such artifacts, although such programs tend to be large and suffer the concomitant problems of being difficult to design, verify and maintain. Managing the size and complexity of large computer systems is a field of research in itself which has yielded many different design methodologies. These tend to be classified as hierarchical top-down techniques or modular bottom-up techniques, the latter coming into favour in the late 1980's with the emergence of object-oriented design techniques, tools and languages [60,93], one influence from the artificial intelligence community anticipated by Sandewall[85].

Following the pioneering work of Sutherland[105] in developing the first useful computer-assisted drawing facility, a whole industry has grown around the development of computer systems for creating, storing, manipulating and displaying drawings. As the technology for storing and displaying drawings advanced, the design emphasis for such systems moved to increasing the throughput of designers by improving the communication

between the designer and the computer system. This emphasis is typified by attempts to build some design skills into the computer-system [111], making it co-operate with the designer as an assistant rather than as a tool [20,56]. Increasing the communication between designer and machine has lead to some interesting attempts at using free-hand sketches as an input medium [44,49,64,89].

Engineering drawings have started to receive attention as a possible input medium for Computer-aided Design and Draughting systems. At the simplest level, this may involve placing the sheet containing the drawing on top of a digitising tablet and identifying the type and the end points of curves using a stylus. Such an approach obviously involves a great deal of repetitious action from the operator, leading to boredom, fatigue and errors. A more promising approach is to use a camera or digitising-scanner to produce an image of the drawing which can be stored in computer memory. Unfortunately, camera derived images are usually in a format which are incompatible with CAD systems. The camera produces an array of values representing the brightness or colour of a corresponding rectangle on the drawing, whereas most CAD systems store the drawing in vector format. Camera images have to be converted into the vector representations before they can be stored and manipulated using a CAD package. Converting a camera or scanned image into a vector image has received considerable attention, not only for its application as an engineering drawing input facility, but also in general as a precursor to image understanding in the field of computer vision[7,14,88].

Commonly the first stage of this conversion process is a thresholding process which maps the grey-scale images produced by the camera onto a binary image of black and white pixels[117]. Line-following procedures then hunt across the image array looking for pixel-patterns which might correspond to lines, arcs, alpha-numeric characters or other symbols. Literature in this area might fairly be divided into that concerned with extracting graphical elements [9,12,22,24,32,65,72,74,80], that concerned with text extraction [71,89,107], and that concerned with symbol extraction [16,88,106]. This two stage process is a popular approach to vectorisation, but is not universal. Some novel apparatus have been described [6,30] which operate directly on the paper drawing, following lines with lasers and optical sensors and so by-passing the requirement to store and process a camera derived image.

Extracting vector and text data from camera images is an important part in the development of new man-machine interfaces for CAD systems, but the graphical primitives and symbols obtained from this process only yields a low level of information about the drawing. The next stage is to construct a topological description of the drawing which provides a framework for representing the interaction and relative positioning between primitives and the more complex shapes that these interactions produce. These concerns were originally attended by artificial intelligence researchers in the early 1970's with the aim of constructing descriptions of drawings expressed in natural language in an attempt to mimic human geometric reasoning [7,23,59]. Lately, these concerns are being addressed by a field of computing science called 'Computational Geometry' which is interested in how geometric data is obtained, stored and manipulated in order to provide solutions to practical problems such as VLSI design and air-traffic control. Results to date have included some intuitively appealing data structures and algorithms. Data structures such as the 'Hammock' [19] and the Voronoi diagram [27,35] have been created along with algorithms to construct and manipulate them which can be used to efficiently answer queries on the relative positioning between entities. One significant algorithm is the plane sweep algorithm which has been variously applied to the efficient reporting of intersections between line-segments [10,11], and to windowing and clipping algorithms for general geometric figures in computer graphics[26]. An interesting enhancement of the plane-sweep algorithm was used to generate descriptions of the regions between line-segments in a restricted drawing[68]. A further development of this algorithm is presented in chapter three of this thesis, allowing a description of the regions in any line-drawing to be constructed.

The man-machine interface can be improved further by imbuing the machine with an expert's understanding of the physical laws which apply to the artifacts being designed and by enabling the machine to construct a three-dimensional model of the objects being designed from its communication with the designer. The task of engineering drawing interpretation is to construct a representation of the intention of the designer from whatever channels of communication the designer chooses to use. This requires that CAD systems become expert systems. They must be able to acquire and store knowledge of their domain of usefulness, this requiring new techniques of knowledge representation [91] to be

laid on top of geometric representation capabilities. The machine could then perform an important role in the loop between creativity and analysis[111] by performing much of the routine analysis involved in design. Warman isolated four computationally intensive design activities which can be assigned to computers:-

Reduction - simplification which retains functionality
Simulation - testing design functionality using models
Optimisation - balancing economics against functionality
Modularisation - deriving re-usable sub-units from the design.

To perform these roles in engineering and architectural design applications, the machine must be able to construct the three dimensional representation of the design in order to analyse it. In electrical or electronic design applications a similar assisting role could be played by a machine with an understanding of the logical design of the circuit in terms of connected components. Machine understanding of what is being designed provides a qualitative improvement over machine understanding of the arrangement of graphic primitives which constitute a diagram of the design.

In engineering applications, understanding designs means understanding the configuration of the components being designed, the materials from which they are made, how they are made, the dynamic interactions between such components and the interactions between components and environment. Examining only the first aspect, understanding the configuration of components requires data structures to be constructed which can represent the geometry of solid objects. Allen's paper [5] provides an overview of the solid modelling techniques currently in use in engineering CAD systems. He divides solid models into three categories, all three allowing quantitative analysis of the described object to be performed[15,31,81]. These three categories describe solids as:-

* combinations of solid primitives - commonly applied in Constructive Solid Geometry systems [113,116], which are primarily useful for designs which can be described in terms of objects being welded, glued together or having other shapes drilled or cut out of them.

* topologically related elementary surfaces - called the boundary representation of an object[8,112]. This has advantages over the Constructive Solid Geometry representation in that sculpted surfaces such as car body shells can be described. The boundary representation is also used in medical applications[115].

* constructional operations performed on two-dimensional shapes - originated in computer vision research [1], where the model is called the 'Generalised Cylinder' representation [14]. Useful in Engineering design for describing extrusions or other types of prismatic objects which appear in many engineering artifacts.

The ability to internally model solids provides an important level of functionality for CAD systems, but our main concern here is with constructing these internal models from the channel of communication between designer and machine. This requires that CAD systems exhibit some basic human abilities of pattern recognition, the ability to determine the similarities and dissimilarities between one image and another. This would enable graphical queries to be made upon the system's memory, finding previous designs similar to the current one, recognising hand-written text and symbols. Unfortunately this is one area where theory is presently thin. Pavel made the surprising statement that shape theory as such does not exist [73] and that this is to blame for the lack of generalised pattern recognition systems. Her argument can be more clearly understood when one attempts to construct purely syntactic or structural descriptions of images, two approaches which have been followed in attempts to provide generalised pattern recognition facilities[29,70]. Deriving structural descriptions of connected components is simply a matter of defining line types and angles between these lines., yielding a rotation and size independent description of an outline[58,61,62]. Further, it is possible describe the hierarchic nesting between such outlines and retain independence from size, orientation and position. However, no further information can be given. Relating adjacent outlines causes problems - saying even that one shape is northwest or upper-left from another suddenly causes problems. Distinctions such as Left-of, Above, Below and Northwest-of place restrictions on the orientation of the picture. These relations change as the alignment of the picture with the observer change and so rotational independence has been lost.

Constructing grammatical, syntactic or structural descriptions of an image which may allow general parsing techniques to be applied to image matching seems doomed to failure. At one point, scene analysis seemed to promise solutions to the problem of image understanding. In 1968 Guzman appeared to have solved a fundamental problem in vision by building descriptions of three-dimensional scenes by assigning convex or concave roles to edges in the image[36,37,38]. Waltz developed this further, enumerating all possible junction types in such a drawing[108,109]. Unfortunately this approach, which is elegant and easily manageable for plane-faced objects[84,92] of an "Origami World"[50,51], becomes complicated when applied to curved surfaces because of the relaxation of the rigidity of the constraints which guide the interpreter. Lines can no longer be assumed to have the same convexity/concavity assignment running their entire length, a requirement central to this approach to interpretation. Kokichi Sugihara has devoted many years pursuing this approach to image understanding applying it to data obtained from drawings and range-finders[94-103], and managing to apply it to objects with curved surfaces. This approach is however limited to interpreting axiometric and perspective views and therefore does not apply to orthographic projections.

Similar problems in producing generalised object descriptions beset the field of model-matching. Model-matching is the process of comparing a stored template against the image data using some criteria of 'likeness' or 'unlikeness'. In order to be effective, model-matching must have some level of independence from size and orientation constraints[110]. Measures of likeness must be made on shape-derived criteria else slight changes in size of the object owing to perspective diminution would prevent two instances of the object from being recognised. Suggestions of criteria which might be used to determine the likeness of an image instance to an image model include number of crossings[90], measures of symmetry [69], region adjacencies [104], junction types[87], relations between features[63,66,75,86], and similarities of grammatical descriptions[33]. Multi-view models [53], bearing some similarity to flat cardboard cut-outs which can be folded into three-dimensional objects, have been proposed which may eventually provide model-matching processes independent from orientation in three- dimensional space.

Approaches to provide working engineering drawing interpretation systems to date have

tended to avoid most of the profound problems facing computer vision researchers, taking advantage of the fact that engineering drawings tend to describe simpler objects than those encountered in in the vision domain. Techniques applied are generally constructive, bottom-up approaches - first extracting lines and symbols, then finding the loops of connected lines, then constructing surfaces and then solid objects. Section 1.2 of this chapter outlines these approaches in more detail.

1.2 Existing Engineering Drawing Interpreters.

1.2.1 A practical application of a digitising scanner.

One of the earliest attempts at constructing an engineering drawing interpreter was presented in some papers by Idesawa and his associates [47,48]. These papers outline the design of a scanner for the input of drawings into a computer, and complement this with a sample application for their scanner, a program which interprets three-view drawings of polyhedral objects. The program has five stages :-

a) Tidying up the drawing data. The drawing features are rotated so that they register squarely with the drawing axis, compensating for any mis-alignment between views which may be caused by inaccurately loading the paper into the scanner.

b) Calculation of three-dimensional vertices. These are a subset of the Cartesian product of the X, Y and Z components of the XY, XZ, and YZ views.

c) Construction of three-dimensional lines. The projections of lines in each view are fitted with three-dimensional vertices from the set generated in (b).

d) Elimination of 'Ghost figures'. Three-dimensional lines which do not have projections in each view are eliminated.

e) Construction of faces. The three-dimensional line set is allocated to the boundaries of the faces in the drawing. Allocation is constrained by the rules of planar geometry: all edges bounding a single face must be co-planar; all edges must be the intersection of two plane faces.

The program was not meant to be comprehensive, more a demonstration of the potential of using drawings as medium for communicating with computers. No attempt was made to deal with the projections of hidden parts, nor to interpret drawings of objects with curved surfaces.

This program identified one of the major difficulties in reconstructing objects from engineering drawings, that of overcoming the ambiguity inherent in the drawings. Each

vertex and edge in a projection can represent any number of vertices and edges of the object which lie perpendicular to each other in relation to the line of sight of the projection. This is the cause of the "ghost figures" which Idesawa's algorithm generates with its simple combinatorial approach to constructing three-dimensional vertices and surfaces. Idesawa's approach to the elimination of "ghost figures" is to generate all possible surfaces, both actual and ghost, and to discard those which fail a series of validation tests. Surfaces which pass the tests are then composed into an object. The validation tests check that the surface is planar and that the surface can be back-projected onto all of the projections given in the drawing.

This approach to "ghost figure" elimination is effective for simple drawings, but could prove prohibitively expensive when the reconstruction produces a large number of candidate surfaces or when the drawing contains a large element of ambiguity, as occurs when hidden detail is shown in drawings of complex objects, and the projections of the details overlap and intersect each other producing spurious surface outlines.

The three main limitations of Idesawa's program have been the focus of most of the subsequent research efforts in the area of the interpretation of engineering drawings.

1) The system requires the user to identify closed loops in the drawing and also to identify correspondences between vertices in each view. This dependence on the user must be reduced to make the system practically applicable as an interface to a drawing system.

2) Idesawa's approach to the "ghost figure" problem is computationally expensive, does not cater for hidden detail and does not eliminate the more subtle "ghost figures" which might be constructed from some projections.

3) Idesawa's program cannot interpret projections of objects with curved surfaces. Curved surfaces increase the ambiguity in the drawing because not all the vertices and edges which occur in the projections of curved surfaces actually occur in the surfaces themselves. For example, the projection of a cylinder might show straight edges meeting a circle: the edges are projections of the curved surface, the 'horizon' of the curved surface, and so do not represent edges of the cylinder at all; similarly, the vertices where the edges meet the circle do not exist because the edges themselves do not exist.

1.2.2 Constructing simple solids from Three view drawings.

A cruder version of Idesawa's system was presented in 1982 by Kimura et al.[52]. The system constructs three-dimensional vertices from the views, allocates these vertices to lines in the views to create three-dimensional lines and then performs a tree-search to construct three-dimensional surfaces from loops of three-dimensional lines. The system makes no attempt to detect "ghost figures", although the authors acknowledge this as an area for future research.

1.2.3 Composing objects from candidate surfaces with a Theorem Prover.

Giles Lafue's program ORTHO [54] generates large sets of three-dimensional faces from the vertices and edges constructed from the two or three orthographic views: these sets often contain some "false" faces resulting from coincidental alignments of edges in the orthographic projections of the object. The most important property of ORTHO is its ability to identify and eliminate the "false" or "ghost" faces by the application of a theorem-proving technique. The set of generated surfaces is structured as a set of mutually dependent hypotheses to which a theorem-proving algorithm is applied. Discovering all the consistent hypotheses also discovers all the consistent sets of surfaces which compose valid objects.

The set of all possible three-dimensional surfaces is generated using the same method as Idesawa. These surfaces are structured around the edges and vertices of the drawing. Each edge and vertex contains a list of the candidate three-dimensional surfaces which contain that edge or vertex as part of their boundaries. These lists are called "Syntactic-sets" or "S-sets". A square surface would appear in the S-set of four edges and four vertices.

Initially, all the surfaces are labelled as being in an undefined or unmarked state. The theorem-prover then proceeds to label the unmarked surfaces as being either "real" or "ghost", attempting to construct systems of labellings which are internally consistent and which satisfy the topological rules of plane-faced solids. Internal consistency requires that a surface marked as "real" in one Edge is "real" in all the other Edges in which it appears. Topological rules insist that an edge is the meeting place of an even number of

surfaces, and that each vertex is the meeting place of three or more surfaces. Both sets of constraint are locally based enabling solutions to be constructed without performing any global analysis of the object being constructed which might prove to be both complex and expensive.

The constraints imposed by these rules can be seen to propagate quite quickly once assertions are made as to which surfaces actually exist, and so false hypotheses which lead to contradictions are soon identified. A contradiction might be that after a certain number of assertions, one edge contains only "ghost" surfaces, or that another edge contains only one "real" surface and all its other surfaces have been marked as "ghosts", or that a vertex contains only two "real" surfaces. Once a contradiction has been reached by the theorem-prover, it must backtrack to the previous assertion and attempt to replace it with a new assertion, typically reversing the recent "real/ghost" assertion and transferring it to some other unmarked surface belonging to that edge. If no new assertion can be made to replace the previous incorrect one, then the theorem-prover must backtrack further. If no further backtracking can be done, then the object is irresolvable.

ORTHO provides improvements upon Idesawa's program in reducing the dependence on the user in the input stage. The user is no longer required to relate features between views. ORTHO matches vertices in one view with those in the others, and provides a degree of tolerencing to absolve digitisation errors. ORTHO still has some dependence upon the user for low-level interpretation of the input data, requiring explicit identification of all the closed loops in the drawing. Overlapping loops must also be identified and isolated from each other.

The main advantage ORTHO has over Idesawa's program is in the elimination of "ghost" surfaces. Here, using topological rules, ORTHO eliminates subtle ghosts which would confuse Idesawa's program, however the "ghost figure" elimination process is still based on the expensive generate and test paradigm, and nothing has been done to cope with hidden detail or curved surfaces.

1.2.4 Constructing solids from Wire-frames: constructing Wire-frames from Engineering Drawings.

Another treatment of the problem of interpreting engineering drawings evolved from an algorithm intended to derive volumetric descriptions of solids from the descriptions of the boundaries of solids given by three-dimensional wire-frames of edges and vertices.

The motivation behind the wire-frame algorithm was to enable volumetric descriptions of solids to be constructed from boundary-representations, which are typically smaller and cheaper to store than volumetric descriptions. This would enable volumetric analysis to be performed on solids cheaply stored in boundary-representation formats. This algorithm was described by Wesley and Markowsky in their paper "Fleshing out wire frames" [57].

The algorithm can construct every possible object which fits a given three-dimensional wire frame. The algorithm is built on established topological foundations and so can be relied upon to construct all possible solutions to a given wire-frame, to the extent that it can produce solutions to pathological cases where the objects constructed are not physically realisable.

The data set input to the program describes a three-dimensional wire-frame. In order to make the program interpret three-view engineering drawings, an extension is applied which first constructs a wire-frame from the drawing. This extension is described in "Fleshing out projections" [114].

A new level of ambiguity is introduced to the problem by attempting to construct a wire-frame from a set of engineering projections. The wire-frame constructed from the drawing might contain edges and vertices which do not exist in the object: these 'virtual' edges and vertices could cause the wire-frame to become contradictory and prevent any solutions from being found, or they could increase the ambiguity of the wire-frame and cause a large increase in the set of feasible solutions. The wire-frame which is constructed from the projections is therefore not assumed to be correct and so the constraints on the solution imposed by the wire-frame are relaxed. New stages are added to the Wire-frame

algorithm which evaluate whether the wire-frame is correct and which amend the wire-frame as errors are found.

This algorithm eliminates all "ghost figures" except those caused by genuine ambiguity in the drawing which can not be resolved without more contextual knowledge about the object being constructed. A further advance on previous interpretation programs is the reduction in the dependence upon the user in the input stage. The algorithm finds all the closed loops in the drawing data, and structures these loops into faces according to the containment relationships between them.

Curved surfaces are not dealt with in detail in these papers, although the authors do actually claim that the new levels of ambiguity introduced by curved surfaces can be coped with by adding some extensions to their algorithms. Curved surfaces increase the disparity between the features of the projection of the object and the features of the wire-frame of the object. This disparity would have to be bridged, probably by searching through the range of possible wire frames fitting a given set of projections. This process would be computationally expensive, especially considering that each wire frame itself has its own, possibly large, space of object solutions.

1.2.5 An Engineering Drawing Interpreter as a CAD system interface.

Kenneth Preiss' engineering drawing interpretation program[76,77,78] shows a similar reduction in the dependence upon the user in the input stage as Wesley and Markowsky's. The program identifies the correspondence between projections of vertices in each view, and also the closed loops of edges which describe the projections of surfaces. This shift in the onus of interpretation from the user to the machine, is very important considering Preiss' goal of producing a practical interface module to a CAD system for the input of engineering drawings and of transforming data from two-dimensional CAD systems into a form suitable for use by solid modelling systems.

Preiss' algorithm for producing three-dimensional surfaces is similar in operation to Lafue's theorem-prover "ORTHO" [54]. Surfaces are generated and validated using topological rules which can be applied locally at edges and vertices. An improvement in

solution time is gained by using a heuristic to structure the search path, although the value of this is unlikely to be significant because the time saving only applies to the generation of surfaces. This is only an itermediate stage in the overall reconstruction process, and it has no interaction with the next stage which assembles the solid from the three-dimensional surfaces. In effect, the construction of candidate surfaces may have improved, but the "ghost" surface problem must be dealt with by the process which composes the surfaces into objects. This provides no advance on Lafue's solution.

1.2.6 Interpreting Engineering Drawings is a Consistent Labelling Problem.

Haralick and Queeney[41] examine the problem of deriving a solid from orthographic views as a constraint propagation problem[39,40]. This is a class of problems which address the assignment of values to systems of variables wherein each assignment somehow constrains the range of values which can be assigned to all the other variables in the system. Consistent systems are found by searching through all the possible sets of assignments, hopefully using some heuristic or pruning method to contain the size of the search.

An interpretation of an Engineering Drawing is a consistent system. The system is a set of initially unassigned or unlabelled variables. These variables correspond to the 2d vertices, lines and closed loops of lines. To produce an interpretation, each variable must be assigned a value or set of values which correspond to 3d vertices, lines and surfaces. 3d vertices assigned to 2d vertices must be consistent with the views of the drawing and so each 3d vertex assigned must project onto a line or vertex in each view of the drawing. 3d lines must also back-project onto every view in the drawing. The 3d lines used to construct a surface must all be co-planar.

Applying these rules yields a set of candidate planar surfaces which back-project onto every view of the drawing. The rules applied so far are all locally based in that the constraints governing the assignment of labels to variables are all specified within the drawing system. A set of global constraints is used to govern the final stage of the object construction. Surfaces are selected from all the candidates in attempts to construct topologically valid objects. The global constraints demand the every edge is the meeting point of two surfaces in different planes, and each vertex is the meeting point of three or

more surfaces. Surface compositions are arranged according to these constraints, and any complete compositions which satisfy these rules are valid object interpretations.

A similar paper by Er[28] constructs the object from the drawing in the same way, but uses the 'Dual Space' of the object, first used by Huffman[45,46] and Draper[25] in scene analysis problems, to simplify the validation of the construction.

These papers do not enhance the techniques of engineering drawing interpretation, but rather establishes the problem as one of constraint propagation similar to Waltz's junction labelling prolem in interpreting scenes of polyhedra [109].

1.2.7 Interpreting Engineering Drawings containing arcs and circles.

Woo and Hammer[118] in 1977 presented the first system capable of interpreting engineering drawings of objects incorporating simple circular surfaces. The system copes with only very simple curved surfaces. A circle is recognised in one view and is assumed to project onto a straight-line segment in the other views. Such matches enable the section of cylinders to be constructed. Obviously this approach is limited to cylinders lying orthogonal to the line of sight of the projection.

A more capable system was presented by Sakurai and Gossard[83]. Although limited to circles and arcs, the program demonstrates a significant increase in the range of objects which can be constructed. This increase in scope is facilitated by the identification of new types of vertices and edges - "p-vertices", "c-vertices" and "c-edges".

The two-dimensional "p-vertices" include all the vertices in the drawing data, plus some which are injected into the drawing at the far-left, far-right, top and bottom turning points of arcs, and also at the meeting points of edges with differing curves. The three-dimensional "c-vertices" of the object are composed by superimposing the "p-vertices" from each view with the projections of other views. The three-dimensional "c-edges", which include curved edges, are constructed from the "c-vertices" using a set of rules to identify silhouettes and tangency-lines.

Following this, a similar process to that described by Wesley and Markowsky [57,114] identifies the faces and surface projections of the object, and then constructs solid blocks

which are arranged into compositions which might yield valid solid objects.

The demonstration is limited to objects which can be described by straight lines, circles and arcs, although in theory this could be extended to other types of curve by extending the set of rules used to direct the interpreter.

The major restriction in the applicability of this system is that it can only cope with curved surfaces whose axis is parallel to one of the drawing axes. This restriction in the orientation of the interpretable objects is significant in that it is the only currently identified obstacle inthe way of producing an automatic engineering drawing interpreter suitable for a reasonable range of real applications. This restriction is imposed to make the task of identifying curved surfaces possible: in the restricted orientation allowed by this system, it is relatively easy to produce a set of rules which identify the correspondences between curves and line-segments in the projections. For example, the correspondences between the projections of a cylinder could be that of rectangles in one view to a circle in a third view. Altering the orientation of the cylinder would produce a more ambiguous description. For example in one view it might look like an ellipse connected to two straight edges connected to an elliptical arc. Projections in other views would be equally convoluted.

1.2.8 Interpreting Engineering Drawings which include Ellipses.

Preiss [79] increases the range of curves which can be handled by an interpreter to include ellipses and elliptical arcs.

All the possible legal combinations of vertices, lines and curves which could appear in a system of orthogonal views of objects comprising plane and cylindrical faces are enumerated. This set of legal combinations imposes a set of constraints which guide an enhanced interpretation mechanism capable of reconstructing objects synthesised from cylindrical and plane-faces. Although the range of objects which can be interpreted is increased to include those with elliptical surfaces, the restrictions on the orientation of the object with regard to the line of sight of the projections is the same as in Sakurai and Gossard's system.

1.2.9 A quicker method of interpreting objects with curved surfaces.

Lequette [55] presented a modified version of the wire-frame based approach to engineering drawing interpretation [57,83,114]. The main differences between this version and that described by Sakurai and Gossard was in its method of finding the "tangent" edges where surfaces with different curvatures meet. Lequette found these tangent edges while attempting to fit surfaces to the wire-frame rather than inferring their existence directly from the drawing views as performed in [83].

The surface fitting routine itself was similar to that developed by Sakurai and Gossard. Pairs of edges at each vertex were examined and the surface type between them established - planar, cylindrical, spherical or conical. The surface was then verified by traversing the edges describing the outline of the surface on the wire-frame, ensuring that the edges of the outline could be fitted to the proposed surface. A refinement to the verification procedure enabled tangent edges to be identified. If at one vertex of an edge no other edge could be found which lay inside the current surface, but another edge did exist which lay in a surface tangential to the current surface, then that vertex was recorded as possibly lying on a tangent edge. Once all the edges of a surface had been explored, all the recorded tangent vertices were examined to see if tangent edges could be fitted through them.

The main benefit in constructing tangent edges during the surface fitting process rather than earlier was that the initial wire-frame would be smaller and contain fewer spurious edges, resulting possibly in a quicker interpretation.

1.2.10 Building the interpretation from solid components.

Aldefeld[2,3,4] presents a new approach to drawing interpretation which constructs solid sub-parts directly from the drawing rather than follow the path of constructing three-dimensional vertices, lines and surfaces pursued in all the other systems described previously.

All prismatic objects which have a constant cross-section perpendicular to their axis can be described as a lamina swept through a third dimension. Aldefeld's system constructs such prismatic objects directly from the drawing by finding the outlines of cross-sections and then finding the length of the axis in the other views. This approach copes with any shape of cross-section, including arbitrary curves. Many engineering drawings comprise a synthesis of such objects: Aldefeld's program decomposes these into sub-parts, reconstructs these independently and then recombines them to form the object in the drawing.

The outlines of cross-sections always appear in a drawing as a set of one or more connected or contained closed loops of curve segments, straight or circular. Each of these loops correspond to the projection of a surface or part of a surface. Pattern finders search the drawing for closed loops, attempting to construct the outlines of cross-sections of feasible components from them.

The simplest type of component to identify is the uniform thickness object. These appear in engineering drawings, when projected from a favourable line sight, as rectangles in two views and as the shape of the cross-section in the remaining view. The cross-section may be any shape - square, rectangular, circular or an irregular synthesis of line and/or arc segments. The pattern finders for uniform thickness objects search through the views attempting to construct rectangles. Finding a pair of correspondingly sized and positioned rectangles in two views would indicate the likelihood of a component's existence. Searching through the third view, the pattern finders attempt to construct a cross-section of size and position appropriate to the two rectangles. In the actual system, the pattern finders worked the other way around. One of the "pattern finders" is the system's user who identifies the cross-section. The automatic pattern finders then locate the corresponding rectangles.

Aldefeld suggested that another class of objects for which pattern finders could be built might be that of objects with rotational symmetry. The pattern finders would look for circles in one view, and the rotated irregular section in the other views.

A construction process builds components from the uniform thickness objects identified by the pattern finders. All the components constructed from the drawing are combined to form feasible objects. This process simply assigns labels to the components declaring them solid or space. The combination process is guided by evaluating the progress of the construction. The evaluator back-projects the constructed object over the drawing and assesses whether they correspond. The composition process is similar to that applied in the other systems described previously.

The unique selling point of Aldefeld's program lies in the way it constructs solid components directly from the two dimensional drawing. The usual approach is to construct three-dimensional surfaces first and then compose these surfaces into solid components, an approach that runs into difficulty when the components incorporate curved surfaces. Edges and vertices become ambiguous requiring an large increase in the intelligence of the surface construction procedure compared to that required to construct planar surfaces. Aldefeld avoids these problems by not constructing surfaces at all, and so the components constructed by his system can incorporate any surface type - as long as the component is an uniform thickness object. Although limited to this sub-set of all object types, the system is capable of interpreting a large set of practical engineering artifacts with which would present difficulties to systems based on the surface construction approach.

There are three drawbacks to this system :-

* First, this system relies upon the user to identify the combinations of closed loops which comprise the cross-section of a prism. No suggestion is made as to how the spurious loops created by the projection of overlapping components might be automatically eliminated.

* Second, this system relies upon the axis of the uniform thickness objects being parallel to the drawing axis. Any unfavourable orientation would cause the shapes of the cross-sections and rectangles to become truncated, distorted and skewed, making the components difficult to construct properly. This problem is not unique to Aldefeld's program - Sakurai and Gossard's system [83] suffers from the same limitation.

* Third, this system copes with only a restricted set of object types - those constructed from combinations of prismatic components. Obviously this range could be expanded by extending the pattern finding and object constructing processes, but even so there would always be the possibility that some object type would be encountered which would not fit into the devised schema.

1.2.11 Sliced prismatic shapes yield larger object range.

The range of objects which can be represented using the swept cross-section construction is increased by allowing 'Cutting planes' to be incorporated into the object description[21]. Specifying one or more arbitrary slices through the prismatic object constructed using swept cross-sections allows complex objects to be described. This 'cutting' process is also a representative of how the object might actually be made by machining a part, such as machining cast parts and slicing extruded parts at angles. Describing such parts in terms of simple component geometries can be considerably more expensive than describing in terms of a simple solid which has been cut in certain places.

The system described by Cheng et al [21]. consisted of three processes:

* drawing decomposition - the separate views in the drawing are identified and the drawing is simplified by removing arcs, circles and their corresponding 'horizon' edges. Straight line-segments are inserted to connect the 'hanging' line-segments at either end of the removed segment. Following simplification, the drawing consists of only straight line-segments and describes a plane-faced polyhedron. A three-dimensional representation of this polyhedron is constructed by sweeping the outline of its cross-section through a third dimension.

* sub-part reconstruction - The three-dimensional polyhedron is projected back onto the drawing. The areas where the drawing and the back-projection differ define spurious sub-parts which must be added to or subtracted from the polyhedron to produce a representation of the original object. These sub-parts include all the parts of the object with curved surfaces. These sub-parts are constructed by applying a plane-cutting algorithm.

* sub-part composition - A final process composes the generated sub-parts into what Chen called a 'volume enclosure relational quasi-tree' describing the total object in terms of sub-parts being WITHIN, ON, IN and OUTSIDE other sub-parts. An algorithm for converting this relational description into a CSG tree was described.

The simplication stage introduced in this algorithm significantly reduces the amount of work that the sub-part identification and reconstruction processes have to perform. In Aldefeld's system, the drawing is simplified only when sub-parts are identified by the system and their outlines are removed from the drawing. Cheng removes all arcs, circles and horizon lines from the drawing, effectively presenting his sub-part identification and reconstruction processes with two simpler drawings: one containing only cylindrical objects; and the other containing only plane-faced objects.

1.2.12 An integrated contextual approach.

Two systems have been implemented which use knowledge-bases to interpret the drawing. The first[13] uses a set of production rules to determine the solid primitives of an object from the two-dimensional segments in the drawing. A more sophisticated and promising approach is outlined by Yoshira et al.[119] which pays more attention to the text in the drawing than to the drawing data itself.

The system is primed with a set of types of data structure which would typically contain information concerning dimensioning, finishes, tolerances and thread sizes. The program hunts through the drawing looking for text strings and for each one found, determines which data structure that piece of text belongs to and creates an instance of that structure. The fields of the structure are filled from the string, typically storing numerical data and the units of any measurement along with any appropriate understood keywords. Once the text is exhausted of meaning , certain fields in the data structure may still be unassigned and for each unassigned field a 'demon' is loosed which attempts to find something to assign to the field. As an example, the description of a thread size is usually applied to a set of holes in the drawing which are connected to the text description by connecting lines. The demon attempts to find some holes to apply to the thread description, possibly by following arrows into parts of the drawing or by finding cirlces of an appropriate size.

Having constructed a series of data structures from the text in the drawing, the program then simplifies the drawing by extracting all the components which are fully specified by the data structures. A three dimensional object is then constructed from the simplified drawing following the path of constructing three-dimensional vertices, lines, surfaces and sub-parts after Wesley and Markowsky [57,114]. The final object is composed from the sub-parts specified by the text and the sub-parts constructed from the drawing.

This system is promising in that it treats the drawing as a whole entity, building its interpretation from textual data as well as graphical data. This is necessary in understanding engineering drawings, and this necessity is underlined by the simple fact that text does appear on engineering drawings. If an engineer cannot fully understand the intention of the designer without text being supplied, then what chance does a machine stand.

No indication is given by the authors as to how successful and independant this system is, but it is obvious that the system is limited mainly by the types of data structures with which it is primed and by its understanding of the words in drawings. With sufficient effort put into maintaining its data base, this could evolve into a powerful and robust system.

1.3 The extent of progress towards fully automatic interpretation.

The main direction of development in systems for interpreting engineering drawings is decreasing the reliance upon the user. The first systems depended on the user to identify the closed loops in the drawings, and to identify the correspondences between the features in each view of the object [47,48]. This dependence successively decreased until, by the time Wesley and Markowsky presented their algorithm, the user was not required to perform these low-level tasks at all. Simple geometric algorithms were sufficient.

However, as algorithms became more competent at interpreting the drawings of plane-faced objects, the systems became reliant upon the user to determine which solution, from all those constructed, was the correct one. These systems could produce all the possible

combinations of arrangements of solids which could be derived from a single drawing, but unfortunately were unable to distinguish solutions which were physically realisable from solutions which were the obvious intention of the drawing. Automatically performing the informed qualitative judgments necessary to identify the correct solution from a set of candidates would require an increase in the intelligence of the systems. They would require increased knowledge about engineering and the type of objects engineers produce. They would probably, like engineers, need to be able to read the text on the drawings to elicit further knowledge which could be used to guide the reconstruction process. A possible route to the future along these lines is shown by Yoshira[119].

Increasing the practical applicability of engineering drawing interpreters has increased the dependence upon the user. Interpreting drawings of objects with curved surfaces requires some assistance now to identify the boundaries between blended surfaces and to separate the overlapping projections of components of an object [2]. Automating this procedure would require an increase in the intelligence and knowledge of the system.

Looking to the future when scanned images of engineering drawings become the main focus of attention rather than the perfect CAD system produced images which are currently being interpreted, the reliance upon the user may increase even more. The images would probably be imperfect, suffering from noise, blurring and other distortions. The user may initially be required to help the system produce a good image from the scanned image - to re-instate lost lines, to separate lines which have blurred together, to remove lines which are only creases in the paper.

Overall, while this technology is developing, the degree of automation will always be compromised by attempts to increase its generality and practical applicability.

1.4 The extent of progress towards a practically applicable engineering drawing interpreter.

The earliest attempts at interpreting engineering drawings concentrated only on the projections of plane-faced objects. Such projections obeyed simple rules, but even so provided the challenge of overcoming the ambiguity inherent in drawings where one line in a projection can represent any number of coincidental edges of the object. Following the expositions of Wesley and Markowsky, the interpretation of engineering drawings of plane-faced objects could be assumed to be complete and established.

Plane-faced objects are only a small sub-set of all the objects which can be represented by engineering drawings. Once the problem of interpreting the drawings of plane-faced objects had been solved, the next problem to be tackled was to extend this solution to a more practically relevant set of object types. Sakurai and Gossard extended the techniques developed by Wesley and Markowsky to cope with objects incorporating some curve types. The later algorithm of Preiss [79] catered for a larger number of curve types, increasing the range still further. Unfortunately, no-one had produced an algorithm which could interpret such objects when projected from arbitrary viewpoints. These programs which interpreted curved surfaces were rule based, and the task of collating the rules was simplified by restricting the orientations of the object in the projections.

Plane-faced objects had been interpreted using only limited knowledge of objects and the laws of projective geometry. Extending interpretation to objects with curved surfaces required much more knowledge about how certain types of objects looked when projected into two dimensions from certain view-points. This approach can become cumbersome when attempting to increase its generality. A large set of rules would be required to describe only a small set of object types when all view-points are to be considered. Alternatively, a new method of describing the structure of objects is required so that only a small set rules are required to describe an object from all view-points. This is a problem of knowledge representation.

The alternative approach to the interpretation problem offered by Aldefeld [2,3], extended the range of surfaces which could be coped with to include any curve. However

the problem of orientation is also apparent in this system: the axes of the object primitives had to be parallel to the axis of one of the projections. This problem applied to primitives with either curved or plane-faced surfaces, and so in effect the system was more restricted in interpreting plane-faced objects than that offered earlier by Wesley and Markowsky. Consideration of the types of objects which could be reconstructed using Aldefeld's method however, revealed his to be of more practical relevance than those limited to plane-faced objects even if they had no restrictions on the orientation of the objects.

Aldefeld's approach suffered from another considerable drawback. His system as presented was semi-automatic and required interaction with a user in order to identify the outlines of geometric primitives in the drawings. In order to develop into a fully automatic system, it would have to isolate the geometries of the projections of primitives from one another, this being difficult when the alignment of the primitives overlap producing intersecting projections. The solution to this problem was to incorporate more knowledge from the user into the system. His system needed the user to separate any overlapping projections of object primitives and to insert auxiliary lines where different surfaces blended together. These tasks could only be performed by a system with considerable knowledge about the projections of the various types of object likely to be encountered, which once again is an example of an open problem of knowledge representation.

1.5 The aims of this project.

The remainder of this thesis describes the principles and components used to construct an engineering drawing interpreter. This system was developed using the work of Aldefeld [2,3,4] as the basis. Aldefeld's approach to the problem was felt to be well suited to the eventual aim of interpreting engineering drawings recovered from a digitising scanner.

Other approaches described in section 1.2 relied heavily on the accuracy of the drawing data used by the interpreters and so would prove problematic when faced with the task of interpreting a scanned image. The drawing data obtained by reading a paper drawing from a digitising scanner is inaccurate. Creases and shadows on the paper can introduce new features in the drawing. Line-segments and arc-segments appear as jagged lines.

Further inaccuracies may be added by the low-level interpretation tasks which are typically performed to obtain vector descriptions from the intensity array provided by the scanner. Thresholding, binarisation, line-encoding and arc-encoding filters are examples of these low-level interpretation tasks, just as susceptible to errors caused by ambiguity in their input data as the high-level drawing interpretation tasks discussed in section 1.2. These filters may generate lines and arcs which were unintended, and may also remove, distort and truncate intentional features.

The engineering drawing interpreters presented in section 1.2 all have one feature in common, they are "bottom-up" or "data-driven" procedures. The efficacy of such procedures is dependent upon the accuracy of the data they operate upon. Poor quality data yields poor results, which in this case would mean misinterpretations or no interpretations at all. An obvious opposite approach would be to develop a "Top-down" interpreter. Such an interpreter would need to know the entire set of possible solutions and would be faced with the task of finding the best fit between known solution and the input data. This interpreter would be less vulnerable to poor quality drawing data because extraneous or missing features would typically register as a small mismatch with the known solution, unless the drawing quality was so poor as to be unreadable. Unfortunately some obstacles are encountered in attempting to realise such a system. The first and most critical obstacle is that the interpreter must know the entire set of possible solutions. Problems here exist in representing this knowledge. One problem is the size knowledge which must be stored - a dictionary of all engineering artifacts would be large. Another problem is the format that such a dictionary would take - how it would be organised, how would the descriptions be generalised to all instances of that object. The second obstacle is that the set of engineering artifacts is not fixed, entirely new designs might be encountered which would somehow need to be interpreted before being added to the interpreter's dictionary. This obstacle apparently contradicts the whole "top-down" paradigm, requiring some "bottom-up" interpreter to construct new dictionary entries.

Following this argument, it would seem that a "top-down" interpreter might overcome some of the difficulties of a "bottom-up" interpreter, actually identifying missing or incorrect features in the drawing. Similarly a "bottom-up" interpreter might overcome the problems encountered by a "top-down" interpreter when encountering an unknown

artifact. Aldefeld's approach to engineering drawing interpretation offers a practical mid-ground solution. It is a "top-down" approach in that it attempts to fit patterns to the drawing data. However it is not actually attempting to fit given objects into the data, but rather types of object. These types are sufficiently generalised to encompass many completely different instances. Associated with each object type is a construction rule sufficiently generalised to work with any drawing data deemed to fit that type. This enables the object to be built "bottom-up".

The main aim of this project is to provide a framework which will enable the principles developed by Aldefeld to be applied to digitised drawings. The system built by Aldefeld was limited to working with CAD system images in which the drawing data was complete, accurate and structured favourably for use by his system. Here we aim to provide some new components which will generate drawing data structures favourable for interpretation by a system similar to Aldefeld's, and also to make the system less dependent upon accurate and complete data.

CHAPTER 2.

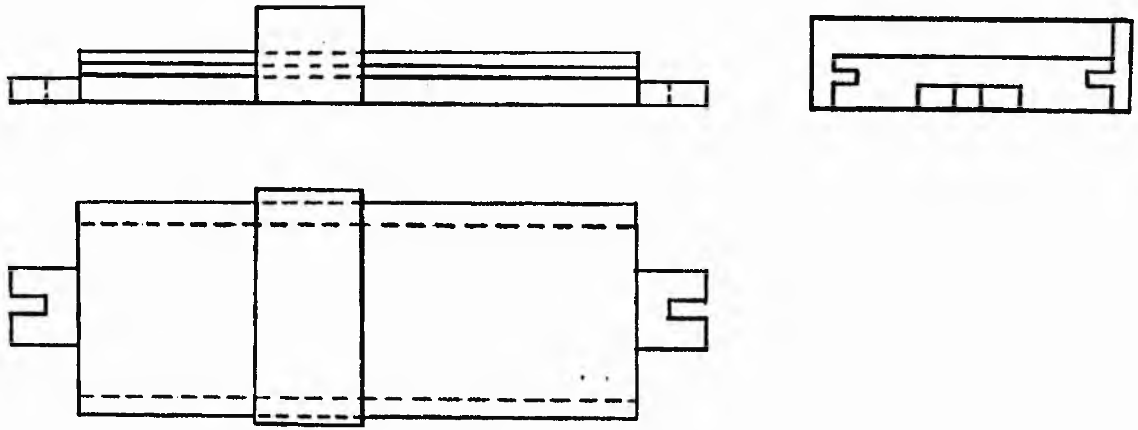
2. Interpretation of an engineering drawing - a tutorial.

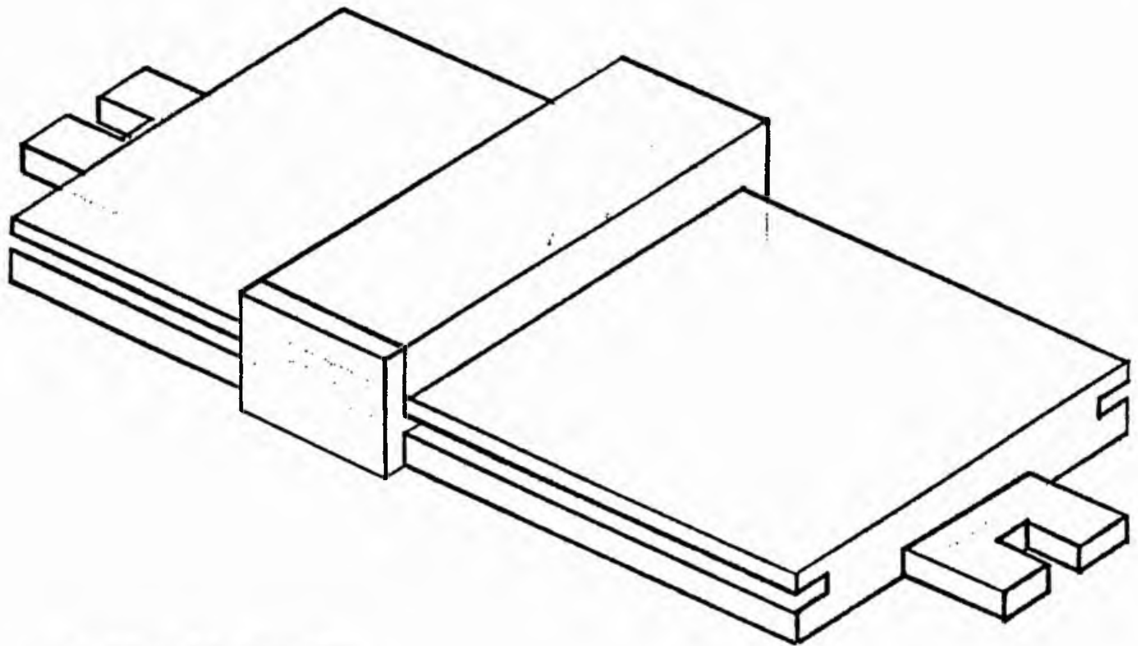
This chapter presents the method for interpreting three-view engineering drawings which is developed in this thesis. This presentation is an informal overview of how the method works illustrated with a simple example. By concentrating on how the interpretation process works, an appreciation for the sub-systems required is developed. These sub-systems are explored in more detail in subsequent chapters.

The example drawing - some stylised components of a bench vice.

Figure 2.1 shows the three-views given by the first-angle projection of the assembly of the base and the sliding jaw of a bench vice. Beneath these projections is an isometric projection of the assembly. The task of the interpreter is to derive sufficent three-dimensional information from the first-angle projections to produce the isometric projection.

The bench vice is not realistic. It has been styled so to be composed entirely of planar surfaces, whereas in reality some of these surfaces might be blended together producing arc segments in the drawing. These have been omitted to reduce the size and complexity of the reconstruction task presented in this example.





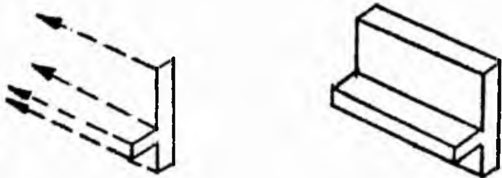
The example drawing:
First angle projection and isometric projection
of a stylised sub-assembly of a bench-vice.

Figure 2.1

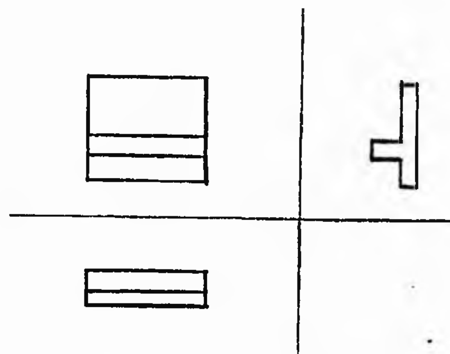
2.1 Uniform thickness objects.

The problem is to find a set of simple three-dimensional geometric components which can be composed to form the three-dimensional object. In this example, the entire assembly can be broken down into five separate components (Fig. 2.3). These components are all from the class of uniform thickness objects. These have the defining property that the two-dimensional cross-section of the component is swept along an axis in a third dimension, perpendicular to the cross-section, to create a solid. Figure 2.2 illustrates the construction of an uniform thickness object, showing a cross-section and the sweep required to form a solid. Alongside this is a first angle projection of the object showing the cross-section in one view, and a rectangle in each of the other two views.

Allowing the simplifying assumption to be made that the orthographic projection of the uniform thickness objects are made from lines of sight parallel to or perpendicular to the axis of the objects, then a useful property of an orthographic projection of any single uniform thickness object is that the cross-section is presented in one view, while a rectangle is presented in each of the other two views. Therefore, to recover all the information required to reconstruct an uniform thickness object from the three projections is the shape of the cross-section and the length of the two rectangles perpendicular to the plane of the cross-section.

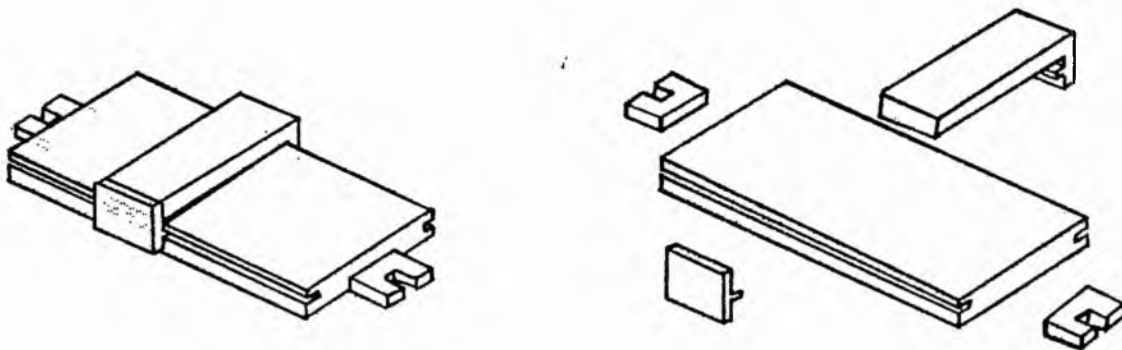


'Sweeping' a lamina through a
a third dimension creates a
solid.



First angle projection
of a uniform thickness
object.

Figure 2.2



The five uniform thickness components of the example object.

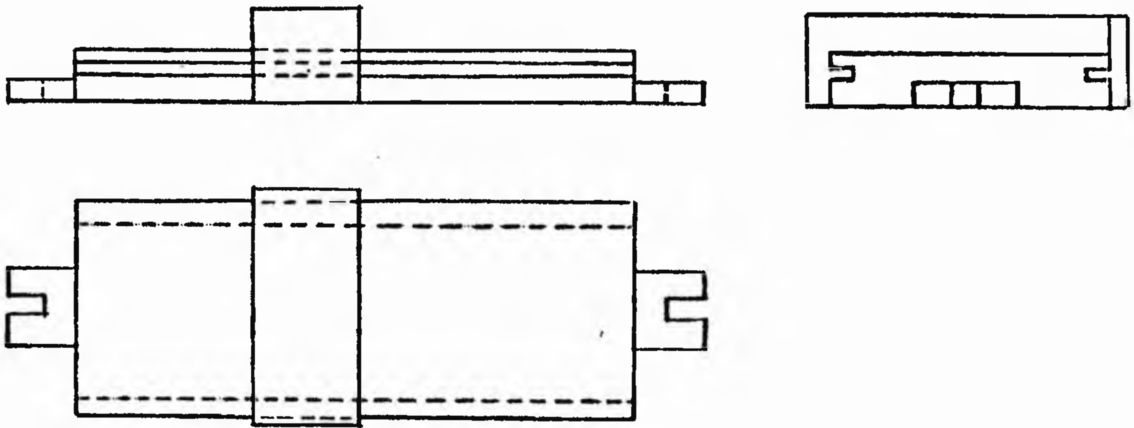
Figure 2.3

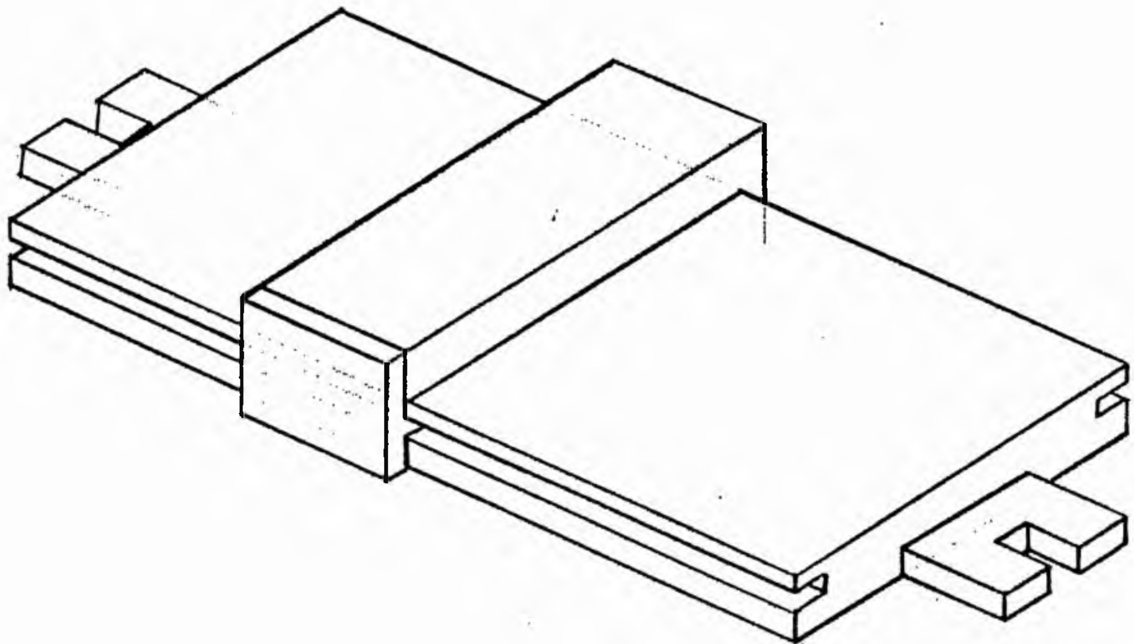
2.2 Interpretation of the three-view engineering drawing.

The task at hand is to identify all the projections of the component uniform thickness objects in the drawing. Identifying a single component requires finding a cross-section in one view, and two corresponding rectangles in the other two views.

This process is illustrated in figures 2.4 to 2.9.

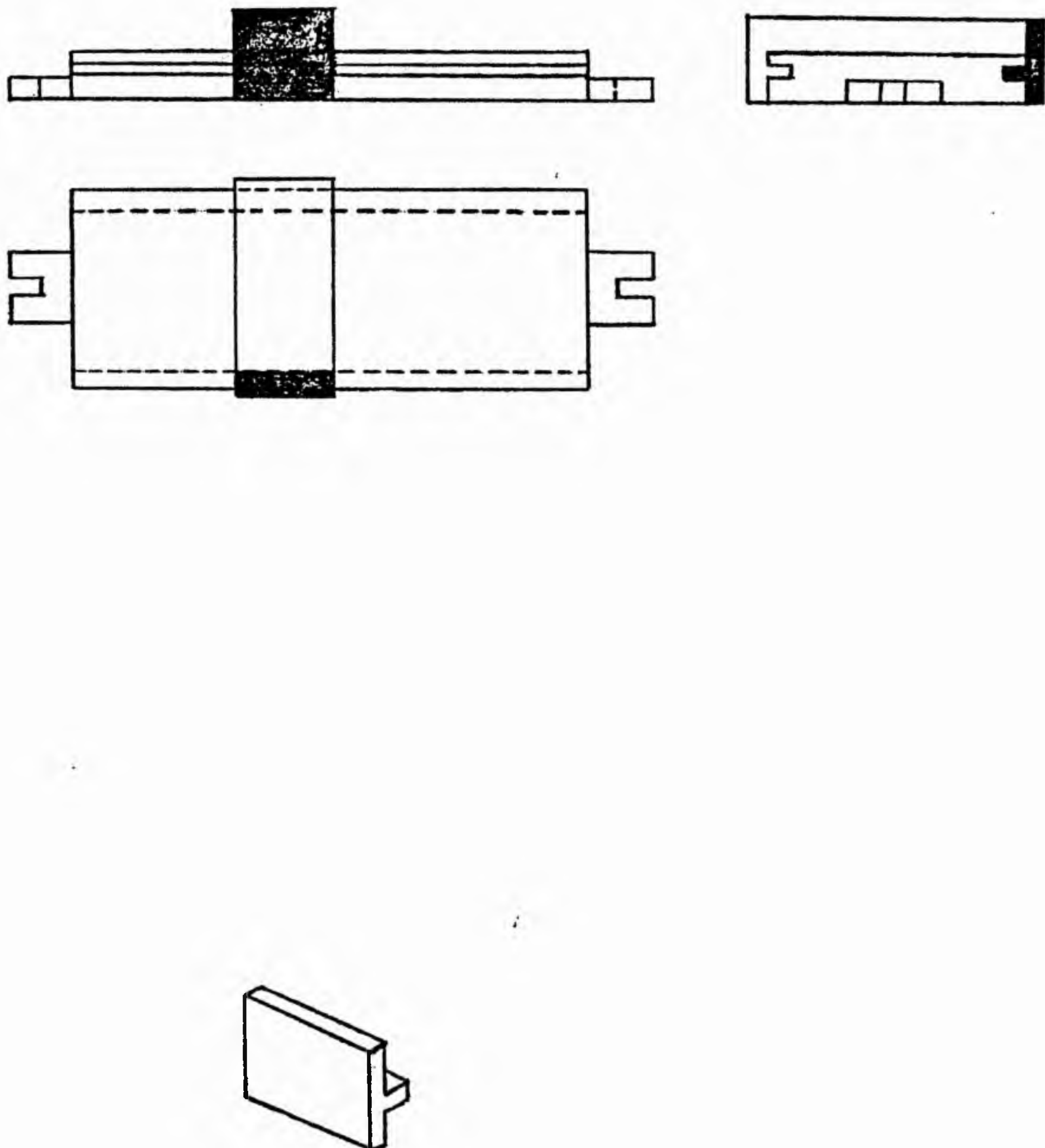
Figure 2.4 shows the completed object below the first angle projections of the object. Figures 2.5 to 2.9 show the three-view drawing with various partitions of each view shaded. These shaded partitions indicate the projections of the uniform thickness object shown in the lower half of each plate.





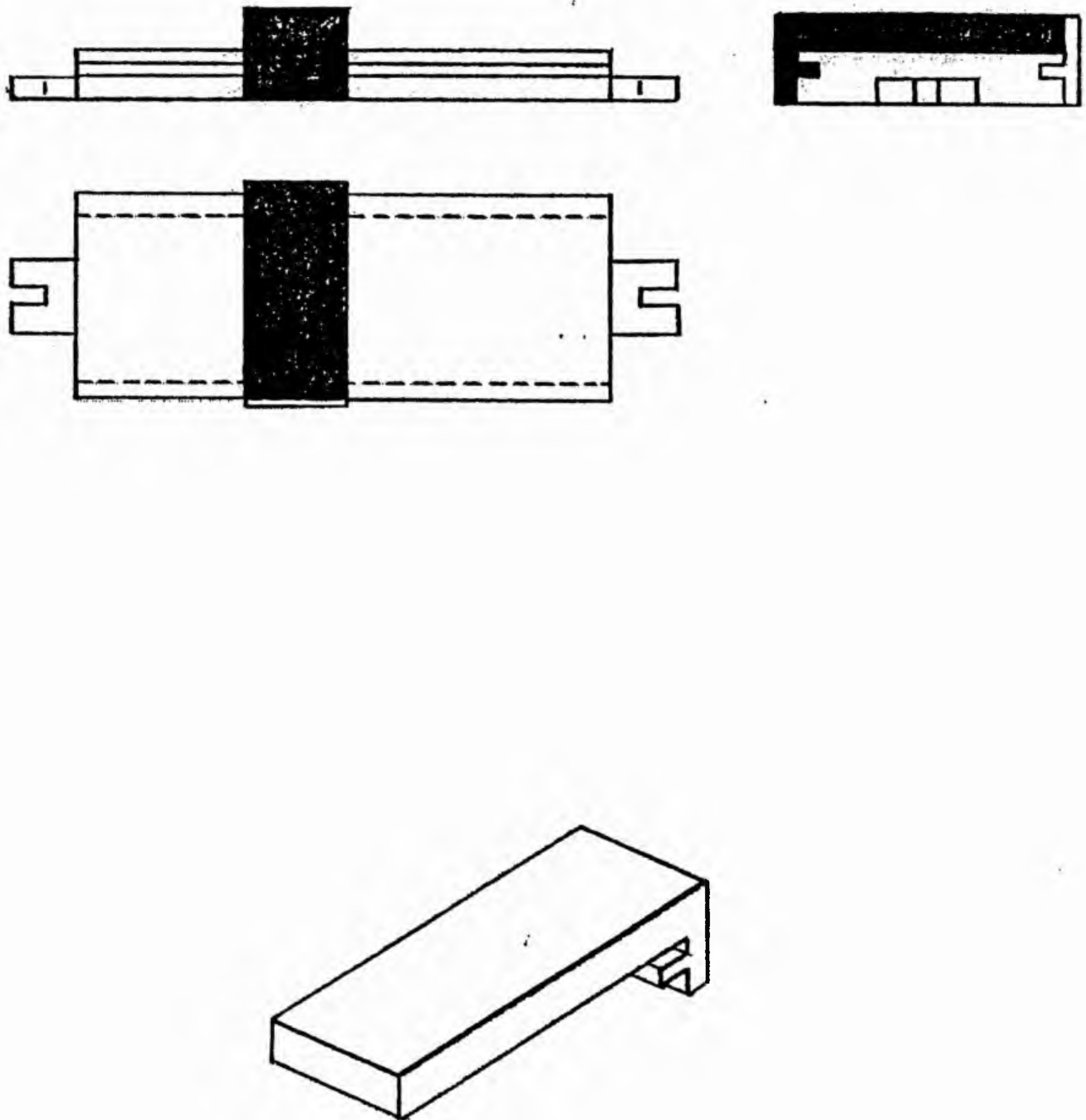
The overall composition.

Figure 2.4



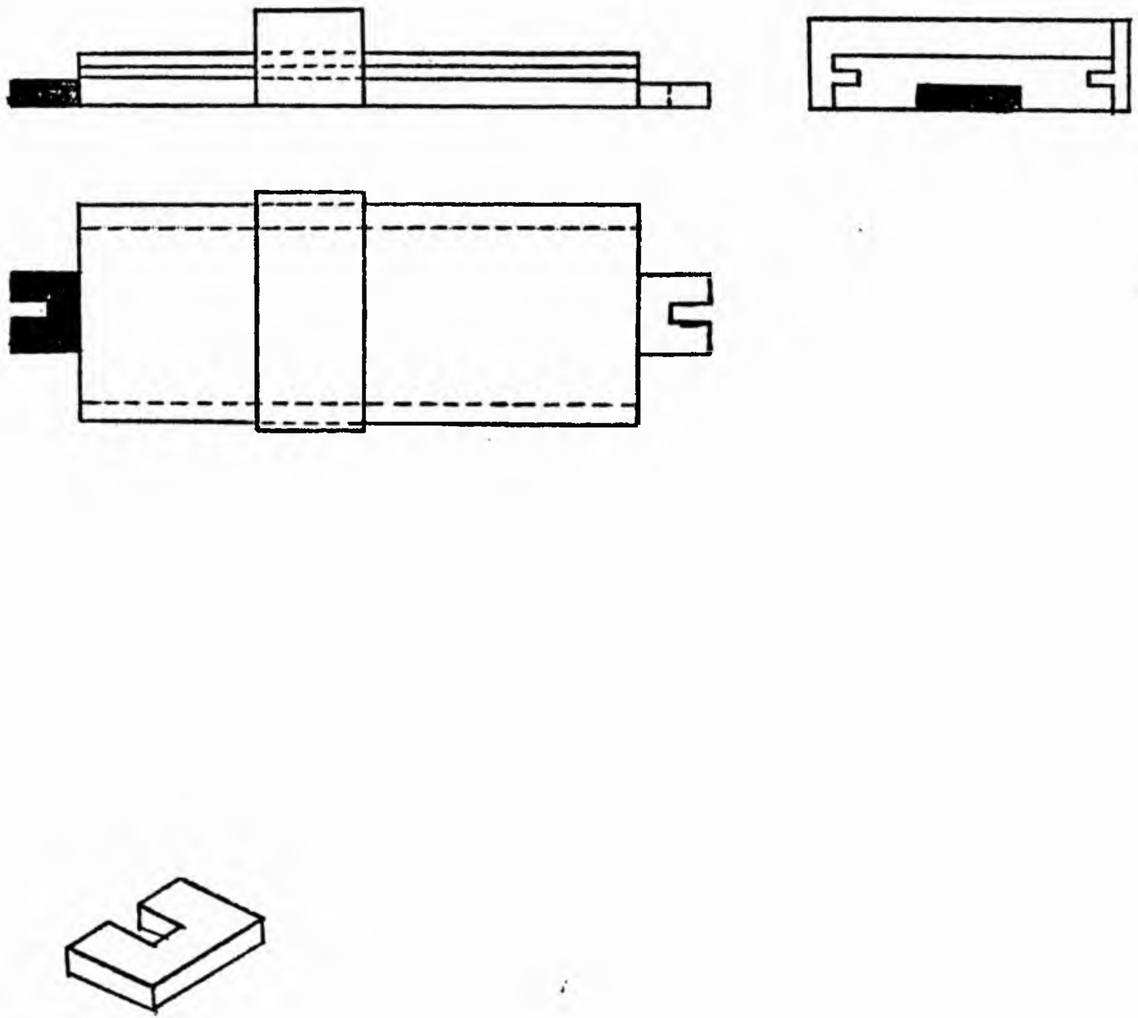
A uniform thickness component and the location of the cross-section and corresponding rectangles in the drawing.

Figure 2.5



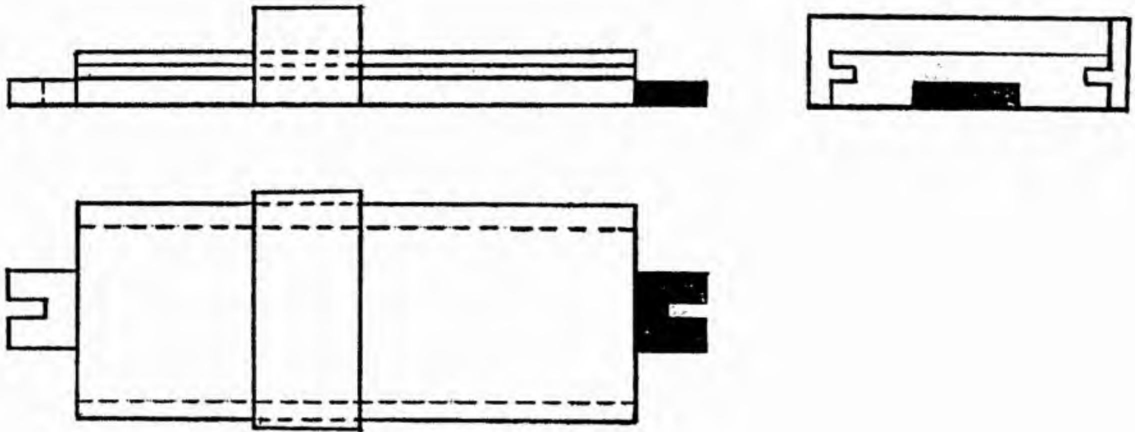
Cross-section and corresponding rectangles of a uniform thickness component.

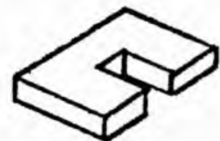
Figure 2.6



Cross-section and corresponding rectangles
of a uniform thickness component.

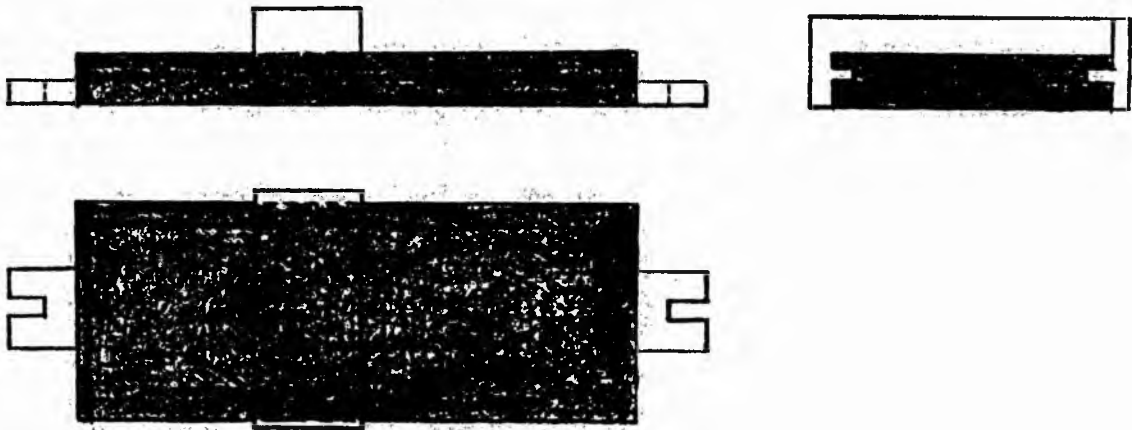
Figure 2.7

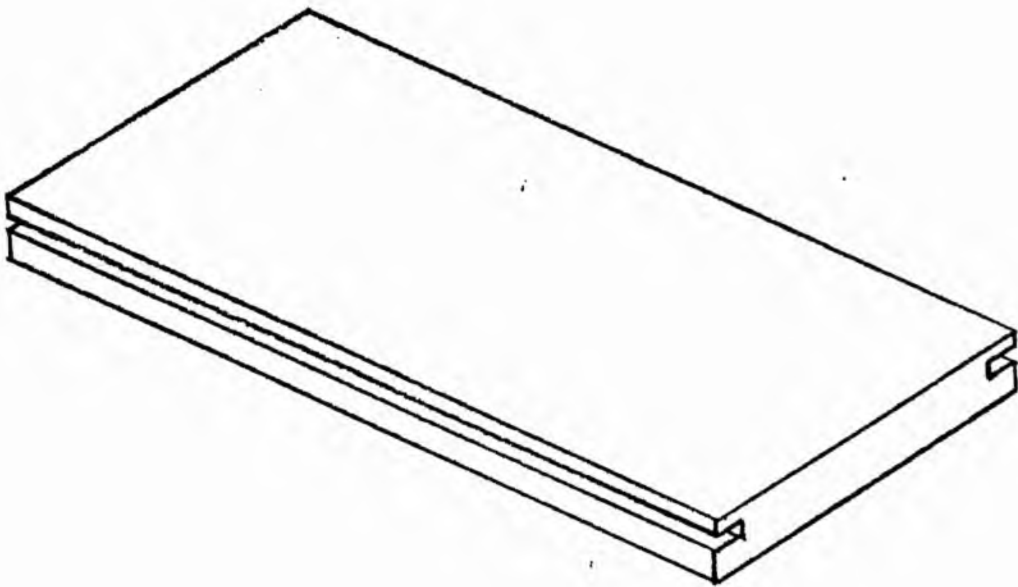




Cross-section and corresponding rectangles
of a uniform thickness object.

Figure 2.8





Cross-section and corresponding rectangles
of a uniform-thickness component.

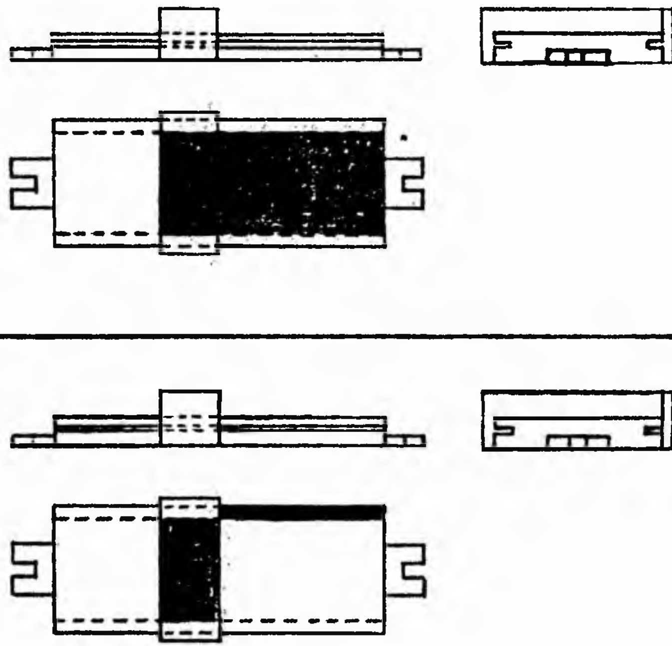
Figure 2.9

2.3 Identifying the projections of uniform thickness objects.

In figures 2.5 to 2.9, the projections of each uniform thickness object within the engineering drawing were shaded. The projections in each view were composed of a set of partitions connected together across shared edges.

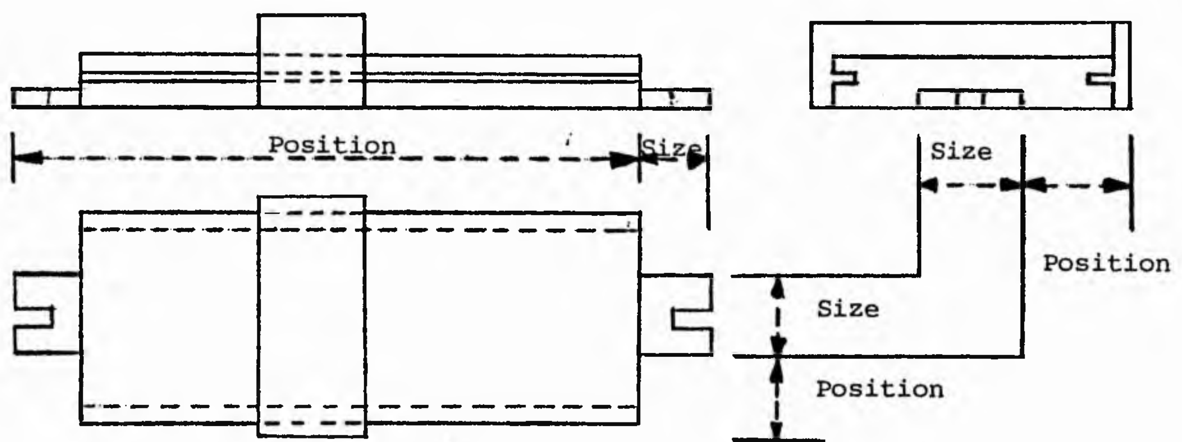
Each view can be seen to consist of a set of partitions. The task of identifying the projections of the uniform thickness objects can be seen as the task of correctly identifying a sub-set of these partitions, fulfilling certain properties. The most local property is that the sub-set of partitions must all be connected across shared edges. Figure 2.10 shows two first angle projections of the example object. In the upper projection, two partitions which share edges are shaded. In the lower, two partitions which share only a vertex are shaded.

Several properties of correspondence across views apply. Assuming that the uniform thickness object is viewed along a line of sight perpendicular or parallel to its axis, then at least two of the sub-sets from the views must have the hull of a rectangle. All of the hulls of the sub-sets must be of a similar size: the two rectangles must have the same length perpendicular to the plane of the cross-section; one of the rectangles must have the depth as the cross-section; the other must have the same breadth as the cross-section. The hulls of the sub-sets must also be in similar positions in each view. These relationships of size and position are illustrated figure 2.11.



Edge-adjacent partitions (top) and
Vertex-adjacent partitions (bottom).

Figure 2.10



Relationships between corresponding positions in the three-views.

Figure 2.11

2.4 False components may be identified.

The rules stated in the last section identified some of the correspondences between views required to construct a uniform thickness solid. Unfortunately, the simple application of these rules to a drawing consisting of an assembly of such solids, might allow more solids to be constructed than were intended to be.

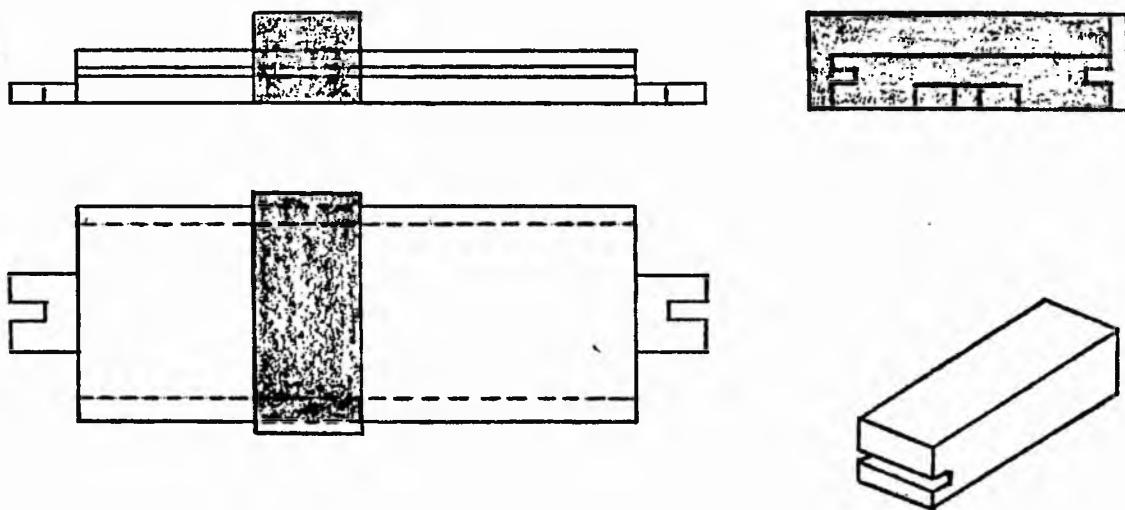
For example, only a part of a cross-section might be identified which corresponds with a spurious rectangle in the other views, or the whole of a cross-section might be identified which corresponds with a number of rectangles in the other views. Figures 2.12 to 2.14 show some examples of false solids which might be constructed following the correspondence rules. In figures 2.12 and 2.13, two incorrect cross-sections and apparently corresponding rectangles are identified and shaded. The feasible but unintended solids are shown alongside. In figure 2.14, a correct cross-section is identified, and six possible correspondences are shown alongside the solids they would produce. 'Common sense' might indicate the bottom left most construction to be the correct one.

These incorrectly constructed solids are false interpretations and need to be eliminated. There are two options for eliminating the contruction of false solids from the drawing.

The first option would be to ensure that only the correct and complete cross-sections and rectangles are identified to the construction process. This requires an amount of intelligence to be applied to the process of selecting the sub-sets of partitions in each view. This would require an amount of a-priori knowledge about what the object being constructed looks like. One immediately realisable source of such knowledge would be a human expert who would guide the interpreter in its selection of partitions, either by selecting the partitions for it, or by verifying its selection of partitions.

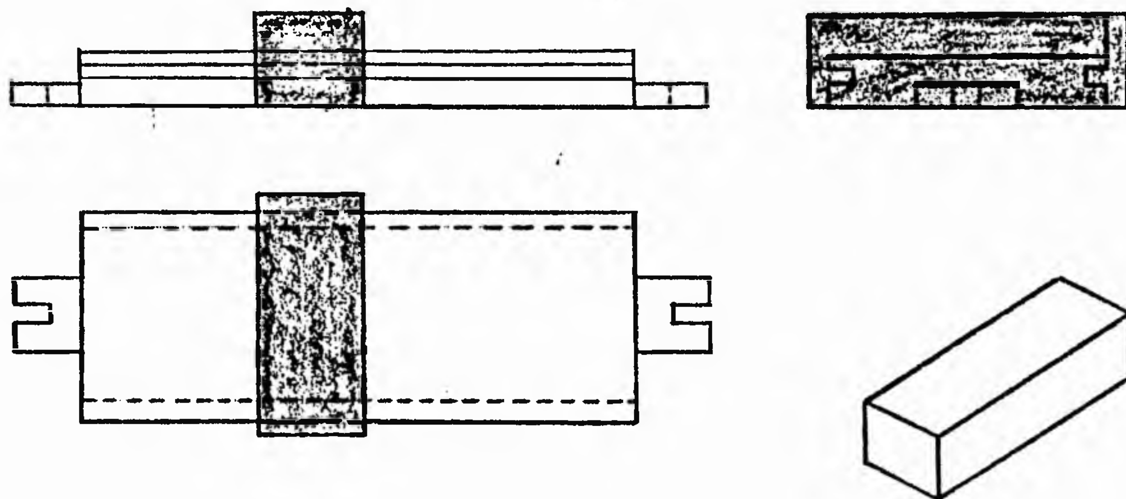
The second option would be to allow the interpreter to construct all the solids it could from the drawing, and then employ a composition process which would attempt to compose sub-sets of the solids into an assembly. The composition process could employ a set of rules concerning three-dimensional soilds to verify its compositions. For example, no two solids could be allowed to occupy the same space, and no compositions would be allowed which

left some of the component solids floating in space or otherwise precariously connected to the rest of the assembly. A further obvious verification test would be to produce the three orthographic projections from the constructed solid and to compare these with those of the original drawing. Assemblies which failed to produce the correct projections could be discarded.



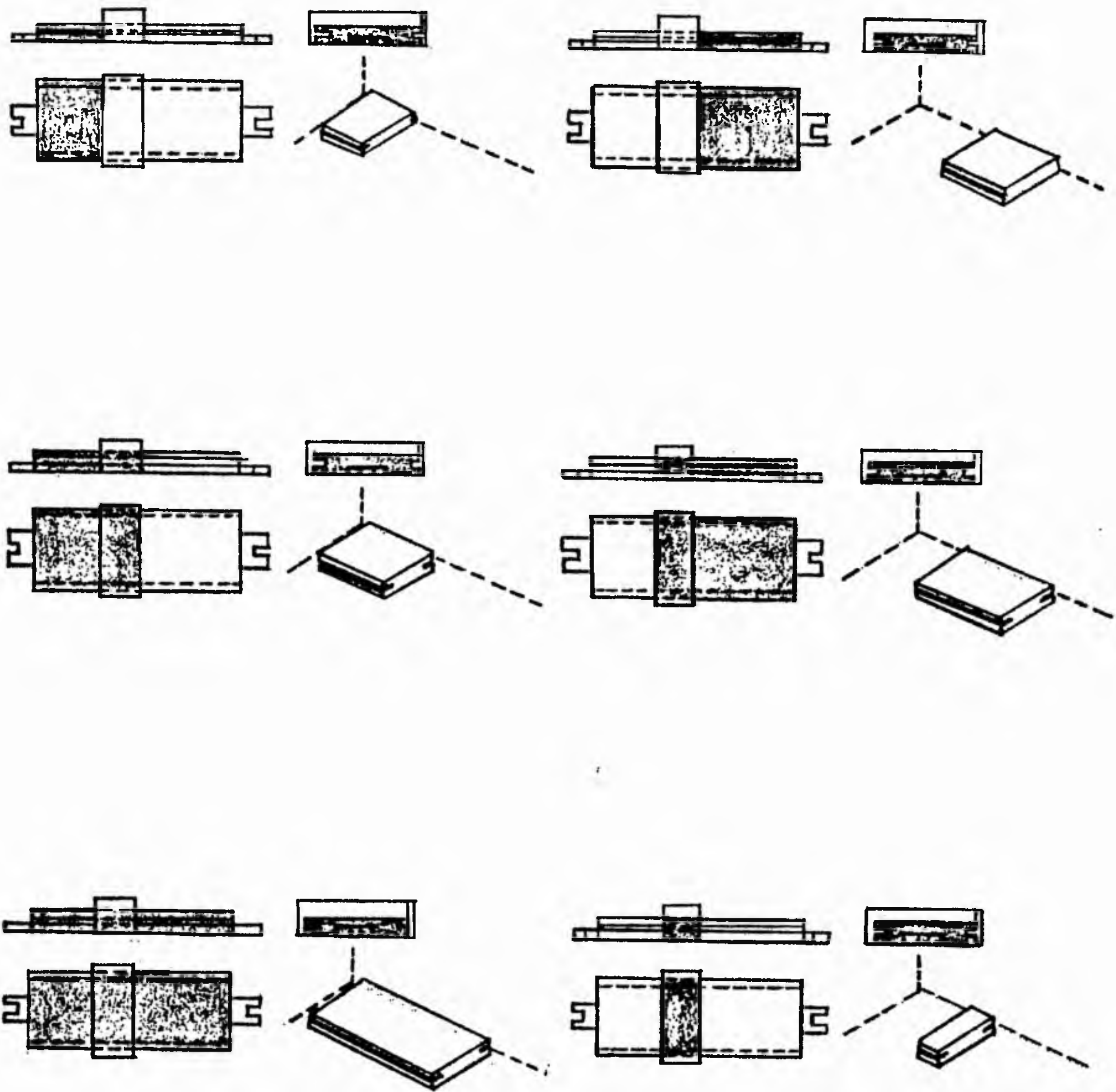
Erroneous Cross-section and corresponding rectangles.

Figure 2.12



Erroneous Cross-section and corresponding rectangles.

Figure 2.13



Correct Cross-section and multiple corresponding rectangles produce multiple candidate solids.

Figure 2.14

2.5 An outline of the svstem which has been developed.

An version of an engineering drawing interpreter has been built following the approach to interpretation described in this chapter. The system has two main components: a path analysis program, and a reconstruction process.

The path analysis program takes the drawing data as input and from this constructs a data structure which describes all the separate partitions in the drawing and the positional relationships between the partitions.

The reconstruction process takes the data structure built by the path analysis program and, guided by the user, selects the sub-sets of partitions describing the cross-sections of uniform thickness objects. An automatic procedure attempts to find the corresponding rectangles in the other views and then constructs the component solids. The process of selecting cross-sections and constructing components is repeated until the complete assembly has been constructed.

A diagram of the system is provided in figure 2.15.

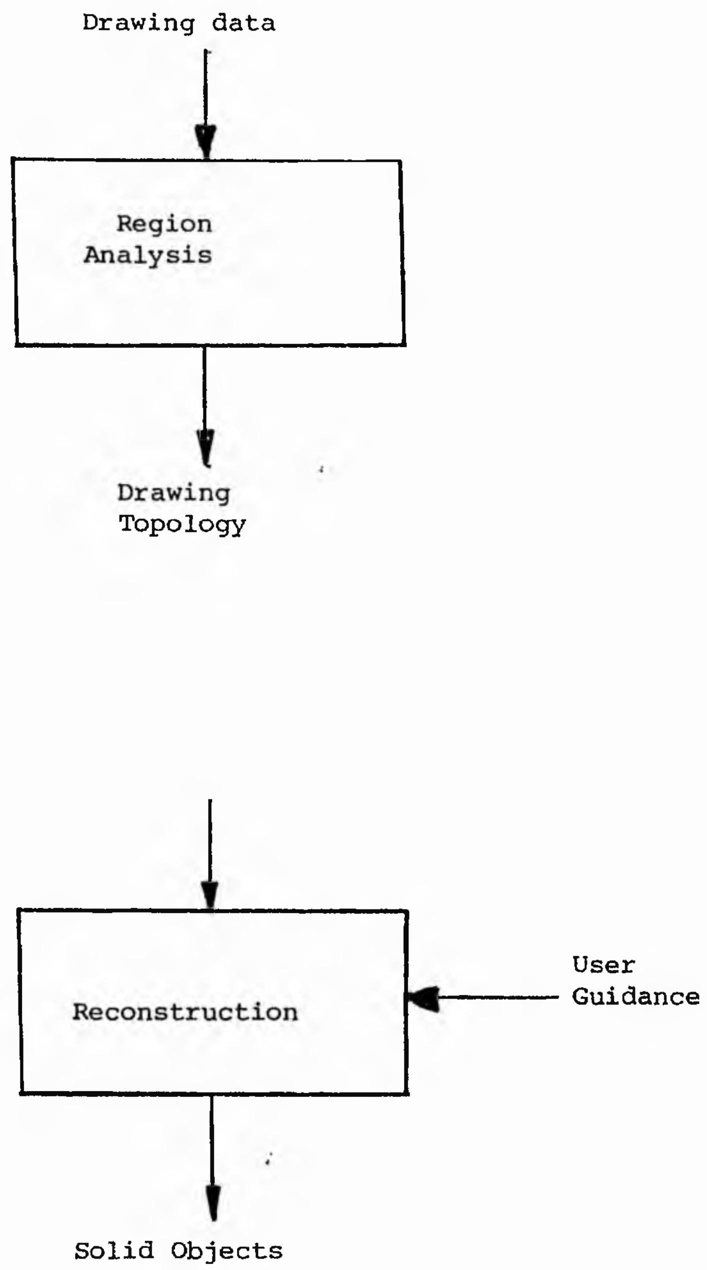


Figure 2.15

CHAPTER 3.

3. A Region Analysis Program.

A significant proportion of the project work was concerned with extracting more sophisticated information from the raw data supplied from drawing files. Many of the previous drawing interpretation projects outlined in the first chapter tended to ignore this issue in favour of concentrating on the object reconstruction algorithms, with the experimenters presumably structuring their drawing data manually or using structured output from Computer Aided drawing systems. Bearing in mind the overall objective of automatically interpreting drawings read from image scanners, the automatic structuring of the image data into a format suitable for interpretation is an important goal to be achieved.

Before describing the algorithms used to structure the image data, it is necessary to describe the properties required of the data structures which we seek to construct. These properties are described informally in terms of the information that the interpretation algorithm requires from the drawing in order to perform its function. Section 3.1 outlines these requirements, section 3.2 describes a set of data structures which fulfill these requirements and section 3.3 provides an example of the use of the data structures to represent a drawing.

Plane-sweep algorithms appeared to provide an excellent base for developing algorithms to derive these data structures, and a brief background to this class of algorithms is given. A series of extensions to existing Plane-Sweep algorithms were required to enable them to cope with the variety of general geometric conditions appearing in line-drawings.

Section 3.4 describes the development of the Region Analysis program which followed two phases. The first phase extended an algorithm which could report the Regions inside and outside a simple self-intersecting polygon into an algorithm capable of reporting the Regions within any shape. The second phase developed upon this by incorporating the desired drawing data structures into the algorithm and so constructing the edge network essential to the representation of the adjacency relationships between Regions.

3.1 Information requirements of the interpreter.

A drawing interpreter must have the drawing information available in a suitable format to answer the questions it must pose as it constructs its interpretations. The raw data from which this information is to be provided is simply a file of vectors. The data will be converted into data structures which express some of the two dimensional properties of the drawing, providing information about how vectors connect with each other, about the distinct shapes connected vectors form, and how these distinct shapes relate to each other.

3.1.1 Identify the separate views of the object.

The interpreter must know how the drawing features are shared among the three views of the object. A hierarchic structure representing the 'containment' relation between views and line-segments adequately fulfills this requirement. The root of the hierarchy will represent the 'paper' which contains the drawing. The next level in the hierarchy will represent the separate views of the object. The level below that will represent the drawing features contained in each view. Further levels of this hierarchy represent the lower levels of containment, namely Regions which contain smaller Regions.

3.1.2 Transform drawing features onto the object co-ordinate system.

In order to construct the object from the three views, the views must be transformed from the drawing co-ordinate scale to the three-dimensional co-ordinate scale of the object. The minimum X and minimum Y co-ordinate of each view must be provided to guide this transformation.

3.1.3 Connecting drawing features to each other.

The interpreter needs to know the entire set of drawing features and how these drawing features are positioned relative to each other. Intersection points of crossing lines need to be enumerated, and the connectivity between lines need to be expressed.

Closed loops of connecting lines form Regions of space in the drawing, and these Regions

are the major tokens manipulated by the interpreter. The interpreter builds candidate three dimensional surfaces from Regions and combinations of adjacent Regions. Therefore, these Regions and the adjacency relationship between Regions need to be identified.

3.1.4 Identify and preserve consistency cues.

Having produced a set of three-dimensional interpretations of each surface, the interpreter must try to compose these interpretations into a consistent arrangement which might constitute a valid three-dimensional object.

Some constraints govern the placement of three-dimensional surfaces next to each other. These constraints are implicit in the adjacencies between two-dimensional surfaces in the drawing. The boundary edge between adjacent surfaces may be either visible or hidden. A visible boundary edge dictates that the adjacent surfaces must not be coplanar in the composition of surfaces nearest the viewer along that line of sight. A hidden boundary edge dictates that the adjacent surfaces must be coplanar in the composition of surfaces nearest the viewer.

These constraints are governed by the attribute type of the line which states whether the line is visible (a solid line) or hidden (a dashed line). These line attributes are contained in the raw drawing data and must be preserved in the data structures. Two Regions, one contained entirely within the other, cannot both be in the same plane. The inner Region must be either a hole or a protrusion. Any existing containment relationships between Regions must be identified.

3.2 Data Structures which supply information to the interpreter.

The data structure used here to represent engineering drawings is a development from the 'Modified Winged-Edge Data Structure' described in Weiler's discussion of CAD data structures[112]. In this structure, the word 'Edge' has a precise meaning which must be understood before the overall structure can be described. An Edge is a length of a line-segment which lies between two consecutive vertices within the line-segment, these vertices being the points where other line-segments cross it or the end-points of that line-segment itself. The sides of the Regions inside the drawing are all 'Edges' - sometimes these Edges are the entire length of the Line-segments that contain them.

Each Edge has two sides. For clarity it is useful to have a consistent and recognisable name for each side of an edge; the names Above-Left and Below-Right are used. When an edge is a vertical edge, the edge has a Left side and a Right side. When the edge is not a vertical edge, the edge has an Above side and a Below side.

The Regions are described by their boundaries. The boundaries of Regions are described using lists of "Sides" of Edges - identifying not only the Edges on the boundary but also which side of each Edge faces into the Region. This enables the Adjacencies between Regions across Edges to be represented. One side of an Edge faces into one Region, the other side faces into an Adjacent Region. Examining all the Edges bounding one Region will identify all the Regions adjacent to that Region.

For convenience the Region boundaries are described by connecting together Sides of Edges. This is done by storing a connection to the next Side inside the Side of an Edge. The Head and Tail of the boundary description are the only connections stored within the Region descriptor.

The data structures are shown in a simple list notation. Square brackets [] are used to enclose data aggregates. For example, an entire list lies within one pair of brackets, and each list item lies within its own pair of brackets. Each Line item contains an Edge list within brackets, which consists of none or more Edge list items each contained by brackets.

List formats:-

Line List :-

[Line-id, Visibility, Intercept, Slope, [Edge-list]];

Visibility is a flag which records whether the line segment is visible (solid line) or hidden (dashed line).

Edge List:-

[Edge-id, Vertices, [Above-Left connection], [Below-Right connection]];

where a Connection has the format :-

[Line-id, Edge-id, side of Edge, Region-id];

Each Edge has two sides - a left side and a right side in the case of a vertical Edge - or an above side and a below side in the case of a non-vertical Edge. The names Left and Right or Above and Below will be used as applicable.

Each side of a particular Edge may participate in the description of a Region. The description of the Region is constructed from connected 'Sides' of Edges. The Connection structure shown above explicitly identifies the Line segment, the Edge within the Line-segment and the side of the Edge which forms the next part of the description of the Region for a given side of an Edge.

Storing the Region-id within the connection of a given Edge allows the Adjacency Graph to be built. Connecting the sides of Edges together to describe Regions automatically builds the Adjacency Graph. Given any Edge, it will be possible to identify the two Adjacent Regions on both sides of that Edge. By traversing the connected Edges for a given Region and examining the other side of each Edge, it is possible to identify all the Regions adjacent to a given Region.

Region List :-

[Region-id, Type of Region, Head Edge, Tail Edge, Parent, Child, Sibling];

The "Type of Region" field indicates whether the Region is an Interior Region - which describes the boundary of a partition of white space - or an Exterior Region - which describes the boundary around the outside of a conglomerate of none (an open polygon) or more adjacent Interior Regions.

The "Head Edge" and "Tail Edge" entries identify the Line-segment and Edge within Line-segment of the Head and the Tail of the list of connected "Sides" of Edges which describe the boundary of the Region. The side of the head and tail Edges need not be identified explicitly: the Head always lies on the Below-Right side of an Edge, and the Tail always lies on the Above-Left side.

The "Parent", "Child" and "Sibling" entries are used to link the Region list item into a Containment hierarchy describing which Regions contain which. These entries contain the Region-id of the Regions which are Parent, Child and Sibling to the given Region.

3.3 An Example of the representation.

Here the practical details of how the adjacency graph will be constructed are ignored, and some simple examples of shapes and the resulting data structures are provided. The data structures are shown in a truncated format for the sake of clarity. Only unessential entries have been omitted.

Line List :-

[Line-id, [Edge-list]];

Edge List:-

[Edge-id, Vertices, [Above-Left connection], [Below-Right connection]];

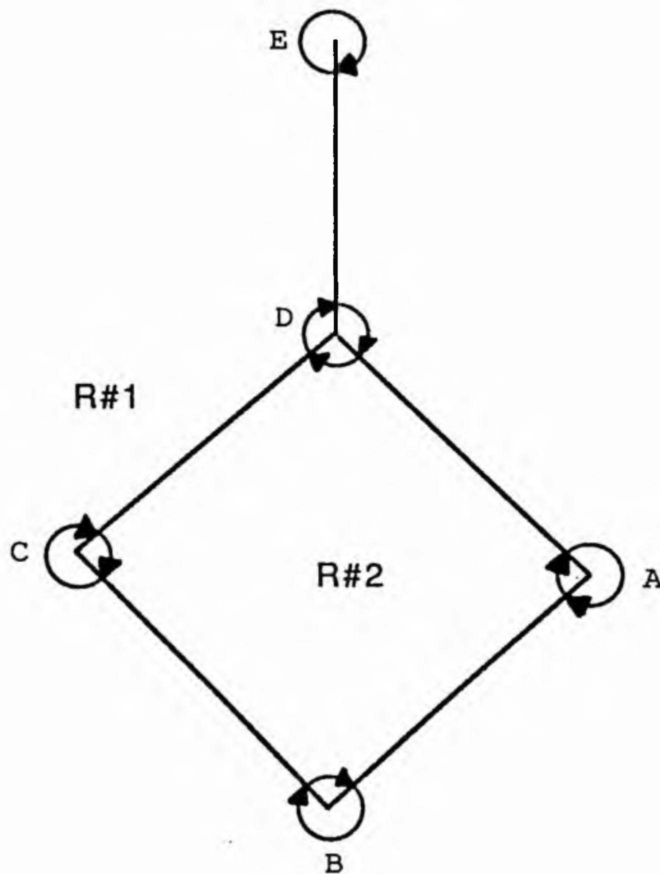
where a Connection has the format :-

[Line-id, Edge-id, side of Edge, Region-id];

Region List :-

[Region-id, Type of Region, Head Edge, Tail Edge, Parent, Child, Sibling];

An Example of the Edge-based Data structures.



Lines.

```
[
[L#BA,[[1,BA,[L#DA,1,below,R#2],[L#CB,1,below,R#1]]]
[L#CB,[[1,CB,[L#BA,1,above,R#2],[L#CD,1,above,R#1]]]
[L#CD,[[1,CD,[L#ED,1,left,R#1], [L#CB,1,above,R#2]]]
[L#ED,[[1,ED,[L#ED,1,right,R#1],[L#DA,1,above,R#1]]]
[L#DA,[[1,DA,[L#BA,1,below,R#1],[L#CD,1,below,R#2]]]
].
```

Regions.

```
[
 [R#1,Exterior,[L#BA,1],[L#DA,1]]
 [R#2,Interior,[L#DA,1],[L#BA,1]]
]
```

Figure 3.1

Reading a Region list from these lists is done as follows - using figure 3.1 as the example.

(a) Select the Region using an entry from the Region table. The Region table entry contains the head edge and the tail edge pointers for the Region list.

eg. Follow Region R2.

Region table entry R2 yields Edge 1 of Line L#DA as the head, Edge 1 of Line L#BA as the tail. .

(b) Following the Head connection, the next connection is contained in the Below-Right entry for Edge 1 in Line Segment DA. This connection is [L#CD,1,below,R#2] - Below side of Edge 1 of Line CD.

(c) Follow connection and obtain successive connection.

eg. Below side of Edge 1 of Line CD -> [L#CB,1,above,R#2].

(d) Repeat (c) until the tail edge of the Region list is reached.

eg Above side of Edge 1 of line CB -> [L#BA,1,above,R#2]
Above side of Edge 1 of Line BA happens to be the tail of the Region list for R#2, therefore the journey is complete.

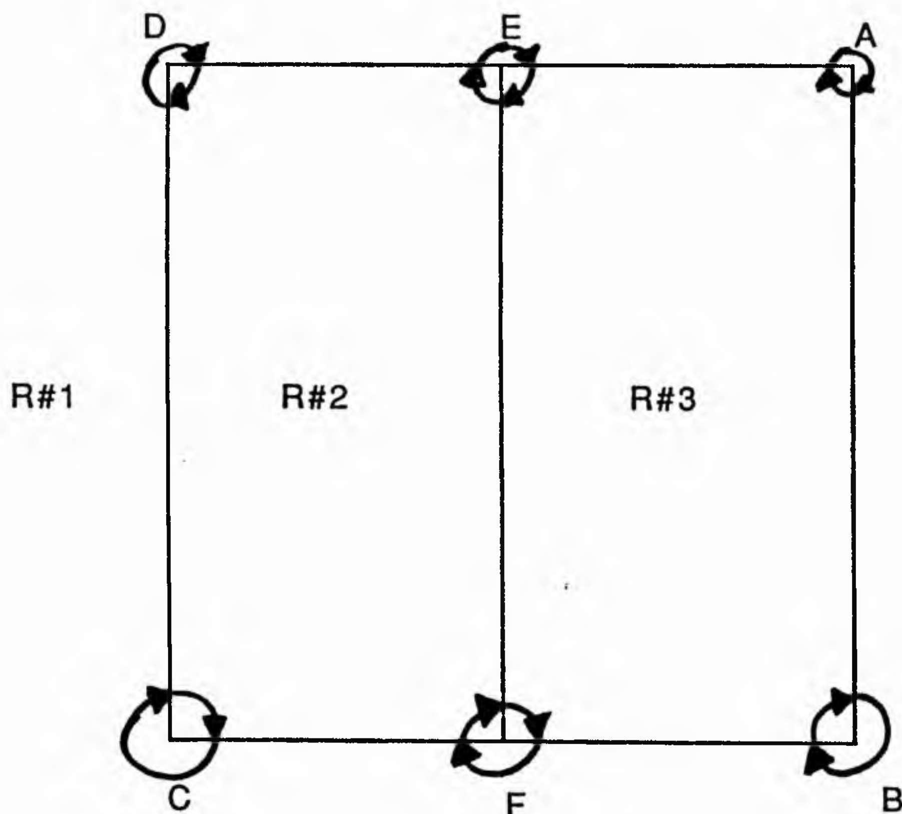
Following Region R1 would yield

(head) [L#BA,1,below,] -> [L#CB,1,below,R#1] ->
[L#CD,above,R#1] -> [L#ED,1,left,R#1] ->
[L#ED,1,right,R#1] -> [L#DA,1,above,R#1] (tail).

Notice that Line ED, sticking out on top of the shape, is correctly navigated. Line ED points to itself in the Above_Left connection, and to line DA in the Below_Right connection.

Figure 3.2 shows a further example of the Edge structures. This example shows two line segments (DA and CB) which have each been split into two edges. The Line entries for these lines contain two Edge list items - with the identifiers 1 and 2. The identity of the Edge EA is [L#DA,2] - Edge 2 of Line DA.

A More Complex example of the Edge-based Data Structures.



Lines.

```
[
[L#BA,[[1,BA,[L#DA,2,below,R#3],[L#CB,2,below,R#1]]]
[L#CB,[[1,CF,[L#FE,1,left,R#2],[L#CD,1,left,R#1]],
       [2,FB,[L#BA,1,left,R#3],[L#CB,1,below,R#1]]]

[L#CD,[[1,CD,[L#DA,1,above,R#1], [L#CB,1,above,R#2]]]
[L#EF,[[1,EF,[L#DA,1,below,R#2],[L#CB,2,above,R#3]]]
[L#DA,[[1,DE,[L#DA,1,below,R#2],[L#CD,1,right,R#2]]
       [2,EA,[L#BA,1,right,R#1],[L#FE,1,right,R#3]]]

].
```

Regions.

```
[
 [R#1,Exterior,[L#BA,1],[L#DA,2]]
 [R#2,Interior,[L#DE,1],[L#FE,1]]
 [R#3,Interior,[L#DE,2],[L#BA,1]]
]
```


Figure 3.2

3.4 Plane-sweep algorithms.

3.4.1 Intersection detection.

The basis of an algorithm which might provide some of the information structures for the drawing interpreter can be found in some of the recent work in computational geometry. We wish to construct representations of the Regions formed by the connections and intersections of line-segments. One such Region reporting algorithm belongs to the class of algorithms for counting and reporting the intersections of geometric entities, the class of plane-sweep algorithms.

Plane-sweep algorithms emerged in the late 1970's as a technique which improved the performance of intersection reporting [10] from the usual $O((n^2 - n) / 2)$ of brute force algorithms to $O(n \lg (n+k))$, where k is the number of intersections in the set. These algorithms reduce the number of intersection tests performed by ordering line-segments and only performing intersection tests on line-segments which are close to each other.

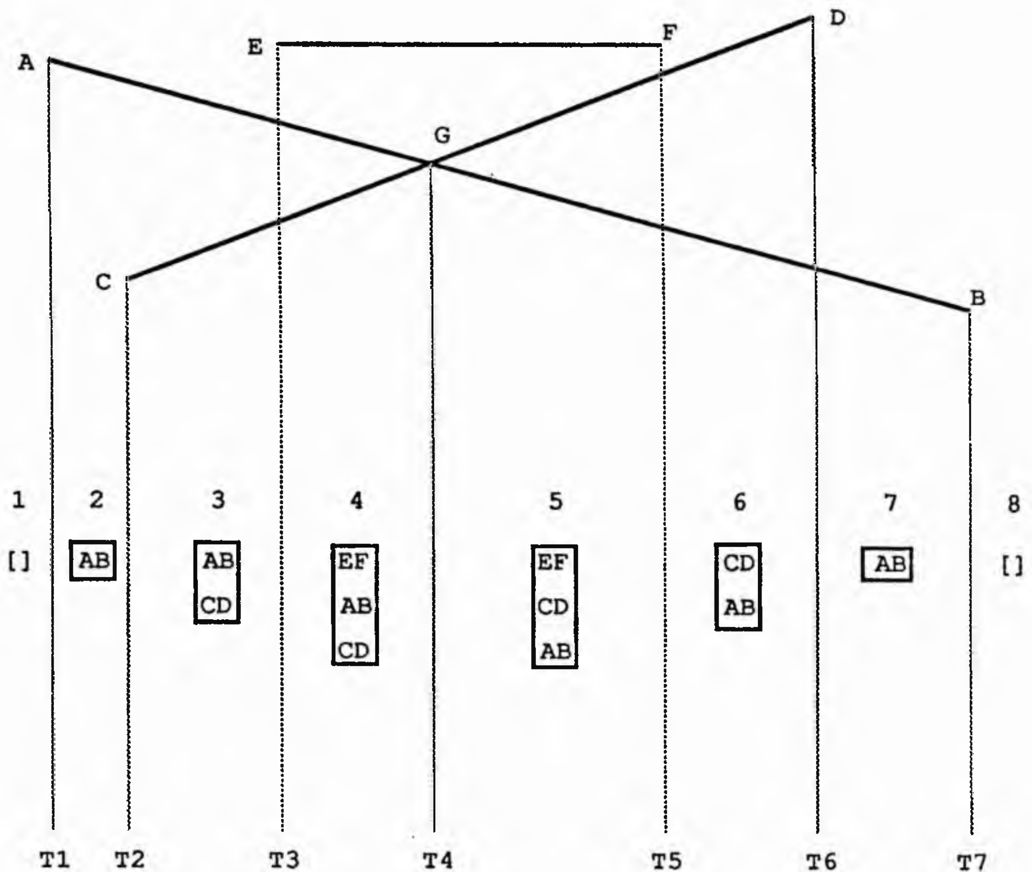
The two-dimensional problem of ordering a set of line-segments is reduced to a series of one-dimensional problems of determining the vertical order of the line-segments passing through a given point on the X-axis. The whole set of one-dimensional orderings when considered as a whole represents the two-dimensional ordering.

The one-dimensional ordering is represented by a simple list structure called a 'Front'. The head of the list, in the example of vertical ranking, would represent the lowest line-segment, the tail of the list would represent the highest line-segment.

In the two-dimensional picture, these rankings change as lines end, other lines begin, or other lines cross over each other. At the points where these changes occur, the Front structure must be amended to reflect the change (see Fig. 3.3). The set of changes which are required to reflect all the changes between two successive Front orderings is called a 'Transition'. For example when the Front reflects the vertical ordering of line-segments, a Transition between one ordering of the Front at a given point along the X-axis and the next

ordering would contain either ending lines, starting lines, intersections of lines or a mixture of the three which occur at a single point along the X-axis between the two successive orderings.

Plane-sweep algorithm for detecting intersections.



8 states of consistency in vertical ranking of line-segments. The 'Front' contains references to lines in order of vertical ranking. The 'Front' must be re-ordered at the transition points (dashed lines) where one or more vertices lie.

Start-points (T1 - T3), Intersection points (T4), End-points (T5 - T9) - according to left-to-right 'sweep'.

Initial Transition List:-
(format:[vertex,[list of line-segments starting at vertex]])

[[A,[AB]], [C,[CD]], [E,[EF]], [F,[]], [D,[]], [B,[]]]

Note:- Intersection point 'G' is not inserted into the transition list until the intersection is found by applying an intersection test to lines AB & CD.

Figure 3.3

The set of changes of orderings along a given axis is represented by the 'sweep' of the plane. The plane is 'swept' by initialising the Front to the empty state and by updating the Front according to the ordered set of Transitions. A horizontal sweep for instance may be from left to right or from right to left, according to the order of the Transitions, whether on ascending or descending X co-ordinate value.

Transitions, as already stated, contain a number of line end-points, start-points and intersection-points all at the same position along the sweep axis. End-points and start-points of line-segments are explicitly stated in the set of line-segments which are being processed. The ordered list of Transitions is prepared by sorting the vertices of the set of line-segments according to the axis of the sweep, X-axis or Y-axis, and according to the direction of the sweep, from left to right or from right to left. Intersection-points are not known when the list of Transitions is initialised, but are discovered and inserted into the list of Transitions during the progress of the sweep.

End_points, start_points and intersection-points cause the following changes to the Front structure.

At a line end-point, any lines which end there are removed from the Front. In a horizontal sweep from left to right, the right-most vertex of a line-segment is the end-point. In a vertical sweep from bottom to top, the top-most vertex is the end-point.

At a line start_point, any lines which start there are inserted into the Front. In a horizontal sweep from left to right, the left-most vertex of a line-segment is the start-point. In a vertical sweep from bottom to top, the bottom-most vertex is the start-point.

At an intersection-point, those lines which intersect there swap positions in the Front. Highest swaps place with Lowest, second-highest with second-lowest and so on.

Intersection detection is incorporated into this scheme by testing lines which find themselves with new neighbours after a re-ordering of the Front occurs. When an intersection test between two lines proves positive, the intersection-point is calculated and inserted into the appropriate place in the Transition list.

3.4.2 Reporting the Regions in non-degenerate polygons.

An advantage of plane-sweep algorithms is that they provide a sense of coherence to the line-segment set, enabling the connections between line-segments to be followed. This quality is exploited in [68] to determine the regions inside non-degenerate polygons. Non-degenerate polygons are a class of polygons with strict limitations on the types of vertex allowed -

vertices must be either

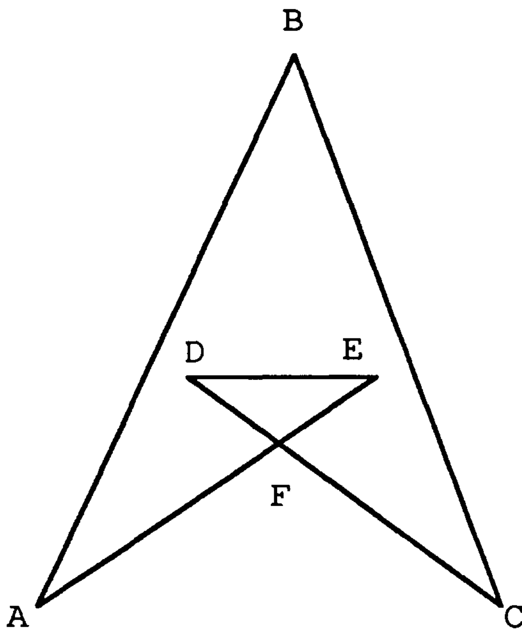
(i) the point where two line-segment end-points meet

or (ii) the point where two line-segments properly intersect. An improper intersection would be a type of T-junction where one line-segment's end-point lies within another line-segment. Vertices where more than two line-segments meet are equally improper.

The approach taken to determine the regions is to attach the heads and tails of new Region-list structures to the Front and stretch these lists around the connections of line-segments and so follow the shapes around the inside and outside of the polygon.

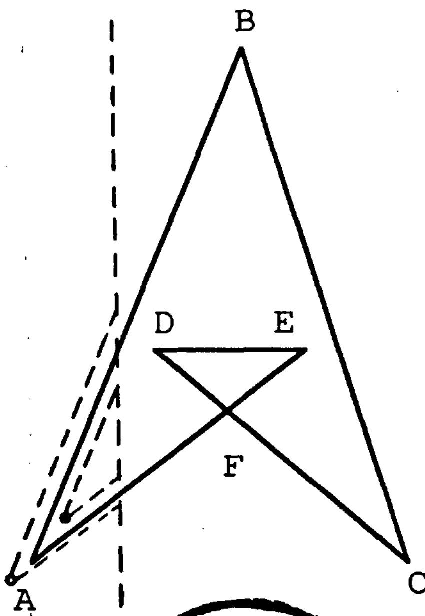
Each entry in the Front represents a line-segment, and each line-segment has two sides. In a vertically ordered Front, each line-segment has an upper and a lower side. To each side of each line-segment is attached a Region-list. In the case of simple non-degenerate polygons, both ends of each line-segment connects to the end of other line-segments. Whenever a line-segment is inserted into the Front, it replaces another Line-segment already in the Front and inherits the Region-lists on both sides of the replaced Line-segment. These lists are extended around the vertex where the Line-segments met. Figures 3.4(i) and 3.4(ii) trace the progress of the construction of the Region lists around a polygon suggested in [68].

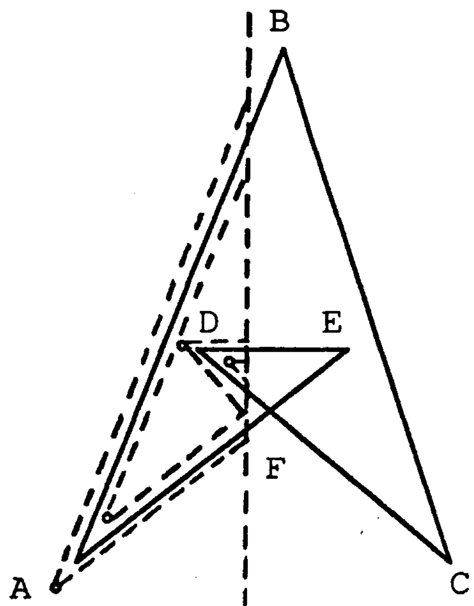
Reporting the Regions in a non-degenerate polygon.

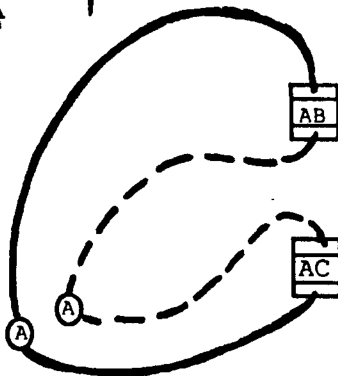


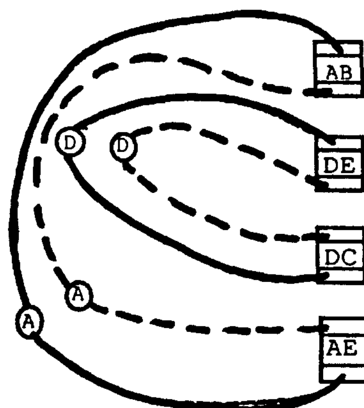
5 Lines.	6 Transitions.	
AB	A	- First
AE	D	
DE	F	
DC	B	
BC	E	
	C	- Last

The polygon has two interior regions and one exterior. Transitions are sorted from left to right.









Front entries and Region lists after first two transitions.

Figure 3.4(i)

Front Entries and Region lists through Transitions F to C.

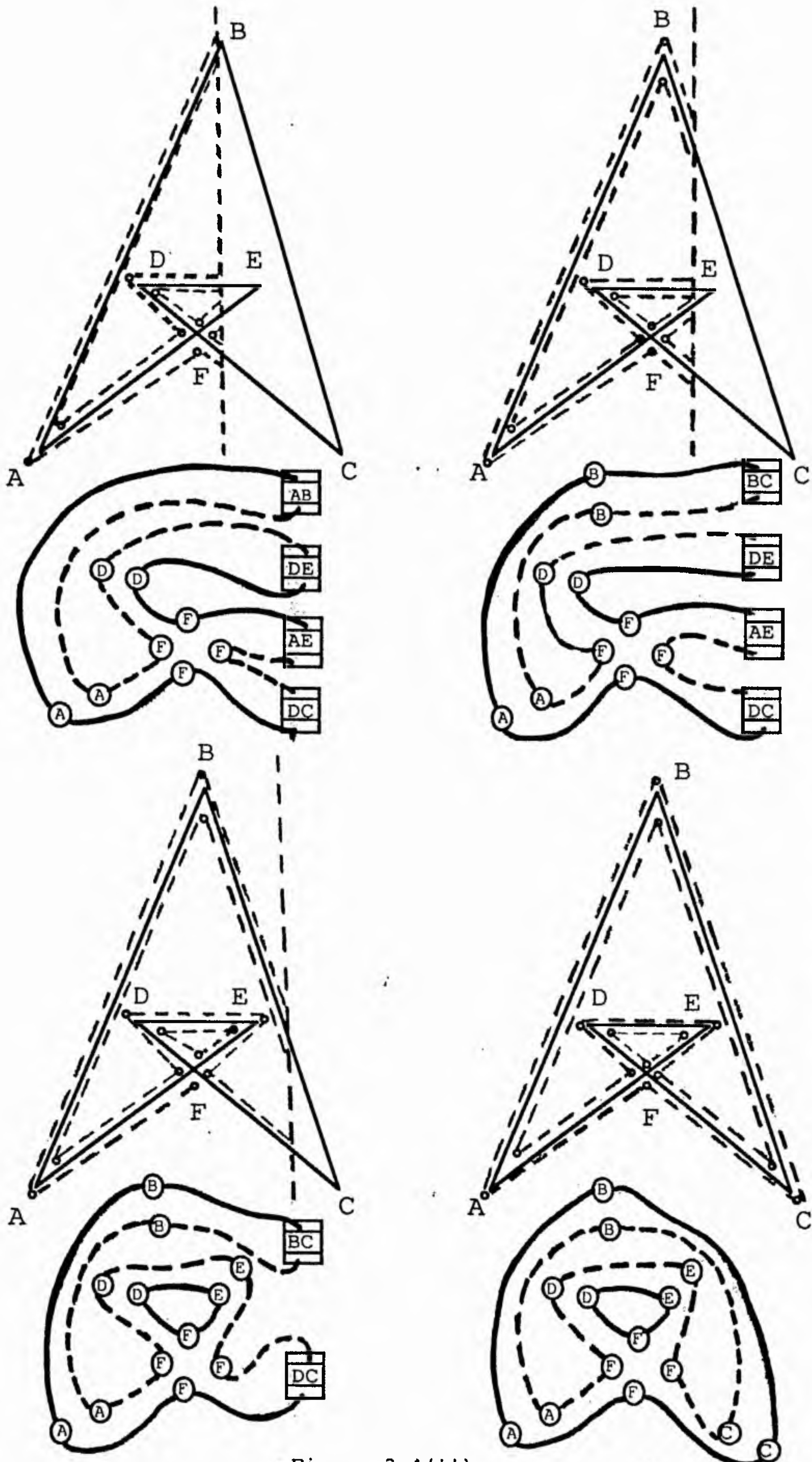


Figure 3.4(ii)

Not all vertices mark the replacement of one line-segment by another.

The very first Line-segments inserted into the front obviously cannot replace their predecessors because there are no predecessors. In the case of non-degenerate polygons, the very first Line-segments inserted into the Front will be a pair and will actually touch each other at their start-points. At such junctions the Region-lists are initialised.

The polygon ends with two lines meeting at the same end-point. Here the Region-lists below the upper line-segment and above the lower line-segment are concatenated and detached from the Front. The Region-lists above the upper line-segment and below the lower line-segment are similarly concatenated and detached.

Intersection-points in a non-degenerate polygon mark the point where only two lines cross. At such a point, four Region-lists exist, one on each side of the two lines. Two of the lists, the list above the lower line-segment and the one below the upper line-segment, are concatenated and detached from the front. The other two lists are extended past the intersection-point. A new Region-list is started with the head attached to the Front as the Region-list below the upper line-segment, and the tail attached as the Region-list above the lower line-segment.

This algorithm forms the basis from which a time-efficient algorithm was developed to determine the Region formed by the inter-relationships of lines describing an engineering drawing.

The following section describes the amendments and extensions needed to the algorithm for it to be able to cope with any of the configurations of line-segments found in a drawing.

3.5 Extending the Region-finding algorithm enabling it to cope with general vertex types.

The region finding algorithm outlined in section 3.4.2 is capable of reporting the Regions inside non-degenerate polygons. Unfortunately the majority of polygonal shapes constituting an engineering drawing are degenerate in the sense that touching vectors often

occur where an end-point of one vector lies somewhere between the end-points of another.

Despite these drawbacks, the plane-sweep approach has some promising qualities :-

- no need for a separate intersection detection phase.
- reasonable performance / size curve.
- conceptual simplicity of the approach.

The problems in implementing such an algorithm are :-

- coping with a general vertex type.
- coping with vertical lines.

The problem of coping with general vertex types can be simplified by considering any vertex in a drawing to be a combination of one or more of the following simple vertex types:-

(a) Line segment End-points;

(b) Intersection points;

(c) Line segment Start-points.

By systematically dealing with these vertex types, complex Junctions which incorporate many of these simple vertices can be coped with.

The Plane-sweep algorithm is assumed to sweep along the horizontal axis from left to right. Under these constraints, an End-point of a line-segment is its right-most vertex, and the Start-point is the left-most vertex. Obviously a vertical line has neither a right-most nor a left-most vertex, and so the Junctions involving this class of lines do not easily fit into the three simple vertex types for which we are designing an algorithm to cope. Vertical lines need to be treated as a special type of Junction rather than as a line. This special type of Junction extends upwards and can pass through many of the complex or simple Junctions which occupy only one point in the plane.

In the following section, the extensions to the algorithm required to deal with the simple vertex types will be described. In section 3.7, further extensions are described which are needed to connect vertices together when a vertical line passes through them.

3.6 An outline of the algorithm.

The algorithm consists of two phases.

The first is an initialisation phase where the set of lines from the drawing are sorted into an ordered set of vertical Transitions.

The second phase is the actual Plane-sweep where the Front data structure is maintained by sweeping it across Transitions. For each Junction in the Transition, a separate procedure is called to deal with each of the three simple vertex types. A Remove procedure is called to remove the Front entries of line-segments with their End-points at the Junction. A Permute procedure is called to swap the order of any line-segments intersecting at the Junction. An Insert procedure is called to insert new entries into the Front for any line-segments with their Start-points at the Junction.

3.6.1 Initialisation - preparing Transitions from line data.

With regard to a horizontal sweep direction, a Transition is the set of Junctions which occur at different vertical positions at a single point along the X-axis. The Junctions in a Transition are complex vertices, each possibly consisting of a mixture of End-points, Start-points and Intersection-points of line-segments. Intersection-points are not explicitly stated in the line data of the drawing, and are calculated later by the Plane-sweep. Therefore at this Initialisation stage, Transitions consist of Junctions which consist of only line-segment End-points and Start-points.

The Remove procedure, as described in more detail later, searches the Front structure for line-segments which have their End-points at the current Junction. Therefore, all the Remove procedure needs to know about a Junction is its position. It does not need to know which lines have their End-point at that Junction.

The Insert procedure needs to insert new entries into the Front structure and so needs to know which line-segments have their Start-points at a given Junction. The Transition list must therefore include sufficient references to the line-segments for Junctions which are the

Start-points of line-segments. The Permute procedure, which copes with Intersection-points, searches the Front structure to find the entries of Intersecting line-segments. Therefore all the Transition list needs to store about an Intersection-point is its position.

The Initialisation phase constructs the initial Transition structure using the Start-points and End-points of line-segments. This structure is later updated by the Plane-sweep phase which discovers any Intersection-points and adds these to the Transition list.

An entry in the Transition list is created for each unique X component in the line data. Each entry in the Transition list contains a list of unique Junctions which have the same X component as the Transition. Each entry of the list of Junctions contains a list of line-descriptors of any line-segments with their Start-points at that vertex.

<u>The format of the Transition list is :-</u>

[X-position [Junction list]]

The Junction list holds an entry for each vertex lying along that Transition at its X co-ordinate.

<u>Junction list item:-</u>

[Y-position,[list of line descriptors]]

The list of line descriptors holds an entry for each line-segment which starts at the position of that Junction within that Transition.

Example Transition list.

```
[
  [100, [ [10,[]], [40,[L4]], [60,[L1,L2,L3]] ]  ]
  [200, [ [10,[L5,L6]], [70,[]] ]  ]
  [250, [ [60,[]] ]  ]
  [300, [ [10,[L7]], [30,[]], [40,[L8]], [70,[L9,L10]] ]]
].
```

A Transition list showing four Transitions, their associated Junction lists, and the lists of line-segments which have their Start-points at those Junctions. An empty list is signified by empty brackets.

In the first entry, representing a Transition at X = 100, there are Junctions at Y co-ordinates 10, 40 and 60. No lines start at point (100,10); Line L4 starts at (100,40); Lines L1, L2 and L3 start at (100,60).

3.6.2 The procedural elements of the Plane-sweep algorithm.

The Front progresses along the sweep axis by reading successive Transitions and updating the Front according to the list of Junctions found contained in the Transition. Each item in the Junction list describes the position of the vertex and contains a list of line descriptors of any lines which have their start-point at that vertex.

The progress of the Front is from left to right. Each Transition which is processed advances the Front one step. Contained in each Transition are one or more Junctions. These are ordered from bottom to top. An executive procedure is responsible for reading the Transitions and calling the procedures to update and advance the Front.

For each Junction, separate procedures are called to deal with the three simple types of vertex from which the Junction may be composed. The Remove procedure is called to deal with any line-segments which have their End-points at the Junction. The Permute procedure is called to deal with any line-segments which intersect at the Junction. The Insert procedure is called to deal with any line-segments which have their Start-points at the Junction. In the remainder of this section, these procedures will be described in more detail. The Region-list handling operations have been deferred until section 3.7 which is devoted to describing the Region-list maintenance which must be performed by the procedures.

Incorporated into the Plane-sweep is an intersection detection function. Generally, the intersection test is only performed when an entry in the Front finds itself adjacent to an entry that it was not adjacent to before. Intersection tests therefore need only be performed in the locality of any re-ordering of the Front. Specifically :-

- after a removal or series of removals, test the entry above the removal against the entry below the removal;

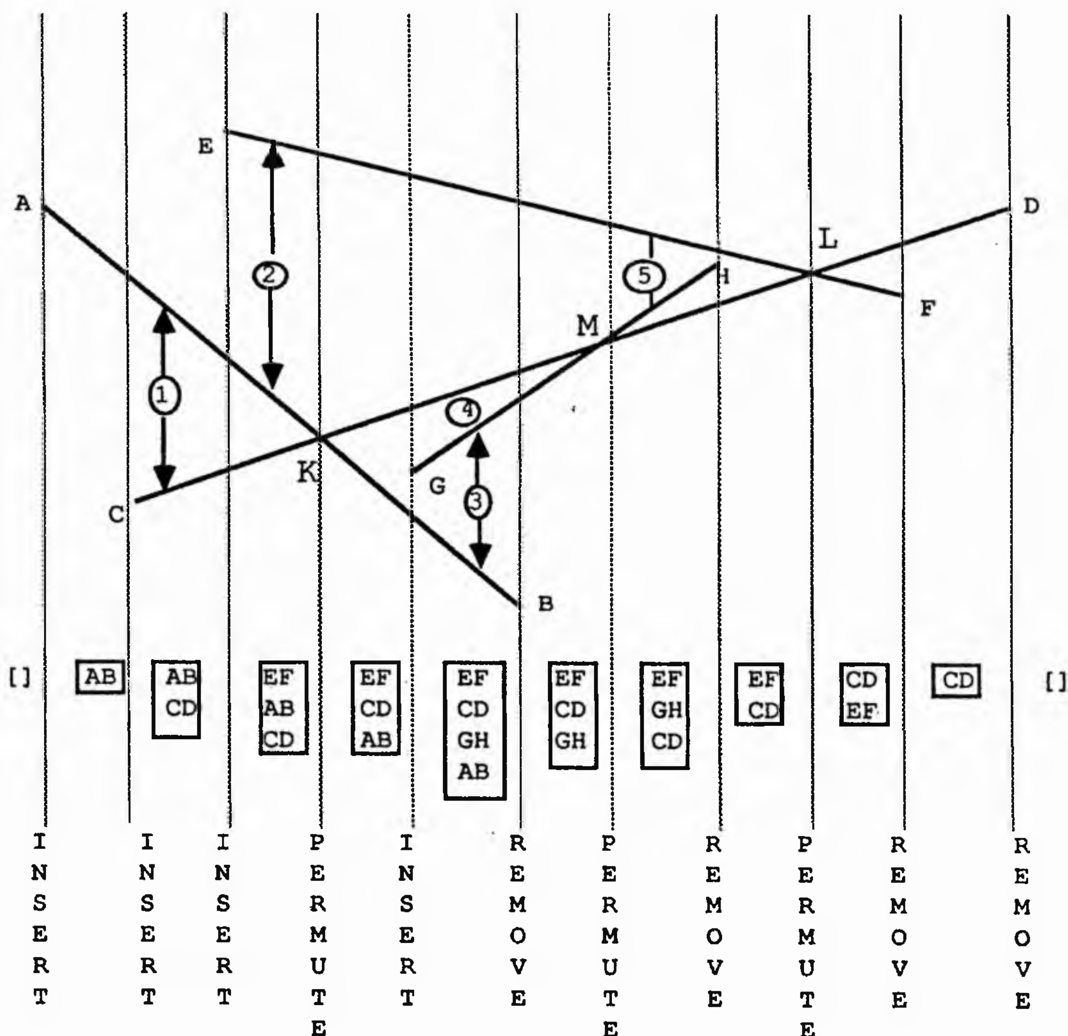
- after a permutation, test the top-most entry coming out of the Junction against its higher neighbour in the Front, and test the bottom-most entry coming out of the Front against its lower neighbour;

- after an insertion, test the newly inserted entry against its higher and lower neighbours.

The test for intersection, if it proves positive, calculates the intersection point and adds that Junction to the Transition list for future processing (fig 3.5).

The Front is a list of entries which must be capable of supporting the following two operations. SUCCESSOR which returns the higher neighbour of a given entry, and PREDECESSOR which returns the lower neighbour of a given entry.

INTERSECTION DETECTION USING A PLANE-SWEEP.



Series of 'Front' states across a set of 11 Transitions. Each Transition here only contains a single vertex. The 'Front' maintenance operation associated with that Transition is named below the Transition line.

Intersection tests are performed at circled numbers 1-5.
Test 1 gives point K, 2 gives L, 4 gives M. Tests 3 and 5 prove negative.

Initial Transition list:

```
[    [A,[AB]], [C,[CD], [E,[EF]], [G,[GH]],
     [B,[]],   [H,[]],  [F,[]],   [D,[]]    ].
```

After intersection point K discovered at test 1:

```
[    [A,[AB]], [C,[CD]], [E,[EF]], [K,[]], [G,[GH]],
     [B,[]],   [H,[]],   [F,[]],   [D,[]]    ].
```

Other intersection points (L & M) are inserted into the Transition list as they are found.

Figure 3.5

3.6.2.1 The Executive procedure.

The progression of the Front is controlled by an executive procedure called 'Advance_Front' which reads successive Transitions, reads through the list of Junctions for each Transition, and calls the Remove and Permute procedures. The Insert procedure is only called when the Junction list item contains a list of line-segments which start at that vertex.

```
Procedure Advance_Front;
begin
Read Transition from Transition_list;

While not( End-of Transition_list )
do   begin
     Read Junction from Junction_list of Transition

     While  not( End-of Junction_list)
     do   begin
          Remove  ( Junction, Front );
          Permute  ( Junction, Front );
          Read Line-segment from Insertion_list of Junction;

          While not( End-of Insertion-list )
           do  begin
               Insert (Line-segment, Front );
               Read Line-segment from Insertion_list of Junction;

               end;

          Read Junction from Junction_list of Transition;
          end;

     Read Transition from Transition_list;
     end;

end;
```

Pseudo-code outline of the Advance_Front procedure.

3.6.2.2 The Remove Procedure.

The Remove procedure climbs the Front structure evaluating the vertical height at that point along the X-axis of the line-segment referred to by each Front entry. The entries in the Front reference line-descriptors which detail the slope and intercept of each line-segment. The Remove procedure evaluates the vertical height of each line-segment at the current point along the X-axis using the slope-intercept formula :-

$$Y = (slope * X) + Intercept.$$

Any lines which pass through the current Junction are tested to see if their line-segment End-points lie there. If the line-segment does end at the current Junction, the entry referencing that line-segment is removed from the Front.

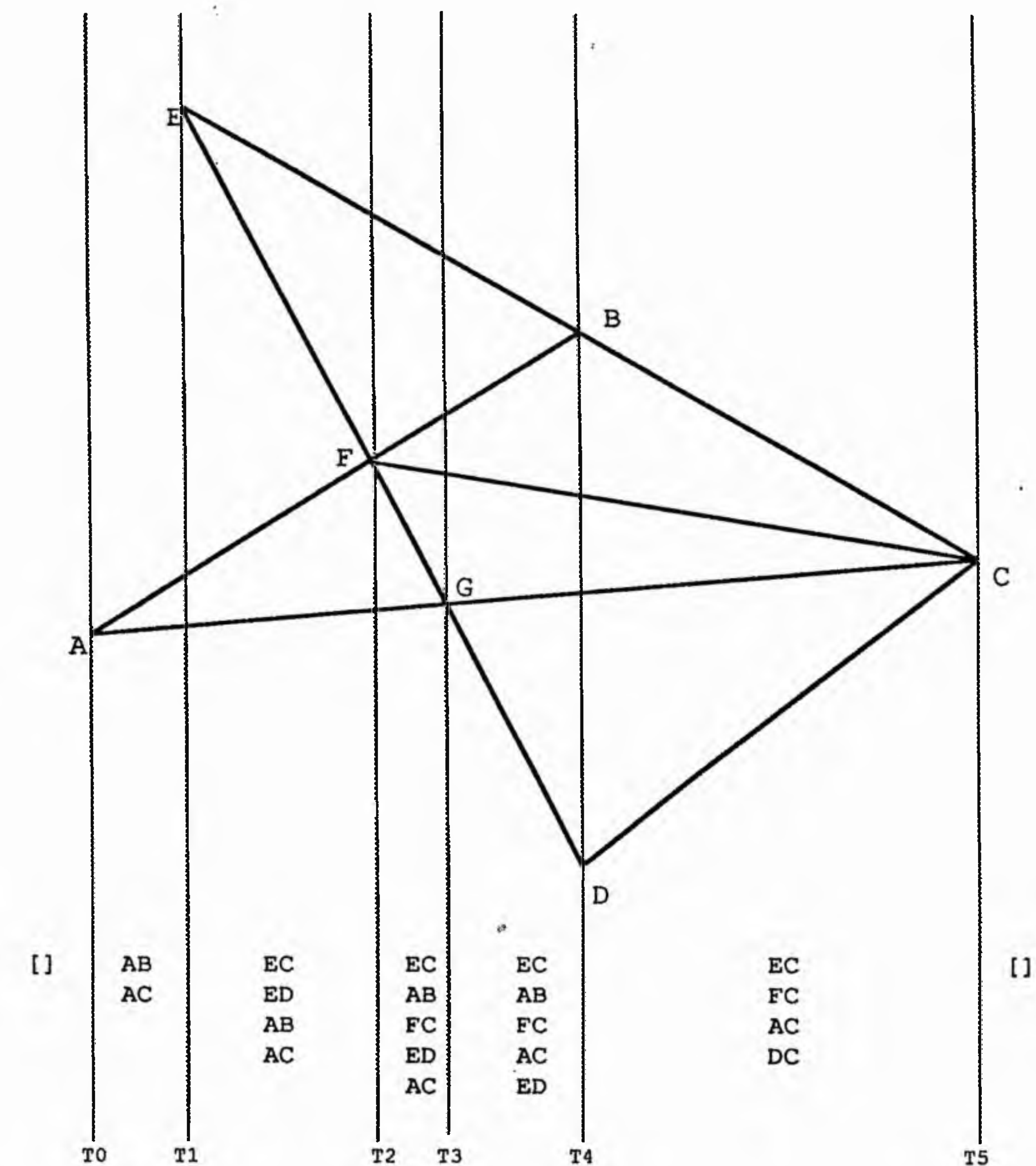
The Front is ordered according the the vertical ranking of the line-segments and so once a line-segment which passes above the current Junction is encountered, the Remove procedure can stop searching because all successive line-segments in the Front will pass even higher than that.

Figure 3.6 illustrates the progress of the Front across a scene. The search through the Front by the call to REMOVE is shown below each Transition line.

```
Procedure Remove ( Junction, Front );
begin
Read Current-entry from the tail entry of the Front;

While ( height of Current-entry < height of Junction )
do   Read Current_entry from next entry in Front;

While ( height of Current-entry = height of Junction )
do   begin
     If    ( End-point of Current-entry = Junction )
     then  Remove Current-entry from Front;
     Read Current-entry from next entry in Front;
     end;

Test-for-Intersection(  Current-entry,
                        predecessor( Current-entry ) );
end;
```

Pseudo-code outline of the REMOVE procedure.

The REMOVE procedure.



T0(A)	T1(E)	T2(F)	T3(G)	T4(D)	T4(B)	T5(C)
end>[]	end>				end>	end>
	2 ->AB	end> EC	EC	EC	4 ->EC	4 ->~~EC~~
	1 ->AC	3 -> ED	AB	AB	3 ->~~AB~~	3 ->~~FC~~
		2 -> AB	end>FC	FC	2 ->FC	2 ->~~AC~~
		1 -> AC	2 ->ED	end>AC	1 ->AC	1 ->~~DC~~
			1 ->AC	1 ->~~ED~~		

Remove is called seven times above. The tables show the transition number and the vertex within the transition. The order of the Front and the progress of Remove's climb is shown with numbered arrows, the climb ending when an entry higher than the vertex is found or the top of the Front is reached.

Entries are only removed from the Front during the calls for T4(D), T4(B) and T5(C).

Figure 3.6

3.6.2.3 The Permute procedure.

The Permute procedure similarly climbs the Front, this time looking for line-segments which pass through the current Junction but which do not have their end-points there. The Remove procedure has already removed any line-segments which pass through the current Junction and end there, and so any line-segments Permute encounters which pass through the Junction are line-segments which do not end there.

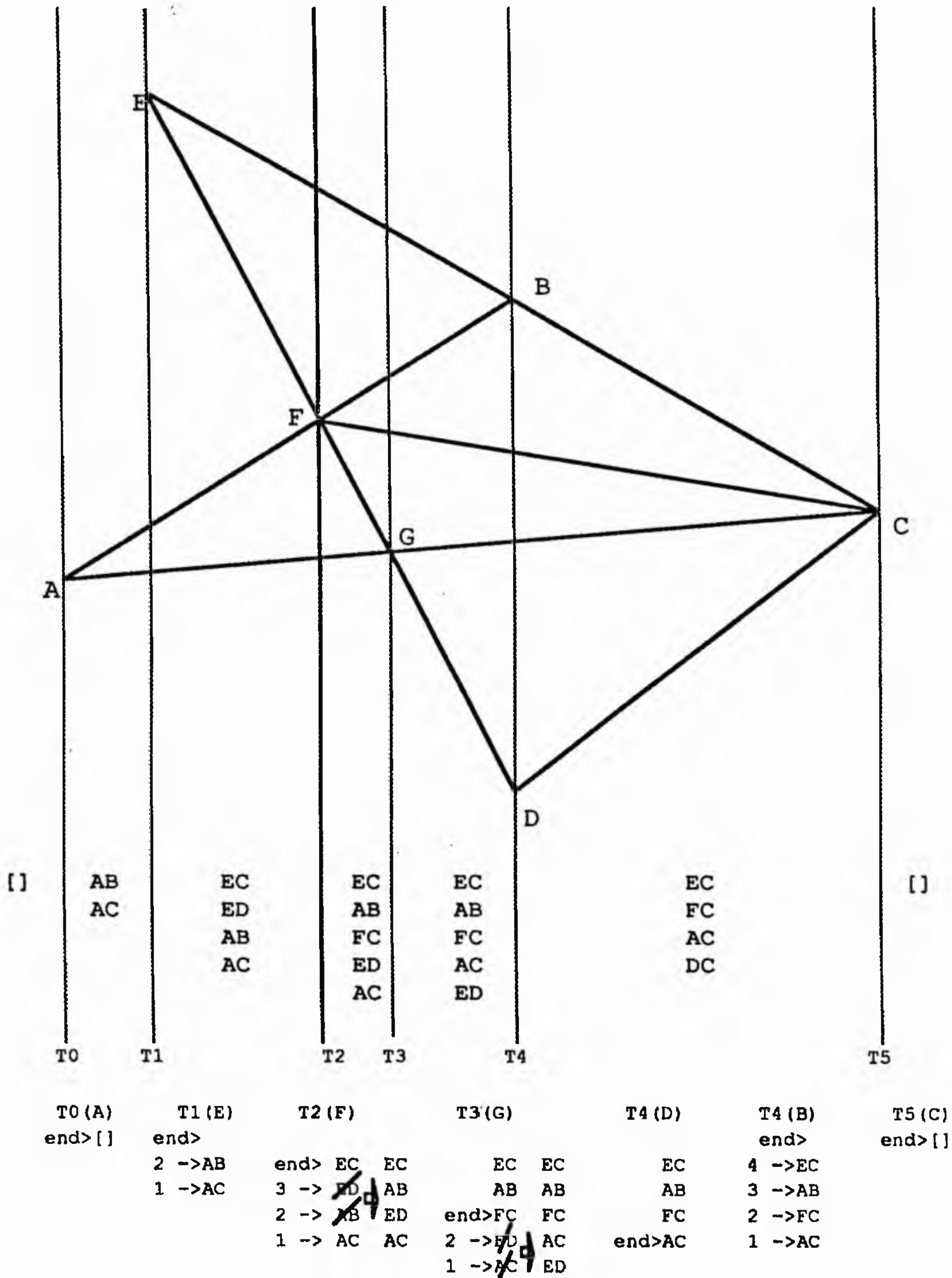
Owing to the ordering of the Front, any line-segments which do pass through the Junction will all be clustered together. The Permute procedure locates the bottom-most and top-most entries of this cluster, if it exists, and then reverses the order of the entries in the cluster. Top-most is swapped with bottom-most; second top-most with second bottom-most and so on. Having adjusted the order of the entries in the Front, the Permute procedure ends. Figure 3.7 illustrates the operation of the PERMUTE procedure.

NOTE :- Uses variables Lowest-entry and Highest-entry to store the addresses of the bottom-most and top-most entries of the cluster of entries which are to be reversed.
Uses variables low-swap-sub and high-swap-sub to respectively climb and descend the cluster of entries during the loop which reverses the order of the entries.
Calls procedure SWAP which exchanges the contents of two entries in the Front.

```
Procedure Permute ( Junction, Front );
begin
Read Current-entry from the tail entry of the Front;

While ( height of Current-entry < height of Junction )
do   Read Current_entry from next entry in Front;

if   ( height of Current-entry = height of Junction )
then begin
     lowest-entry := Current-entry;

     While ( height of Current-entry = height of Junction )
     do   begin
          highest_entry := Current_entry;
          Read Current_entry from next entry of Front;
          end;

     low_swap_sub <- lowest-entry;
     high_swap_sub <- highest-entry;

     While ( low-swap-sub <> High-swap-sub )
     do   begin
          swap( low-swap-sub, high-swap-sub );
          low-swap-sub  := successor( low-swap-sub );
          high-swap-sub := predecessor( high-swap-sub );
          end;

     Test-for-Intersection(    Highest-entry,
                               successor( Highest-entry ));

     Test-for-Intersection(    Lowest-entry,
                               predecessor( Lowest-entry ) );

     end;
end;
```

Pseudo-code outline of the PERMUTE procedure.

The PERMUTE procedure.



Seven calls to the permute procedure. The Front is searched from bottom to top for lines passing through the current vertex (in brackets). Only in calls for vertex F and vertex G do any lines get permuted.

Permute is always called AFTER Remove for a given vertex. Therefore in the calls for Transition 4 (T4), line ED has already been removed, and for Transition 5, all the lines have already been removed.

Figure 3.7

1 OCT 1996

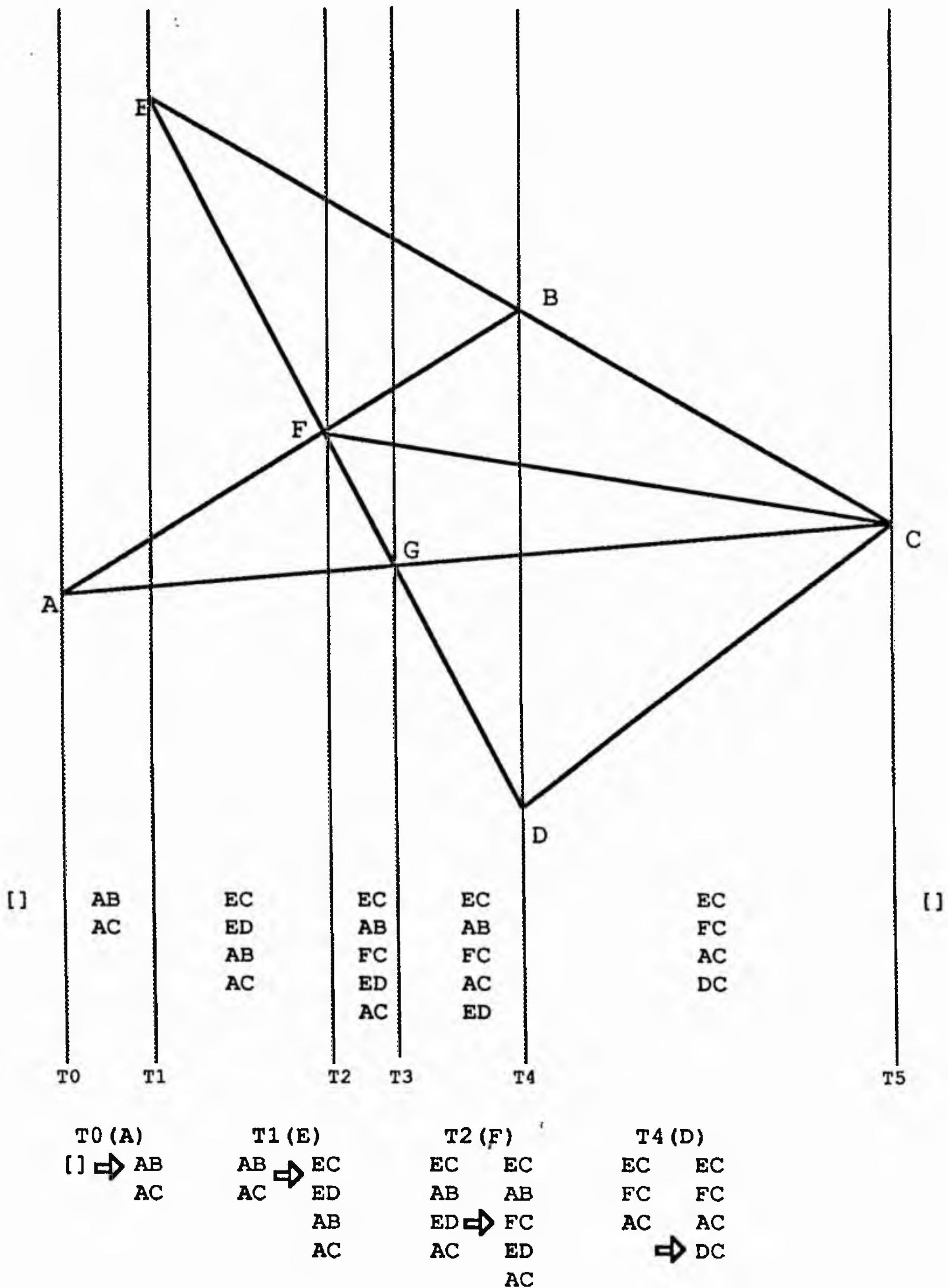
3.6.2.4 The Insert procedure.

The Insert procedure is called once for each line-segment appearing in the list of insertions attached to the current Junction. The Insert procedure climbs the Front looking for the place to insert each of the line-segments. If it finds any entries in the Front which pass through the Junction, it must search through the referenced line-segments, comparing their gradients, until it finds the entry with the gradient nearest to that of the line-segment which is to be inserted. The line-segment is inserted next to the nearest entry; above if the nearest entry has a lower gradient or below if the nearest entry has a higher gradient. If the Insert procedure does not find any entries in the Front which pass through the current Junction, the line-segment is inserted in the Front immediately below the next higher entry in the Front. The Insert procedure is illustrated in Figure 3.8.

```
Procedure Insert( Line-segment, Front );
begin
Read Current-entry from the tail entry of the Front;

While  ( height of Current-entry < height of Line-segment )
or    ( ( height of Current-entry = height of Line-segment )
                  and
        ( slope of Current-entry < slope of Line-segment ) )
do   Read Current_entry from next entry in Front;

Create New-entry;
Assign Line-segment to New-entry;
Attach New-entry between Current-entry
                         and predecessor of Current-entry;
Test-For-Intersection(  Successor( New-entry ), New-entry );
Test-For-Intersection( Predecessor( New-entry), New-entry );
end;
```

Pseudo-code outline of the INSERT procedure.

The INSERT procedure.



INSERT procedure is called AFTER Remove and Permute for a given vertex. Insert is called only for vertices where line-segments start, which in the case above are the four vertices A, E, F and D.

Insert climbs the Front looking for a line which passes above the start vertex of the new line. The new line is placed below its higher neighbour in the Front. Where two or more lines share the same start vertex, they are arranged in order of increasing slope (see vertices A and E).

Figure 3.8

3.6.3 Maintaining the Region-lists.

From the point of view of a horizontal plane sweep, any general Junction can be considered to have two sides, a left side and a right side.

On the left side of the Junction, none, one or more line-segments come together and meet at the Junction. When more than one line-segment meet, the Region-lists between adjacent pairs of these line-segments also meet. Region-lists which meet are concatenated together. Therefore, as the Remove procedure climbs the Front, it needs to concatenate the Region-lists of those adjacent line-segments which pass through the current Junction. The concatenation is always that of the Region-list above the lower line-segment to the Region-list below the upper line-segment. The Remove procedure performs this concatenation to all line-segments it encounters passing through the current Junction, whether they end there or intersect there.

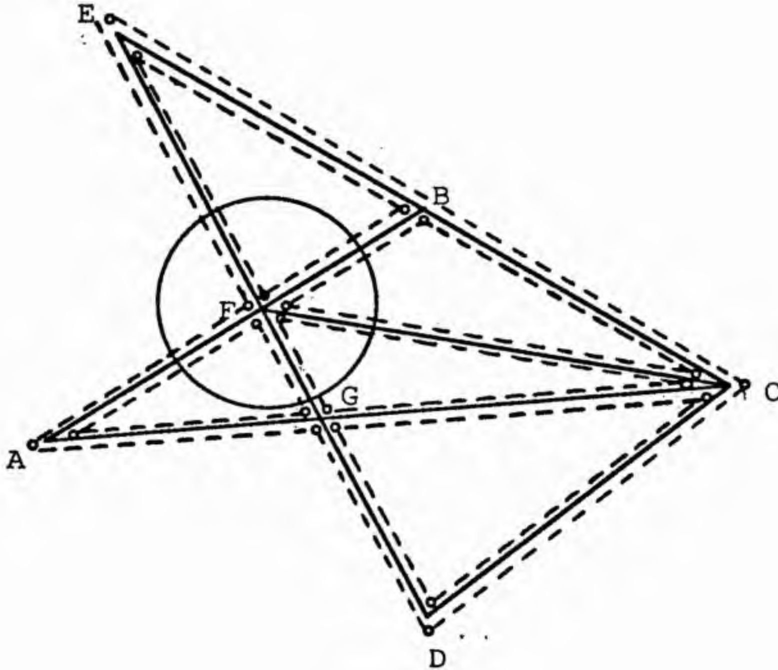
On the right side of the Junction, none, one or more line-segments pass through the Junction and diverge away from each other to the right. Some of these diverging line-segments are line-segments which intersect at the Junction, and others have their Start-points at the Junction. Between these diverging line-segments new Region-lists begin. The Permute procedure is responsible for initialising new Region-lists between line-segments which intersect at the Junction. The Insert procedure is responsible for initialising new Region-lists between line-segments which start at the Junction.

The Region list maintenance performed across a complex vertex with both a left-hand side and a right-hand side is shown in figure 3.9.

The complicated part of Region-list maintenance is concerned with the two Region-lists which pass across the Junction from left to right. These are the Region-lists which are above the top-most line-segments and below the bottom-most line-segments which pass through the Junction. Three cases have to be considered when dealing with these lists:-

i) The Junction has a left-hand side and a right-hand side. The top-most and bottom-most region-lists entering the Junction from the left must be extended to become the top-

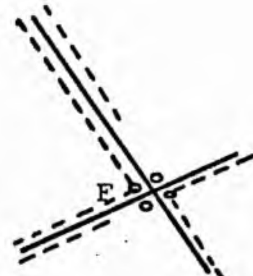
Region list maintenance across a vertex.



Three vertex types:-
(i) Lines on the left and lines on the right (F,G,D & B).
(ii) Lines on the left only (C).
(iii) Lines on the right only (A, E).

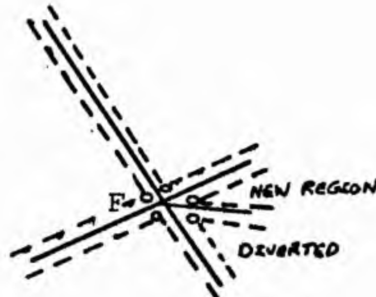
Left-hand side lists converge. As REMOVE sees vertex F.

Two converging lists are concatenated by REMOVE.



Right-hand side:- new list created by PERMUTE after front re-ordered.

Lists passing across the Junction from left to right. Top-most on left becomes top-most on right. Bottom--most on left becomes bottom-most on right.



INSERT diverts and creates lists:-

New line FC causes list between FB and FE to be diverted beneath FC. New list created above FC.

Figure 3.9

most and bottom-most lists exiting the Junction to the right.

ii) The Junction has a left-hand side but no right-hand side. The top-most and bottom-most region-lists entering the Junction from the left must be joined together to flow around the right-hand side of the 'butt-end' which is this Junction.

iii) The Junction has a right-hand side but no left-hand side. There are no region-lists to pass from the left to the right of the Junction, so a new list must be formed around the left-hand side of this 'butt-end'. The head and tail of this new list become the bottom-most and top-most region-lists exiting the Junction to the right.

The complicated aspect of these situations is that by dividing the processing of the Transition into a Remove procedure, which deals exclusively with the left-hand side of the Junction, and into Permute and Insert procedures, which deal with the right-hand side, is that these procedures need to communicate with each other to decide which of the three cases (i - iii) fits the current Junction.

The Remove procedure is called first, and so must communicate to the Permute and Insert procedures something about the situation it has encountered. The Remove procedure might find one or more line-segments which pass through the Junction, in which case it must communicate the region-lists above the top-most line-segment and that below the bottom-most line-segment. The Permute and Insert procedures need access to these region-lists in order to attach them to the top-most and bottom-most line-segments exiting the Junction.

Alternatively, the Remove procedure might find that no line-segments pass through the Junction. This fact must communicated to the Insert procedure so that it knows that case (iii) occurred and that it must initialise a new Region-list. The Permute procedure need never be called in this situation.

The Permute procedure is the next to be called. This procedure swaps the order of any remaining line-segments in the Front which pass through the current Junction. The Permute procedure only performs Region-list maintenance if it encounters one or more line-segments which must be reversed. Encountering one line-segment may seem a trivial case in that the single line-segment swaps places with itself, but the Region-list maintenance

is significant and must be performed.

Two Region list maintenance tasks are performed by Permute. First, it initialises new Region-lists between adjacent pairs of line-segments exiting the Junction. The tail of each new list is attached to below the upper line-segment of each pair, and the head is attached to above the lower line-segment. Second, it attaches the Region-lists passed to it by the Remove procedure to the top-most and bottom-most line-segments exiting the Junction.

The Insert procedure is called once for each line-segment which is to be inserted into the Front. Having located the position where the line-segment is to be inserted, one of four cases will be observed.

a) The new line-segment does not touch any other line-segment in the Front. In this case, the Insert procedure must consult the communication sent by the Remove procedure. If Remove encountered no line-segments, then the current Junction is a 'butt-end' and a new Region-list must be initialised and the ends of the list attached to the two sides of the line-segment. If Remove did encounter some line-segments, then the Region-lists from above the top-most must be extended and attached above the new line-segment, and the list from below the bottom-most must be extended and attached below the new line-segment (Fig. 3.10(i) Top).

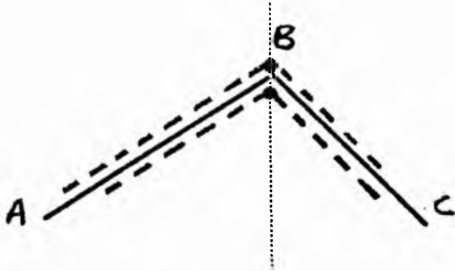
b) The new line-segment touches other line-segments in the Front and is the highest line-segment exiting the Junction. In this case, the Region-list attached above the second-highest line-segment must be detached and re-attached above the new line-segment. A new Region-list must be initialised between the new line-segment and the second-highest (Fig. 3.10(i) Bottom).

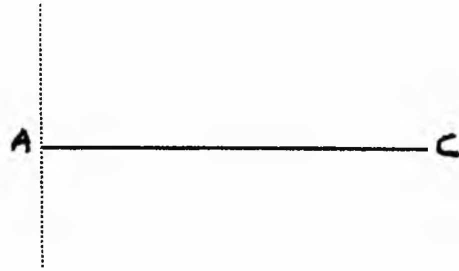
c) The new line-segment touches other line-segments in the Front and is the lowest line-segment exiting the Junction. Here the Region-list attached below the second-lowest line-segment must be detached and re-attached below the new line-segment. A new Region-list must be initialised between the new line-segment and the second-lowest (Fig 3.10(ii) Top).

d) The new line-segment touches other line-segments in the Front but is neither the

INSERT Region List maintenance.

CASE A :- new line is the only line at the vertex.

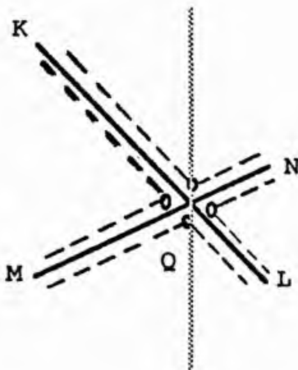


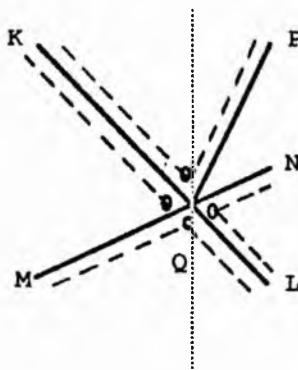


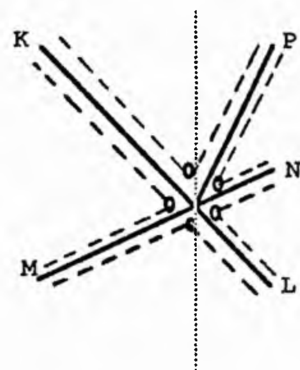
All previous entries have been REMOVEd.
Front at vertex B.
Line entry AB has been removed from the Front.
New line BC must inherit lists Above and Below AB.

No previous entries near vertex.
Front at 'butt-end' vertex A.
No lines existed at A and so new region list created around AC.

CASE B :- new line is highest of many lines at the vertex.







Before new line QP inserted.

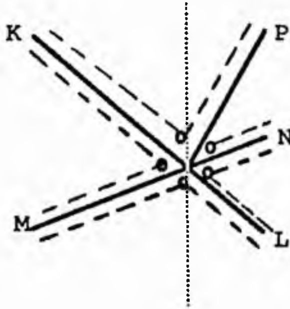
List above previous top-most line diverted around new top-most line QP.

New list initialised between new insertion QP and lower neighbour QN.

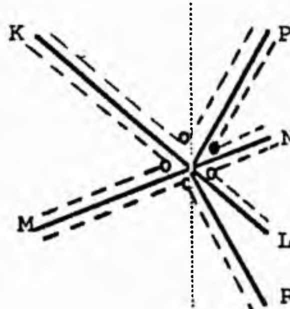
Figure 3.10(i)

INSERT Region List maintenance (continued).

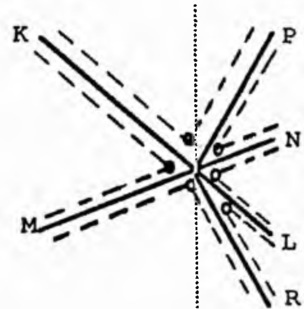
CASE C :- new line is LOWEST of many lines at the vertex.



Before new line QR inserted.

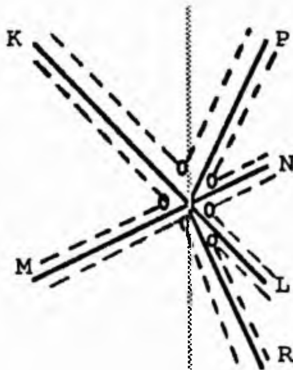


List BELOW previous bottom-most line diverted below new bottom-most line QR.

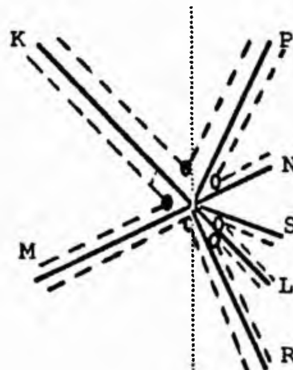


New list initialised between new insertion QR and higher neighbour QL.

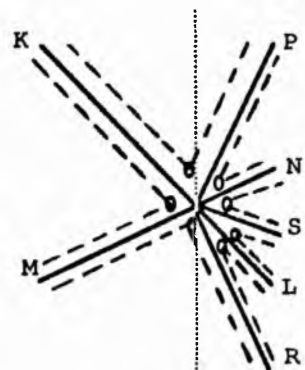
CASE D :- new line is in the midst of many lines at the vertex.



Before new line QS inserted.



List BELOW higher neighbouring line QN diverted below new line QS.



New list initialised between new insertion QS and higher neighbour QN.

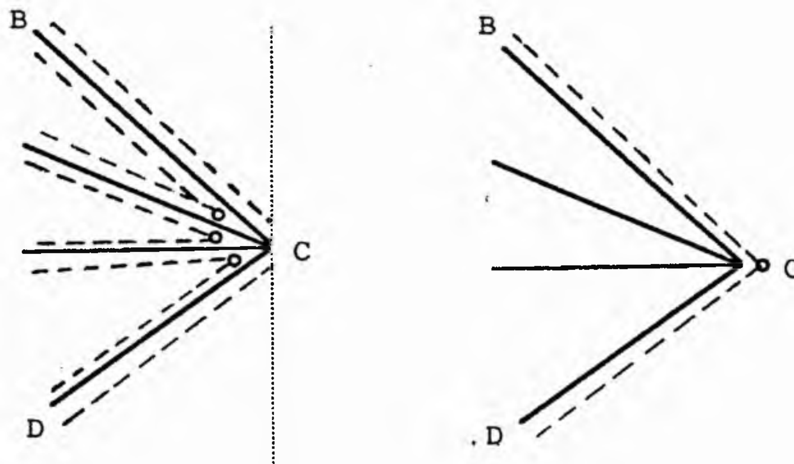
Figure 3.10(ii)

highest nor the lowest line-segment exiting the Junction. The Region-list attached below its higher adjacent line-segment must be detached and re-attached below the new line-segment. A new Region-list must be initialised between the new line-segment and its higher adjacent neighbour (Fig. 3.10(ii) Bottom).

Cases (a), (b) and (c) involve some processing of one or both of the Region-lists passing from left to right (if they exist). In cases (b) and (c), these lists are merely diverted from what at one point in the processing appeared to be the highest or lowest lines to what currently appears to be the highest or lowest lines. These diversions could occur many times in the processing of one Junction. Case (d) has no involvement with the Region-lists which pass from left to right, although close examination of the Region-list processing involved shows that it is identical to that performed in case (c).

When the Junction is a 'butt-end' with no right-hand side - case (ii) - the Region-list above the highest line-segment entering the Junction must be concatenated with the Region-list entering the Junction below the lowest line-segment (Fig. 3.11). This concatenation describes the shape around the right-hand side of the Junction. This concatenation could be considered to be part of the processing of the left-hand side of the Junction and so would appear to naturally belong to part of the Remove procedure. In fact it is included as part of the executive procedure performed after the right-hand side procedures, Permute and Insert, have been executed. This concatenation is only required when the current Junction has no right-hand side, and the decision that no right-hand side exists can only be made after the Permute and Insert procedures have found respectively that no line-segments pass through the Junction and that no line-segments start at the Junction. The Permute and Insert procedures need to communicate whether a right-hand side exists to the executive procedure to enable it to decide whether to perform the concatenation of the two Region-lists. These Region-lists, if they exist, are already required to be returned by the Remove procedure for reasons previously discussed.
Pseudo-code for the Remove, Permute and Insert procedures, extended to cope with General Vertex Types, is given in Appendix A.

Butt-end With No Right-hand Side.



After REMOVE Region Lists 1,2 & 3 have been maintained.

After PERMUTE & INSERT - no lines on right-hand side of Junction so concatenate list above top-most line (BC) with list below bottom-most line (DC).

Figure 3.11

3.7 Extensions required for coping with vertical lines.

As far the horizontal plane-sweep is concerned, a vertical line is an exceptional entity. It confounds the right-most / left-most categorisation of end-points and it has an absurd slope and intercept. It does not exist between Transitions.

For these reasons, vertical line-segments are not to be thought of as line-segments but rather as an extended Junction within a Transition. A single Transition might contain many separate vertical line-segments (Fig. 3.12).

The first extension to the algorithm is obviously to amend the items in the Junction list of the Transitions to enable them to describe vertical line-segments when necessary. For the sake of consistency then a vertex can be thought of as a trivial vertical line-segment with its bottom-most vertex in the same place as its top-most vertex. Items in the Junction lists now all describe vertical line-segments, many of them trivial single-point line-segments. The Junction lists are ordered on ascending bottom-most vertex. Figure 3.13 provides an example of the new Transition list structure.

In order to simplify the preparation of Transitions, Insertion lists, which always contain line-segments which are not vertical, are always attached to Junction list entries which hold the Starting-point of these line-segments as its bottom-most vertex. The Insertion list of a Junction list entry for a vertical line-segment only contains those line-segments with their Starting-point at the bottom-most vertex of the vertical line-segment. Any other line-segments whose Starting-points lie within the vertical line-segment, excepting those at the bottom-most vertex, are stored in the Insertion lists of separate Junction list entries. This arrangement avoids the complications which could arise if the Insertion lists describing all the line-segments starting within a vertical line-segment were to be stored in the same Junction list entry. Problems could arise under that arrangement when successive vertical line-segments touch or overlap each other.

Processing a vertical line-segment Junction entry is similar to that described in the last section for simple single-point Junction entries. Processing is again divided into left-hand side and right-hand side procedures.

Vertical lines overlap single-point Junctions.

A scene incorporating 4 vertical line-segments AB, CD, EF & GH.

Line-segment AB lies across 8 single-point Junctions A, K, L, M, N, O, P, B.

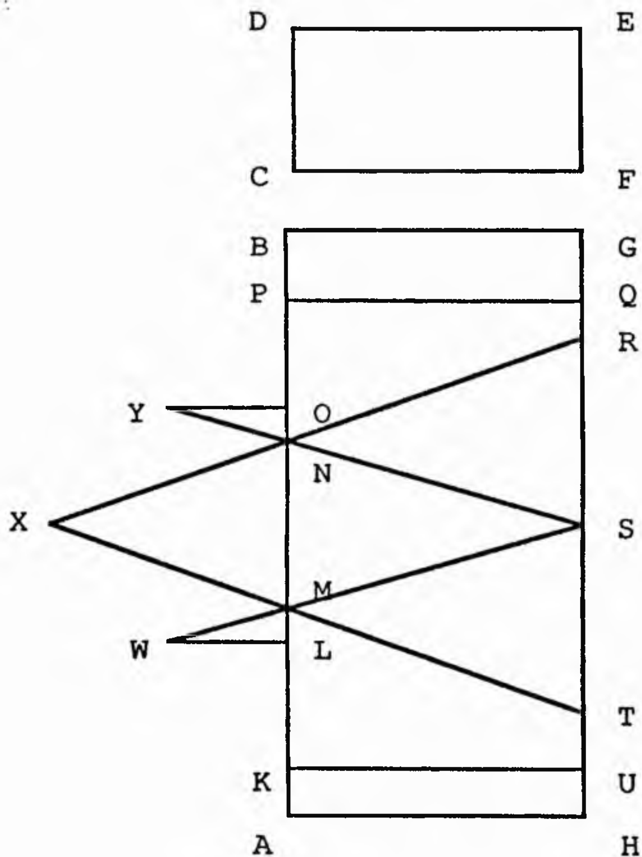
Junctions A, K, P, B are points where new entries are inserted into the Front.

Junctions L & O are points where lines YO and WL are removed from the Front.

Junction N is the vertex where lines XR and YS swap places in the Front, and Junction M is where lines XT and WS swap places.

Other vertical lines:-
- CD - two insertions at Junctions C and D;
- EF - two removals at Junctions E and F;
- GH - eight removals at Junctions G, Q, R, S, T, U and H.

Figure 3.12



Transition List incorporating vertical Lines.

```
[   [X,[XR,XT],,],
    [W,[WS,WL],,],
    [Y,[YO,YS],,],
    [A,[AH],AB],
    [K,[KU],,],
    [P,[PQ],,],
    [B,[BG],,],
    [C,[CF],DC],
    [D,[DE],,],
    [H,[],GH],
    [F,[],EF]      ].
```

Format of List item:- [vertex,[insertions],vertical line].

Lines with End-points lying within a vertical line are now REMOVE'd by the call for that vertical line - hence no entries are needed for vertices U,T,S,R,Q,G within GH.

Intersection points M & N are not inserted into this list until they are discovered by the plane-sweep.

Figure 3.13

In sections 3.7.1 to 3.7.4, the principles of operation is described for the Advance-Front, Remove, Permute and Insert procedures. Pseudo-code outlines of these procedures can be found in appendix B.

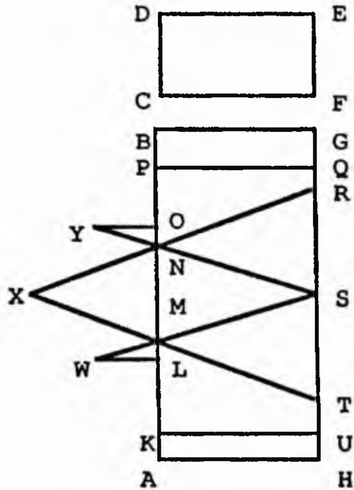
3.7.1 Advance-Front.

The executive procedure is changed to deal with vertical line-segments. Only one call to REMOVE and PERMUTE are made for each Vertical Edge, regardless of how many single point junctions occur within the vertical edge (Fig. 3.14). The most noticeable change is in the way the Insert procedure is called. Insert is now not only called for each entry in the Insertion list of the current Junction, but also for all the entries of all the Insertion lists of all the Junctions in the current Transition which lie within the current vertical line-segment. In the trivial case, when the vertical line-segment is a single-point, this will only include one Junction list entry and one Insertions list, if any.

More subtle changes include the necessity to store the top-most vertex of any vertical line-segments encountered. This enables the routine controlling the Insert procedure to make the evaluation needed to know when it has encountered a vertex which is outside of the current vertical line-segment.

The 'butt-end' processing has also changed slightly. In the event that a 'butt-end' is encountered which extends along a vertical line-segment, the Region-list manipulations required, either initialisation in the case of right-handed Butt-ends (Fig. 3.15) or concatenation in the case of left-handed Butt-ends (Fig. 3.16), must extend the Region-lists to include both vertices of the vertical line-segment.

Advance Front.




```
[vertex,[insertions],vertical line].

     [   [X,[XR,XT],,],
         [W,[WS,WL],,],
         [Y,[YO,YS],,],
         [A,[AH],AB],
         [K,[KU],,],
         [P,[PQ],,],
         [B,[BG],,],
         [C,[CF],DC],
         [D,[DE],,],
         [H,[],GH],
         [F,[],EF]        ].
```

Transition list.

Advance-Front calls REMOVE & PERMUTE for Transition List entries for vertices X,Y,W,A,C,H & F. Four of these entries - A,C,H & F - refer to vertical lines, and the calls to REMOVE and PERMUTE for these lines cover all the single-point vertices lying within them.

Advance-Front calls INSERT for entry containing an Insertions List - ie. all entries except F and H. An entry is required for each vertex where lines start - even if the vertex is lying within a vertical line. Lines with their start-point on the lowest vertex of the vertical line are included in the insertions list of the entry containing the vertical line - eg entries A and C.

Figure 3.14

Right-hand 'Butt-end' Processing.

If, after calling REMOVE for a given Transition List entry it is realised that no left-hand side exists at that Junction - as occurs at vertices X,Y & Z and vertical line CD above - then ADVANCE-FRONT initialises Region-lists for the outsides of the new shapes.

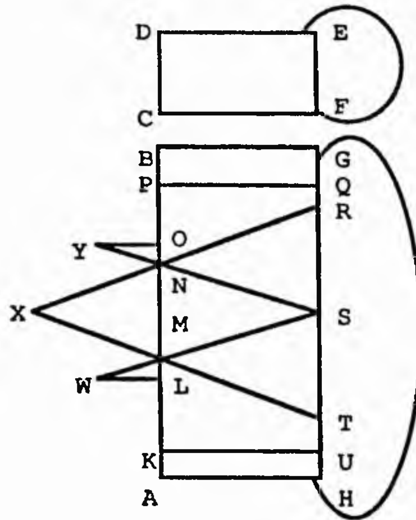
These new Region-lists contain a single vertex in the cases of vertices X,Y & Z, and contain two vertices in the case of vertical lines such as CD.

The new Region-lists are not attached to any entries in the Front. The assumption here is that the Junction had no left-hand side and so it must have a right-hand side. The right-hand side of the Junction is not represented in the Front until after INSERT has placed new entries in the Front at that Junction. The new Region-lists are passed to INSERT so that it can attach them to the new entries as they are placed in the Front.

The assumption that the Junction has a right-hand side may prove false in the case where a vertical line exists which does not touch any other line. The new Region-list is formed around the vertical line, but INSERT is not called because there is no Insertions list. The Left-hand butt-end processing, also performed by ADVANCE-FRONT and described later, then closes this region-list around the right-hand side of the vertical.

Figure 3.15

Left-hand 'Butt-end' Processing.



If, after all Front entries and Transition list entries for a given Junction have been processed, it is discovered that no right-hand side exists, then ADVANCE-FRONT must concatenate the Region-lists above the top-most line and below the bottom-most line on the left-hand side.

In the above scene, 2 left-hand butt-ends exist, both around vertical line segments - CD and GH.

* Region list above DE is joined to that below CF

* Region list above GH is joined to that below AH.

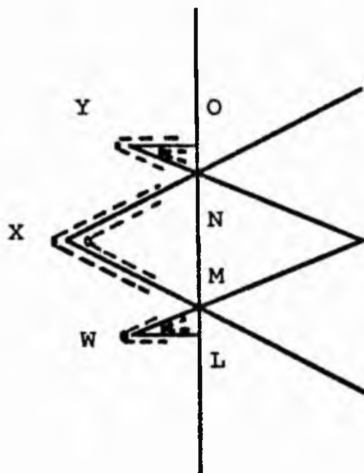
Figure 3.16

3.7.2 Remove.

A modified version of the Remove procedure climbs through the Front looking for all the line-segment entries which pass through the vertical line-segment, removing those which have their End-point there (Figures 3.17(i) and 3.17(ii)).

The Region-list processing performed by Remove is only slightly altered. Remove still identifies the top-most and bottom-most Region-lists falling within the length of the vertical line-segment, but now must ensure that these Region-lists incorporate the top-most and bottom-most vertex of the vertical line-segment, extending them to do so if necessary. Remove still concatenates the Region-lists between adjacent pairs of line-segments which fall within the Junction; however, the Junction is now a vertical line-segment, and so two vertices must be added to adjacent line-segments which both fall within the vertical line-segment, but are vertically separate and do not share the same End-point.

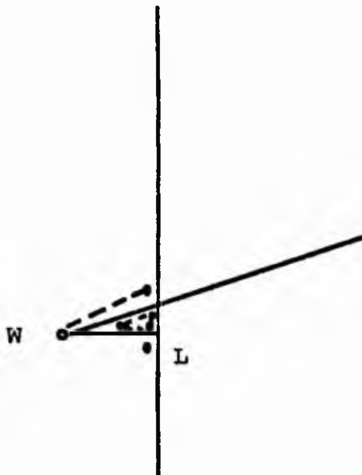
REMOVE incorporating Vertical line and Region-list processing.



State of Region lists before 'REMOVE' is called.

First line found.

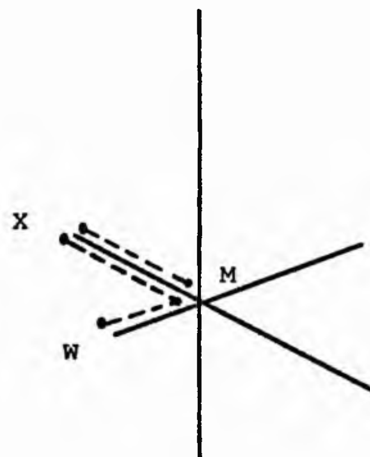
Extend lists above and below it to the Junction.



Next line found.

Below list extended down vertical to meet above list of previous line.

Above list extended to Junction.



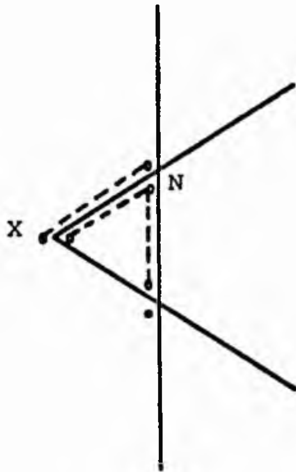
Next line.

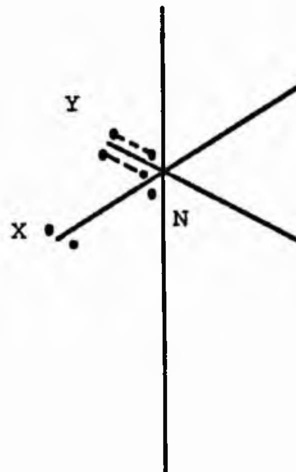
Below list joined to above list of previous line.

Above list extended to Junction.

Figure 3.17(i)

REMOVE incorporating Vertical line and Region-list processing (cont).





Next Line.

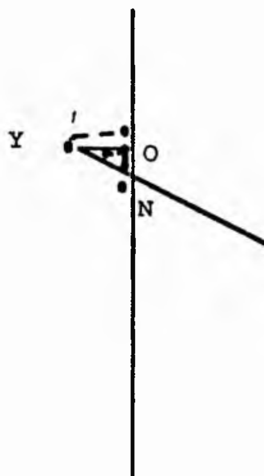
Below list extends down vertical to join to Above list of previous line.

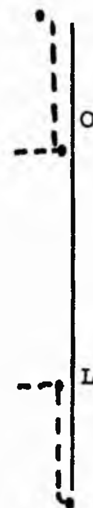
Above list extended to Junction.

Next Line.

Below list extends to join to Above list of previous line.

Above list extended to Junction.





Next Line.

Below list extends down vertical to join to Above list of previous line.

Above list extended to Junction.

No lines remain.

Extend above list of top-most to top vertex.

Extend below list of bottom-most list to bottom vertex.

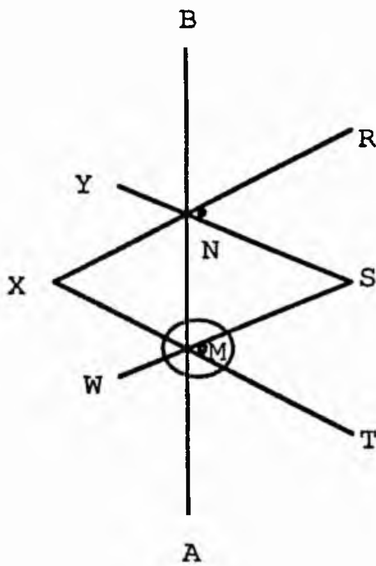
Figure 3.17(ii)

3.7.3 Permute.

A modified version of Permute climbs the Front looking for all those line-segments passing through the vertical line-segment which have been left after the Remove procedure (Fig. 3.18). An essential detail of this processing is that Permute must break these line-segments into groups of line-segments passing through the same point within the vertical line-segment. These individual groups are permuted as they are identified, and all groups within the vertical line-segment are processed so.

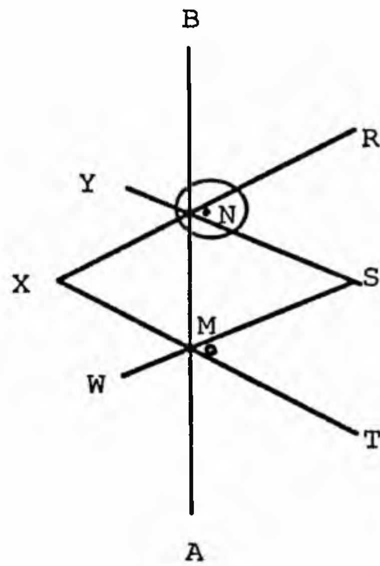
With regard to Region-list processing, Permute does the following. The top-most and bottom-most line-segments exiting the vertical line-segment are identified and the Region-lists passed forward from Remove (the top-most and bottom-most left-side Region-lists) are attached above and below these line-segments respectively. In some cases, when these line-segments are not at the top-most or bottom-most vertex of the vertical line-segment, these Region-lists must be extended by adding the vertex where the appropriate line-segment crosses the vertical. Permute must still initialise Region-lists between adjacent pairs of line-segments, but sometimes these adjacent pairs may be vertically separate in which case an additional vertex needs to be added to the new list to incorporate the points where the two line-segments cross the vertical.

PERMUTE with vertical line and Region-list processing.



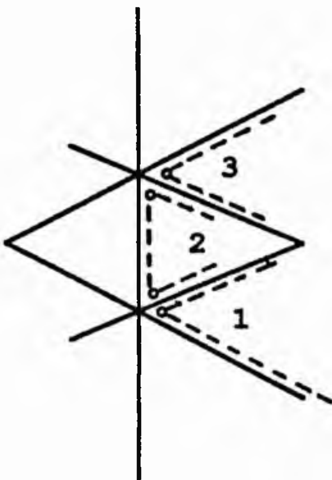
YS	⇒	YS
XR		XR
XT		WS
WS		XT

PERMUTE first intersecting group at M.



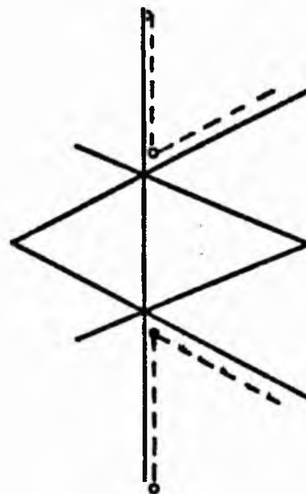
YS	⇒	XR
XR		YS
WS		WS
XT		XT

PERMUTE second intersecting group at N.



Region lists are initialised between adjacent pairs of lines which have been re-ordered.

Region lists 1 & 3 require only one vertex, whereas region list 2 requires two vertices to describe the vertical connection.



Lowest and highest region lists from left-hand side (set by REMOVE) are extended to meet the lowest and highest lines after the Permutations are performed.

Figure 3.18

3.7.4 Insert.

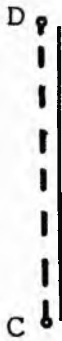
The Insert procedure changes only slightly. The only difference between this version and the previous version occurs when the new line-segment which is being inserted is found not to touch any other line-segment in the Front. Now Insert must examine the entries in the Front immediately above and below the new line-segment (Fig. 3.19).

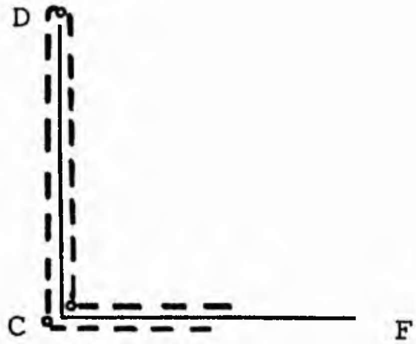
If neither of the two neighbours lie within the vertical line-segment, then Insert must extend the Region-lists passed to it by the Advance-Front procedure and attach these to the new line-segment, ensuring that both lists include the Start-point of the new line-segment.

If the higher of the two neighbours lies within the vertical line-segment, then the Region-list below the neighbour is split in two at the Start-point of the new line-segment. The lower part is attached below the new line-segment, and the higher part is used in the initialisation of a new list between the new line-segment and the higher neighbour.

If the lower of the two neighbours lies within the vertical line-segment, then the Region-list above the neighbour is split at the Start-point of the new line-segment. The higher part is attached above the new line-segment, and the lower part is used in the initialisation of a new Region-list between the new line-

Insert Procedure for Vertical Lines Incorporating Region-list Processing.

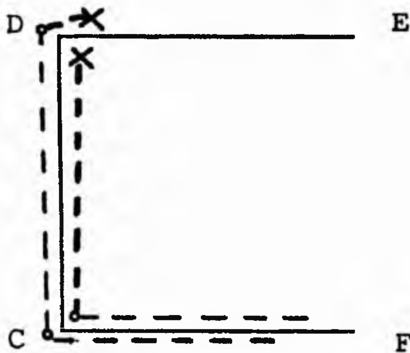


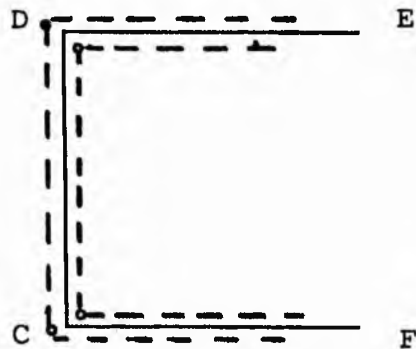


Left-side Region list initialised by ADVANCE-FRONT.

First line inserted has no neighbour.

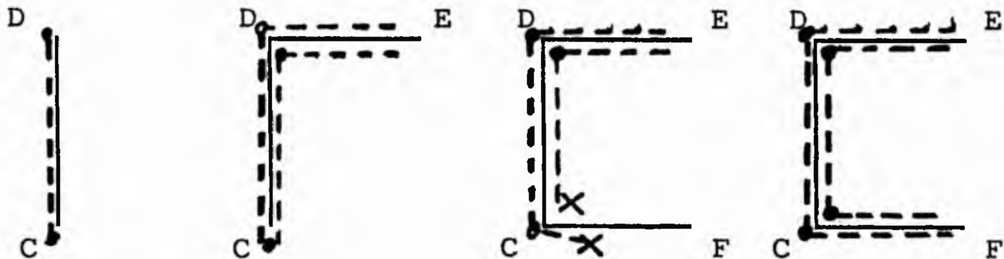
Extend left-side Region lists and attach to First line.





New line (DE) causes Region list around top vertex to be split at start point of new line (D).

Split Region list is extended to start point of new line (if necessary) and attached to new line.



These insertions could have been performed in the reverse order. Region-list processing remains the same.

Figure 3.19

3.8 Enabling the algorithm to construct an Adjacency Graph.

Sections 3.4 to 3.7 have described the extensions necessary to generalise the plane-sweep algorithm for reporting the regions inside a non-degenerate polygon. These extensions enable the algorithm to report all the regions inside a collection of separate or overlapping polygons regardless of the type of polygon, including "open" polygons. This algorithm is now capable of reporting all the regions inside a line-drawing comprised of straight line-segments.

Section 3.2 described the set of data structures needed to provide a drawing interpreter with sufficient information to construct its interpretations. So far we have made no attempt to construct these data structures with the extended plane-sweep algorithm. The algorithm as it stands constructs no more than a series of lists of vertices which describe the separate regions. No links are provided which might show which regions are adjacent to which, nor which regions contain which. The extensions needed to describe these relationships are introduced now. The construction of the Adjacency Graph is the subject of the remainder of this section, and the construction of the Containment Hierarchy is discussed in section 3.9.

The plane-sweep algorithm currently constructs its descriptions of Regions from ordered lists of the vertices of the Regions. We now seek to alter this so that Regions are described by Edge-lists. Section 3.2 described in detail two of the data structures which are the components of the Adjacency Graph, these being the 'Line-segment' and the 'Edge'. Here we show how these data structures can be fitted into the Plane-sweep algorithm.

3.8.1 Pre-processing - initialising the Adjacency Graph and preparing a new Transitions structure.

In its initial state the Adjacency Graph contains a set of unconnected Graph structures, each consisting of one Line-segment whose Edge-list contains only one Edge. A function of the plane-sweep algorithm is to connect these small Graphs together to form a small number of larger Graphs, these connections being made where Line-segments intersect.

The first task of the Pre-processing phase is to construct this initial state of the Adjacency Graphs. This is simply the procedure of reading each Line-segment from the file of drawing data and constructing one Line-segment structure for each one read. The Edge in the Edge-list of the Line-segment describes the single Edge of that Line-segment running from its Start-point to its End-point. Figure 3.20 shows a scene and its corresponding Adjacency Graphs, both before and after the Plane-sweep has discovered all the Edges.

The second task of the Pre-processing phase is to construct the Transition list. The only difference between the new Transitions structure and that used in the previous algorithms is that this time Edges are stored in the Insertion-lists of the Junction structures where previously Line-segments were stored. Vertical Edges pose the same difficulty as Vertical line-segments in the previous algorithm. This time, any Vertical Edges which start at the vertex are stored in a separate list inside the Junction structure - it is not enough to store just the Top-most and bottom-most vertices of the Vertical Edge because that Edge must be woven into the Adjacency Graph and so the plane-sweep procedures must be able to reference the actual Edge structure.

The new Junction structure contains the following elements :-

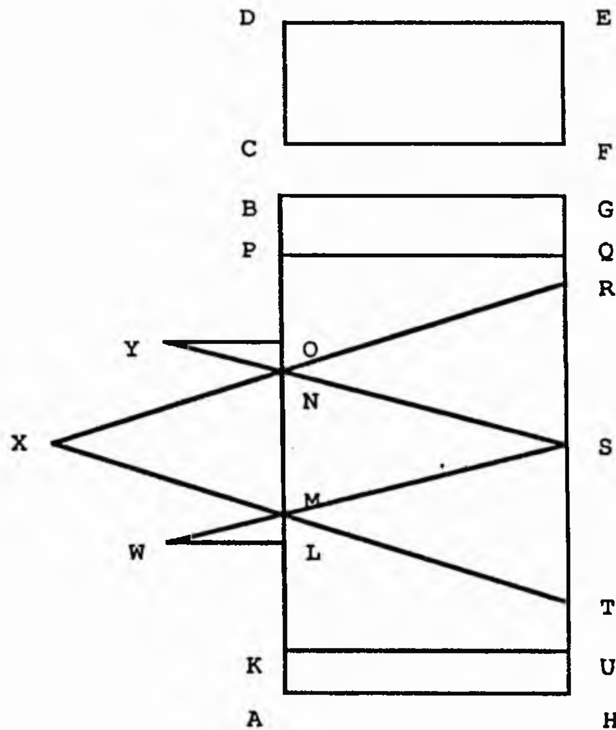
Bottom-most vertex;

Insertion-list containing references to Edges;

List of Vertical Edges;

Edges in the Insertion-lists are those Edges whose left-most vertices are at the vertex. Edges in the list of Vertical Edges are vertical Edges whose bottom-most vertex lie on the vertex. Figure 3.21 shows a scene and its corresponding Transition List.

Line and Edge-based drawing descriptions.



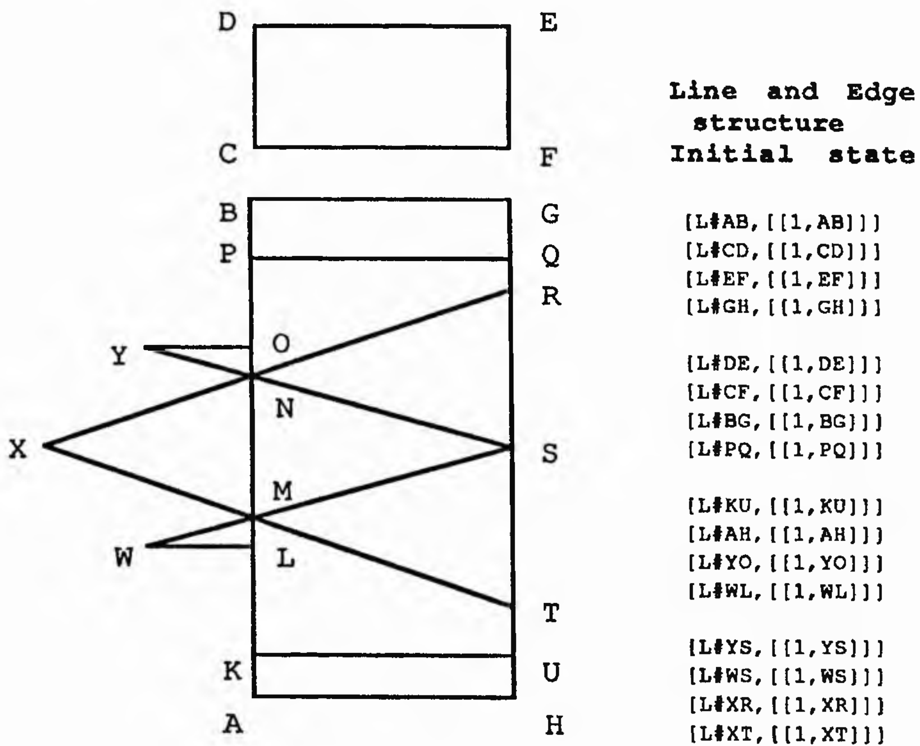
Line data structures for the above scene.

Initial state	Final state.
[L#AB,[[1,AB]]]	[L#AB,[[1,AK],[2,KL],[3,LM],[4,MN],[5,NO],[6,OP],[7,PB]]]
[L#CD,[[1,CD]]]	[L#CD,[[1,CD]]]
[L#EF,[[1,EF]]]	[L#EF,[[1,EF]]]
[L#GH,[[1,GH]]]	[L#GH,[[1,HU],[2,UT],[3,TS],[4,SR],[5,RQ],[6,QG]]]
[L#DE,[[1,DE]]]	[L#DE,[[1,DE]]]
[L#CF,[[1,CF]]]	[L#CF,[[1,CF]]]
[L#BG,[[1,BG]]]	[L#BG,[[1,BG]]]
[L#PQ,[[1,PQ]]]	[L#PQ,[[1,PQ]]]
[L#KU,[[1,KU]]]	[L#KU,[[1,KU]]]
[L#AH,[[1,AH]]]	[L#AH,[[1,AH]]]
[L#YO,[[1,YO]]]	[L#YO,[[1,YO]]]
[L#WL,[[1,WL]]]	[L#WL,[[1,WL]]]
[L#YS,[[1,YS]]]	[L#YS,[[1,YN],[2,NS]]]
[L#WS,[[1,WS]]]	[L#WS,[[1,WM],[2,MS]]]
[L#XR,[[1,XR]]]	[L#XR,[[1,XN],[2,NR]]]
[L#XT,[[1,XT]]]	[L#XT,[[1,XM],[2,MT]]]

Line data structure format :-
[line id, [[edge id, edge vertices],..]].

Edge lists are ordered from left to right, except for vertical lines, in which edge lists are ordered from bottom to top.

Figure 3.20



Transition List for Edge-based plane-sweep.

```
[  [X,[[L#XR,1],[L#XT,1]],,],
   [W,[[L#WS,1],[L#WL,1]],,],
   [Y,[[L#YO,1],[L#YS,1]],,],
   [A,[[L#AH,1]],L#AB],
   [K,[[L#KU,1]],,],
   [P,[[L#PQ,1]],,],
   [B,[[L#BG,1]],,],
   [C,[[L#CF,1]],L#CD],
   [D,[[L#DE,1]],,],
   [H,[],L#GH],
   [F,[],L#EF]      ].
```

Format of List item:-

```
    [vertex,[insertion list],vertical line].
```

Insertion list format :-

```
    [  [line id, edge id],..].
```

Figure 3.21

3.8.2. The Plane-sweep.

The basic ideas remain the same. A Front is maintained during a sweep from left to right across the plane. The plane-sweep passes across many Transitions where changes occur in the vertical ordering of lines in the Front. Each Transition consists of one or more discrete Junctions where localised changes occur in the Front.

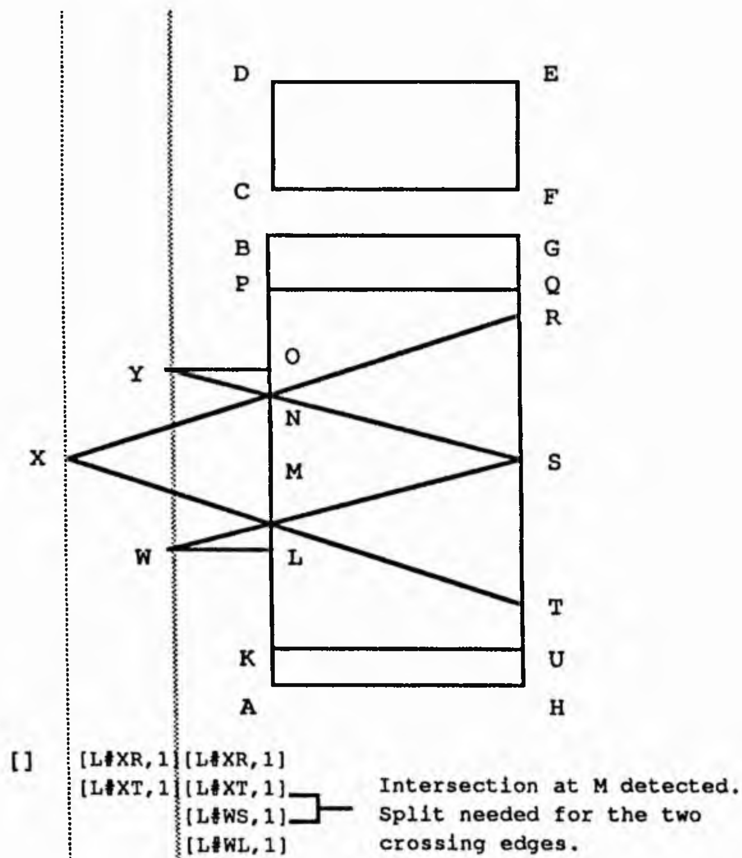
The changes to the Front structure at each Junction is done in two phases, one dealing with the line-segments entering the Junction from the left-hand side, and the other dealing with those exiting to the right-hand side. As the plane-sweep progresses, some additional processing is performed to construct descriptions of the Regions of space enclosed within the drawing.

Changes are needed now to enable the descriptions of Regions to be constructed in the form of a graphs of Edges as opposed to the Region-lists of vertices used in the previous algorithms. An initial set of Adjacency Graphs already exists. The task confronting the plane-sweep is to weave these unconnected Edges together to form more complex graphs reflecting the structure of the drawing. A fundamental part of the problem of weaving the Edges together is that of how are Edges which meet at a Junction to be joined together.

With regard to the definition of Edges given in section 3.2 and considering the Edges involved in an intersection separately, there are two types of intersections which can occur between Edges. The first type is where the Edge is intersected somewhere between its end-points, this requiring that the Edge be split into two separate Edges. The second type is where the Edge is intersected on one of its end-points. Here the Edge does not need to be split into two.

A first requirement then is to split Edges in the Adjacency Graphs. Each Line-segment has an Edge-list, and each Edge belongs to only one Edge-list. Splitting an Edge involves inserting a new Edge into the Edge-list, inheriting the End-point of the existing Edge and with a Start-point at the point of intersection. The End-point of the existing Edge is amended to the point of intersection, where the new Edge begins. Edge-lists are ordered, so the new Edge must be inserted immediately after the existing Edge (Fig. 3.22).

Splitting Edges.



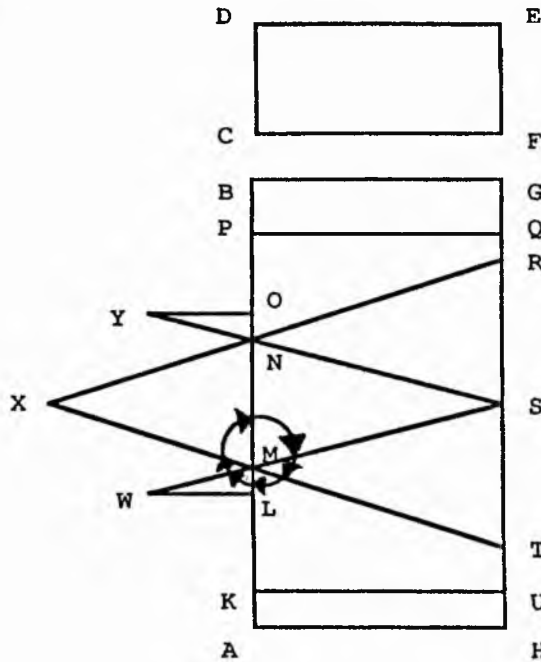
Line structure before and after update due to split of edges [L#XT,1] and [L#WS,1] at point M.

BEFORE	AFTER
. . .	. . .
. . .	. . .
[L#WL,[[1,WL]]]	[L#WL,[[1,WL]]]
[L#YS,[[1,YS]]]	[L#YS,[[1,YS]]]
[L#WS,[[1,WS]]]	[L#WS,[[1,WM],[2,MS]]]
[L#XR,[[1,XR]]]	[L#XR,[[1,XR]]]
[L#XT,[[1,XT]]]	[L#XT,[[1,XM],[2,MT]]]

Front entries now contain references to edges in form [line id, Edge id]. This maintains consistency through changes made to edges caused by splits. [L#WS,1] before the split refers to the edge from W to S. After the split [L#WS,1] refers to edge from W to M. This is consistent because the Front must contain the part of the edge beginning at W in this case.

Figure 3.22

Connecting Edges at a Junction.



State of line structure when vertex M has been reached.

[L#AB,[[1,AK],[2,KL],[3,LM],[4,MB]]]

. . . .

. . . .

[L#WS,[[1,WM],[2,MS]]]

. . . .

[L#XT,[[1,XM],[2,MT]]]

Connections between Edge-list items.

Format of edge-list item incorporating a connection for each side:-
[edge-id,edge vertices,[above-left connection],[below-right connection]]

Format of connections:-
[connected edge-id, connected to side]

[L#AB,[..,.. [3,LM,[L#WS,1,below],[,,]],[4,MB,[,,],[L#WS,2,above]]]]

. . . .

. . . .

[L#WS,[[1,WM,[L#XT,1,below],[,,]],[2,MS,[,,],[L#XT,2,above]]]]

. . . .

[L#XT,[[1,XM,[L#AB,4,left],[,,]],[2,MT,[,,],[L#AB,3,right]]]]

Above the three lines and six edges involved in the connection around vertex M. Connections not involved in vertex M have been ignored.

Following the cycle around a vertex involves following the connection from one side of an edge to the side OPPOSITE that indicated by the connection.

Following the cycle around M from [L#AB,3,below]:-

[L#AB,3,below] -> [L#WS,1,Below] -> [L#XT,1,below] -> [L#AB,4,left] ->
[L#WS,2,above] -> [L#XT,2,above] -> [L#AB,3,below]-> . . .

Figure 3.23

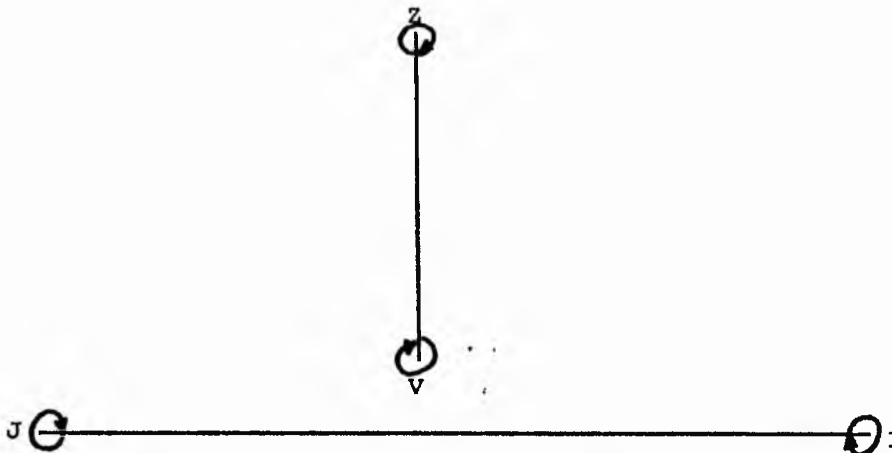
Having split any Edges which require it, the second requirement is to connect together all the Edges meeting at the vertex (Fig. 3.23). These Edges may include those which have been split by intersection, and those which have not been split but which start or end at the vertex. Edges which meet at a vertex are connected in a cyclical chain in a clockwise direction around the vertex. This connection between Edges in the chain is supported by the Region-list elements of the Edge structure. Each Edge can reference its clockwise neighbours through the Region-list elements. The Left-above Region-list elements reference the Edge's clockwise neighbour at its Right-most vertex, or Top-most vertex for a Vertical Edge. The Right-Below Region-list elements reference the Edge's clockwise neighbour at its Left-most or Bottom-most vertex.

The clockwise connection of Edges is consistent across all vertices in the drawing, including those vertices where only one Edge occurs (Fig. 3.24). A vertex where a single Edge has its Start-point, which might include the bottom-most vertex of a vertical edge, only has a right-hand side. Here the Right-below Region-list of that Edge references its Left-above side. A vertex where a single Edge has its End-point, which might include the Top-most vertex of a Vertical Edge, only has a Left-hand side. Here the Above-left Region-list of that Edge references its Right-below side.

The fact that Edges are split at intersection points means that the Permute procedure is no longer necessary. This procedure in the previous algorithms reversed the order of line-segment entries in the Front to reflect the change in vertical ordering of those line-segments across the intersection point. Now however, the Front contains entries for Edges rather than for line-segments and by definition these Edges have their end-points at the intersection points of the line-segments to which they belong. The re-ordering which now needs to be done across an intersection point is the removal of the Edges which end at the intersection point, and the insertion of the those which start there. These functions are performed by the Remove and Insert procedures respectively.

The following sections describe the Plane-sweep algorithm in terms of the three procedures - 'Advance-Front', 'Remove' and 'Insert'. Pseudo-code outlines of these procedures are given in Appendix C.

**Connecting Edges at a vertex (cont)
- Trivial cases.**



Line structure including edge connections describing the trivial cases where a line does not connect to any other line.

[L#VZ,[[1,VZ,[L#VZ,1,right],[L#VZ,1,left]]]]

[L#JI,[[1,JI,[L#JI,1,below],[L#JI,1,above]]]]

The right side of the vertical is connected in a cycle to its left side.

The above side of the horizontal is connected in a cycle to its below side..

Figure 3.24

3.8.3 Advance-Front.

This procedure, as before, is the executive which controls the progression of the Front across the plane. It reads successive entries from the Transition-list, each entry containing one or more Junction entries. Each Junction entry describes either a single point vertex or a vertical edge. For each Junction entry, the executive calls procedures which perform the amendments to the Front order and the associated Region-list maintenance operations on first the left-hand side of the Junction and then the Right-hand side.

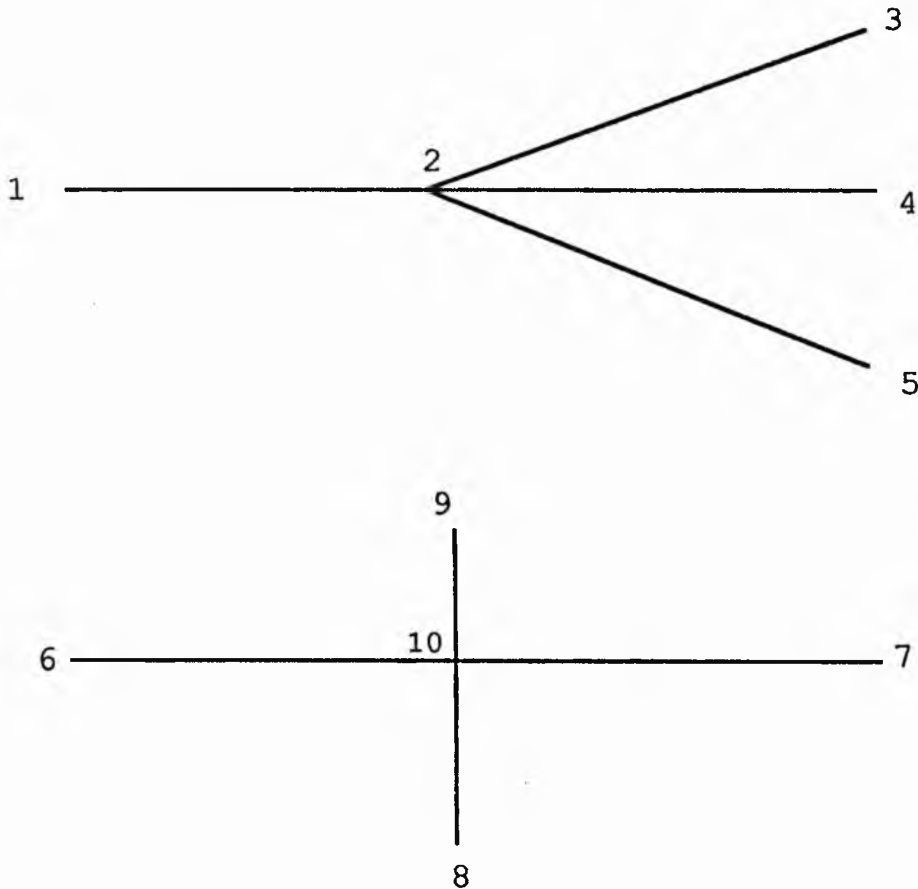
The Insert procedure is called once for each Edge with its Start-point on the Junction. When the Junction describes a Vertical Edge, the Insert procedure may be called for successive Junction entries whose Insertion-lists contain Edges starting in the Vertical Edge. The possibility of successive Junction entries falling within the range of a Vertical Edge is coped for by recording the Top-most-vertex of a Junction Entry before the Remove procedure is called. Any successive Junction entries whose Bottom-most vertex falls on or below the current recorded Top-most-vertex must lie within a Vertical Edge. In the case of Vertical Edges it is important that Remove is only called for the Vertical Edge and not for successive Junctions lying within the Vertical Edge.

3.8.4 The Remove Procedure.

The Remove procedure climbs the Front removing all Edges it finds which pass through the Junction. By definition all Edges which hit a Junction end there, and so the test to see whether Edges end at the Junction or not is no longer needed. However some extensions are needed to the intersection detection routines to handle the new list formats, and some of this extra processing is included in the Remove procedure.

In previous versions, intersecting Edges were handled by the Permute procedure which reversed the positions of intersecting Edges in the Front. Now intersection-points are where one set of Edges are removed from the Front to be replaced by a set of new Edges. The intersection detection routine must now not only update the Transition list by adding a new Junction entry for the Intersection-point but must also build an Insertion-list for that

Undiscovered Intersections.



Two types of intersection which are not discovered before they are reached by the Plane-sweep. They are not discovered because only one of the lines is in the Front before the intersection point is reached. These 'undiscovered' intersections always involve either a vertical line or a number of lines starting at the intersection. The REMOVE procedure can determine when a line it encounters in the Front has to be split because of an 'undiscovered' intersection - such a line is always the only line at that particular vertex.

Figure 3.25

Junction entry containing the set of Edges to be inserted at that point.

This has added one small but significant new feature to the Remove procedure, the need to split solitary edges which pass through the Junction and update the Transition-list. Most intersections of two or more non-vertical edges are discovered in advance as the Plane-sweep progresses, however the intersection between two or more non-vertical edges, where all but one of the Edges start at the intersection-point, and the intersection between a non-vertical edge and a vertical edge are not discovered. This is because such intersecting edges have not been adjacent to each other in the Front before the intersection-point is reached (Fig.3.25).

The implication is that any single Edge passing through the Junction without intersecting with any other Edge there must be split at that point into two Edges, regardless of whether the Junction describes a vertical edge or a single-point. One of these Edges is the one in the Front which ends there, and the other one must be added to Insertion-list of the Junction to describe the part of the Edge which begins there. A precaution must be taken to prevent an Edge of a line-segment which ends there from being split and so extended on the right-hand side of the Junction.

This requirement is satisfied by counting the number of Edges which are removed from each discrete single-point within the Junction. Whenever a new discrete point is reached, including the termination point where the first point above the Junction is read, the number of Edges removed from the previous lower point is checked and if only one Edge has been removed from there, and if the line-segment of that Edge did not end there, then that Edge is split. (The variable 'Last-Edge' always references the edge examined before the current one, even though it has been removed from the Front.)

Most of the procedure is concerned with connecting Edges to other Edges. Mainly this is concerned with connecting the side below one Edge to the side above its lower adjacent Edge. Sometimes, when the Junction describes a Vertical Edge, part of the Vertical Edge lies inbetween the Edge and its lower neighbour. In this case the Vertical Edge must be split into two, one part which lies between the Edges and the other which lies above them. The left side of the part between the Edges is used to connect them together.

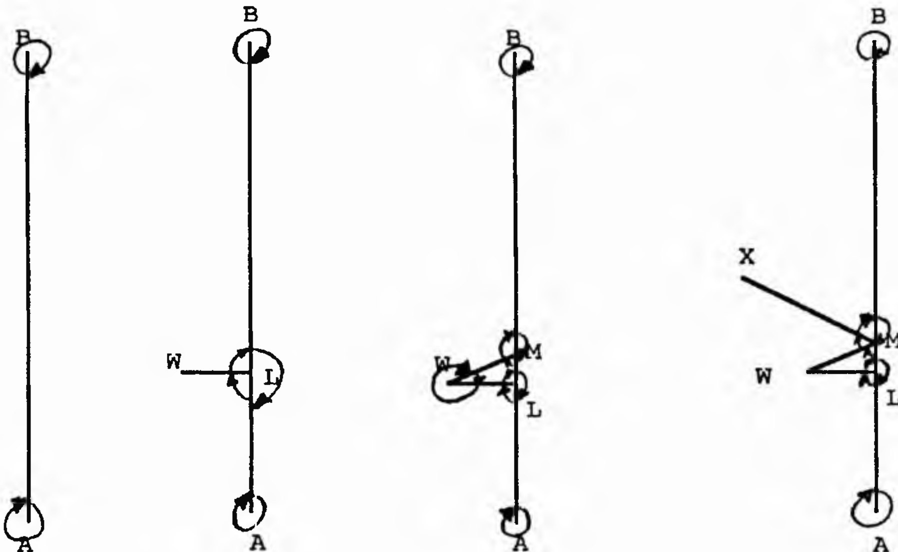
Connecting Edges around the 'butt-end' of a Junction is now performed by the Remove procedure, and this is done irrespective of the type of Junction, Vertical or single-point and also irrespective of whether the Junction has a right-hand side or not.

In the case of a Vertical Edge, maintaining the 'butt-end' requires that the two sides of the Edge are connected together into a cycle. This initial state assumes that the Junction has no Left-hand or Right-hand side. As the Remove procedure progresses, it might very well discover that the Vertical Edge has a Left-hand side, and so modifications are made to the Edge list around the Vertical Edge to incorporate these other Edges which have been discovered. The First Edge found for example might lie on the Bottom-most vertex. In this case the right side of the Vertical Edge must be connected to the below side of the First Edge, and the above side of the First Edge must be connected to the Left side of the Vertical Edge.

As mentioned previously, sometimes the Vertical Edge must be split. Maintaining the Edge list around the Vertical Edge when this happens is not as straight-forward as it might appear (Fig. 3.26). Always the Top-most-edge on the Left-hand side, which was initially the Vertical Edge but may eventually be another non-vertical Edge which ends on the Top-most vertex of the Vertical Edge, must be connected to the Top-most edge on the Right-hand side of the Junction. When a Vertical Edge is split, always below the Top-most vertex, the new Edge resulting from the split is always the new Top-most Edge. Therefore the Left side of the new Edge is connected to its Right side. On the Right side of the Junction, the new Edge must always be connected to the old lower Edge to maintain the cycle around the Vertical Edge.

The 'butt-end' processing for a single-point Junction is very much simpler than that for Vertical Edges. The above side of the Top-most Edge on the Left-hand side is connected to the below side of the Bottom-most Edge. In its implementation, all that is required is to connect the side above the First Edge found to its below side. Any successive Edges which pass through the Junction inherit the connection from Top-most to Bottom-most by a simple assignment.

REMOVE splits vertical lines.

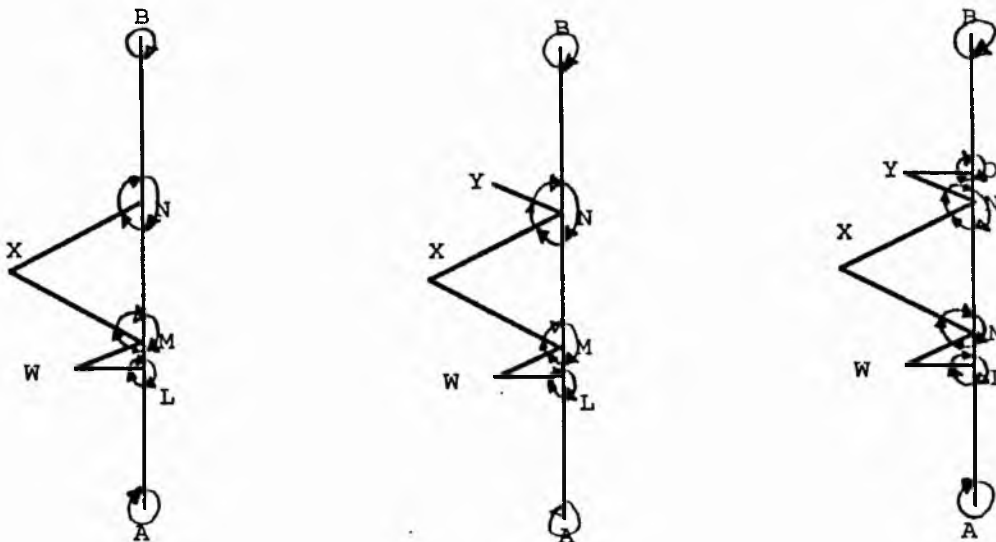


Line AB contains only one edge - AB.

Split at L. Connect edges around vertex L. Connect around top of new edge LB.

Split at M. Connect edges around M. Connect around top of new edge MB.

No split. Connect Edges around M.



Split at N. Connect edges at N. Connect around top of new edge NB.

No split. Connect edge YN into group around N.

Split at O. Connect edges at O. Connect around top of new edge OB.

Figure 3.26

The final part of the procedure contains the usual Intersection-Test performed between the Edge immediately below those Removed and the Edge immediately above those Removed. After this the Top-most Edge encountered during the procedure is identified. This Edge is passed back to the calling executive procedure and for subsequent use by the Insert procedure so that the Edge list around the 'butt-end' of the Junction may be picked up and diverted around any Insertions. The Top-most edge is identified because of the clockwise direction of the Edge list :- by having the Top-most edge on the Left-hand side any of the Edges on the Right-hand side can be found.

Identifying the Top-most Edge is quite complicated. If no Edges at all fall within the Junction, and that means no Vertical Edge either, then the Junction has no Left-hand side and so a 'Nil' reference is returned. If the Junction has only a Vertical Edge, or if the Junction has a Vertical Edge and all other Edges falling within it fall below its Top-most vertex, then the Vertical Edge is the Top-most Edge. (Note that when the Vertical Edge is split, the Top-most of the two resulting smaller Edges is always carried forward by the variable 'Vertical-Edge'.) If some non-vertical Edges fall on the Top-most Vertex, regardless of whether the Junction is a Vertical Edge or single point, then the last such Edge encountered is the Top-most Edge.

3.8.5 The Insert Procedure.

The Insert procedure is called once for each Edge which has its Start-point lying within the range of the current Junction. Insert climbs the Front looking for the position where the Edge should be inserted. Having placed the Edge in the Front, Insert performs some Region-list maintenance functions.

The overall effect of these Region-list maintenance operations is similar to those of the previous versions of Insert. Region-lists are formed on the right-hand side of the Junction between adjacent pairs of Edges, some of which might not share the same Start-point but might be at separate positions up the Vertical Edge. The Top-most and Bottom-most Region-lists on the Left-hand side of the Junction are inherited respectively by the Top-most and Bottom-most Edges on the Right-hand side of the Junction. Region-lists need to be extended around the Left-hand side of any 'butt-end's which are constructed.

Region-list maintenance is determined by the Edges that Insert finds in the Front lying within the range of the Junction. The Vertical-Edge, if it exists, is not an entry in the Front but is available to Insert and does indeed count as a neighbour of any Edge inserted into the Front at that Junction. Region-list maintenance largely consists of connecting new Edges to its neighbours already in the Front - or to the Vertical-Edge. When no neighbours are found, it searches for traces remaining of the Left-hand side and attempts to connect these to the new Edge (Fig. 3.27).

Where a Junction has a Left-hand side, all the Edges on the Left-hand side of the Junction have been removed from the Front. The Left-hand side processing, namely the Remove procedure, assumes that no Right-hand side exists for the Junction and so knits the Region-lists around the right-hand side of the Junction as if it were a 'butt-end'. The first Edge to be inserted in a Junction therefore has to determine whether or not any Left-hand side exists and if so unravel the Region-list around the butt-end and divert them around itself.

If the Junction does not contain a Vertical-Edge, then the Left-hand side exists only if the Top-most-Edge has been passed forward from the Remove procedure. If the Top-most-Edge exists, then the Region-list around the 'butt-end' of the Junction must be broken and diverted around the new insertion. The below side of the new insertion inherits the connection to the below side of the Bottom-most Edge from above the Top-most-Edge, and the above side of the Top-most-Edge is connected to the above side of the new insertion. If the Top-most-Edge does not exist, then there is no left-hand side in the Junction and therefore the right-hand side of the Junction forms a 'butt-end'. The Region-list around this 'butt-end' is initially the connection from below the new insertion to above it. This may be diverted later as further Edges are inserted above or below the first Edge.

If the Junction contains a Vertical-Edge, then the Left-hand side does exist, even if it consists of no more than that Vertical Edge (Fig. 3.28). In this case, the Vertical-Edge must be examined to find out whether it had been split during the Left-hand side processing, and if so to find that part of the Vertical-Edge whose Start-point is the same as the Edge being inserted. If no part of the Vertical-Edge has the same Start-point as the

INSERT - new insertion has no touching neighbour in the Front.

(Top-most Edge = nil)

X

[L#XT,[[1,XT,[],[]]]]

Before XT inserted.

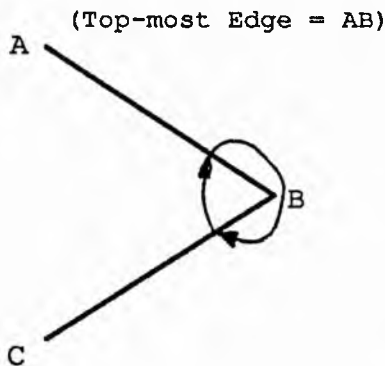
(Top-most Edge = nil)

[L#XT,[[1,XT,[],[L#XT,1,above]]]]

After XT inserted in Front.

INSERT knows New insertion forms a right-handed 'butt-end' because there is no Top-most Edge from the Left-side available. Below side is connected around start-point to Above side.

(Top-most Edge = AB)

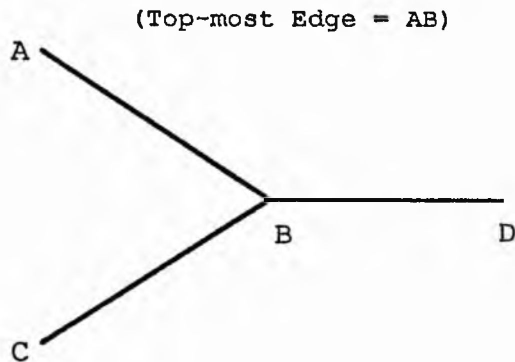


REMOVE connects above side of AB to below side of CB.

AB and CB are removed from the Front.

The Top-most edge on the left-side of the Junction -AB- is stored in case subsequent insertions start at vertex B.

(Top-most Edge = AB)



INSERT finds nothing in Front at vertex B. (AB and CB have been REMOVED).

Top-most edge on left-side is checked and found to be AB.

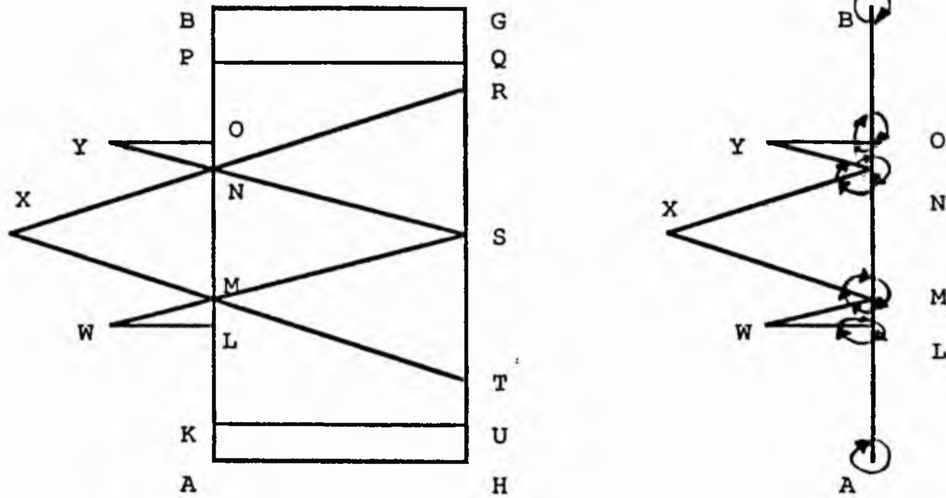
Connection from above AB is inherited by below side of BD. Connection from above AB is made to above BD.

[L#AB,[[1,AB,[L#CB,1,below],[]]]]
[L#CB,[[1,CB,[L#AB,1,below],[]]]]
[L#BD,[[1,BD,[],[]]]]]

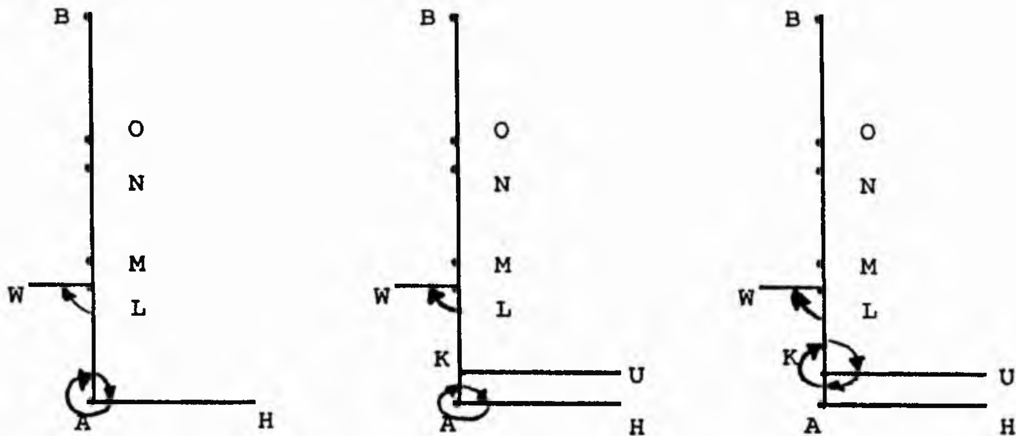
[L#AB,[[1,AB,[L#BD,1,above],[]]]]
[L#CB,[[1,CB,[L#AB,1,below],[]]]]
[L#BD,[[1,BD,[],[L#CB,1,below]]]]

Figure 3.27

INSERT - new lines intersect with Vertical.



State of the Edge connections around AB after REMOVE called for line AB of scene on left.



New insertion AH starts at a vertex of vertical edge AL.

No split is needed.

Below of AH connected to left of AL - ie AH inherits right from AL.

Right of AL diverted to above AH.

New insertion KU causes edge AL to be split at K into edges AK and KL.

New Edge AK inherits identifier of AL within line AB, and so below AH is automatically connected to left of AK. AK inherits right connection to AH from AL.

New edge KL inherits left side from AL maintaining connection to WL.

Cycle around vertex K is built: Right of KL is connected above KU; Below KU is connected right AK; Left AK is connected left KL.

Right of KL's higher neighbour ML is connected to right of KL.

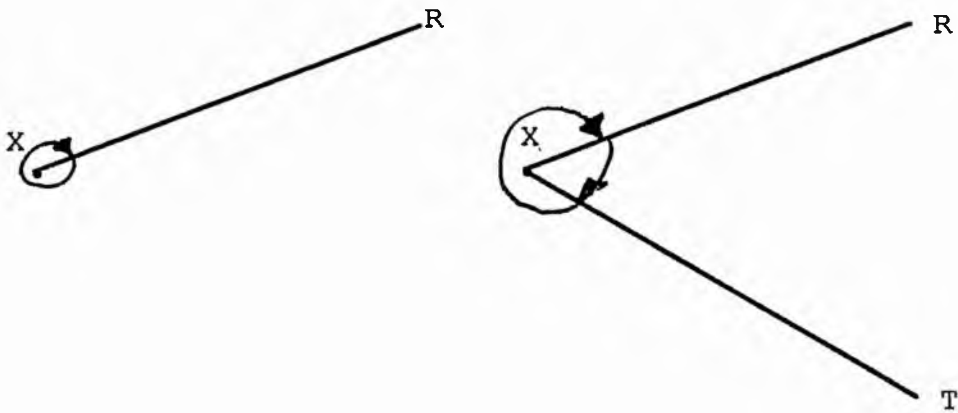
Figure 3.28

new insertion then the nearest lower part of the Vertical-Edge must be split. The Left-hand side parts of the split must be connected together to maintain the Left-side Region-lists, which in the case of the Top-most Edge of the Vertical may involve connecting the Left-hand side of the Edge to the Right-hand side, if no non-vertical Edges on the Left-hand side end on the Top-most vertex of the Junction. On the Right-hand side, the parts of the Vertical-Edge which lie above and below the new insertion are respectively diverted around the top and bottom of the new insertion.

This procedure for connecting an Edge to the Vertical-Edge is the same for all Edges which do not share the same Start-point as their neighbours. Always the Vertical-Edge is split, if it has not already been so by the Left-hand side processing, and the resulting parts are connected to the new insertion.

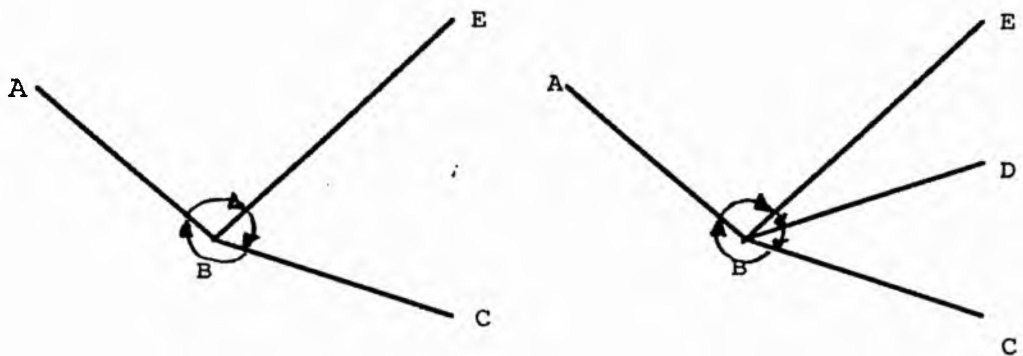
Inserted Edges which do share the same Start-point as their neighbours which are already in the Front fall into two categories from the point of view of Region-list maintenance: those which have a higher neighbour including those which have both a higher and a lower neighbour (Fig. 3.29); and those with only a lower neighbour (Fig. 3.30). Generally the procedure is to divert an existing Region-list around one side of the new Edge, and to initialise a new Region list between the new Edge and the appropriate neighbour. If a higher neighbour exists, the Region-list connected below that Edge is diverted below the new Edge and a new list is initialised by connecting the below side of the higher neighbour to the above side of the new Edge. If only a lower neighbour exists, the Region-list above that Edge is diverted above the new Edge and a new region-list is initialised between them.

INSERT - new edge is NOT highest at vertex.



Before inserting XT. After inserting XT.

New insertion XT inherits below side of its higher neighbour (XR). Below side of higher neighbour (XR) is connected to new insertion XT.

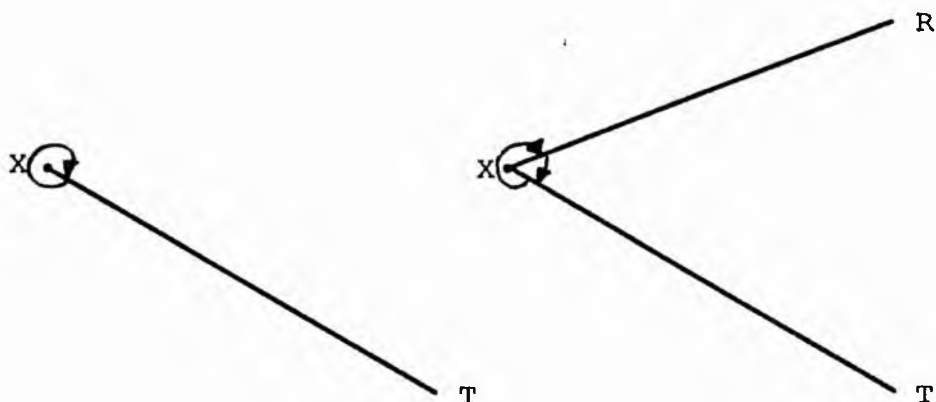


Before inserting BE. After inserting BE.

Below side of new insertion (BD) inherits below side of its higher neighbour (BE). Below side of higher neighbour (BE) is connected above new insertion (BD).

Figure 3.29

INSERT - new edge is highest at vertex.

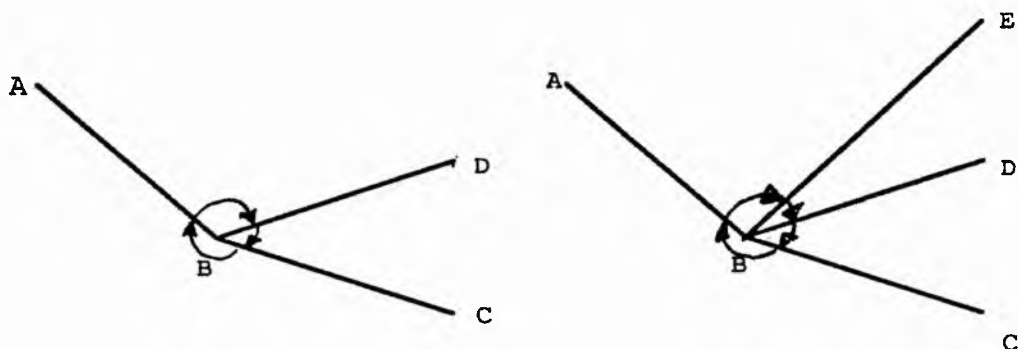


Before inserting XR.   After inserting XR.

XR is found to be the top-most edge emerging from vertex X. When inserting XR, the cyclic list around vertex X is followed from below the previous top-most edge (XT) until an edge is found which connects to the above side of the previous top-most edge(XT).

In the trivial case above, the below edge of XT is found to connect to the above edge of the previous top-most edge, which also happens to be XT.

The connection to the previous top-most edge (XT) is diverted above the new top-most edge (XR). Below XR is connected above XT.



Less trivial case.

The cyclic list around B in the left hand diagram is followed from below BD to above AB, which connects to the top-most edge (BD) emerging from B.

The connection from above AB to above BD is diverted above new top-most edge BE in the right hand diagram. Below BE is connected above BD.

Figure 3.30

3.9 Constructing the Containment Hierarchy.

3.9.1. Introduction.

The extended Plane-sweep algorithm provides a set of Adjacency Graphs for each set of interconnecting line-segments in the drawing. A Containment Hierarchy now needs to be constructed to relate these separate Adjacency Graphs to each other. The relationships which should be identified are of the nature of 'Adjacency Graph #A is contained by Adjacency Graph #B' (Parent), 'Adjacency Graph #B contains Adjacency Graph #A' (Child), and 'Adjacency Graph #C is contained by the same Adjacency Graph as contains Adjacency Graph #A' (Sibling).

The first stage in achieving this goal is to introduce Region Labels into the scheme (Fig. 3.31). Each Adjacency Graph constructed by the extended Plane-sweep algorithm contains one or more Regions, these explicitly described by the separate Edge-lists which are constructed within each Adjacency Graph. These Region-lists are in fact a basic component of the Adjacency Graph. Each separate Adjacency Graph consists of a conglomerate of none, one or more partitions which are connected together by sharing Edges. Each partition is described by a separate Region-list. The case where the Adjacency Graph contains no partitions is when the Adjacency Graph describes a set of connected line-segments which do not form any closed loops and so form an open rather than closed polygonal shape. One other Region-list describes the hull of the shape of the conglomerate. Therefore we have Region-lists which describe partitions, and Region-lists which describe the hulls of conglomerates of partitions. We will call the former type Interior Region-lists, and the latter type Exterior Region-lists. Every Adjacency Graph can be said to comprise one Containment Hierarchy wherein the Exterior Region-list contains all those Interior Region-lists which describe the partitions from which the conglomerate is comprised.

Each Region-list constructed by the Plane-sweep must be given a unique label to identify it, and this label must also identify the type of the Region-list, whether Interior or Exterior. These Labels are to be constructed into a hierarchy supporting the relationships mentioned previously, namely Parent, Child and Sibling, and so each Label must contain references to other labels which are its Parent, Children and Siblings (Fig. 3.32). This

Modified Edge structures link to Region labels.

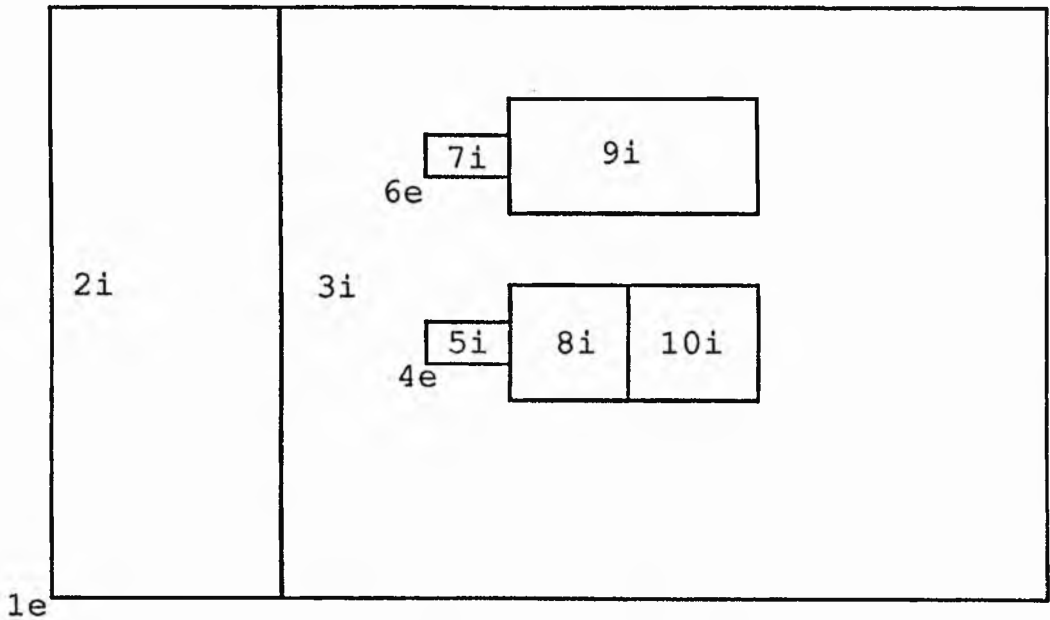
Line-list item format.
[line-id,[edge-list]].

Edge-list item format:
[edge-id,vertices,[above connection],[below connection]].

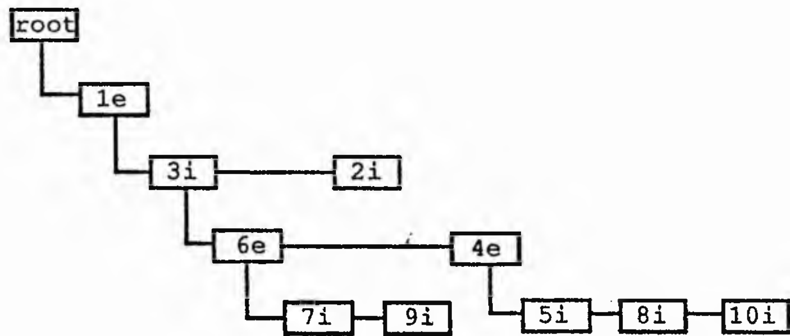
Modified Connection structure:
[line-id,edge-id,side,Region-id]

The Edge data structure has been extended to include the name of the Region list that each side - Above/left and Below/right - participates.

Figure 3.31



Seven partitions - 2i,3i,5i,7i,8i,9i,10i.
Three exterior Regions - 1e,4e,6e.



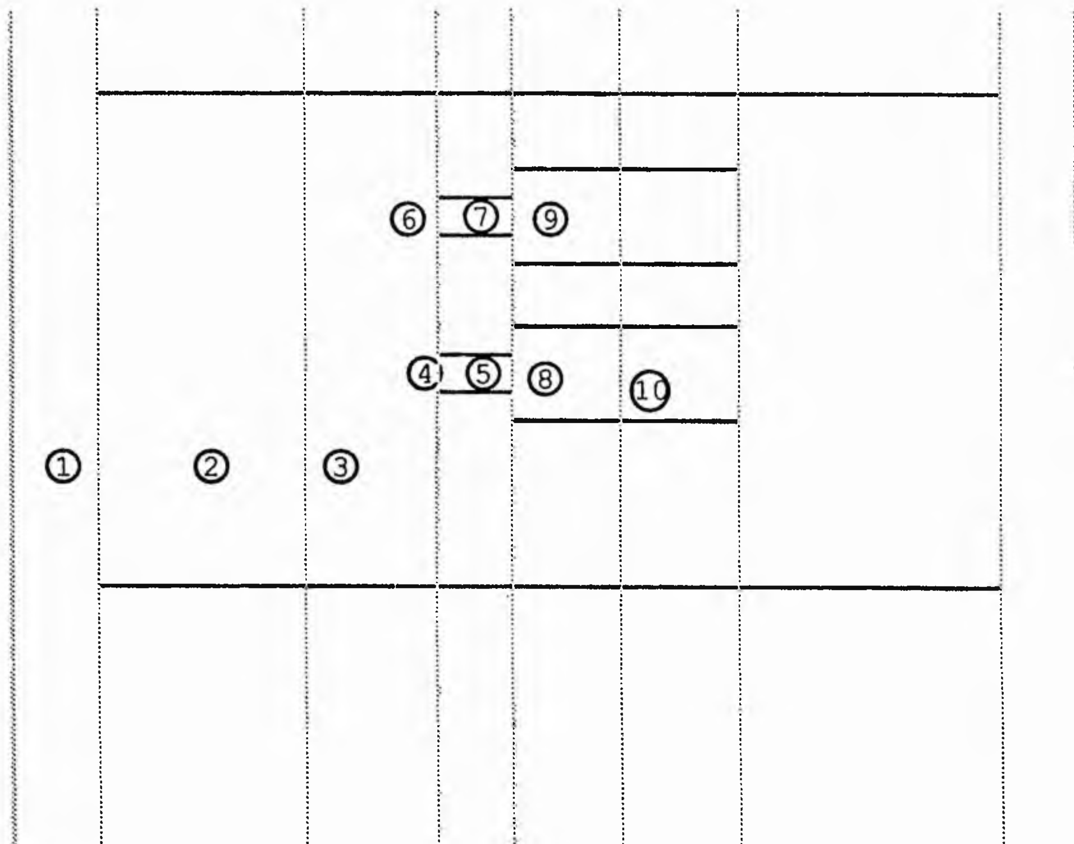
Region hierarchy.

Vertical connections express the 'Containment' relationship.

Horizontal connections express the 'Sibling' relationship.

Figure 3.32

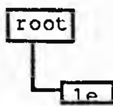
Building the Containment hierarchy.

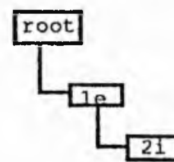


Transitions: numbers in circles show which vertex and which side - left or right - of the Transition where Region list maintenance operations occur. The stages of construction of the Containment hierarchy are shown overleaf with references to numbered vertices.

Figure 3.33(i)

Building the Containment hierarchy (cont).



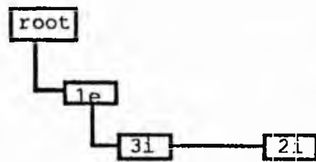


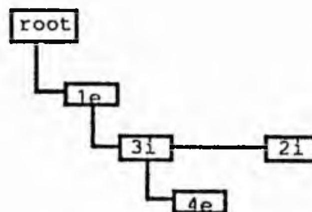
(1)

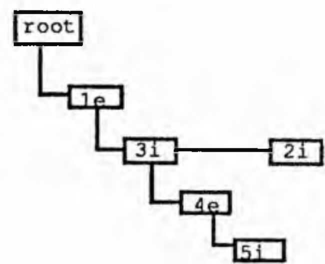
(2)

(start state).

Butt-end so new Region is an exterior





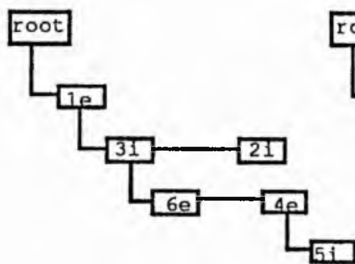


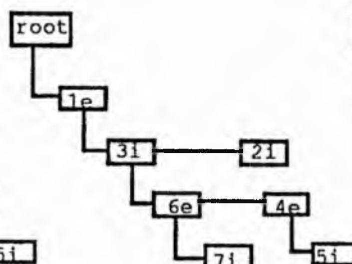
(3)

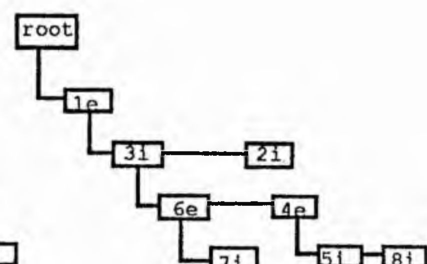
(4)

(5)

Butt-end so new Region is an exterior





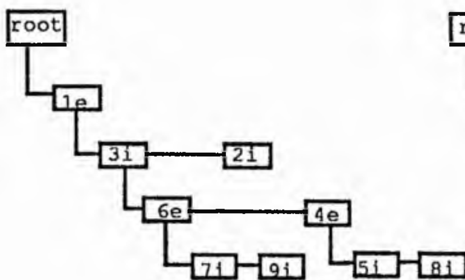


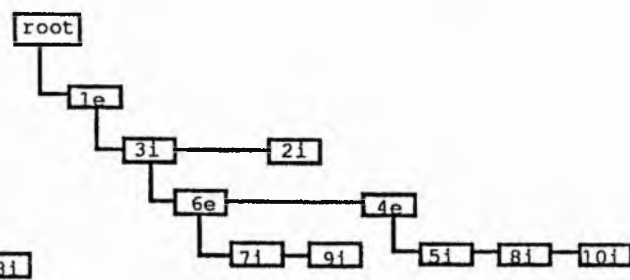
(6)

(7)

(8)

Butt-end so new Region is an exterior





(9)

(10)

Figure 3.33(ii)

data structure is shown in Section 3.2. An additional requirement is to be able to reference Region-lists from the Region-Labels, for which connections to the head and tail Edges of the Region-lists have been included.

The approach developed here is to construct the Containment Hierarchy on the fly. Every time a new Region-list appears to be starting, a new Region-label is created and tentatively positioned in the Containment Hierarchy (Figs. 3.33(i) and 3.33(ii)). The position the new Region label adopts is determined by the proximity of other Region-lists in the Front, this described in more detail in sections 3.9.2 to 3.9.4.

This on-the-fly approach makes assumptions about Region-labels' types and positions in the Hierarchy based on evidence provided by Local features. Any new Region-list which is initialised around the Left-hand side of a 'butt-end' is assumed to be an Exterior Region. Similarly, any Region-list initialised between two adjacent touching edges on the Right-hand side of a Junction is assumed to be an Interior Region. These assumptions may subsequently be proved to be wrong and give cause to some restructuring of the Containment Hierarchy as contrary evidence is discovered. The causes and remedies of these wrong assumptions is discussed in detail later.

Despite the necessity of having to make assumptions during the construction of the hierarchy, caused by constructing the relationships between Region-lists before the complete shape of the Region is known, this on-the-fly approach has some advantage. If the algorithm waited until the complete shape of the Regions were known before allocating Region-labels to them, the Containment Hierarchy would have to be constructed bottom-up. The Children at a given level would always be described fully before their Parents. This would cause problems in that the overall Hierarchy would have to be constructed by merging together smaller Hierarchies as the common Parent of Sibling hierarchies were discovered. Temporary holding areas would be needed to hold partial results, and some method of identifying the relationships between separate hierarchies. In constructing the assumed relationships between Regions before the Region-lists are complete, the Hierarchy is constructed top-down so obviating some of the problems of bottom-up construction.

3.9.2 Determining the position of Region-Labels in the Containment Hierarchy.

The Containment Hierarchy consists of at least two levels, though usually more, of Region-labels. The top most level of the hierarchy contains a single root Region which has an Interior type. The root Region can be considered to be the inside of the frame of the drawing. All other Regions in the drawing are contained within this frame.

The rest of the hierarchy depends on the drawing, but two general observations can be made. Each level in the Hierarchy consists of labels of the same type, Interior or Exterior, and alternate levels in the hierarchy have different types. Interior contains Exterior contains Interior contains Exterior. The levels above and below an Interior Region level will both contain Exterior Regions. The levels above and below an Exterior Region level will both contain Interior Regions.

Placing a new label in the Hierarchy is guided by the situation local to the Edges which constitute the boundaries of the new Region-list. How the decision is made depends on the type of the new Region-list, and on the type of the Regions belonging to Edges nearby in the Front.

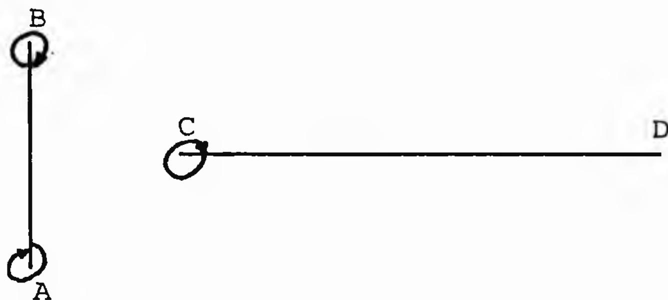
3.9.3 Determining the position of an Exterior Region Label in the Containment Hierarchy.

Exterior Regions are always initialised around the Left-hand side of Junctions which are 'butt-ends'. The 'butt-end' is initialised in either the Remove procedure when the Junction contains a Vertical-Edge, or in the Insert procedure when the Junction is a single-point (Fig. 3.34(i)).

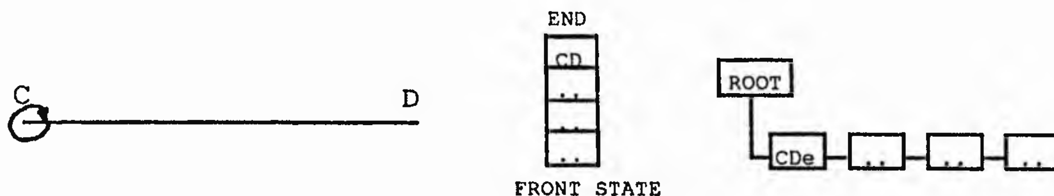
The main feature that the two types of Exterior Region initialisation share in common is that the new Region list consists of only one Edge (at that time), and that the Edge does not touch any other Edge in the Front (at that time).

The position that the Region-label adopts in the Containment Hierarchy is dependent on the position of Regions of Edges in the Front nearest the Junction. The two Regions which

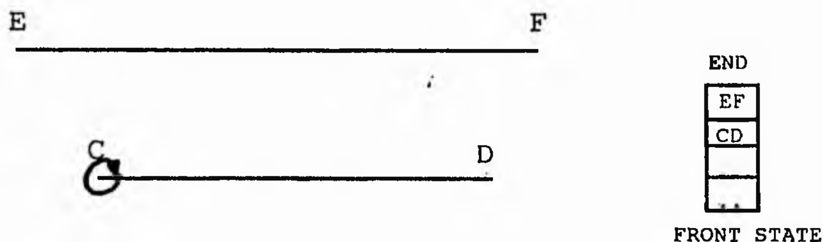
Positioning a new Exterior Region in the Hierarchy.



The two types of butt-ends (Vertical AB and vertex C) around which new exterior Regions are formed.



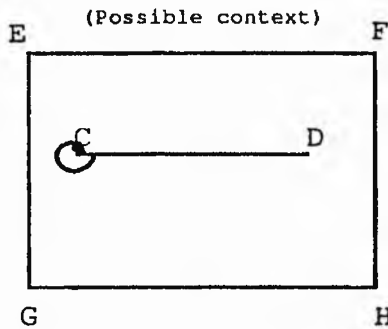
If the front is empty above the new insertion CD, the new exterior Region around CD is placed as a child at the start of the list of children of the ROOT of the hierarchy.

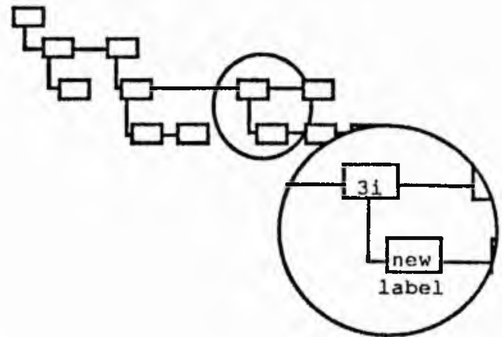


The front contains an entry (EF) above new insertion CD. The new exterior Region around CD will be placed in the hierarchy in a position relative to the Region list below EF.

Figure 3.34(i)

Positioning a new Exterior Region in the Hierarchy (cont).



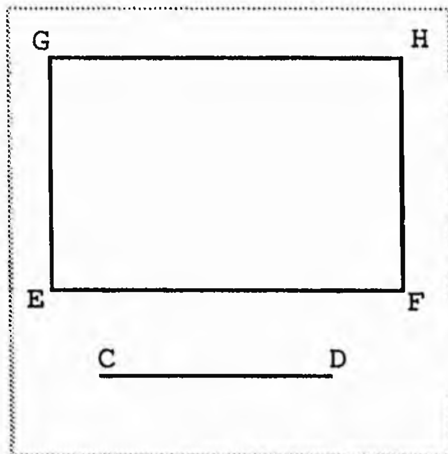


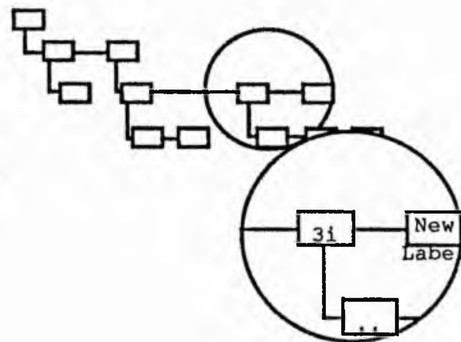
Line entry for EF.

[L#EF,[[1,EF,[],[L#GE,1,right,3i]]]]

Higher Neighbouring Region is an Interior.

Below entry of edge EF shows that the Region below EF is interior Region 3i.
New exterior around CD is placed as a 'child' of 3i





Line entry for EF.

[L#EF,[[1,EF,[],[L#GE,1, left,3e]]]]

Higher Neighbouring Region is an Exterior.

Below entry of edge EF shows that the Region below EF is exterior Region 3e.
New exterior Region around CD is placed as a 'sibling' of 3e.

Figure 3.34(ii)

could be consulted with equal validity are that below the higher neighbouring Edge to the Junction, and that above the lower neighbouring Edge. Arbitrarily, the former shall be chosen. If the neighbouring Region is an Interior Region, then it is considered to be the Parent of the new Exterior Region. The New Label is positioned in the level of the Hierarchy immediately below the neighbouring Region (Fig. 3.34(ii) Top).

If the neighbouring Region is an Exterior Region, then it is considered to be a Sibling of the new Exterior Region. The New Label is positioned in the Hierarchy next to the neighbouring Region (Fig. 3.34(ii) Bottom).

3.9.4 Determining the position of an Interior Region Label in the Containment Hierarchy.

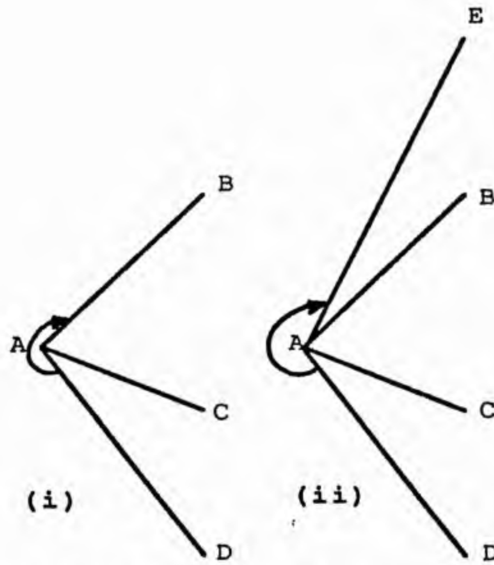
Interior Regions are always generated between two Edges which are connected together either directly, in which case they both share the same Start-point, or indirectly, in which case they are separated from each other by a part of a Vertical-Edge (Fig. 3.35(i)).

The position of the New Interior Label is determined by examining one of the Regions on the other side of one of the two non-vertical Edges which bound the new Regions. Arbitrarily, the Region above the higher of the two Edges shall be chosen as a guide.

If the Region is an Exterior Region, then that Exterior Region is the hull of a conglomerate of Partitions, a member of which will be described by the new Region-list. The new Region-label is therefore nominated as a child of the Exterior Region (Fig 3.35(ii) Top).

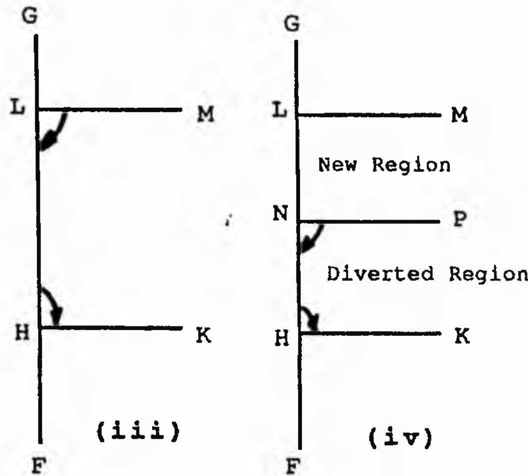
If the Region is an Interior Region, then that Interior Region is a member of the same Adjacency Graph as the new Interior Region, and so the new Region is a Sibling of its neighbour and is placed accordingly beneath the Parent of that neighbour (Fig. 3.35(ii) Bottom).

Positioning a new Interior Region in the hierarchy.



Insert when new Edge highest.

Exterior Region from AD to AB (i) is diverted around new Edge AE (ii) by Region-list maintenance in Insert.
New Interior Region created between AE and AB.



Insert when new Edge not highest.

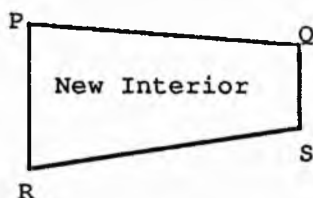
Region between LM and HK (iii) is diverted below new Edge NP (iv).
New Interior Region created between LM and NP.

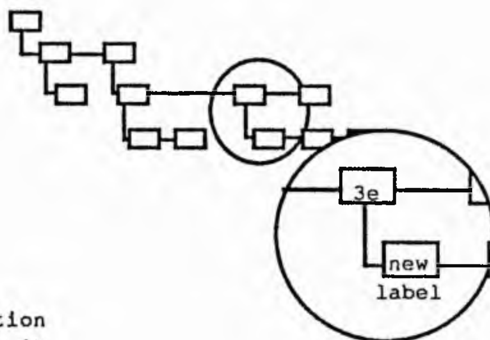
Figure 3.35(i)

Positioning a new Interior Region in the hierarchy.

The Region above the higher of the two non-vertical Edges bounding the new interior is examined to determine the position of the new Interior label.

Region above higher neighbour is an Exterior.



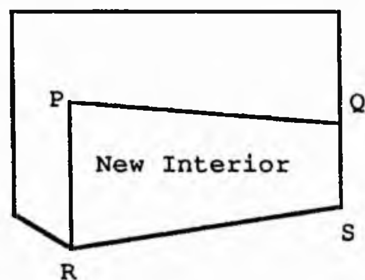


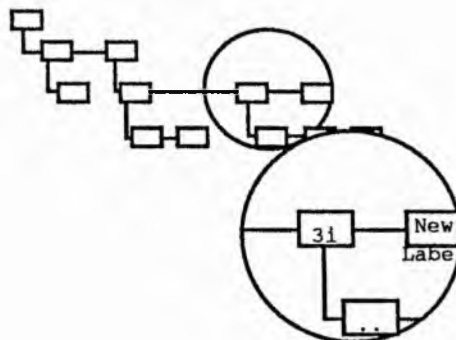
[L#PQ,[1,PQ,[[,,,3e],[]]]]

(note. Above entry of PQ has no connection to any other edge because at this time it is the TAIL of Region 3e.)

The Above entry of Edge AD shows that the Region above it is exterior Region 3e.
New interior Region is a 'child' of 3e.

Region Above higher neighbour is an Interior.





[L#PQ,[1,PQ,[[,,,3i],[]]]]

(note. Above entry of PQ has no connection to any other edge because at this time it is the TAIL of Region 3i.)

The Above entry of Edge PQ shows that the Region above it is interior Region 3i.
New interior Region is a 'sibling' of 3i.

Figure 3.35(ii)

3.9.5 Labelling conflicts.

Labelling conflicts are only encountered when two separate Region-list are to be concatenated - by the Remove procedure in the case of Region-lists meeting at the Left-side of a Junction, or by Advance-Front when Region-Lists meet on the Right-hand side of a Junction. A labelling conflict occurs when the Concatenate operation is given two ends of Region-lists, one a Head and the other a Tail, to join together and the two ends belong to different Region-lists. An invocation of Concatenate which does not encounter a labelling conflict is called a Closure, in which the Head and the Tail of the same list are joined together to form a closed loop.

Two types of labelling conflict can occur. In the first type the two meeting Regions have different labels, but the labels are of the same type, Interior or Exterior. The second type of conflict is when the two labels are of different types, one Interior and the other Exterior.

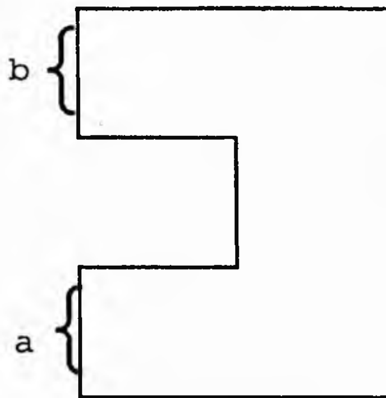
3.9.6 Conflict - Labels have the same type.

This type of conflict only occurs in Regions which contain more than one Right-handed 'butt-end's. A Junction forms a Right-handed 'butt-end' if it only has a right-hand side.

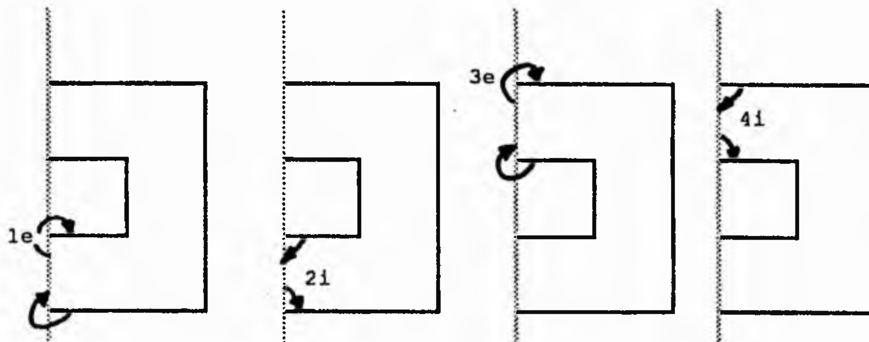
The logical progression leading up to the labelling conflict is as follows. Each right-handed 'butt-end' of the shape is encountered separately. On the Left-hand side of the 'butt-end' a new Exterior Region is formed, and on the Right-hand side a new Interior Region is formed. Eventually a Junction is encountered which marks the turning point of a cavity in the shape. Here, on the left-hand side of the Junction, the two separate Exterior Regions are submitted for Concatenation and the first labelling conflict is discovered. On the Right-hand side of the Junction, the two separate Interior Regions are submitted for concatenation and the second conflict is discovered (Fig. 3.36).

The problem to be solved at each conflict is that one of the Region Labels is superfluous and must be removed from the Containment Hierarchy. Decisions must be made as to which of the two labels is the redundant one and as to what manipulations need to be performed on the Hierarchy to safely remove a label.

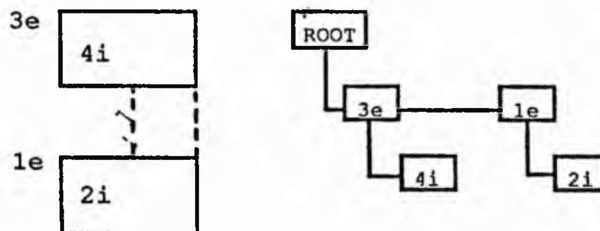
Labelling Conflict: Labels have same type.



Shape with two right-hand butt-ends a and b.



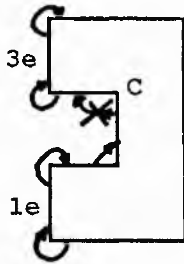
4 snapshots of the Region list initialisations occuring around the butt-ends.

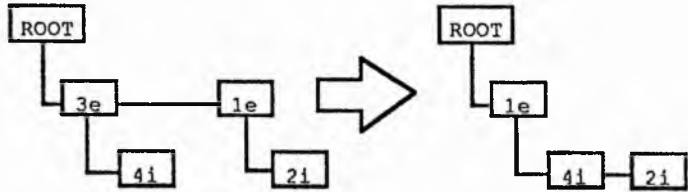


These Region lists would be correct if the butt-ends were in separate convex polygons.

Figure 3.36

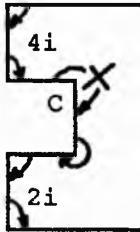
Repairing the hierarchy when conflicting labels have the same type.

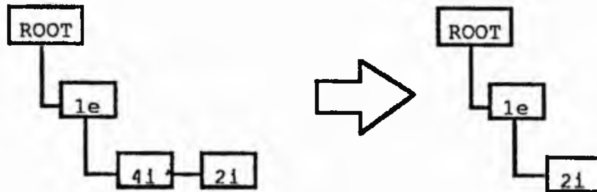




At vertex C, Exterior Regions 1e and 3e meet.

Region label 3e is removed from the hierarchy.
The other Region in the conflict 1e inherits 3e's 'children'.





At vertex C, Interior Regions 2i and 4i meet.

Region label 4i is removed from the hierarchy.
The other Region in the conflict 2i inherits 4i's 'children' if it has any.

Figure 3.37

The nomination of the redundant label is purely arbitrary, and by way of example we shall choose the label which was most recently created. Having chosen the redundant label, it must be removed from the hierarchy. It is removed from the Child list of its parent and it is also removed from the Sibling list of other children of its Parent. The remaining problem is to find where to put any descendants of the redundant label. Since the Concatenation is effectively merging two Labels into one, the obvious and correct solution of this problem is to merge the descendants of the Redundant label with those of the surviving label. This is performed by joining the list of Children of the Redundant Label to the list of children of the surviving label (Fig. 3.37).

3.9.7 Conflict - Labels have different types.

Conflicts of this type occur in shapes with more than one left-handed 'butt-end'. Left-handed 'butt-end's occur at Junctions with only a left-hand side.

Shapes with more than one left-handed 'butt-end' always have cavities eating into the right-hand of the shape. The turning points of such cavities always form right-handed 'butt-end's. These right-handed 'butt-end's, which are always encountered by the sweep before the left-handed 'butt-end's, are the origin of the labelling conflict. The plane-sweep mistakenly assumes the Junction where the cavity's turning point lies is a Right-handed 'butt-end' marking the start of a convex polygon. Two Region-lists are initialised here, an Exterior Region around the Left-hand side of the cavity, and an Interior Region around the right-hand side of the cavity. When one of the left-handed 'butt-ends' is encountered, the labelling conflict is identified (See Figures 3.38 and 3.40 for examples).

Inside the shape, the Interior Region originating from the Left-most turning point of the shape meets the Exterior Region initialised around the cavity. The Exterior Region-label must be disposed of.

Outside the shape, the Exterior Region originating from the Left-most turning point of the shape meets the Interior Region initialised around the cavity. The Interior Region label must be disposed of.

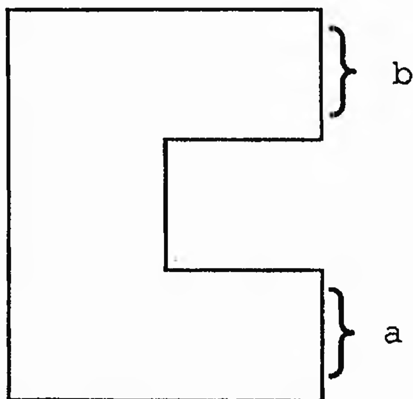
Having identified the Redundant label, which is always the label most recently generated, the problem of how repair the Containment Hierarchy arises. Removing the redundant label from the Hierarchy is simple enough, but the solution to the problem of where to place the Children of the redundant label is less obvious than in conflicts where both labels have the same type.

The position in the drawing of the children of the redundant label should be considered. The Redundant label was created under the assumption that it was a convex polygon, and the label was generated at the left-most vertex of that assumed polygon. Therefore any children of the Redundant Region lie to the right of the Edges around which it was

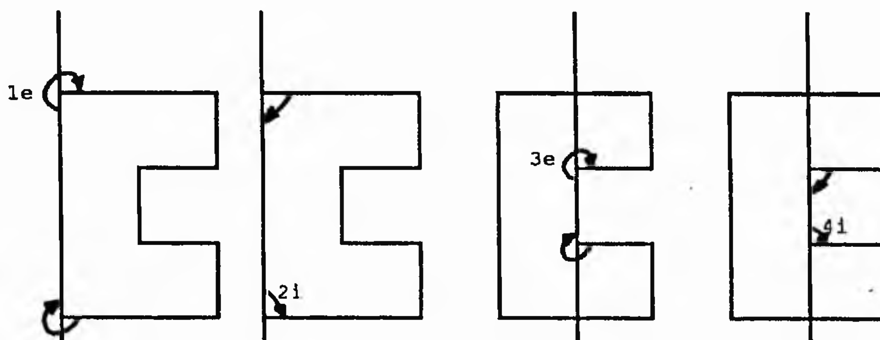
formed, and therefore these children lie outside of the actual shape, but within the cavity. These children then are Siblings of the actual shape. Figures 3.39 and 3.41 illustrate how the hierarchy is repaired.

The Redundant label, regardless of type, is always a Child of the surviving label. The children of the Redundant label are therefore always of the same type as the surviving label, and so altering their position within the Hierarchy so that they become Siblings of the survivor maintains the consistency of the Hierarchy.

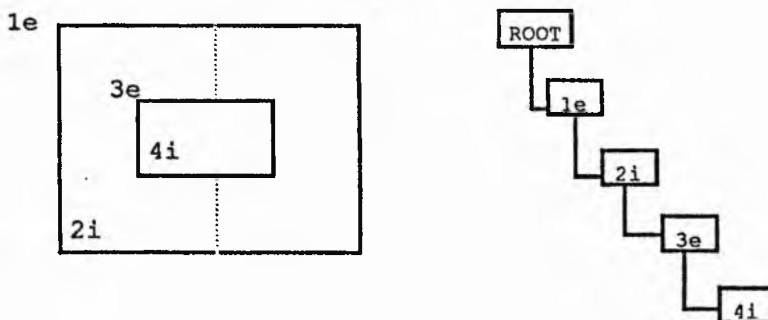
Labelling Conflict: Labels have opposite types.



Shape with two left-hand butt-ends a and b.



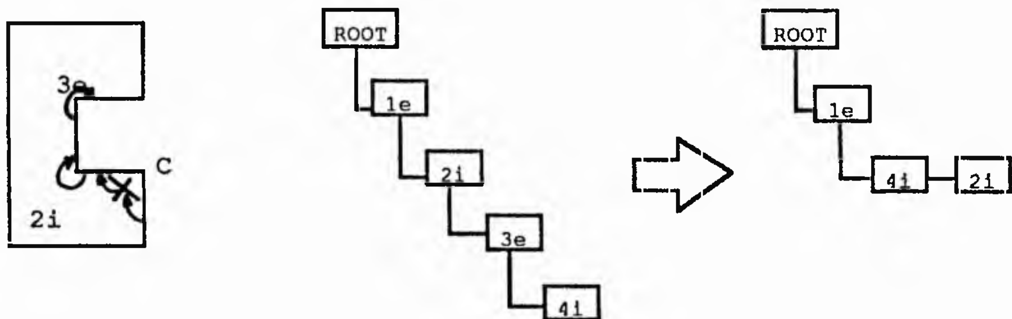
4 snapshots of the Region list initialisations.



These Region lists would be correct if the Region lists were initialised around nested convex polygons.

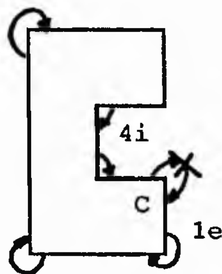
Figure 3.38

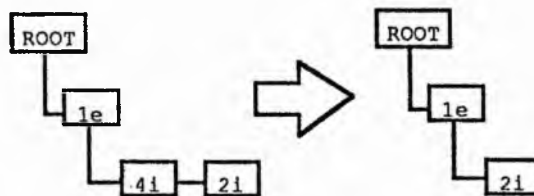
Repairing the hierarchy when conflicting labels have opposite types.



At vertex C, Interior Region 2i and Exterior Region 3e meet.

At Vertex C the most recently created Region list (3e) in the conflict is removed from the hierarchy.
3e's 'children' become 'siblings' of 3e's 'parent' - 2i.



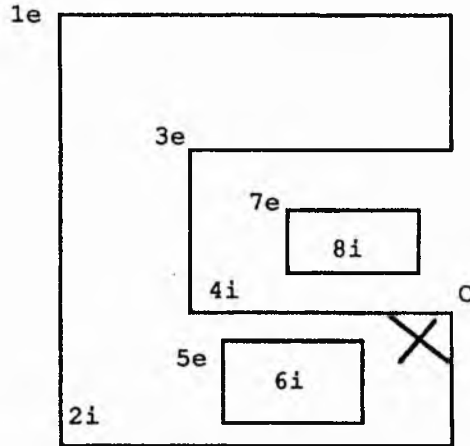


At vertex C, Exterior Region 1e meets Interior Region 4i.

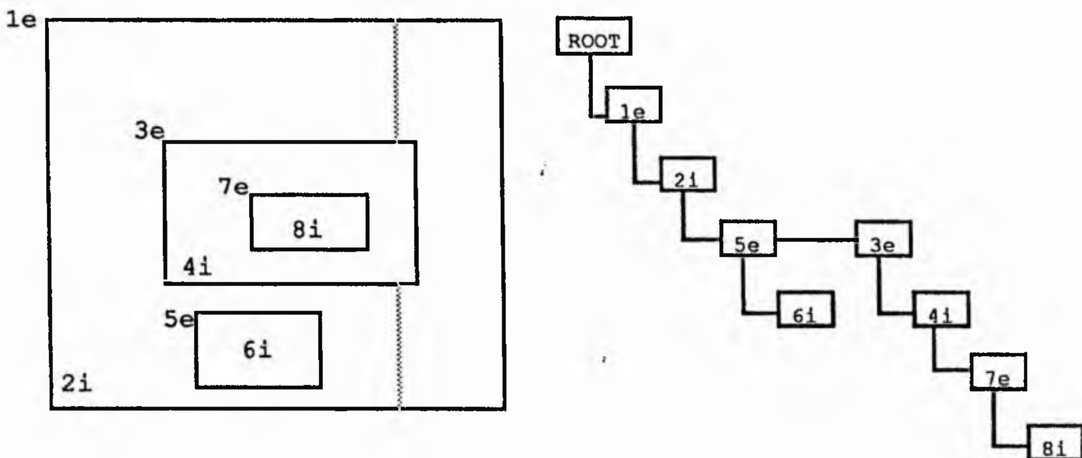
At Vertex C the most recently created Region list (4i) in the conflict is removed from the hierarchy.
4i's 'children' become 'siblings' of 4i's 'parent' - 1e.

Figure 3.39

Labelling conflict when labels have opposite types - a more complex example.



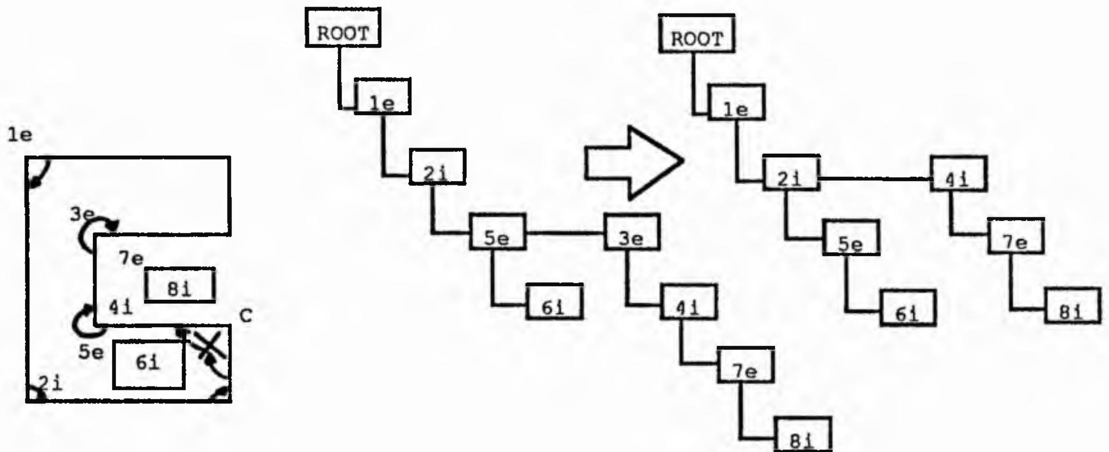
A scene similar to that in the previous diagram, again a labelling conflict involving opposite types being discovered at vertex C. By this time however, some of the Regions involved in the conflict have 'children'.



The labelling hierarchy built would be consistent with a scene constructed from nested convex polygons.
Note how 4i contains 7e, and how 5e and 3e are 'Siblings'.

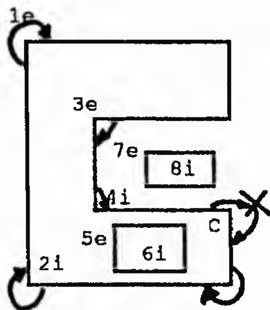
Figure 3.40

Repairing the hierarchy when conflicting labels have opposite types - a more complex example.



At vertex C, Interior Region 2i and Exterior Region 3e meet.

At Vertex C the most recently created Region list (3e) in the conflict is removed from the hierarchy.
3e's 'children' become 'siblings' of 3e's 'parent' - 2i.



ROOT
1e
2i
4i
5e
7e
6i
8i

ROOT
1e
7e
2i
8i
5e
6i

At vertex C, Exterior Region 1e meets Interior Region 4i.

At Vertex C the most recently created Region list (4i) in the conflict is removed from the hierarchy.
4i's 'children' become 'siblings' of 4i's 'parent' - 1e.

Figure 3.41

CHAPTER 4.

4. The Reconstruction Process.

4.1 Introduction.

Reconstruction is the process of building a three-dimensional description of an object by using the information provided by an engineering drawing of that object.

This chapter describes the Reconstruction program which has been produced. The program relies on the user to perform certain visual tasks which cannot presently be automated. The interaction with the user is briefly described in section 4.2, this giving a feeling of what parts of the Reconstruction process the program can perform automatically. The rest of the chapter describes the principles and procedures employed by the program to perform its part of the Reconstruction process.

A significant principle of Reconstruction is the matching of the features of components presented in the separate views. Section 4.3 describes the correspondences occurring between the images of components in the separate views. Section 4.4 considers the implications of another set of correspondences, this time those between the co-ordinate systems of the three-dimensional object and the co-ordinate system of the drawing.

The Reconstruction procedures employed by the program are divided into two categories.

The first category contains the procedures required to retrieve a three-dimensional description from the engineering drawing, these described in section 4.5.

Section 4.6 describes the second category which are the procedures required to convert this three-dimensional description into a more convenient and conventional surface-based description suitable for input to standard three-dimensional graphics display algorithms.

4.2 The user and the Reconstruction program.

The Reconstruction program builds a description of an object component by component.

The Reconstruction program relies upon the user to identify and select the cross-sections of

each component in the drawing. Each cross-section consists of one or more connected partitions in the drawing. A cross-section is identified by selecting all the individual partitions from which it is composed.

The data structures expressing the topology of the drawing are read from a file, and the drawing is displayed on the screen. A cross-hair cursor which can be moved in response to keyboard key presses is then positioned on the screen.

The user selects partitions in the drawing by moving the cursor over the desired partition and by then pressing the select key. The selected partition is recorded and is highlighted on the screen by filling its outline with colour. When all the partitions describing a cross-section have been selected, the terminate key is pressed. The process of selecting partitions is illustrated in figure 4.1.

The program then searches the drawing file for evidence supporting the assumption that a component with such a cross-section exists. If such evidence is found, it provides sufficient dimensioning information to construct a solid from the cross-section. If no such evidence is found, the cross-section is admitted to be invalid and is rejected.

The process of constructing solid components from cross-sections is repeated until the user decides that the object has been completely described, when the command can be given to stop the process.

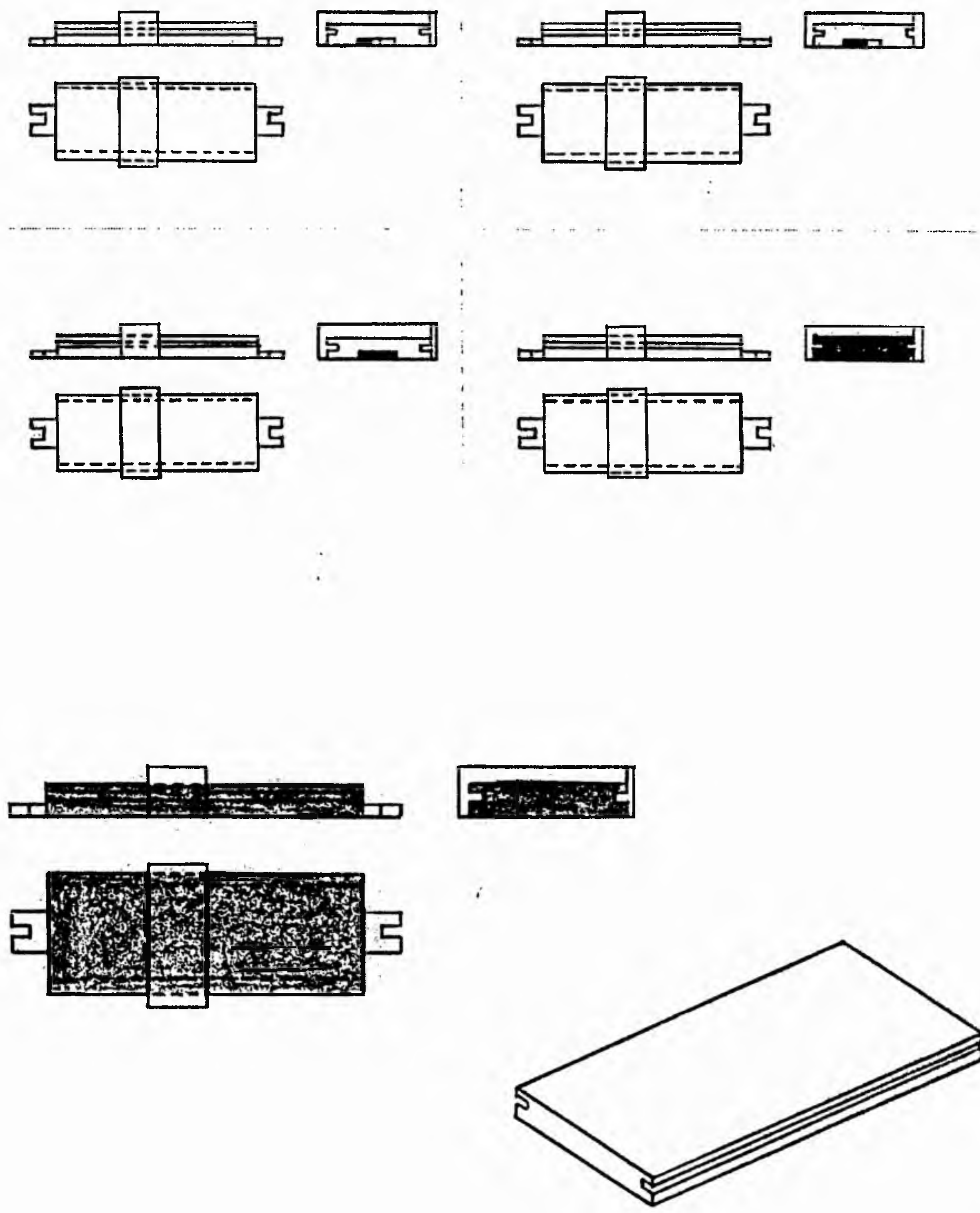


Figure 4.1

4.3 Correspondences between views.

This reconstruction process is only capable of building solids which are composed of uniform-thickness components. A uniform-thickness component is a solid which has a constant cross-section along an axis, and that axis is perpendicular to the plane of the cross-section (fig. 4.2). The cross-section may be a shape of arbitrary complexity, composed of any curve types.

Allowing a simplifying assumption to be made that the uniform-thickness component is only viewed from a line of sight parallel or perpendicular to the axis of the component, the observation may be made that any series of orthographic projections of the component would always show the cross-section in one view, and a rectangle in each of the other two views (fig 4.3). Therefore, given a two-dimensional cross-section of a component, all that need be done to construct a solid is to identify the corresponding rectangles in either or both of the other views and to extract the appropriate dimensioning information from them.

The cross-section in one view and the corresponding rectangle in another view share a common dimension. For example, if the cross-section was in the XY view and the rectangle was in the YZ view then the common dimension would be the Y dimension. The cross-section and the matching rectangle would both be of the same length and position in the Y dimension. Further, for every unique Y component of the vertices of the cross-section there would be a line, perpendicular to the cross-section, partitioning the rectangle in the YZ view. This additional property can be used as a more selective criteria when attempting to find a matching rectangle for a given cross-section - the rectangle must have a partitioning line for each unique vertex component in the cross-section.

The correspondences between views are elaborated in the next few sections, structured around the common dimension between pairs of views - the X dimension is common to the XY and XZ views, the Y dimension is common to the XY and YZ views, the Z dimension is common to the YZ and XZ views (fig 4.4). In order to simplify the expositions, correspondences are described in terms of the three dimensional co-ordinate axes of the object without worrying about how this three dimensional system is represented in the two dimensional drawing system. The complexities and complications involved in projecting the three dimensions of the object onto the orthographic views are discussed in section 4.4.

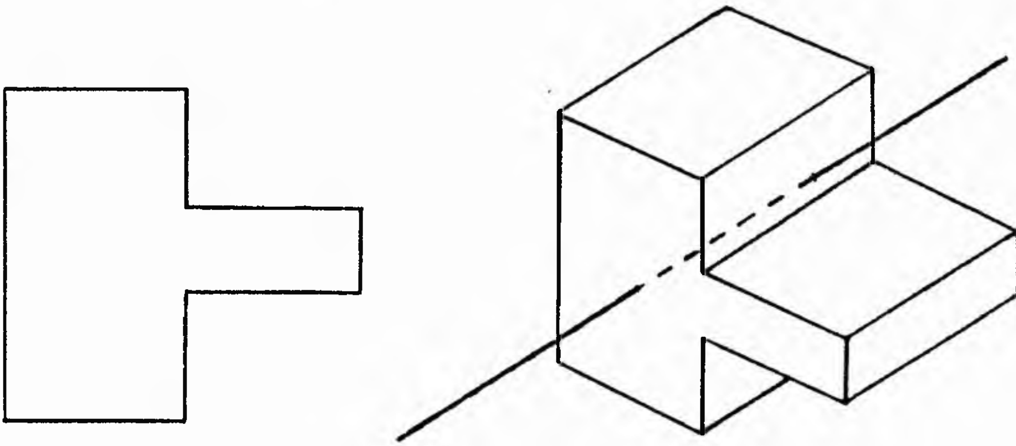
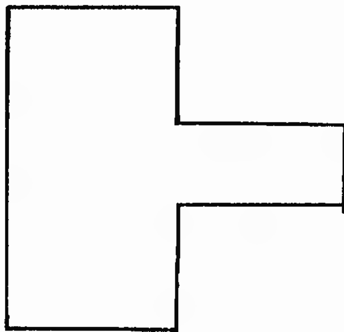
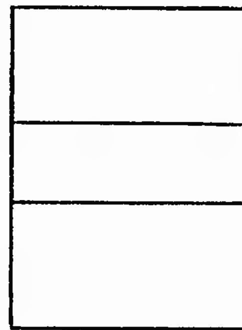


Figure 4.2





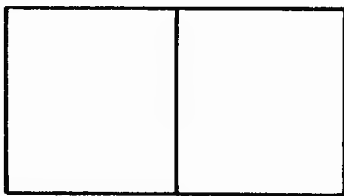


Figure 4.3

Correspondences between views.

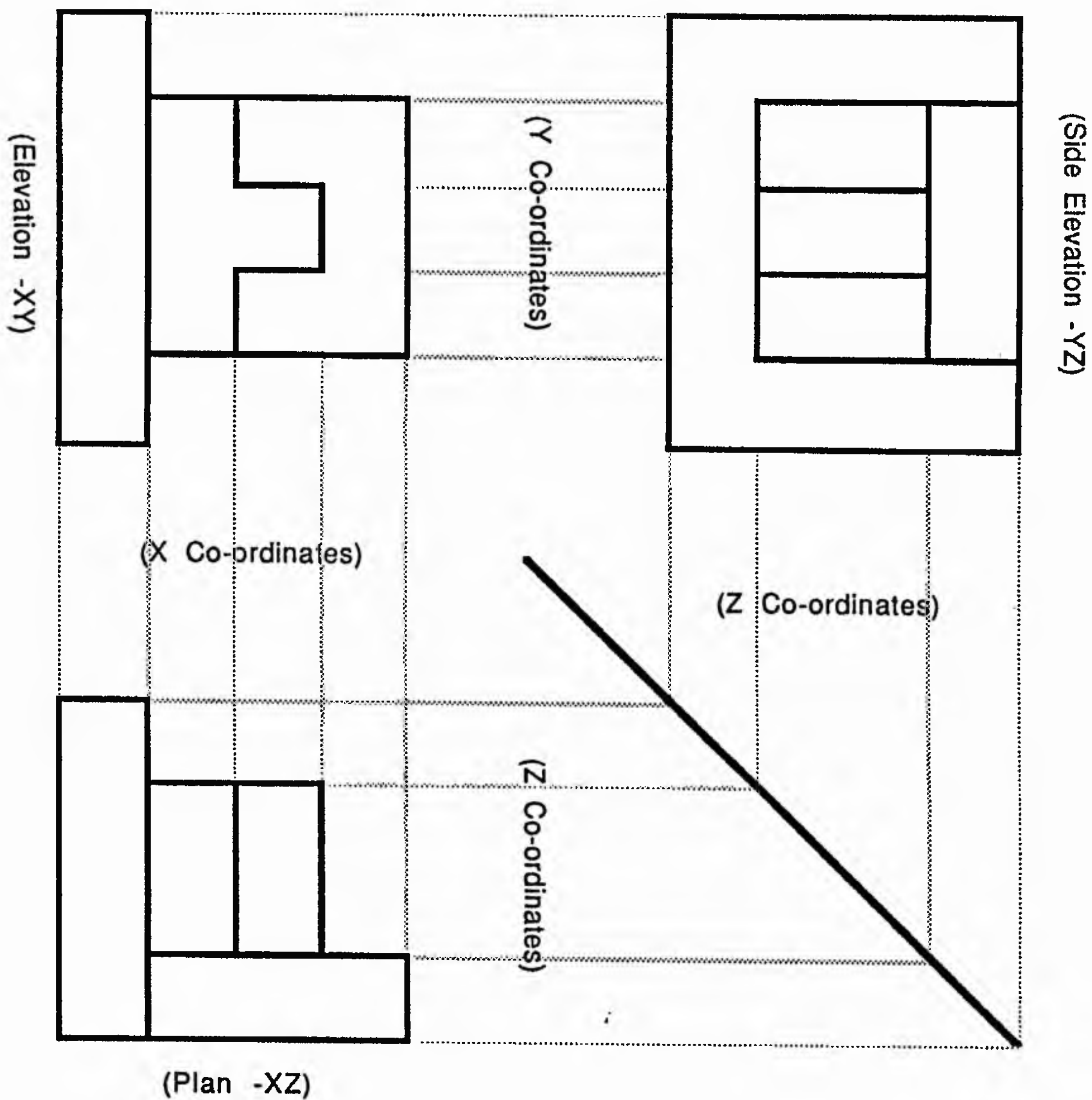


Figure 4.4

4.3.1 X is the common dimension.

The X dimension is the common dimension between the XY and the XZ views.

A cross-section in the XY view is supported by a rectangle in the XZ view, and in the reverse case, a cross-section in the XZ view is supported by a rectangle in the XY view (fig 4.5).

In both the cases mentioned above, the cross-section corresponds to a rectangle in the other view with corners at the minimum and maximum X points of the cross-section. This rectangle must be partitioned by lines perpendicular to the plane of the cross-section, one occurring for each unique X component of the set of vertices of the cross-section. The perpendicular length, with respect to the plane of the cross-section, of the sides of the rectangle are used to set the length of the axis of the component.

In the case where the cross-section is in the XY view the axis is in the Z dimension. In the reverse case, where the cross-section is in the XZ view, the axis is in the Y dimension.

X is the common dimension -
the location of the axis in the corresponding view.

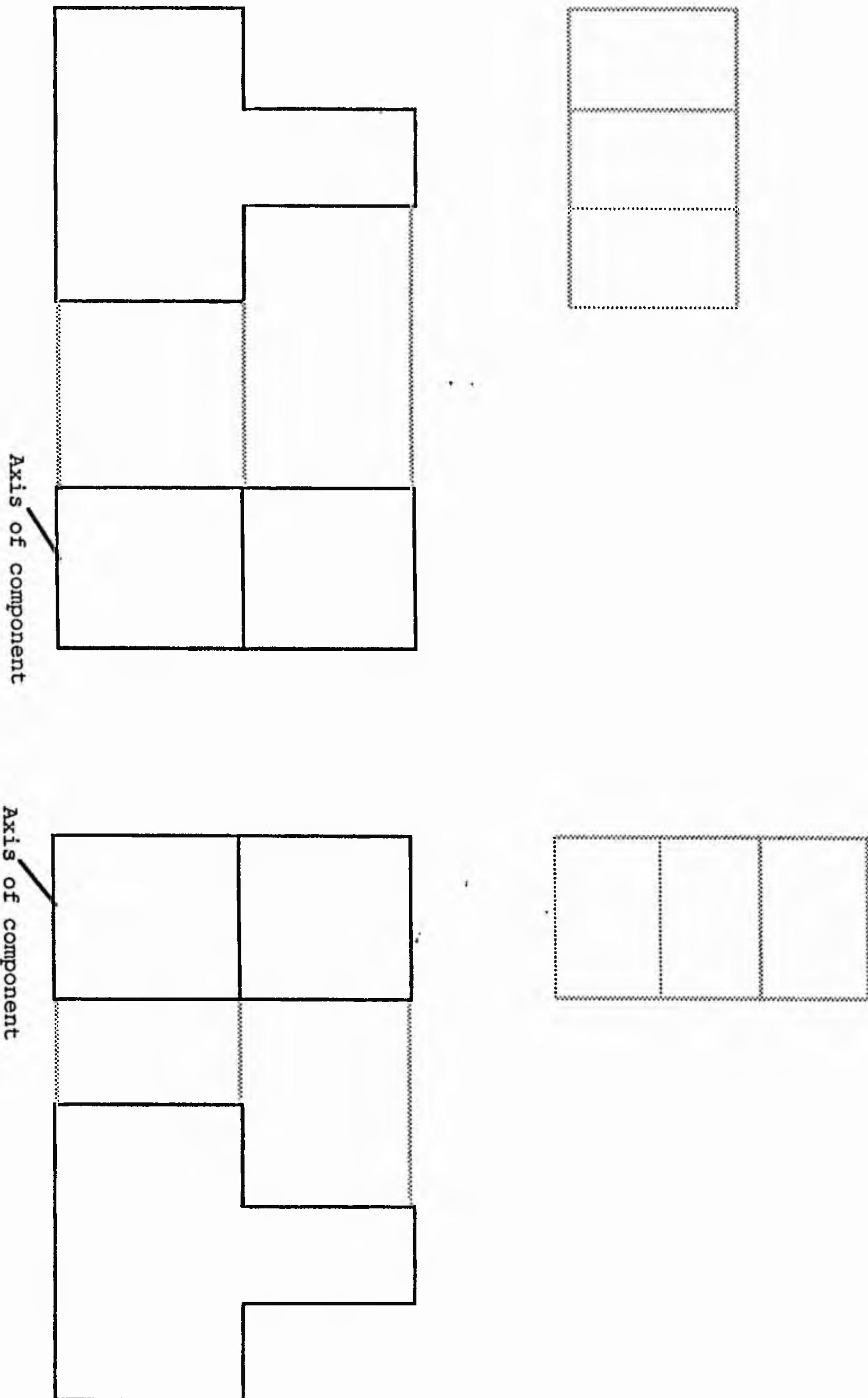


Figure 4.5

4.3.2 Y is the common dimension.

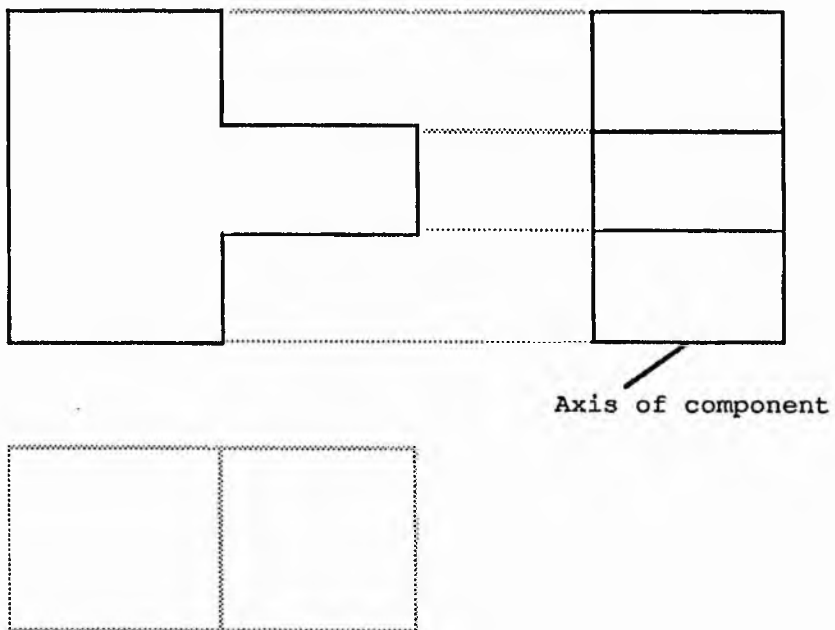
The Y dimension is the common dimension between the XY and the YZ views.

A cross-section in the XY view is supported by a rectangle in the YZ view, and in the reverse case, a cross-section in the YZ view is supported by a rectangle in the XY view (fig 4.6).

In both cases, the cross-section corresponds to a rectangle in the other view with corners at the minimum and maximum Y points of the cross-section. This rectangle must be partitioned by lines perpendicular to the plane of the cross-section, one occurring for each unique Y component of the set of vertices of the cross-section. The perpendicular length, with respect to the cross-section, of the sides of the rectangle are used to set the length of the axis of the component.

In the case where the cross-section is in the XY view the axis is in the Z dimension. In the reverse case, where the cross-section is in the YZ view, the axis is in the X dimension.

Y is the common dimension -
the location of the axis in the corresponding view.



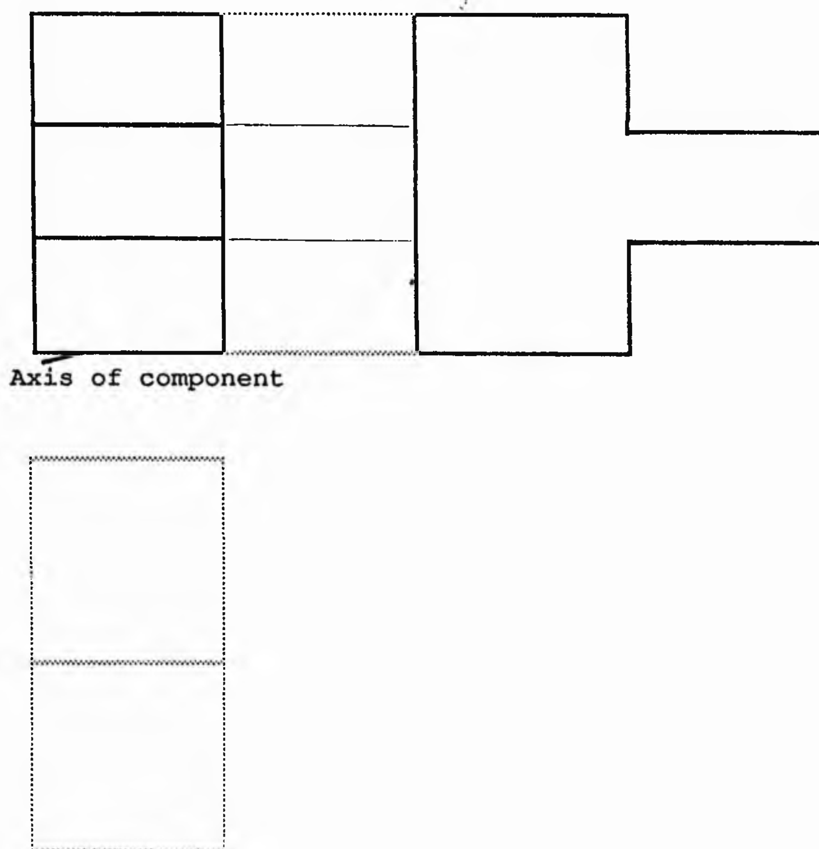


Figure 4.6

4.3.3 Z is the common dimension.

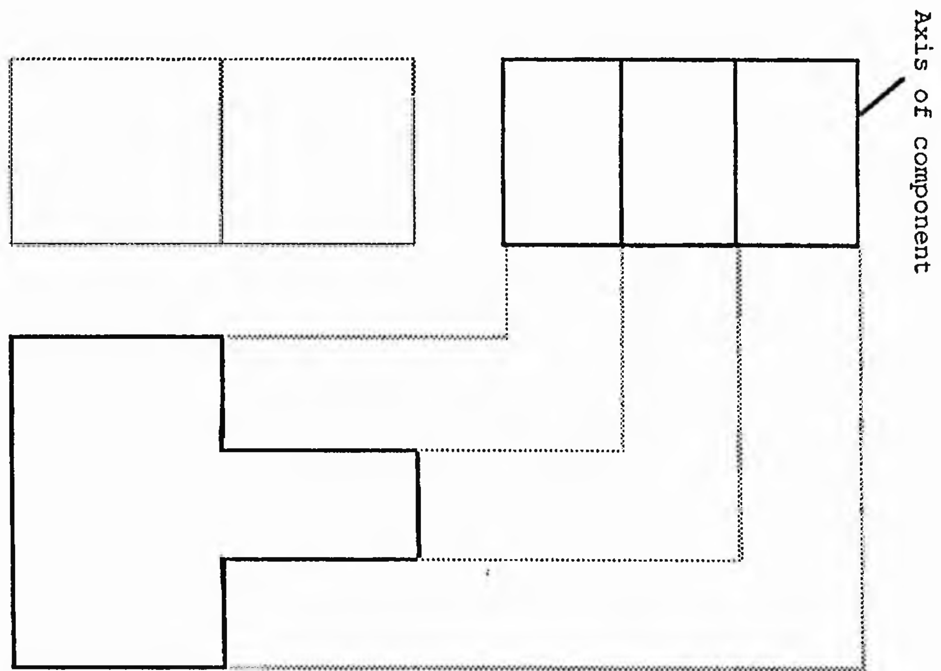
The Z dimension is the common dimension between the XZ and the YZ views.

A cross-section in the XZ view is supported by a rectangle in the YZ view, and in the reverse case, a cross-section in the YZ view is supported by a rectangle in the XZ view (fig 4.7).

In both cases, the cross-section corresponds to a rectangle in the other view with corners at the minimum and maximum Z points of the cross-section. This rectangle must be partitioned by lines perpendicular to the plane of the cross-section, one occurring for each unique Z component of the set of vertices of the cross-section. The perpendicular length, with respect to the cross-section, of the sides of the rectangle are used to set the length of the axis of the component.

In the case where the cross-section is in the XZ view the axis is in the Y dimension. In the reverse case, where the cross-section is in the YZ view, the axis is in the X dimension.

Z is the common dimension -
the location of the axis in the corresponding view.



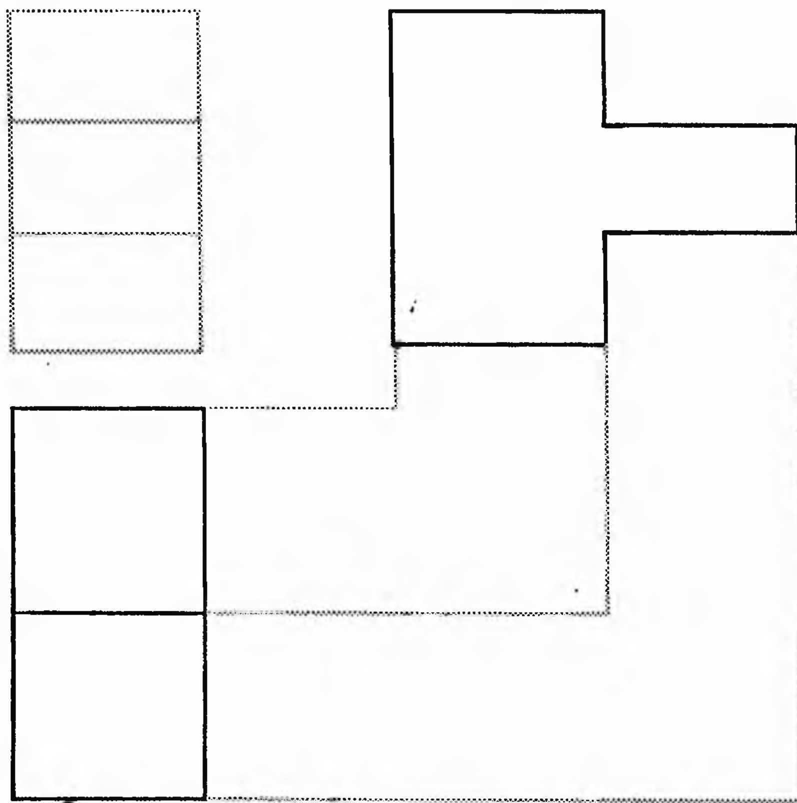


Figure 4.7

4.4 Mapping three-dimensional co-ordinates onto two-dimensional drawing co-ordinates.

In the figure 4.8, an illustration is given of the two principle systems of orthographic projections, first-angle projection at the top, and third-angle projection at the bottom. In both illustrations, the z-axis origin is along the intersection of the planes X=0 and Y=0, and the value of z increases as we draw away from the page to the right.

The first point to notice is that the three-dimensional views of the object are all drawn on the same plane, the drawing plane, which is conventionally described in terms of an XY co-ordinate system. Conveniently, the dimensions of the orthographic views common to the XY drawing system correspond in a simple manner. The X co-ordinates of the XY and XZ views increase as the X co-ordinates of the drawing system increase, and the Y co-ordinates of the XY and YZ views increase as the Y co-ordinates of the drawing system increase. All that is needed to transform an X or a Y point in the drawing into an X or Y point in the object is a straight forward translation followed by a scaling operation. Each view has its own independent translation distance. The translation distance is simply the minimum X or minimum Y drawing co-ordinate of the set of points of that view.

The second point to note is that the Z co-ordinates of the object are mapped onto the XY drawing co-ordinate system. The Z co-ordinates of the YZ view are mapped onto the X co-ordinates of the drawing system, and the Z co-ordinates of the XZ view are mapped onto the Y co-ordinates of the drawing system. Referring to the plate we can see that in either of the projection systems the Z co-ordinates of the object increase as the X co-ordinates of the YZ view increase, but decrease as the Y co-ordinates of the XZ view increase. Therefore the mapping between the representation of the Z dimension in the drawing and the Z dimension of the object is, in the case of the YZ view, a transformation involving translation and scaling. In the case of the XZ view, the transformation requires an additional reflection operation to map drawing co-ordinates to object co-ordinates or object co-ordinates to drawing co-ordinates.

Using the preceding observations, we can now describe the mapping between drawing co-ordinates and object co-ordinates.

i) Translate the vertices of each view to the drawing origin. The minimum X and Y

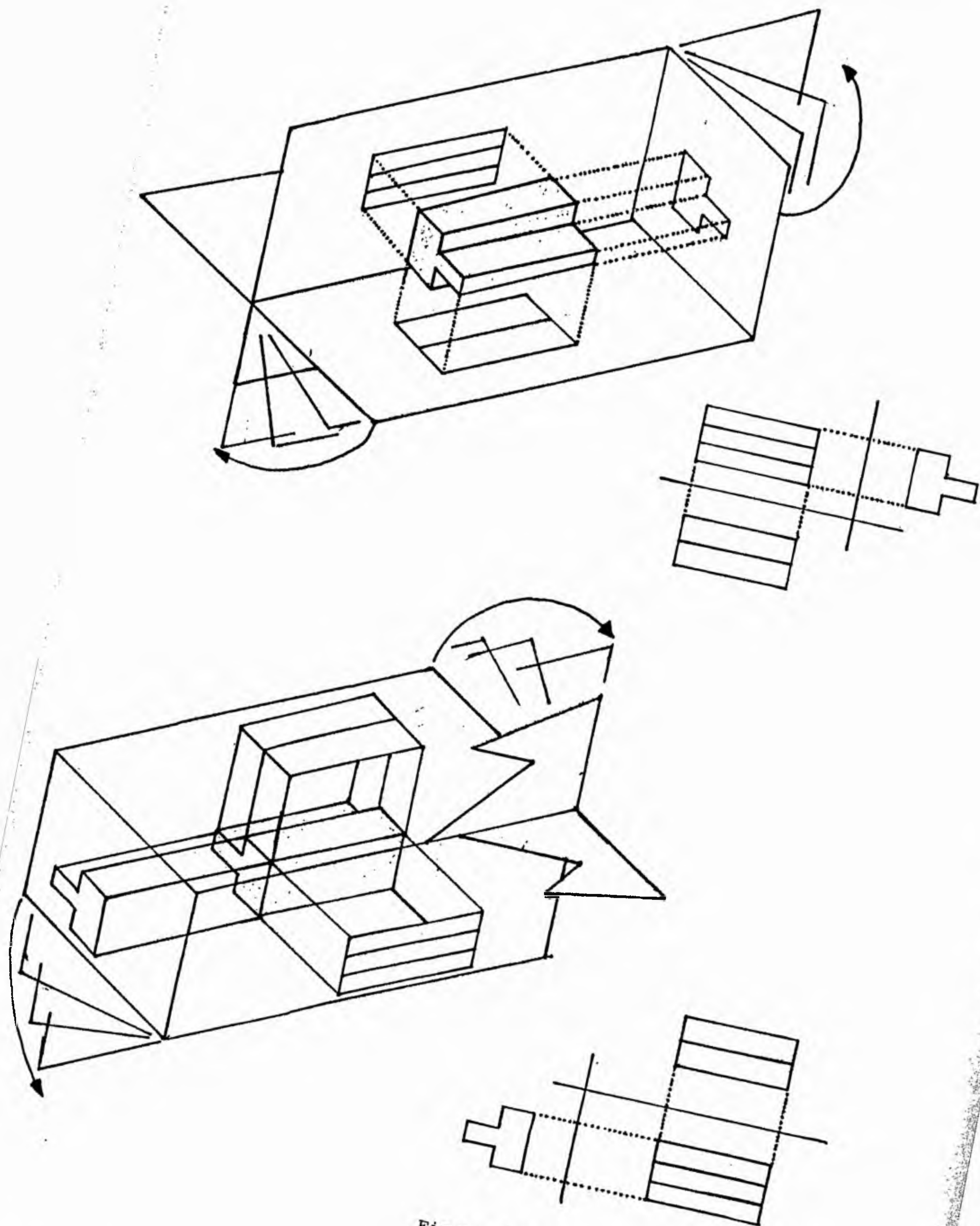


Figure 4.8

drawing co-ordinates of the set of vertices of a view are found, and these are subtracted from the X and Y co-ordinates of all vertices of the view.

ii) Reflect the Z components of the vertices of the XZ view by applying the following formula to the Z component of each vertex :-

```
reflected_Z_component = minimum_z_component_of_view
                                          +
                        maximum_z_component_of_view
                                          -
                        Z_component_to_be_reflected;
```

It should be borne in mind that the Z components referred to are the Y components of the drawing co-ordinates of the XZ view. Minimum_Z_component_of_view is minimum_Y of view XZ. Maximum_Z_component_of_view is maximum_Y of view XZ. Also, after the translation performed in (i) above, the minimum_Z_component_of_view of view XZ would probably be zero.

4.5 The Reconstruction Procedures.

4.5.1 An overview of the Reconstruction Procedures.

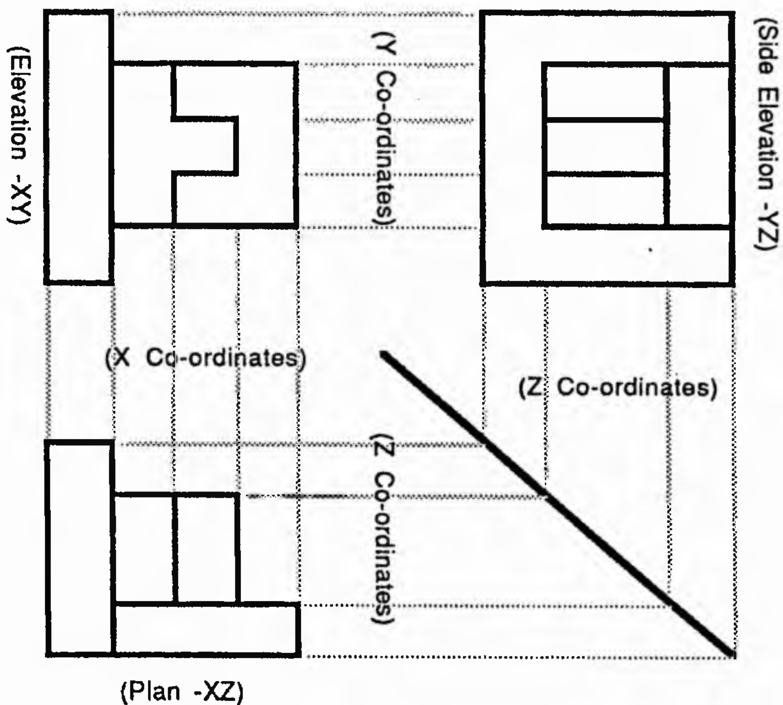
The Reconstruction program accepts from one of the drawing views a set of partitions identified by the user as being a candidate cross-section section of a component in the drawing. The program performs a matching procedure to attempt to find rectangles in the other views of the drawing which correspond in position and size to the candidate cross-section. As a result of the matching procedure, either the candidate cross-section is rejected on the grounds that no corresponding rectangle can be found, or some dimensioning data is returned which enables the two-dimensional cross-section to be expanded into a three-dimensional component. This dimensioning data describes the length and position of the axis of the component.

Section 4.3 discussed the correspondence between a cross-section in one of the views and rectangles in the other two views. It was observed that every vertex in the cross-section matched to partitioning lines in the corresponding rectangles. This property exerts an extra constraint on the matching procedure, namely that not only are the position and size of rectangles to be considered when attempting to find matches, but also the presence of partitioning lines inside the rectangle.

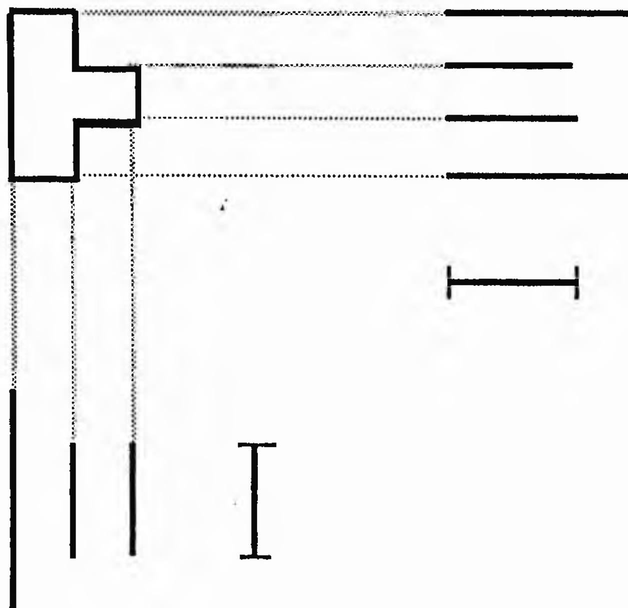
In order to simplify the matching task, the partitioning constraint is exploited. Any rectangle matching the cross-section would contain one partitioning line for each unique vertex position in the common dimension of the cross-section and the supporting view being searched. These partitioning lines would all be of the same length, and would start and end at the same points along the dimension of the axis being sought. Instead of looking for rectangles in the supporting views, all the matching procedure needs to do is to find all the lines which correspond with the vertices of the cross-section, and then find the places along the dimension of the axis where these lines overlap. Only overlaps where every vertex of the cross-section is represented provide sufficient evidence to assume a satisfactory match (fig 4.9).

The length and positions of any overlaps satisfying these conditions are assumed to be the length and position of the axis of the component. Matches are further validated by

Evidence of Rectangles.



Correspondences between views.



Search for overlapping lines along common dimension between pairs of views.

Figure 4.9

ensuring that the cross-section has a match in both of its supporting views, and that both matches return the same dimensioning data. When the matching procedure finds more than one satisfactory overlap, it is assumed that more than one component with that cross-section exists, and one three-dimensional component is constructed for each set of dimensioning data.

The matching procedure consists of four stages, and these stages are described in more detail in the next four sub-sections. The first stage accepts the list of connected partitions chosen by the user and constructs the hull of the shape containing those partitions. The second stage creates targets for the attempted matches with the other views. The third stage searches through the other views, and records all the lines which hit parts of the targets. The fourth stage examines the list of lines which hit the targets, looking for overlaps where all parts of the targets are hit simultaneously. The position of the overlaps, if any exist, are passed to the construction procedure along with the hull of the cross-section. The construction procedures are described in section 4.6.

4.5.2 Determining the hull of the cross-section.

This procedure makes use of some of the data structures derived from the drawing by the Region Analysis Program presented in chapter 3. A number of separate list structures are used to describe some of the topological relationships between the features of the drawing.

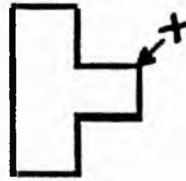
One ordered list of edges exists for each Region in the drawing describing its boundary. For convenience this list will be referred to as the Region List. A Region list exists for every partition in the drawing, describing the interior boundary of the partition. Region lists which describe the interior of a partition are called Interior Region Lists. A Region list also exists for every conglomerate of adjacent partitions. Such a Region List is called an Exterior Region List, and it describes the Exterior boundary of shapes made up of one or more partitions.

Interior Region Lists contain edges ordered in a counter-clockwise direction around the inside of the partition. The head of the list is the top-most edge coming from the top-most vertex of the right-most vertices of the Region (fig 4.10).

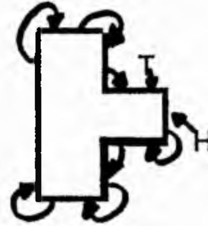
Each item in the Region List is an Edge Descriptor. These Edge Descriptors define the vertices of the edges and also identify the Regions on each side of the edge. Every edge has two sides and each may participate in a separate partition boundary, although degenerate cases may occur where both sides participate in the same boundary. A naming convention has been adopted to differentiate between the two sides. One side is the the Below-Right side, which is the side facing downwards of any non-vertical edge, or the side facing right of any vertical edge. The other is the Above-Left side, facing upwards from any non-vertical edge, or facing left from any vertical edge. Figure 4.11 shows two connected partitions with their vertices labelled. Beneath them is the Line list which describes the shape. Each Line list item contains an Edge List, and each Edge List item contains an entry showing the connection from each side of the edge to the next edge in the corresponding Region List.

The cross-section is composed of the union of one or more partitions. In the trivial case where the cross-section consists of only one partition, the hull of the cross-section is simply

Region Lists of Shapes.

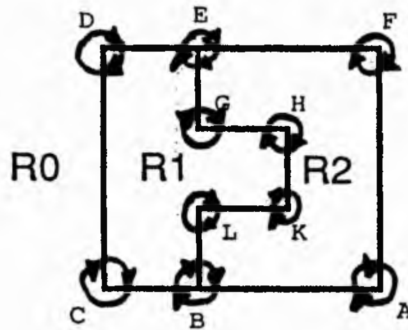


Top-most
right-most
vertex.



Region List:
H-head, T-tail.

Figure 4.10



Line-list For Above Shape.

```
[
[L#CA,[1,CB,[BL,1,left,R1],[CD,1,left, R0]],
      [2,BA,[AF,1,left,R2],[CA,1,below,R0]]]
[L#CD,[1,CD,[DF,1,above,R0],[CA,1,above,R1]]]
[L#DF,[1,DF,[DE,2,above,R0],[CD,1,right,R1]],
      [2,EF,[AF,1,right,R0],[GE,1,right,R2]]]

[L#AF,[1,AF,[DE,2,below,R2],[CA,2,below,R0]]]
[L#BL,[1,BL,[LK,1,above,R1],[CB,2,above,R2]]]
[L#LK,[1,LK,[HK,1,left,R1],[BL,1,right,R2]]]

[L#KH,[1,KH,[GH,1,below,R1],[LK,1,below,R2]]]
[L#GH,[1,GH,[KH,1,right,R2],[EG,1,left,R1]]]
[L#GE,[1,GE,[DF,1,below,R1],[GH,1,above,R2]]]
]
```

Line-list Item:
[Line-id,[Edge-list]];
Edge-list Item:
[Edge-id,Vertices,[Above-left Connection],[Below-right Connection]]]
Connection:
[Line-id,Edge-id,Side,Region List Id];

Figure 4.11

the boundary of that partition.

To determine the hull of a cross-section which is composed of many partitions, it is necessary to determine the top-most vertex of the right-most vertices of the set of partitions. This vertex is found because it is guaranteed to lie on the hull. From this starting vertex a navigation algorithm is applied to follow the edge lists around the outside of hull (fig 4.12(a)).

The navigation algorithm begins at the starting vertex and follows the edges around the outside of the hull, building a list of the vertices of the hull as it moves. The first vertex to be recorded is the starting vertex.

The navigation algorithm must now decide which edge to follow. The set of edges which are candidates to be followed are those edges which emerge from the starting vertex. The navigation algorithm must search through the candidate edges to find that edge which is closest, following a clockwise direction, to the twelve o'clock position (fig 4.12(b)). Along that edge, the vertex opposite to the starting vertex is added to the description of the hull.

Having found this first edge, the following procedure is repeatedly applied until it arrives back at the first edge. In the description of this procedure, the most recently obtained vertex is referred to as the 'Current Vertex' and the most recently obtained edge of the hull is referred to as the 'Current Edge'.

Starting from the Current Edge search around the Current Vertex in a clockwise direction until the candidate edge furthest from the Current Edge is reached (fig 4.12(c) and 4.12(d)). This furthest edge is nominated as the new Current Edge, and the vertex along that edge and opposite to the Current Vertex is nominated the new Current Vertex. The new Current Vertex is added to the description of the hull.

An illustration of this procedure is given in figure 4.13 showing the steps involved in finding the hull of a section composed of six partitions.

From this description of what the navigation procedure does, an implementation is derived which performs the navigation by using the Region lists and Edge Descriptors. In the following descriptions, the Region List is assumed to be a cyclical list where the tail is succeeded by the head. The next entry in a list beside a current entry is that entry which is next nearest the tail, unless the current entry is the tail entry in which case the next entry is the head entry.

The top-most vertex of the right-most vertices of the set of edges is found by searching through the set of Edge Descriptors. Having found this starting vertex, any one of the edges emerging from this vertex should be chosen.

By definition, all edges emerging from the starting vertex are to the left, or are vertical and below the starting vertex. Therefore, the most clockwise edge emerging from the starting vertex will be from the six o'clock position up to twelve o'clock. The Below-Right side of this edge will face into one of the selected partitions, and the Above-Left side will face into a partition which has not been selected as part of the section.

Searching through the edges around the starting vertex, the first edge on the hull is found when the edge with an Above-Left side facing away from the section is found. Taking any of the candidate edges around the starting vertex, test the Above-Left side. If it indicates a partition inside the section, then another candidate must be chosen. The next candidate is selected from the Region List indicated by the Above-Left side of the current candidate edge. Looking in this Region List, the next candidate is the neighbour of the current candidate edge.
Candidates are selected and tested until one is found whose Above-Left side contains the identifier of a partition which is not in the section. This candidate is elected as the first edge.

Once the first edge on the hull has been determined, the vertex at the opposite end of the first edge from the starting vertex is appended to the list of vertices describing the hull.

The subsequent edges of the hull are determined by repeated application of the same procedure. In the following description, the 'Current Edge' is the most recently obtained

edge on the hull. 'Current Partition' is used to reference one of the partitions in the section, and is explicitly set in the description.

i) The Current Edge has two sides, one facing away from the section and the other facing into the section. The Current Partition is read from the side facing into the section. The neighbour of the Current Edge in the Region List is obtained and is nominated the candidate edge.

ii) The side of the candidate facing away from the Current Partition is examined. If it holds the identifier of a partition which is part of the section, a new candidate edge must be selected. The partition identified by the side of the current candidate facing away from the Current Partition is selected as the new Current Partition. The neighbour of the current candidate edge in the Current Partition's Region list is selected as the new candidate edge. This process of selecting and testing candidates is repeated until a candidate is found with a side facing away from the section.

iii) The current candidate is elected as the new Current Edge of the hull. The vertex is appended to the hull description.

iv) Steps (i) to (iii) are repeated until the navigation arrives back at the first edge of the hull.

Starting Vertex. (a)

3 Candidate Edges. (b)

Next Point. (c)

2 Candidates. (d)

(X's mark confirmed edges on hull.)

Figure 4.12

Figure 4.13

4.5.3 Building targets for the searches for correspondences between views.

The Reconstruction program, as outlined in section 4.5.1, is based around matching a cross-section in one view with rectangles in the other views. To find rectangles to match with the cross-section, it is sufficient to search for lines which might constitute the sides of appropriate rectangles. Besides being of the right size and in the right position, the rectangles must also contain partitioning lines which correspond to the extension of the vertices of the cross-section to three-dimensions.

An important component of the reconstruction program then is a mechanism which can, given a cross-section in one view, determine what lines should be looked for in the other views. We will call this procedure targeting. Given a cross-section in one view, it is necessary to construct targets for the line searches through the other views. The targeting procedure constructs these targets.

Before describing how the targeting procedure works, it is first necessary to establish what types of lines we look for in one view to match a cross-section in another view. With regard to the restricted object types that we are trying to reconstruct, a cross-section in one view always corresponds to rectangles in the other views whose sides are parallel to the co-ordinate axes. The sides of these rectangles are always shown as vertical and horizontal lines. The line-segments we seek to match with the cross-section lie along vertical and horizontal lines which can be drawn through the vertices of the cross-section (Fig. 4.14).

Unfortunately, this simple observation needs some qualification. Horizontal and vertical lines in the drawing co-ordinate system do not strictly correspond to horizontal and vertical lines in the object co-ordinate system. The object co-ordinate system is three-dimensional and so we have three types of lines which run parallel to the various co-ordinate axes, lines which we can call for convenience X-lines, Y-lines and Z-lines running parallel to the X-axis, Y-axis and Z-axis respectively. The views in the drawing are each in different three-dimensional planes and so a vertical line in one view may be a line in a different dimension in the object co-ordinate system than a vertical line in another view.

Setting Targets From A Cross-section.

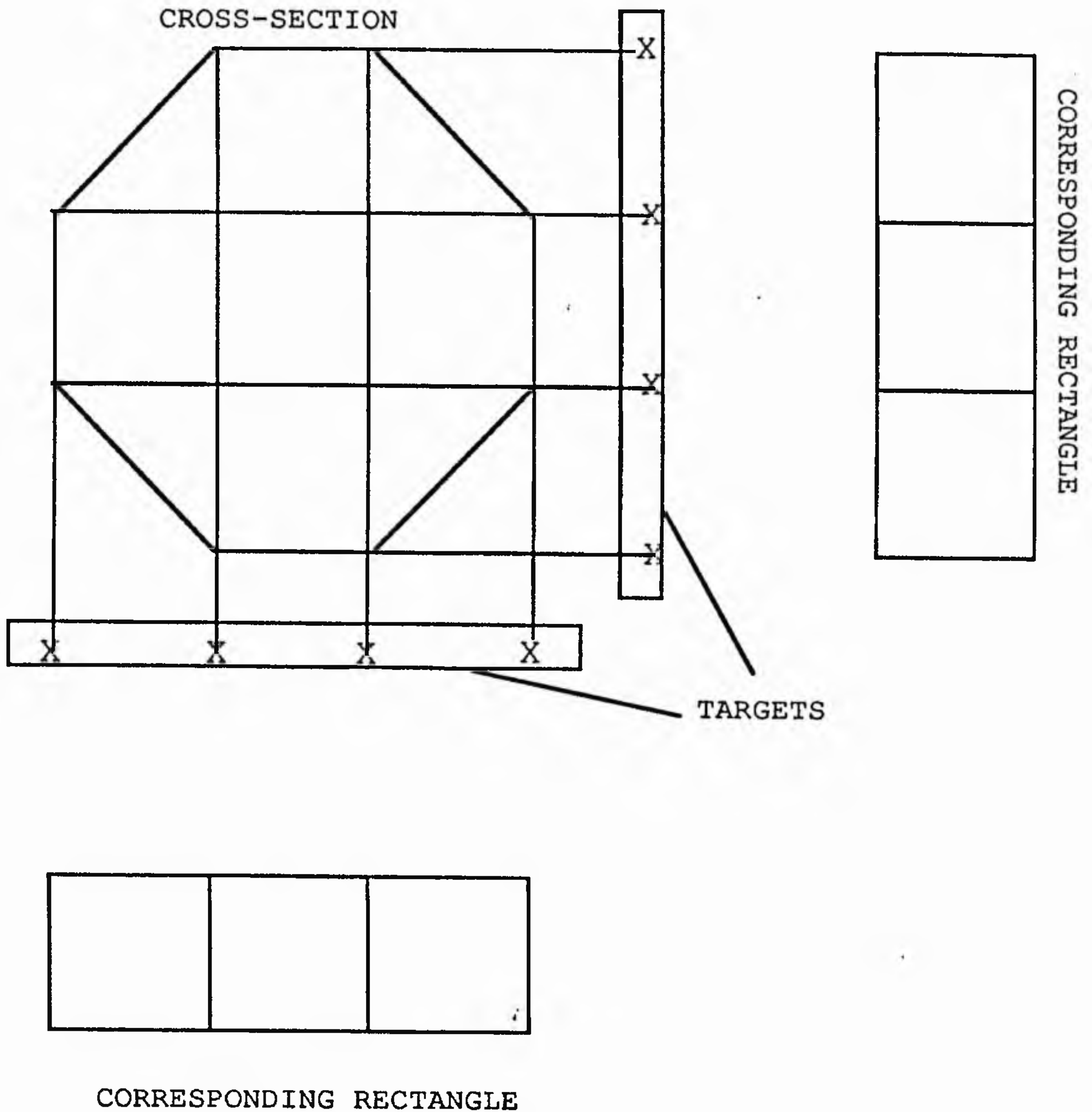


Figure 4.14

Horizontal and vertical lines passing through the vertices of the XZ view should be thought of as X-lines and Z-lines. Similarly, horizontal and vertical lines passing through the vertices of the YZ view should be thought of as Z-lines and Y-lines. Horizontal and vertical lines passing through vertices in the XY view should be thought of as X-lines and Y-lines.

Targets must be created in the object co-ordinate system and not in the drawing co-ordinate system. Each view in the drawing is the projection of a different object plane, so to be able to match features in one view with features in another view it would be necessary to convert the target into object co-ordinates and then apply a second transform to turn the object co-ordinates into the drawing co-ordinates of the view in which the match is sought. It is easier to once transform the features in each view into the three-dimensional object co-ordinate scale and to perform the matching procedures in that object co-ordinate scale.

Having converted the drawing co-ordinates into object co-ordinates by following the procedures outlined in section 4.4, the pair of targets for a cross-section in any view are created by preparing lists of the unique components of the co-ordinate points of the vertices of the cross-section, one list for the components of each dimension. A cross-section in the XZ view would produce one list containing all the unique X components of the vertices of the cross-section, and another list containing all the unique Z components.

These lists are used as targets for finding corresponding X-lines, Y-lines and Z-lines in the other views. A target created from Z components from a cross-section in one view would be used to find Z-lines in the other view with a Z dimension. A target created from X components in one view would be used to find matching X-lines in the other view with X components. A target created from Y components from one view would be used to find matching Y-lines in the other view with Y components.

4.5.4 The mechanism for searching for correspondences.

The matching mechanism is built around the plane sweep algorithm which also featured

prominently in the topological analyser program described in chapter 3. In this case, the plane sweep is used to find lines which lie close to the members of a set of target lines.

The plane sweep algorithm analyses the positional relationships between geometric entities on the same two dimensional plane, and it does this by reducing the two dimensional relationships into a series of one-dimensional relationships. One dimensional ordering of line-segments is based on relative positioning. Depending upon which dimension the ordering relation is based, we can make judgements like:-

'line #A is above line #B'
- ordering the relationship along the Y-axis;

'line #C is to the left of line #D'
- ordering the relationship along the X-axis;

'line #E is further away than line #F'
- ordering the relationship along the Z-axis.

The ordering relationships between lines is represented using an ordered list data structure called the 'front'. The tail of the list represents one extremity of the direction, and the head of the list represents the other. A sweep along the X-axis in the XY plane would maintain a front which represents the vertical ordering of the lines. The head item in the front list might represent a point at minus infinity along the Y-axis and the tail item might represent a point at plus infinity. All other items in the front list would represent lines passing through points between the two extremes, ordered on their position relative to the two extremes.

The ordering relationship between lines changes at certain points along the plane. The relationship 'line #A is above line #B' holds along a section of the XY plane until a point is reached along the X-axis when line #A is no longer above line #B. This occurs when one of three events happen at a point:-

when the furthest vertex of one or both of the lines is reached, and so one or both of

lines #A and #B no longer exist;

when another line, say line #C, is encountered which lies between line #A and line #B;

when lines #A and #B intersect and so swap relative positions so that line #B is now above line #A.

It is necessary to introduce some direction into the progression of the analysis of the lines in order to bring meaning to the terms 'start of a line' and 'end of a line'. This direction is referred to as the 'sweep direction'. The plane sweep algorithm keeps track of the changing order of lines in one direction as the sweep progesses along another direction. For example, in a plane sweep along the XY plane, the algorithm keeps track of the vertical order of lines while the sweep progresses horizontally along the X-axis. If the sweep direction in this case happens to be from left to right, then the left-most vertices of lines are the 'starts of lines' - the left-most vertex is encountered before the right-most - and the right-most vertices of lines are the 'ends of lines'. Happily in this instance vertical lines may be ignored because we are looking for lines which run parallel to the sweep direction. More generally any lines perpendicular to the axis of the plane sweep can be ignored. Such lines usually cause problems with the plane sweep paradigm in that their start and end-points occur at the same point along the sweep axis and so some additional mechanism is required to cope.

In order to maintain the correct order in the relationships between lines it is necessary to adjust the order to reflect the changes wrought by the interference of the three event types. These events are the encounter of the end of a line, the encounter of the start of a line and the encounter of the intersection point of two or more lines. The change in the ordering of the front in response to these events is called a 'transition'. A transition is the change from one state of consistency in the ordering relation to another state of consistency at a single point along the sweep axis. One transition may consist of many events, all at the same point along the axis of the sweep but all at different positions in the front. For example in a sweep along the X-axis of the XY plane, transitions might take place at several individual points along the X-axis, and each transition might consist of many events causing changes to the order of the front at many different points

along the Y-axis.

Two types of event which contribute to transitions are explicitly stated in the data describing the drawing, namely the start and end-points of lines. Intersection points can be prepared by applying an exhaustive enumeration algorithm which tests for intersection every combination of pairs of lines in the drawing data, which can be computationally expensive. Alternatively the intersection tests can be incorporated as part of the mechanism which evaluates the relationships between lines. This second approach exploits the fact that at least two lines in any set of any number of intersecting lines will be adjacent to each other before the intersection point is reached. The mechanism performs an intersection test on those pairs of lines which find themselves newly adjacent to each other as a result of the re-ordering of the front, which can lead to fewer tests being performed than under the exhaustive enumeration approach.

The plane sweep mechanism has two parts, the first of which is a pre-processing part which prepares the data structure used by the plane-sweep and the second part is the plane-sweep itself.

The pre-processing part sorts the lines in the plane into a series of transitions. All start-points and end-points of lines which occur at the same transition point along the sweep axis are inserted into a list representing all the events which happen at this transition. The event list of each transition is itself ordered according to the order of the front. For example the lines in the XY plane for a horizontal sweep would be sorted into transitions along the X-axis from left to right. Each transition would contain a list of events which would be sorted according to the same vertical order relation as the front.

In order to perform the operation which inserts new lines into the front, the pre-processing phase must imbed additional information into the list of transitions. When a transition includes a vertex which is the start of one or more lines, a list must be created containing the descriptors of those lines.

After the pre-processing part, the front list is initialised and then the plane-sweep is performed by successively reading from the list of transitions. The events in each

transition are used to maintain the consistency of the front structure by removing, adding and swapping the positions of line items in the front. Every time a change occurs in the order of the front, an intersection test is performed on those lines which find themselves newly adjacent to each other. If any intersection points are calculated which lie on a transition which has not been encountered yet, then those intersection points are added to the event lists of those transitions.

The data structure representing the front is initialised. The front structure is an ordered list of line-segment descriptors. The order is based on the ranking of the relative positions of lines along a given dimension, which in this application could be parallel to either the X, Y or Z axis depending upon which view is being searched. Two sentinel line descriptors are inserted into the front in the initialisation. These line descriptors describe two reference lines which are at the two extreme points of the front. All other lines fit between these reference lines. This simplifies the algorithms needed to maintain the front: they never have to search past the top or the bottom of the list of line descriptors.

The sweep progresses by reading transitions from the sorted list prepared during the pre-processing phase. When a transition is read, the front is updated to reflect the change in line order that occurs at that transition. The updates required are a combination of DELETE, INTERSECT, and INSERT operations.

Delete.

The front is searched from bottom to top for any line-segments which have their end vertices at the transition point. Any such lines are removed from the front.

Intersect.

The front is searched from bottom to top for any sets of line-segments which intersect within the transition. The positions in the front of any lines in such a set are reversed. The top-most line of the set is swapped with the bottom-most line of the set, the second top-most with the second bottom-most, and so on.

Insert.

If the transition entry contains a list of line-segments which start at the transition, these lines are inserted into the front in positions relative to their ranking among the existing entries in the front. Any of these lines which have the same ranking because they have the same start point, are placed in the front relative to their order at some marginal advance in the sweep position.

Applying the plane sweep to matching lines to targets.

A property of the plane sweep is that the relative ordering of lines is established at all points along the sweep direction. This is useful for the matching procedure because it enables decisions to be made as to which lines are closest to a given target line at various points along the axis of the sweep. This enables us to find lines nearest to the target line, a useful ability in an application where practical drawing data may have been obtained from noisy and distorted souriven target scanners, and so the correspondences between views may be badly registered.

The simplest way of incorporating a target seeking mechanism into the plane sweep algorithm is to have this mechanism separate from the plane sweep mechanism. After every transition, a target checking procedure can be called to compare the positions of lines in the front against the set of targets.

Targets are held in a list with the same ordering relation as the front structure. The targets are prepared once only, see 4.5.3, and stay constant throughout the progression of the plane sweep. After each transition, the front and the list of targets are compared to each other. For each target in the list of targets, those lines in the front are found which lie closest on both sides of the target, and these lines are appended to a list detailing the nearest neighbouring lines on each side of the target. Any line which actually lies on the target is considered, arbitrarily and for convenience only, to lie on one side of the target and is added to the appropriate list of neighbouring lines. In the event that a number of lines intersect at the same point neighbouring the target, all these lines are added to the list.

4.5.5 OVERLAPS.

For different types of object we will be looking for different types of correspondences between views. In the case of uniform thickness objects, we have been given a cross-section, and we are looking for rectangles in the other views. More specifically, we are looking for rectangles orthogonal to the cross-section in the two views which are orthogonal to the view containing the cross-section. This means that in this case we are looking for either horizontal or vertical lines, depending upon which pairs of views are being examined. The correspondences are expressed in the following table.

Cross-section	corresponding view	lines sought
XY	XZ	vertical
XY	YZ	horizontal
YZ	XY	horizontal
YZ	XZ	horizontal
XZ	XY	vertical
XZ	YZ	vertical

The neighbours lists for each target contains all the lines which lie close to or congruent with the target line. To determine candidate rectangles within these lists, we look only at those neighbours which run parallel to the targets, these being either vertical or horizontal lines. From this set of parallel lines, we seek to determine and report any segments of the axis which lies parallel to the targets for which there exists a line-segment in each target: we look for the overlaps where every target is close to a neighbouring parallel line. The start and end points of such overlaps are recorded.

4.6 Constructing solids.

The length of the overlaps is the final information required to construct solid components from a cross-section. The components are assumed to be uniform thickness objects, which have the same cross-section all along their axes. To construct a solid from a cross-section in the XY view, all that is needed is to find the Z co-ordinates of the start and the end of the component. These Z co-ordinates are the start and the end points of the sides of the rectangles corresponding to the cross-section in either the YZ or the XZ view. These Z co-

ordinates would also happen to be the start and end points of the overlap determined by the plane-sweep. If more than one overlap, and hence rectangle, were to be found, then more than one solid component would be constructed.

The three-dimensional components are represented by a circular list of vertices which describe the hull of the cross-section, a flag to indicate the plane in which the cross-section lies, and the start and end points of the length of the axis of the component. The axis of the component lies perpendicular to the plane of the cross-section. This representation is compact and requires the application of a simple algorithm to construct a set of three-dimensional surfaces which can be rotated and rendered by conventional graphics algorithms [67,72,82].

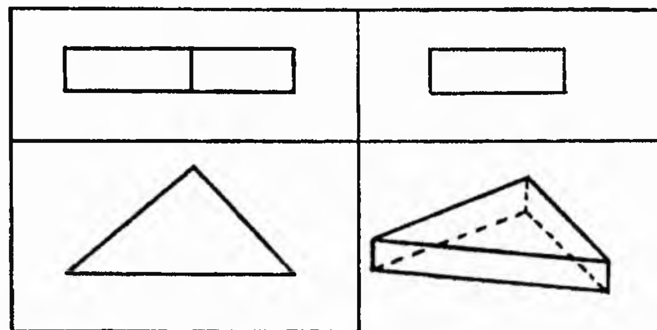
The surface construction algorithm first elaborates the two section ends of the component. These two ends are constructed by copying the cross-section: first using the start of the axis as the third dimension of each vertex; second using the end of the axis as the third dimension. Then the algorithm travels around the hull of the cross-section constructing rectangular three-dimensional surfaces between adjacent pairs of vertices.

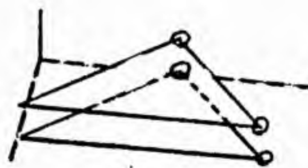
The rectangles are described by loops of four three-dimensional vertices. Two dimensions of each vertex are copied from one of the pair of vertices, and the third dimension is derived from one of the ends of the axis (Fig. 4.15).

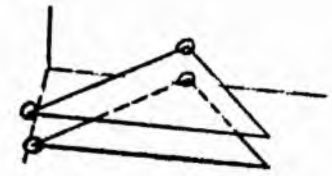
These vertices are constructed in order:-

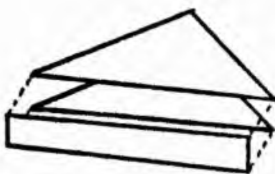
	Two-dimensions from -	Third dimension from -
i)	first vertex	start of the axis
ii)	first vertex	end of the axis
iii)	second vertex	end of the axis
iv)	second vertex	start of the axis

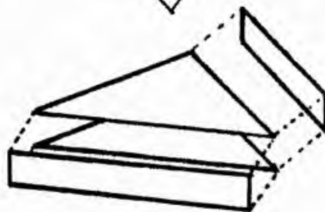
From this explicit description of the component in terms of planar three-dimensional surfaces, any other convenient representation can be derived for input to a solid modeller.











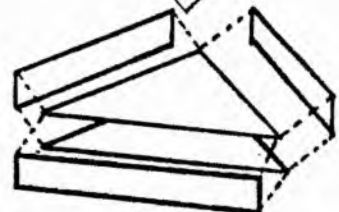


Figure 4.15

4.7 Some examples of reconstruction.

In the next four pages, two drawings are shown which have been reconstructed using the reconstruction program. The drawings were prepared using a rudimentary computer-aided drawing system which constructed unordered files of vectors describing the drawing.

These vector files were submitted to the Region Analysis program and then reconstructed using the program described in this chapter.

Figures 4.16 to 4.19 show a brace constructed from planar surfaces. Figure 4.16 shows the third angle projection of the brace, and figure 4.17 shows the reconstructed solid with hidden lines removed. Figure 4.18 agains shows the solid above figure 4.19, which shows an exploded view of the uniform thickness components from which it is constructed.

Figures 4.20 to 4.23 show a 'Guide Bracket' - taken from page 54 of Hart [42] - composed of planar surfaces. Figure 4.20 show the First angle projection; Figures 4.21 and 4.22 are identical, showing the solid; Figure 4.23 shows the uniform thickness objects from which the object is composed.

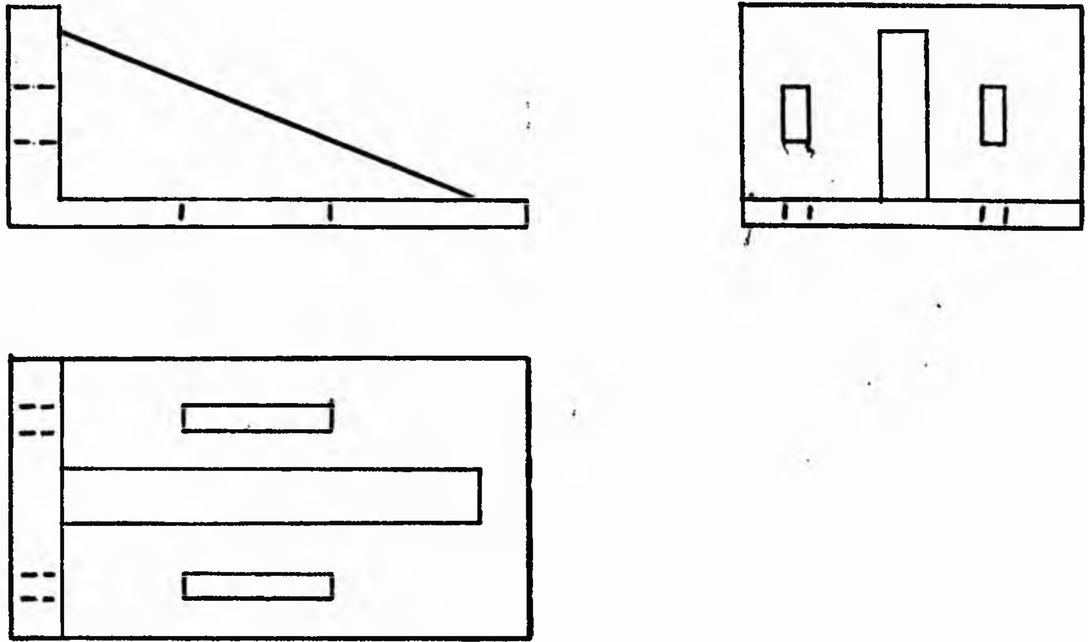


Figure 4.16

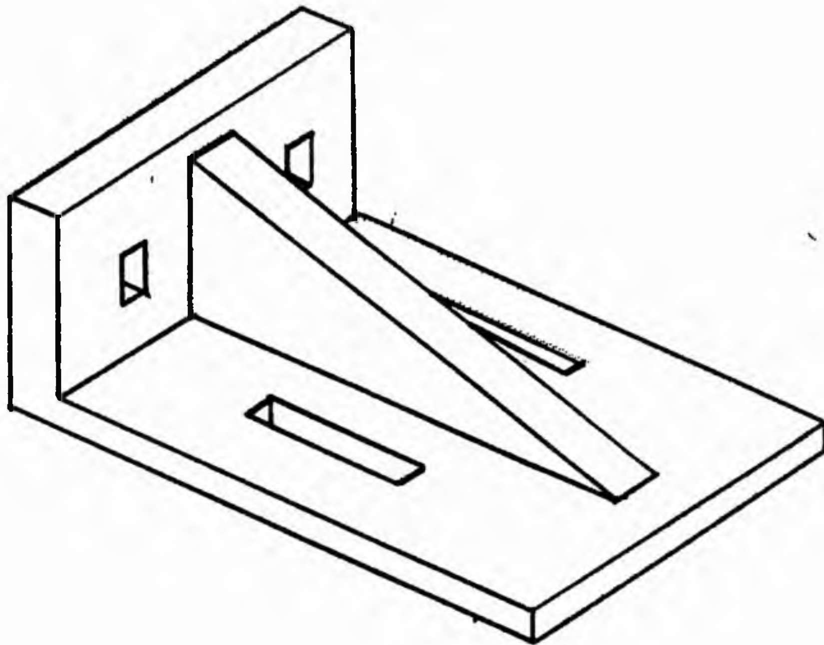


Figure 4.17

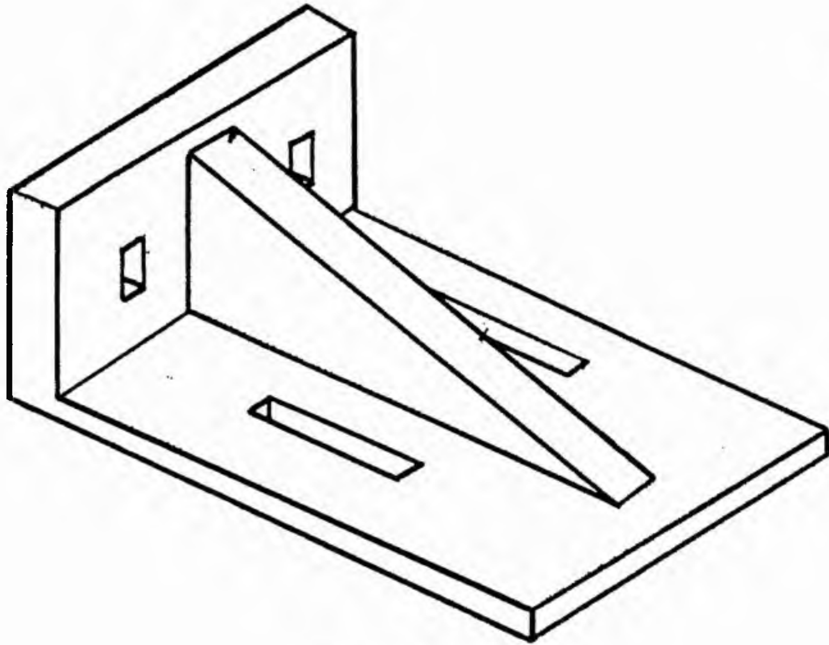


Figure 4.18

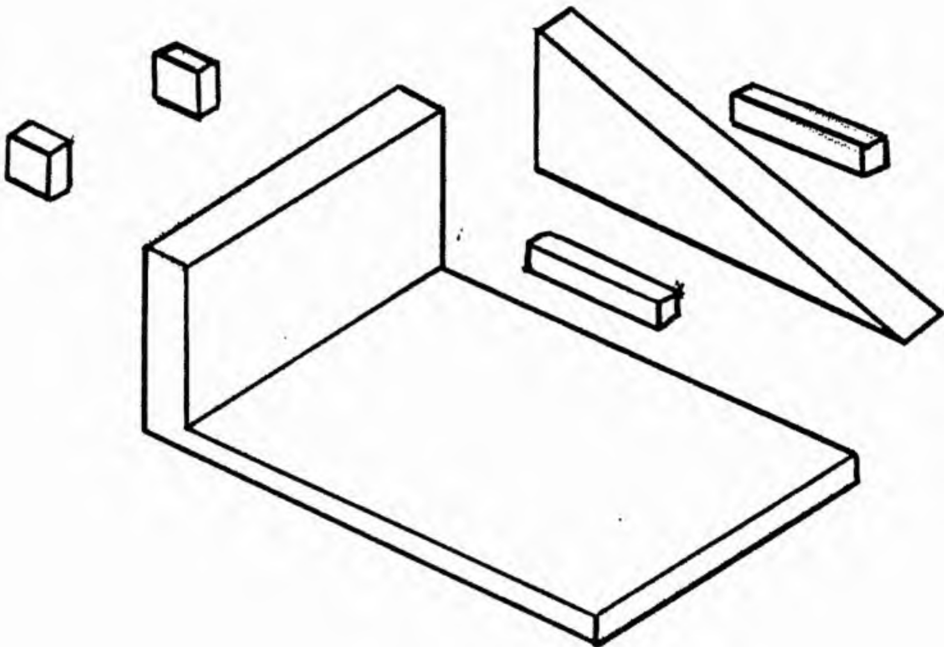


Figure 4.19

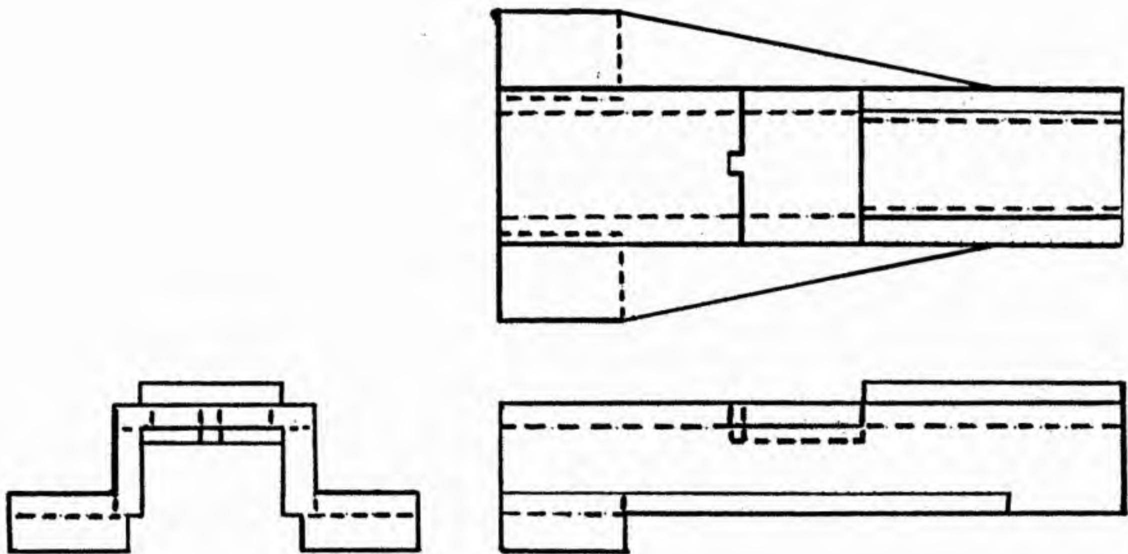


Figure 4.20

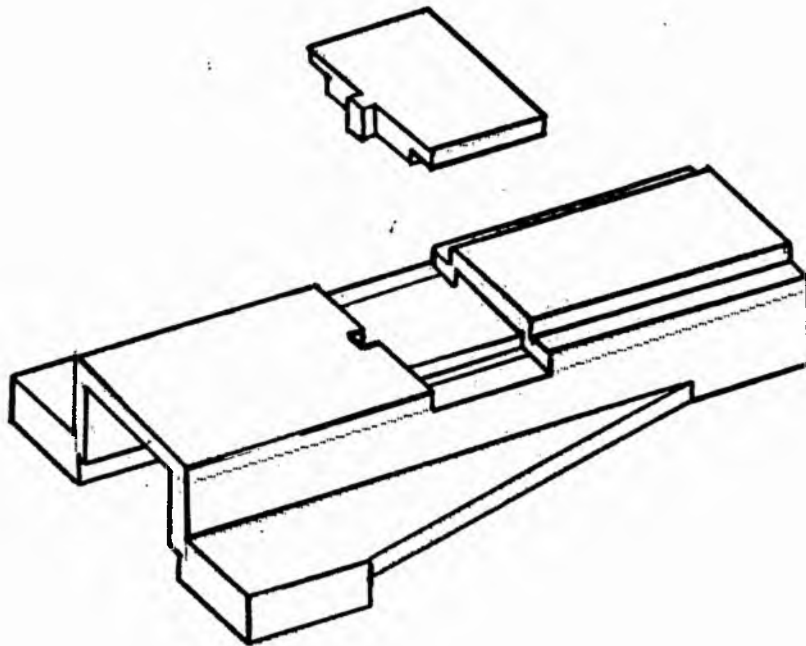


Figure 4.21

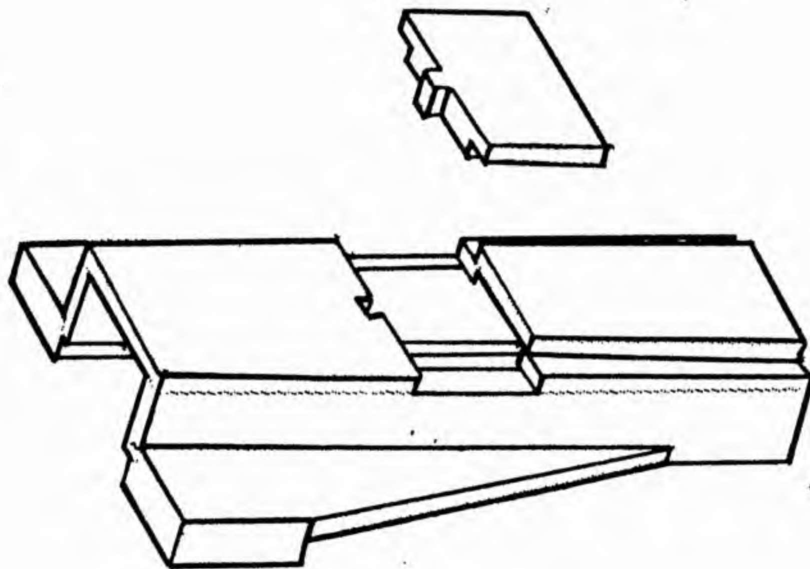


Figure 4.22

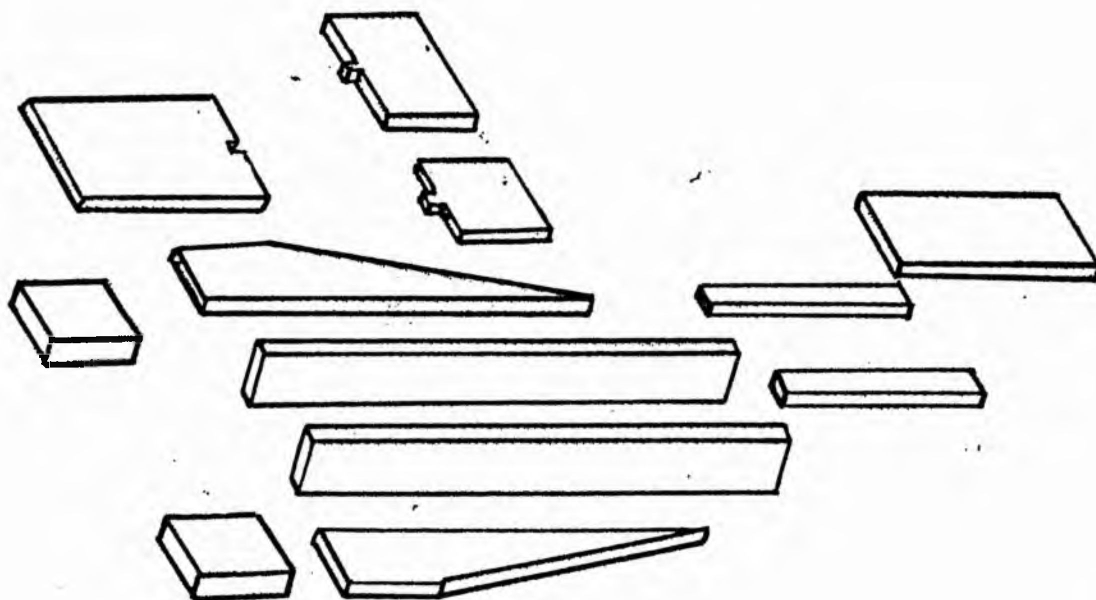


Figure 4.23

CHAPTER 5.

5. Conclusion.

This thesis has presented an engineering drawing interpretation system capable of producing three-dimensional representations of some objects from their two-dimensional orthogonal projections. This chapter makes an evaluation of the system and identifies some areas in which efforts for future development should be made, both specific to the components of this system and to the field of interpreting engineering drawings in general.

The system consists of two major components: a Region Analysis program which structures drawing data into a format susceptible for interpretation; a Reconstruction program which matches the projections of objects found in each view and combines the information contained in these projections to construct a three-dimensional representation. An evaluation of each component is made and areas for future enhancements identified.

5.1 An evaluation of the Region Analysis program.

An important aspect of drawing interpretation is that of ordering the drawing data to feed subsequent analysis and interpretation procedures. Drawing data is two-dimensional and so the ordering property must reflect the two dimensional arrangement of the drawing features. The data structures constructed by the Region Analysis program reflect the two dimensional arrangement. An unordered collection of line-segments is translated by the program into a description of the drawing in terms of the closed loops of white space - the Regions or partitions - which are caused by the two-dimensional arrangement of lines. This description is quite a rich one:- given any Region of the drawing, all the Regions which are entirely contained within that Region can be enumerated; given any Region, all the Regions which are directly adjacent to that Region can be identified; given any vertex, all the Regions meeting at that vertex can be found.

This description of the drawing is built from Edges. Edges, or more precisely Sides of Edges, are linked together to describe Regions. This is a significant improvement on the algorithm upon which the Region Analysis program was based, that presented by Nievergelt and Preparata [68] which constructs descriptions of Regions in terms of vertices. The vertex based description is adequate for describing the shape of the Regions but

presents difficulties when attempting to represent the adjacencies of Regions. By constructing the description from linked Sides of Edges, the adjacency relationships are constructed as a side-effect. Given any Edge, the Regions on each side of the Edge are known. Given any Region, all the Edges of that Region are known. Therefore, given any Region, all the Regions directly adjacent to that Region are known. Examining the edges of these directly adjacent Regions allows the indirectly adjacent Regions to be identified.

The largest development of the Region Analysis program from the algorithm of Nievergelt and Preparata is that of constructing the Containment Hierarchy during the progress of the Plane-sweep. By employing an analysis of the connection of Edges which is localised to the junctions of Edges - effectively guessing the role of the Edge purely by looking at how it connects to its immediate neighbours at its start-point and end-point - a global description of the drawing is constructed which links together all the separate adjacency graphs which might exist in the drawing. For example, one adjacency graph might be entirely contained within a single Region of another adjacency graph. The contained graph has no connecting edge to the containing graph - if such a connection existed the two separate graphs would merge into one. The Containment Hierarchy describes how Regions contain other Regions.

5.2 Areas for further development of the Region Analysis program.

There are two areas in which further development could be made from the Region Analysis program.

The first is to extend the range of line types from which a description can be constructed. The most obvious extension is to include curved line-segments. In terms of representing curved segments, very few amendments are required to the actual data structures used. Edges would require no amendment; currently the description of an Edge only includes its vertices and the links on each side of the Edge to other Edges in Region lists. There is no need to alter this description because no use is made of the type of the line from which the Edge is constructed. Line items would need amending:- a flag indicating the type of the line is needed, as are some parameters describing the line-segment. Currently with only straight line-segments, the parameters are intercept and slope. With a circular arc, the

parameters would be centre-point and radius. Elliptical arcs might describe the minor and major sub-axes.

The new line list formats are illustrated below.

General Format of line items:-

```
[line-id, line type, [parameters],[Edge list] ]
```

Format of straight line segments:-

```
[line-id, "straight", [intercept, slope], [Edge list] ].
```

Format of Circular Arc segments:-

```
[line-id,"circular arc",[radius,centre-point],[Edge list]]
```

Format of Elliptical Arc segments:-

```
[line-id,"elliptical arc",[sub-axis vertices],[Edge list]]
```

Alterations to the Region Analysis program would involve minor local changes to the intersection detection procedures. New procedures would need to be built to calculate the intersection points between two arcs, and between arcs and straight lines. These tests would have to be performed more often. Currently an Edge is never tested for intersection with an Edge which it touches at its start-point - straight lines can only intersect once so any test would be redundant. However arcs can intersect twice, and arcs and straight lines can also intersect twice - therefore whenever an arc finds itself with a new neighbour, an intersection test must be performed - even if it touches that neighbour.

The second area in which improvement in the Region Analysis program can be made is in the quality of the description of drawings produced by the program. Currently the description is at one level the vague but useful topological description identifying the Containment and Adjacency relationships between Regions. At the other level the description is an explicit enumeration of the vertices of edges, a description which must be used when trying to identify the positional relationships between features in the separate

views of an engineering drawing. Using the vertex description is unpleasant because of the sheer size of the data that can be involved in describing a drawing.

An improvement upon this situation might be to incorporate positional relationships into the topological description of the drawing. Some sort of network might be constructed which would represent more explicitly the two-dimensional relationships between Regions enabling the correspondences between Regions in separate views to be identified without having to search through masses of vertex data.

5.3 An evaluation of the Reconstruction program.

This project has explored some of the issues involved in building an engineering drawing interpreter. In particular the approach to Engineering drawing reconstruction followed by Aldefeld [2,3,4] has been chosen as a basis for further development work. The Reconstruction program described in chapter 4 follows Aldefeld's technique in its approach.

The Reconstruction program can reconstruct collections of plane-faced uniform-thickness solids from the three-view engineering projections of those collections. These solids are reconstructed one by one. The cross-section of a solid is identified in one view, and the outlines corresponding to that same solid, viewed from different lines of sight, are identified in the other views of the drawing. The information obtained from the shape of the cross-section and its supporting outlines are used to construct a three-dimensional representation of the solid.

In applying the Region Analysis program as a pre-processor to obtain a description of the topology of a drawing, this system is capable of applying the reconstruction algorithm to a wider range of input data sources than the system described by Aldefeld. Aldefeld's system was limited to accepting the structured drawing data obtained from a specific computer aided draughting system. The Region Analysis program applied here can structure drawing data from any source of vectors into a format suitable for reconstruction.

5.4 Areas for future development of the reconstruction program.

There are two major areas in which the system could be developed further. First, the reconstruction phase could be made more automatic. Second, the range of object types with which it can cope could be extended. Increasing the level of automation of the reconstruction phase could be addressed by developing an algorithm to select candidate cross-sections from the drawing. The system already employs an algorithm which automatically finds the matching rectangles for a given cross-section. The new algorithm to select candidate cross-sections would have to verify its selections.

The first most obvious verification test is to reject all cross-section candidates for which no matching rectangles can be found. Having passed this test, each cross-section candidate would yield a solid, called a candidate solid.

Another verification test would have to analyse the interactions of the candidate solids. The whole set of configurations which can be produced from the set of candidate solids would have to analysed. Those configurations in which two or more solids occupy the same space, or those configurations in which impossible objects were produced would have to be eliminated. Configurations would also have to produce projections which match against the original drawing, particularly with regard to the appearance of hidden detail. This second verification stage is an object-based verification, requiring some expert knowledge of the world of solids. This could be further improved by increasing the expert knowledge into the field of realisable engineering shapes. The verifier could reject objects which were unstable or which produced objects composed of solids which were improbably connected, for example by balancing on a corner. Further analysis could determine the 'manufacturability' of the final object - although this may cause rejection of drawings whose intention were to be bad designs.

Extending the range of objects with which the system can cope requires two separate lines of attack. The first is to complete the implementation of the system. The system has been designed to cope with prismatic objects. Theoretically, this should include shapes with a cross-section of arbitrary complexity composed from any type of line-segment. The implementation is restricted to straight line-segments. Some further work could easily extend the range of the topology analyser to work with curved line-segments. Once these

had been incorporated into the data structures used by the interpreter, the interpretation phase would work much the same as it currently does, with a few amendments to determine where any horizon lines might appear in matching rectangles before the search could begin. A more sophisticated display mechanism would also be required to display objects incorporating curved surfaces, such mechanisms being readily available as parts of the solid modelling CAD systems.

Realising a full implementation of an interpreter for uniform thickness objects would extend the capability of the system to include many realistic engineering artifacts. Aldefeld suggests extending the range of types of objects to include those with rotational symmetry. This would lead a way into interpreting many drawings of those rotationally symmetric components which are usually drawn as a cross-section with another view giving concentric rings. Another range of objects under investigation is the type of deformed uniformed thickness objects - a uniform thickness object which can be arbitrarily sliced[21]. These provide a more compact representation than by extracting all the derived sub-components.

5.5 Areas for future work on Engineering Drawing interpretation.

The system upon which the work presented in thesis was based [2,3,4] was not originally designed to interpret engineering drawings read from pieces of paper, but to interpret perfect images working on the data structures provided by a Cad system with the intention of providing a new man-machine interface to that system. The work presented in this thesis has aimed at extending this approach to engineering drawing interpretation to lower level image descriptions. The Region Analysis program acts as an intermediary, obtaining the topological information needed to interpret a drawing from unordered vector data.

The central issue remaining in providing a practically useful engineering drawing interpreter is the reduction of the dependence of such interpreters upon perfect image data. This system, like all other attempts at engineering drawing interpretation, is dependent upon three things:-

a) All line-segments in the drawing must be present.

b) All line-segments must be complete and correctly positioned.

c) No extraneous line-segments must be present.

The most immediate application of engineering drawing interpreters is to provide a paper interface to Computer Aided Draughting systems. Unfortunately Engineering drawing data derived from image scanners is usually incomplete, inaccurate and noisy.

Overcoming the problems of dealing with scanned images requires a degree of guess-work to be applied by the interpretation system. It must try to guess where line-segments are missing - these can be lost in the scanning process or may have been neglected by the draughtsman because of error or to enhance clarity at the expense of completeness. It must guess which line-segments have been accidentally truncated or extended and remedy them. It must try to guess which line-segments are extraneous - line-segments added by the mis-interpretation of creases or marks on the paper.

No solutions to these problems are readily forthcoming. In fact the approach adopted by this project was chosen precisely because it was the one least affected by inaccuracies in the drawing data. Matching cross-sections in one view to rectangles of approximate size and approximate positions in other views are much less susceptible to registration errors than the other reconstruction approaches whose first stages include a three-dimensional vertex construction phase.

One approach to reducing the dependence upon accurate image data is to try to force fit some of the object primitives which might be encountered. In the matching algorithm, rectangles are used as evidence of uniform thickness primitives. A useful activity might be to read through the drawing and construct as many rectangles as possible from the set of line-segments. Almost any pair of lines could be used as scant evidence of the existence of a rectangle, assuming that the line-segments may be distorted or incorrectly positioned. This could produce a new description of the drawing in terms of rectangles, and some of the largest rectangles may describe the separate drawing views.

Returning to the drawing data, cross-section candidates could be selected and then matched against the description of the drawing in terms of rectangles to find a supporting

match. Obviously such an approach would be messy and computationally expensive. Several orders of magnitude of incorrect rectangles could be generated and similarly large numbers of incorrect solid components would be constructed as a result. The emphasis would then shift onto applying heuristics to narrow down the number of attempts with as little expenditure as possible.

Having obtained a superset of rectangles, the only remaining limitation is in the number of cross-sections which can be constructed. In order to alleviate the necessity of having all line-segments connected to each other, it is necessary to be able to construct cross-sections from collections of lines which do not form closed loops. All valid cross-sections will be contained in the set of partitions created by extending all line-segments. Only those which are bounded by absent line-segments will not necessarily be in there. Cross-sections can then be generated by combining sets of partitions which are adjacent across edges.

These approaches are only rough sketches of what might need to be done. Always an element of uncertainty is introduced by attempting to cope with uncertainty. It seems unlikely that any practical drawing interpreter can be developed for understanding real as opposed to ideal images until some method of codifying visual memory into generalised forms is developed.

Understanding engineering drawings as performed by humans most likely involves some memory feat. The drawing reminds the person of a previously encountered object or drawing. From the recalled likeness verification of the match to the drawing can be made. If differences between the two are limited to some localised details, the visualised image can be modified by incorporating new local features or by performing limited distortions. Larger differences may cause larger distortions to the visualised solution to be required, causing a sufficiently new instance to be constructed that it may be rememorised as an entirely new entity. Even larger differences between visualised solution and the drawing may cause a complete failure which can only be remedied by examining the represented artifact and so creating a new visual for future reference. It should be remembered that producing engineering drawings is a skill requiring some expertise on the part of the human practitioner. A draughtsman may be able to read and understand drawings which would hopelessly confound untrained laymen. A system capable of

understanding engineering drawings must be capable of incorporating the expertise of the draughtsman and so surpass the abilities of the untrained human. In order for the system to be of any commercial value, it must be able to work as effectively as a draughtsman and more cost-effectively. To this end, a more sensible approach might be to work down from the opposite direction to that employed currently.

A better objective might be to construct an 'artificial draughtsman' with an inbuilt ability to design solutions to engineering problems. This artificial draughtsman might require less initiative than is immediately considered. Engineering design, in common with most design tasks, is evolutionary in nature. The solution to one specific problem usually requires only a little mutation of a known solution to a previous problem. The more 'expert' the designer, the more experience in previous problems and solutions it has. The relative 'intelligence' of the designer might be measured by how quickly the satisfactory solution can be 'mutated' into existence. Then confronted by a drawing, it might read the title and consider known solutions to similar problems and be able to compare these to the image with which it is confronted.

The most important task in this case is how to represent this expertise. Databases of the geometry of solids can quite easily be constructed and indeed are commonly used in CAD systems of any merit. Unfortunately few of the considerations which must be applied by engineering designers in developing a design are encompassed by geometry. What must be considered is how a given function can be achieved. The engineering database must not only store geometries but must also store the roles of the separate features of each drawing. Brackets must be described in terms of how they are used, bearings in terms of the motions they allow, mechanisms in terms of what they achieve and some essence of how they achieve it. A pump must be described in terms of a chamber of whatever geometry into which is drawn fluid from a given inlet pipe with a valve, this performed by some piston which then forces the fluid through the outlet via another valve. Knowing that this is how a general pump works, the expert when looking at a drawing of a pump will know he is looking for a leak-proof chamber, an inlet with valve, an outlet with valve and a piston. Knowing the function of these components, his memory can quite happily accept the geometries in the drawing and compare them against the constraints needed to be satisfied for them to work, and against previous well-known geometries which satisfy

these constraints. If the drawing geometry is particularly ingenious or stupid, it may be remembered and appropriately placed in a priority queue of solutions, possibly arranged in order of cost, weight, size, and power, for future reference.

A system capable of understanding engineering drawings must encompass the abilities of a skilled designer. The common perception of computer systems is that they are worth buying because they perform their tasks more cost-effectively and in a more timely and a more precise manner than man, and these perceptions are true for many commercial applications of computing power which involve little more than performing numerical manipulations and simply indexing large collections of data. Unfortunately, attempts to mimic human expertise run into problems precisely because humans are very efficient at doing tasks which computers cannot do well, namely understanding images and recovering and learning from mistakes.

No-one should pay for a system which often makes mistakes in its interpretations, or which continually called for expensive expert human assistance to complete its interpretations. Little value can be derived from a system which can only understand very simple drawings because such drawings can be quickly transferred onto the machine by hand. In order for an automatic engineering drawing interpreter to be accepted as useful, it must display the capabilities of an expert draughtsman.

References.

[1]
Agin G.J, Binford T.O. (1963)
"Computer Description Of Curved Objects"
3rd International Conf. Artificial Intelligence, Stanford

[2]
Aldefield B. (1983)
"Automatic Reconstruction From 2D Geometric Part Descriptions"
IEEE Proc. Computer Vision And Pattern Recognition 1983, 66-72

[3]
Aldefeld B. (1983)
"On Automatic Recognition Of 3D Structures From 2D Representations"
Computer Aided Design, 15:2, 59-64

[4]
Aldefeld B., Richter H. (1985)
"Semi-Automatic Three-Dimensional Interpretation Of Line Drawings"
Comput. & Graphics, Vol 8 No 4, 371-380

[5]
Allen G. (1985)
"An Introduction To Solid Modelling"
Comput. & Graphics, Vol 8 No 4, 439-447

[6]
Altman N.G. (1978)
"Automatic Digitizing Of Engineering Drawings"
Proc. IEEE Electronic Show And Convention, Boston, May 1978,

[7]
Barrow H.G., Popplestone R.J. (1971)
"Relational Descriptions In Picture Processing"
Meltzer B., Michie D.(eds) Machine-Intelligence 6,Edinburgh Univ. Press, 377-396

[8]
Barnhill R.E. (1985)
"Surfaces In Computer-Aided Geometric Design: A Survey With Results"
Computer-Aided Geometric Design, Vol 2, North-Holland, 1-17

[9]
Basu S., Fu K.S. (1987)
"Image-Segmentation By Syntactic Method"
Pattern Recognition Vol 20,No 1, 33-44

[10]
Bentley J.L., Ottmann T.A. (1979)
"Algorithms For Reporting And Counting Geometric Intersections"
IEEE Trans. Computers, Vol C-28 No 9, 643-647

[11]
Bentley J.L., wood D. (1980)
"An Optimal Worst Case Algorithm For Reporting Intersections Of Triangles"
IEEE Trans. Computers, C-29(7), 571-577

[12]
Bixler J.P., Sanford J.P. (1985)
"A Technique For Encoding Lines And Regions In Engineering Drawings"
Pattern Recognition, Vol 18 No 5, 367,377

[13]
Bocquet J., Tichkiewitch S. (1983)
"An 'Expert System' For Identification Of Mechanical Drawings"
Advances In CAD/CAM ed T.R. Ellis & Semenkov, North-Holland, 453-461

[14]
Brady M., Ponce J., Yuille A., Asada H. (1985)
"Describing Surfaces"
Computer Vision Graphics And Image Proc.,Vol 32, 1-28

[15]
Brown C.M. (1981)
"Some Mathematical And Representational Aspects Of Solid Modelling"
IEEE Trans. Pattern Recognition And Mach. Intell., 3:4, 444-453

[16]
Bunke H. (1982)
"Automatic Interpretation Of Lines And Text In Circuit Diagrams"
Kittler Fu Pau (eds),D. Reidel (pubs),Pattern Recognition Theory And Applic., 297-310

[17]
Carlbom I., Paciorek J. (1978)
"Planar Geometric Projections And Viewing Transformations"
Computing Surveys, V10 N 4, 465-501

[18]
Cavagna C., Cugini U. (1977)
"Data-Structure For The Description And Handling Of Engineering Drawings"
Computer Aided Design, Vol 9 No 1, 17-22

[19]
Chazelle B. (1984)
"Intersecting Is Easier Than Sorting"
ACM Proc. 16th Ann. Symp. Theory Of Computing, 125-134

[20]
Cheng K.Y., Cheng T.C., Huang S.C. (1980)
"An Interactive Design System For Engineering Drawings"
Journal Of the Chinese Institute Of Engineers, Vol 3 No 1, 51-59

[21]
Cheng Z., Perng D. (1988)
"Automatic Reconstruction Of 3D Solid Objects From 2D Orthographic Views"
Pattern Recognition, v21, N5, 439-449

[22]
Clement T.P. (1981)
"The Extraction Of Line-Structured Data From Engineering Drawings"
Pattern Recognition, Vol 14 No 1, 43-52

[23]
Cugini U., Mussio P., Cavagna C., Meravalia A. (1978)
"On An Image Generation System"
IFIP,Artificial Intelligence And Pattern Recognition In CAD,North-Holland Pub., 429-455

[24]
Cugini U., Ferri G., Mussio P., Protti M. (1985)
"Pattern Directed Restoration And Vectorization Of Digitised Engineering Drawings"
Comput. & Graphics, Vol 8 No 4, 337-350

[25]
Draper S.W. (1981)
"The Use Of Gradient And Dual Space In Line-Drawing Interpretation"
Artificial Intelligence, Vol 17, 461-508

[26]
Edelsbrunner H., Overmars M.H., Seidel R. (1984)
"Some Methods Of Computational Geometry Applied To Computer Graphics"
Computer Vision And Image Processing, V28, 92-108

[27]
Edelsbrunner H., Guibas L.J., Stolfi J. (1986)
"Optimal Point Location In A Monotone Subdivision"
SIAM Journal Comput., 15(2), 317-340

[28]
Er M.C. (1983)
"Interpreting Engineering Drawings"
Proc 6th Australian Computer Science Conf 10-12 Feb 1983, 65-70

[29]
Fu K.S. (1974)
"Syntactic Methods In Pattern Recognition"
Academic Press, New York

[30]
Fulford M.C. (1981)
"The Fastrak Automatic Digitising System"
Pattern Recognition, Vol 14 No 1, 65-74

[31]
Gasson Peter C. (1984)
"Geometry Of Spacial Forms "
Ellis Horwood

[32]
Gibson L., Lucas D. (1982)
"Vectorization Of Raster Images Using Hierarchical Methods"
Computer Graphics And Image Processing, 20, 82-89

[33]
Gotchev G.V. (1984)
"Computer Linguistic Analysis Of Line Drawings"
Pattern Recognition Vol 17,No 4, 433-440

[34]
Grabowski H., Eigner M. (1982)
"A Data Model For A Design Data Base"
IFIP File Structures And Data Bases For CAD (North-Holland),117-134

[35]
Guilbas L.J., Stolfi J. (1983)
"Primitives For the Manipulation Of General Subdivisions And the Computation Of Voronoi Diagrams"
ACM Proc. 15th Sigact Symp, April 1983, 221-234

[36]
Guzman A. (1968)
"Computer Recognition Of Three-Dimensional Objects In A Visual Scene"
Tech. Rep. MAC-TR-59, Artificial Intelligence Lab., MIT

[37]
Guzman A. (1968)
"Decomposition Of A Visual Scene Into Three-Dimensional Bodies"
Proc. AFIPS Fall Joint Conf., 291-304

[38]
Guzman A. (1971)
"Analysis Of Curved Line Drawings Using Context And Global Information"
Meltzer B., Michie D.(eds) Machine-Intelligence 6,Edinburgh Univ. Press, 325-375

[39]
Haralick R.M., Shapiro L.G. (1979)
"The Consistent Labelling Problem: Part 1"
IEEE Trans. Pattern Anal. And Machine Intelligence, Vol PAMI-1 No 2, 173-184

[40]
Haralick R.M., Elliot G.L. (1980)
"Increasing Tree Search Efficiency For Constraint Satisfaction Problems"
Artificial Intelligence, No 14, 263-313

[41]
Haralick R.M., Queeney D. (1982)
"Understanding Engineering Drawings"
Computer Graphics And Image Processing, Vol 20, 244-258

[42]
Hart K.R. (1970)
"Engineering Drawings With Problems And Solutions"
English Universities Press

[43]
Hartzband D.J., Maryanski F.J. (1985)
"Enhancing Knowledge Representation In Engineering Databases"
IEEE Computer, Sept. 1985, 39-46

[44]
Hosaka M., Kimura F. (1982)
"Using Handwriting Action To Construct Models Of Engineering Objects"
Computer, Nov 1982, 35-47

[45]
Huffman D.A. (1977)
"A Duality Concept For the Analysis Of Polyhedral Scenes"
Elcock E.W. Michie D.(eds) Machine Intelligence 8, Ellis Horwood, 475-492

[46]
Huffman D.A. (1978)
"Surface Curvature And Applications Of The Dual Representation"
Hanson A.R. Riseman E.M.(eds) Computer Vision Systems, Academic Press, 213-222

[47]
Idesawa M. (1973)
"ASystem To Generate ASolid Figure From three-view"
Bulletin Jsme 16:92, 216-225

[48]
Idesawa M., Soma T., Goto E.,Shibata S. (1975)
"Automatic Input Of Line Drawing And Generation Of Solid Figure From three-view Data"
Proc. International Computer Symposium 1975, Vol II, 304-31

[49]
Jansen H., Krause F. (1985)
"Interpretation Of Freehand Drawings For Mechanical Design Purposes"
Comput. & Graphics, Vol 8 No 4, 351-369

[50]
Kanade T. (1980)
"A Theory Of Origami World"
Artificial Intelligence, Vol 13, 279-311

[51]
Kanade T. (1981)
"Recovery Of three-Dimensional Shapes Of An Object From A Single view"
Artificial Intelligence, Vol 17, 409-460

[52]
Kimura S., Agui T., Hoshina N. (1983)
"An Algorithm For Extracting A Solid Object From Three Views"
Trans. IECE Japan, V.E66, N1, 51-53

[53]
Korn M.R., Dyer C.R. (1987)
"3-D Multiview Object Representations For Model-Based Object Recognition"
Pattern Recognition Vol 20,No 1, 91-103

[54]
Lafue G. (1978)
"A Theorem Prover For Recognising 2-D Representations Of 3-D Objects"
Latombe J.C.(ed) Artificial Intell. In CAD, North-Holland, 391-401

[55]
Lequette R. (1988)
"Automatic Construction Of Curvilinear Solids From Wireframe Views"
Computer-Aided Design, v20, N4, 171-180

[56]
Liardet M., Holmes C., Rosenthal D. (1978)
"Input To Cad Systems: Two Practical Examples."
Proc IFIP TC-5 Working Conf. Art. Intell. Cad, Grenoble France 17-19 March 1978, 403-414

[57]
Markowsky G., Wesley M.a. (1980)
"Fleshing Out Wire Frames"
Ibm Res. Develop., 24(5), 582-597

[58]
Masini G., Mohr R. (1983)
"Mirabelle, A System For Structural Analysis Of Drawings"
Pattern Recognition Vol 16,No 4, 363-372

[59]
Maxwell P.C. (1972)
"Perception And Description Of Line Drawings By Computer"
Computer Graphics And Image Processing, v1, 31-46

[60]
Meyer B. (1988)
"Object-Oriented Software Construction"
Prentice Hall

[61]
Mohr R., Masini G. (1978)
"Drawing Analysis And Computer Aided Design"
Artificial Intelligence And Pattern Rec. In CAD (ed Latombe), North-Holland, 477-495

[62]
Mohr R. (1986)
"Precompilation Of Syntactical Descriptions And Knowledge Directed Analysis Of Patterns"
Pattern Recognition Vol 19,No 4, 255,266

[63]
Mulgaonkar P.G, Shapiro L.G., Haralick R.M. (1984)
"Matching 'Sticks, Plates And Blobs' Objects Using Geometric And Relational Constraints"
Image And Vision Computing, Vol 2 No 2, 85-98

[64]
Murase H. Wakahara T. (1986)
"Online Hand-Sketched Figure Recognition"
Pattern Recognition Vol 19,No 2, 147-160

[65]
Nadler M. (1984)
"Survey: Document Segmentation And Coding Techniques"
Computer Vision Graphics And Image Processing, v28, 240-262

[66]
Neveu C.F., Dyer C.R., Chin R.T. (1986)
"Two-Dimensional Object Recognition Using Multi-Resolution Models"
Computer Vision Graphics And Image Processing, 34, 52-65

[67]
Newman W.M., Sproull R.F (1981)
"Principles Of Interactive Computer Graphics "
McGraw Hill

[68]
Nievergelt J., Preparata F.P. (1982)
"Plane-Sweep Algorithms For Intersecting Geometric Figures"
Comm. ACM, Vol 25 No 10, 739-747

[69]
Parui S.K., Majumder D.D.(1983)
"Symmetry Analysis By Computer"
Pattern Recognition Vol 16,No 3, 63-67

[70]
Pavlidis T. (1977)
"Structural Pattern Recognition"
Springer

[71]
Pavlidis T., Cherry L.L. (1982)
"Vector And Arc Encoding Of Graphics And Text"
IEEE Ch1801-0/82/0000 0610, 610-613

[72]
Pavlidis T. (1982)
"Algorithms For Graphics And Image Processing"
Rockville,md.: Computer Science Press

[73]
Pavel M. (1983)
"'Shape Theory' And Pattern Recognition"
Pattern Recognition Vol 16,No 3, 349-356

[74]
Pferd W., Ramachandran K. (1978)
"Computer Aided Automatic Digitizing Of Engineering Drawings"
IEEE Ch1338-3/78/0000-0630, 630-635

[75]
Pong T.C., Shapiro L.G., Haralick R.M.(1985)
"Shape Estimation From Topographic Primal Sketch"
Pattern Recognition Vol 18,No 5, 333-347

[76]
Preiss K. (1980)
"Constructing The 3-D Representation Of A Plane-Faced Object From A Digitised Engineering Drawing"
Cad 80 4th International Conference On Comp. In Design Engineering, 257-265

[77]
Preiss K. (1981)
"Algorithms For Automatic Conversion Of A 3-View Drawing Of A Plane-Faced Part To The 3-D Representation"
Computers In Industry (North-Holland), Vol 2, 133-139

[78]
Preiss K., Kaplansky E. (1983)
"Solving CAD/CAM Problems By Heuristic Programming"
Computers In Mechanical Engineering, Sept 1983, 57-60

[79]
Preiss K. (1984)
"Constructing The Solid Representation From Engineering Projections"
Computers And Graphics, Vol 8 No 4, 381-389

[80]
Ramachandran K. (1980)
"Coding Method For Vector Representation Of Engineering Drawings"
Proc. IEEE, Vol 68 No 7, 813-817

[81]
Requicha A.A.G (1980)
"Representations For Rigid Solids: Theory, Methods And Systems"
Computer Surveys, 12:4, 437-464

[82]
Rogers D.F (1985)
"Procedural Elements For Computer Graphics"
McGraw Hill

[83]
Sakuri H., Gossard D.C. (1983)
"Solid Model Input Through Orthographic Views"
Computer Graphics, 17:3, 243-252

[84]
Sanker P.V. (1977)
"A Vertex Coding Scheme For Interpreting Ambiguous Trihedral Solids"
Computer Graphics And Image Processing, Vol 6,61-89

[85]
Sandewall E. (1978)
"A Survey Of Artificial Intelligence With Special Respect To Computer Aided Design"
Artificial Intell. And Pat. Rec. In CAD (ed Latombe), North-Holland, 19-34

[86]
Shapiro L.G., Moriarty J.D., Haralick R.M., Mulgaonkar P.G.(1984)
"Matching Three-Dimensional Objects Using A Relational Paradigm"
Pattern Recognition Vol 17,No 4, 385-405

[87]
Shirai Y. (1981)
"Use Of Models In Three-Dimensional Object Reconstruction"
IFIP Man-Machine Comms. In CAD/CAM (North Holland), 137-159

[88]
Shirai Y. (1982)
"Image Processing For Data Capture"
IEEE Computer, Nov 1982, 21-34

[89]
Shridhar M., Badreldin A. (1984)
"Handwritten Numeral Recognition By Tree Classification Methods"
Image And Vision Computing, Vol 2 No 3, 143-149

[90]
Sinha R.M.K. (1983)
"Plang - A Picture Language Schema For A Class Of Pictures"
Pattern Recognition Vol 16,No 4, 373-383

[91]
Sowa J.F. (1983)
"Conceptual Structures: Information Processing In Mind And Machine"
Addison-Wesley

[92]
Strat T.M. (1984)
"Spatial Reasoning From Line Drawings Of Polyhedra"
Proc. Workshop Computer Vision Representation And Control, 219-224

[93]
Stroustrup B. (1986)
"The C++ Programming Language"
Addison-Wesley

[94]
Sugihara K. (1978)
"Picture Language For Skeletal Polyhedra"
Computer Graphics And Image Processing, 8, 382-405

[95]
Sugihara K. (1979)
"Range-Data Analysis Guided By Junction Dictionary"
Artificial Intelligence, 12, 41-69

[96]
Sugihara K. (1982)
"Classification Of Impossible Objects"
Perception, Vol 11, 65-74

[97]
Sugihara K. (1982)
"Mathematical Structures Of Line Drawings Of Polyhedrons - Toward Man-Machine Communication By Means Of Line Drawings"
IEEE Trans. PAMI-4,458-469

[98]
Sugihara K. (1983)
"On Some Problems In The Design Of Plane Skeletal Structures"
SIAM Journal On Algebraic And Discrete Methods, Vol 4,355-362

[99]
Sugihara K. (1984)
"An Algebraic Approach To Shape-From-Image Problems"
Artificial Intelligence,Vol 23,59-95

[100]
Sugihara K. (1984)
"A Necessary And Sufficient Condition For A Picture To Represent A Polyhedral Scene"
IEEE Trans. PAMI-6, 578-586

[101]
Sugihara K. (1984)
"An Algebraic And Combinatorial Approach To The Analysis Of Line Drawings Of Polyhedrons"
Discrete Applied Mathematics,Vol 9, 77-104

[102]
Sugihara K. (1985)
"Interpretation Of Axonometric Projection Of A Polyhedron"
Comput. & Graphics, Vol 8 No 4, 391-400

[103]
Sugihara K. (1985)
"Detection Of Structural Inconsistency In Systems Of Equations With Degrees Of Freedom And Its Applications"
Discrete Applied Mathematics,Vol 10,297-312

[104]
Suk M., Oh S.J. (1986)
"Region Adjacency And Its Application To Object Detection"
Pattern Recognition Vol 19,No 2, 161-167

[105]
Sutherland L.E. (1963)
"SKETCHPAD: A Man-Machine System"
MIT Technical Report N296

[106]
Tudhope D.S., Oldfield J.V. (1983)
"A High-Level Recognizer For Schematic Diagrams"
IEEE Computer Graphics And Applic., May/June 1983, 33-40

[107]
Wahl F.M, Wong K.Y., Casey R.G. (1982)
"Block Segmentation And Text Extraction In Mixed Text/Image Documents"
Computer Graphics And Image Processing, 20, 375-390

[108]
Waltz D. (1972)
"Generating Semantic Descriptions From Drawings Of Scenes With Shadows"
Tech. Report AI-TR-271 MIT

[109]
Waltz D. (1975)
"Understanding Line Drawings Of Scenes With Shadows"
Winston P.H.(ed) The Psychology Of Computer Vision, McGraw-Hill, 19-91

[110]
Wallace A. (1987)
"An Informed Strategy For Matching Models To Images Of Fabricated Objects"
Pattern Recognition Vol 20,No 3, 349-363

[111]
Warman E.A. (1978)
"Computer Aided Design: An Intersection Of Ideas"
Artificial Intell. And Pat. Rec. In CAD (ed Latombe), North-Holland, 1-17

[112]
Weiler K. (1985)
"Edge-Based Data Structures For Solid Modeling In Curved-Surface Environments"
IEEE CG&A, Jan. 1985, 21-40

[113]
Weitek (1985)
"Weitek Solids Modeling Engine"
Comput. & Graphics, Vol 8 No 4, 437

[114]
Wesley M.a., Markowsky G. (1981)
"Fleshing Out Projections"
Ibm Res. Develop., 25(6), 934-954

[115]
Whitrow R.J., Stubbs J. (1978)
"Three-Dimensional Representation Of Anatomical Structures"
Proc. Dec Users Society (DECUS), Sept. 1978, 417-420

[116]
Wittek D. (1985)
"Solid Modelling And System Design"
Comput. & Graphics, Vol 8 No 4, 423-431

[117]
Wojcik Z.M. (1985)
"A Natural Approach In Image Processing And Pattern Recognition: Rotating Neighbourhood Technique, Self-Adapting Threshold, Segmentation And Shape Recognition"
Pattern Recognition Vol 18,No 5, 299-326

[118]
Woo T.C., Hammer J.M. (1977)
"Reconstruction Of three-Dimensional Designs From Orthographic Projections"
Proc 9th Cirp Conf. Cranfield Institute Tech., 247-255

[119]
Yoshiura H., Fujimura K., Kunii T.L. (1984)
"Top-Down Construction Of 3-D Mechanical Object Shapes From Engineering Drawings"
IEEE Computer, Dec 1984, 32-40

Appendices.

Appendix A.

Pseudo-code Outline of the Region Analysis program.

Version which builds Region_lists from Vertices.

This program is an extension of the algorithm described in [68]. This extension allows the algorithm to report the Regions around any scene composed from any arrangement of straight line-segments, with the significant exception of vertical line-segments which are dealt with in Appendix B.

Descriptions of Regions are constructed from linked lists of vertices. As they are being constructed, the lists are attached either above or below entries in the front. Tails of lists are always attached above a front entry, and heads of lists are always attached below a front entry. The lists are extended either by attaching new vertices to the head or the tail of the list, or by concatenating pairs of lists to form longer single lists. These operations are performed according to the context of how lines are found to connect to each other.

This appendix presents Pascal-like pseudo-code describing how these descriptions of Regions are constructed by a plane-sweep algorithm. The algorithm has four main procedures:

> **Advance_Front** which is the executive procedure controlling the progress of the program;
>
> **Remove**, the procedure which removes line-segments from the Front when their end-points are reached;
>
> **Permute**, which re-orders entries in the Front when they are involved in an intersection;
>
> and **Insert**, the procedure which places new entries in the Front when their Start_points are reached.

Before the pseudo-code is presented, some minor procedures used in the code are described.

Front operators.

Front_Successor(front-entry) - returns the next higher entry in the Front to the given front-entry.

Front_Predecessor(front-entry) - returns the next lower entry in the Front to the given front-entry.

Below(front-entry) - references the head of the Region-list attached below the given front-entry.

Above(front-entry) - references the tail of the Region-list attached above the given front-entry.

Region list operators.

Initialise_Region_Head (Side of front-entry) - this procedure generates a new list containing a single vertex corresponding to the start-vertex of the line-segment described by the given front-entry. A pointer to the head of the new list is stored in the specified side, Above or Below, of the given front entry.

Concatenate (tail of list) with (head of list) - joins the head and tail of the list(s) together. This either results in two separate lists being combined into one, or in a single list being turned into a closed loop of vertices by having its head attached to its tail.

Extend_Tail (Front-entry) To (new_vertex) - appends new-vertex onto the tail end of the list above the specified Front-entry.

Extend_Head (Front-entry) to (new_vertex) - appends the new_vertex to the head of the Region_list below the specified front-entry.

Attach (end of list) to (Side of front-entry) - attaches either the head or tail of a list to the specified side of the front-entry.

A1.1 Pseudo-code of the Advance Front procedure incorporating Region-list maintenance.

Notes:-

Procedure Remove returns three items (the two lists are parameters to the Insert and Permute procedures) :-

Above-list - (if it exists) the Region-list attached above the top-most line-segment entering the Junction;

Below-list - (if it exists) the Region-list attached below the lowest line-segment entering the Junction;

Left-side-exists - a boolean flag (true or false) indicating whether the Junction has a left-hand side.

Procedure Permute is only called if Remove has found any line-segments which touch the current Junction, regardless of whether it removed them all.

Procedure Permute returns a flag indicating whether it found any line-segments which pass through the Junction. This is used as part of the test as to whether the Junction has a right-hand side.

Lines-inserted is a flag which indicates whether the Insertions-list of the current Junction is empty.

The Concatenate procedure is only called when the Junction is a 'butt-end' with no right-hand side. This is performed after all the right-hand side processing for the current Junction has been performed.

The Initialise procedure is only called when the Junction is a 'butt-end' with no left-hand side, as indicated by Left-side-exists. This initialisation does not attach the new Region-list to any Front entry, but attaches the tail to Above-list and the head to Below-list. If a Junction exists but has no left-hand side, it must by implication have a right-hand side. Above-list and Below-list in this case are used to communicate the new region-list to the Insert procedure to enable it to extend these around the first line-segment to be inserted at that Junction.

```
Procedure Advance_Front;
begin
Read Transition from Transition_list;

While not( End-of Transition_list )
do begin
    Read Junction from Junction_list of Transition

    While not( End-of Junction_list)
    do  begin
        (Above-list,
        Below-list,
        Left-side-exists) <- Remove(Junction,Front);

        If(Left-side-exists)
        then Lines-permuted <- Permute(Junction,Front,
                                       Above-list,Below-list)
        else begin
             Lines_permuted  <- false;
             Initialise_Region_Head(Below_list);
             Attach tail(Below-list) to Above-list;
             end;

        Lines-inserted <- false;
        Read Line-segment from Insertion_list of Junction;

        If not  (End-of Insertions-list)
        then  Lines-inserted <- true;

        While not(End-of Insertions-list)
        do    begin
              Insert(Line-segment,Front,Above-list,Below-list);
              Read Line-segment from Insertion_list of Junction;
              end;

        If not(Lines-permuted or Lines-inserted)
        then Concatenate(Above-list,Below-list);

        Read Junction from Junction_list of Transition;
        end;

    Read Transition from Transition_list;
    end;

end;
```

Pseudo-code of the Advance Front procedure incorporating Region-list maintenance.

A1.2 Pseudo-code outline of the REMOVE procedure incorporating Region-list maintenance.

NOTES.

Three new variables are used in this version of Remove.

Left-side-exists is a boolean flag set to true if any line-segments are found which have the same height as the Junction.

`Below-list` holds the Region-list attached below the lowest line-segment found to enter the Junction. The lowest line-segment also happens to be the first line-segment found, if it exists.

Above-list holds the Region-list attached above the last line-segment found to enter the Junction. Once the search has ended, when the Current-entry lies above the current-Junction, Above-list happens to hold the Region-list attached above the highest line-segment found to enter the Junction.

The **RETURN** statement at the end of the procedure is merely a marker to indicate that the values of the parenthesised variables are returned to the calling procedure. How this is actually performed in an actual implementation depends on the language used.

```
Procedure Remove ( Junction, Front );
begin

Read Current-entry from the tail entry of the Front;

While (height of Current-entry < height of Junction)
do    Read Current_entry from next entry in Front;

If    (height of Current-entry = height of Junction)
then begin
      Left-side-exists <- true;
      Below-List  <- Below(Current-entry);
      Above-list  <- Above(Current_entry);
      If     (End-point of Current-entry = Junction)
      then  Remove Current-entry from Front;
      Read Current-entry from next entry in Front;
      end

else Left-side-exists <- false;

While (height of Current-entry = height of Junction)
do    begin
      Concatenate(Below(Current-entry),Above-List);
      Above-list <- Above(Current-Line);
      If     (End-point of Current-entry = Junction)
      then  Remove Current-entry from Front;
      Read Current-entry from next entry in Front;
      end;

Test-for-Intersection( Current-entry,
                       Front-predecessor(Current-entry ) );

Return( Above-list, Below-list, Left-side-exists );

end;
```

Pseudo-code outline of the REMOVE procedure incorporating Region-list maintenance.

A1.3 Pseudo-code outline of the PERMUTE procedure incorporating Region-list maintenance.

NOTE :-

A new variable **Region-sub** has been introduced which is used to climb the Front from Lowest-entry to Highest-entry and initialise new Region-lists between adjacent pairs of line-segments. Two new parameters, **Above-list** and **Below-list**, respectively hold the Region-list above the highest line-segment on the left-side of the Junction and below the lowest line-segment on the left-side of the Junction. These Region-lists were found by the Remove procedure, and may have belonged to line-segments which might or might not have been removed from the Front.

```
Procedure Permute (Junction,Front,Above-list,Below-list);
begin

Read Current-entry from the tail entry of the Front;

While (height of Current-entry < height of Junction)
do    Read Current_entry from next entry in Front;

if    (height of Current-entry = height of Junction)
then begin
      lowest-entry <- Current-entry;

      While (height of Current-entry = height of Junction)
      do    begin
            highest_entry <- Current_entry;
            Read Current_entry from next entry of Front;
            end;

      low_swap_sub <- lowest-entry;
      high_swap_sub <- highest-entry;

      While (low-swap-sub <> High-swap-sub)
      do    begin
            swap(low-swap-sub, high-swap-sub);
            low-swap-sub  <- Front_successor(low-swap-sub);
            high-swap-sub <- Front_predecessor(high-swap-sub);
            end;

      Test-for-Intersection(  Highest-entry,
                              Front_successor(Highest-entry));

      Test-for-Intersection(  Lowest-entry,
                              Front_predecessor(Lowest-entry) );

      Attach Below-list below(Lowest-entry);
      Attach Above-list above(Highest-entry);
      Region-sub <- Lowest-entry;

      While (Region-sub <> Highest-entry)
      do    begin
            Initialise_Region_Head(Front_Successor(Region-sub));

            Attach below(Front_Successor(Region_sub))
                                    to above(Region_sub);

            Region-sub <- Front_Successor(Region-sub);
            end;

      end;

end;
```

<u>Pseudo-code outline of the PERMUTE procedure incorporating Region-list maintenance.</u>

A1.4 Pseudo-code outline of the INSERT procedure incorporating Region-list maintenance.

NOTES.

All the Region-list maintenance is performed in the section of code between the intersection tests and the end. The four cases of insertion likely to be encountered are catered for in only three code sections. The case where the inserted line lies between a higher and a lower line is implicitly dealt with in the section of code dealing with the case where the inserted line has a higher neighbour.

This version of Insert has two extra parameters.

Above-list and **Below-list** hold the end of the Region-list above the top-most left-side line-segment, and that of the Region-list below the bottom-most left-side line-segment. If there is no Left-hand side to the current Junction, these parameters hold the ends of a Region-list newly initialised by the Advance_Front procedure which wraps around the 'butt-end' of the Junction.

```
Procedure Insert( Line-segment, Front, Above-list, Below-list );
begin
Read Current-entry from the tail entry of the Front;
While (height of Current-entry < height of Line-segment)
or    (      (height of Current-entry = height of Line-segment)
      and    (slope of Current-entry < slope of Line-segment)
      )
do    Read Current_entry from next entry in Front;

Create New-entry;
Assign Line-segment to New-entry;

Attach New-entry between Current-entry
                 and Front_predecessor of Current-entry;

Test-For-Intersection( Front_Successor( New-entry ), New-entry );
Test-For-Intersection( Front_Predecessor(New-entry), New-entry );

If    (height of Current-entry <> height of New-entry)
and   (height of Front_predecessor(Current-entry)
                            <> height of New-entry)
then begin
      Attach Above-List above(New-entry);
      Attach Below-List below(New-entry);
      end

else If     (New-entry touches a Higher-Line)
      then  begin
            Attach Below(Higher-Line) to  below(New-entry);
            Initialise_Region_Head Below(Higher_line );
            Attach  Below(Higher_line) to Above(New-entry);
            end

      else  begin
            (* New-entry touches a Lower-Line *)

            Attach Above( Lower-Line ) to  above New-entry;
            Initialise_Region_Head Below( New-entry );
            Attach Below( New-Entry) to above( Lower-line );
            end;

end;
```

Pseudo-code outline of the INSERT procedure incorporating Region-list maintenance.

APPENDIX B.

Pseudo-code Outline of the Region Analysis program.

Vertex-based Version which copes with Vertical Lines.

This version of the Region Analysis program develops that presented in Appendix A into a version which can cope with any scene composed of straight-line segments, including vertical lines.

The main difference between this version and that presented in appendix A is that this version can cope with vertical line segments. This has required considerable reworking of the main procedures - Advance_Front, Remove, Permute and Insert.

Remove and **Permute** have been altered to operate on all vertices that they might find within a given vertical range of the Front instead of operating on a single point as in the previous version. These procedures can still work for single point junctions, this achieved by setting the top of the vertical range to the same value as the bottom of the range. The Region-list handling aspects of Remove and Permute remain essentially the same. Remove concatenates lists between successive front entries that it finds in the range with the added difference that some of these pairs of entries MIGHT now be vertically remote from each other, the connection between the lists requiring two vertices in this case, one for the top line and one for the bottom. Permute initialises new lists between pairs of adjacent lines coming out of the right-hand side of the junction, again with the added difference that some of these lines might be vertically separate and that connections between such require two vertices.

Insert has required a similar amendment. In the previous version, Insert diverted one Region-list and initialised another whenever the inserted line touched another entry in the Front. Now these operations have to be performed whenever Insert finds a neighbour for the inserted line which lies within the vertical range. If the neighbour is vertically remote from the new insertion, the Region-list coming from the neighbour towards the new insertion must be split at the start-vertex of the new insertion and diverted around one of its sides. The other end of the split list must be attached to a new list initialised to attach to the other side of the inserted line. When Insert finds that the new insertion has no neighbours, the Region-lists carried forward from the left-hand side of the Junction (if it exists) are extended around the new insertion. When the Region_lists are extended from the left-hand side to the right-hand side, care must be taken to divert them around the top and bottom vertices of the vertical range if they do not already pass through those

points.

Similar care must be taken when initialising and concatenating Region-lists around the left and right side respectively of a vertical line. Such operations are performed by the Advance-Front procedure, and must ensure that both the top and bottom vertices of vertical lines are included in the Region lists.

Before the pseudo-code is presented, some minor procedures used in the code are described.

Front operators.

Front_Successor(front-entry) - returns the next higher entry in the Front to the given front-entry.

Front_Predecessor(front-entry) - returns the next lower entry in the Front to the given front-entry.

Below(front-entry) - references the head of the Region-list attached below the given front-entry.

Above(front-entry) - references the tail of the Region-list attached above the given front-entry.

Region list operators.

Initialise_Region_Head (Side of front-entry) - this procedure generates a new list containing a single vertex corresponding to the start-vertex of the line-segment described by the given front-entry. A pointer to the head of the new list is stored in the specified side, Above or Below, of the given front entry.

Concatenate (tail of list) with (head of list) - joins the head and tail of the list(s) together. This either results in two separate lists being combined into one, or in a single list being turned into a closed loop of vertices by having its head attached to its tail.

Extend_Tail (Front-entry) To (new_vertex) - appends new-vertex onto the tail end of the list above the specified Front-entry.

Extend_Head (Front-entry) to (new_vertex) - appends the new_vertex to the head of the Region_list below the specified front-entry.

Remove_tail (Front_entry) - removes the item at the tail of the specified list and re-attaches new tail to given Front_entry.

Remove_Head (Front_Entry) - removes the head item of the specified list and re-attaches new head to given Front_entry;

Attach (end of list) to (Side of front-entry) - attaches either the head or tail of a list to the specified side of the front-entry.

B.1 Pseudo-code of the Advance Front procedure incorporating Vertical-line and Region-list maintenance.

NOTES :-

Variable **Junction** is now a record structure which contains **Junction.Top** and **Junction.Bottom** which respectively hold the top-most and bottom-most vertex of any vertical line-segment.

In the case of single-point Junctions, **Junction.Top** and **Junction.Bottom** both hold the same vertex.

```
Procedure Advance_Front;
begin
Read Transition from Transition_list;

While not(End_of Transition_list)
do     begin
       Read Junction from Junction_list of Transition

       While  not( End_of Junction_list)
       do     begin
              Top_of_vertical_line <- Junction.top;
              Bottom_of_vertical_line <- Junction.bottom;

              (Above_list,
              Below_list,
              Left_side_exists) <- Remove (Junction,Front);

              If     (Left_side_exists)
              then Lines_permuted <- Permute ( Junction,Front,
                                               Above_list,
                                               Below_list)
              else   begin
                     Lines_permuted  <- false;
                     Initialise_Region_Head (Below_list);
                     Attach Below_list to Above_list;
                     If (Top_of_vertical_line <> Bottom_of_vertical_line)
                     then Extend_tail Below_list to Top_of_vertical_line;
                     end;

              (* start of section which reads through all the      Junction
              records for the current vertical line and Inserts all the
              Insertion-list entries for each Junction record.     *)

              Lines_inserted <- false;

              While ( not (End_of Junction_list)
              and    (Junction.bottom <= top_of_vertical_line))
              do     begin
                     Read Line_segment from Insertion_list of Junction;
                     If     not(End_of Insertions_list)
                     then   Lines_inserted <- true;
                     While  not(End_of Insertions_list)
                     do     begin
                            Insert (Line_segment, Front,
                                          Top_of_vertical_line,
                                          Bottom_of_vertical_line,
                                          Above_list, Below_list);
                            Read Line_segment from Insertion_list of Junction;
                            end;
                     Read Junction from Junction_list of Transition;
                     end;

              If     not(Lines_permuted or Lines_inserted)
              then   Concatenate( Above_list, Below_list);
              end;
       Read Transition from Transition_list;
       end;
end;
```

Pseudo-code of the Advance Front procedure with Vertical-line and Region-list maintenance.

B.2 Pseudo code outline of the REMOVE procedure with Vertical line and Region list maintenance.

NOTES :- Variable 'Previous_height' is used to determine whether the current line-segment found between Junction.Bottom and Junction.Top is vertically separate from the vertex previously found. If so, an extra vertex must be appended onto the Region-list before the Region-lists between the two line-segments are concatenated.

```
Procedure Remove (Junction, Front);
begin

Read Current_entry from the tail entry of the Front;

(*    climb the Front until the first line-segment passing above
      Junction.Bottom is found                                    *)

While (height of Current_entry < height of Junction.bottom)
do    Read Current_entry from next entry in Front;


(*    process first line-segment found between Junction.Bottom and
      Junction.Top - if it exists                                 *)

If (height of Current_entry <= height of Junction.top)
then  begin
      Left_side_exists <- true;
      Below_List   <- Below(Current_entry);
      Above_list   <- Above(Current_entry);
      Previous_height <- height of Current_entry;

      If     (End_point of Current_entry in Junction)
      then   Remove Current_entry from Front;

      Read Current_entry from next entry in Front;
      end

else  Left_side_exists <- false;



(*  process other line-segments between Junction.Bottom and Junction.Top  *)

While (height of Current_entry <= height of Junction.top)
do    begin

      If     (height of Current_entry > Previous_height)
      then   Extend Above_list to Previous_height;

      Concatenate( Below(Current_entry), Above_List);
      Above_list          <- Above(Current_entry);
      Previous_height     <- height of Current_entry;

      If     (End_point of Current_entry in Junction)
      then   Remove Current_entry from Front;

      Read Current_entry from next entry in Front;
      end;


Test_for_Intersection(Current_entry, predecessor(Current_entry));

Return( Above_list, Below_list, Left_side_exists );

end;
```

Pseudo code outline of the REMOVE procedure with Vertical line and Region list maintenance.

B.3 Pseudo code outline of the PERMUTE procedure with Vertical line and Region list maintenance.

NOTES :- Only one intersection test is done between groups, that between the lowest entry in the group and the entry below it in the Front. The test on the highest entry is redundant between Groups, but is performed after the last group has been processed. The Region-list initialised between successive groups must be extended to include the vertices of the two groups. A variable 'First Group' is used to prevent this being performed below the lowest group.

```
Procedure Permute ( Junction, Front, Above_list, Below_list );
begin

Read Current_entry from the tail entry of the Front;


While (height of Current_entry < height of Junction.bottom)
do     (*     climb the Front looking for the first entry
              above Junction.Bottom                              *)
       Read Current_entry from next entry in Front;

if     (height of Current_entry <= height of Junction.top)
then   begin

       (*     process Line-segments found between
              Junction.Bottom and Junction.top         *)

       lowest_entry <- Current_entry;
       First-group <- True;



       While  (height of Current_entry <= height of Junction.top)
       do     begin
              (*     Line-segments between Junction.Top and Junction.Bottom
                     are in groups which intersect at the same point      *)

              Group_lowest <- Current_entry;
              Group_height <- height of Current_entry;


              (*     forms Region-list between successive groups -
                     must not be performed for first group found    *)


              If     not(First-Group)
              then   begin

                     (*     group_lowest contains entry at bottom of
                            next group, group_highest contains the entry
                            at the top of the last- hence lower -group *)

                     Initialise_Region_Head Below( Group_Lowest );
                     Attach Below(Group_Lowest)
                                   to Above(Group-Highest);

                     Extend_tail Above(Group_highest) to Group_Highest;
                     end

              else   First-Group  <-  false;



              While  (height of Current_entry = Group_height)
              do     begin
                     (* Search for top-most Line-segment of current group *)

                     Group_Highest <- Current_entry;
                     Read Current_entry from next entry of Front;
                     end;
```

(Continued...)

```
        (*     Swap positions of entries in current group      *)

        low_swap_sub  <- Group_Lowest;
        high_swap_sub <- Group_Highest;


        While ( low_swap_sub <> High_swap_sub )
        do     begin
               swap(low_swap_sub, high_swap_sub);
               low_swap_sub  <- Front_successor(low_swap_sub);
               high_swap_sub <- Front_predecessor(high_swap_sub);
               end;


        Test_for_Intersection(Group_Lowest,
                              Front_predecessor(Group_Lowest));


        Region_sub <- Lowest_entry;


        While  (Region_sub <> Group_Highest)
        do     begin
               (*     Initialise new Region-lists between pairs
                      of adjacent Line-segments  *)

               Initialise_Region_head
                      Below( Front_Successor(Region_Sub));

               Attach Below( Front_Successor( Region_Sub) )
                                    to Above( Region_sub );

               Region_sub <- Front_Successor( Region_sub );
               end;

        end;


    Test_for_Intersection( Group_Highest,
                           Front_successor( Group_Highest ));


    Attach Below_list to below(Lowest_entry);
    Attach Above_list to above(Group_Highest);

    end;


end;
```

B.4 Pseudo code outline of the INSERT procedure with Vertical line and Region list maintenance.

```
Procedure Insert(   Line_segment, Front, Top_of_Vertical_line,
                    Bottom_of_vertical_line, Above_list, Below_list );
begin

Read Current_entry from the tail entry of the Front;


While (height of Current_entry < height of Line_segment)
or    ((height of Current_entry = height of Line_segment)
        and (slope of Current_entry < slope of Line_segment)      )

do     Read Current_entry from next entry in Front;
       (*     Find position in Front of the new Line-segment, first searching
              for those at the same position, then searching through the
              slopes of those at the same position - if they exist.       *)



(*     Place new Line-segment into the Front     *)

Create New_entry; Assign Line_segment to New_entry;
Attach New_entry between Current_entry
                                 and Front_predecessor of Current_entry;

Test_For_Intersection(Front_Successor(New_entry), New_entry);
Test_For_Intersection(Front_Predecessor(New_entry), New_entry);



(*     Region-list maintenance    *)

If     (height of Current_entry > Top_of_vertical_line)
and    (height of Front_predecessor(New_entry)
                                 < Bottom_of_vertical_line )
then   begin
       (*     If the new line-segment doesn't touch any other and its nearest
              neighbours don't lie within the vertical line-segment *)

       If     (height of New_entry <> Top_of_vertical_line)
       then   Extend_Tail Above_list to height of New_entry;
              (*     Extend the Region-lists passed forward from the
                     left-hand side from the Top of the vertical to the
                     Start-point of the New line-segment - if necessary    *)


       If     (height of New_entry <> Bottom_of_vertical_line)
       then   Extend_Head Below_list to height of New_entry;
              (*     Extend the Region-lists passed forward from the
                     left-hand side from the Bottom of the vertical to the
                     Start-point of the New line-segment - if necessary    *)


       Attach Above_List to above(New_entry);
       Attach Below_List to below(New_entry);
       end
```

(Continued...)

```
else  If     (height of Current_entry <= Top_of_vertical_line)
      then   begin
             (*     line-segment doesn't touch any other, but its higher
                    neighbour lies within the vertical line-segment...   *)

             Remove_Head below(Current_entry);
             Extend_Head below(Current_entry) to height of New_entry;
             Attach below( Current_entry) to below( New_entry );
             Initialise_Region_Head below( Current_Entry );
             Attach below(Current_Entry) to Above( New_Entry);
             Extend_tail Above( New_Entry) to height of New_Entry;
             end;


else  If     (height of Front_Predecessor(New_entry)
                                       >= Bottom_of_vertical_line)
      then   begin
             (*     the line-segment doesn't touch any other, but its
               lower neighbour lies within the vertical line-segment...*)

             Remove_Tail above( Front_predecessor(New_entry) );
             Extend_Tail Above(Front_Predecessor(New_Entry) )
                                       to height of New_entry;

             Attach above( Front-Predecessor( New-Entry ) )
                                       to above(New_entry);

             Initialise_Region_Head Below(New_entry);
             Attach Below(New-Entry)
                    to Above(Front-Predecessor( New_entry ) );

             Extend_tail Above(Front_Predecessor( New_Entry )
                    to height of Front-predecessor( New_Entry );

             end;

      else   If     (New_entry touches a Higher_Line)
             then   begin
                    (*     new Line-segment touches a line-segment which
                           is higher than it ...                    *)

                    Attach Below( Higher_Line)
                                   to below( New_entry );

                    Initialise_Region_Head Below(Higher_line);

                    Attach Below(Higher_Line)
                                   to Above( New_entry);

                    end


      else   begin
             (*     the new line-segment touches a line-segment
                    which is lower than it ... *)

             Attach Above( Lower_Line ) to above( New_entry );
             Initialise_Region_Head Below(New_Entry);
             Attach Below( New_Entry) to Above( Lower_Line );

             end;

end;
```

APPENDIX C.

Pseudo-code Outline of the Region Analysis program.

Edge-based version which builds an Adjacency Graph.

This version of the Region Analysis program is a development from that presented in Appendix B. Here the Region-lists are constructed by linking together sides of Edges instead of constructing lists of vertices. Building the Region lists from sides of Edges has the side-effect of constructing an Adjacency Graph which allows the relationships between neighbouring Regions to be identified.

A significant difference between a line-segment and an Edge is that a line-segment may pass through many vertices where intersections take place, whereas an Edge is purely the connection between two successive vertices along a line-segment. An implication of using Edges to build the Region-lists is that Edges and not Line-segments are referred to by Front entries. A further implication is that the Permute procedure is no longer required. At an intersection vertex, Edges are now removed from the Front and replaced by new Edges which start at the Intersection vertex.

A fundamental part of the processing required for constructing Edge-based descriptions of Regions is that of Splitting Edges into smaller Edges when they are found to intersect inbetween their end-points. A new function Split_Edges performs this operation, returning an ordered pair of pointers to the Higher/Right-most and Lower/Left-most edges produced by splitting the given Edge. Always the Lower/Left-most Edge resulting from a split inherits the identity of the Edge which was split. Edges which connected to the Edge which was split now all automatically point to the Lower/Left-most Edge resulting from the split.

The connections between Edges are actually embedded in the data structures of the Edges, one connection for each side of the Edge. The Front entries no longer contain pointers to the heads and tails of lists below and above the line-segments referred to by the entries. Front entries now contain only references to Edges - the embedded connections allowing all the list maintenance operations to be performed. These list operations need no longer be performed in terms of operations to the head and tails of lists - lists can now be extended at any point along their length by splitting the list and connecting in new edges, as must be performed when Insert causes splits to occur in the vertical line and hence extends the Region lists on the left-hand side of the vertical line.

Constructing the Region-lists from Edges allows a new approach to 'Butt-end' processing to be developed. Every Junction with a Left-hand side (with Edges to be Removed) is assumed to be a Left_hand 'butt-end'. As Edges are found during Removes climb up the Junction in the Front, a connection around the Right-hand side of the Junction is constructed when the first Edge is encountered, and maintained by diverting the connection above successive Edges. A handle is maintained on the Top-most edge on the left-hand side and is passed to Insert, if it is called. Insert can then split the connection around the right-hand side of the Junction and divert the two ends of the connection above and below the first Edge inserted.

The only slight deviation from this method occurs in the case of Vertical Lines. In such cases, the connection around the right-hand side of the Junction is constructed in the Advance-Front procedure before Remove is called. At the same time, Advance-Front forms a connection around the Left-hand side of the the Vertical-edge, which effectively takes care of right-hand butt-ends (which have no Edges to Remove) involving Vertical lines. Insert need only concern itself with Right-hand butt-ends which occur around single point Junctions. Here the approach is the same as that adopted in Remove. A connection is formed around the left-hand side of the Junction when the first Edge is inserted (if there is no Edge passed to insert from the Left-hand side). This connection is maintained as other Edges are inserted by diverting the connection around the Junction above or below successive Edges which become the highest or lowest Edges emerging from the Junction. This maintenance is performed regardless of whether a left-hand side exists or not, and so the 'butt-end' processing is largely transparent.

Diverting the connection above the Top-most-edge emerging from the right-hand side of the Junction is performed in two contexts, each requiring a different solution.

One context is that the new (most recently inserted) Top-most-edge does not share the same start-vertex as the previous Top-most-Edge. An implication of this is that the Junction contains a vertical line, and that therefore a left-hand side (which may merely be the left-hand side of the Vertical Line in the case of a butt-end) exists. Diverting the connection above the new Top-most Edge consists of following the list above the Top-most-edge on the left-hand side and diverting it above the new Top-most-edge just before the

previous Top-most-edge is reached.

The other context is that the new Top-most-edge shares the same start-vertex as the previous Top-most-edge. In this situation no judgement can be made as to whether a left-hand side exists or not, and fortunately no such judgement needs to be made. The Edge which connects to the previous Top-most-Edge has one of its vertices at the start-point of the previous Top-most-edge. Finding that edge requires the clockwise cyclic link around the vertex to be followed until a connection back to the previous Top-most-Edge is found. That connection is then diverted above the new Top-most-Edge.

This method of navigating the vertex is applied whenever Insert places an Edge which touches only lower neighbouring Edges in the Front. A new function has been written to perform this navigation - Navigate-Vertex - which returns a pointer to the side of the Edge connecting to the side of the Edge from which the navigation started.

Before the pseudo-code is presented, some minor procedures used in the code are described.

Edge functions.

Right(edge) and **Below(edge)** effectively perform the same task, that of returning the address of the Below-Right side of the given edge. They are provided as separate functions for purely to assist in visualising which side of an Edge is being referred to.

Left(edge) and **Above(edge)** return the address of the Above-Left side of an Edge. They are identical yet separate to assist in visualisation of the side being referred to.

First_Edge(line) - returns the first edge, ie Bottom-most (Verticals) or Left-most edge, in the Edge-list of the given line-segment data structure.

Last-Edge(line) - returns the last edge, ie Top-most (Verticals) or Right-most edge, in the Edge-list of the given Line-segment.

Next_Edge(edge) - returns the next edge, ie higher (Verticals) or further right edge, in the Edge-list of the Line-segment to which the given Edge belongs.

Start_Vertex(edge) - returns the XY co-ordinates of the lowest (Verticals) or left-most vertex of the given Edge.

End_Vertex(edge) - returns the XY co-ordinates of the highest (Verticals) or right-most vertex of the given Edge.

Split (edge) at (vertex) - splits the given Edge at the given vertex. It returns the ordered pair (High_Split_Edge, Low_Split_Edge) which contains the addresses of the higher (vertical) or further right (non-vertical) Edge, and the lower (vertical) or further left (non-vertical) Edge resulting from the split.

Navigate_Vertex (Start side of Edge) - follows the clockwise cyclic list around the bottom/left-most vertex of the Edge if the Above-left side is specified, or around the Top/Right-most vertex if the Below-right side is specified. The function returns the side of the Edge which connects (clockwise) to the starting side of edge.

Front operators.

Front_Successor(edge) - returns the next higher entry in the Front to the given Edge.

Front_Predecessor(edge - returns the next lower entry in the Front to the given Edge.

Region list operators.

Assign (source side) to (new side) - the connection from the source side is assigned to the new side.

Connect (source side) To (destination side) - source side is connected to destination side. Source side is labelled as belonging to the same Region_list as destination side.

Next_Side (Source side) - returns a pointer to the next side after source side iin the Region-list.

C.1 Pseudo-code of the Advance Front procedure with Edge-based Region-list maintenance.

```
Procedure Advance_Front;
begin
Read Transition from Transition_list;

While  not( End_of Transition_list )
do  begin

    Read Junction from Junction_list of Transition

    While   not( End_of Junction_list)
    do begin

        If    ( Junction.Vertical_Line <> nil)
        then begin
              Bottom_Vertex   <- Junction.Bottom;
              Top_Vertex      <- Junction.Top;
              Vertical_Line   <- Junction.Vertical_Line

              (*  Builds Initial Cyclical state of the
                  Edge_list around the Vertical_Line   *)

              Connect  Right(First_Edge(Vertical_Line))
                             to   Left(First_Edge(Vertical_Line));
              Connect   Left(First_Edge(Vertical_Line))
                             to Right(First_Edge(Vertical_Line));
              end

        else begin
              Vertical_Line <- nil;
              Bottom_Vertex <- Junction.Bottom;
              Top_Vertex    <- Junction.Bottom;
              end;

        Top_most_edge   <- Remove(Junction,Bottom_Vertex, Top_Vertex, Front);
        Read Edge from Insertion_list of Junction;

        While not  (End_of Junction_list) and (Junction.Bottom < Top_vertex)
        do    begin

              While not( End_of Insertions_list )
              do    begin
                    Insert( Top_vertex, Bottom_vertex, Vertical_Line,
                                             Top_most_edge, Edge );
                    Read Edge from Insertion_list of Junction;
                    end;
              Read Junction from Junction_list of Transition;
              end;
        end;
    Read Transition from Transition_list;
    end;
end;
```

C.2 Pseudo code outline of the REMOVE procedure with Edge based Region list maintenance.

```
Procedure Remove( Junction, Bottom_Vertex, Top_Vertex, Front );
begin
Search Front for Lowest_Edge Between Current_Vertex and Top_Vertex;

If   (Lowest_Edge exists)
then begin
     If   ( Lowest_Edge > Current_Vertex )
     then begin
          (*   Can only occur if a Vertical_Line _ split Vertical Edge into
               Low_split and High_split and connect them to Lowest_Edge    *)

          Split First_Edge(Vertical_Line)   -> (high_split, low_split );
          Connect Left(High_split)          to Right(high_split);
          Connect Right(High_split)         to Right(low_split);
          Connect Left(Low_split)           to Below(Lowest_Edge);
          Connect Above(Lowest_Edge)        to Left(high_split);
          Current_Vertex                    <- height of Lowest_Edge;
          end

     else if   (Junction has Vertical_Line)
          then begin
               (*   Divert cyclical Edge list from around Vertical
                    to include the First Edge          *)

               Assign Above(Lowest_Edge) to
                         Right(First_Edge(Vertical_Line ));
               Connect Right(First_Edge(Vertical_Line)) to
                                                     Below(Lowest_Edge);
               end
          else begin
               (*   Construct Edge_list around 'butt_end' of
                    Single_point Junction    *)

               Connect Above(Lowest_Edge) to Below(Lowest_Edge);
               end;

     Previous_Edge <- Lowest_Edge;
     Remove record of Previous_Edge from Front;
     Read Current_Edge from next entry in Front;
     Edges_at_Current_Vertex <- 1;
```

```
     While     ( Current_Edge <= Top_Vertex )
     do   begin
          If   ( Current_Edge = Current_Vertex )
          then begin
               Above(Current_Edge)          <- Above(Previous_Edge);
               Connect Above(Previous_Edge) to Below(Current_Edge);
               Increment Edges_at_Current_Vertex;
               end

          else begin
               (*   Current_Edge doesn't touch Previous_Edge *)

               If   (Edges_at_Current_Vertex = 1)
               and  (End_Vertex(Previous_Edge)<>Transition_point)
               then begin
                    (*   Previous_Edge was solitary so split it in two and put
                         right_most part in Transitions List        *)

                    Split Previous_Edge      -> (New_edge, Previous_Edge );
                    Add New_Edge to Transition_list;
                    end;

               (*   Split Vertical_Line and connect Lower part
                    between Current_Edge and Previous_Edge   *)

               Split Vertical_Line           -> (high_split, low_split );
               Connect Left(High_split)      to Right(High_split);
               Connect Right(High_split)     to Left(Low_split);
               Connect Left(Low_split)       to Below(Current_Edge);
               Connect Above(Current_Edge)   to Left(High_split);
               Current_Vertex                <- height of Current_Edge;
               Edges_at_Current_Vertex       <- 1;
               end;

          Previous_Edge <- Current_Edge;
          Remove entry of Previous_Edge from Front;
          Read Current_Edge from next Entry in Front;
          end;

     (*   Intersection test - Current_Edge is above the Previous_Edge
          removed, its predecessor is below    *)

     Test_intersection(Front-Predecessor(Current_Edge), Current_Edge );
     end;  (* if (lowest_edge exists);*)

(*   Set Top_most Edge for passing forward to Insert procedure *)

If   (Junction has a Vertical_Line) and (Lowest_Edge exists)
then If    ( height of Previous_Edge < Top_Vertex )
     then Top_most_Edge <- Last_Edge(Vertical_Line)
     else Top_most_Edge <- Previous_Edge

else begin
     If    (Junction has a Vertical_Line)
     then Top_most_Edge <- First_edge(Vertical_Line)
     else begin
          if    ( Lowest_Edge exists )
          then Top_most_Edge <- Previous_Edge
          else Top_most_Edge <- nil;
          end;
     end;

Return( Top_most_Edge );
end;
```

C.3 Pseudo code outline of the INSERT procedure with Edge based Region list maintenance.

```
Procedure Insert( Top_vertex, Bottom_vertex,  Vertical_Line,
                                              Top_most_edge, Insertion);
begin
Place Insertion in Front;

Test_For_Intersection( Insertion, High_neighbour);
Test_For_Intersection( Insertion, Low_neighbour);

If     ( High_neighbour hits Insertion )
then   begin
       Assign Below(Insertion)          to Below( High_neighbour );
       Connect Below(High_neighbour)    to Below(Insertion);
       Exit;
       end;

if     ( Insertion hits Low_neighbour )
then   begin
       Preceeding_side      <- Navigate_Vertex( Above(Low_Neighbour) );
       Connect Preceeding_Side    to Above(Insertion);
       Connect Below(Insertion)   to Above(Low_neighbour);
       Exit;
       end;

If     ( Vertical_Edge = nil ) and (Top_Most_Edge <> nil )
then begin
       (*     No touching neighbouring Edge has been found and the Current
              Junction does not contain a Vertical_Line _ this insertion is
              therefore the first and must inherit the Region_list from the
              Left_hand side else form a 'butt_end'           *)

       Assign Below( Insertion )        to Above( Top_most_edge);
       Connect Above( Top_most_edge )   to Above( Insertion );
       Exit;
       end;

If     ( Vertical_Edge = nil ) and (Top_Most_Edge = nil )
then   begin
       (*     No touching Edge has been found and current Junction does not
              contain a vertical edge - this insertion is therfore the first
              and since No Left_hand side exists - Form a 'Butt_end'       *)

       Connect Below( Insertion ) to Above( Insertion );
       Exit;
       end;

if     ( Start_vertex(Insertion) = Top_vertex )
then   begin
       (*     Special case _ Insertion is first Edge to be found lying
              at the Top_most_Vertex. Region_list must be diverted from
              Vertical right to above Insertion        *)

       Assign Below(Insertion)     to Above(Top_most_Edge);
       Connect Above(Top_most_Edge) to Above(Insertion);
       Exit;
       end
```

```
If      ( Start_Vertex( Insertion ) <> Top_Vertex )
then  begin
      (*      Search up the Vertical for an Edge starting at the same
              point as the new Insertion        *)

      V <- First_edge(Vertical_Line);
      Finish <- false;

      while  not( Finish )
      do     begin

             If     ( Next_Edge( V ) = nil )
             then   finish <- true;
             else   if  (Start_Vertex(Next_Edge(V)) >
                                            Start_Vertex(insertion))
                    then   finish <- true;
                    else   V <- Next_Edge( V );

             end;


      If     ( Start_Vertex( V ) < Start_vertex(Insertion) )
      then   begin
             (*     No Vertical_Line exists starting at same point as
                    Insertion _ so make one! Careful with connections
                    on Left_side in case they must loop over the Top *)


             Split V -> (high_split, low_split);

             If     ( Next_Side(Left(low_split)) = Right(low_split) )
             then   Connect Left(High_split)   to Right(high_split)
             else   Assign Left(High_split)    to Left(low_split)

             Connect Left(Low_split)     to Left(High_split);
             Connect Right(High_split)   to Above(Insertion);
             Connect Below(Insertion)    to Right(low_split);
             end;


      else   begin
             (*     V contains an Edge starting at the same point as
                    the Insertion - connect them together    *)


             Assign Below(Insertion)     to Right( V );
             Connect Right( V )   to Above(Insertion);
             end;

      end;


end;
```

APPENDIX D.

Pseudo-code Outline of the Region Analysis program.

Edge-based version which builds a Containment Hierarchy.

In Appendix C, a Region_Analysis program capable of constructing Adjacency Graphs by connecting Sides of Edges together was described.

In this version, a Containment Hierarchy is also constructed. To this end, Region_lists are now given unique labels and the Sides of Edges which compose the Region-lists now carry references to these Region_labels. This makes it necessary to substantially modify the code that constructs the Adjacency Graph. In the version described in Appendix C, Region-lists could be split, concatenated and extended with impunity. Now more careful consideration is needed to maintain the consistency of the labelling of Edges within the Region_lists. In this version of the program, Edge operations will be performed entirely within the context of the Region-lists. Typical operations are Extend_Head, Extend_Tail and Concatenate. All the Region_list operators are described before the pseudo-code listings appear.

Using labelled lists to describe the Regions causes problems in dealing with 'Butt-ends'. In the version in Appendix C, it was safe to connect the Top-most-edge on the Left-hand side of a Junction to the Bottom-most-edge to form a list around the right-hand side of the Junction. Any lines subsequently emerging from the Junction on the right-hand side could split this list and divert it around themselves. Positing the assumption that all Junctions are left-handed 'Butt-ends' and then admitting exceptions later enabled a lot of processing to be done with a little code. Unfortunately this is no longer possible. Positing a Left_hand butt-end would require a concatenation of the list above the Top-most-edge with that below the Bottom-most-edge. Should the two lists have separate labels, the two Regions would be combined by this operation leading to the disposal of one of the Region-labels - an undesirable result if a line should later be inserted at that Junction because then two separate lists would be expected, one to extend below and one above the new line.

As a result of using labelled lists, the program is substantially longer and more explicit in dealing with the various possible combinations of lines meeting at Junctions. The Remove and Insert procedures have each been split into smaller procedures. Remove does still exist, but now as an executive procedure which calls other procedures to perform most of its Region-list handling.

Remove calls two procedures: Remove_Lowest_Edge - which performs the list handling associated with the first Edge that Remove encounters at a Junction; and Remove_Edge - which performs the list handling for all subsequent Edges discovered by Remove at that Junction. Remove_Edges, generally speaking, concatenates the list below the Current_Edge with that above the preceeding Edge.

Insert is replaced by Insert_First_Edge and Insert_Edge. Insert_First_Edge performs the special job of extending the Region_lists from above and below the extreme lines on the Left-hand side of the Junction, over to the above and below side of the first Edge inserted on the right-hand side of the Junction. All subsequent Edges to be inserted at that Junction are processed by Insert_Edge - which generally diverts the Region above or below the nearest neighbour of the new Edge above or below the new E

dge itself, and then initialises a new Region between the new Edge and that neighbour. Some 'Butt-end' processing has been shifted back to the Advance-Front procedure - the executive which controls the plane-sweep.

At a Right-hand 'Butt-end' a new Exterior Region is initialised. Advance_Front performs this initialisation for Junctions with Vertical_lines should the Remove procedure fail to find any Edges to remove. For right-hand 'butt-end's without a Vertical_Line, this initialisation is performed in Insert_First_Edge. The rationale behind this is that Insert does not actually encounter Vertical-lines - so Advance-Front must cope with them - and Advance_Front does not process Insertions lists and so cannot handle 'butt-end's constructed around Insertions, which are non-vertical lines.

All Left-handed 'Butt-end's are handled by Advance-Front. This processing - concatenating the Lowest and Highest lists coming out of the Left-hand side of the Junction - is only performed if Insert never gets called, ie. if there is no right-hand side to the Junction. Assuming all Junctions to be 'Butt-end's until proved otherwise, as was done in Appendix C, is no longer possible for reasons outlined previously. Once concatenated, labelled Lists cannot be split into their component parts later.

Edge functions.

Right(edge) and **Below(edge)** effectively perform the same task, that of returning the address of the Below-Right side of the given edge. They are provided as separate functions for purely to assist in visualising which side of an Edge is being referred to.

Left(edge) and **Above(edge)** return the address of the Above-Left side of an Edge. They are identical yet separate to assist in visualisation of the side being referred to.

First_Edge(line) - returns the first edge, ie Bottom-most (Verticals) or Left-most edge, in the Edge-list of the given line-segment data structure.

Last-Edge(line) - returns the last edge, ie Top-most (Verticals) or Right-most edge, in the Edge-list of the given Line-segment.

Next_Edge(edge) - returns the next edge, ie higher (Verticals) or further right edge, in the Edge-list of the Line-segment to which the given Edge belongs.

Start_Vertex(edge) - returns the XY co-ordinates of the lowest (Verticals) or left-most vertex of the given Edge.

End_Vertex(edge) - returns the XY co-ordinates of the highest (Verticals) or right-most vertex of the given Edge.

Split (edge) at (vertex) - splits the given Edge at the given vertex. It returns the ordered pair (High_Split_Edge, Low_Split_Edge) which contains the addresses of the higher (vertical) or further right (non-vertical) Edge, and the lower (vertical) or further left (non-vertical) Edge resulting from the split.

Front operators.

Front_Successor(edge) - returns the next higher entry in the Front to the given Edge.

Front_Predecessor(edge) - returns the next lower entry in the Front to the given Edge.

Region list operators.

Initialise_Exterior_Region_Head (Side of Edge) - creates a new Region_label of type Exterior, sets the given "Side of Edge" as the Head and tail, and positions the label in the Containment_hierarchy. To position the label, this function in reality needs access to the higher-neighbour of the given Edge as described in section 3.9.3.

Initialise_Interior_Region_Head (Side of Edge) - creates a new Region_label of type Interior and positions it in the Containment Hierarchy (See 3.9.4). The given "Side of Edge" is set as the Head and Tail.

Concatenate (tail_edge) with (head_edge) - joins the head and tail edges together. If the two edges belong to separate Region_labels, then amendments to the Containment Hierarchy are required according to context (See 3.9.5 - 3.9.7). If the two Edges belong to the same Region_list, then joining Head to tail causes a closed loop to be formed, indicating that the Region_list is completed.

Extend_Tail (source side) To (new_tail side) - joins the new_tail side to the tail of the Region_list to which the source side belongs. Source side need not necessarily BE the tail itself.

Extend_Head (source side) To (new_head side) - joins the new_head side to the head of the Region_list to which the source side belongs. Source side need not be the Head side.

Extend_Tail_Down_Vertical (source side) To (vertex) - the tail of the Region_list to which the source side belongs must lie on a vertical edge. The tail is extended down the Edges on the right side of that Vertical_line until a vertical Edge starting at the specified vertex is encountered. The vertical Edge starting at that vertex must exist. The source side specified need not be the tail of the Region-list.

Replace_Tail (source side) with (new_tail side) - the tail of the Region-list to which source side belongs is removed. The tail is replaced with the new-tail side.

Replace_Head (source side) with (new_head side) - the head of the Region_list to which source side belongs is removed. The head is replaced by the new_head side.

Retrace_Tail (new_tail side) - the Region_list to which new_tail side belongs is followed. The Region_list is truncated at new_tail, which becomes the new tail of the Region_list.

Retrace_Head (new_head side) - the Region_list to which new_head belongs is followed. The Region_list ahead of new_head is removed, the new_head becoming the new head, and the list between new_head and tail becomming the entire list.

Connect (source side) To (destination side) - source side is connected to destination side. Source side is labelled as belonging to the same Region_list as destination side.

Next_side (source side) - returns the next side in the Region-list to which source side belongs.

D.1 Pseudo-code of the Advance_Front procedure with Containment Hierarchy maintenance.

```
Procedure Advance_Front;
begin

Read Transition from Transition_list;

While not( End_of Transition_list )
do begin

   Read Junction from Junction_list of Transition

   While  not( End_of Junction_list)
   do begin

      If    ( Junction has a Vertical_Line )
      then begin
           Bottom_Vertex   <- Junction.Bottom;
           Top_Vertex      <- Junction.Top;
           Vertical_line   <- Junction.Vertical_line;
           end

      else begin
           Bottom_Vertex   <- Junction.Bottom;
           Top_Vertex      <- Junction.Bottom;
           Vertical_Line   <- nil;
           end;


      (Top_Most_Edge, Bottom_Most_Edge) <- Remove( Bottom_Vertex,Top_Vertex,
                                                    Vertical_Line, Front );



      If    (Top_Most_Edge = nil) and (Vertical_Line <> nil)
      then begin

           (* Nothing has been removed and a Vertical Line exists- therefore
           a Right_Hand butt end - extend new Region around vertical line*)

           Initialise_Exterior_Region_Head
                                  -> Right(First_Edge(Vertical_Line));

           Extend_Tail Right(First_Edge( Vertical_Line )
                                  to Left(First_Edge( Vertical_Line );

           Top_Most_Edge          <- Left( First_Edge(Vertical_Line) );
           Bottom_Most_Edge       <- Top_Most_Edge;
           end;
```

(Continued...)

```
(* ensure first Insertion of Junction is handled by Insert_First *)

First_Insertion <- true;


While not (End_of Junction_list) and (Junction.Bottom < Top_Vertex )
do   begin
     Read Edge from Insertion_list of Junction;

     While not( End_of Insertions_list )
     do begin

        If (First_Insertion)
        then begin
             First_Insertion <- false;
             Insert_First( Top_Vertex, Bottom_Vertex,
                           Vertical_Line, Top_Most_Edge, Edge );

             end

        else Insert_Edge( Top_Vertex, Bottom_Vertex,
                          Vertical_Line, Top_Most_Edge, Edge );


        Read Edge from Insertion_list of Junction;
        end;

     Read Junction from Junction_list of Transition;
     end;


If   ( Vertical_Line <> nil ) and (First_Insertion)
then
     (* nothing inserted - so left-hand butt-end around
        Vertical_Line *)

     Concatenate( Above(Top_Most_Edge),
                           Right( Last_Edge(Vertical_Line) ) );


If   ( Vertical_Line = nil ) and (First_Insertion)
then
     (* nothing inserted - so left-hand butt-end
     around single point Junction *)

     Concatenate( Above(Top_Most_Edge), Below(Bottom_Most_edge) );

end;

   Read Transition from Transition_list;
   end;

end;
```

D.2 Pseudo-code outline of the REMOVE procedure with Edge_based Region_list maintenance.

```
Procedure Remove( Bottom_Vertex, Top_Vertex, Vertical_Line, Front );
begin

Current_Vertex <- Bottom_Vertex;
Search Front for Bottom_Most_Edge between Current_Vertex and Top_Vertex;

If (Bottom_Most_Edge exists)
then begin


     If   (Vertical_Line Exists)
     then
          (* Remove_Lowest_Edge is only called when the lowest edge has
               to be connected to something - ie a vertical line *)

          Remove_Lowest_Edge( Bottom_Most_Edge, Current_Vertex,
                                              Vertical_Line);



     Remove record of Bottom_Most_Edge from Front;
     Previous_Edge  <- Bottom_Most_Edge;
     Current_Vertex <- Start_Vertex( Bottom_Most_Edge );
     Edges_at_Current_Vertex <- 1;

     Read Current_Edge from next entry in Front;


     While ( Current_Edge <= Top_Vertex )
     do   begin

          Edges_at_Current_Vertex <-
               Remove_Edge( Previous_Edge, Current_Edge,
                            Current_Vertex, Edges_at_Current_Vertex,
                                                        Vertical_Line );


          Current_Vertex <- Start_Vertex( Current_Edge );
          Previous_Edge <- Current_Edge;
          Remove entry of Previous_Edge from Front;
          Read Current_Edge from next Entry in Front;

          end;



     (*Intersection test -Current_Edge is above the Previous_Edge
       removed, its predecessor is below   *)

     Test_intersection(Front_Predecessor(Current_Edge), Current_Edge );
     end;
```

(Continued...)

```
(* Set Top_most Edge for passing forward to Insert procedure *)


If    (Vertical_Line Exists) and (Previous_Edge exists)
then begin

     If    ( Start_Vertex( Previous_Edge ) < Top_Vertex )
     then begin
          Top_Most_Edge <- Last_Edge( Vertical_Line );

          Extend_Tail Above( Previous_Edge )
                            to Left( Top_Most_Edge );
          end;


     If    ( Start_Vertex( Previous_Edge ) = Top_Vertex )
     then Top_Most_Edge <- Previous_Edge;

     end


else begin

     If    (Vertical_Line Exists)
     then Top_Most_Edge <- First_Edge( Vertical_Line );

     If    Not(Vertical_Line Exists) and ( Previous_Edge exists )
     then Top_Most_Edge <- Previous_Edge

     If    Not(Vertical_Line Exists) and Not( Previous_Edge exists )
     then Top_Most_Edge <- nil;

     end;


Return( Top_Most_Edge, Bottom_Most_Edge );
end;
```

D.3 Pseudo-code outline of the REMOVE-LOWEST_EDGE procedure with Edge based Region list maintenance.

```
Procedure Remove_Lowest_Edge( Lowest_Edge, Bottom_Vertex,
                                  Top_Vertex, Vertical_Line);
begin

If ( Start_Vertex( Lowest_Edge ) = Top_vertex )
then  begin

      Extend_Head Below( Lowest_Edge )
                to Left( First_Edge( Vertical_Line) );
      Extend_Head Left( First_Edge( Vertical_Line ) )
                to Right( First_Edge( Vertical_Line ) );

      Exit;
      end;



if   ( Start_Vertex( Lowest_Edge ) =  Bottom_Vertex )
then  begin

      Extend_head Below( Lowest_Edge )
                to Right( First_Edge( Vertical_Line ) );

      Exit;
      end;



If   ( Start_Vertex( Lowest_Edge ) > Bottom_Vertex )
then  begin

      (*    split Vertical Edge into Low_Split_Edge and High_Split_Edge and
            connect them to Lowest_Edge *)

      Split First_Edge(Vertical_Line) at Start_Vertex(Lowest_Edge)
                              -> (High_Split_Edge, Low_Split_Edge );

      Extend_head Below(Lowest_Edge)          to Left(Low_Split_Edge);
      Extend_head Left(Low_Split_Edge)        to Right(Low_Split_Edge);
      Extend_Head Right( Low_Split_Edge)      to Right(High_Split_Edge);

      Exit;
      end;


end;
```

D.4 Pseudo-code outline of the REMOVE-EDGE procedure with Edge_based Region_list maintenance.

```
Procedure Remove_Edge( Previous_Edge, Current_Edge, Current_Vertex,
                          Top_Vertex,Edges_at_Current_Vertex, Vertical_Line)
begin

If     ( Start_Vertex( Current_Edge ) = Current_Vertex )
then   begin
       Concatenate Above( Previous_Edge ) with Below( Current_Edge );
       Return( Edges_at_Current_Vertex + 1 );
       Exit;
       end

If    ( Start_Vertex( Current_Edge ) <> Current_Vertex )
then   begin

       (* Current_Edge doesn't touch Previous_Edge *)

       If    (Edges_at_Current_Vertex = 1) and
                     not(Transition_point <> End_point( Previous_Edge) )
       then   begin

              (*     Previous_Edge was solitary -so split it in two and put
                     right_most part in Transitions List             *)

              Split Previous_Edge at Transition_point
                                             -> (New_Edge, Previous_Edge );
              Add New_Edge to Transition_list;
              end;

       If    ( Start_Vertex( Current_Edge ) <> Top_Vertex )
       then   begin

              (*     Split Vertical_Line and connect Lower part between
                     Current_Edge and Previous_Edge  *)

              Split Last_Edge(Vertical_Line) at Start_Vertex(Current_Edge)
                                    -> (High_Split_Edge,Low_Split_Edge);

              Extend_Head Above(Previous_Edge) to Left(Low_Split_Edge);
              Concatenate Left( Low_Split_Edge ) with Below( Current_Edge );
              Extend_Tail Right( Low_Split_Edge ) to Right(High_Split_Edge);

              end;

       If    ( Start_Vertex( Current_Edge ) = Top_Vertex )
       then   begin

              Extend_Tail Above(Previous_Edge)
                                    to Left(Last_Edge( Vertical_Line) );
              Concatenate Below(Current_Edge)
                               with Left( Last_Edge( Vertical_Line) );

              end;

       Edges_at_Current_Vertex <- 1;
       end;

Return( Edges_at_Current_Vertex );
end;
```

D.5 Pseudo-code outline of the INSERT procedure with Edge_based Region_list maintenance.

```
Procedure Insert_First( Top_Vertex, Bottom_Vertex, Vertical_Line,
                                                 Top_Most_Edge, Insertion);
begin

Place Insertion in Front;
Test_For_Intersection( Insertion, High_neighbour);
Test_For_Intersection( Insertion, Low_neighbour);

If Not( Vertical_Line Exists ) and ( Top_Most_Edge = nil )
then  begin

      (*  No Left_Hand side - initialise new Region around insertion *)

      Initialise_Exterior_Region_Head -> Below( Insertion );
      Extend_Tail Below( Insertion ) to Above( Insertion );
      Exit;
      end;

If Not( Vertical_Line Exists ) and ( Top_Most_Edge <> nil )
then  begin

      (*  Left_Hand side exists  -Extend lists around new Insertion *)

      Extend_Tail Above( Top_Most_Edge ) to Above( Insertion );
      Extend_Head Below( Bottom_Most_Edge ) to Below( Insertion );
      Exit;
      end;

If  ( Vertical_Line  Exists ) and ( Start_Vertex( Insertion ) = Top_Vertex )
then  begin

      Extend_Head Right(Last_Edge( Vertical_Line ) ) to Below( Insertion );
      Extend_Tail Above( Top_Most_Edge ) to Above( Insertion );
      Exit;
      end;


If   ( Vertical_Line Exists ) and ( Start_Vertex( Insertion ) < Top_Vertex )
then  begin

      (*     Search up the Vertical for an Edge starting at the same point
             as the new Insertion                                      *)

      V <- First_Edge( Vertical_Line );
      Finish <- false;

      while not( Finish )
      do     begin

             If   ( Next_Edge( V ) = nil )
             then  finish <- true;
             else  if (Start_Vertex(Next_Edge(V)) >Start_Vertex(Insertion))
                   then  finish <- true;
                   else  V <- Next_Edge( V );

             end;
```

(Continued...)

```
    If     ( Start_Vertex( V ) < Start_Vertex( Insertion ) )
    then   begin

           (* No Vertical_Line exists starting at same point as Insertion
                  -so make one!  Careful with connections on Left_side
                  in case they must loop over the Top                    *)

           Split V  at Start_Vertex( Insertion )
                              -> (High_Split_Edge, Low_Split_Edge);

           Left( High_Split_Edge ) <- Left( Low_Split_Edge );
           Connect Left( Low_Split_Edge ) to Left( High_Split_Edge );
           Retrace_Head Right( Low_Split_Edge );
           Extend_Head  Right( Low_Split_Edge ) to Below( Insertion );

           If ( Top_Most_Edge = Low_Split_Edge )
           then   Top_Most_Edge <- High_Split_Edge;

           Extend_Tail Above( Top_Most_Edge )
                              to Right( Last_Edge( Vertical_Line));

           Extend_Tail_Down_Vertical from Right(Last_Edge(Vertical_Line))
                                          to Start_Vertex( Insertion ):

           Extend_Tail Right( Last_Edge( Vertical_Line )
                                                 to Above( Insertion );

           Exit;
           end;

    If    ( Start_Vertex( V ) = Start_Vertex( Insertion ) )
    then   begin

           (* A Vertical_Line exists starting at same point as Insertion*)
           Connect Left( Low_Split_Edge ) to Left( High_Split_Edge );
           Retrace_Head Next_Side(Right( Low_Split_Edge )) ;


           Extend_Tail  Next_Side(Right( Low_Split_Edge ))
                                                 to Below( Insertion );

           Extend_Tail Above( Top_Most_Edge )
                                   to Right(Last_Edge( Vertical_Line));

           Extend_Tail_Down_Vertical from Right(Last_Edge(Vertical_Line))
                                          to Start_Vertex( Insertion );

           Extend_Tail Right( Last_Edge( Vertical_Line )  )
                                          to Above( Insertion ):

           Exit;
           end;

    end;

end;
```

D.6 Pseudo-code outline of the INSERT-EDGE procedure with Edge based Region list maintenance.

```
Procedure Insert-Edge( Top_Vertex, Bottom_Vertex, Vertical_Line,
                                              Top_Most_Edge, Insertion);
begin

Place Insertion in Front;

Test_For_Intersection( Insertion, High_neighbour);
Test_For_Intersection( Insertion, Low_neighbour);


If ( Start_Vertex(Insertion) = Start_Vertex(High_neighbour) )
then begin
      Replace_Tail Below(High_Neighbour) with Below(Insertion);
      Initialise_Interior_Region_Head -> Below(High_Neighbour);
      Extend_Tail Below(High_neighbour) to Above(Insertion);
      Exit;
      end;


if ( Start_Vertex(Insertion) = Start_Vertex(Low_neighbour) )
then begin
      Replace_Head Above(Top_Most_Edge) with Above(Insertion);
      Initialise_Interior_Region_Head -> Below(Insertion);
      Extend_Tail Below(Insertion) to Above(Low_Neighbour);
      Exit;
      end;


If    ( Vertical_Line Exists )
then  begin

      (*     Search up the Vertical for an Edge starting at the same point
             as the new Insertion             *)

      V <- First_Edge( Vertical_Line );
      Finish <- false;

      while not( Finish )
      do     begin

             If     ( Next_Edge( V ) = nil )
             then   finish <- true;
             else   if (Start_Vertex(Next_Edge(V)) >Start_Vertex(Insertion))
                    then   finish <- true;
                    else   V <- Next_Edge( V );
             end;


If    ( Start_Vertex( V ) < Start_Vertex( Insertion ) )
then  begin

      (* No Vertical_Line exists starting at same point as Insertion -so
             make one!  Careful with connections on Left_side in case they
             must loop over the Top                                    *)

      Split V  at Start_Vertex( Insertion )
                               -> (High_Split_Edge, Low_Split_Edge);
```

(Continued...)

```
If      ( Start_Vertex( High_Neighbour ) <= Top_Vertex )
        then    begin
                Left( High_Split_Edge ) <- Left( Low_Split_Edge );
                Connect Left( Low_Split_Edge ) to Left( High_Split_Edge );
                Retrace_Head Right( Low_Split_Edge );
                Extend_Head  Right( Low_Split_Edge ) to Below( Insertion );
                Initialise_Interior_Region_Head -> Below( High_Neighbour );

                Extend_Tail_Down_Vertical from Below( High_Neighbour )
                                                  to Start_Vertex( Insertion );

                Extend_Tail Below( High_Neighbour ) to Above( Insertion );
                Exit;
                end;


        If      ( Start_Vertex( Low_Neighbour ) >= Bottom_Vertex )
        then    begin
                Left( High_Split_Edge ) <- Left( Low_Split_Edge );
                Connect Left( Low_Split_Edge ) to Left( High_Split_Edge );
                If ( Top_Most_Edge = Low_Split_Edge )
                then   Top_Most_Edge <- High_Split_Edge;
                Retrace_Tail Above( Top_Most_Edge );
                Extend_Tail Above( Top_Most_Edge )
                                    to Right( Last_Edge( Vertical_Line));

                Extend_Tail_Down_Vertical from Right(Last_Edge(Vertical_Line))
                                                  to Start_Vertex( Insertion );

                Extend_Tail Right(Last_Edge(Vertical_Line))to Above(Insertion);
                Replace_Head Right( Low_Split_Edge );
                Extend_Head  Right( Low_Split_Edge ) to Below( Insertion );

                Initialise_Interior_Region_Head -> Below( Insertion );

                Extend_Tail Below( Insertion ) to Right( Low_Split_Edge );
                Extend_Tail_Down_Vertical from Right( Low_Split_Edge )
                                          to Start_Vertex( Low_Neighbour );
                Extend_Tail Right( Low_Split_Edge ) to Above( Low_Neighbour);
                Exit;
                end;


        If    ( Start_Vertex( V ) = Start_Vertex( Insertion ) )
        then    begin

                (* A Vertical_Line exists starting at same point as Insertion*)


                If ( Start_Vertex( High_Neighbour ) <= Top_Vertex )
                then   begin
                       Retrace_Head Next_Side( Right( V ) );
                       Initialise_Interior_Region_Head ->Below(High_Neighbour);

                       Extend_Tail_Down_Vertical from Below( High_Neighbour )
                                                  to Start_Vertex( Insertion );

                       Extend_Tail Below( High_Neighbour ) to Above(Insertion);
                       Exit;
                       end;
```

(Continued...)

```
If ( Start_Vertex( Low_Neighbour ) >= Bottom_Vertex )
then    begin

        Replace_Tail Above( Top_Most_Edge );
        Extend_Tail Above( Top_Most_Edge )
                    to Right( Last_Edge( Vertical_Line));

        Extend_Tail_Down_Vertical
                    from Right( Last_Edge( Vertical_Line ) )
                                 to Start_Vertex( Insertion ):

        Extend_Tail Right( Last_Edge( Vertical_Line ) )
                                 to Above( Insertion );

        Initialise_Region_Head -> Below( Insertion );
        Extend_Tail Below( Insertion ) to Next_Side( Right(V) );

        Extend_Tail_Down_Vertical from Next_Side( Right( V ) )
                              to Start_Vertex(Low_Neighbour);

        Extend_Tail Next_Side( Right( V ) )
                                 to Above( Low_Neighbour );

        Exit;
        end;



Connect Left( Low_Split_Edge ) to Left( High_Split_Edge );
Retrace_Head Next_Side(Right( Low_Split_Edge )) ;

Extend_Tail  Next_Side(Right( Low_Split_Edge ))
                                 to Below( Insertion );

Extend_Tail Above( Top_Most_Edge )
                    to Right( Last_Edge( Vertical_Line));

Extend_Tail_Down_Vertical from Right(Last_Edge(Vertical_Line))
                                 to Start_Vertex( Insertion ):

Extend_Tail Right( Last_Edge( Vertical_Line ) )
                                          to Above( Insertion );

Exit;
end;


    end;


end;
```