A SYSTEM THAT LEARNS TO RECOGNIZE 3-D OBJECTS

BY

GABRIEL GABRIELIDES, M.PHIL.

A Doctoral Thesis

submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

May, 1988.

SUPERVISOR: DR. C.J. HINDE,

Department of Computer Studies

für meine Frau Michaela

... στη μνήμη του
πατέρα μου που
δεν πρόλαβε να
τη δει τελειωμένη ...

DECLARATION

I declare that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

G. GABRIELIDES

MAY, 1988.

CONTENTS

PAGE

ACKNOWLEDGEMENTS i

ABSTRACT ii

CHAPTER 1: INTRODUCTION

1.1 VISION 1

1.2 LEARNING 3

1.3 ABOUT THE ORGANIZATION OF THIS THESIS 5

REFERENCES 6

CHAPTER 2: A SURVEY OF COMPUTER VISION

2.1 INTRODUCTION 8

2.2 EARLY PROCESSING 9

2.2.1 Primal Sketch 11

2.2.2 Stereo Vision 14

2.3 SEGMENTATION 20

2.3.1 Boundary and Region Tracking 20

2.3.2 Texture 22

2.3.3 Motion 24

2.4 LATE VISION 29

2.4.1 Representation of Shapes 29

2.4.2 Recognition 31

2.4.3 Recognition Systems in Practice 33

REFERENCES 38

PAGE

CHAPTER 3: THE 3-D FIGURE SIMULATOR

3.1 INTRODUCTION 45

3.2 LINE DRAWINGS - A SURVEY 46

3.2.1 Roberts' Program 48

3.2.2 Guzman's Program 49

3.2.3 Line Labeling 51

3.2.4 Other Techniques for Line Drawing Interpretation 57

3.2.5 Understanding of Curved-surface Bodies 57

3.3 A SIMULATOR FOR LINE DRAWINGS OF 3-D SCENES 62

3.3.1 3-D Transforms 63

3.3.2 Rotation 63

3.3.3 Translation 65

3.3.4 Scaling 66

3.3.5 Projection of 3-D Objects onto 2-D Space 66

3.3.6 Homogeneous Coordinates - Combined Transformations 68

3.3.7 Viewing Parameters of Projection Systems 72

3.3.8 Back-face Removal 76

3.4 FUNCTION OF THE SIMULATOR 79

3.4.1 The Input 79

3.4.2 The Scene Build-up 81

3.4.3 The Output 84

REFERENCES 91

CHAPTER 4: A SURVEY OF MACHINE LEARNING

4.1 INTRODUCTION 94

PAGE

4.2 LEARNING STRATEGIES 96

4.2.1 Learning from Instruction 96

4.2.2 Learning by Analogy 100

4.2.3 Learning from Observation and Discovery 104

4.3 KNOWLEDGE REPRESENTATION 107

4.3.1 Knowledge Representation Using Logic 108

4.3.2 Structured Knowledge Representations 110

4.4 SPECIAL LEARNING SYSTEMS - GAMES 111

REFERENCES 113

CHAPTER 5: THE FIGURE LEARNER

5.1 INTRODUCTION 117

5.2 LEARNING BY EXAMPLE - A SURVEY 118

5.3 FIGURE DEFINITIONS 127

5.3.1 1-D Figures 127

5.3.2 2-D Figures 129

5.3.3 3-D Figures 133

5.4 THE LEARNER 141

5.4.1 The Elementary-Concept Learner 143

5.4.2 The Multiple-View Learner 154

5.4.3 The Single-View Learner 165

REFERENCES 171

CHAPTER 6: THE RECOGNIZER

6.1 INTRODUCTION 173

6.2 PICTURE SEGMENTATION 174

PAGE

6.3 SPECIAL FEATURES 178

6.3.1 Containment of a Vertex 179

6.3.2 Convexity of Quadrilaterals 182

6.3.3 Convexity of Contour 183

6.4 RECOGNITION 185

6.4.1 Single-View Recognition 186

6.4.2 Multi-View Recognition 193

6.4.3 Assumptions 194

6.5 FUNCTION OF THE RECOGNIZER 202

REFERENCES 209

CHAPTER 7: DISCUSSION

7.1 INTRODUCTION 210

7.2 THE SIMULATOR 210

7.3 THE LEARNER 212

7.4 THE RECOGNIZER 215

7.5 CONCLUSION 219

APPENDIX 1

1.1 THE 'C' PROGRAMS/REFERENCES 220

1.2 LISTING OF THE 'C' PROGRAMS 221

APPENDIX 2

2.1 THE 'PROLOG PROGRAMS/REFERENCES 236

2.2 LISTING OF THE 'PROLOG' PROGRAMS 238

PAGE

APPENDIX 3

3.1 INTRODUCTION TO PREDICATE CALCULUS 271

3.2 INTRODUCTION TO PROLOG 274

3.2.1 Syntax 274

3.2.2 Programming 276

3.2.3 PROLOG Versions 278

REFERENCES 279

ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks to the following people, for making this work possible:

his supervisor, Dr. C.J. Hinde, for his thorough guidance and his invaluable assistance throughout the duration of the research, and especially for his successful comments and suggestions during the period of writing up,

the director of research, Professor D.J. Evans, for his encouragement to undertake this project,

members of staff, Mr. G.S. Samra and Mr. S. Bedi, for their advice in programming,

and administrative assistant, Mrs. H.M. Johnson, for being helpful with providing office facilities.

He is also grateful to:

his parents, Helene and Thanasses for their moral and financial support during his studies,

his friends Vassili, Denise, Maria and Nikos, for their hospitality on his numerous 'visits' to the U.K., during the writing up period,

and finally the typist, for doing a thorough and precise job.

ABSTRACT

A system that learns to recognize 3-D objects from single and multiple views is presented. It consists of three parts: a *simulator of 3-D figures*, a *learner*, and a *recognizer*.

The 3-D figure simulator generates and plots line drawings of certain 3-D objects. A series of transformations leads to a number of 2-D images of a 3-D object, which are considered as different views and are the basic input to the next two parts.

The learner works in three stages using the method of *learning from examples*. In the first stage an *elementary-concept learner* learns the basic entities that make up a line drawing. In the second stage a *multiple-view learner* learns the definitions of 3-D objects that are to be recognized from multiple views. In the third stage a *single-view learner* learns how to recognize the same objects from single views.

The recognizer is presented with line drawings representing 3-D scenes. A *single-view recognizer* segments the input into faces of possible 3-D objects, and attempts to *match* the segmented scene with a set of single-view definitions of 3-D objects. The result of the recognition may include several alternative answers, corresponding to different 3-D objects. A unique answer can be obtained by making assumptions about hidden elements (e.g. faces) of an object and using a *multiple-view recognizer*. Both single-view and multiple-view recognition are based on the structural relations of the elements that make up a 3-D object. Some analytical elements (e.g. angles) of the objects are also calculated, in order to determine *point containment* and *convexity*.

The system performs well on polyhedra with triangular and quadrilateral faces. A discussion of the system's performance and suggestions for further development is given at the end.

The simulator and the part of the recognizer that makes the analytical calculations are written in C. The learner and the rest of the recognizer are written in PROLOG.

CHAPTER 1

INTRODUCTION

1.1 VISION

Sight is a vital sense of human (and other living) beings. It involves a series of processes, known as *vision*, that allow them to perceive world scenes. Vision begins with forming a 2-D image of objects through an input device, the *eye*. The eye captures the light deflected by an object, through a system of a pinhole (the *pupil*) and a lens of adjustable curvature, and focuses it on a spherical surface, the *retina*. The retina is a thin sheet of interconnected *nerve cells*, including light-sensitive cells which convert light into electrical pulses. Each nerve cell is simulated by the light originated by a point of the surface of the object and sends a signal to the brain through the *optic nerve*. A special part of the *cortex* - brain surface - is engaged with the task to use the incoming information in order to recognize the object and use it appropriately [Gregory '79]. The process of human perception [Lindsay & Norman '77] is still an unanswered problem and a lot of disciplines are concerned with its solution.

The design, construction and use of intelligent machines (e.g. robots) stems from the human desire to lighten his life from tedious tasks and thus employ his brain with high-level activities (e.g. research). At the same time, this tendency can be regarded as an attempt to understand his bodily functions and thus improve his efficiency. Since the introduction and use of digital computers in physical and applied sciences, new fields dealing with the above problem have appeared. One of them with a tremendously fast development in the last twenty years is *Artificial Intelligence*, the primary goal of which is to equip computers with information-processing capabilities comparable to those of biological organisms [Ballard & Brown '82].

Computer vision is the branch of artificial intelligence concerned with the visual aspect of human perception. The problem of computer vision can be formulated as: given a 2-D image, infer the objects that produce it, including their shapes, positions, colours and sizes [Charniak & McDermott '85]. The solution of the problem is carried out by two major processes: *image processing* studying devices, methods and techniques that are responsible for the image formation and the extraction of useful information from it, and *object recognition*, which is responsible for the interpretation of the image according to the experience of the system. Visual perception is the comparison of visual input with already existing models of the real world. The passing from the raw image to the final decision that places it into a certain class is performed in several intermediate stages characterized by respective *representations*. Advanced vision systems rely on powerful, cooperating and rich representations, which are the 'link' between their input and output.

Modern general-purpose computer vision systems proceed in the following basic stages: They begin with the formation of the *generalized image*, a 2-D intensity array, an analogical representation of the input data. The usual input device is an electronic camera that converts the reflected light intensity into electronic signals which are then digitized. Physical properties of the imaged scene like range, texture and motion, are then used to obtain the *Intrinsic image* that makes the scene description more explicit. Next, the image is segmented into sets of elements likely to be meaningful objects. The information contained in the intrinsic image is integrated in an explicit 3-D representation that presents the image to the recognizer. The latter *matches* the segmented image to *models* i.e. *knowledge representations* of the system's previous experience. *Control* mechanisms check the correct flow of information and activities, while *inference* and *planning* enrich the system's knowledge with new facts and seek the best sequence of actions that solves the problem respectively.

1.2 LEARNING

An important characteristic of intelligent beings is their ability to learn. Learning is a complex multi-stage process including knowledge acquisition, effective knowledge representation, development of cognitive skills through instruction or practice, and discovery of new theories and facts via observation and experimentation. The term 'learning' is a rather vague one and is liable to several interpretations. However, a satisfactory definition of learning [Simon '83] is: Learning is any change in a system that allows it to perform better the second time on repetition of the same task or another task drawn from the same population. There are two basic forms of learning: *knowledge acquisition* i.e. learning new symbolic information and applying it in an effective manner, and *skill refinement* i.e. learning by repeated practice and by correcting deviations from a desired behaviour [Carbonell et al '83]. Research on human learning considers it as a slow and tedious process that is a mixture of both forms. It is also highly dependent on the individual and cannot be transferred by copying (a useful computer ability). Despite the efforts of several sciences to explain the mechanisms of human learning, very little has been achieved in this direction.

Machine or *computer learning* is the branch of artificial intelligence concerned with the study and computer modelling of learning processes in their multiple forms of appearance. The objectives of machine learning can be grouped into three basic categories: development of *task-oriented* learning systems, specialized on the study and improvement of certain predetermined tasks; investigation and discovery of new general, *theoretical* methods and algorithms concerning task-independent learning; and computer *simulation*, evaluation and testing of human learning processes i.e. using human-behaviour understanding in favour of machine learning. Each one of the objectives forms a separate research stream. This however, does not prevent the sharing of results and progress by one another.

The basic aspect in machine learning systems is the *learning strategy*. The distinguishing factor that determines each strategy, is the amount of *inference* performed by the learner on the available information. Thus, the spectrum of strategies varies from simple *memorization* of facts and data requiring no inference at all, to learning *by observation and discovery* using a great deal of inference and induction. A very common strategy using a mixed approach is learning *by example*. As in vision, knowledge representation plays an important role in machine learning, depending mainly on the kind of knowledge to be acquired and on the type of task aimed for by the system. Unlike the early steps of machine learning, current research shows interest in developing learning methods based on knowledge-rich systems. The idea is that in order for a system to acquire new knowledge, it must already possess a fair amount of initial knowledge. The importance of machine learning is underlined by the wide area of its applications.

A relatively recent example of the power of knowledge-rich systems, is the appearance of *expert systems*. These are computer systems that encapsulate specialist knowledge about a particular domain of expertise and are capable of making intelligent decisions within that domain [Forsyth '84]. The basic features of an expert system are: a powerful *corpus of knowledge* accumulated during the building of the system by a process called *knowledge engineering;* its knowledge is *explicit* and *accessible*, unlike most conventional programs; a *high-level of expertise* leading to accurate and effective solutions; its *predictive modelling power* that adapts the solutions of the system to changing situations; finally, its *institutional memory*, a concensus of high-level opinions and permanent records of the best strategies and methods, and a *training facility* provided by extra inference software and knowledge about teaching methods [Waterman '86]. Lately however, there is a new tendency to call 'expert', intelligent systems possessing only part of the above features.

1.3 ABOUT THE ORGANIZATION OF THIS THESIS

The aim of this work is to present a system that employs a learning by example method, in order to recognize 3-D objects using as input, simulated 3-D scenes of line drawings. The structure of this thesis is organized round two main topics, *computer vision* and *machine learning*, providing a relevant background and putting the new ideas into context with past and current research.

Chapter 1 is an *introduction to* the concepts of *vision* and *learning* regarded as functions of humans and intelligent computer systems. *Chapter 2* is a *survey of computer vision* systems, methods, techniques and theories, including some of the current research on this topic. *Chapter 3* concentrates on line drawings and describes the low-level part of the vision sub-system, the *simulator of 3-D scenes* that provides the input to the system. *Chapter 4* is a *survey of machine learning* providing a useful background and summarizing the current research on the topic. *Chapter 5* contains the programs that constitute the *learner* and concentrates on the method of learning by example. It also includes the definitions of the recognizable 3-D figures. *Chapter 6* analyses the 3-D figure *recognizer*, that is the high-level part of the vision sub-system. Finally *Chapter 7* contains a *discussion* of the system's performance as well as several *conclusions* and suggestions for further work.

Appendices 1 and *2* contain listings of the programs in *C* and *PROLOG* respectively with explanatory notes and comments on them. In *Appendix 3* there is a brief introduction to *predicate calculus* and the basic concepts of *PROLOG*.

REFERENCES

1. BALLARD, D.H. and BROWN, C.M. 1982: *Computer Vision*, Prentice-Hall p. xii.

2. CARBONELL, J.G., MICHALSKI, R.S. and MITCHELL, T.M. 1983: *Machine Learning: An A.I. Approach*, Tioga, p.6.

3. CHARNIAK, E. and McDERMOTT, D. 1985: *Introduction to A.I.*, Addison-Wesley, p. 89.

4. FORSYTH, R. 1984: *Expert Systems: Principles and Case Studies*, Chapman-Hall Computing, p. vii.

5. GREGORY, R.L. 1979: *Eye and Brain: The Psychology of Seeing*, Weiden-Nicolson, pp. 49-76.

6. LINDSAY, P.H. and NORMAN, D.A. 1977: *Human Information Processing: An Introduction to Psychology*, Academic Press, pp. 3-55.

7. SIMON, H.A. 1983: *Machine Learning: An A.I. Approach*, Eds: Michalski, Carbonell, Mitchell, Tioga, p. 28.

8. WATERMAN, D.A. 1986: *A Guide to Expert Systems*, Addison-Wesley, pp. 6-8.

CHAPTER 2

A SURVEY OF COMPUTER VISION

2.1 INTRODUCTION

Computer vision is the enterprise of automating and integrating a wide range of processes and representations used for vision perception [Ballard & Brown '82,a]. Computer vision systems consist of a series of *low-level* and *high-level* processes, which try to relate their visual input to previously existing models. Low-level or 'early' processing includes techniques like image processing and statistical pattern classification. Their task is to extract *intrinsic images* of brightness, colour, range and other physical properties of a scene, in order to prepare the route for its perception. The task of the high-level processes such as cognition, geometric modelling and planning, is to recognize the scene using their existing experience and knowledge. The success of the system depends to a great extent on the representation of the image, which is the intermediate phase between the image (input) and its final interpretation (output). In the last 20 years a lot of work has been done on vision and a great deal of techniques have been developed for both low and high-level processing. The following paragraphs examine several of these techniques [Brady '86], and refer to some of the most interesting topics and trends in the current research. The first two parts of this chapter cover the processes that lead to the low-level representation of a scene, while the third part examines the high-level representation that leads to the recognition.

2.2 EARLY PROCESSING

The vision process begins with the formation of an image. An imaging device (photographic, television camera) uses a more or less sophisticated method (film-sensitivity, electronic signals, ultra-sound) in order to convert light intensity into an array of samples of some kind of energy [Ballard & Brown '82,b].

An image is represented mathematically by a vector-valued function with a number of arguments called an *image function*. Black and white images are represented by an image function of the form: $f(\vec{x})=f(x,y)$, where $f(x,y)$ is the brightness of the gray level of a point with coordinates (x,y) on the image. There are also multispectral images $\vec{f}$ with components $(f_1,f_2,\ldots,f_n)$ (e.g. colour images), time varying images $\vec{f}(\vec{x},t)$ and 3-D images with $\vec{x}=(x,y,z)$. A special case of image functions is the *digital image function* where the arguments and the values of it, are all integers. These represent *digital images* that are obtained by 'passing' the original image through a *sampling* device, the *digitizer*. The domain of a digital image function is finite (e.g. a rectangle in 2-D) and its range is bounded $0 \leq f(\vec{x}) \leq M$, where M is an integer depending on the resolution of gray levels. Digital images are very important in computer vision because they can be easily processed by computers.

A *generalized image* is a set of *iconic* and *analogical* representations, which may include related images and results of significant processing, and leads to extraction of intrinsic images. An *intrinsic image* [Barrow & Tenenbaum '81] is a grouping of the image into regions corresponding to important physical quantities such as surface orientation, illumination, depth, velocity or colour. It also shows physical boundaries between regions occurring due to abrupt changes in some of the above characteristic quantities. Intrinsic images can be related to physical objects a lot easier than the values of the original input, which reveal physical information only indirectly. However, computing intrinsic images is a difficult research problem and a costly process. An answer to the cost problem is *parallel processing*, since many of the early processing computations can be done simultaneously. Figure 2.1 shows the stages of visual processing.

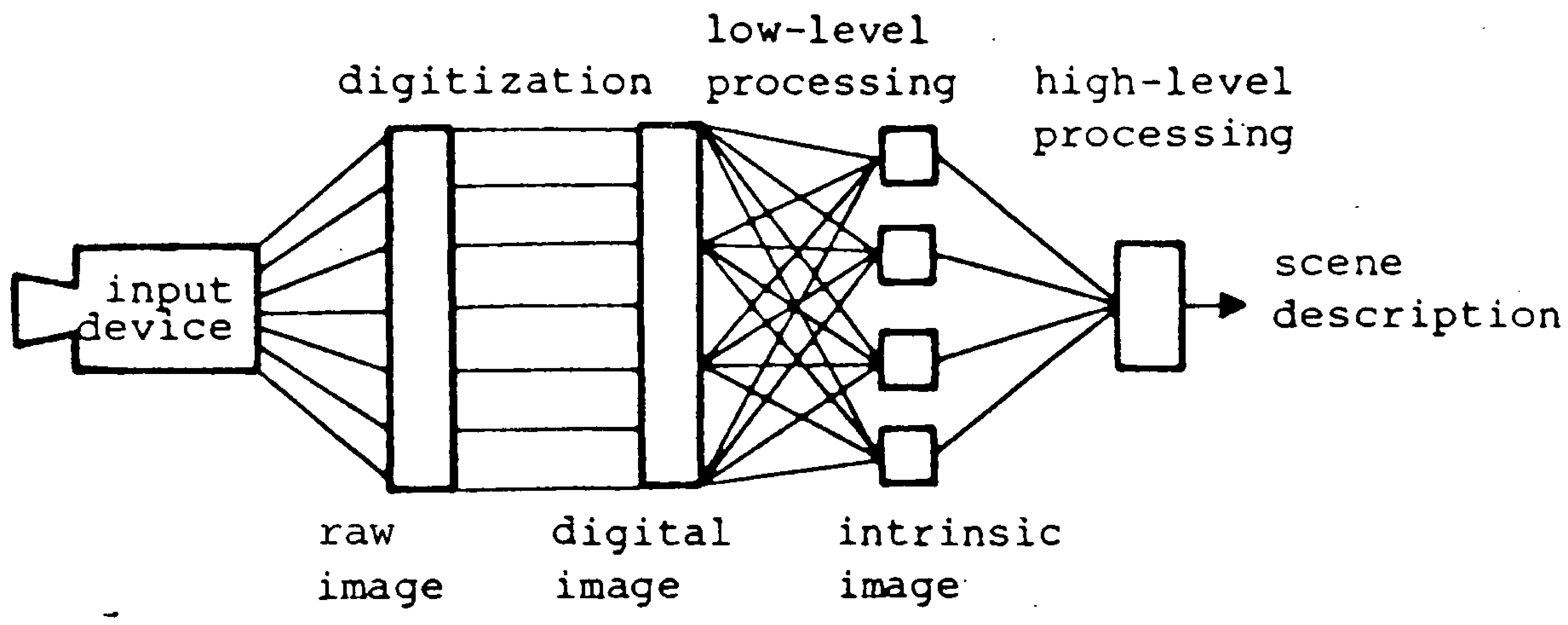


FIGURE 2.1

The first step of the low-level processing is to produce the *(raw) primal sketch*, a cartoon-like representation making explicit image intensity changes (edges) and other features of interest in the image [Marr '76]. A number of other processes work with the primal sketch to extract more information. These are *grouping processes*, *texture analysis*, *motion analysis*, *stereodisparity*, *photometry*, *colour perception* and others. The results of these processes are put together to interpret the lines and regions in the image as physical boundaries and areas and thus produce the intrinsic image.

2.2.1 Primal Sketch

The digitized image consists of a 2-D array of cells, called *pixels* (from picture element), each representing the image intensity at a particular point according to a gray level scale. The smaller the area represented by the pixel, the more detailed the digital picture. Also the finer the gray level resolution, the better the quality of the digital picture. Such an image is called a *gray level image*, and it carries explicit information about local intensity values. All other information is implicit. Gray level images are very important because they lead to the primal sketch.

Primal sketch is the database obtained by searching the gray level image for changes in light intensity. This is done by applying a differential operator on the image. Such an operator is called *spatial operator*, or *weighting function*, *mask* [Marr '76], or *template*, and looks for small distinguishable patches where the level goes from dark to light called *edglets* (= small pieces of edge). The first spatial operators were the *directional derivatives* [Roberts '65], that measured the gradients in the digital image. These were not satisfactory because they required a very large number of devices measuring the gradients for different orientations. The second derivative came to substitute the directional first, and gradient was detected by a *zero-crossing* (second derivative =0) rather than a peak. As suitable operator was chosen the *Laplacian** which is an isotropic operator with a circularly symmetric field. Figure 2.2a shows the behaviour of brightness derivatives at an edglet, in a 1-D picture, and 2.2b the Laplacian.

**The Laplacian of a two-variable function $f(x,y)$ is defined as:*

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}\ , \text{ [Gonzalez \& Wintz '77].}$$

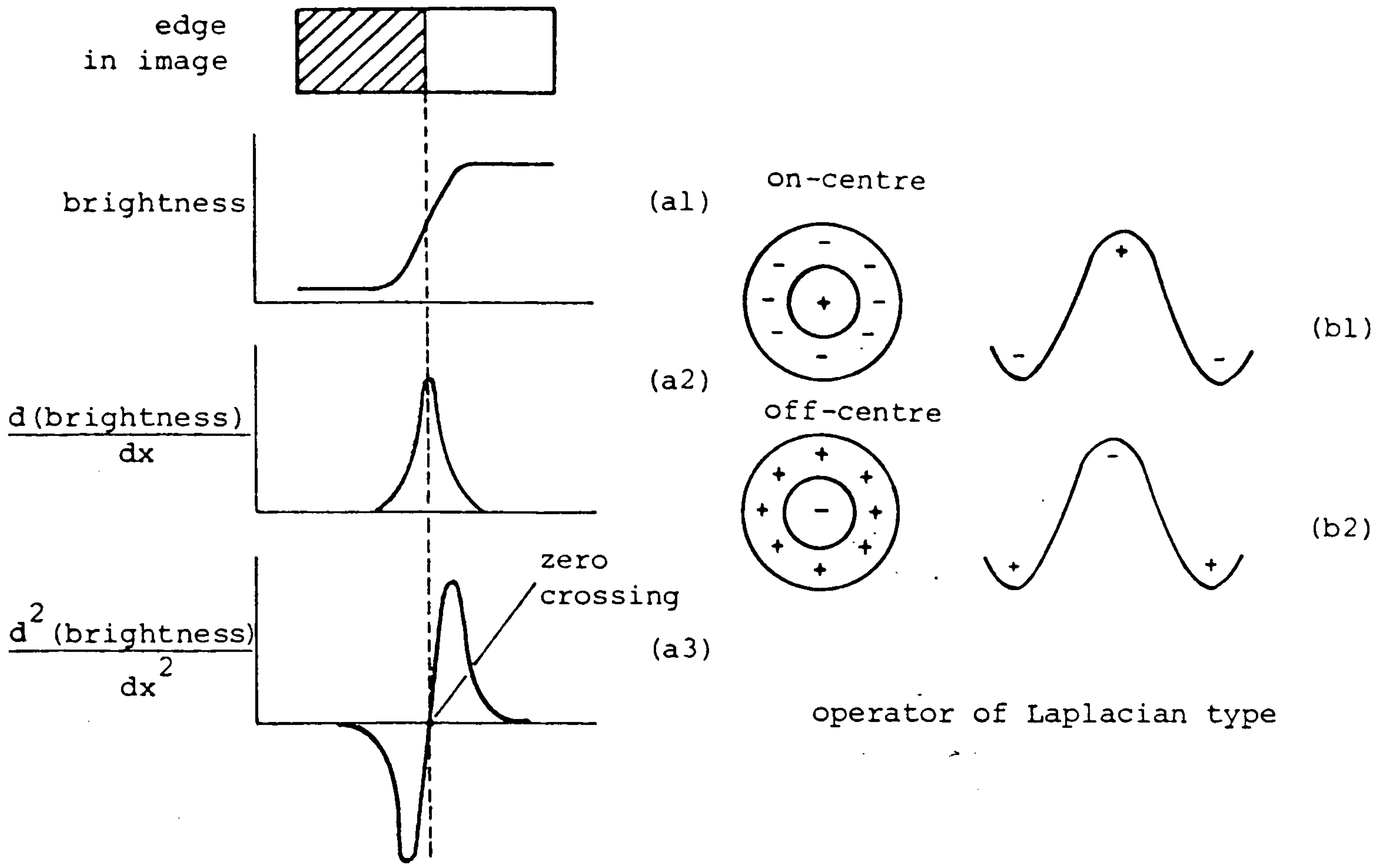


FIGURE 2.2

Intensity changes occur due to spatially localised primitives like edges, bars, blobs, shadows, etc., which must be preserved by the device that measures their gradients. Soon, the need to smooth the gray level picture by applying an *averaging* algorithm was obvious. Averaging is performed by affecting the value of gray level of every pixel in the scene with respect to the values of its neighbouring pixels. The most prevailing idea was to do a weighted average based on the *normal* or *Gaussian* distribution i.e. mathematically to *convolve* the scene with the Gaussian curve [Marr & Hildreth '80]. Convolution* is the application of an operator all over an image [Mayhew & Frisby '84]. In the first step of the primal sketch computation, a convolution of the image (i) with a range of Gaussians (G) of different width was

---

The convolution of two functions $f(x)$ and $g(x)$ denoted by $f(x)*g(x)$, is defined by,

$$f(x)*g(x) = \int_{-\infty}^{+\infty} f(w)g(x-w)dw,$$ *w is a dummy variable of integration.*

Recent research [Georgiou & Anastassiou'86] has designed an architecture for a single chip implementation of specific 3×3 convolution operations.

made symbolically expressed as: G*i. The intensity changes of the smoothed image were determined by the zero crossings in the second derivative over the convolved image, denoted by: $D^2(G*i)$. It can be shown that $D^2(G*i)=D^2G*i$, which means that the same result is obtained by convolving the second derivative of a Gaussian with the image. Finally, the need of an isotropic operator led to the Laplacian and the most suitable operator became a 'Laplacian convolved with a Gaussian'. Figure 2.3 shows the 2nd derivative of a Gaussian curve and its application to an intensity change.

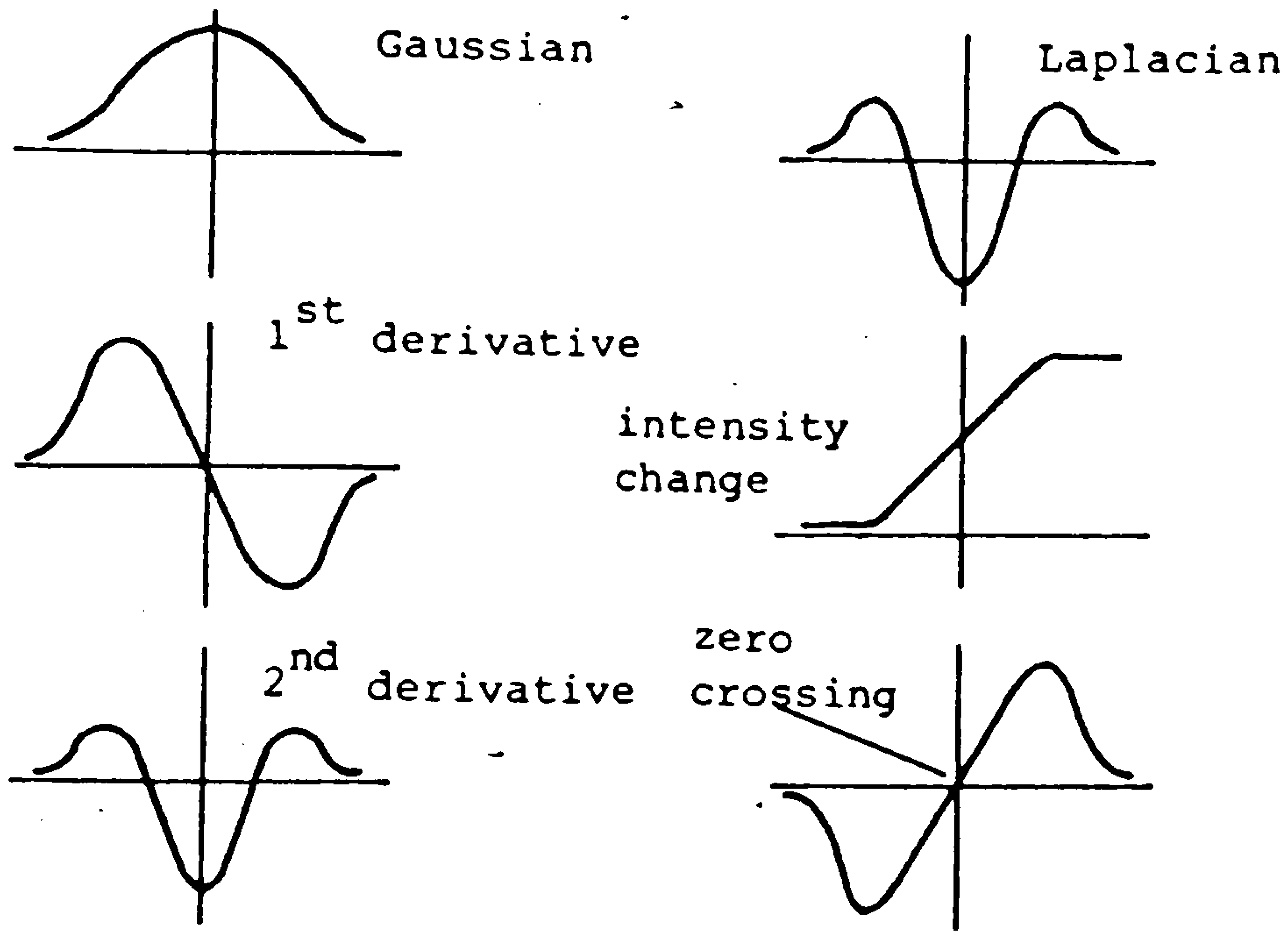


FIGURE 2.3

Further analysis proved that a good approximation to D^2G can be obtained by using an operator (named: 'Difference of Gaussians') that is created by the addition of a positive and a negative Gaussian-weighting function, with a variance ratio of about 1.6. The 3-D analogue of this curve is found by revolving it around the vertical axis, commonly termed the 'Mexican Hat'.

Averaging over a range of Gaussians gives rise to images with *high* or *low resolution* depending on whether the Gaussian is sharp or wide. This has the effect of interpreting edglets with no natural existence as physical boundaries. This is solved by considering the zero-crossing segment in a set of independent maps of Gaussian convolutions. If it has the same position and orientation, it indicates the presence of an intensity change due to a single physical phenomenon. This is known as the *spatial coincidence assumption* [Marr '82].

Another kind of edge detecting operators considers a widely used concept which is the basis of several other approaches, the *edge-template*. One of them [Kirsch '71] matches four separate templates with the image and reports the magnitude and direction associated with the maximum match.

Edge operator measurements can be improved by *relaxation* techniques. These are parallel-iterative techniques that use measurements of neighbouring edges in order to adjust the detected edges. The basic idea is to assign to every edge a *confidence* and then try to use recognizable local edge patterns which cause its confidence to be modified. The algorithm is repeated until the measurements of the edges converge or the possible edge is classified as 'no-edge' [Prager '80].

2.2.2 Stereo Vision

Stereoscopic vision is the feature of humans (and other living beings) to point their eyes in approximately the same direction, so that they get two slightly different views of the same object. The same result can be obtained by two cameras (or one camera from two positions), that provide the system with an input of two images which differ very little from one another. The advantage of stereo vision systems is that they can give the relative depth or the absolute 3-D location* of points that appear in both images. From the geometry of stereo vision (Fig.2.4) the formula that gives the depth of a point P is:

$$d = \frac{b\sin(a1)\sin(a2)}{\sin(\pi - a1 - a2)}$$

b is the distance between the centres of the two eyes and a1,a2 are the two angles of the triangle formed by the baseline defined by the two lines and point P.

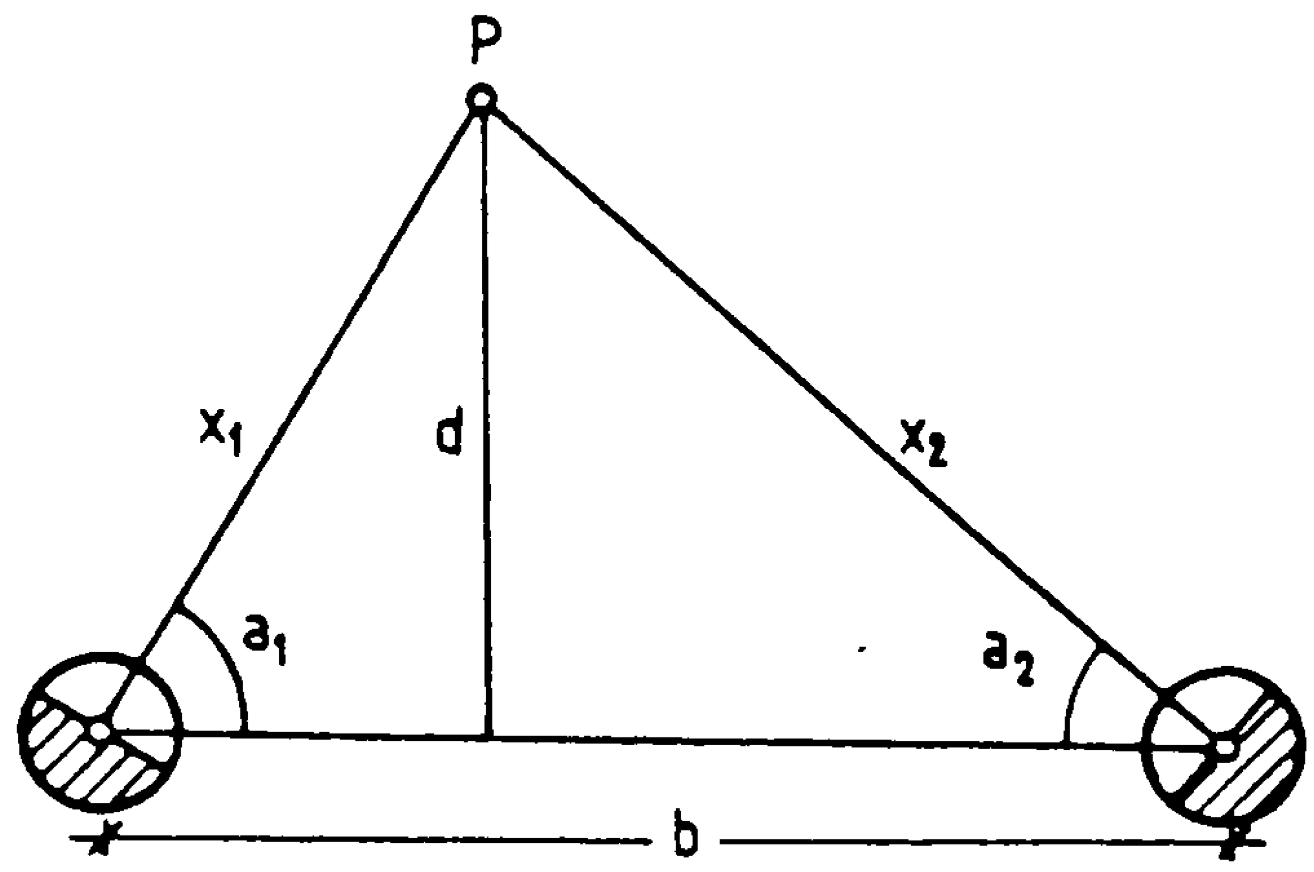


FIGURE 2.4

Angles a1 and a2 are computed by measuring the eye tilt angle t and the displacement x of P from the middle line of the stereo images (Fig.2.5):

$$a \simeq \arctan(\frac{f}{x} - t)$$

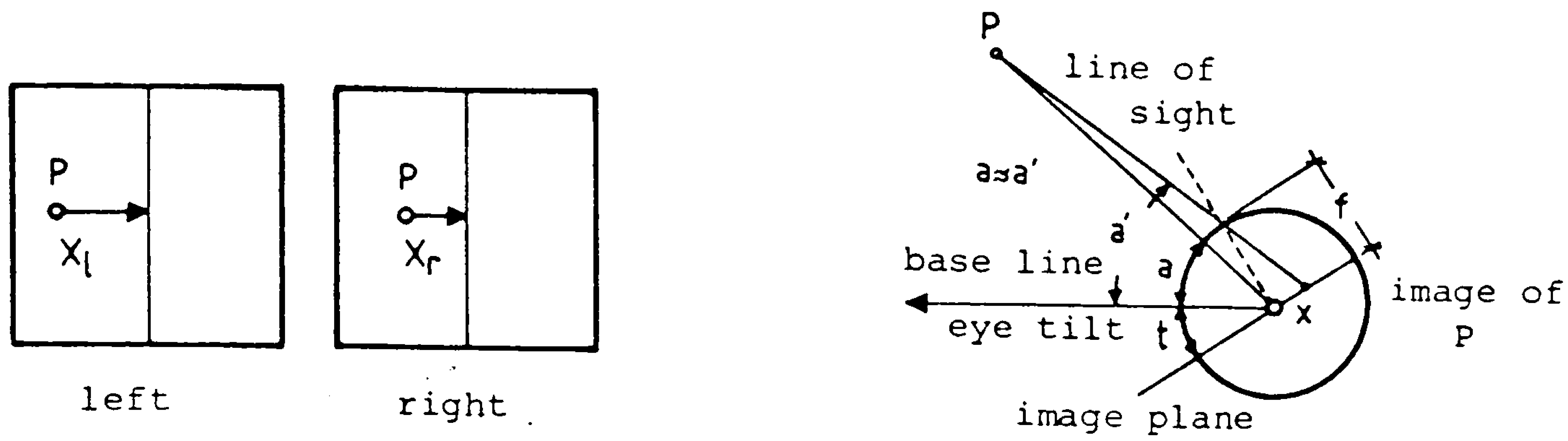
The displacement of P in one image from its counterpart in the other is called *stereo disparity* [Charniak & McDermott '85], and it is the key issue in the determination of depth. The idea of the various techniques [Hannah '74, Gennery '77] that attempt to measure the depth of points

---

**The third dimension can be also derived from monocular visual input using techniques like* *light striping* *[Popplestone et al '75],* *spot ranging* *and* *ultrasonic ranging* *[Ballard & Brown '82,c].*

in a scene is summarized in the following steps:

1. Take two images of the object, separated by a base line.



(a) (b)

FIGURE 2.5

2. Match points between the two images.

3. Derive the two lines that determine a point in the 3-D space (using e.g. triangulation) and from their intersection calculate its depth.

The most difficult part of the method is that of matching points in the two images. A solution to this problem is to use correlation, or template matching. A common point of the systems using correlation is the need for feature extraction that helps the matching process.

A stereo vision system [Burr & Chien '77] takes 3 images of a scene and computes edge orientation by preprocessing the centre picture to find edge chains, approximated by line segments. Stereo correlation, in the form of a mean-square difference of image intensity over 9×9 pixel windows, restricts the edges to single line segments in the corresponding view. Finally, the 3-D location of every node is determined by triangulation, while a separate program attempts to match this 3-D structure to a model. Another system [Baker & Binford '81] tackles the problem of depth-measuring by using an edge-based line-by-line stereo correlation scheme. It starts with extracting edge descriptions for a stereo pair of images and links these edges to their nearest neighbours to obtain the edge connectivity structure. The edge descriptions are correlated on the basis of local edge properties and a cooperative algorithm removes those edges which violate the connectivity structure of the two images. The result of the processing is a full image array disparity map of the viewed scene. Parallel matching is also used for feature-based stereo vision [Nishimoto & Shirai '85]. A disparity histogram for zero-crossings (ZC) is computed all over the image. The image is divided into small areas and a local disparity histogram for each local area is computed. A fusion evaluator detects the most probable disparity in each local area by examining the local disparity histograms. Then, disparity for all the finest ZC points are determined in the local area to obtain a high resolution disparity map. The matching pairs are removed from a set of ZC points and the process is iterated until no more disparities are determined.

On the other hand, if 3-D information is to be used in order to help perception, it is contradictory to use perception in order to do stereo. This conclusion, combined with the capability of humans to fuse *random-dot stereograms* and thus perceive 3-D shapes without having to recognize single cues in either image, led to other techniques. An algorithm [Marr & Poggio '76] computes disparity from random stereograms based on the following rules, which determine the appropriateness of a match:

1. To each point in an image may correspond only one depth value - *uniqueness*.

2. Neighbouring points are likely to have depth values near to each other since they are the result of smooth surfaces - *continuity*.

Matches are denoted by value 1 and non-matches by value 0. A matrix of the points of the two images that match is formed. The procedure produces a series of such matrices by modifying its points based on the idea that, alternative matches for a point inhibit each other while matches of equal depth reinforce each other. The initial matrix converges after a number of iterations when the number of modified points are below a certain threshold. The stereopsis algorithm has been refined to agree better with psychological data [Marr & Poggio '77].

Another new source of depth information is that of *focal gradients* resulting from the limited depth of field inherent in most optical systems [Pentland '85]. This method uses the fact that most lens systems produce images only parts of which are exactly focused. The distance of a point in the image is a function of the parameters of the lens system and a constant that determines the amount of defocus. This constant is measured by comparing sharp discontinuities in the image or by comparing two identical images with different depths of field (different aperture comparison). The advantage of this method is that it makes reliable measurements of depth avoiding the matching problems.

Stereo disparity can be used to obtain information about surface orientation, given information about the light source and the object's reflectivity. One algorithm introduces the concept of *gradient space* (gradient of surfaces) and uses multiple light source positions to remove ambiguity in surface orientation [Horn '75]. Another uses a single source and assigns initial ranges of orientations to surface elements on the basis of intensity. Neighbouring orientations are 'relaxed' against each other using certain constraints until each converges to a unique orientation [Ikeuchi & Horn '81].

It is frequently possible to recover depth from a stereo pair consisting of a conventional perspective (original) image and an orthographic (virtual) one [Start & Fischler '85]. The image is segmented into regions that can be described by a single underlying model. Finding such a model is a very difficult problem. The basic idea is to construct a virtual image independent of the actual shape of the imaged surface, based only on the model. The virtual camera model is normally an orthographic projection and the 3-D scene is constructed from eight point-correspondence between this and a perspective image. Depth is determined by triangulation.

Registration is a (usually costly) technique used in image matching for stereo vision and motion analysis. The method is summarized as: given two image functions $F(\vec{x})$ and $G(\vec{x})$, find a vector $\vec{h}$ that minimises the difference $F(\vec{x}+\vec{h})-G(\vec{x})$ in some region of interest R (Fig.2.6). An improved method [Lucas & Kanande '81] uses the *spatial intensity gradient* of images to find a good match by a type of Newton-Raphson iteration.

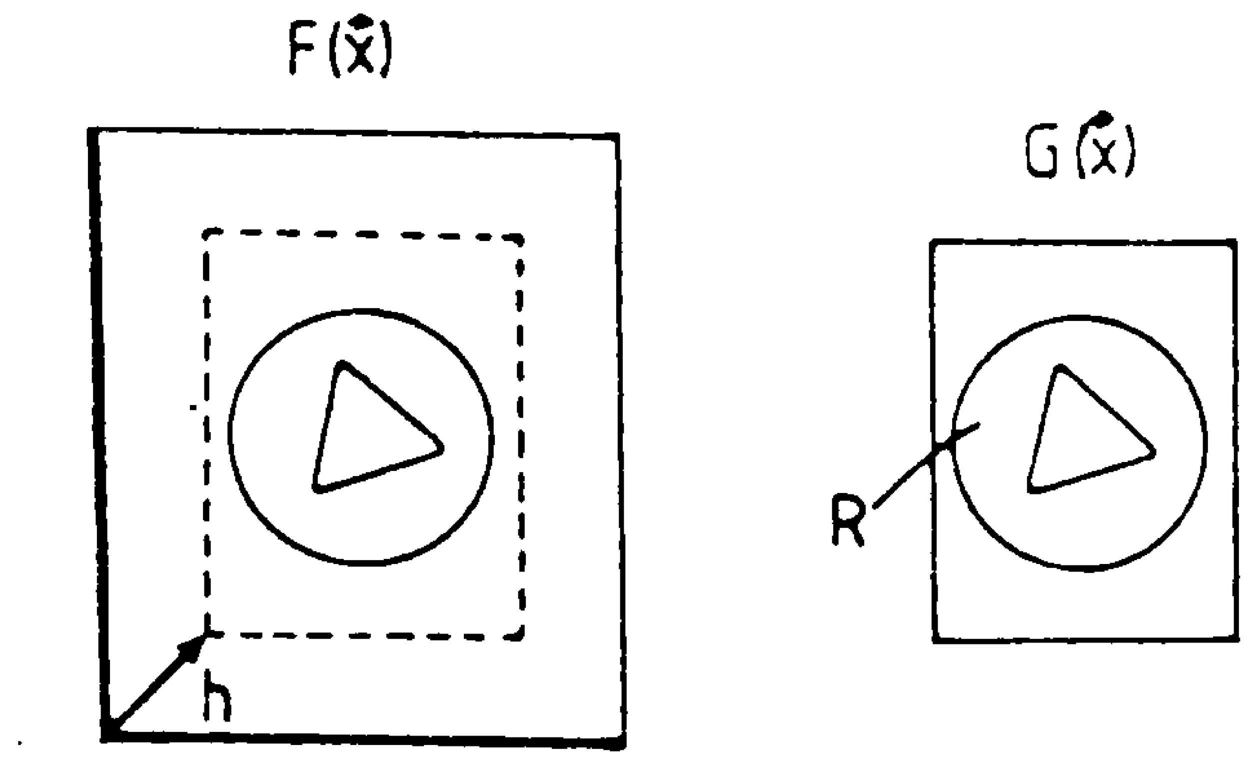


FIGURE 2.6

2.3 SEGMENTATION

Segmentation covers a number of techniques that organize parts of a generalized image into groups characterized by one or more special features. The idea of segmentation is based on grouping (Gestalt) principles arising from features such as similarity, proximity and continuity. Two important aspects of segmentation are: the data structure that is used to perform an efficient grouping of the features and the transformations leading to the computation of these features. The following paragraphs examine image segmentation with respect to *boundary and region tracking*, *texture* and *motion*.

2.3.1 Boundary and Region Tracking

Boundary is the real physical entity that separates two areas of different orientation *(folds)*, illumination *(shadows)* or reflectance *(marks)*. Boundaries are very important natural sorts of segments, that link the raw image data with their interpretation. Boundary-detecting techniques are based on strict or loose constraints about the likelihood of a certain grouping (i.e. knowledge about the image).

An approximately located boundary can be determined, by fitting an analytic curve through those discrete points that maximize an edge operator, applied along the perpendicular to the boundary at each point [Bolles '77]. A *Divide-and-Conquer* method can also be used to determine a low-curvature boundary between two points provided that the noise in the image is low [Selfridge et al '79]. The boundary between two points is searched along the perpendicular to the line joining the two points. The point of maximum magnitude becomes a *break point*, and the technique is applied recursively to the two line segments between the three known boundary points.

If the shape of a boundary can be described as a parametric curve, its location can be detected by the Hough technique [Ouda & Hart '72]. This computes the possible loci of reference points in parameter space from edge-point data in image space and increments the parameter points in an accumulator array. The technique has been generalized [Ballard '81] to cope with curves not expressed by a single analytic form.

Other methods for boundary detection form a *weighted graph* and look for the lowest-cost path between two nodes or use *dynamic programming* to obtain the'best boundary', together with a number of heuristics. If nothing about the shape of the boundary is known a *boundary follower* is used (best for images with little noise).

A *region* is a set of pixels in form of a uniform 2-D patch in the image, that might correspond to a world object or parts of one.

Determining regions is called *region-growing* and together with boundary detection are considered as the basic segmentation techniques.

Region growing techniques can be categorized into:

Local, where regions are formed on the basis of single-pixel properties and their direct neighbourhood. *Blob-colouring* is a local technique that uses an L-shaped template to assign each of the blobs in a gray image a different label ('colour it').

Global, where the grouping of pixels is based on large numbers of pixels throughout the image. A global approach divides the image pixels into either object or background using an appropriate threshold. This threshold is given by the minimum separating the two peaks of the bimodal histogram of gray levels in the image. [Zucker '76].

Splitting and merging, that are based on the feature of image-pixel homogeneity. The use of such an algorithm [Horowitz & Pavlidis '74], requires organization of the image pixels into a pyramidal grid structure where the regions form groups of four. Inhomogeneous groups are further divided into four groups, while homogeneous ones are merged into larger groups. *State space* techniques use graph structures to represent regions and boundaries.

The goal of boundary detecting techniques is to combine individual edge elements into meaningful boundaries of scene objects.

The goal of region growing techniques is to map small parts of an image with common features, to scene objects.

2.3.2 Texture

Texture is another important feature that contributes to the intrinsic image with information about surface orientation. There is no precise definition of texture but one can think of texture as a set of repeated elements closely related to each other that constitute an organized whole (e.g. a surface). The elements that make up a texture are called *texels* (for texture elements) and consist of *texture primitives*. Texels are characterized by certain invariant properties which occur repeatedly in different places, orientations and distortions inside a certain area.

The main problem that arises from texture is how to relate its primitives to the aim of recognizing and classifying it. The techniques that attempt to solve this problem can be divided into two broad categories. The *structural* model regards textures as composed of primitives forming a repeated pattern and describes such patterns in terms of a set of rules that generate them. The set of rules constitute a *grammar*. Texture grammars offer an infinite number of choices for rules and symbols and therefore are described as *syntactically and semantically ambiguous* [Zucker '76]. The various grammars are based on the size of the primitives resulting in *shape grammars* (high-level primitives or shapes), *tree grammars* and *array grammars* (texels≡pixels). The *statistical* model describes texture by statistical rules that govern the relation and distribution of gray levels. This method employs *pattern recognition* techniques [Tou & Gonzalez '74] in order to describe textures that do not have the geometrical regularity of the textures described by the structural model.

Human experience regards texture as a property of *surfaces*. Several techniques use *texture gradient* to determine surface orientation [Stevens '79]. The gradient is examined with respect to the direction of the greatest change in both the size and the spatial placement of primitives. The magnitude of the gradient combined with further geometrical information about the camera determine the 'tilt' and 'slant' of a plane. Finally other techniques are based on the shape of the texels themselves (e.g. vanishing points in a grid, circles appearing as ellipses, etc.).

2.3.3 Motion

Motion understanding is the part of computer vision that extracts low-level information using as input image sequences of moving objects. The techniques that are employed in order to solve the problem of motion are divided into two basic categories, depending on the major assumptions on which they are based. Those that consider motion as an extension of static vision, and those that regard motion itself as the basic characteristic that can provide cues for image segmentation. The techniques of the first category assume that motion vision is a problem of analysing very quickly a sequence of static images and then linking up the results by matching operations. This method makes an effective use of the techniques available in static vision but it needs large amounts of data and it is extremely complex. On the other hand, the methods of the second category seek new techniques to derive useful information, giving a primary role to motion as being closer to mechanisms of biological systems [Snyder '81]. Each category is successful in a certain number of problems and applications, although the current trend is towards the second one.

Biological visual systems perceive the imaged world using a very fine quantization, practically treated as a continuous flow of information across the retina called *optical flow*. Given an intensity function f(x,y,t), the following equation constrains optical flow:

$$-\frac{\partial f}{\partial t} = \vec{\nabla} f.\vec{u} \quad , \tag{2.1}$$

where $\vec{\nabla} f$ is the spatial gradient of the image and $\vec{u}(u,v)$ is the velocity [Ballard & Brown '82,a]. From (2.1) the optical flow can be determined by minimising the flow error using an iterative algorithm [Horn & Schunck '80].

For a moving observer in a given direction and for a certain direction of gaze, the projection of a world of static objects on the retina, appears to flow out of a particular point called the *focus of expansion*. The equation that gives the 'flow-path', considering a view direction along the positive Z axis and focal length f=1, is:

$$(x',y') = \left(\frac{x_0+ut}{z_0+wt} , \frac{y_0+vy}{z_0+wt}\right) \tag{2.2}$$

where $(u,v,w) = (\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt})$ are the components of velocity of word coordinates (x,y,z). Equation (2.2) gives the coordinates x',y' of the perspective projection of a point in initial position (x_0,y_0,z_0) on the plane of projection after a time interval t. For $t=-\infty$ (2.2) becomes:

$$(\frac{u}{w} , \frac{v}{w}) \tag{2.3}$$

which is the position of the focus of expansion on the image. Depth of field can also be derived from (2.2). Considering that the time that a point P takes to move from P_0 to P with velocity V(t), is the same on the image plane and in the real world then it is:

$$\frac{D(t)}{V(t)} = \frac{z(t)}{w(t)} \tag{2.4}$$

where D(t) is the distance of the point from the focus of expansion,

$z(t)$ the depth of the real point and $w(t)$ its velocity on the Z axis. If points of the same object move with the same velocity w, then knowing the depth of one of them, allows the depth of the others to be determined from:

$$z_2(t) = \frac{z_1(t)D_2(t)V_1(t)}{V_2(t)D_1(t)} \tag{2.5}$$

Several motion vision understanding techniques use optical flow as their basic input making the weakest possible assumptions about the world. They are therefore called *domain independent*. The study and derivation of equations that describe the motion of the observer from the optical flow can obtain information about surface orientation and edges. By inverting the equations that describe the observers' instantaneous velocity vector, a method for deriving relative depth is obtained [Prandzy '79]. This is used further, in order to obtain surface orientation by several relative depth measurements in a small area where the surface normal varies slowly. Another method [Clocksin '80] assumes a monocular observer with a spherical retina and obtains a formula for optical flow measurement in spherical coordinates. The two angles that determine the surface orientation are expressed in terms of the flow equation. Edges are detected by the fact that the Laplacian (see §2.2.1) of the flow measure and that of the depth have singularities at the same points.

A different approach to motion vision understanding uses a sequence of discrete static images as its input. The individual images are segmented into regions of interest (e.g. edges). Thus a description of each image is obtained. The descriptions are compared to detect similarities or differences and finally the results are used to form a description of the sequence as a whole. The problem of optical flow calculation is achieved by identifying discrete points different from their surrounding in each of two images at different times, and matching the corresponding points *(correspondence)*. One matching algorithm [Moravec '77] chooses match-point candidates by measuring the distinctness of local parts of the image from their surroundings for every frame, so obtaining two sets of points. Then, assuming that world points do not move large distances between frames, proceeds by iterating, allowing a maximum disparity between points. Another one [Barnard & Thompson '79] identifies first the candidate points and attaches match probabilities to the pairs of matching points. It then uses an iterative method, to determine whether a match is correct or incorrect from its high or low probability respectively.

Several methods based on the natural ability of the human visual system to interpret projections of moving 3-D objects as rigid objects attempt to do the same. It is possible to recover 3-D shape from three orthographic projections with established correspondences among at least four points [Ullman '79]. A 'polar equation' allows computation of shape when the motion of the scene is restricted to a rotation about the vertical axis and arbitrary translation. Further work [Nagel & Neumann '81] provides a compact system of three nonlinear equations for the unrestricted problem when five point correspondences between the two perspective images are known. A more recent method uses one orthographic and one perspective image whose internal imaging parameters may not be fully known [Start & Fischler '85] (see also §2.2.3).

Moving polygons and line drawings are also worth mentioning. The viewer is presented with a sequence of frames of line drawings. The goal is to segment the scenes into polygons and extract information about direction, motion, speed, etc. The idea is to extract an object in one frame, and search for it in the next frame. The basic techniques are a description-extraction process and a matching process. Matching can be performed by doing a similarity analysis first and then looking for differences between the objects in two frames. Another approach uses the above two processes in reverse order. An idea [Nagel '78] is to let the polygons move enough, so that they do not overlap in the two successive frames.

A number of approaches are human motion orientated. *Moving light displays* is a method that uses sequences of images which track only a small number of discrete points per frame (e.g. attach light sources to a person's major joints) and looks for point correspondence between the frames [Rashid '80]. Human motion can be studied by a program using a body model to simulate it ('bubble man' input). Knowledge of the imaging process provides constraints on the structure of the represented figure. A cooperative algorithm [Badler & Smoliar '79] reduces the uncertainty about part locations and their improved estimates are further propagated throughout the model.

Correspondence is always a difficult problem in motion vision. A method [Kanatani '85] succeeds in detecting 3-D structure and motion without using correspondence. A two-stage procedure extracts first the *flow parameters* which completely characterize the viewed motion for each planar region of the object by measuring 'features' of the image. Then the structure and motion are computed from the flow parameters, and the solution is given in the form of analytical expressions.

An interesting idea [Asada & Tsuji '85] projects a stripe pattern onto a time-varying scene to find moving objects and acquire scene features in the consecutive frames for estimating 3-D motion parameters. First a 2 ½ -D representation of moving objects is obtained for every frame, by estimating surface normals from the slopes and intervals of stripes in the image. Then the image is further divided into planar surfaces by examining the distribution of the surface normals in the gradient space. Finally, the rotational motion parameters of the objects are estimated from changes in the geometry of these surfaces between frames.

2.4 LATE VISION

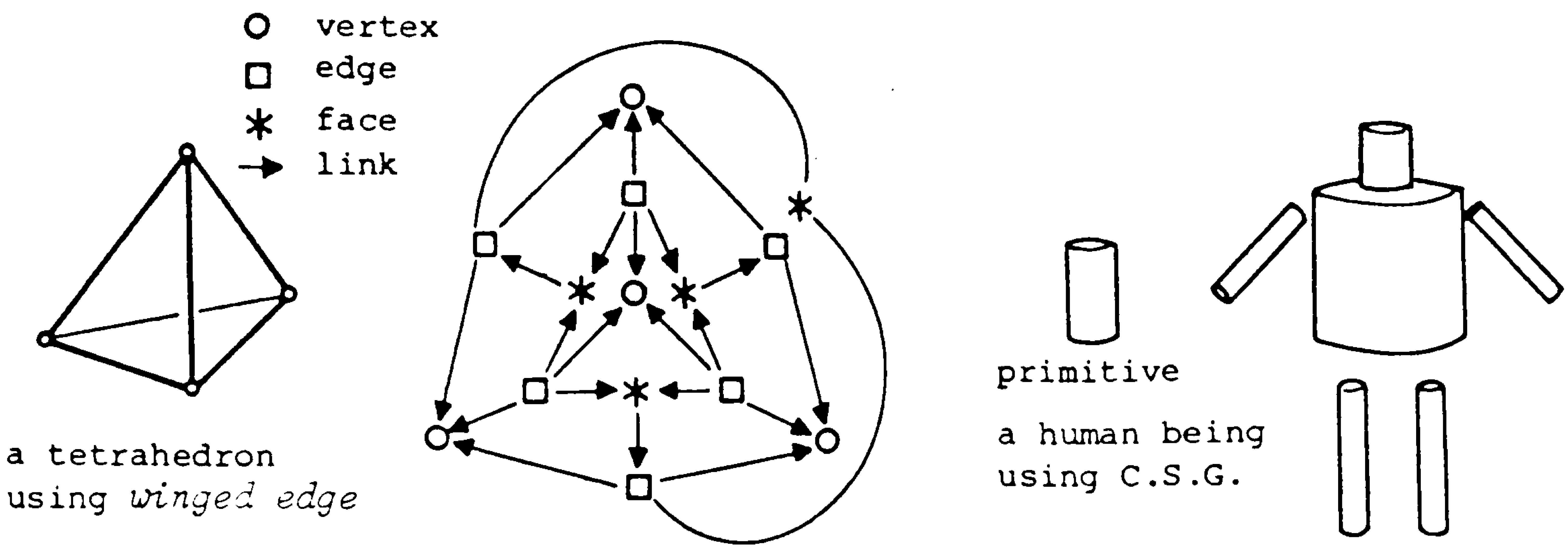
The low-level phase is completed by calculating the intrinsic image. The main problem of the high-level (or late) vision phase is to produce a description of an object using as input its intrinsic image. The solution consists of a *representation* stage which switches from a region-oriented representation to an object oriented one, and a *recognition* stage which is to determine the shape of the object in the image.

2.4.1 Representation of Shapes

The shape of an object can be defined as the set of points that compose its outline or external boundary. In the 2-D space this boundary is a (closed) line, while in the 3-D space it is its surface. 2-D shapes are boundaries (1-D) and regions (2-D). Some of the most common representations for boundaries are: *polylines*, *chain codes*,

ψ-*s curves*, *Fourier descriptions* and *strip trees*, while for regions they are: *spatial occupancy arrays*, *y-Axis* and *medial axis*.

Rigid objects representations are based on their enclosing surface, their enclosed volume or line drawings. *Surface* representations are: the *winged edge* polyhedron representation with primitives like vertices, edges, faces and polyhedra connected in a pointer relational structure (Fig.2.7a), *splines* used to represent four-sided patches approximating surfaces, and *Fourier descriptors* as in the 2-D case. The representation of generalized cylinders (or cones) is used for solids that are regarded as products of a *translational* or *rotational sweep*. A generalised cylinder is a solid whose axis is a 3-D space curve at any point of which a cross section is defined. *Volumetric* representations use *ray-casting* to depict objects on a raster plane and can use either *spatial occupancy* or *cell decomposition*. *Constructive solid geometry* represents solids as composed by a set of basic primitives (e.g. cylinders Fig. 2.7b, [Marr & Nashihara '78]), via a set of operations. *Line-drawing* representations are groups of points connected with one another with straight lines. (See also §3.2).



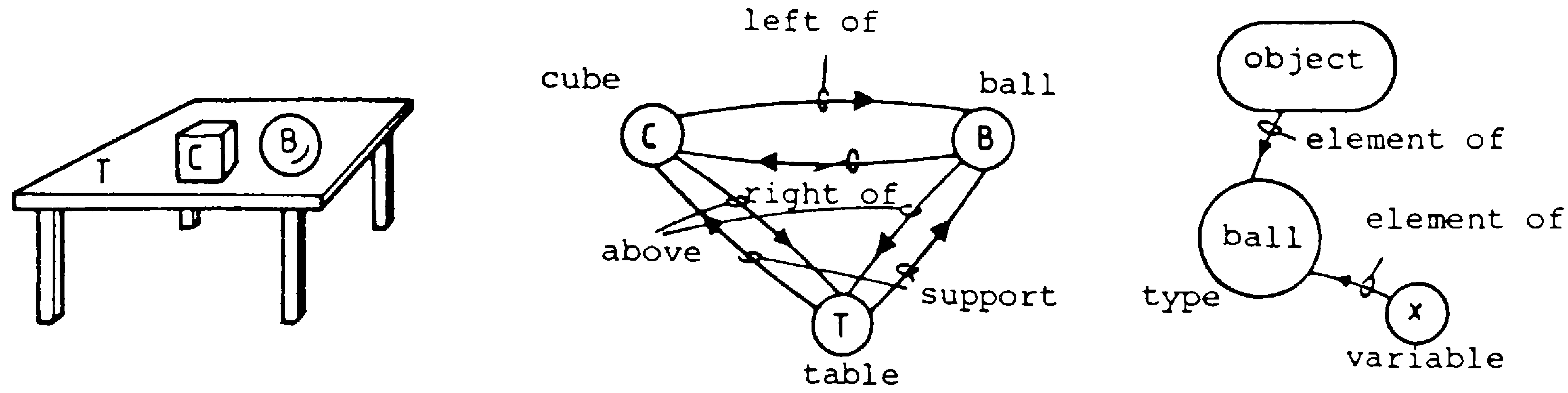
(a) (b)

FIGURE 2.7

2.4.2 Recognition

Recognition is the set of processes that attempt to interpret the viewed scene according to their experience. It is the link that relates an image to its real word representative, based on certain models which are products of the system's knowledge. Right from the beginning of the recognition phase, it is obviously important to have an abstract representation of the world useful to the system. This is called a *knowledge base*. Some of the basic properties of a knowledge base are: the capability to represent different sorts of structures and convert between them, quick access to information, extensibility and allowing for inference and planning. Knowledge base can be *analogical* and *propositional*. Analogical representations are coherent (one element for one represented situation), continuous, reflect exactly the relational structure of the situation and can be simulated. Propositional representations allow one element to represent several situations, are discrete, abstract and are manipulated by computations that use inference. Low-level representations are purely analogical, while high-level ones can be both analogical or propositional. In computer implementations it is often possible to convert one kind of representation to the other without loss of information.

A data structure that accesses both analogical and propositional representations is that of *semantic nets*. A semantic net or network (first introduced by [Quillian '68]) is a graph-like structure of *nodes* representing objects and *arcs* that represent relationships between the objects (Fig.2.8). Nodes may represent general concepts *(types)*, specific instances of them *(tokens)*, or *variables*. Special nodes are the *virtual* and the *default value* nodes. Nearby nodes usually represent



scene A

semantic net for scene A

FIGURE 2.8

relevant objects or objects involved in the same computation *(indexing property)*. Arcs represent two-argument relations between nodes. Complex relations may be represented by nodes. Relevant nodes or arcs may be dealt with as a unit using labelled delimiters called *partitions*. Finally *conversions* are used in order to transform analogical into propositional representations and vice versa. Two special cases of semantic nets are the *frame system* [Minsky '75] and the *location networks* [Russell '79].

The basic mechanism that leads to recognition is *matching*. The goal of matching is to obtain a meaningful interpretation of input data that corresponds to a world scene, by finding their association to computer represented models. Semantic nets are a useful representation for matching *relational structures*. Some algorithms *(graph-theoretic)* use graph-like representations for relational structures and use their properties to perform matching. Some of the techniques used by graph-theoretic algorithms are: *graph isomorphism*, 'measure of goodness' *(matching metric)*, *backtrack search*, *association graphs* [Ballard & Brown '82,e; Rich '83].

Three more basic concepts employed by modern recognition systems are: *inference*, *control* and *planning*. Inference is a process that deduces facts from other known facts. *Classical inference* is expressed mainly by *predicate logic*, a system that derives consequences of facts from propositions (see also Appendix 3). *Extended inference* is useful for implementation in automated systems including: *production systems* (rewriting rules of the form 'situation-action' used to control computational activities), *labelling schemes* (using mathematical optimization and probabilities to interpret entities) and *active knowledge* (treating pieces of knowledge as program procedures). The function of *control* in computer vision is to check the correct flow of information and activity throughout the different representations. There are two basic control strategies: the *bottom-up* (or image data driven) and the *top-down* (or *goal-driven*). Nevertheless, most actual systems use *mixed* top-down and bottom-up control. *Planning* represents (simulates) world states with actions and looks for the best sequence of actions that solves the problem. A tree formation can be used for the states with branches resulting from different actions. The planner proceeds by minimising the cost assigned to the actions.

2.4.3 Recognition Systems in Practice

This last paragraph includes some interesting recognition systems developed over the last decade.

A program [Tenenbaum '73] that recognizes complex objects uses model descriptions based on distinguishing features of the objects. Recognition is based on knowledge about the systems perceptual abilities and contextual knowledge about the world situation. A particular object is 'found' by searching a sampled image for characteristic features *(acquisition stage)* and analysing the results in more detail *(validation stage)*. Based on the same principles is the system by [Nevatia & Binford '77] that builds model descriptions of a set of objects (toy dolls, horses and hand tools) using only range data as its input. Semantic nets represent connectivity relations, parts (described by linear cones), junctions and parts with special properties *(distinguished pieces)*. The recognition stage looks for distinguished pieces and matches them to their counterparts in the stored models. A scheme by [Baker '77] argues that model building is complementary to recognition, since the former describes models of the present environment, while the latter associates experiences of the present with those of the past. It also considers 3-D modelling as very important and obtains its models by analysing individual images of a scene. It first extracts contour descriptions based upon irregularity measures. Then, it takes sequential pairs of such descriptions, correlates them and constructs meshed networks representing their shapes.

Primary concern of the system by [Freuder '77], recognizing objects (hammers) in visual scenes, is the control of the recognition process. This use of knowledge to direct control *(active knowledge)* is based upon partial results obtained on the scene. The system uses a special descriptive formalism and control structure, that can apply knowledge actively to the processing of a given scene. An analogous approach is considered by [Ballard et al '77] in an attempt to locate objects (e.g. ribs) in particular contexts (chest radiograms). An executive (program) examines semantic net descriptions to determine the most useful procedure in the context. [Shneier '78] does not use models for individual objects but a global database which is a semantic net. The user must provide the name of the model and recognition proceeds by interpreting, first fragments of the scene, and then the scene as a whole.

A system that recognizes stacked objects with planar and curved faces using an input of range data [Oshima & Shirai '81], works in two stages. In the first stage, it learns the description of objects which are used as object models for the recognition (second) stage. The same description is used for both models and scene objects and a matching process performs the recognition. A similar method is followed by [Zdrahal '81] who uses relational structures to describe his models.

An *extended Gaussian image* (EGI) is used by [Ikeuchi '81] for interpreting 2½-D representations for recognition of 3-D objects. The EGI is constructed by mapping each surface normal of an object to the Gaussian sphere. A number of constraints reduces the possible viewing directions and a matching function applied to EGI of a candidate set makes the final decision. An EGI is independent of both the position of the origin and the scale of axes of the coordinate system. The system proposes an algorithm for reconstruction of the original shape of a convex polyhedron from its EGI. An interesting approach [York et al '81] employs cubic B-splines and Coons surface patches for matching 3-D object models against 2-D object descriptions.

[Sabbah '81] proposes a visual recognition system that uses relaxation in a *conceptual hierarchy* defined as an active semantic network of computing units logically partitioned into abstract levels). Each level is defined by a set of basic parameters for recognizing the features associated with the level. Communication along the units is obtained by relations imposed by world constraints. A parallel formulation allows the pruning of the search tree dynamically by incorporating results from partial computations, and thus cutting down the search for a correct match. The method is goal directed and suitable for low and intermediate level vision. The 3-D model building system (for objects with cylinder-like body) from several 2-D views, by [Abe et al. '83] establishes a linguistic communication with the user about ambiguous labels attached to subparts.

[Matsuyama & Hwang '85] developed a system (SIGMA) for image understanding consisting of three experts: Geometric Reasoning Expert (GRE) for spatial reasoning, Model Selection Expert (MSE) for appearance model selection, and Low Level Vision Expert (LLVE) for knowledge-based picture processing. The control mechanism for the GRE integrates bottom-up (establish relations between objects) and top-down (find certain objects) analyses into a unified reasoning process.

Arguing that geometric frames provide a natural way of talking about shapes [Ballard & Tanaka '85] developed particular constraints for polyhedra, that lead to algorithms for extracting frame information from a scene and matching it against stored prototypes. *Frame primitives* are geometric coordinate frames that can be extracted from more primitive image features. Matching a 3-D prototype to an image is hierarchically organized into:

a) Recovery of 3-D lines from stereo image data.

b) Construction of a 3-D polyhedral scene model.

c) Matching positions of that model to a library of stored prototypes.

This strategy has advantages over the methods that perform matching in one step. It also carries out matching by a parallel probabilistic relaxation algorithm. A system by [Allen & Bajcsy '85], integrates vision and tactile sensing in a robotic environment to perform object recognition tasks. It uses multiple sensor systems to compute 3-D primitives that can be matched against a model database of complex curved surface objects containing holes and cavities. It starts by detecting sensory data and proceeds by invoking an object model, globally consistent with the sensed data, which it verifies by further sensing at several levels.

[Lowe '87]'s computer vision system recognizes 3-D objects from unknown view points in single 2-D images without the use of any depth information. Three basic mechanisms bridge-up the gap between the 2-D image and knowledge of 3-D objects:

a) A process of perceptual organization searches the image for groups or structures invariant over a wide range of view points.

b) A probabilistic ranking method reduces the size of the search space during model-based matching.

c) A process of spatial correspondence brings the projections of 3-D models into direct correspondence with the image by solving for unknown view point and model parameters. It also argues that human vision is based on similar mechanisms and constraints.

Computer vision systems are faced with the problem of receiving an image (input) and attempting to interpret it (output). This is achieved by a number of *low-level* and *high-level* processes and a range of representations relating the input to the output.

At low-level, an *intrinsic image*, containing important physical information about a scene is extracted from a *generalized image*, via a series of *image processing techniques*. The intrinsic image is subjected to *segmentation* to form a set of elements that are likely to be associated with meaningful objects, using *boundary detection*, *region growing*, *texture* and *motion analysis*. A *geometric image* may also be obtained at this level, to represent the all important concept of *shape*, and to be used in *scene simulation* or for producing *matching models*. Low-level processes can be implemented with *parallel* computation, and they use purely analogical *representations*.

High-level processes use *relational models*, such as *semantic nets*, to represent prior knowledge and past experience. *Recognition* is achieved by *matching* the database produced in the low-level phase with computer models, in order to obtain a meaningful interpretation. *Propositional* representations are made up of assertions that are true or false with respect to a model, and are manipulated by *inference* methods. *Planning* and *control* are two other important functions at this level. High-level processes are (basically) implemented with *serial* computation, and they use a *mixture* of analogical and propositional representations.

REFERENCES

1. ABE, N., ITHO, F. and TSUJI, S. 1983: *Toward Generation of 3-D Models of Objects Using 2-D Figures and Explanations in Language*, Proc. 8th IJACI, pp. 1113-1115.

2. ALLEN, P. and BAJCSY, R. 1985: *Object Recognition Using Vision and Touch*, Proc. 9th IJCAI,pp. 1131-1137.

3. ASADA, M. and TSUJI, S. 1985: *Utilization of a Stripe Pattern for Dynamic Scene Analysis*, Proc. 9th IJCAI, pp. 895-897.

4. BADLER, N.I. and SMOLIAR, S.W. 1979: *Digital Representations of Human Movements*, Computer Surveys 11, 1, March, pp. 19-38.

5. BAKER, H.H, 1977: *Three-dimensional Modelling*, Proc. 5th IJCAI, pp. 649-655.

6. BAKER, H.H. and BINFORD, T.O. 1981: *Depth from Edge and Intensity Based Stereo*, Proc. 7th IJCAI, pp. 631-636.

7. BALLARD, D.H. 1981: *Generating the Hough Transform to Detect Arbitrary Shapes*, Pattern Recognition 13, 2, pp. 111-122.

8. BALLARD, D.H. and BROWN, C.M. 1982: *Computer Vision*, Prentice-Hall, a: p.2, b: pp.42-52, c: pp.52-56, d: pp.102-103, e: pp.355-380.

9. BALLARD, D.H. and TANAKA, H. 1985: *Transformational Form Perception in 3-D: Constraints, Algorithms, Implementation*, Proc. 9th IJCAI, pp. 664-670.

10. BALLARD, D.H., BROWN, C.M. and FELDMAN, J.A. 1977: *An Approach to Knowledge-directed Image Analysis*, Proc. 5th IJCAI, pp. 664-670.

11. BARNARD, S.T. and THOMPSON, W.B. 1979: *Disparity Analysis of Images*, Technical Report 79-1, Comp.Scien.Dept., Univ. Minnesota, January.

12. BARROW, H.G. and TENENBAUM, J.M. 1981: *Computational Vision*, Proc. IEEE 69, 5, May, pp. 572-595.

13. BOLLES, R. 1977: *Verification Vision for Programming Assembly*, Proc. 5th IJCAI, pp. 569-575.

14. BRADY, M. 1986: *Machine Vision: The Advent of Intelligent Robots*, Addison-Wesley, pp. 7-66.

15. BURR, D.J. and CHIEN, R.T. 1977: *A System for Stereo Vision with Geometric Models*, Proc. 5th IJCAI, p. 583.

16. CHARNIAK, E. and McDERMOTT, D. 1985: *Introduction to Artificial Intelligence*, Addison-Wesley, pp. 114-118.

17. CLOKSIN, W.F. 1980: *Computer Prediction of Visual Thresholds for Surface Slant and Edge Detection from Optical Flow Fields*, Ph.D. Dissertation, Univ. Edinburgh.

18. DUDA, R.O. and HART, P.E. 1972: *Use of the Hough Transformation to Detect Lines and Curves in Pictures*, Commun. ACM 15, 1, January, pp. 11-15.

19. FREUDER, E.C. 1977: *A Computer System for Visual Recognition Using Active Knowledge*, Proc. 5th IJCAI, pp. 671-677.

20. GENNERY, D.W. 1977: *A System for Stereo Computer Vision with Geometric Models*, Proc. 5th IJCAI, pp. 576-582.

21. GEORGIOU, C.J. and ANASTASSIOU, D. 1986: *An Architecture for a Single Chip Performing Real-Time Image Convolution*, Image Processing and Its Applications, IEE 2nd International Con., June, pp. 107-111.

22. GONZALEZ, R.C. and WINTZ, P. 1977: *Image Processing*, Addison-Wesley, p. 58.

23. HANNAH, M.J. 1974: *Computer Matching of Areas of Stereo Images*, Stanford AI Memo AIM-239, Stanford Univ., July.

24. HORN, B.K.P., 1985: *Shape from Shading*, in PCV.

25. HORN, B.K.P. and SCHUNK, B.G. 1980: *Determining Optical Flow*, AI Memo 572, AI Lab., MIT, April.

26. HOROWITZ, S.L. and PAVLIDIS, T. 1974: *Picture Segmentation by a Digital Split-and-Merge Procedure*, Proc. 2nd IJCPR, August, pp. 424-433.

27. IKEUCHI, K. 1981: *Recognition of 3-D Objects Using the Extended Gaussian Image*, Proc. 7th IJCAI, pp. 595-600.

28. IKEUCHI, K. and HORN, B.K.P. 1981: *Numerical Shape from Shading and Occluding Boundaries*, AI 16, Special Issue on Vision.

29. KANATANI, K. 1985: *Structure from Motion Without Correspondence: General Principle*, Proc. 9th IJCAI, pp. 886-888.

30. KIRSCH, R.A. 1971: *Computer Determination of the Constituent Structure of Biological Images*, Computers and Biomedical Research 4, 3, June, pp. 315-328.

31. LOWE, D.G. 1987: *3-D Object Recognition from Single 2-D Images*, AI, 31, March, pp. 355-395.

32. LUCAS, B.D. and KANADE, T. 1981: *An Iterative Image Registration Technique with an Application to Stereo Vision*, Proc. 7th IJCAI, pp. 674-679.

33. MARR, D. 1976: *Early Processing of Visual Information*, Phil.Trans. Roy.Soc.Lond. B., 275, pp. 483-524.

34. MARR, D. 1982: *Vision*, Freeman, San Fransisco, p. 70.

35. MARR, D. and POGGIO, T. 1976: *Cooperative Computation of Stereo Disparity*, Science 194, pp. 123-134.

36. MARR, D. and POGGIO, T. 1977: *A Theory of Human Stereo Vision*, AI Memo, 451, AI Lab., MIT, November.

37. MARR, D. and NISHIHARA, H.K. 1978: *Representation and Recognition of Spatial Organization of 3-D Shapes*, Proc.Roy.Soc.Lond. B, 200, pp. 269-294.

38. MARR, D. and HILDRETH, E. 1980: *Theory of Edge Detection*, Proc.Roy. Soc.Lond. B, 207, pp. 187-217.

39. MATSUYAMA, T. and HWANG, V. 1985: *Sigma: A Framework for Image Understanding - Integration of Bottom-up and Top-Down Analyses*, Proc. 9th IJCAI, pp. 908-915.

40. MAYHEW, J. and FRISBY, J. 1984: *Computer Vision*, Artificial Intelligence, Eds: O'Shea, Eisenstadt, Harper-Row, p. 305.

41. MINSKY, M.L. 1975: *A Framework for Representing Knowledge*, in PCV.

42. MORAVEC, H.P. 1977: *Towards Automatic Visual Obstacle Avoidance*, Proc. 5th IJCAI, p. 584.

43. NAGEL, H. 1978: *Analysis Techniques for Image Sequences*, Proc. 4th IJCPR, November, pp. 186-211.

44. NAGEL, H. and NEUMANN, B. 1981: *On 3-D Reconstruction from Two Perspective Views*, Proc. 7th IJCAI, pp. 661-663.

45. NEVATIA, R. and BINFORD, T.O. 1977: *Description and Recognition of Surved Objects*, AI 8, pp. 77-98.

46. NISHIMOTO, Y. and SHIRAI, Y. 1985: *A Parallel Matching Algorithm for Stereo Vision*, Proc. 9th IJCAI, pp. 977-980.

47. OSHIMA, M. and SHIRAI, Y. 1981: *Object Recognition Using 3-D Information*, Proc. 7th IJCAI, pp. 601-606.

48. OTHA, Y. 1985: *Knowledge-based Interpretation of Outdoor Natural Scenes*, Research Notes in AI, 4, Pitman Advanced Publishing Program.

49. PENTLAND, A.P. 1985: *A New Scene for Depth of Field*, Proc. 9th IJCAI, pp. 988-993.

50. POPPLESTONE, R.J., BROWN, C.M., AMBLER, A.P. and CRAWFORD, G.F. 1979: *Forming Models of Plane-and-Cylinder Faceted Bodies from Light Stripes*, Proc. 4th IJCAI, pp. 664-668.

51. PRAGER, J.M. 1980: *Extracting and Labelling Boundary Segments in Natural Scenes*, IEEE Trans. PAMI 2, 1, January, pp. 16-27.

52. PRANZY, K. 1979: *Egomotion and Relative Depth Map from Optical Flow*, Comp.Scien.Dept., Univ. Essex, March.

53. QUILLIAN, M.R. 1968: *Semantic Memory*, in Semantic Information Processing, Ed: Minsky, Cambridge, MA, MIT Press.

54. RASHID, R.F. 1980: *LIGHTS: A System for Interpretation of Moving Light Displays*, Ph.D. Diss., Comp.Scien.Dept., Univ. Rochester, April.

55. RICH, E. 1983: *Artificial Intelligence*, McGraw-Hill, pp. 135-242.

56. ROBERTS, L.G. 1965: *Machine Perception of 3-D Solids*, Optical and Electro-optical Information Processing, Ed: Tippett et al, Cambridge, MA, MIT Press.

57. RUSSELL, D.M. 1979: *Where Do I Look Now?*, Proc. PRIR, August, pp. 175-183.

58. SABBAH, D. 1981: *Design of a High Parallel Visual Recognition System*, Proc. 7th IJCAI, pp. 722-727.

59. SELFRIDGE, P.G., PREWITT, J.M.S., DYER, C.R. and KANADE, S. 1979: *Segmentation Algorithms for Abdominal Computerized Tomography Scans*, Proc. 3rd COMPSAC, November, pp. 571-577.

60. SHNEIER, M.D. 1978: *Object Representation and Recognition in Machine Vision*, Ph.D. Thesis, Edinburgh.

61. SNYDER, W.E. 1981: *Computer Analysis of Time Varying Images*, IEEE Computer 14, 8, August.

62. START, T.M. and FISCHLER, M.A. 1985: *One-eyed Stereo: A General Approach to Modelling 3-D Scene Geometry*, Proc. 9th IJCAI, pp. 937-943.

63. TENENBAUM, J.M. 1973: *On Locating Objects by Their Distinguishing Features in Multi-sensory Images*, SRI Tech. Note No. 84, September.

64. TOU, J.T. and GONZALEZ, R.C. 1979: *Pattern Recognition Principles*, Reading, MA, Addison-Wesley.

65. ULLMAN, S. 1979: *The Interpretation of Visual Motion*, MIT Press, Cambridge, Mass.

66. YORK, B.W., HANSON, A.R. and RISEMAN, E.M. 1981: *3-D Object Representation and Matching with B-Splines and Surface Patches*, Proc. 7th IJCAI, pp. 648-650.

67. ZDRAHAL, Z. 1981: *A Structural Method of Scene Analysis*, Proc. 7th IJCAI, pp. 680-682.

68. ZUCKER, S.W. 1976: *Toward a Model of Texture*, CGIP 5, 2, June, pp. 190-202.

69. ZUCKER, S.W. 1976: *Region Growing:Childhood and Adolescence*, CGIP 5, 3, September, pp. 382-399.

CHAPTER 3

THE 3-D FIGURE SIMULATOR

3.1 INTRODUCTION

In the last chapter a survey of vision systems was presented. This chapter examines the methods of interpreting line drawings, which represent simple polyhedra or, in the more general case, simply curved objects.

The digitized picture of an object (or a scene of objects) is subjected to a number of image processing procedures (e.g. averaging, edging, isolation [Gonzalez & Wintz '77]) and as a result of this, the basic features comprising the picture are extracted. These features are characterised as *figure primitives* and are the key to the subsequent phase of recognition. In the case of polyhedrals the figure primitives consist of a list of lines and points which correspond to the edges and the vertices of the solid objects. A *line drawing* is a representation of a 3-D object in 2-D, as a group of points connected to one another with straight lines. One could envisage line drawings as wire frames where, the particular way in which they are connected carries information about the planes comprising the solid objects [Dixon '77]. Line drawings are structures, that often carry a lot of ambiguity, when they are used to represent 3-D objects and it takes experience, training and knowledge of physics in order to obtain a good interpretation. For example, Figure 3.1a shows a 3-D

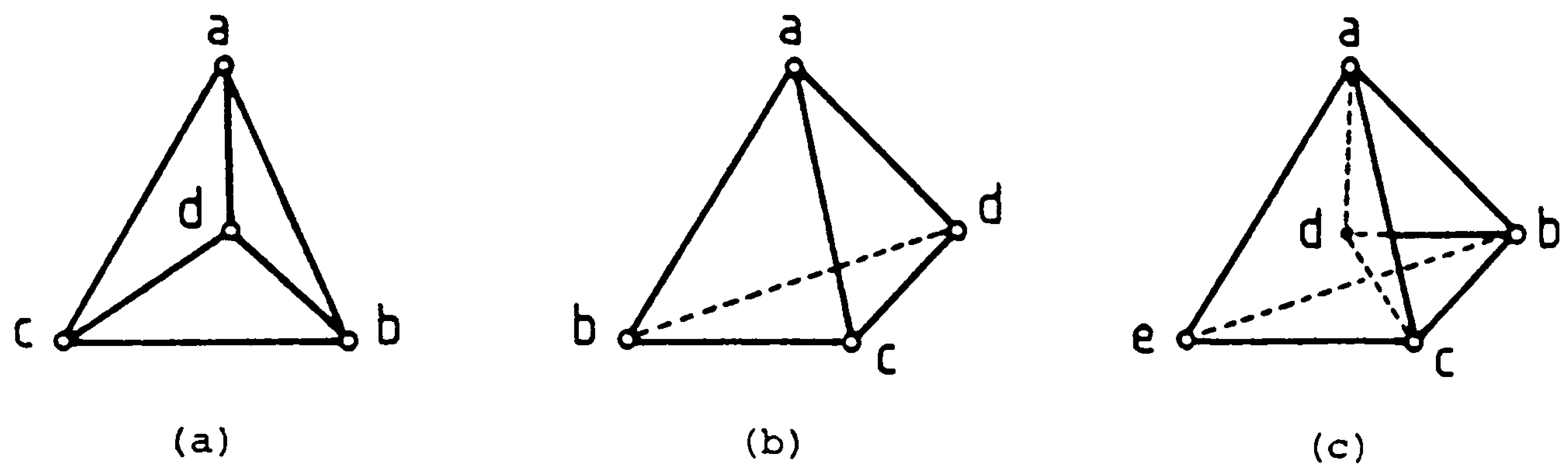


FIGURE 3.1

figure with an ambiguous vertex d. A rotation of 90° about the Z axis can reveal whether its vertex d is convex (3.1b) or non-convex (3.1c). The first part of this chapter examines line drawings closer and presents a survey of previous work on them.

A reasonable line drawing representation requires adequate hardware (camera, high resolution digitizer, etc.) and software (routines for smoothing, thinning, etc.). A successful way round these two basic conditions, is to simulate wireframe objects and use their 2-D projections as input to the recognizer. These 2-D projections must be realistic and therefore hidden lines and face occlusion should be taken into consideration. The second part of this chapter describes a 3-D wireframe image simulator.

3.2 LINE DRAWINGS - A SURVEY

In science and engineering (but also in everyday life), line drawings have been the main medium of conveying information about 3-D objects. It has been shown [Shapira '74] that, there is a maximum number of projections that are required to determine a particular object unambiguously. This number depends on the number of edges in the object. However, a construction of a wireframe object by a given number of projections, may have an ambiguous result.

Despite a certain degree of ambiguity, basically due to inadequate low level operators in the image processing phase, there are several reasons why line drawings were a natural target from the early days of computer vision. Here are some of the reasons:

a) They are represented in an exact and clear way, free from noise and consequences due to poor visual processing.

b) They carry direct information about surfaces of polyhedral objects.

c) They appeal to the natural tendency of human beings to relate objects with their contours, therefore their interpretation problem is approachable.

d) They can be easily simulated. That means that they can be studied separately and independently from the phases of preprocessing and visual processing.

The first computer programs in computer vision can be traced to the middle '60s. These were chiefly heuristic programs, that used ad-hoc techniques often limited only to the specific problem in hand. At the beginning of the '70s, the programs were based on a broader theoretical background and became more flexible and versatile, capable of coping with more general problems.

3.2.1 Roberts' Program

Roberts' program, which appeared in 1965 tries to describe block-scenes, such as Fig.3.2a, in terms of unions of transformed primitive blocks (also called models) [Roberts '65]. A scene includes both simple and compound polyhedral blocks. A simple polyhedral block is defined as an instance of a transformed primitive, where a transformation may involve translation, scaling and rotation. The set of primitives consists of a *cube*, a *wedge* and a *hexagonal prism*, shown in Fig.3.2b. Compound polyhedral blocks are regarded as those, that can

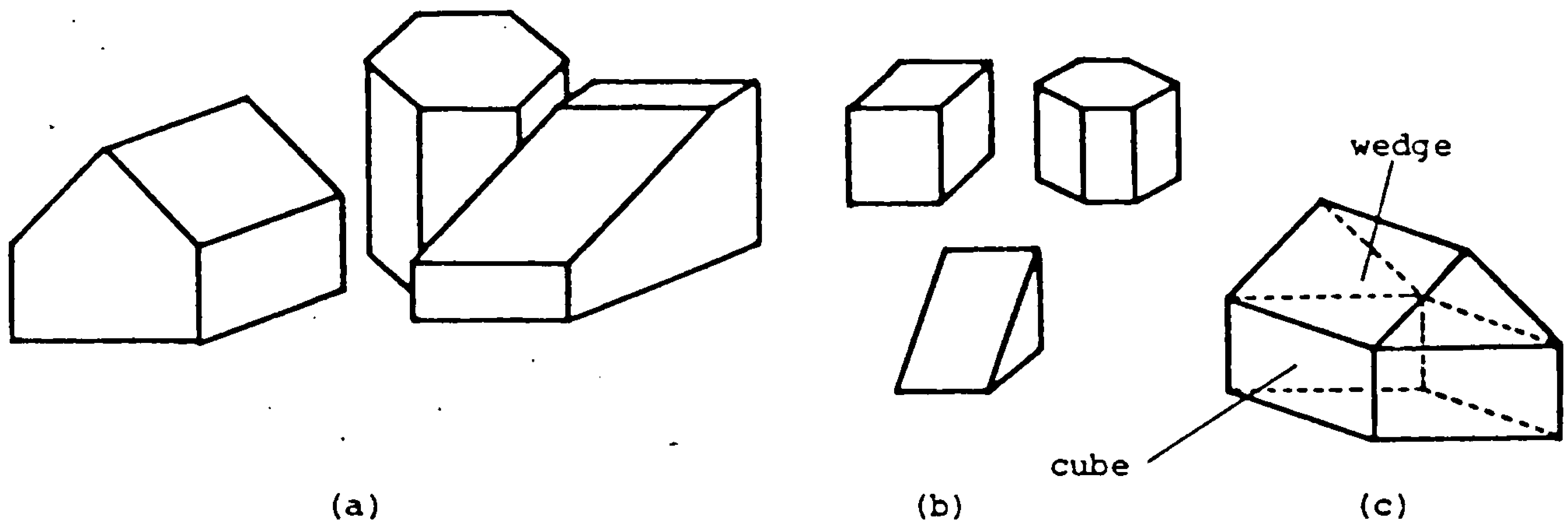


FIGURE 3.2

be decomposed into simple ones. Fig.3.2c shows a compound polyhedron, consisting of a cube and a wedge 'glued together'. The system can be divided into two independent parts:

a) producing a line drawing from a photograph

b) producing a 3-D object list from a line drawing.

The program achieves the following tasks. First, it looks for all the primitives that have been used to construct the scene. Then, it derives all the transformations which have been applied on the list of primitives comprising the scene. Finally, it is able to produce a line drawing of the scene from any viewpoint, based on its previously derived description.

A complete line drawing, in the form of lists of lines and end-points, provides the input for the modelling/recognition program. Each end-point has pointers indicating which lines it is connected to in order of angle. The first step is to find the polygons which make up the surfaces of objects, by tracking lines and jumping to adjacent lines at junctions. The starting point in the modelling process, consists of polygons defined as *approved*. These are convex polygons, with 3,4 or 6 sides and without any *T-junction* vertices, because these occur in the 3 primitive blocks and thus are the only *legal* projections onto the line drawings. The line drawings are searched for a number of topological features, which are the basis for the transformation of primitive blocks. If a transformed model completely fits a group of connected lines then the model is assumed to represent the object. If not a compound object construction procedure is used. According to this, when a transformed primitive block is met in the scene, it is removed from it and a new scene is derived by filling in the missing lines. The new *subscene* is analysed in the same way, until the whole of the original scene has been examined.

Roberts' program, despite its limitations and deficiencies, is very important, because it is the first serious work in the study of polyhedral scene analysis.

3.2.2 Guzman's Program

In 1969 appeared a program by [Guzman '69], which used line drawings of polyhedral scenes (sometimes quite complicated as in Fig. 3.3) as input and tried to interpret them as 3-D scenes. The basic principle of the program is to group the lines, that divide the scene into polygonal regions, into sets and interpret each set as a polyhedral block. The grouping of the polygons is done by examining the scene for local evidence. Regions that may belong to the same body are connected with *links*. Links converge to vertices, which are classified into types according to the shape of the polyhedral angle they form. Each type always preserves the same number of links; the background is completely isolated from the scene and thus no links connect it with the main scene region. Figure 3.4 shows the set of basic links used by Guzman.

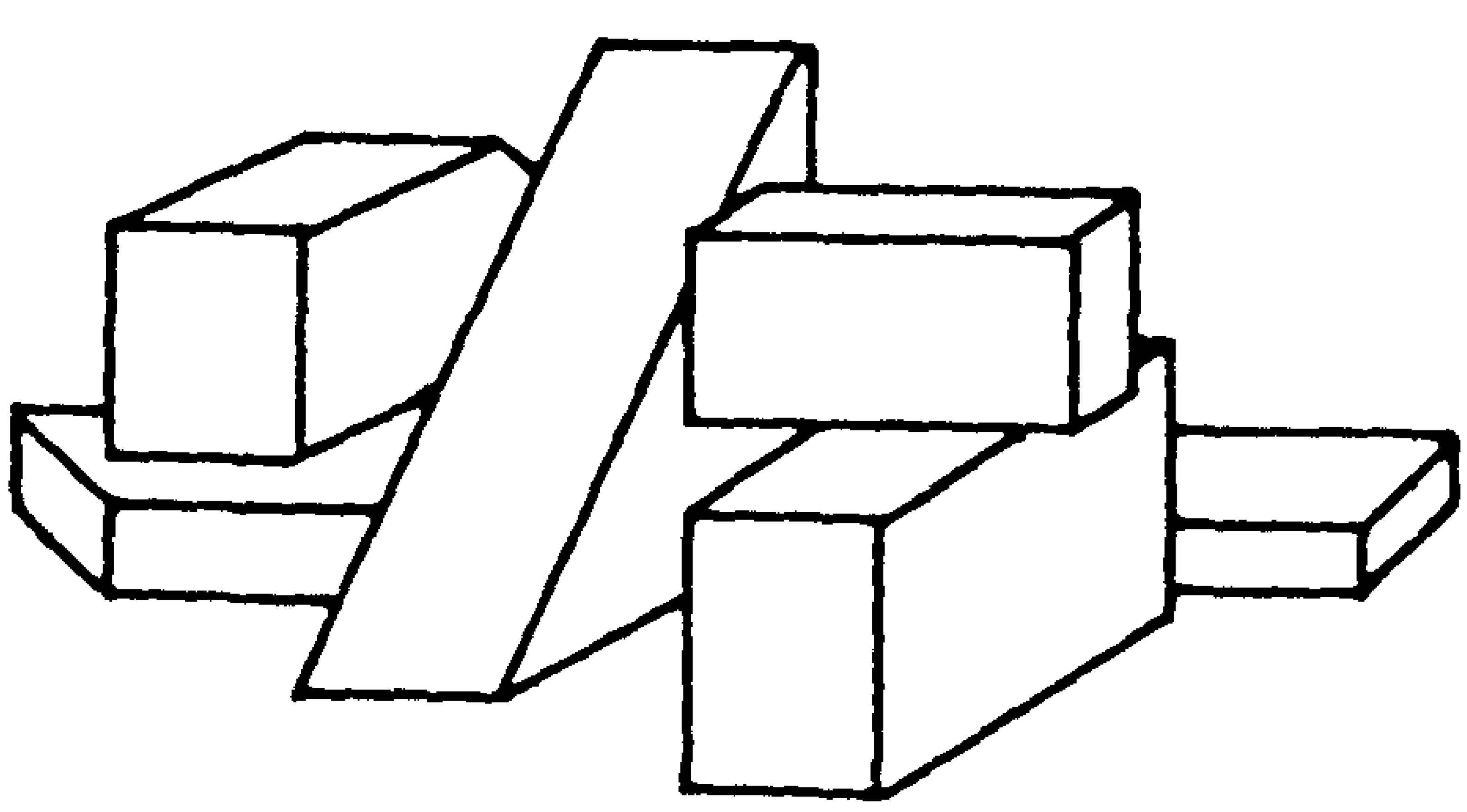


FIGURE 3.3

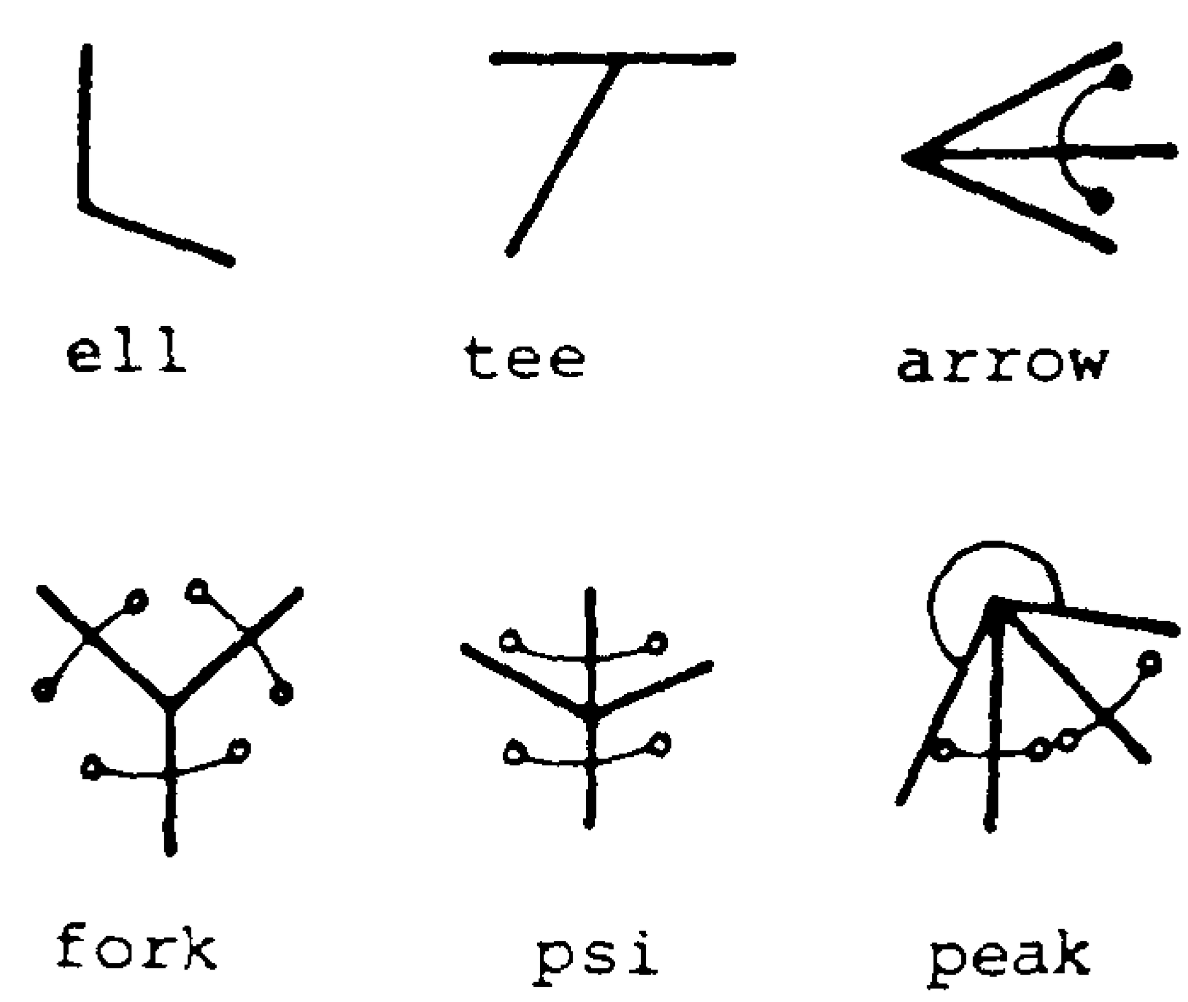


FIGURE 3.4

The rules that determine the grouping of the lines as well as the regions that belong to the same body are often complex, they include *ad-hoc* conditions and they use several exceptions (i.e. *inhibitory links*). In general the program is successful on scenes with convex isolated trihedral polyhedra and it performs reasonably well on a considerable number of scenes. However it does not give a satisfactory answer when alignment is present, it is not based on general rules and it does not have a physical explanation about the nature of the links. Regardless of these disadvantages the Guzman program led to the next phase of line drawing interpretation, the *line labelling*.

3.2.3 Line Labelling

In 1971 [Huffman '71] and [Clowes '71], working independently, developed a system which coped with scenes similar to those referred to in the last paragraph. Their approach was different from Guzman's, in that they tried to interpret the lines of the scene (basically object edges), by labelling each one of them. Figure 3.5 shows the labels that they used on a wedge resting on a planar surface. With a '+' are labelled lines which correspond to convex edges of the object (e.g. ad, dc, de). A '-' is used for edges corresponding to concave edges (e.g. bc, ec, fe). Edges belonging to faces which occlude other faces are marked with '>'. The direction of the arrow is such, that has always the occluding surface on the right of the line and the occluded one on its left. In case of illumination Clowes used arrows perpendicular to the shadow boundaries and pointing to the inside of it.

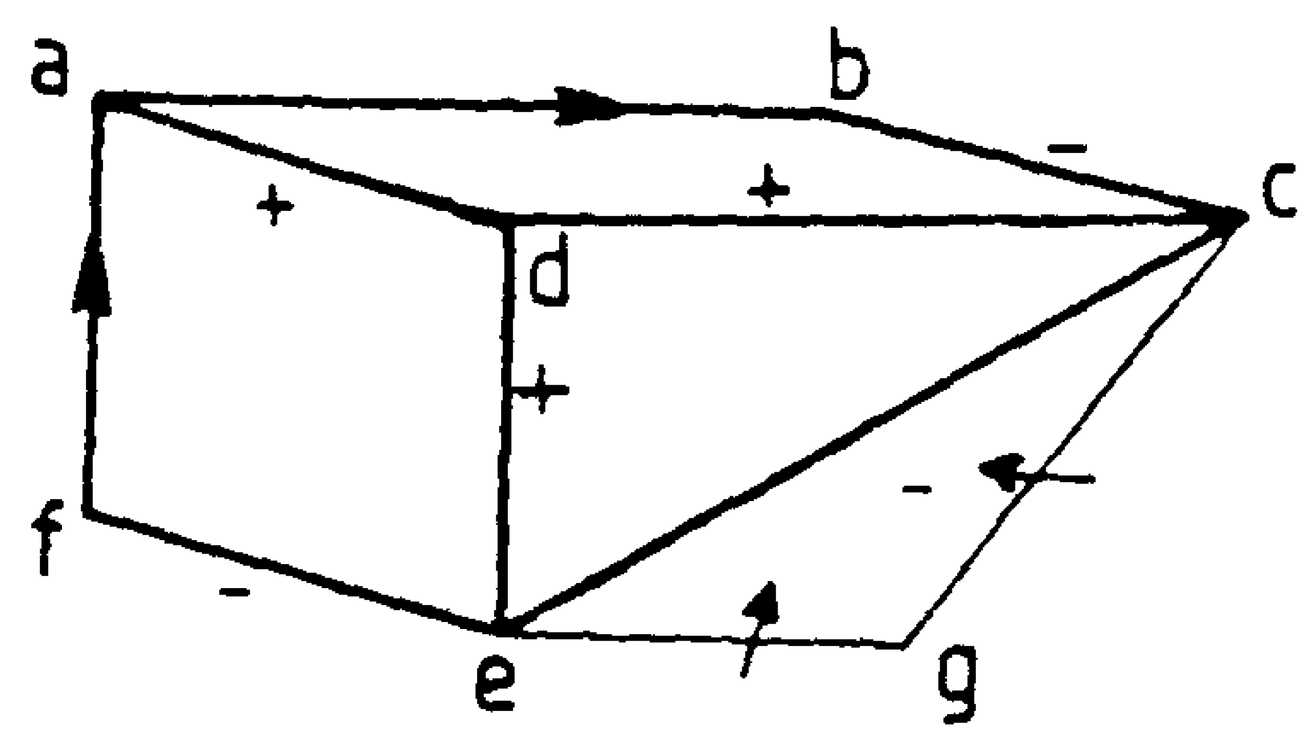


FIGURE 3.5

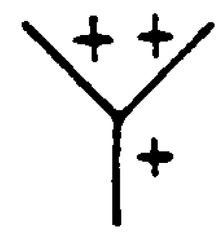
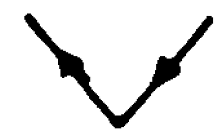
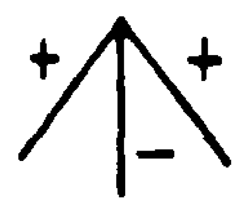
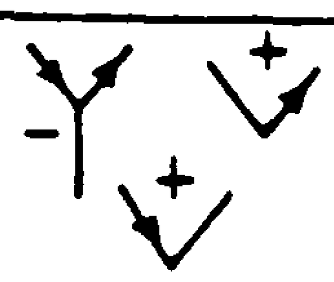
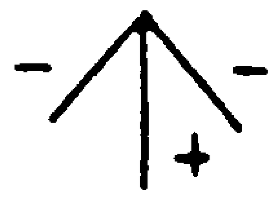
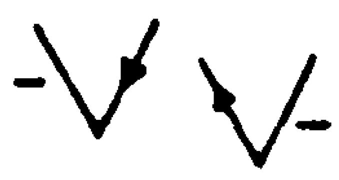
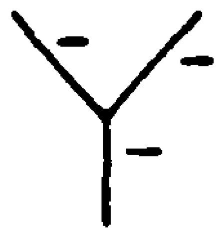
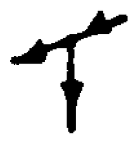
Visible surface / Octants filled	3	2	1	0
1				–
3				–
5			–	–
7		–	–	–
Occlusion				

TABLE 3.1

After a systematic investigation, Huffman and Clowes, noticed that three planes forming a trihedral corner divide the 3-D space into eight sections called *octants*. Every trihedral corner, that belongs to a solid object, is characterized by the number of octants filled by the matter of the object around it and can be classified by it. Close examination of all the possible trihedral corners, seen from all possible angles, showed that only some of the vertex types (discussed in §3.2.2) and a few of the possible line labellings can really occur. Table 3.1 is a catalogue of all possible vertices, in a world of trihedral polyhedra. This catalogue can be easily extended to include more corner types.

Taking into account that for every line there are four possible labels (+, -, >, <), then there are theoretically 64 ($=4^3$) possible labels for the fork, the arrow and the 'T' and 16 ($=4^2$) possible labels for the 'L'. However, a closer study of scenes with trihedral objects shows that only a small fraction of the possible labels can actually occur. Thus, according to the catalogue in Table 3.1, only 3/64, 3/64, 4/64 and 6/16 possible line labels, for the above corner types occur. If this is combined with the *coherence rule*, namely:

'no line may change its label between vertices',

the number of possible interpretations for a scene is restricted even more. Figure 3.6 shows an example of a line drawing that fails the label coherence rule.

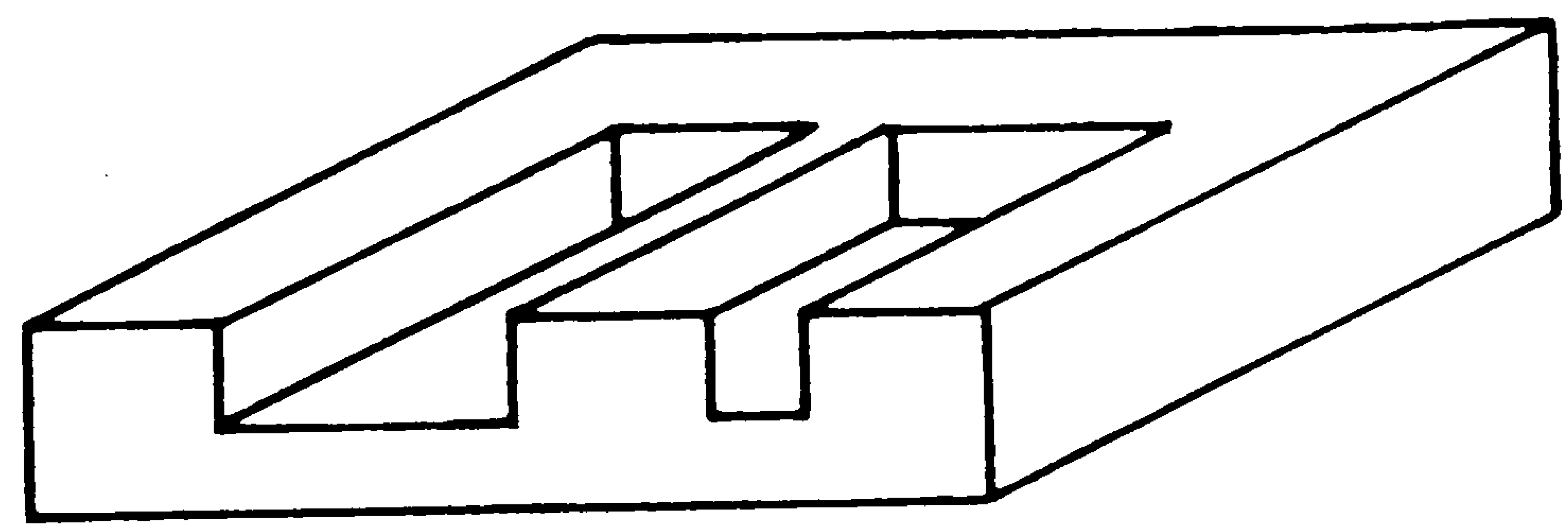


FIGURE 3.6

A comparison between the line labelling and Guzman's region grouping system shows that in an environment of convex polyhedra, both systems are based on the same geometric and physical principles. The main difference is that each labelled line carries information about the objects in which it belongs (e.g. a convex edge separates two faces of the same object; a concave one is evidence of different objects), while Guzman needs two vertices in order to decide whether the two faces converging to an edge are linked with one another or not.

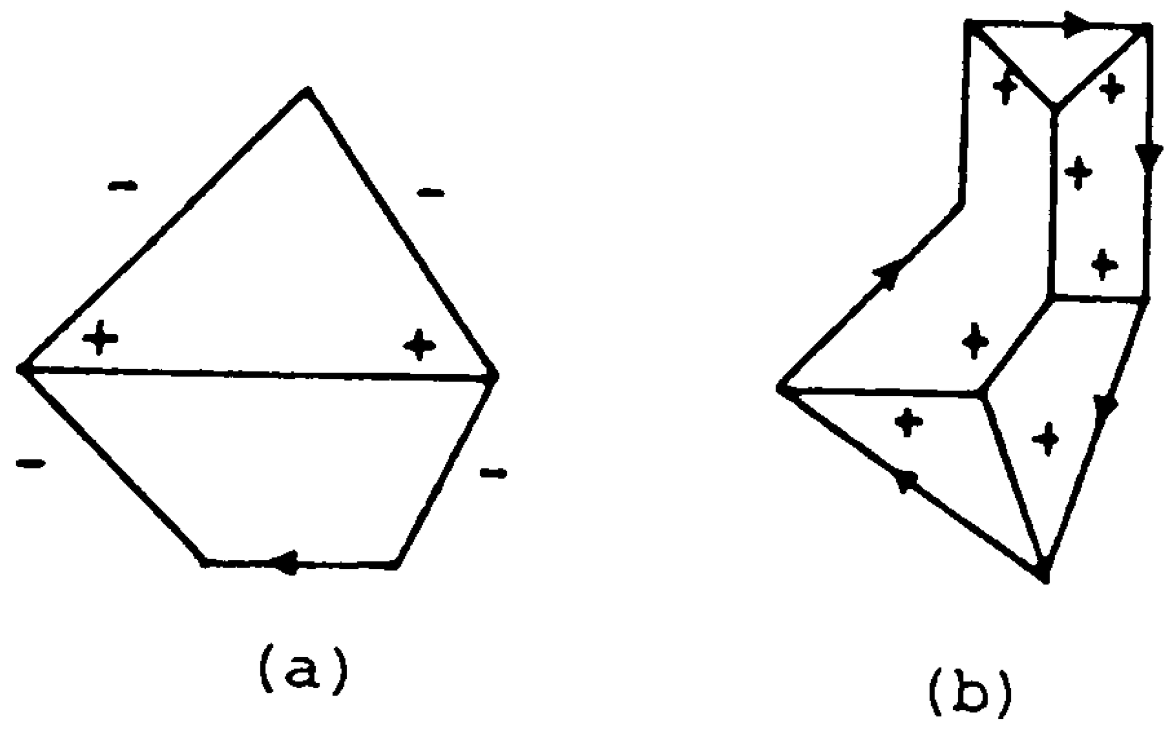


FIGURE 3.7

Finally the coherence rule substitutes Guzman's uncertainty and heuristics in the case of link disagreement. Scene labelling is more successful and consistent than the previous scheme but has its own problems in that: geometrically inconsistent scenes are labelled correctly and/or geometrically legal scenes are given impossible labels. (Fig. 3.7).

[Waltz '75] introduced some new ideas and improved the labelling system so that it could cope with scenes like the one in Figure 3.8. He included shadows and three illumination codes for each face on the side of an edge, he separated the objects in the scene at cracks and concave edges, and finally he increased the number of the possible vertex types. The result of these extensions was to bring up the number of line labels as well as the possible vertex labellings. Thus, he increased the information coded in each line and reduced drastically the percentage of geometrically meaningful labels for a vertex.

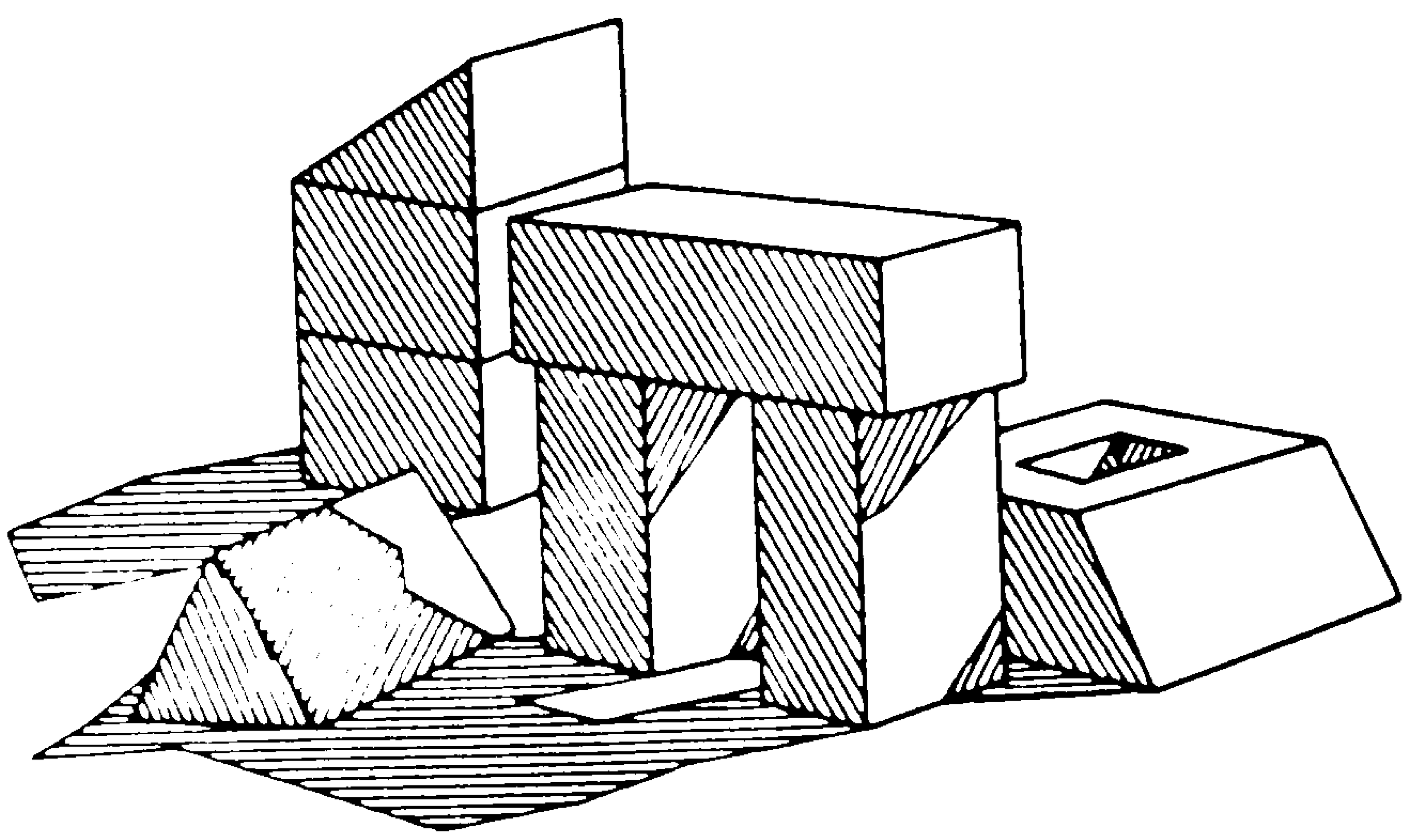


FIGURE 3.8

Another interesting extension by Waltz was the introduction of a *constraint-propagation* algorithm. Every vertex is labelled and certain constraints arise from the fact that this labelling must be consistent with the labellings of neighbouring vertices. These constraints are propagated back so that the labelled vertices in a group remain consistent with each other while the number of possible labellings is reduced. The algorithm continues recursively until all the vertices of the scene have been labelled. This continuous elimination of impossible labellings often leads to a unique correct solution. In case of multiple solutions, explicit labellings are obtained through a tree search or further constraints. The problem of missing lines is solved by adding the vertices that result if lines are missing from legal vertices, to the legal vertex catalogue.

The line-labelling scheme can be further extended to cope with objects made up of planar faces (*origami world* [Kanade '78]), and simply curved objects [Turner '74, Chakravarty '79].

3.2.4 Other Techniques for Line Drawing Interpretation

The line-labelling programs had certain problems in interpreting objects that are not planar polyhedra. This gave rise to a new approach, which is based on plane orientation and geometric coherence [Mackworth '73, Sugihara '81].

The program relies on the relation between orthographic projections of planar scenes and their gradients. Every line L in the scene is the result of the intersection of two planes Π_1 and Π_2 (Fig.3.9). It can be shown that line G, defined by the gradient vectors of Π_1 and Π_2 in the gradient space, is perpendicular to L, i.e. L constrains the gradients of Π_1 and Π_2. Lines are further characterized as 'connect' or 'occluding' according to the kind of edge that they come from. It is also possible to determine whether connected edges are convex or concave and for occluding ones, which surface is in front.

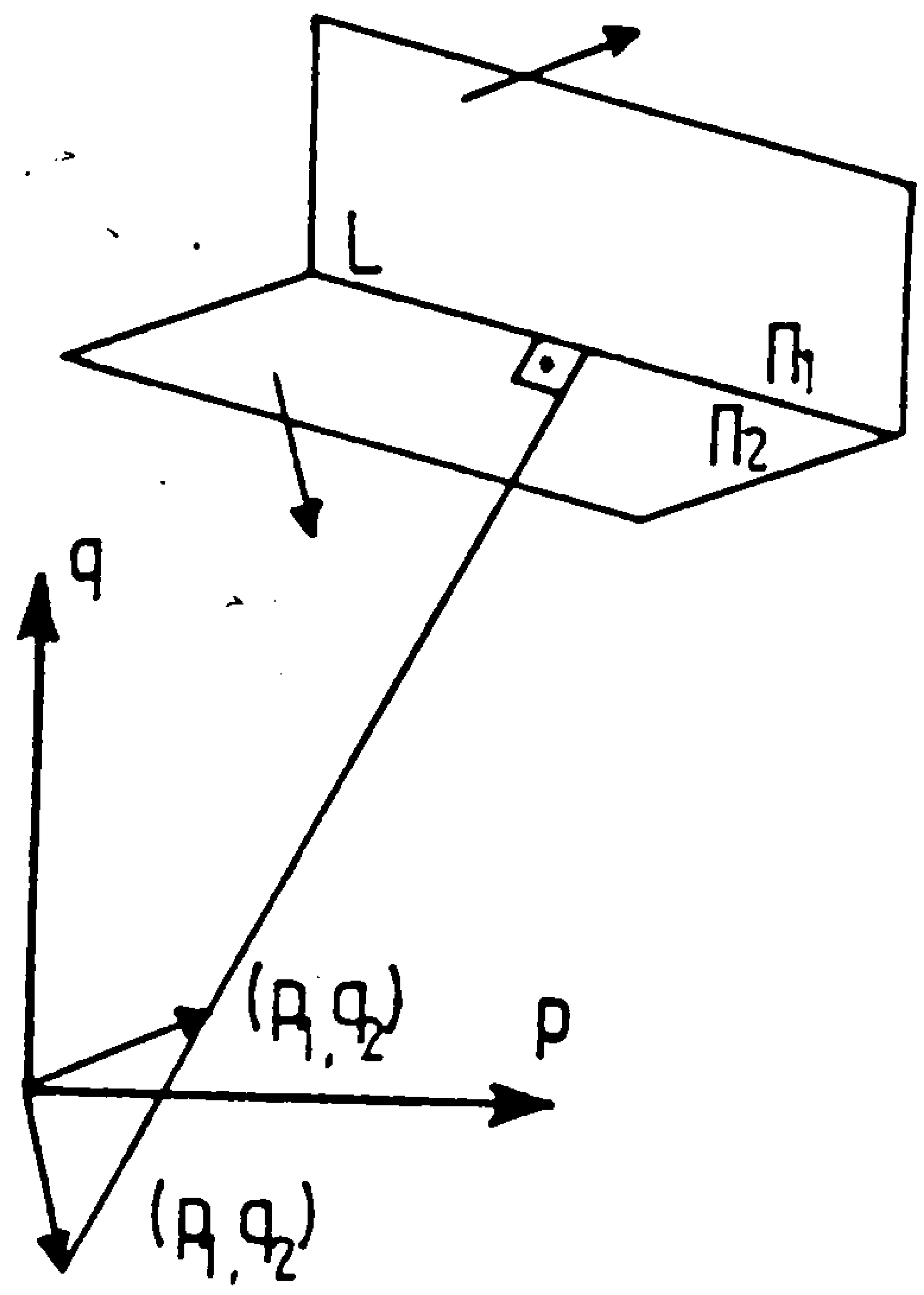


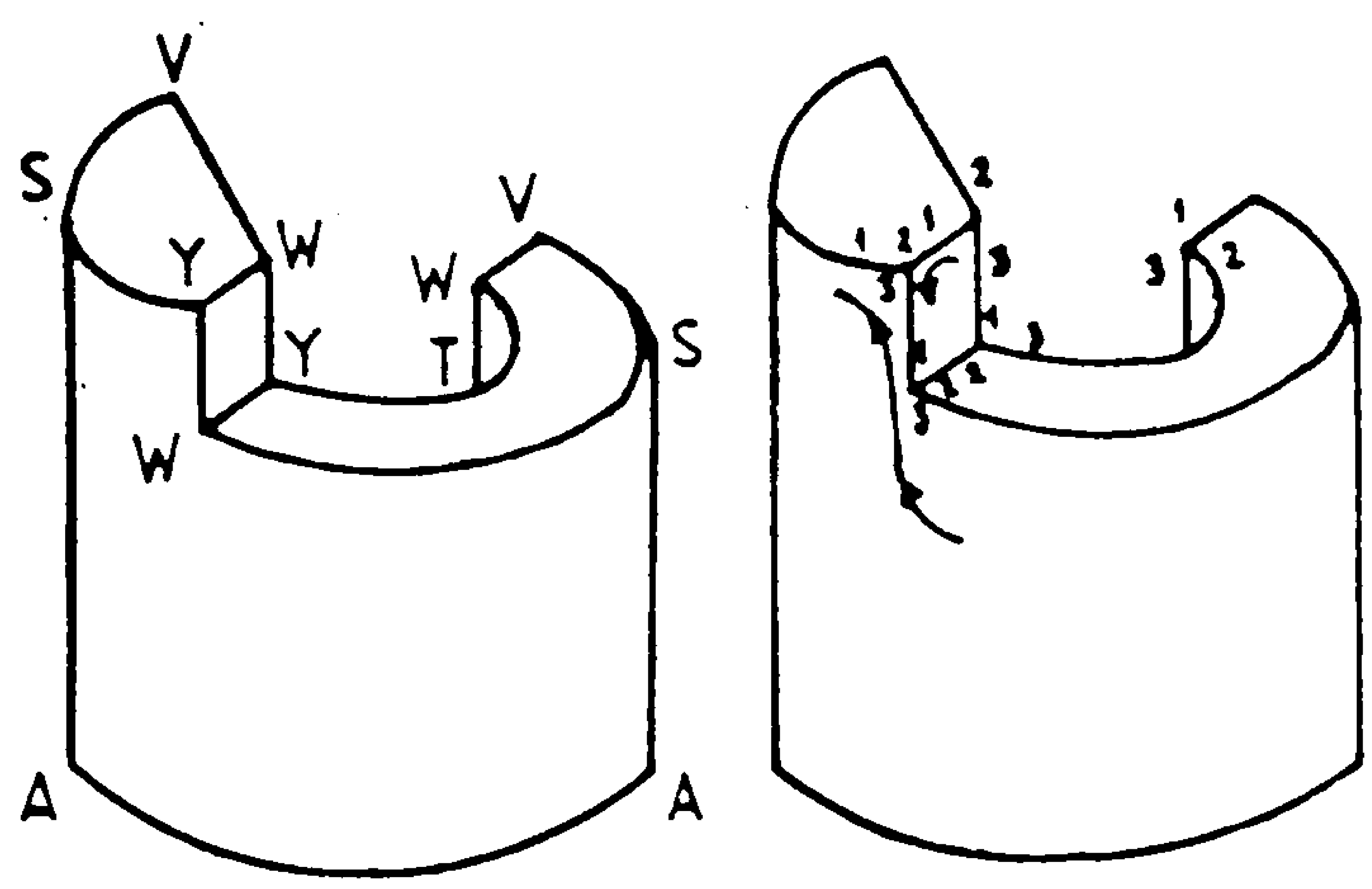
FIGURE 3.9

Mackworth's program constructs dual graphs (point per face, edge per edge, face per point) for the polyhedrons of the scene, in the gradient space. The construction is based on certain coherent rules and constraints, that do not contradict with each other. Then the program examines convexity, occlusion and finally suggests hidden lines that are consistent with the interpretation. Sugihara's work generalizes Mackworth's.

Another program uses *skewed symmetry* (tilted symmetry about an axis) to determine face orientations [Kanade '79]. Like in the previous approach, skewed symmetry resulting from a tilted real one, provides with constraints in the gradient space, which derive the orientation of planes in the scene.

3.2.5 Understanding of Curved-Surface Bodies

This program describes a set of bodies with planar or quadric faces, using defective data extracted from a set of multiple photographs [Shapira '78]. The basic assumptions of the program are: every vertex in the scene is formed by exactly three surfaces and belongs to exactly three edges, no smooth transition between two different faces is allowed and there are no limb passes through a vertex. Junctions can have no more than three lines and can be classified into types W, Y,V,T,A and S (Fig. 3.10a). A junction is *valid* when it is a projection of a vertex. Thus Y and W are valid, A,S and T are not valid and V is undecided until more evidence is found. A *cyclic order* is induced to the edges of a vertex, by following around the vertex in a clockwise way and numbering its edges in the order 1<2<3<1. The tracing of the edges of a boundary is chosen so that the edges meeting at each vertex



(a) (b)

FIGURE 3.10

follow an increasing cyclic order (Fig.3.10b). The cyclic order of missing or invisible lines is not known and must be determined. As *line assembly* (LA) is defined as the directed path followed in tracing out lines belonging to a single boundary in increasing cyclic order. An LA in picture i is denoted by: $A_{i,1}...A_{i,n}$, with $A_{i,k}A_{i,(k-1)} < A_{i,k}A_{i,(k+1)}$ for every 1<k<n. Two LA's are *distinct* if they trace out lines corresponding to two distinct boundaries. Two rules determining a cyclic order in a junction are:

Rule 1: If $A_{i,1}...A_{i,n}$ is an LA in which $A_{i,1} \equiv A_{i,n}$, then there must be in $A_{i,1}$ the relation $A_{i,1}A_{i,(n-1)} < A_{i,1}A_{i,2}$.

Rule 2: If $A_{i,1}...A_{i,n}$ and $A_{j,1}...A_{j,m}$ are two distinct LA's with $A_{j,m} = A_{i,1}$ and $A_{j,(m-1)} \not\equiv A_{i,2}$, then there must be in $A_{i,1}$ the relation $A_{i,1}A_{i,2} < A_{j,m}A_{j,(m-1)}$.

Unknown cyclic order is determined by the use of these rules directly or indirectly. The final description of the 3-D scene is achieved by comparing the line structures from the given pictures (three in this case) against each other and use the information found in one picture to verify the information found in another.

Two junctions from different pictures that satisfy certain geometric rules are defined as *matchable*. A set of three junctions, from different pictures that are matchable in pairs, form a *triple*. Two matchable junctions that are projections of the same vertex *match* each other. A triple in which the junctions match each other is a *match triple*. To establish matches, for every junction all matchable junctions in the two other pictures are found and all match triples are detected. This detection is based on the context provided by line connections between lines. A *pair match* follows the triple match phase. Next, line matching is done for two pictures at a time for the lines terminating at matched junctions. During this procedure unordered junctions can be cyclically ordered and missing connections between junctions can be detected.

The collection of lines from all pictures that correspond to the boundaries of a single face will be referred to as a *face group*. This is built by starting at a line that does not belong to two face groups and following the LA in a direction not followed before. All new matches from other pictures are added to the group as they are met in the line trace. The face-group generation is completed when the trace returns to the starting line. Otherwise the trace is continued in a new picture. If the process is blocked in all three pictures a new trace is tried from the starting point following the LA in the opposite direction, 'jumping' from picture to picture if necessary. When all lines have been traced twice, further data recovery is called, based on the fact that two components assembled simultaneously into two face-groups must correspond to the intersection of associated faces.

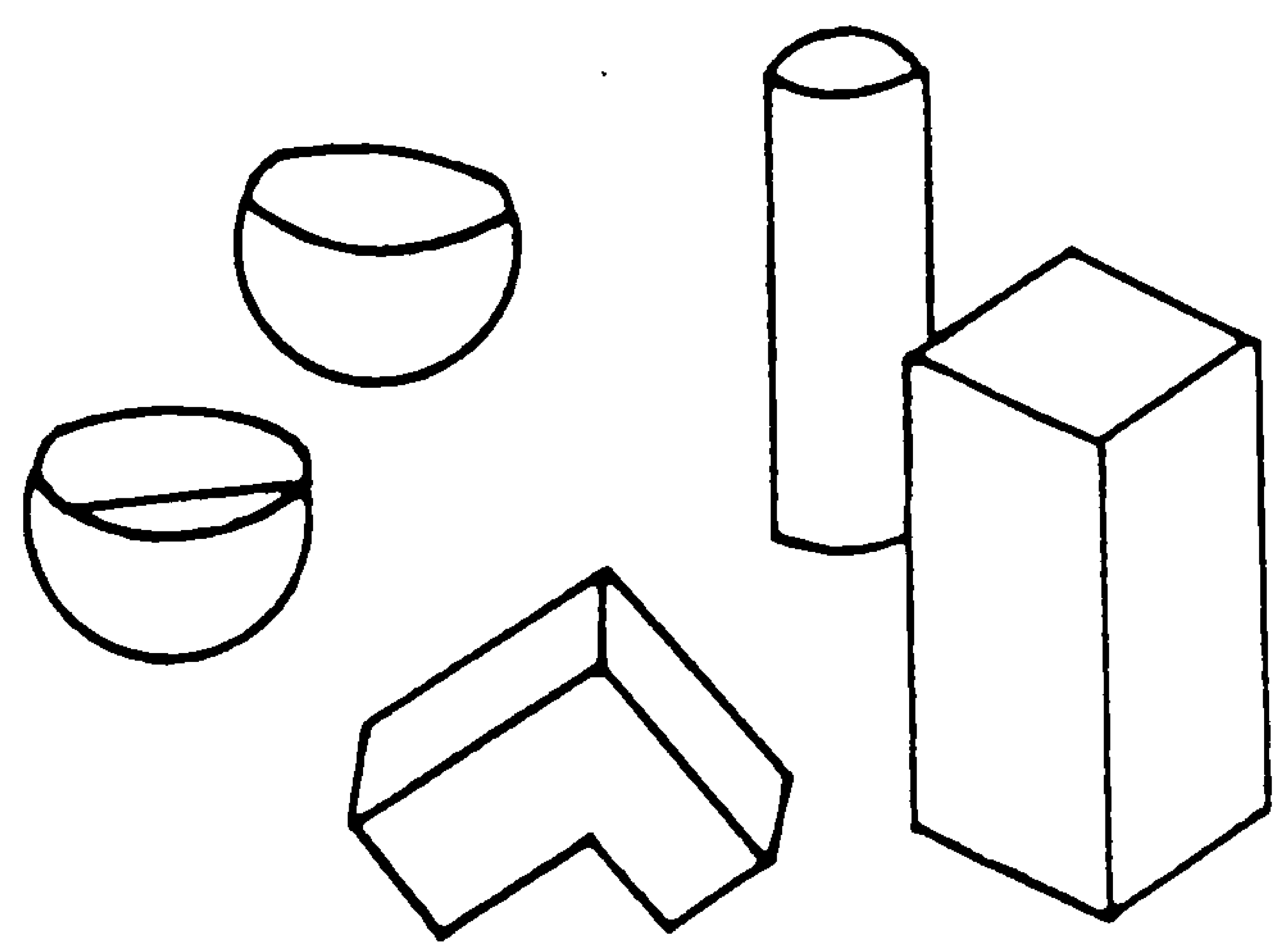


FIGURE 3.11

Each object is a set of face-groups, with no common elements between two sets. A new object is formed by taking a face group that is unassigned and recursively adding to it every unassigned face-group that has a line in common with any of the face groups already in the set. Finally a special procedure determines the curved faces.

The program has very good results on scenes like the one in Fig.3.11 and it is a serious attempt to extend the work on polyhedra, to include curved bodies. An earlier attempt was made by [Chien & Chang '74]. They analysed perfect line drawings representing bodies with planar, conical or cylindrical faces, where no intersection of two curved faces was allowed.

Another interesting system examines the appearance of trihedral vertices from different viewpoints, using the Huffman-Clowes junction dictionary [Thorpe & Shafer '83]. The objects considered consist strictly of trihedral blocks. The purpose of this analysis is to solve the problem of identifying the same vertices in different pictures.

Firstly, a way of deriving vertex types from junction types by inference rather than by enumeration and table look-up, is shown. As the viewpoint crosses into another octant the vertex will appear as a different junction type, following two strong constraints. The first is conservation of vertex type and the second is, that each octant is adjacent to three other octants. These two constraints give a *transition graph*, showing all possible junction types. Then a matching process, relying on topology, to identify the same vertices in two line drawings of the same scene, is presented. A consistent match satisfies three constraints. *Vertex conservation:*

since the same objects are in each image, the same vertices must also be present. *Type conservation:* a vertex must always keep the same shape. *Conservation of adjacency:* two junctions in one image directly connected by an edge, must match two junctions in the second image, that are connected with a line or have the possibility of an invisible line between them. The matching algorithms are based on a central data structure, the *correspondence graph*. Its nodes consist of a junction from one image and a junction from the second image, which may correspond to the same actual vertex. *Consistent* nodes in the graph are connected with links. A complete match consists of a sub-graph of the correspondence graph (called a *clique*), such that the nodes contain every junction in each scene and are linked to each other.

The process works for scenes with more than one object or with partly invisible objects. Topological symmetry can cause spurious minor image matches. The algorithm can be extended to non-trihedral objects, provided that extra constraints are added, in order to limit the number of correct matches.

This method, considers the problem of interpreting the shape of a 3-D space curve from its 2-D perspective image contour [Barnard & Pentland '83]. The basic idea is to segment the line drawings in such a way, that each segment is likely to comprise a projection of a planar segment. Then the planes implied by the segmentation are calculated and the curvature of the contour segments is measured. Another test examines if the curvature of the segments remains constant. If not, a further segmentation is performed. Finally the results of the procedures are assembled into an estimate of the shape of the entire 3-D space curve. The method has satisfactory results on helical curves.

Until now, little attention has been paid to the importance of the viewpoint in the vision theories. Cowie argues that this is the 'missing link' in observer/observed relationships [Cowie '83]. Attempting to clarify the logic associated with such relationships, he examines three main types of them: when the viewpoint is *general*, when it is *representative* and when it is *privileged*. Finally he suggests that vision systems should be thought of as viewing-viewed configurations rather than viewed configurations. This is supported by the idea, that understanding observer/observed relationships feeds back into understanding what may be observed.

3.3 A SIMULATOR FOR LINE DRAWINGS OF 3-D SCENES

The purpose of this program is to create line drawings of 3-D polyhedral scenes, which are to be used as input to a recognition program. The set of line drawings is treated as a 'photograph' of a scene of 3-D objects taken from a certain viewpoint. Assuming that the procedures of feature extraction [Gabrielidis '82] are successfully done, the picture can be summarized by a set of vertices - each determined by its x and y coordinates - and a set of lines, which show the connectivity between the vertices.

The program considers first of all, the objects that comprise the scene. To each one of them two arrays are allocated. The first one contains the coordinates x,y and z of the vertices* which determine the object and the second one contains all the vertices, which are connected to each other. The object is then rotated, translated and scaled, so that

In the more general case where objects with curved surfaces are included, instead of vertices other basic features like centre of curvature, radius, etc. are used.

it takes the desired place in the scene. A central projection of the composed scene is obtained and all the lines and vertices that belong to occluded faces are removed from the two arrays. Finally the connectivity array is translated into PROLOG predicates, which are saved in a file. The result of the above transformations is a line drawing of the scene which can be visually obtained by a plotter.

3.3.1 3-D Transformations

Three main transformations are considered; *rotation*, *translation* and *scaling*. If x,y,z are the coordinates of a point in a 3-D cartesian system and x',y',z' are its new coordinates after being submitted to a transformation T, then using matrices, this is:

$$[x'y'z'] = [x\ y\ z] \times T$$

where T is a 3×3 matrix.

3.3.2 Rotation

The transformation of rotation in a cartesian XYZ-space is shown better if it is considered as three separate rotations with respect to the axes Z, Y and X respectively. The original system XY and the system X'Y' obtained after a rotation by an angle α, is shown in Figure 3.12 and is given by:

$$\begin{aligned} x' &= x\cos\alpha + y\sin\alpha \\ y' &= -x\sin\alpha + y\cos\alpha \end{aligned} \qquad (3.1)$$

If this rotation is considered as a special case of a planar rotation in the XYZ-space, with respect to the Z-axis, then equations (3.1) can be extended to include z and z':

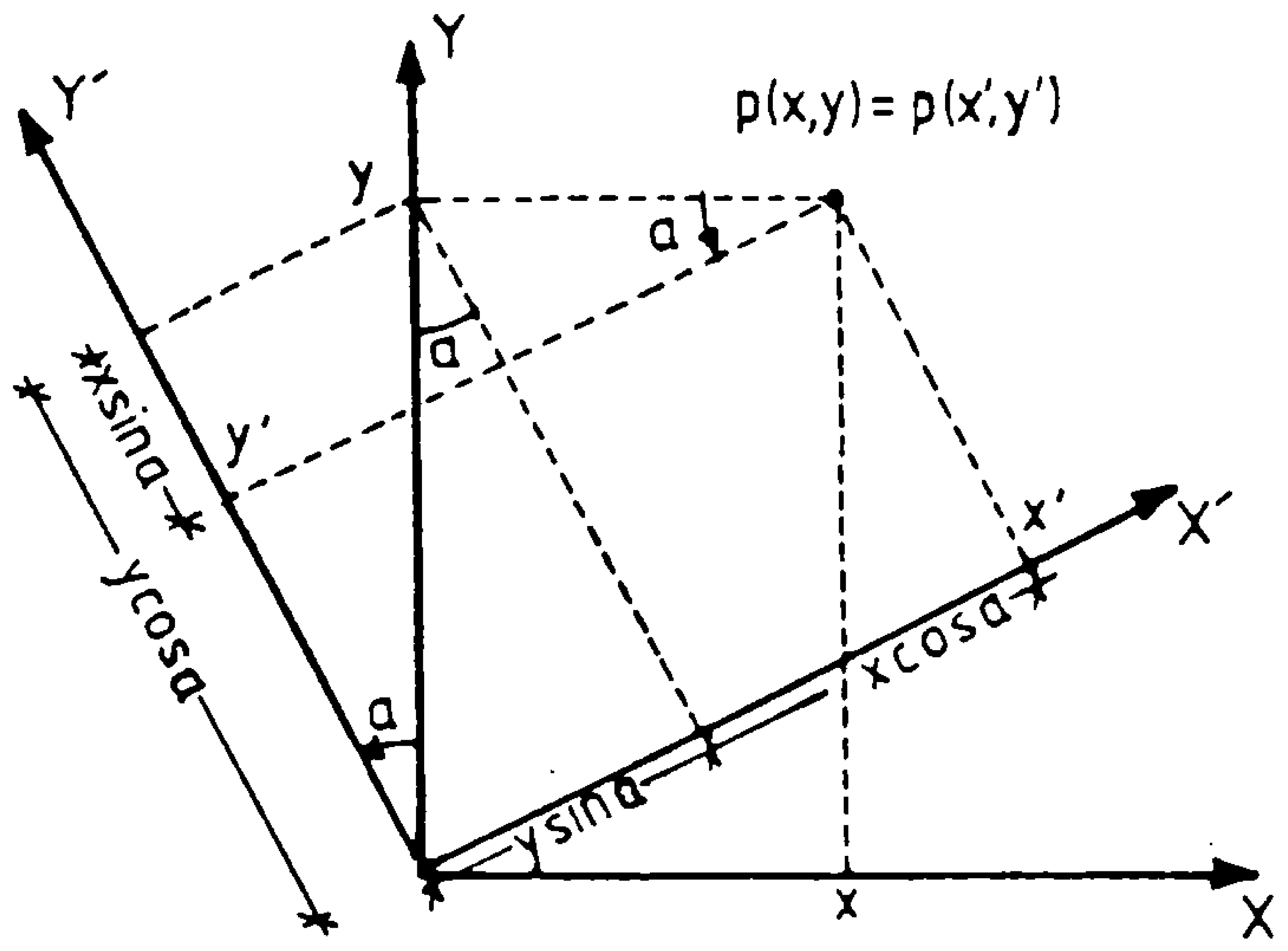


FIGURE 3.12

$$
\begin{aligned}
x' &= x.\cos\alpha + y.\sin\alpha + z.0 \\
y' &= -x.\sin\alpha + y.\cos\alpha + z.0 \qquad (3.2) \\
z' &= x.0 + y.0 + z.1
\end{aligned}
$$

or in a matrix notation:

$$[x'y'z'] = [x\ y\ z] \times \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (3.3)$$

Analogously, the transformed coordinates of a point after rotations with respect to the Y-axis and X-axis by angles β and γ respectively, are given by:

$$[x'y'z'] = [x\ y\ z] \times \begin{bmatrix} \cos\beta & 1 & -\sin\beta \\ 0 & 0 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \qquad (3.4)$$

$$[x'y'z'] = [x\ y\ z] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & -\sin\gamma & \cos\gamma \end{bmatrix} \qquad (3.5)$$

Obviously, a general rotation in the 3-D space with respect to all three axes, is obtained by simply multiplying the three matrices (3.3), (3.4) and (3.5).

By using the notation $R(\alpha)$, $R(\beta)$, $R(\gamma)$ for the matrices (3.3), (3.4) and (3.5) respectively and accepting that the rotation with respect to the X-axis is performed first (matrix multiplication is not commutative!), the coordinates x_R, y_R, z_R of a point after rotation are given by:

$$[x_R \; y_R \; z_R] = [x \; y \; z] \times R(\gamma) \times R(\beta) \times R(\alpha)$$

or with

$$p = [x \; y \; z], \; p_R = [x_R \; y_R \; z_R] \text{ and } R(\alpha,\beta,\gamma) = R(\gamma) \times R(\beta) \times R(\alpha) \tag{3.6}$$

$$p_R = p \times R(\alpha,\beta,\gamma) \tag{3.7}$$

$R(\alpha,\beta,\gamma)$ is called the *rotation matrix* and if the matrix multiplications are executed, $R(\alpha,\beta,\gamma)$ takes the form:

$$\begin{bmatrix} \cos\alpha.\cos\beta & \sin\alpha.\cos\beta & -\sin\beta \\ -\sin\alpha.\cos\gamma+\cos\alpha.\sin\beta.\sin\gamma & \cos\alpha.\cos\gamma+\sin\alpha.\sin\beta.\sin\gamma & \cos\beta.\sin\gamma \\ \sin\alpha.\sin\gamma+\cos\alpha.\sin\beta.\cos\gamma & -\cos\alpha.\sin\gamma+\sin\alpha.\sin\beta.\cos\gamma & \cos\beta.\cos\gamma \end{bmatrix} \tag{3.8}$$

3.3.3 Translation

Translation of a point is performed by adding a positive or negative constant to each coordinate of the point. If the *parameters of translation* are denoted by t_x, t_y and t_z for translation in directions x, y and z respectively, the translated coordinates are given by:

$$[x_t\ y_t\ z_t] = [x\ y\ z] + [t_x\ t_y\ t_z]$$

or

$$p_T = p + T(t_x, t_y, t_z)\ . \qquad (3.9)$$

3.3.4 Scaling

Scaling is performed by multiplying each one of the coordinates x,y and z, with a *scaling factor*. In matrices this is given by:

$$[x_s\ y_s\ z_s] = [x\ y\ z] \times \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

or

$$p_S = p \times S. \qquad (3.10)$$

S is the *scaling matrix* and s_x, s_y, s_z are the scaling factors in the directions X,Y and Z respectively. The scaled object is compressed or stretched, depending on whether the scaling factors are smaller or greater than 1. In the special case that $s_x = s_y = s_z = c$, the object is *homogenously* scaled. The scaling matrix S of an homogeneous scaling can be replaced by the constant c and thus (3.10) becomes:

$$p_S = c.p\ .$$

It is interesting to note that the order in which the three transformations are applied to the same set of points determines the result of the combined transformation and a different order will give different results. New forms of the transformation matrices will be discussed in 3.3.6.

3.3.5 Projection of 3-D Objects onto 2-D Space

Once a 3-D model has been made, it is projected on 2-D to form a line drawing. Projections generate a perspective view of an object and are generally expensive transformations. Therefore in order to keep the execution time within certain limits, the form of projection used is usually restricted to the *central projection* or its simpler form the *orthographic* projection. The problem of central projection, is to determine the projection of an object point, located somewhere in a 3-D space, onto a plane in that space. The projection of the object point is called the *image point* and the plane on which it is projected is called the *image plane*. The *viewpoint* of the projection is located on one of the axes of the 3-D cartesian co-ordinate system and is called the *centre of projection*. Figure 3.13 shows a central projection where the viewpoint is on the Z-axis and the image plane is parallel to the XY-plane. The latter is not necessary and it is used only for simplicity reasons.

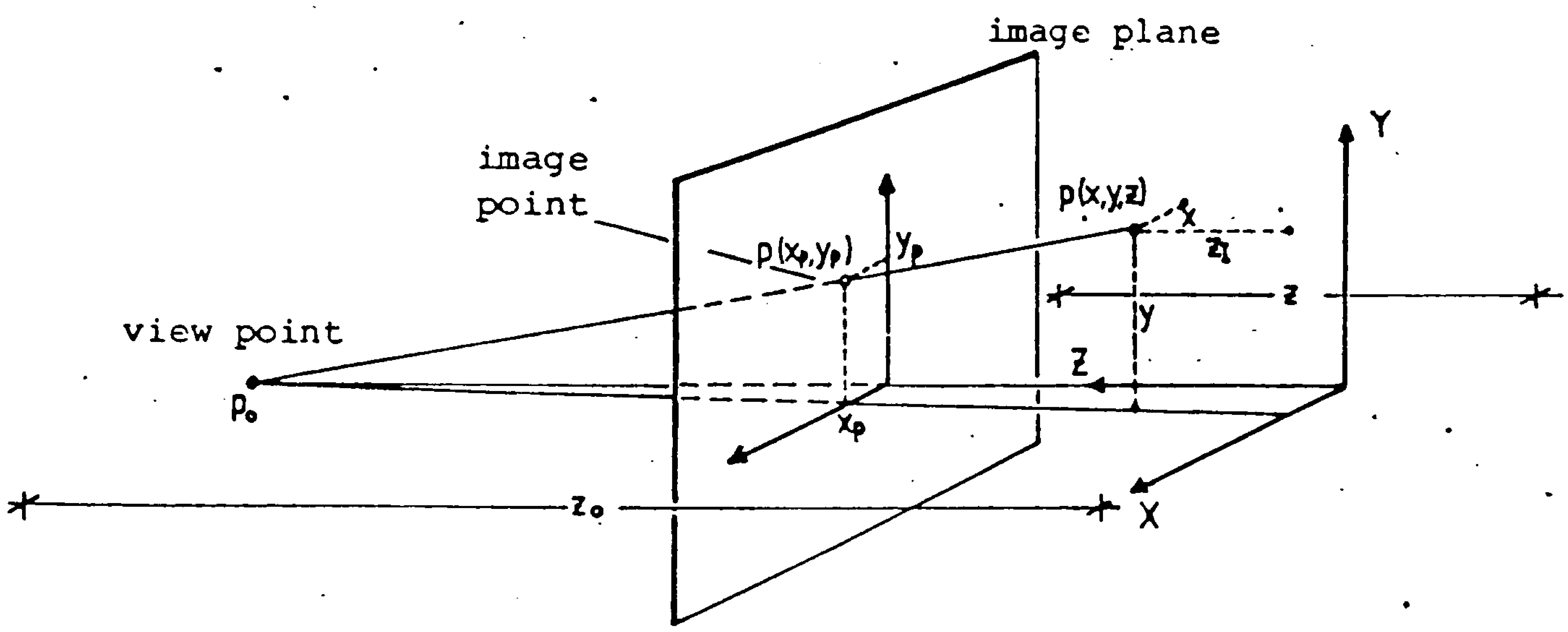


FIGURE 3.13

If x_p and y_p are the coordinates of the 2-D system on the image plane and z_O and z_I are the distances of the origin of 3-D coordinate system from the viewpoint and the image plane respectively, then from the theorem of similar triangles follows that:

$$x_p = \frac{z_O - z_I}{z_O - z} x , \qquad y_p = \frac{z_O - z_I}{z_O - z} y \tag{3.11}$$

If the image plane coincides with the XY-plane of the 3-D coordinate system i.e. if $z_I = \emptyset$, the equations of (3.11) are simplified into:

$$x_p = \frac{z_O}{z_O - z} x = \frac{1}{1-(z/z_O)} x$$

$$y_p = \frac{z_O}{z_O - z} y = \frac{1}{1-(z/z_O)} y \tag{3.12}$$

Finally by moving the viewpoint to infinity, i.e. $z_O \to \infty$, then $\frac{1}{1-(z/z_O)} \to 1$ and the solution $x_p = x$, $y_p = y$ is obtained. This projection is a special case of the central projection and is called *orthographic*. It is also a special form of another projection called *parallel*, which projects parallel lines of a 3-D object, onto parallel lines of its image.

3.3.6 Homogeneous Coordinates - Combined Transformations

The transformations examined in §3.3.4 and §3.3.5 can be more useful, if they are performed by matrix multiplication (like rotation and scaling). Successive or combined transformations are made a lot simpler, since they can all be performed in one operation. This is achieved by substituting the set of cartesian coordinates [x y z] for a point in the 3-D space, by the new set: [wx wy wz], where $w \neq 0$. These are called the *homogeneous coordinates* and are obtained by multiplying the original ones with the non-zero constant w and at the same time using w as a fourth 'dummy' coordinate. In order to get back to the cartesian set of coordinates only a division by w is needed. The important point from this trick is, that it is now possible to express all four transformations as 4×4 matrices multiplied by the 1×4 matrix of point coordinate set. For simplicity w can be set to 1 and thus a point is represented as [x y z 1]. The rotation matrix can be written as 4×4:

$$\tilde{R}(\alpha,\beta,\gamma) = \begin{bmatrix} A & B & C & 0 \\ D & E & F & 0 \\ G & H & I & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A,B,...,I are the components of the 3×3 matrix of (3.8).

The proof is simple: if $[x_R\ y_R\ z_R\ 1] = [x\ y\ z\ 1] \times \tilde{R}$ is the image point of [x y z 1] (both points represented in homogeneous coordinates), the execution of the multiplication gives [Ax+Dy+Gz Bx+Ey+Hz Cx+Fy+Iz 1], or x_R=Ax+Dy+Gz, y_R=Bx+Ey+Hz, z_R=Cx+Fy+Iz, i.e. the same as in (3.7).

Analogously the scaling matrix of (3.10) takes the following 4×4 form:

$$\tilde{S} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (3.13)$$

Similarly, translation can be performed by multiplying a point with the matrix:

$$\tilde{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \qquad (3.14)$$

Proof: $[x\ y\ z\ 1] \times \tilde{T} = [x+t_x\ \ y+t_y\ \ z+t_z\ \ 1]$.

For the perspective transformation the following matrix is used:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/(z_O-z_I) \\ 0 & 0 & 0 & z_O/(z_O-z_I) \end{bmatrix} \tag{3.15}$$

The multiplication $[x\ y\ z\ 1] \times P$ yields $[x\ y\ \emptyset\ \frac{z_O-z}{z_O-z_I}]$ which gives the same result as in (3.11) if the first two coordinates are divided by the fourth one.

Combined transformation matrices can be obtained by multiplying two or more single transformation matrices in a certain order. The order in which the multiplications are performed is important because matrix multiplication is *not* commutative. For example scaling after rotation will have a different result than when scaling is applied first, unless all three scaling factors are equal. The two different transformation matrices are shown below:

$$\tilde{R} \times \tilde{S} = \tilde{RS} \quad \begin{bmatrix} A.s_x & B.s_y & C.s_z & 0 \\ D.s_x & E.s_y & F.s_z & 0 \\ G.s_x & H.s_y & I.s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

$$\tilde{S} \times \tilde{R} = \tilde{SR} \quad \begin{bmatrix} s_x.A & s_x.B & s_x.C & 0 \\ s_y.D & s_y.E & s_y.F & 0 \\ s_z.G & s_z.H & s_z.I & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

in general $\tilde{RS} \neq \tilde{SR}$ with $s_x \neq s_y \neq s_z$ as in (3.15) above.

Analogously, the combined transformation matrix that performs rotation, scaling and translation is obtained by multiplying RS×T:

$$\tilde{RS} \times \tilde{T} = \tilde{RST} = \begin{bmatrix} A.s_x & B.s_y & C.s_z & 0 \\ D.s_x & E.s_y & F.s_z & 0 \\ G.s_x & H.s_y & I.s_z & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \qquad (3.17)$$

Finally the matrix that gives the perspective p_1 of a point p after it has been rotated, translated and scaled, will be:

$$p_1 = p\times(\tilde{RST}\times P) = p\times T_4, \quad \text{where } T_4 = \tilde{RST}\times P .$$

$$T_4 = \begin{bmatrix} A.s_x & B.s_y & 0 & -C.s_z/(z_O-z_I) \\ D.s_x & E.s_y & 0 & -F.s_z/(z_O-z_I) \\ G.s_x & H.s_y & 0 & -I.s_z/(z_O-z_I) \\ t_x & t_y & 0 & (z_O-t_z)/(z_O-z_I) \end{bmatrix} \qquad (3.18)$$

Thus, the entire process of the four transformations can be performed in three steps:

1. All points of a structure in the 3-D space are represented in homogeneous coordinates:

$$p = [x\ y\ z] \rightarrow p_h = [x\ y\ z\ 1].$$

2. Every point is transformed by being multiplied by the 4×4 matrix of (3.18):

$$p_h \times T_4 = [x_{RST}\ y_{RST}\ 0\ \frac{z_O-z_{RST}}{z_O-z_I}] .$$

3. The first two components of the resulting vector must be divided by the fourth component i.e. $1-z_{RST}/z_O$. (In the case of orthographic projection the final division is omitted).

The combined transformation can be written as an individual subroutine, that has as input the coordinate values of a point in the 3-D space (x,y,z) together with ten transformation parameters, i.e. three angles of rotation α, β, γ, three scaling factors s_x, s_y, s_z, three translating parameters t_x, t_y, t_z and parameter z_0, defining the centre of perspective. Output of the subroutine is the values x_p and y_p of the perspective representation. Figure 3.14 shows the subroutine as a transfer system.

If any of the geometric transformations are not to be applied, their control parameters are substituted with the respective default sets. These default sets are:

[0 0 0] for rotation,

[1 1 1] for scaling,

and [0 0 0] for translation.

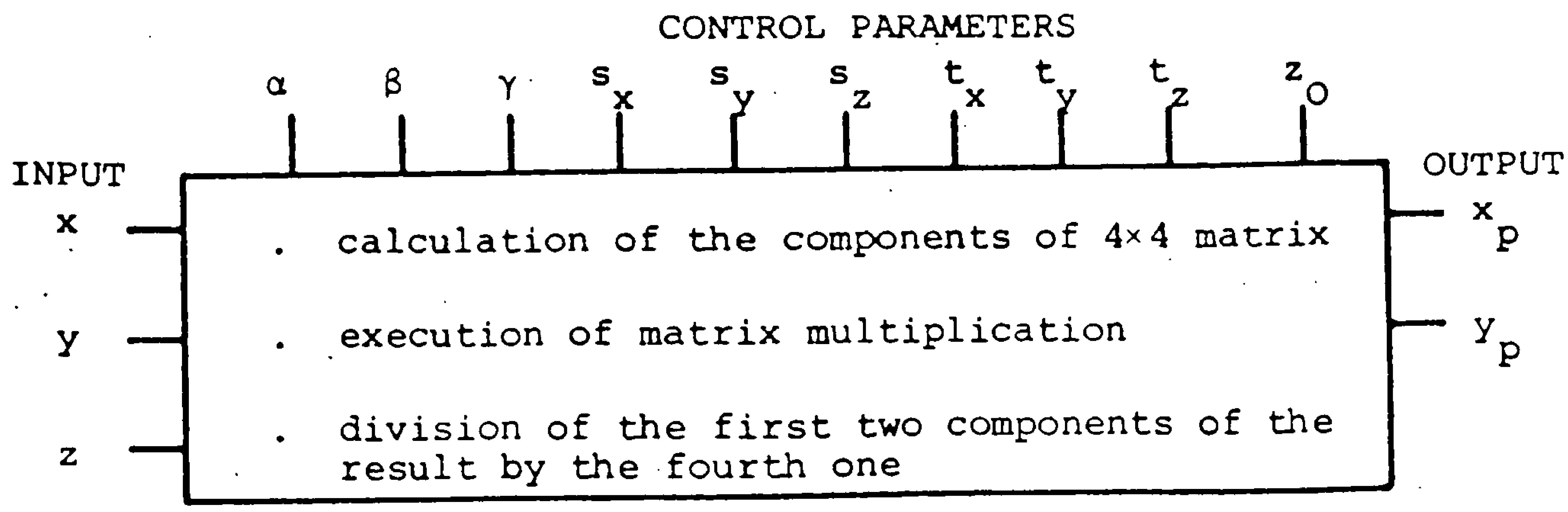


FIGURE 3.14

3.3.7 Viewing Parameters of Projection Systems

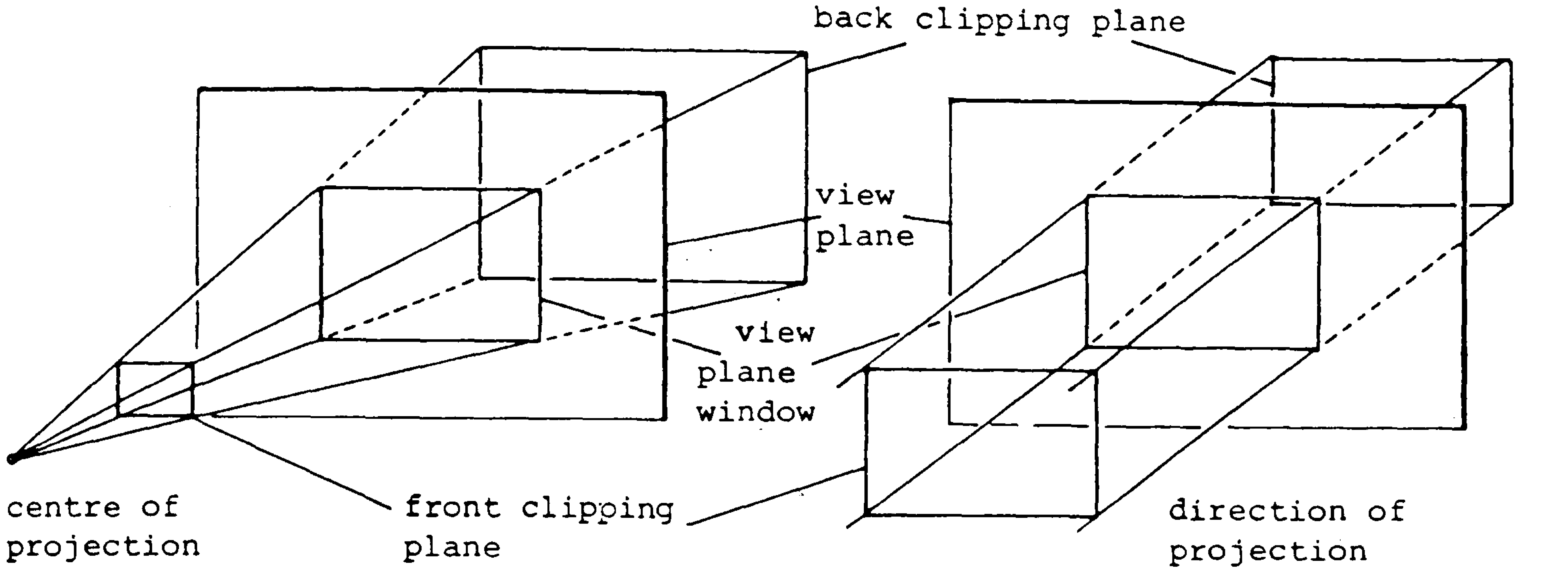
Graphic objects are defined always in the 3-D cartesian space so that they are inside a 3-D window, called a *view volume*, the form of which depends on the type of projection. The view volume is determined by a window on the view plane and the centre of projection (for central projection) or the direction of projection (for parallel projection). The view plane is defined by the following set of parameters:

The *view reference point*, which is the centre of attention i.e. where the object is situated. All other viewing parameters are expressed relative to this point.

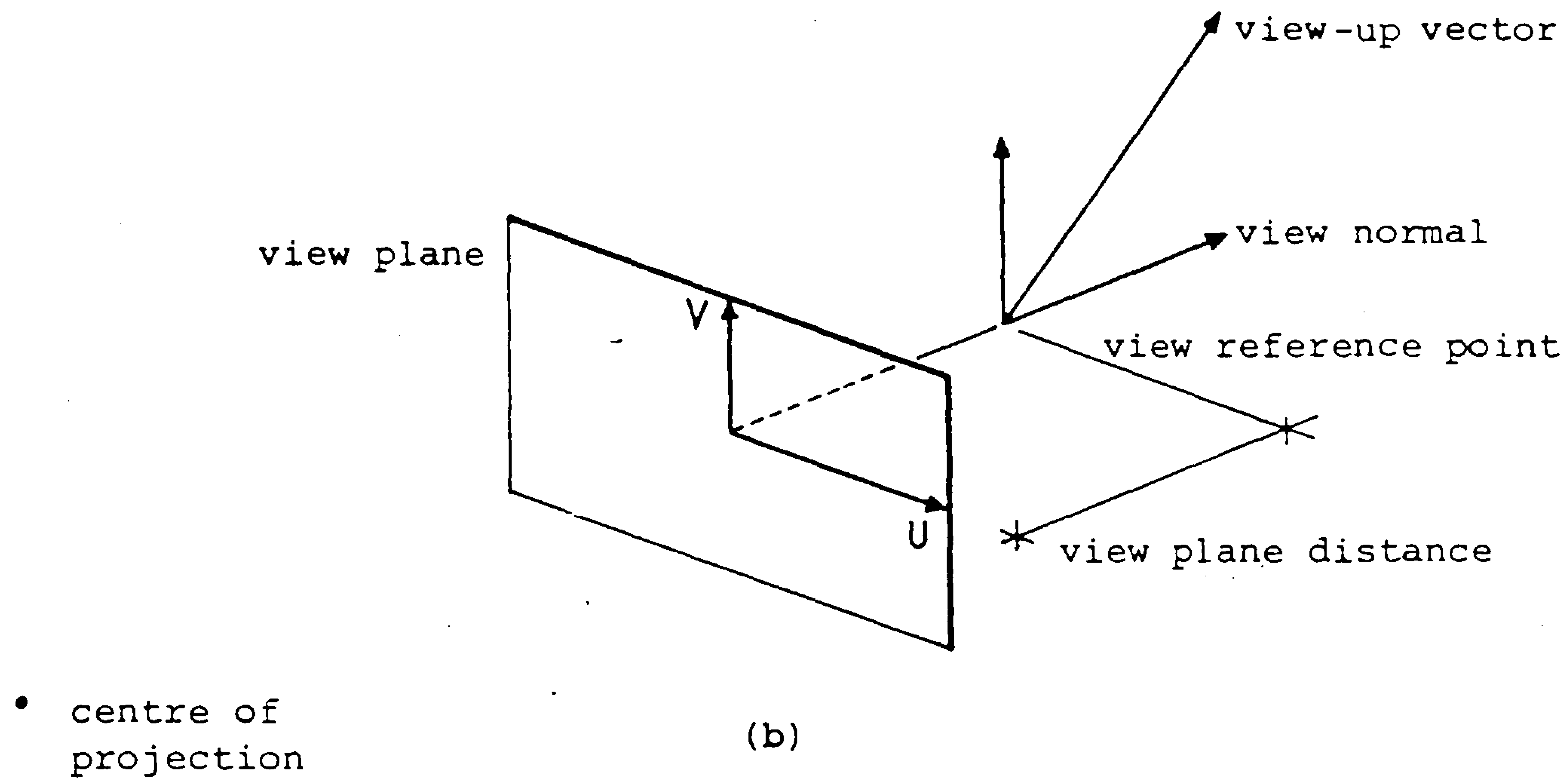
The *view-plane normal*, which is a vector perpendicular to the view plane with its origin at the view reference point. It specifies the orientation of the view plane in the 3-D world space.

The *view-plane distance*, which is a signed quantity determining the distance between the view reference point and the view plane. The view-plane distance is measured along the view-plane normal.

Once the view plane has been specified, a coordinate system UV is defined on it. The UV origin is the point where the line colinear to the view-plane normal intersects the view plane and its orientation is specified by a *view-up vector*. The view-up vector starts at the view reference point and ends at a point specified by the programmer. The view-up vector is projected onto the view plane in a direction parallel to the view-plane normal and the coordinate system is orientated so that this projection point is always 'upwards'. A window can be defined on the view plane in terms of maximum and minimum U and V values. The view volume determines the clipping of the 3-D object, while the view plane window determines the clipping of its projection. The infinity (parallel projection) or semi-infinity (central projection) of the view volumes can be limited, by introducing two planes parallel to the view plane called the *front* and *back clipping plane* respectively. Figure 3.15 depicts the viewing parameters.



View volume of central projection (a) View volume of parallel projection



(b)

FIGURE 3.15

Following from the above discussion, the problem of obtaining several different views of an object (scene) can be considered as follows:

either the object is kept in a fixed position and the view plane is repositioned every time a new view is required,

or the view plane - and its viewing parameters - remains fixed and the object is rotated.

The first case can be seen as a simulation of a camera (film plane = view plane) moving around the object and it is the method used by CORE*. The second case depicts more or less what actually happens when the view plane is considered as the screen of a graphics VDU (or the plane of a graphics plotter), and is the one used in this work. Figure 3.16 shows this viewing system.

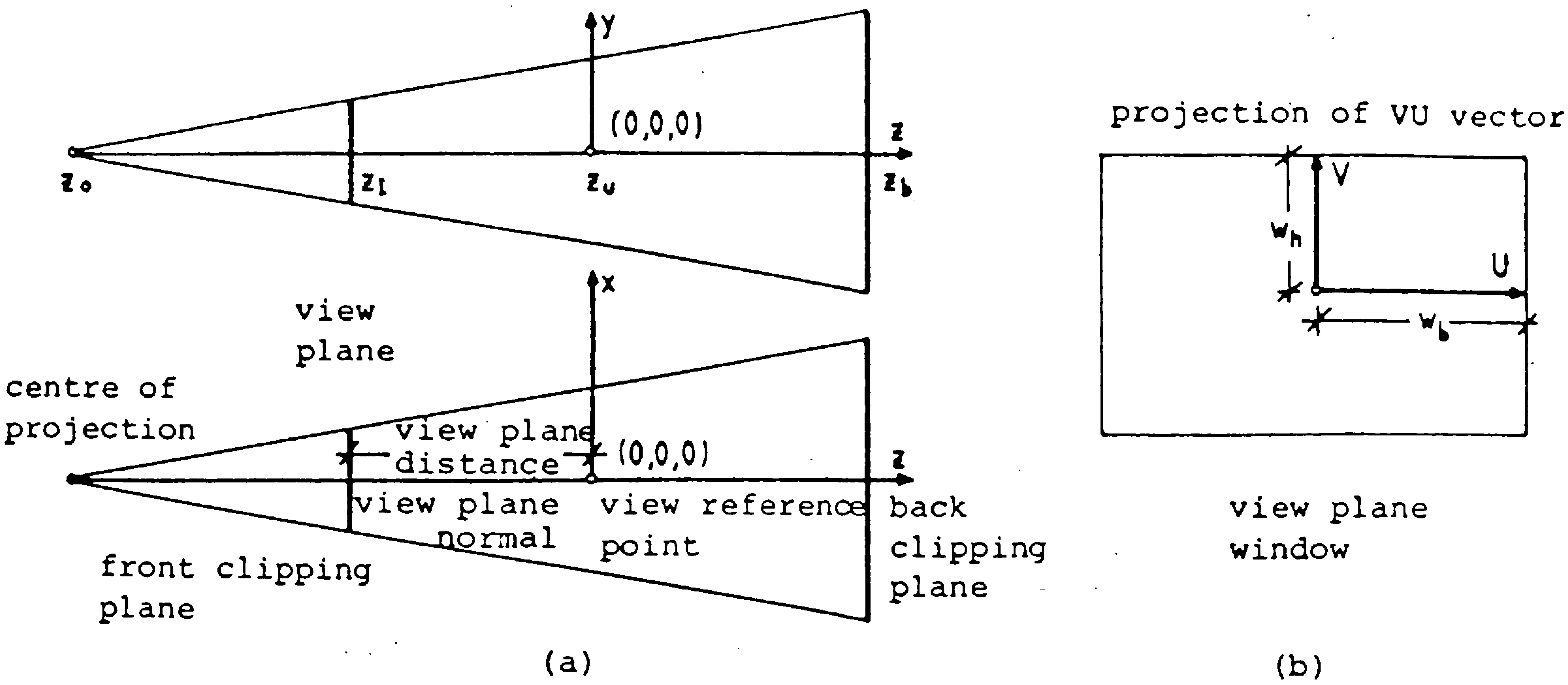


FIGURE 3.16

From Fig. 3.16a it is obvious that the object is placed at z_U the origin of the real world and user coordinate system. The view plane coincides with the front clipping plane at a viewing distance of z_I from the viewing reference point z_U. The centre of projection z_O lies

---

CORE is the standard graphic software system proposed by the Graphic Standard Planning Committee of ACM/SIGGRAPH. See Computer Graphics 11, 3, Fall 1977.

on the view normal and $\overline{z_O z_U}=2\overline{z_O z_I}$. The back clipping place is placed at z_b, so that $\overline{z_U z_b}=\overline{z_I z_U}$. The height of the window is $2w_h$ and its breadth $2w_b$. The view plane coordinate system XY uses the same coordinates X and Y as the user coordinate system. The default values used by the simulator for z_O, z_I, z_U, w_h and w_b are -40,-20,0,10 and 15 (in user units) respectively.

3.3.8 Back-Face Removal

The result of the above discussed transformations is a perfect line drawing showing all the edges and the vertices of a 3-D object. In real life however some of these elements are invisible, because they are behind solid faces that hide them. Thus, the faces of an object can be divided into *visible* or *front faces* and *invisible* or *back faces*. If an edge belongs to two invisible faces, then it is invisible itself and thus, it should not appear in the line drawing. Likewise, vertices that belong to invisible edges, should not be shown in the line drawing either. The back faces of an object can be determined by certain tests and depend always on the view-point. If all the invisible edges and vertices are removed from the list of transformed data, a realistic line drawing is obtained and such are the images, that are given as input to the recognizer. Invisibility may be caused either from parts of the same body (single objects) or from other objects situated in front of it (in scenes). Hidden-line removal in the first case is quite simple and requires only a *visibility test*, in order to determine the invisible elements. The second case is more complicated and requires lengthy and costly computer time processes in order to be achieved [Giloi '78, Harrington '83]. For the 3-D objects examined in this work (see also §6.1, §5.3.3), the visibility test described below is sufficient.

Every solid 3-D object consists of planar faces, which are polygons. Every polygon has two surfaces, a front one and a back one. In order to distinguish between the two faces, the following notation is adopted. The sides of a polygon are considered as vectors with starting and ending points the two corresponding vertices. The polygon can be regarded now as a closed cyclic formation of vectors. By convention a surface is visible if the sense of the vectors is clockwise, and invisible if the sense of the vectors is anti-clockwise. The orientation of a surface is determined by the direction of the cross product of two vectors lying on it. The cross product between two vectors a and b is given by:

$$\vec{s} = \vec{a} \times \vec{b} = \begin{bmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \qquad (3.19)$$

where $\hat{i}$, $\hat{j}$ and $\hat{k}$ are the unit vectors in the usual three cartesian directions,

or $$\vec{s} = (a_y b_z - a_z b_y)\hat{i} + (a_x b_z - a_z b_x)\hat{j} + (a_x b_y - a_y b_x)\hat{k} \qquad (3.20)$$

i.e. the cross product of $\vec{a}$ and $\vec{b}$ is a vector of magnitude $|\vec{a}|.|\vec{b}|\sin\theta$ and direction perpendicular to the plane formed by $\vec{a}$ and $\vec{b}$. The direction in which it points depends on the angle θ between $\vec{a}$ and $\vec{b}$. By convention, two adjacent sides of an invisible polygon meeting at a convex vertex, yield a cross-product vector pointing towards the viewer. The same vector would point in the opposite direction if the vertex was non-convex. Thus, the direction of the cross-product vector of two adjacent sides (represented by two vectors) of a polygon-face forming a convex vertex, determines the visibility of the face. The face is invisible if the normal vector points towards the viewer, and visible if it points away from the viewer.

The direction of vector $\vec{s}$ can be determined by comparing it with the known direction of another vector $\vec{p}$. The dot product of the two vectors is given by:

$$q = \vec{s}.\vec{p} = s_x p_x + s_y p_y + s_z p_z \qquad (3.21)$$

or

$$q = |\vec{s}|.|\vec{p}|\cos\theta \qquad (3.22)$$

if $0 \leq \theta < \frac{\pi}{2}$, then $q > 0$ and the two vectors, $\vec{p}$ and the projection of $\vec{s}$, point in the same direction. Otherwise, if $\frac{\pi}{2} < \theta \leq \pi$, then $q < 0$ and the two vectors point in opposite directions. As vector $\vec{p}$ can be chosen the vector with origin the view point z_0, and end the convex vertex of the two sides the cross product of which is considered. Thus, the face is visible if $q > 0$ and invisible if $q < 0$. Figure 3.17 demonstrates the visibility test.

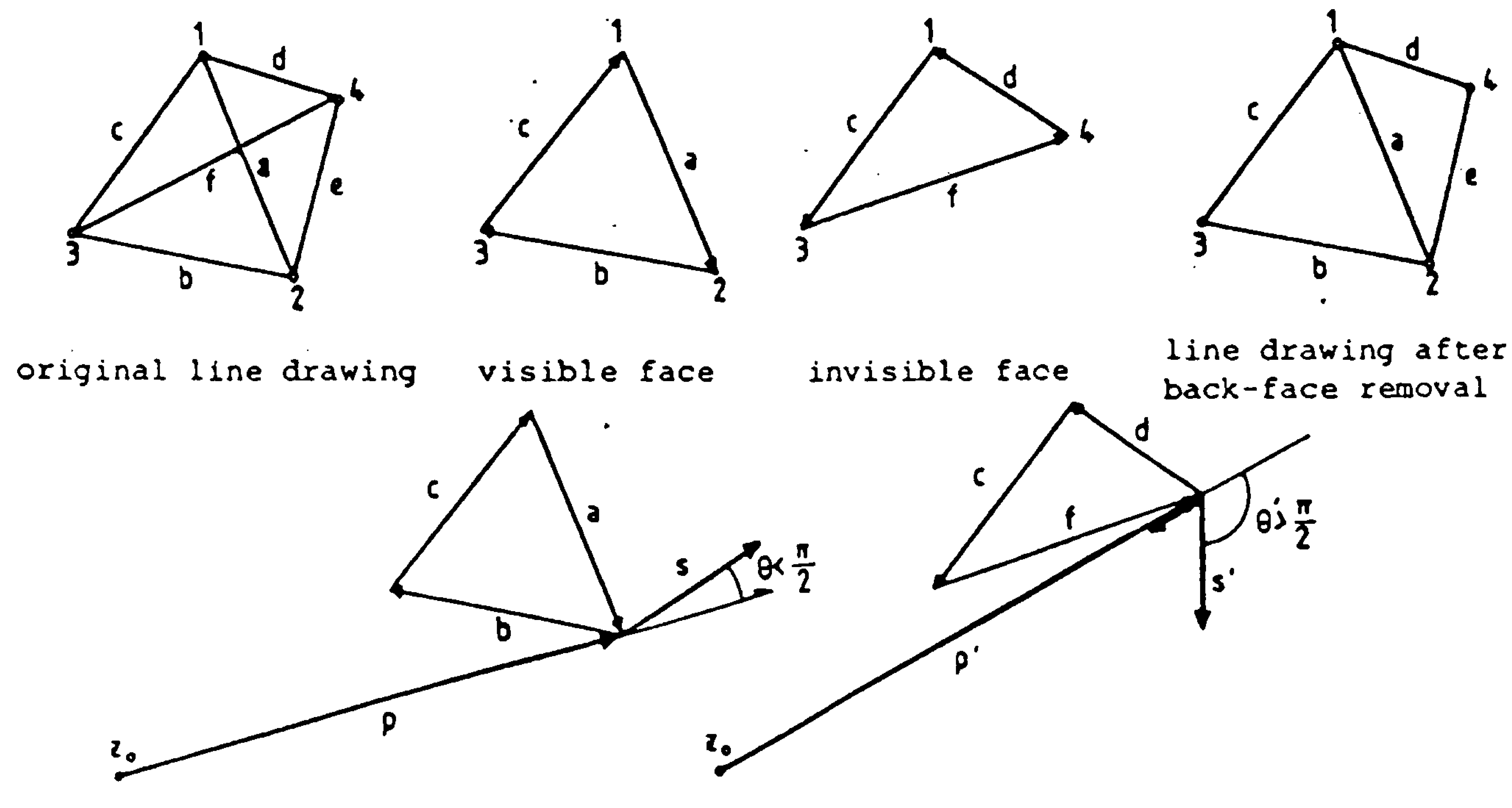


FIGURE 3.17

Care must be taken that the vertex, on which the visibility test is applied must be a convex one. This can be done by performing an extra test for selecting an extreme vertex (e.g. leftmost or uppermost).

3.4 FUNCTION OF THE SIMULATOR

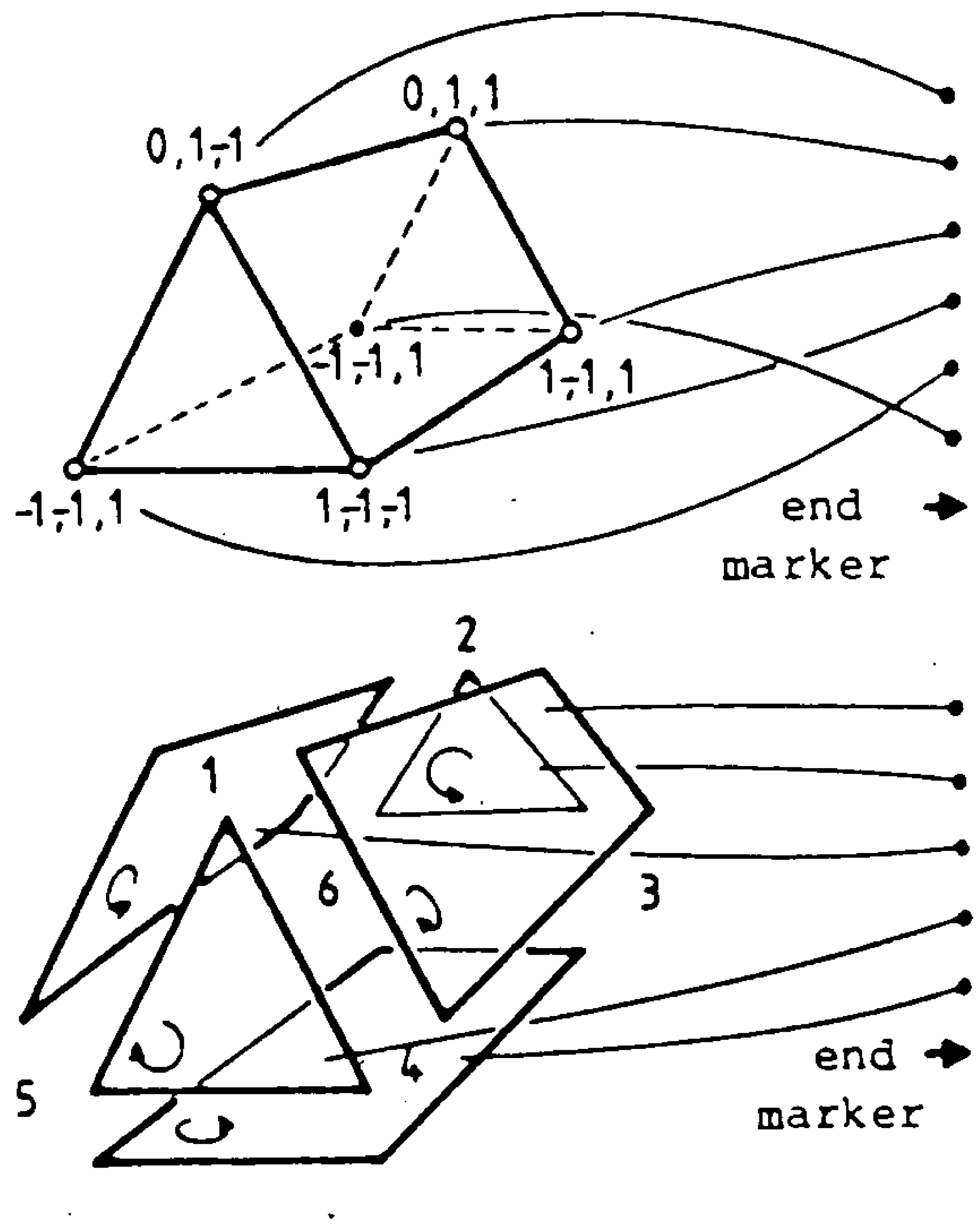
This section examines in detail the function of the simulator, with relation to the routines that use it as part of their operation. The simulator creates 2-D projections of either individual 3-D figures or scenes of 3-D figures, which will act as objects to the phases of learning or recognition. The function of the simulator can be divided into three parts: the *input*, the *scene build-up* and the *output*. The structure of every part is such that both phases can use the simulator in a similar general way with minor differences. These differences are examined at the end of every part.

3.4.1 The Input

The input to the simulator consists of two arrays, the *vertex-array* T and the *face-array* f. The vertex-array is a *2-D, N×3 array* containing the three coordinates x,y and z of every one of the N vertices that make up the scene. Here, it should be mentioned that a scene may consist of one or more 3-D figures with a maximum of N vertices (here N=120). The end of the array is marked by an end-marker, which is a value that lies outside the rectangle of the projection plane (here =100).

The face-array is also a 2-D array with dimensions M and 10. M is the number of faces of the scene (M maximum is here =120) and 10 is the maximum size of the face-list. The *face-list* is a list of integers which correspond to the vertices comprising each face. The first element of the list is also the last, in order to form a cyclic list and the end of the list is marked with an end marker (here =0). The end of the face-array is marked with another end marker (here =-1). The entries of the face-array are pointers to the vertex-array (actually its indices) in order to be able to link every vertex to its 3-D coordinates. From the above it is obvious that $1 \leq M \leq N$ and the maximum number of vertices of a face is ≤ 8 (=10-2).

The simulator offers the possibility of inserting these two arrays manually or by reading them from a file. In the first case the user types in the vertex-array in rows of threes (x,y,z coordinates) and terminates the array with the end marker (100) followed by 0,0. Similarly the face-array is typed-in, in rows of, at the most eight elements, re-entering the first element of the row and terminating it with the end marker (0). Finally another end marker (-1) is placed at the end of the array. In the second case, a file containing the scene is created and the user enables the simulator to input the two arrays from that file, by only typing in the file-name. The file contains the two arrays in the format mentioned above. An important point here is, that care should be taken that always the second element of each face list, indicates a convex vertex of that face. This condition saves the system from looking for an extreme vertex (minimal or maximal test), in order to perform the visibility test (see §3.4.2). Another point is, that the cyclic order of each face is such that list elements follow a clockwise order when the face is viewed from the outside of the object. Thus, visible faces are clockwise lists and invisible faces are anti-clockwise lists. Figure 3.18 gives a visual example of the input of a prism to the simulator.



vertex-array

	x	y	z
1	0.0	1.0	-1.0
2	0.0	1.0	1.0
3	1.0	-1.0	1.0
4	1.0	-1.0	-1.0
5	-1.0	-1.0	-1.0
6	-1.0	-1.0	1.0
7	100.0	0.0	0.0

face-array

1	1	2	3	4	1	0
2	1	4	5	1	0	0
3	1	5	6	2	1	0
4	2	6	3	2	0	0
5	5	4	3	6	5	0
6	-1	0	0	0	0	0

FIGURE 3.18

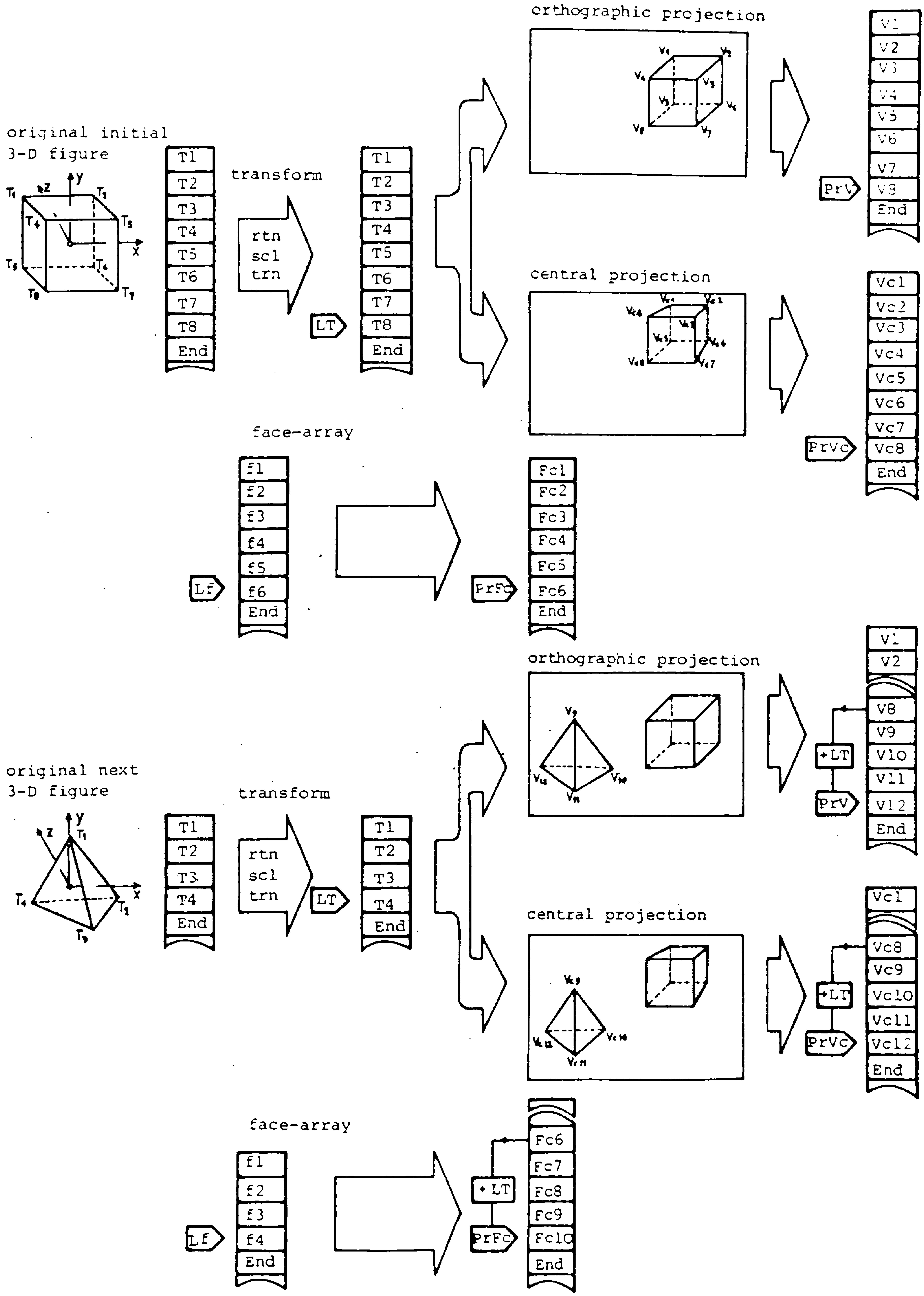
3.4.2 The Scene Build-Up

The input to the simulator may be a complete scene (provided by the user or read from a file) or may be built up by the user. A scene build-up is performed by inserting single 3-D figures, transforming to a desired shape and place, and finally projecting them on the projection plane.

In the case of a complete scene input, the only meaningful transformation would be a rotation. Scaling and translation are likely to cause soft clipping, if one considers a scene that covers most of the projection plane.

The case of scene build-up is more interesting and offers the user more flexibility and better control over the scene composition. The input of every 3-D figure can be done manually by typing in the

two arrays, the vertex-array and the face-array. However it is more reliable and much faster to read the two arrays from already existing files. Every file contains the two arrays defining a single 3-D figure. The coordinates of the figure-vertices are chosen in such a way that the geometrical centre of the figure coincides with the origin (0,0,0) of the user coordinate system. After every transformation of the figure, the system returns a screen-plot to the user, in order to allow him to obtain an acceptable viewpoint. The screen-plot is an orthographic projection of the 3-D figure containing all its edges, whether these are visible or not. When the user has decided on a certain view, an orthographic and a central projection of the 3-D figure are performed and the results are saved in two 2-D arrays, V and Vc respectively. At the same time a soft-clip procedure examines if the coordinates of the transformed figure are within the volume determined by the viewing-parallelepiped and the viewing-pyramid for each projection respectively. In the latter case a warning message with the out of bounds coordinate is given to the user. The face-array is also stored in a new array Fc. The procedure can be repeated and everytime a new figure is added to the scene until the user decides that the scene build-up has been completed. Everytime that a new 3-D figure is added to the scene the values of the vertex-array and the face-array are stored into array V and Vc, and Fc respectively. The new entry is stored one place below the last element of the last figure in an accumulative way. At the end of the procedure the end of each one of the arrays is marked with a special end-marker.
During the scene composition, care should be taken that in the final result, there are no figures with overlapping sides. Figure 3.19 illustrates the scene build-up procedure.



original initial
3-D figure
transform
rtn
scl
trn
LT
T1
T2
T3
T4
T5
T6
T7
T8
End
orthographic projection
central projection
PrV
PrVc
V1
V2
V3
V4
V5
V6
V7
V8
End
Vc1
Vc2
Vc3
Vc4
Vc5
Vc6
Vc7
Vc8
End
face-array
Lf
f1
f2
f3
f4
f5
f6
End
PrFc
Fc1
Fc2
Fc3
Fc4
Fc5
Fc6
End
original next
3-D figure
+LT
V9
V10
V11
V12
Vc9
Vc10
Vc11
Vc12
Fc7
Fc8
Fc9
Fc10

FIGURE 3.19

The scene composition is followed by a hidden-line removal procedure, which applies a visibility test upon every figure-face of the scene. The result of this procedure is the creation of an array of the visible lines and an array of the invisible lines of the scene. From these two arrays only the first is important because it represents more or less the scene and gives rise to the translation of the picture into PROLOG predicates, namely the *conn*'s.

The procedure follows a slightly different course when it is used in the learning phase. The basic idea is to use a certain view of a 3-D figure (which is defined as *frontal*) and produce five further views of the same figure, namely: *back*, *top*, *bottom*, *left-hand* and *right-hand side*. Therefore, the original is transformed as before (translation is not necessary since the original - or frontal view - is always placed at the top-left corner of the projection-plane) until a desired view is obtained. Then the system transforms the figure accordingly and produces the rest of the views, which it places in certain standard positions in the projection-plane. This composition of a scene with the six views is only needed in order to obtain a better image of the relative views of the figure.

3.4.3 The Output

The output covers the *visual* and *structural representation* of the scene. The visual representation can be plotted on a screen or on paper. The user has the option of getting a hard copy of an orthographic projection, or a central projection of the scene, or both. Another facility enables the hidden lines of the scene to appear in the plot in the form of dashed-lines. Finally an optional pair of scaled coordinate axes can be drawn at the user's request. Figures 3.20-3.22 demonstrate the output capabilities of the simulator by a scene consisting of a tetrahedron, a prism, a pyramid and a truncated-pyramid.

Figure 3.23 shows the six views of a prism, drawn by the simulator during the learning phase.

The structural representation consists of a set of *conn-predicates*, which form a description of the scene based on the internal structure of its figures. For every new 3-D figure that is to be added to the scene, a central projection is obtained. This central projection consists of only the visible edges of the figure. Every edge is translated into a *conn*-predicate of the form: *conn(a,b,2)**, which is written into a file called <scene>. At the end of the scene build-up, file <scene> contains all the conn-statements that describe this scene and can be used as input by the recognition-phase. This is actually the basic purpose of the simulator, i.e. to create a scene of 3-D figures, decompose it to its fundamental primitives (lines and points), and present the result to the recognizer. In addition to the file <scene> another file called <coord> is created. This file contains the values of array Vc and is used by the recognizer in order to obtain further information about the individual figures, such as containment of points, convexity, etc. Figure 3.24 illustrates the creation of the two files <scene> and <coord>.

---

* *a and b are the two vertices which define the edge and 2 indicates the number of faces containing the edge (see §5.3.1).*

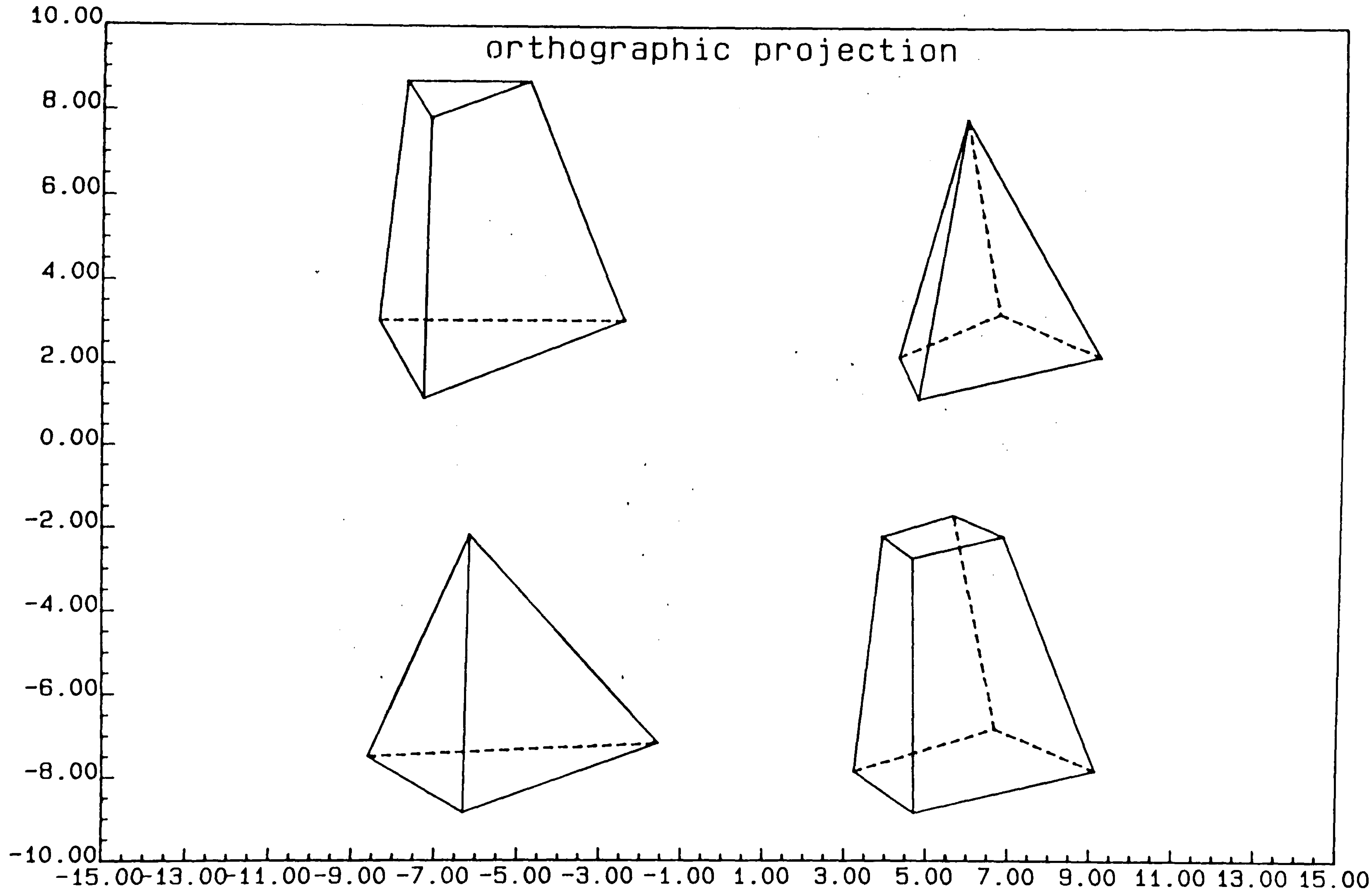

orthographic projection
10.00
8.00
6.00
4.00
2.00
0.00
-2.00
-4.00
-6.00
-8.00
-10.00
-15.00 -13.00 -11.00 -9.00 -7.00 -5.00 -3.00 -1.00 1.00 3.00 5.00 7.00 9.00 11.00 13.00 15.00

FIGURE 3.20

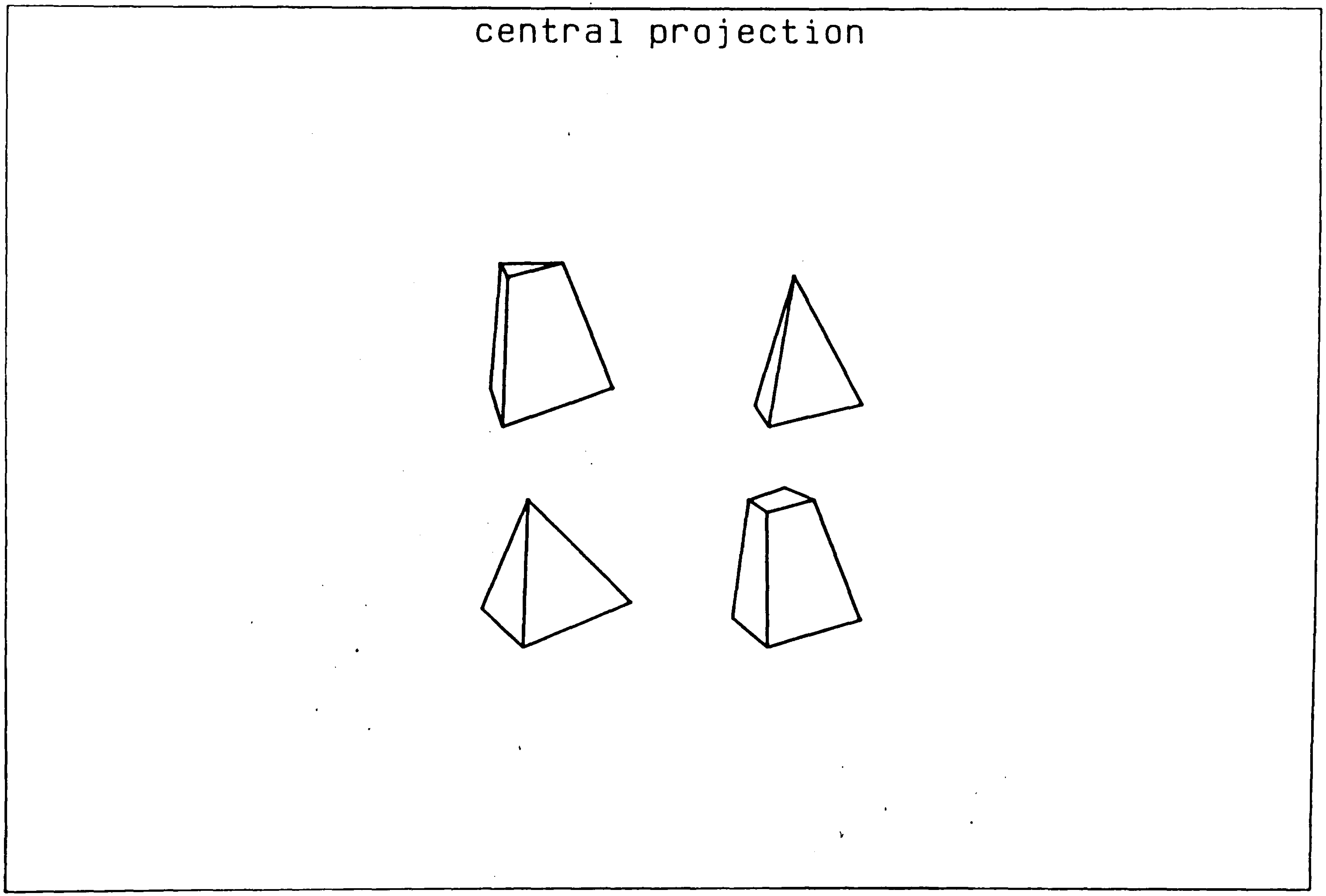


FIGURE 3.21

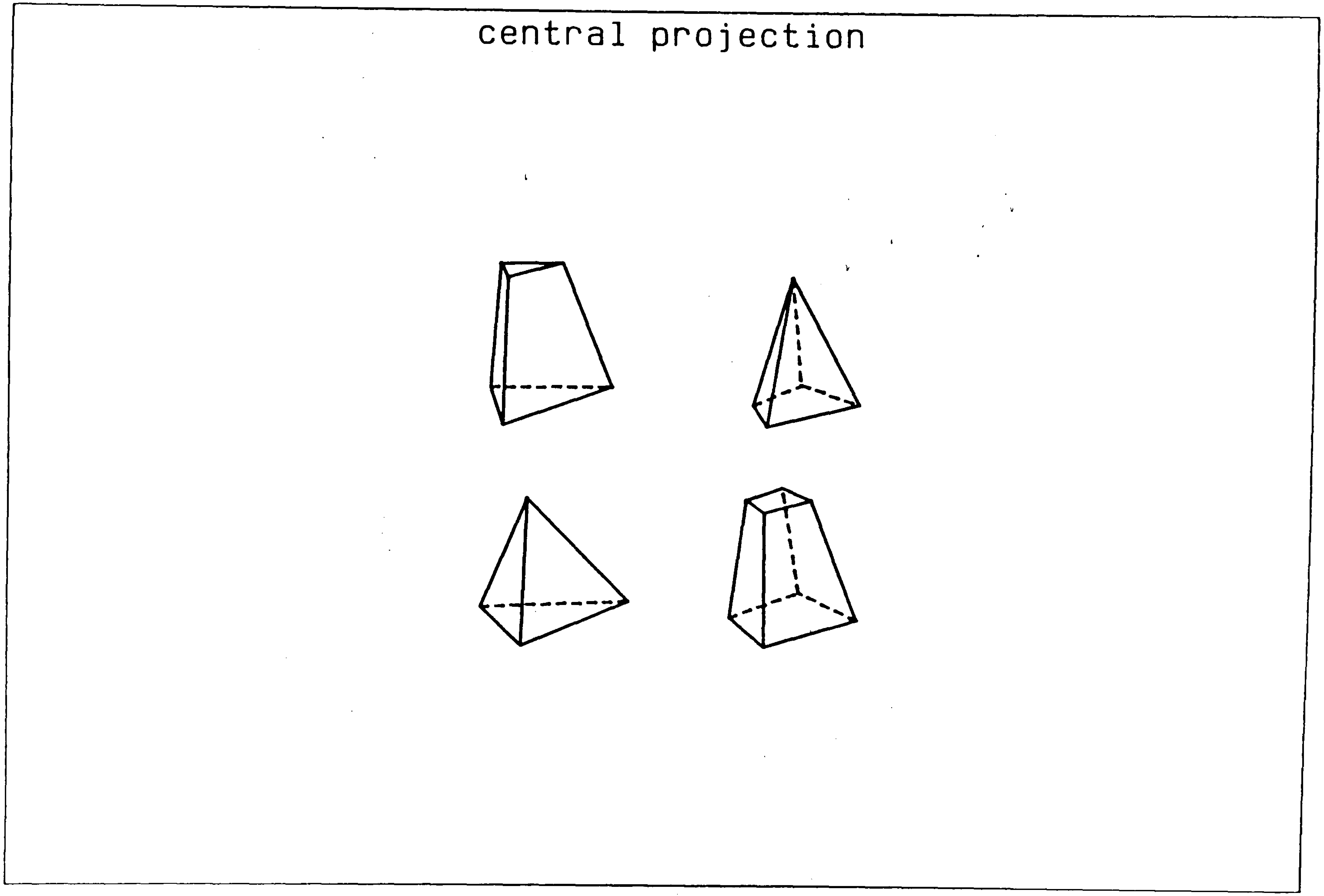
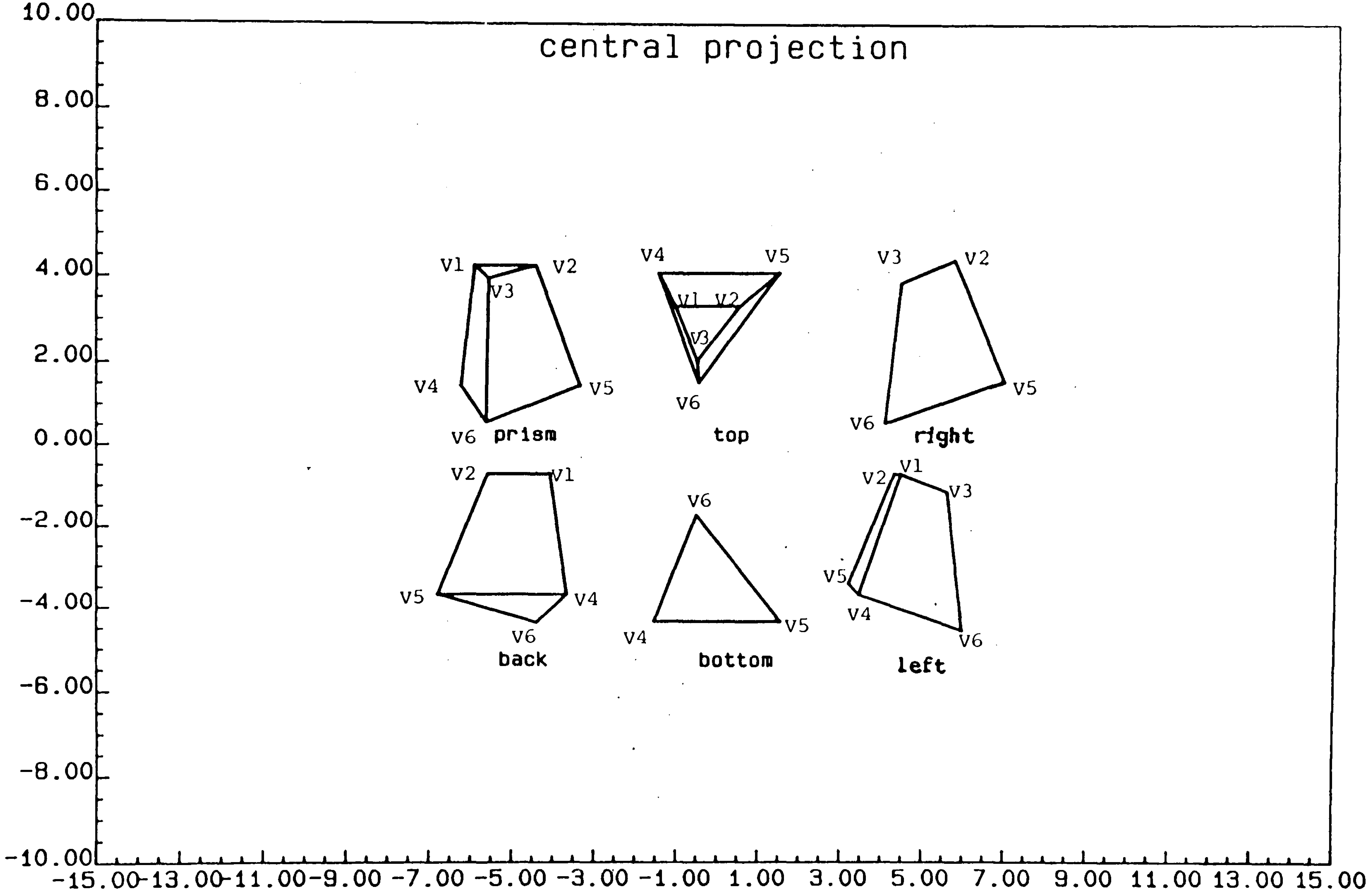
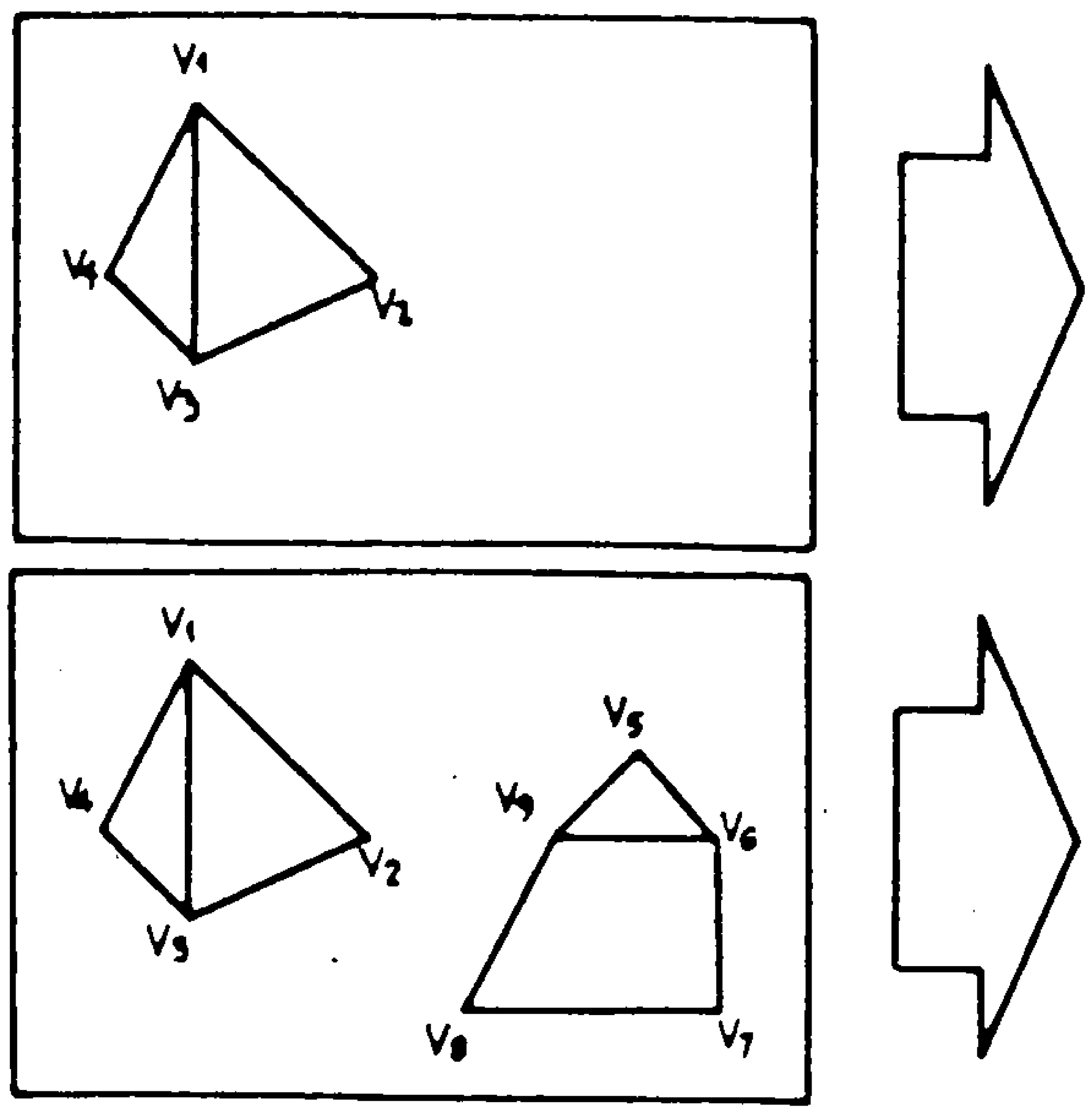


FIGURE 3.22



central projection
prism
top
right
back
bottom
left
V1
V2
V3
V4
V5
V6
10.00
8.00
6.00
4.00
2.00
0.00
-2.00
-4.00
-6.00
-8.00
-10.00
-15.00 -13.00 -11.00 -9.00 -7.00 -5.00 -3.00 -1.00 1.00 3.00 5.00 7.00 9.00 11.00 13.00 15.00

FIGURE 3.23




```
<scene>

conn(v1,v2,2).
conn(v2,v3,2).
conn(v3,v4,2).
conn(v4,v1,2).

conn(v5,v6,2).
conn(v6,v7,2).
conn(v7,v8,2).
conn(v8,v9,2).
conn(v9,v5,2).
```

```
<coord>

v1x  v1y
v2x  v2y
v3x  v3y
v4x  v4y
End
v5x  v5y
v6x  v6y
v7x  v7y
v8x  v8y
v9x  v9y
v10x v10y
End
```

FIGURE 3.24

In the learning phase, the simulator produces a plot with the six basic views of the 3-D figure, the structure of which is to be learnt. Again here, the user has the options of central or othographic projection, hidden lines and axes. The structural output consists of six files which correspond to the six views of the 3-D figure with the following names: *p_(figure_name)*, *p_bac*, *p_top*, *p_bot*, *p_lef*, *p_rig*. (figure-name) depends on the kind of 3-D figure e.g. *tetra*, *pgram*, *prism*, *box*, etc. The format of the data in these files is also different because it should be adapted to the requirements of the learning procedure. The data is a PROLOG-predicate, which contains the name of the 3-D figure, a list of *conn*-predicates describing the structure of this particular view and finally a term determining whether this view is a positive or a negative training instance (see also §5.4.2). The six files corresponding to the six views of Figure 3.23 are given below:

```
<p_prism>
[prism,
 [conn(v1,v2,2),
  conn(v2,v3,2),
  conn(v3,v1,2),
  conn(v4,v6,2),
  conn(v6,v5,2),
  conn(v4,v1,2),
  conn(v6,v3,2),
  conn(v5,v2,2)],
 positive].
```

```
<p_top>
[prims,
 [conn(v1,v2,2),
  conn(v2,v3,2),
  conn(v3,v1,2),
  conn(v4,v5,2),
  conn(v5,v6,2),
  conn(v6,v4,2),
  conn(v1,v4,2),
  conn(v2,v5,2),
  conn(v3,v6,2)],
 positive].
```

```
<p_lef>
[prism,
 [conn(v1,v3,2),
  conn(v3,v6,2),
  conn(v6,v4,2),
  conn(v4,v1,2),
  conn(v1,v2,2),
  conn(v2,v5,2),
  conn(v5,v4,2)],
 positive].
```

```
<p_bac>
[prism,
 [conn(v1,v2,2),
  conn(v2,v4,2),
  conn(v4,v5,2),
  conn(v5,v2,2),
  conn(v6,v5,2),
  conn(v4,v6,2)],
 positive].
```

```
<p_bot>
[prism,
 [conn(v4,v5,2),
  conn(v5,v6,2),
  conn(v6,v4,2)],
 positive].
```

```
<p_rig>
[prism,
 [conn(v3,v2,2),
  conn(v2,v5,2),
  conn(v5,v6,2),
  conn(v6,v3,2)],
 positive].
```

The 3-D figure simulator is a simple and inexpensive way of creating scenes of solids and representing them by line drawings. It allows the user to compose her/his own scenes that act as input to the recognizer. It is also used by the learner to produce alternative views of the objects being learnt.

REFERENCES

1. BARNARD, S.T. and PENTLAND, A.P. 1983: *Three-dimensional Shape from Line Drawings*, Proc. 8th IJCAI, pp. 1062-1064.
2. CHAKRAVARTY, I. 1979: *A Generalized Line and Junction Labelling Scheme with Applications to Scene Analysis*, IEEE Trans. PAMI, April, pp. 202-205.

3. CHIEN, R.T. and CHANG, Y.H., 1974: *Recognition of Curved Objects and Objects Assembly*, Proc. 2nd IJCPR, IEEE Publ. No. 74, CH0885-4C, Copenhagen, August, pp. 465-510.

4. CLOWES, M.B. 1971: *On Seeing Things*, AI 2, 1, Spring, pp. 79-116.

5. COWIE, R.I.D. 1983: *The Viewer's Place in Theories of Vision*, Proc. 8th IJCAI, pp. 952-958.

6. DIXON, A.H. 1977: *Generation of Descriptions for Line Drawings*, Proc. 5th IJCAI, p. 607.

7. GABRIELIDIS, G. 1982: *Recognition of Simple 3-D Objects by the Use of Syntactic Pattern Recognition*, M.Phil. Thesis, Dept. Comp. Studies, Loughborough Uni. Tech., June, pp. 18-71.

8. GILOI, W.K. 1978: *Interactive Computer Graphics: Data Structures, Algorithms, Languages*, Prentice-Hall, pp. 165-184.

9. GONZALEZ, R.C. and WINTZ, P. 1977: *Image Processing*, Addison-Wesley, pp. 115-181.

10. GUZMAN, A. 1969: *Decomposition of a Visual Scene into 3-D Bodies: Automatic Interpretation and Classification of Images*, Ed: Grasseli, E., Ph.D. Thesis, N.York, Academic Press.

11. HARRINGTON, S. 1983: *Computer Graphics: A Programming Approach*, McGraw-Hill, pp. 284-353.

12. HUFFMAN, D.A. 1971: *Impossible Objects as Nonsense Sentences*, in M16.

13. KANADE, T. 1978: *A Theory of Origami World*, CMU-CS-78-144, Comp. Scien.Dept.Carnegie-Mellon Univ.

14. KANADE, T. 1979: *Recovery of the 3-D Shape of an Object from a Single View*, CMU-CS- Comp.Scien.Dept., Carnegie-Mellon Uni., October, pp. 79-153.

15. MACWORTH, A.K. 1973: *Interpreting Pictures of Polyhedral Scenes*, AI 4, 2, June, pp. 121-137.

16. ROBERTS, L.G. 1965: *Machine Perception of 3-D Solids: Optical and Electro-optical Information Processing*, Ed: Tippett et al, Cambridge, MA, MIT Press.

17. SHAPIRA, R. 1974: *A Technique for the Reconstruction of a Straight-edge, Wire-frame Object from Two or More Central Projections*, CGIP 3, 4, December, pp. 318-326.

18. SHAPIRA, R. and FREEMAN, H. 1977: *Reconstruction of Curved-Surface Bodies from a Set of Imperfect Projections*, Proc. 5th IJCAI, pp. 628-634.

19. SUGIHARA, K. 1981: *Mathematical Structures of Line Drawings of Polyhedra*, RNS-81-02, Dept.Info.Scien., Nagoya Uni., May.

20. THORPE, C. and SHAFER, S. 1983: *Correspondence in Line Drawings of Multiple Views of Objects*, Proc. 8th IJCAI, pp. 959-965.

21. TURNER, K.J. 1974: *Computer Perception of Curved Objects Using a T.V. Camera*, Ph.D. Thesis, Univ. Edinburgh.

22. WALTZ, D.I. 1972: *Generating Semantic Descriptions from Drawing of Scenes with Shadows*, Ph.D. Thesis, AI Lab., M.I.T., Also PCV 1975.

CHAPTER 4

A SURVEY OF MACHINE LEARNING

4.1 INTRODUCTION

Machine learning refers to any automated improvement in the performance of a computer system over time, as a result of experience [Forsyth '84]. Machine learning research is directed towards two basic streams. The *application* stream that develops learning systems capable of achieving certain tasks, and the *scientific* stream that explores alternative learning mechanisms, discovers new induction algorithms and tests the performance of the different methods. The basic characteristics of machine learning systems are the *representation of knowledge* that depends on the acquired type of knowledge, the *learning strategy* depending on the amount of inference performed by the learner, and the *domain of applications*. These three main characteristics may act as the key-concepts for classifying machine learning research.

A general model of a learning system is proposed by [Smith et al '77] and consists of the following components: The *performance element* that generates an output in response to a training instance selected by the *instance selector*. The *critic* that analyses the output of the performance element, and the *learning element* that makes the suitable changes to the system based on the results of the critic-analysis. A *blackboard* is responsible for communication among the functional components and ensures their access to the current information. Finally all constraints, assumptions and methods that define the domain of activity of the system make up the *world model* (Fig. 4.1). Approaches to learning systems are divided into *adaptive* that give emphasis on *parameter learning* and use statistical methods to achieve optimal performance, and *artificial intelligence* influenced that believe in a sufficient internal structure and a strong knowledge base.

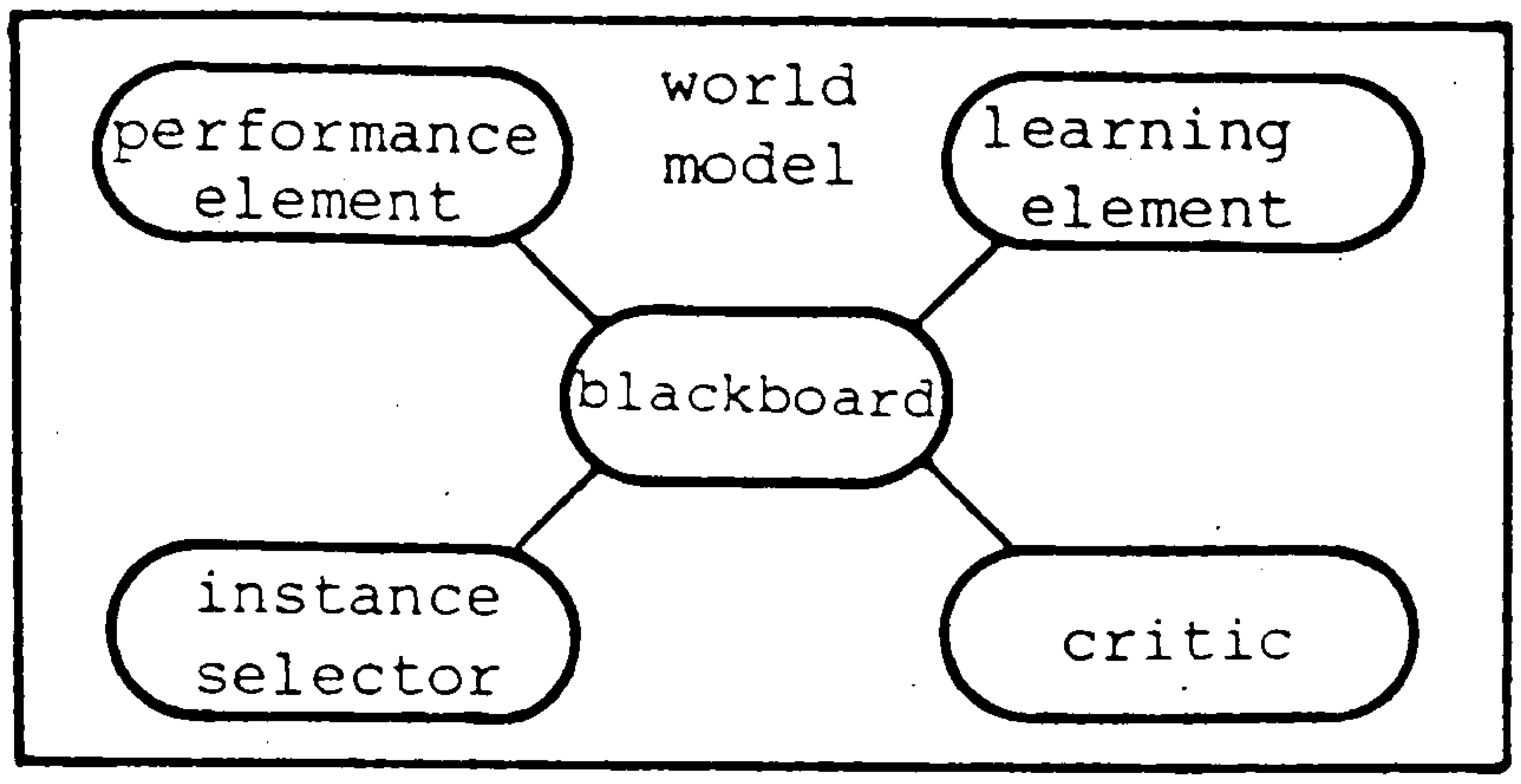


FIGURE 4.1

A historical sketch of machine learning [Carbonell et al '83] divides the period of research according to the following three paradigms:

neural modelling and decision-theoretic techniques with main feature the building of general purpose learning systems based on very little (or no) initial task-oriented knowledge,

symbolic concept-oriented learning using logic or graph structure representations for high level knowledge making strong assumptions about the concepts to be acquired, and finally,

knowledge-intensive learning that reflects the latest trend that began with investigating a large number of learning methods based on knowledge-rich systems. In the following sections a survey of recent research of machine learning is presented. The first part covers the learning strategies, the second part refers to the knowledge representation and the third part examines some special purpose programs.

4.2 LEARNING STRATEGIES

The simplest form of learning is *rote learning*, which is a direct acquisition of knowledge without any inference or any other transformations of information by the learner. The term 'rote learning' is used primarily for *memorizing of facts and data*, or for learning by *being programmed*, where the programmer is the only source of modifications in the systems. The rest of learning strategies imply some degree of influence and require an amount of effort by both the learner and the teacher.

4.2.1 Learning from Instruction

The task of machine learning by instruction is to build a system that acquires knowledge from an organized source (e.g. teacher) and then uses it effectively by integrating it with its prior knowledge. The main problem of the method is to transform the incoming new information from the input language to a system-acceptable representation in order to combine it optimally with the information already existing in the system.

A potentially useful kind of learning from instruction is by accepting high-level advice. The advice is most of the time non-operational, which means that before it is used it must be made operational (i.e. directly executable by the learner). This is performed by a process called *operationalization*. UNDERSTAND [Simon '77] is one of the first programs that uses operationalization. It reads an English description of the Tower of Hanoi problem and it transforms it to a means-ends analysis problem. [Mostow '81] regards operationalization as converting domain knowledge into intelligent programs. He takes a problem expressed in the language of a particular task domain, together with prior knowledge about the domain, and transforms them to fit a general computational method like heuristic search. The operators that perform the transformations are implemented in a program called FOO. The problem is represented using a LISP-like language. Domain knowledge is encoded as definitions of concepts represented as functions. To reason about the problem on the basis of such domain knowledge inference methods are used represented as *problem transformation rules*. Each rule consists of a left-hand, a right-hand pattern and a condition. Rule conditions are tested by simple procedures or by generating sub-problems that are solved by a sequence of rules. The control knowledge required for operationalization is generated interactively. The advice is mapped onto a general method (heuristic search) by representing it as a *data-flow graph*, in which boxes represent generators and tests to be filled in from the current problem. Operationalization of a problem in terms of a method is performed through some transformation rules which represent the knowledge of how to map the problem to the method. FOO proceeds by instantiating components of the general data-flow graph and by refining the procedure based on domain knowledge and reasoning methods. The method is applied to a card-game and its generality is tested successfully by composing a music-piece.

KLAUS (Knowledge Learning And Using Systems) are machine learning systems that can communicate with the user in English about a specific domain of interest, retrieve and display data inserted by the user, and apply several sorts of external software systems in order to solve user problems. A pilot KLAUS called NANOKLAUS developed by [Haas & Hendrix '83] consists of the following principle components: a *natural language* processing module which uses a pragmatic grammar consisting of highly specific, special purpose rules for processing various types of sentences; a *formal deduction* module which uses first-order logic as a knowledge representation scheme, a general and flexible scheme used by many problem-solving systems; and a number of supporting procedures used to assimilate knowledge about new object domains and to maintain the current database. The main objective of the system is to acquire information about domains with which it is unfamiliar from people who are experts in the same domains, but have limited training in computer science. NANOKLAUS is equipped with a fixed set of semantic and syntactic rules that cover a small subset of the English language, as well as with *seed concepts* and a *seed vocabulary*, which is extended as the system learns about new domains. Concept acquisition is a procedure of introducing new concepts by the user and progressively retaining them by relating these to other concepts, rather than directly defining the concepts. As the system is presented with a new concept, it calls a special routine that assumes a dialogue between itself and the user seeking additional information required to assimilate the new concept. Every new fact is checked to determine its consistency with previous knowledge. The representation system is based on a sort hierarchy (tree structure, where any sort may have multiple ancestors) and information about the immediate ancestors and descendants of each sort is maintained. NANOKLAUS despite its limited capabilities marks the beginning for further research on *learning by being told*.

Instructible Production Systems (IPS) [Rychener '83] are learning systems that are built by gradual instruction rather than by deliberate programming. The intelligence of such systems results from their ability to deal with more situations, as their size increases. Knowledge acquisition is obtained by a dialogue between the instructor and the system that is ruled by a number of constraints. These constraints embody the essence of instructions as it occurs in certain natural situations. IPS use *production systems* as their underlying knowledge organization and obtain behaviour via a simple *recognize-act cycle* with a sophisticated set of principles for resolving rule-conflicts. They possess a dynamic short-term memory called the *working memory* which is used to transfer information from the environment to the system, and a long-term memory called the *production memory* which contains the rules. During a cycle a match between the contents of the working memory and the rule-conditions of the production memory is attempted. Learning in an IPS is a process of compiling fairly specific rules and methods. A *method* is a set of rules that work together to satisfy a goal. Methods consist of a number of steps and are specialized to a certain goal class. A *rule* (or production) consists of a set of conditions and a set of actions. Conditions are patterns that match elements of the working memory, such as goals and structures describing environmental or internal states. A *goal* is a data structure representing an external command, or an internal need to achieve some state, or to execute successfully a sequence of actions. Unlike most other approaches, IPS seek temporary solutions to cope with problematic tasks, rather than doing long-term planning and anticipation of difficulties. The design of IPS is organized around the following functional components: *interaction language*, *organization of procedural elements*, *accommodation of new knowledge*, *connection of goals with system capabilities*, *explanation of system behaviour*, *evaluation of behaviour*, *reformulation of knowledge* and *compilation to achieve efficiency and automacity*. Each of these components can be interpreted as a dimension along which learning systems can vary. The IPS work has a closer relation to intelligent computer-aided instruction and to the construction of expert systems than other approaches.

4.2.2 Learning by Analogy

This type of learning acquires new facts and skills by exploiting their similarity to existing knowledge about these facts and skills based on past experience.

LEX is a program [Mitchell et al '81] that incorporates *domain-independent* methods to discover *domain-dependent* problem solving heuristics through practice. It begins with a heuristic problem solver without heuristics in the mathematical space of indefinite integrals and it is supplied by the normal integration and algebraic equivalence rules. Its goal is to derive a problem state that contains no unsolved integrals. The problem is decomposed into four main tasks, each one corresponding to a separate module in LEX. The *problem solver* utilises the currently available operators and heuristics to solve a given practice problem, and keeps a trace of the search performed during the solution process. The *critic* analyses the search of the problem solver and comes out with a set of positive and negative training instances, corresponding to successful or unsuccessful steps in the course of solving the problem. The *generalizer* generalizes the training instances provided by the critic and suggests new refined heuristics for more effective problem solving. Finally a *problem generator* produces a new practice problem to be treated by the current level of the system expertise. The weaknesses of this system lie in the representation language (that has to be learnt) and the inappropriate generalizations.

Acquisition of proof skills in geometry [Anderson '81] is used as part of a simulation system (ACT) that investigates basic mechanisms of human cognition. A number of rules is declaratively encoded into a schema representation to which general problem-solving productions can apply. *Knowledge compilation* is the process that transforms each rule into a procedural form in two stages: *composition* and *proceduralization*. *Analogy* to worked out problems is one of the mechanisms that improve the learning performance of the system. The importance of analogy as a powerful computational mechanism is also noted by [Carbonell '83], who presents an analogical inference engine based on two fundamental hypotheses:

a) Problem-solving and learning are inalienable aspects of a unified cognitive mechanism, and

b) The same learning mechanisms that account for concept formation in declarative domains, operate in acquiring problem solving skills and formulating generalized plans.

The main principle of the system is to gradually transform an existing solution of a problem into one that satisfies the requirements of a new problem. The analogical problem-solving problem is based on an extension of the means-ends analysis (MEA) and consists of two phases.

The first phase is called *reminding* and works by recalling a previously-solved problem whose solution may transfer to the new problem under consideration. A difference function (same as in MEA) is used as a *similarity metric* to retrieve the solution of a previously-solved problem resembling the present problem. The second phase is faced with the problem of finding an appropriate analogical transformation that transfers the old solution sequence into one satisfying the criteria of the new problem. Therefore it uses a different problem space called, *analogy transform problem space* (T-space) defined as follows:

a) States in the T-space are potential solutions to problems in the original problem.

b) The initial state in the transform space is the solution to a similar problem retrieved by the reminding process.

c) A goal state in the transform space is the specification of a solution that solves the new problem, satisfying its path constraints.

d) An operator in the T-space (T-operator) maps an entire solution sequence into another potential solution sequence.

e) The *difference metric* (D_T) in the T-space is a 4-value vector consisting of the difference measures between initial states, final states, path constraints and degree of applicability on the retrieved solution in the new problem environments.

f) A difference table for indexing the T-operators, which are ordered according to their measure of utility in reducing the given difference.

g) There are no path constraints in the transform space (for simplification reasons).

Analogical transformation provides a method that can exploit prior experience effectively. [Scott & Vogt '83] describe a task-oriented learning system (PAN) that tries to improve its performance by reducing its own uncertainty regarding the outcome of its actions. The system builds an organized representation of its experience and is implemented as a multiple concept learning task from noisy data. An interesting method for learning by practice is discussed by [Arya '83] based on knowledge transfer between domains. An application uses the systems experience in figure symmetry to solve elementary physics problems. This is achieved in four steps:

a) mapping certain components of physics problems into the domain of figures.

b) applying the available knowledge on that domain.

c) mapping the results back into the original domain and

d) testing the validity of the transfer.

LS-1 is a learning system [Smith '83] that acquires problem solving heuristics through experience. It maintains a knowledge base of structures, each being a candidate production set (PS) for solving the task at hand. A cycle through the learning loop begins with each PS applied by a *program solving component* to n instances of the task. A *critic* analyses the n operator sequences generated by the problem solver and assigns a performance measure indicative of the relative worth of the PS as a potential solution of the task. Once all structures in the knowledge base have been evaluated, a *genetic algorithm* constructs a new knowledge base of structures for testing and the cycle is repeated. The knowledge base of PS, together with the associated performance measures, is viewed as LS-1's internal memory representing the sum of the system's experience in the task domain at any point in time. LS-1's current hypothesis to a solution of a task is the PS that has been highest rated by the critic so far in the search. The system's progress is monitored by considering the sequences of hypotheses generated over time.

4.2.3 Learning from Observation and Discovery

This is a very general form of inductive learning that lacks the benefit of an external instructor (also known as *unsupervised* learning). The learner is required to perform more inference than in the previous methods and may need to concentrate on several concepts that need to be acquired. According to the degree of interaction with the external environment unsupervised learning is subdivided into *passive observation* and *active experimentation*. In the first one the learner is limited to classification and taxomony of observations, while the second one uses results of experimental strategies in order to test its theories and gradually changing hypotheses.

BACON.4 [Langley et al '83] is one of a series of systems which learn from discovery. It is a production system that discovers descriptive laws that summarize data. Information is presented at varying levels, from which the lowest is called *data* and the highest *hypotheses*. A description at one level acts as an hypothesis with respect to the descriptions below it, and as a datum for the description above it, while all intermediate levels are hybrids of the lowest and the highest level. The program uses a small set of heuristics, stated as productions, in order to formulate hypotheses and to define theoretical terms based on these regularities. *Theoretical terms* are combinations of directly observable variables. The search for regularities is directed by heuristics called *trend detectors*. The program has the ability to postulate new intrinsic properties which may be associated with independent terms taking nominal values. The BACON.4 heuristics, are a general mechanism applicable to discovery in several domains such as physics, chemistry, etc.

The use of heuristics to guide learning by discovery is examined by [Lenat '83]. He presents a plan of a research program for coping with the problem of automatic knowledge acquisition, which consists of five main points:

a) *New domains of knowledge can be developed by using heuristics.* To accomplish this the system requires heuristics of varying levels of generality and power, an adequate knowledge representation and some initial hypotheses about the nature of the domain.

b) *As new domains of knowledge emerge and evolve, new heuristics are needed.* Since a field may change, the corpus of heuristics dealing with that field may also change.

c) *New heuristics can be developed by using heuristics.* This is achieved by considering heuristics as a domain of knowledge and using a large set of heuristics, an adequate representation for heuristics and some hypotheses for their nature.

d) *As new domains of knowledge emerge and evolve, new representations are needed.* The representation scheme used for a certain domain must be evolved as domain knowledge accrues.

e) *New representations can be developed by using heuristics.* Since representation of knowledge is a field, heuristics can be allowed to manage the development of new representations.

The system uses a single control loop for both using and acquiring knowledge. The corpus of heuristics and representations is continuously modified until the system reaches a kind of equilibrium. If at this stage new discoveries upset this equilibrium a set of meta-heuristics detects it and new representations and heuristics are pursued. Two programs AM and EURISKO demonstrate the use of heuristics in knowledge acquisition and development of new heuristics respectively.

Learning from observation can be regarded as a classification of a set of objects and situations. [Michalski & Stepp '83] describe a new technique called *conceptual clustering* in which a configuration of objects forms a class only if it is described by a concept from a pre-defined class. The clustering is *conjunctive* if the predefined concept class consists of conjunctive statements involving relations on selected object attributes. The method arranges objects into a hierarchy of classes closely defined by conjunctive descriptions. The goal of the process is to construct a *concept network* characterizing the above objects, with nodes representing concepts describing object classes and links representing the relationships between the classes. Classes of objects are created by using a numerical measure of *similarity* of these objects. The method is demonstrated by program CLUSTER/2 that classifies a collection of spanish folk songs. Another system [Rajamoney et al '85] attempts to continuously update its model through constant monitoring of the real world. It begins with a world model of its domain and a number of implicit beliefs in its structure, that drives its reasoning. When situations of contradiction occur, the system questions its beliefs starting with those closer to the contradiction. If these fail to give a consistent explanation, the system questions secondary causes of the contradiction i.e. beliefs supporting the primary beliefs as well as those behind the questioning and investigating process. The underlying assumption is that the number of errors in the model is small. Once a faulty belief is detected through a number of experiments, the model is revised to be consistent with the current observations. The system uses a planner to predict the observations it will make when the plans are executed. The predictions are constantly compared with the actual observations. When it finds changes in the world that are not a consequence of some plan, it tries to explain them as an effect of processes running independently from the system. The system is used to learn chemical phenomena.

4.3 KNOWLEDGE REPRESENTATION

One of the basic needs of machine learning systems is the organization of large amounts of knowledge and the development of mechanisms for manipulating that knowledge effectively. Therefore, a variety of ways to represent-knowledge has been devised to allow for more specific and more powerful inference algorithms that operate on it. All these different knowledge representation models deal with two main kinds of entities:

Facts, which are the objects of representation, representing *truths* in a relevant environment, and *representation of facts* in a certain formalism, which are the things to manipulate. Good representations are also functions mapping facts to representation elements and vice versa (Fig.4.2).

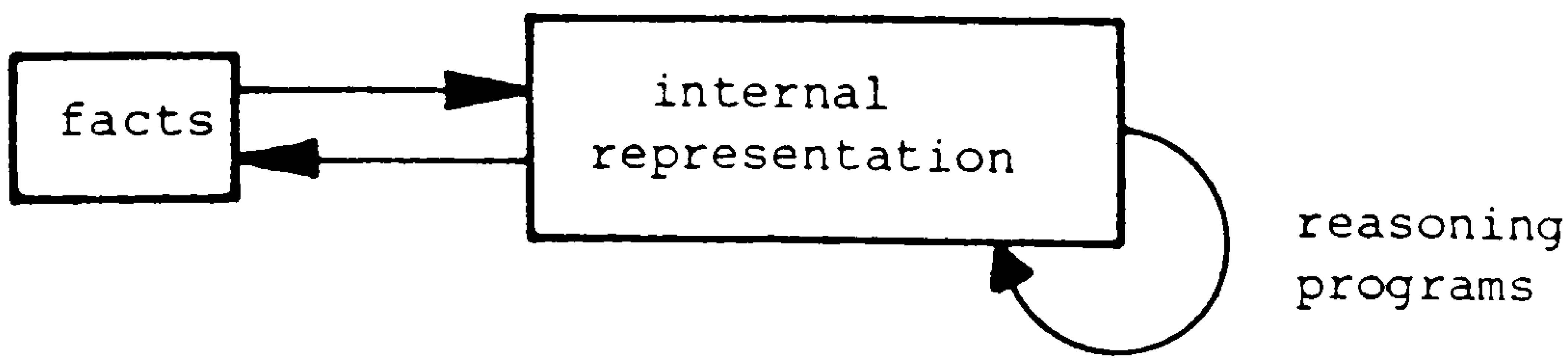


FIGURE 4.2

For the convenience of the user some representation systems use also a *natural language* representation.

4.3.1 Knowledge Representation Using Logic

The logical formalism is a powerful mechanism based on mathematical deduction. The truth of a new statement can be proven by using statements the truth of which is already known. *Predicate logic* is a knowledge representation that allows real-world facts to be represented as *statements* written as *well-formed formulas* (wff's): *predicate-name (variable-list)*, or a:- $c_1, c_2, \ldots, c_n$, meaning that a is true only if the conjunction of *conditions* $c_1, c_2, \ldots, c_n$ is also true. The advantage of using predicate logic to represent knowledge is that it provides a good way of reasoning with that knowledge. *Resolution* is an iterative procedure that produces proofs, operating on statements converted to a standard form called: *conjunctive normal form* [Davis & Putnam '60]. A new statement is proved by showing that its negation produces a contradiction with the known statements. Resolution begins with two clauses, called *parent clauses*, that contain the same literal in positive and in negative form respectively. The two clauses are compared *(resolved)* and a new clause is obtained *(resolvent)* consisting of the literals of the parent clauses except the ones that cancel one another *(complementary literals)*, (Fig.4.3). Then the resolvent is *unified* and if it is the

parent clauses   resolvent

$$\left.\begin{array}{c|c} a & b \\ \\ \bar{a} & c \end{array}\right\} \longrightarrow \quad b \mid c$$

FIGURE 4.3

empty clause contradiction is met, otherwise it is added to the set of available clauses. Unification is performed by instantiating the variables of the literals with the same value. The choice of parent clauses is determined by one of the following methods: *set-of-support* strategy (contradiction involves currently tested statements), *unit-preference* strategy (prefer clauses with single literal), *clause-elimination* strategy (drop tautologies or subsumed clauses), and *negative-predicate* strategy (resolve clauses with complementary literals). [Rich '83a]. *Natural deduction* is a combination of techniques to solve problems that are not tractable by any one method alone (e.g. arranging knowledge by objects involved in predicates) [Boyer & Moore '79].

Sometimes problem solving involves uncertain and fuzzy knowledge. Some of the techniques that deal with such problems are the following:

Nonmonotonic logic [Reiter '80] which allows deleting and/or adding of statements from/to the database, *probabilistic reasoning* [Szolovits & Pauker '78], which deals with representing likely but uncertain inferences, *fuzzy logic* [Zadeh '75], which makes it possible to represent fuzzy or continuous properties of objects, and *belief spaces* [Grosz '77] which provides a way of representing nested models of belief-sets.

4.3.2 Structured Knowledge Representations

Knowledge structures are data structures that contain a complete database of information about particular problem domains. The systems that are used to represent structured knowledge [Rich '83b], satisfy a number of basic requirements such as: The ability to represent all forms of knowledge in a certain domain, inferential efficiency and adequacy and efficiency in acquiring new information. The techniques employed by these systems can be divided into two main categories *declarative* and *procedural*. The declarative methods represent knowledge as a static collection of facts and include a number of procedures for manipulating them. Some of the most common of them are: *Semantic nets* (see §2.4.1), that are able to describe both events and objects, *conceptual dependency*, representing relationships among the components of an action, *frames*, a general structure for representing complex objects, and *scripts*, a more specialized structure used to represent common sequences of events. The above type of structures rely on kinds of descriptions called *schemas*. Some common characteristics of these structures are, the notion of complex entities as a collection of attributes and associated values, and the use of *associative memory*. Two major advantages of declarative representations are, their ability to store facts only once, and that it is easy to augment their information base without having to do any changes.

The procedural methods represent the bulk of the knowledge as procedures for using it. Some of the major advantages of these methods are, their efficiency in representing knowledge of how to do things, their ability to use heuristics for the same reason, and their ability to represent knowledge where the declarative methods fail. Structures seem an easy way to represent knowledge independently of how this knowledge is obtained. However, for most domains a combination between structured and declarative representation of knowledge is needed.

4.4 SPECIAL LEARNING SYSTEMS - GAMES

Learning by doing is a combination of learning by being told and learning by exploration. It allows the instructor of the system to watch and advise the system while it is solving problems in its chosen domain of expertise [Anzai & Simon '79]. The instructor can verify that the new knowledge is appropriate to the system's current knowledge due to the fact that the system is doing something and at the same time can receive instruction. In principle, there is an interaction between new and old knowledge in a situation where this knowledge is being applied. [Ohlsson '83] describes a system (Universal Puzzle Learner) that learns problem solving heuristics by doing. UPL consists of: a *production system language* that represents productions with words, a task-independent, weak *problem solver* working by analysing the current goal into *subgoaling rules*, and a *learning mechanism* that assembles instantiated productions which correspond to rule instances that should have fired but failed to do so.

A prototypical approach to machine learning based on the fundamental process of *categorization* is presented by [Phelps & Musgrove '85]. The method uses a knowledge representation scheme adopted for objects and categories that fit in with the prototype theory. It decomposes 2-D silhouettes of objects into convex parts (holistically) and applies clustering in order to categorize these parts based on similarities of individual parts.

Games provide a very good domain for exploring machine learning because they are structured tasks in which it is easy to measure failure or success and because they do not require large amounts of knowledge. [Samuel '63] built a program for playing checkers that improved its performance by learning from its mistakes. This program was later able to beat its creator! Computer game-playing strategies rely on search-based techniques and involve a great deal of heuristics to reduce the number of examined moves [Rich '83c]. The main components of such programs are a *plausible-move generator* that generates a small number of promising moves, and a *static evaluation function* that uses the available information to evaluate certain moves by estimating how likely they can lead to a victory. Several games have been explored with a view to machine learning. The first, checkers [Samuel '59], used a decision-theoretic approach based on parameter learning (§4.2.3). *Chess* has been addressed by several researchers. A survey of chess computer programs can be found in [Berliner '78]. [Quinlan '80] presents an algorithm (ID3) that learns chess end-games by generating rules from examples and is the foundation for the expert system shell

EX-TRAN7 [Allwood et al '85]. *Backgammon* is a game involving a chance element and is treated by [Berliner '80]. Finally, a program about *bridge* bidding is presented by [Stanier '75].

The above survey leads to a number of interesting conclusions about the course of machine learning research:

a) old fashioned general purpose, knowledge-poor systems have been replaced by new knowledge-rich ones that use task-oriented knowledge.

b) Several current machine learning systems incorporate heuristics to control their focus of attention by directly planning their knowledge acquisition.

c) A wide variety of new methods, such as learning from instruction, by analogy or by observation and doing have been investigated.

d) The classic method of learning from examples (see §5.2), can be combined with some of the new ideas to provide a powerful basis for machine learning.

REFERENCES

1. ALLWOOD, R.J., STEWART, D.J., HINDE, C.J. and NEGUS, B. 1985: *Report on Expert System Shells Evaluation for Construction Industry Applications*, internal report, Loughborough U.T., August, pp. EXTRAN7 1-13.

2. ANDERSON, J.R. 1981: *Tuning of Search of the Problem Space for Geometry Proofs*, Proc. 7th IJCAI, pp. 165-170.

3. ANZAI, Y. and SIMON, H.A. 1979: *The Theory of Learning by Doing*, Psychological Review, Vol.86, No.2, pp. 124-140.

4. ARYA, A.A. 1983: *Learning by Controlled Transference of Knowledge Between Domains*, Proc. 8th IJCAI, pp. 439-443.

5. BERLINER, H.J. 1978: *A Chronology of Computer Chess and Its Literature*, AI, Vol.12, No.1.

6. BERLINER, H.J. 1980: *Backgammon Computer Program Beats World Champion*, AI, Vol.14, No.1.

7. BOYER, R.S. and MOORE, J.S. 1979: *A Computational Logic*, Academic Press, N. York.

8. CARBONELL, J.G. 1983: *Learning by Analogy: Formulating and Generating Plans from Past Experience* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 137-161.

9. CARBONELL, J.G., MICHALSKI, R.S. and MITCHELL, T.M. 1983: *An Overview of Machine Learning* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 14-16.

10. DAVIS, R. and PUTNAM, H. 1960: *A Computing Procedure for Quantification Theory*, Journal ACM, Vol.7.

11. FORSYTH, R. 1984: *Machine Learning Strategies in Expert Systems: Principles and Case Studies*, Ed: Forsyth, Chapman and Hall Computing, p. 153.

12. GROSZ, B.J. 1977: *The Representation and Use of Focus in a System of Understanding Dialogs*, Proc. 5th IJCAI, pp. 67-76.

13. HAAS, N. and HENDRIX, G.G. 1983: *Learning by Being Told: Acquiring Knowledge for Information Management* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 405-427.

14. LANGLEY, P., BRADSHAW, G.L. and SIMON, H.A. 1983: *Rediscovering Chemistry with the BACON System* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 307-329.

15. LENAT, D.B. 1983: *The Role of Heuristics in Learning by Discovery: Three Case Studies* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 243-306.

16. MICHALSKI, R.S. and STEPP, R.E. 1983: *Learning from Observation: Conceptual Clustering* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 331-363.

17. MITCHELL, T.M., UTGOFF, P.E., NUDEL, B. and BANERJI, R. 1981: *Learning Problem-Solving Heuristics Through Practice*, Proc. 7th IJCAI, pp. 127-134.

18. MOSTOW, D.J. 1981: *Mechanical Transformation of Task Heuristics into Operational Procedures*, Ph.D. Thesis, Carnegie-Mellon Univ.

19. OHLSSON, S. 1983: *A Constrained Mechanism for Procedural Learning*, Proc. 8th IJCAI, pp. 426-428.

20. PHELPS, R.I. and MUSGROVE, P.B. 1985: *A Prototypical Approach to Machine Learning*, Proc. 9th IJCAI, pp. 698-700.

21. QUINLAN, J.R. 1983: *Learning Efficient Classification Procedures and Their Application to Chess End Games* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 463-482.

22. RAJOMONEY, S., DEJONG, G. and FALTINGS, B. 1985: *Towards a Model of Conceptual Knowledge Acquisition Through Experimentation*, Proc. 9th IJCAI, pp. 688-690.

23. REITER, R. 1980: *A Logic for Default Reasoning*, AI, Vol.13, April.

24. RICH, E. 1983: *Artificial Intelligence*, McGraw-Hill, a: pp. 157-166, b: pp. 215-242, c: pp. 113-131.

25. RYCHENER, M.D. 1983: *The Instructible Production System: A Retrospective Analysis* in Machine Learning: An AI Approach, Eds: Michalski, Carbonell, Mitchell, Tioga, pp. 429-459.

26. SAMUEL, A.L. 1959: *Some Studies in Machine Learning Using the Game of Checkers*, IBM Journal of Research and Development, No.3, pp. 211-229.

27. SAMUEL, A.L. 1963: *Some Studies in Machine Learning Using the Game of Checkers* in Computers and Thought, Eds: Feigenbaum, Feldman, McGraw-Hill, N.York, pp. 71-105.

28. SCOTT, R.D. and VOGT, R.C. 1983: *Knowledge Oriented Learning*, Proc. 8th IJCAI, pp. 432-435.

29. SIMON, H.A. 1977: *AI Systems that Understand*, Proc. 5th IJCAI, pp. 1059-1073.

30. SMITH, S.F. 1983: *Flexible Learning of Problem Solving Heuristics Through Adaptive Search*, Proc. 8th IJCAI, pp. 422-425.

31. SMITH, R.G., MITCHELL, T.M., CHESTER, R.A. and BUCHANAN, B.G. 1977: *A Model for Learning Systems*, Proc. 5th IJCAI, pp. 338-343.

32. STANIER, A. 1975: *BRIBIP: A Bridge Bidding Program*, Proc. 4th IJCAI, pp. 374-378.

33. SZOLOVITS, P. and PAUKER, S.G. 1978: *Categorical and Probabilistic Reasoning in Medical Diagnosis*, AI, Vol.11.

34. ZADEH, L.A. 1975: *Fuzzy Logic and Approximate Reasoning*, Synthese, Vol. 30.

CHAPTER 5

THE FIGURE LEARNER

5.1 INTRODUCTION

Inductive learning is the process of acquiring knowledge by drawing *inductive inferences* from facts provided by a teacher or the environment [Michalski '83]. This method of learning appears in two major forms: *learning by example* and *learning from observation*, and its study and modelling is a central topic in machine learning. The applications of inductive-learning programs can be divided into two main categories: automated construction of knowledge bases for learning systems (e.g. expert systems), and technique improvement of knowledge-acquisition methods. The problem of learning by example is to find plausible general descriptions explaining a given database, using a set of positive (examples) and negative (antiexamples) training instances. The basic idea is to start with a set of specific descriptions, using an appropriate description language, and gradually produce a more general set by a repeated application of generalization rules. The method uses a *description space* consisting of *characteristic descriptions* (common properties of a class) and *discriminant descriptions* (distinguishing properties between classes), and employs a *representational system* (e.g. predicate calculus, semantic nets, frames, etc.) to describe events and their generalization. The generalization rules are *constructive* or *selective*, depending on whether they involve assertions of new *descriptors* (functions, variables or predicates used by descriptions) or not. Finally the search for generalized descriptions is controlled by a *bottom-up* (data-driven), *top-down* (model-driven) or *mixed* method.

The first part of this chapter surveys several learning by example programs. The second part contains the definitions of the figures that are to be learnt by the learner. The third part analyses the function of the routines that make up the learning program called *the learner*.

5.2 LEARNING BY EXAMPLE - A SURVEY

The well known work by [Winston '70, '75] deals with learning structural descriptions of simple toy block constructions. The method proceeds by developing conjunctive generalizations of an input of positive and negative examples ('near misses'). An 'intelligent' teacher decides the kind and order of presentation of the examples. Input events, existing knowledge and concept descriptions generated by the program are represented by semantic networks (see also §2.4.2). The use of nodes in the network is threefold. They represent: primitive concepts that are properties of objects or their parts (constants), individual examples and their parts (quantified variables), and link types. The nodes are connected with labelled links representing binary relations between them. Special *(a-kind-of)* links join all the nodes into one generalization hierarchy, used to implement the *climbing generalization tree rule*. Figure 5.1b shows the semantic network representation of scene S.

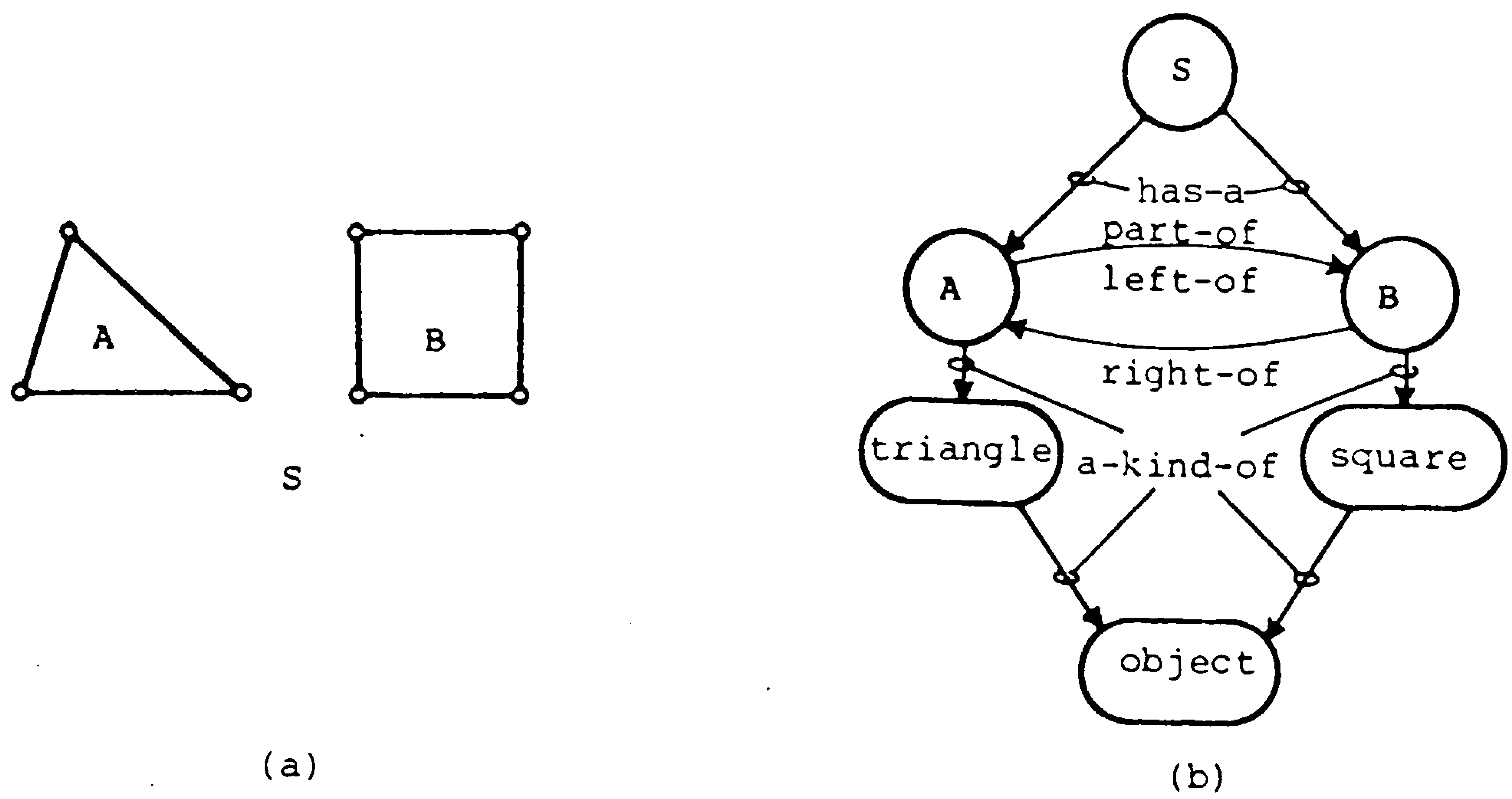


FIGURE 5.1

The learning algorithm is a two-step process. First, a difference description is developed from the comparison between the current concept description and each example. The comparison is accomplished by graph-matching and the result is recorded with *comment notes* (C-NOTES), that describe the degree of success of the matching. The algorithm looks for the 'best' match between the current concept description and the training example. Then a generalizing procedure obtains a skeleton of the links and nodes that match exactly and attaches the C-NOTES to the skeleton. The next action of the program is determined by the types of the C-NOTES according to a certain table. The algorithm is quite fast but makes inefficient use of memory for storing intermediate descriptions. It works with conjunctive descriptions but it can be extended [Iba '79] to include disjunctive ones too. It depends a lot on the teacher and the amount of noise in the examples. Finally it performs some types of constructive induction.

A program by [Hayes-Roth & McDermott '77], finds MSC (maximally-specific conjunctive)-generalizations (called *maximal-abstractions*) using a set of positive examples as its input. Both input events and their generalizations are represented by *parametrized structural representations* (PSR's) which are conjunctions of predicates of the form: *case-label (parameter-list)*. The induction algorithm begins by generalizing the first input example and continues in a bottom-up fashion. In the ith step, a new set of generalizations G_{i+1} is obtained by performing a partial match between each element of G_i and the current example. The partial matching (called *inference match*) looks for the longest one-to-one match of parameters and case frames, and it is performed in two steps. First, a set M is obtained by matching the case frames in all possible ways. Second, a subset of a consistently bound subset of the parameter correspondence in M is selected, by pruning unpromising nodes in a node-building process. The algorithm uses a quite powerful representational language with no disjunction operator. However, it has no constructive induction facilities, limited extensibility to include disjunctions and low computational efficiency. A similar method [Vere '77] represents examples as a conjunction of *literals* (lists of *terms* treated uniformly) and performs the generalization in four steps with a selective matching of literals. The method has been extended to discover disjunctions and exceptions.

Meta-DENDRAL is a model-driven system (developed also as expert system DENDRAL) [Buchanan & Feigenbaum '78], designed to learn cleavage rules used by a mass-spectrometer simulator. The cleavage rules are written as condition-action rules, where conditions describe portions of the molecular structure and actions the bonds that will break. The procedure begins by producing highly-specific cleavage rules for every broken fragment based on very simple background knowledge (subroutine INTSUM). It proceeds by generalizing these rules in two phases. First, a generate-and-test subroutine (RULEGEN) performs a coarse search which results in approximate and - sometimes - redundant rules. Then, a second subroutine (RULEMOD) modifies the existing rules and makes them more precise and less redundant. Meta-DENDRAL is an important complex special-purpose learning system with limited extensibility in non-chemical domains, it also has low computational efficiency. In the context of Meta-DENDRAL [Mitchell '78], introduced a new method of learning rules from examples based on the concept of *version spaces*. The method uses an algorithm called *candidate elimination*, which maintains and modifies a representation of the space of all plausible rule versions. As version space is defined as the set of current hypotheses of the correct statement of a rule which predicts some fixed action. Version spaces are represented by the sets of *maximally general versions* and *maximally specific versions* obtained by a general-to-specific ordering. The algorithm begins with the space of all rule versions consistent with the first positive training instance and progressively modifies the version space to eliminate candidate versions inconsistent with subsequent training instances. Some advantages of the method are: it proceeds without needing to backtrack (and modify rules), it finds *all* correct versions of the rule induced by a complete training database, allows the program to generate its own set of critical training instances and is a consistent method for merging sets of rules generated from distinct training data sets. However, it requires reliable (noise-free) training data and small extremal sets.

[Michalski '80] and [Dietterich & Michalski '81] describe a general method for determining disjunctive structural descriptions that can be used to discover MSC-generalizations. Events are represented by conjunctions of *selectors* (i.e. relational statements of the form: predicate descriptor [variable list]). The method used hierarchical planning in order to speed up the search for generalizations. The idea is to first search the description space defined by the *structure-specifying descriptors* (i.e. non-unary ones) in order to find plausible generalizations. Then, the *attribute-specifying descriptors* (i.e. unary ones) are searched in order to fill out the detailed generalizations. The method has computational advantages (through the hierarchical approach), representational economy, good noise immunity and allows for domain-specific knowledge to be incorporated by the program. Among its disadvantages are: it is fast *only* in problems using both unary and non-unary descriptors; it is difficult to define plausible descriptions in structure-only space and to conduct the attribute search; it has a low computational efficiency. An extension of Winston's work in learning structural descriptions, examines resemblances among a set of examples looking for 'promising' partitions [Loisel & Kodratoff '81]. Near-misses are classified according to their ambiguity into *highly ambiguous*, *ambiguous* and *discriminant*.

These classes convey different types of information expressed by a set of defined indices, that are also a measure of the best (most promising) portion of a set of examples. The structural description is conducted by dividing discriminable sets or by minimising the ambiguity between the two subsets in case of several discriminating partitions or by maximising the ambiguity of each subset in the case of indecision. This means that examples which are discriminable are kept together for as long as possible. A system by [Wysotzki et al '81], for learning concepts of structured objects from examples uses a mathematical formalism to represent the training samples. Labelled graphs (as in Winston's case) are transformed into linear representations, the *feature vectors*, that describe a structured object unambiguously. A feature vector is a triple of the form: $v^{(i)}=(x_1, \ldots, x_r; y_1, \ldots, y_s; z_1, \ldots, z_t)$, where x_i and y_i are the numbers of node and arc-relations in the graph respectively, and z_i indicates the presence of the triple i in the graph (value=1, =0 otherwise). The concept learning algorithm develops decision trees from structural hypotheses and proceeds with a general-to-specific depth-first search with re-examination of past events. This approach gives the possibility of learning concepts defined by other concepts related to them (e.g. context).

[Silver '83], describes a program (LP) that learns new techniques for solving (complex) equations by examining worked examples. The basic operators of LP are called *methods*. Each method has an associated set of rewrite rules, and some control information indicating when the method should be applied. Equations are solved by applying methods, which in turn may apply rewrite rules. The control information includes *preconditions* (facts that should be true before the method is applied) and *postconditions* (to ensure that the method had the desired effects). In order to learn a new technique, LP may need to learn at several levels. That is, new algebraic identities that will be used as rewrite rules (lowest level), new methods, involving their control information and their association with some rewrite rules (next level up) and meta-control information controlling the order in which methods are used. The latter is recorded in a plan, called *schema*, that records how the equation is solved and can be used to solve new equations. The process of learning begins with the *justification step*, in which LP examines consecutive pairs of lines in a worked example trying to find the method that transforms each line to the next one. Then, LP examines its analysis to see if any new methods need to be created, in order to explain the applications of new rules. The new method consists of the old one augmented by the new rule. The preconditions of the method become the union of the subset of preconditions of the method that are satisfied by the new rule and those of the new rule, while the post-conditions of the new method remain those of the old one. By examining all the lines of the worked example, a list of all the methods used for this example, called *schema*, is created. Schema is used to solve new equations, acting as a plan that can be executed in a flexible way. The techniques used by LP seem to be applicable to many domains (e.g. symbolic integration, algebra). Another method [Kibler & Porter '83], for learning to solve simultaneous linear equations from examples, uses *perturbation* operators to create near examples and/or near misses. Perturbation breaks up the generalization process in two steps. Each example is perturbed multiple times to create near examples and near misses. A set of minimal generalizations is formed in order to sift out non-essential conditions. The generalizations are further refined with additional teacher assistance. The low cost of generating and applying perturbation operators makes the system independent of the order in which they are tried. It also relies less on the teacher for appropriate examples. A model-driven method for machine learning of inference rules by induction and 'being told' is presented by [Emde et al '83]. It uses *higher concepts* (transitivity, convexity) attributes to induce relations among two-place predicates, represented as *meta-facts* and expressed by *meta rules*. Meta facts are true in certain domains called *support sets* which are the basis for discovering new concepts. Reconstruction of support sets resolves contradictions and makes inference rules more precise.

ALEX is a program that learns to solve simple equations from examples [Neves'85]. It first learns four legal operators by looking at examples showing what they do. ALEX goes through the lines of each example comparing two at a time. Using a *means-ends* routine it tries to apply operators to the first line until the second is reached. If the routine fails a new operator needs to be generated and the difference conditions of the two lines are registered. ALEX uses two methods to work problems in a textbook. It first uses its working forward rules to solve the problem. If it fails, a problem solving routine (means-ends) takes over and suggests a new operator, leading to the creation of a new rule. New rules are indexed by their differences, making the learning system capable of recognizing more complex examples as well as developing high-level operators to work on difficult problems. One of the results of this method is to show how learning, problem solving and performance can be combined into a single system. A weakness of the system is the creation of the condition side of a rule using heuristics rather than *rule-tuning*.

Generation of decision trees from a set of examples provided by a domain expert is a practical method for knowledge acquisition. [Arbab & Michie '85] have developed a generator of linear (every node has at most *one* non-terminal son) and yet efficient decision trees. The algorithm (RG) assumes that structured induction is feasible and absolute priority of linearity over efficiency. It proceeds by assigning a decider status (decider, non-decider) to the attributes of an example-set and builds up the decision tree by selecting attributes from a set of deciders. This will not affect the linearity of the final tree. A measure of linearity is used to obtain the optimal linear tree. A scheme for learning complex descriptions, from examples with errors is presented by [Segen '85]. Learning is based on a selection criterion (*minimal representation criterion*), which minimises a combined measure of discrepancy of a description with training data, and complexity of a description. Learning rules for two types of descriptors are derived: one for finding descriptors with good average discrimination over a set of concepts, second for selecting the best descriptor for a specific concept. Once these descriptors are found, an unknown instance can be identified by a search using the descriptors of the first type for fast screening of candidate concepts, and the second for the final selection of the closest concept. The method is compatible with other methods for learning structural descriptions.

5.3 FIGURE DEFINITIONS

The following definitions cover the figures that are to be learnt by the learner, and are presented in order of increasing complexity. The figures that are considered in this work are 3-D polyhedra with triangular of quadrilateral faces, represented by line drawings. The definitions are based on the connectivity of the points that constitute the line drawings, and their structure depends on whether these line drawings refer to 2-D or to 3-D figures. In the case of 3-D figures, some extra conditions must be satisfied. Each definition is represented by a PROLOG predicate, the left-hand side of which is the name of the defined figure, while the right-hand side consist of the conditions required to make up the figure. For convenience reasons the figures are divided into three classes according to their dimensionality and each is examined in a separate section.

5.3.1 1-D Figures

The two basic elements that constitute a line drawing are points and straight-line segments. The special way in which the points are connected with each other is characteristic for each line drawing and it is defined by a *figure-name* (or a set of figure-names). Two points A and B, directly connected with each other constitute a *connectivity primitive* that is represented by the PROLOG-predicate: *conn(A,B,N)*. Where N is an integer $0 \leq N \leq 2$ called a *face-counter*, and it is used to indicate the number of polygonal faces that share it. For example, if a *conn*-predicate is used to represent a 3-D polyhedron edge, N will have the original value 2, because it belongs to two separate faces, and thus it must be considered twice. Face-counters play an important role in the determination of occluded faces and hidden lines. Their importance will become clear when their exact use, by the learner (§5.4) and the recognizer (§6.4), is discussed in detail.

1-D figures are the straight-line segments that connect two points, otherwise known as *sides* of a polygon. These are represented by *line-predicates* which are defined by means of *conn*'s as follows:

line(A,B):-(conn(A,B,N);conn(B,A,N)),

atom(A),atom(B),integer(N),A \ =B,N>0,N=<2. (5.1)

This means that *line(A,B)* is the element that connects point A with point B (or vice versa), where A and B are not integers and are not identical, while integer N takes values 1 or 2. In parallel with this main definition of a line, there are three rather auxiliary definitions of lines:

a_line(A,B):- conn(A,B,1); conn(B,A,1). , (5.2)

b_line(A,B):- conn(A,B,2); conn(B,A,2). and (5.3)

c_line(A,B):- conn(A,B,0); conn(B,A,0). (5.4)

This means that *a_line* is a side that belongs to *one* polygon only, *b_line* is shared by two polygons and *c_line* is an existing side that does not belong to any current polygon (i.e. the polygon(s) that contain(s) it has (have) already been considered). The latter three versions of *line* are used by the multi-view recognizer in the assuming phase (§6.4.3). *line*'s are the basic predicates that make up the 2-D figures.

5.3.2 2-D Figures

In this category belong two classes of closed polygons, namely *triangles* and *quadrilaterals*. An important property of both classes of polygons is that they are faces of polyhedrons and as such they are *planar*.

Triangle is the closed polygon defined by three points connected with each other by three line segments. The points are the *vertices* and the line segments the *sides* of the triangle. From the above definition it is obvious that triangles are always planar (since three points always define a plane), and they are faces with minimum number of sides (or vertices). However, a closer examination of this definition shows that it classifies as triangles, and thus faces, combinations of vertices and sides that do not correspond to real faces, as Figure 5.2b very clearly demonstrates; in the 3D figure (abcde), (abc) is not a face. In fact the structure of line drawings is

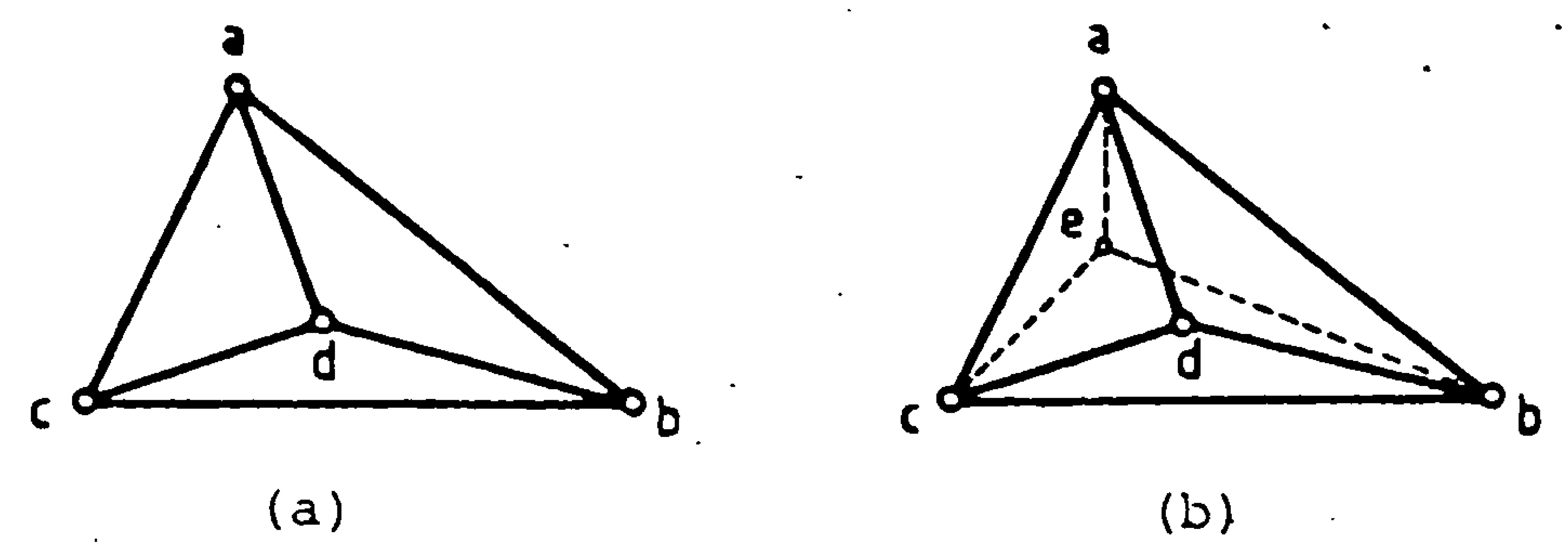


FIGURE 5.2

such that may give rise to false interpretations. In this case several other cues, together with a general understanding of concepts like three-dimensionality, occlusion and visibility are required to lead to a correct interpretation. The following conclusions are drawn from the basic assumption that: in a line drawing, every line connecting two points is in fact an edge that separates two faces of a polyhedron.

Thus, the existence of a line eliminates the possibility of the faces which it separates being coplanar. For example, faces (triangles) (abd), (dbc) and (adc) of the polyhedron in Figure 5.2a form a 3-D angle in vertex d. Containment of a point by a polygon indicates occlusion of a *possible* face. Triangle (abc) of Figure 5.2 represents an occluded face only in 5.2a.

These conclusions lead to the following definitions of a triangle (visible face) and a possible triangle (occluded face):

trian(A,B,C):- line(A,B),line(B,C),line(C,A), (5.5)
*not(poin_trn(A,B,C)).**

p_trian(A,B,C):- line(A,B),line(B,C),line(C,A), (5.6)
*poin_trn(A,B,C).**

The cyclic order in which the three vertices appear in the definition, denotes that the triangle is a closed polygon. The line-predicates represent the sides of the triangle, and predicate *poin_trn(A,B,C)* indicates whether triangle *(ABC)* contains a point or not.

Quadrilateral is the *planar* closed polygon defined by four points (vertices) and four line segments (sides). Before proceeding into the analytical definition of quadrilaterals it is important to examine the three quadrilaterals shown in Figure 5.3. 5.3a shows a line drawing

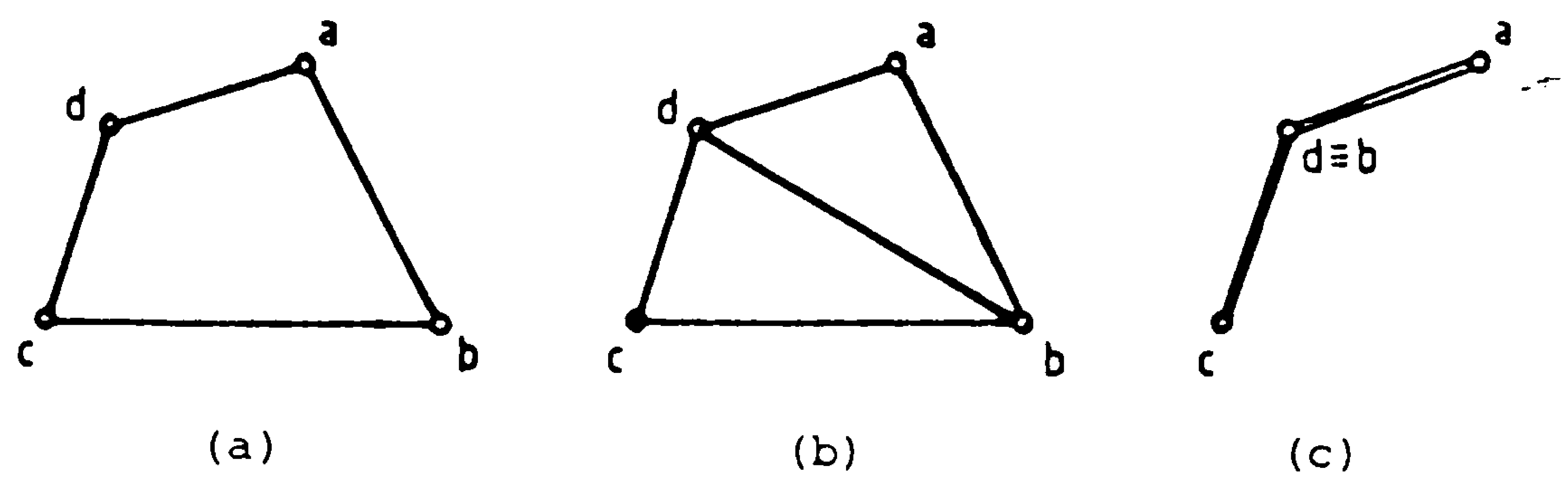
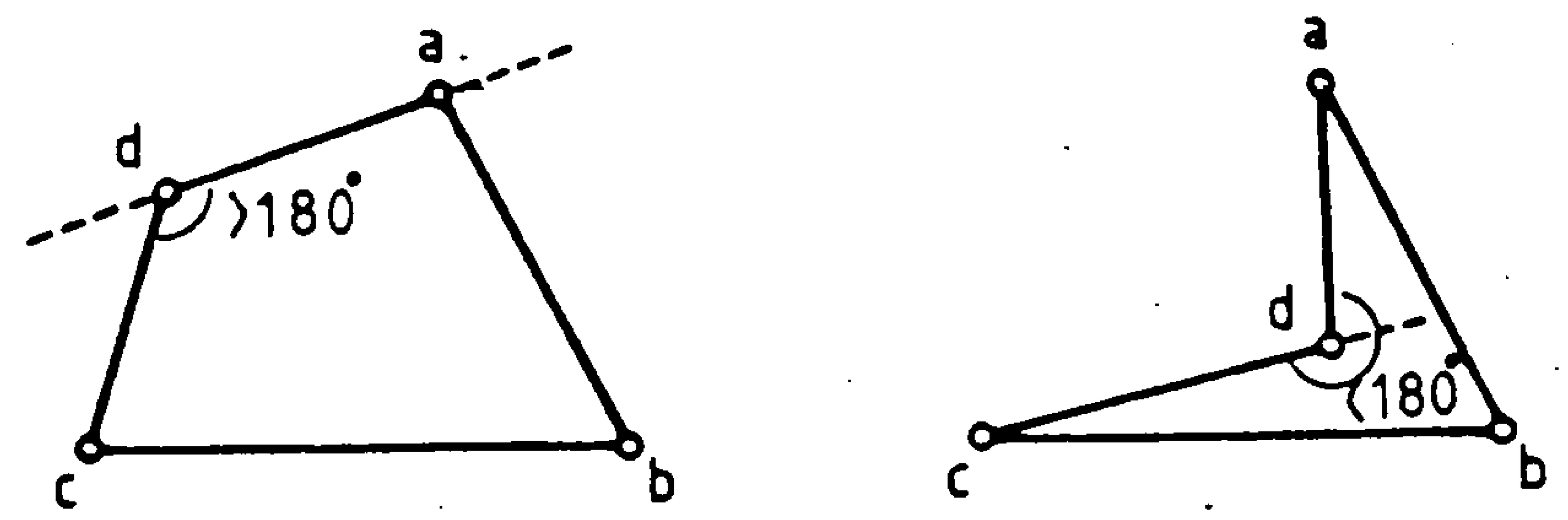


FIGURE 5.3

*For *poin*_predicates, see p.270.

representing a quadrilateral; 5.3b represents two triangular faces connected with a common side (db), and 5.3c shows a degenerate quadrilateral. The distinguishing factor between line drawings in Figures 5.3a and b is the existence of the diagonal (db) in 5.3b, while Figures 5.3a and 5.3c differ in that vertices b and d in 5.3c coincide. Also, quadrilaterals can be *convex* or *non-convex* (have an internal angle >180°) as Figure 5.4 shows. After all these remarks,



convex (a) non-convex (b)

FIGURE 5.4

the definitions covering the cases (visible, possible, convex, non-convex), a quadrilateral will be:

```
c_quadril(A,B,C,D):-line(A,B),line(B,C),line(C,D),line(D,A)
                    A\=C, B\=D,not(line(A,C)),not(line(B,D)),
                    not(poin_qul(A,B,C,D)),
                    not(nocnvx(A,B,C,D)).                    (5.7)

nc_quadril(A,B,C,D):-line(A,B),line(B,C),line(C,D),line(D,A),
                     A\=C,B\=D,not(line(A,C)),not(line(B,D)),
                     not(poin_qul(A,B,C,D)),
                     nocnvx(A,B,C,D)).                       (5.8)
```

```
p_c_quadril(A,B,C,D):-line(A,B),line(B,C),line(C,D),line(D,A),
                      A\=C,B\=D,not(line(A,C)),not(line(B,D)),
                      poin_qul(A,B,C,D),
                      not(nocnvx(A,B,C,D)).                      (5.9)

p_nc_quadril(A,B,C,D):-line(A,B),line(B,C),line(C,D),line(D,A),
                       A\=C,B\=D,not(line(A,C)),not(line(B,D)),
                       poin_qul(A,B,C,D),
                       nocnvx(A,B,C,D).                          (5.10)
```

The first line of the definition is self-explanatory, as in the case of *trian*. In the second line, *A\=C* and *B\=D* prevent a degenerate case when two diagonal vertices coincide, and the rest of the line makes sure that no diagonal line exists. Finally, the last two predicates examine the containment of a point by the quadrilateral and its convexity respectively.

The definitions of the two routines that check the point containment in a triangle and a quadrilateral are given below:

```
poin_trn(A,B,C):- point_in_trn(A,B,C);point_in_trn(A,C,B);
                  point_in_trn(B,A,C);point_in_trn(B,C,A);
                  point_in_trn(C,A,B);point_in_trn(C,B,A).
                                                                 (5.11)
poin_qul(A,B,C,D):- point_in_qul(A,B,C,D);point_in_qul(A,D,C,B);
                    point_in_qul(B,A,D,C);point_in_qul(B,C,D,A);
                    point_in_qul(C,B,A,D);point_in_qul(C,D,A,B);
                    point_in_qul(D,A,B,C);point_in_qul(D,C,B,A).
                                                                 (5.12)
```

The body of the definitions is a disjunction of the feasible commutations of the basic predicates *point_in_trn* and *point_in_qul* respectively.

The following group of *set*-predicates is used to define predicates created by the system in the phase of single-view learning. Like the above two definitions, they are disjunctions of feasible commutations of the system-predicates: *atrian(A,B,C)*, *ptrian(A,B,C)*, *aquadril(A,B,C,D)*, *nquadril(A,B,C,D)*, *pquadril(A,B,C,D)* and *p_nquadril(A,B,C,D)*, denoting a visible triangle, a possible triangle, a visible convex quadrilateral, a non-convex quadrilateral, a possible convex quadrilateral, and a non-convex quadrilateral respectively.

set_atr(A,B,C):- atrian(A,B,C);atrian(A,C,B);atrian(B,A,C);
atrian(B,C,A);atrian(C,A,B);atrian(C,B,A). (5.13)

set_ptr(A,B,C):- ptrian(A,B,C);ptrian(A,C,B);...ptrian(C,B,A). (5.14)

set_aqu(A,B,C,D):- aquadril(A,B,C,D);aquadril(A,D,C,B);
aquadril(B,A,D,C);aquadril(B,C,D,A);
aquadril(C,B,A,D);aquadril(C,D,A,B);
aquadril(D,A,B,C);aquadril(D,C,B,A). (5.15)

set_nqu(A,B,C,D):- nquadril(A,B,C,D);...nquadril(D,C,B,A). (5.16)

set_pqu(A,B,C,D):- pquadril(A,B,C,D);...pquadril(D,C,B,A). (5.17)

set_p_nqu(A,B,C,D):- p_nquadril(A,B,C,D);...p_nquadril(D,C,B,A). (5.18)

5.3.3 3-D Figures

This category contains the following convex polyhedrons:

tetrahedron, *pyramid*, *prism* and *truncated-pyramid*. A common property of these 3-D figures is that their faces are either triangles or convex quadrilaterals.

A *tetrahedron* consists of four faces all of which are triangles (Fig. 5.5a). The definition of a tetrahedron obtained by multiple views is:

$$tetra(A,B,C,D){:}{-}trian(A,B,C),trian(A,C,D),trian(A,B,D),trian(B,C,D). \tag{5.19}$$

A *pyramid* consists of five faces, a quadrilateral (base) and four triangles with a common vertex (Fig. 5.5b). Its multiple-view definition is:

$$\begin{aligned} pyram(A,B,C,D,E){:}{-}\ & c\_quadril(A,D,E,B),trian(A,B,C),trian(A,D,C),\\ & trian(C,D,E),trian(B,E,C). \end{aligned} \tag{5.20}$$

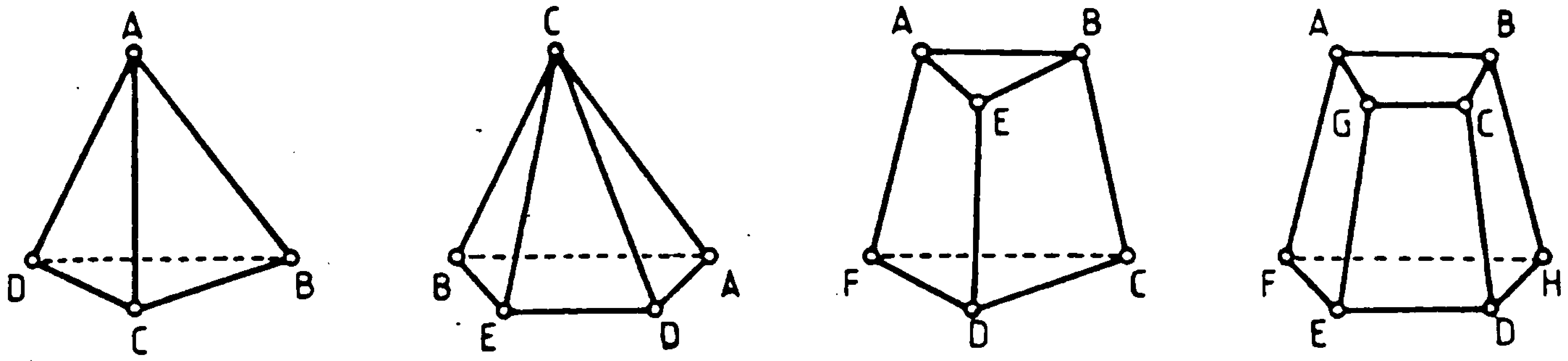
A *prism* consists of five faces, three of which are quadrilaterals connected in pairs with a common (hard) edge, while the other two are triangles with no common elements (Fig. 5.5c). This is a general definition containing the case (usually meant by 'prism') where the common edges of the quadrilaterals are parallel to each other. The multiple-view definition of a prism is:

$$\begin{aligned} prism(A,B,C,D,E,F){:}{-}\ & c\_quadril(E,A,F,D),c\_quadril(A,B,C,F),\\ & c\_quadril(E,B,C,D),trian(A,B,E),trian(F,C,D). \end{aligned} \tag{5.21}$$

A *truncated-pyramid* consists of six faces, all of which are quadrilaterals (in fact it is a pyramid with a pyramid of smaller base taken off its top, Fig. 5.5d). In the definition below the name *box* is used as head of the predicate for reasons of convenience:

```
box(A,B,C,D,E,F,G,H):- c_quadril(A,B,C,G),c_quadril(G,C,D,E),
                       c_quadril(A,G,E,F),c_quadril(B,A,F,H),
                       c_quadril(C,B,H,D),c_quadril(H,F,E,D).
```

(5.22)



tetrahedron (a) pyramid (b) prism (c) truncated-pyramid (d)

FIGURE 5.5

The multi-view definitions are unique for each 3-D figure i.e. there is a one-to-one correspondence of the set of definitions onto the set of 3-D figures. These definitions are used by the multiple-view recognizer and they may succeed only if another view is given or an assumption is made.

The single-view definitions refer to a set of views that may be interpreted as one of the above defined polyhedrons. In principle, they can be seen as subsets of the four multi-view definitions, with respect to the number of faces they consist of (only visible faces). However, their difference from the above is that they make use of the system predicates *atrian*, *aquadril*, *ptrian*, etc. rather than the user *conn*-predicates, a fact that makes them faster. The head of a definition referring to a certain 3-D figure uses the same name as in the multiple-view definition preceded with prefix *p_*, which stands for 'possible', and followed by a code number. This code number shows the number of missing faces from respective multi-view definitions and is used to represent different definitions (views) of the same 3-D figure. In the case of two definitions having the same code-number (visible faces) the second one will take an extra suffix *a* (e.g. *prism*, *p_prism1*, *p_prism1a*). The single-view definitions and the views they represent are given below:

```
p_box1(A,B,C,D,E,F,G,H):- set_aqu(A,B,C,D),set_aqu(A,B,F,E),
                          set_aqu(B,C,G,F),set_aqu(D,C,G,H),
                          set_aqu(A,D,H,E),set_pqu(E,F,G,H),
                          E\=G,E\=C,H\=F,H\=B,F\=D,G\=A.
```
(5.23)

```
p_box2(A,B,C,D,E,F,G,H):- set_aqu(A,B,C,D),set_aqu(B,C,G,F),
                          set_aqu(B,C,G,F),set_aqu(A,D,H,E),
                          not(non-convex_contour_angle(A)),
                          not(non_convex_contour_angle(B)),
                          not(non_convex_contour_angle(G)),
                          not(non-convex_contour_angle(H)),
                          H\=B,H\=F, G\=A,G\=E, F\=E,B\=E,F\=A.
```
(5.24)

```
p_box3(A,B,C,D,E,F,G):- set_aqu(A,B,C,D),set_aqu(B,C,G,F),set_aqu(D,C,G,E)
                        not(non_convex_contour_angle(B)),
                        not(non_convex_contour_angle(G)),
                        not(non_convex_contour_angle(D)),
                        A\=E,A\=G,A\=F,B\=E,D\=F.
```
(5.25)

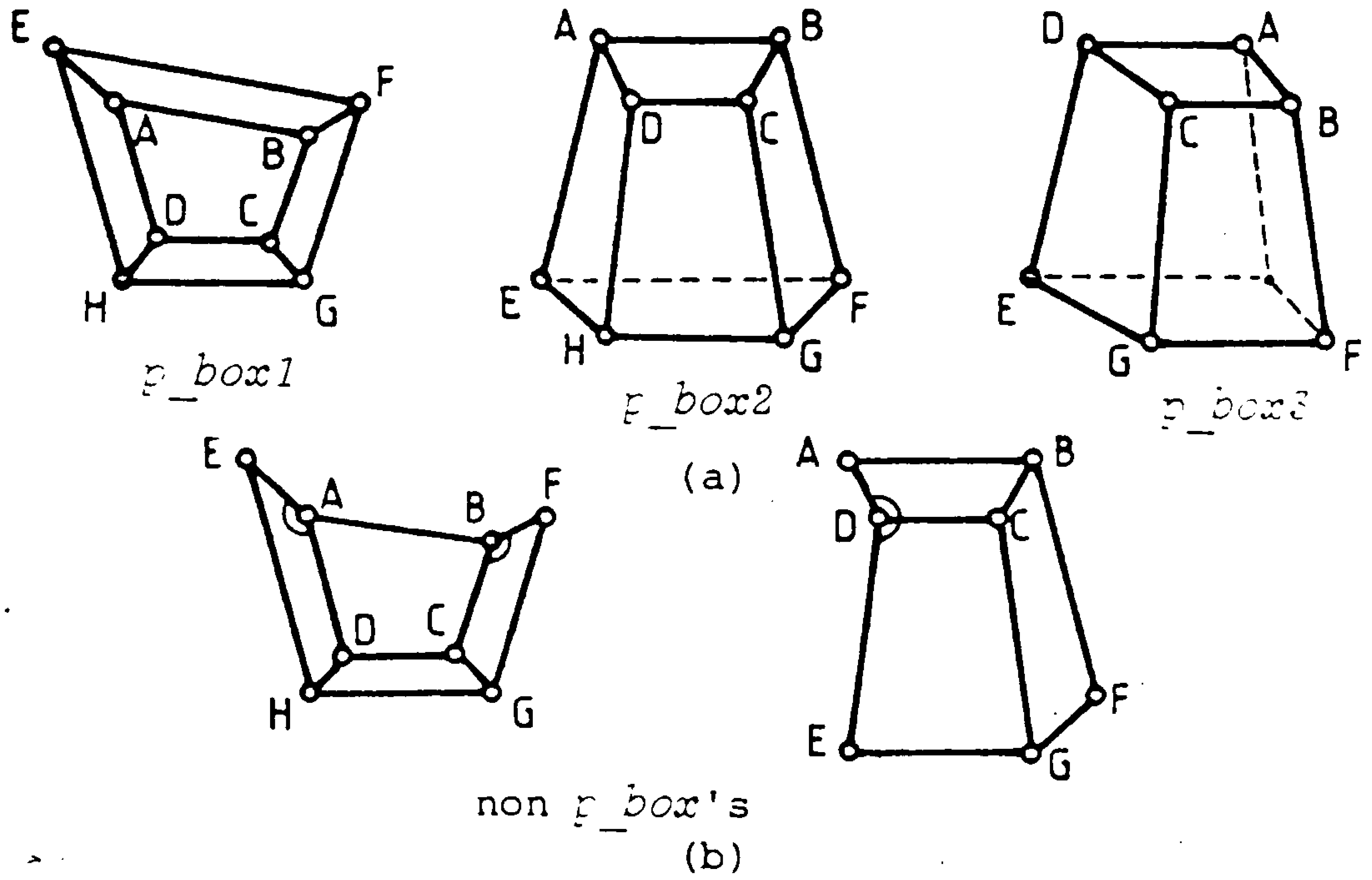


FIGURE 5.6

The *not(non_convex_contour_angle(X))*-predicates prevent cases like the ones in Figure 5.6 from being identified as *p_box2* or *p_box3* respectively.

p_prism1(A,B,C,D,E,F):- set_aqu(A,B,E,D),set_aqu(B,E,F,C),
set_aqu(C,F,D,A),set_atr(A,B,D),set_ptr(D,E,F).
(5.26)

p_prism1a(A,B,C,D,E,F):- set_aqu(B,E,F,C),set_aqu(C,F,D,A),
set_atr(A,B,C),set_atr(D,E,F),set_pqu(A,B,E,D).
B\=F,C\=E.
(5.27)

p_prism2(A,B,C,D,E,F):- set_aqu(C,F,D,A),set_aqu(B,E,F,C),set_atr(A,B,C),
not(non_convex_contour_angle(A)),
not(non_convex_contour_angle(B)),
not(non_convex_contour_angle(F)),
A\=E,B\=D.
(5.28)

```
p_prism2a(A,B,C,D,E,F):- set_aqu(A,B,E,D),set_atr(A,B,C),set-atr(E,D,F),
                         not(non_convex_contour_angle(A)),
                         not(non_convex_contour_angle(B)),
                         not(non-convex_contour_angle(E)),
                         not(non_convex_contour_angle(D)),
                         B\=F, A\=F,C\=F,C\=E,D\=E.
```

(5.29)

```
p_prism3(A,B,C,D,E,F):- set_aqu(A,B,C,D),set_aqu(A,D,E,F),
                        not(non_convex_contour_angle(A)),
                        not(non_convex_contour_angle(D)),
                        B\=F,B\=C,C\=E,E\=F.
```

(5.30)

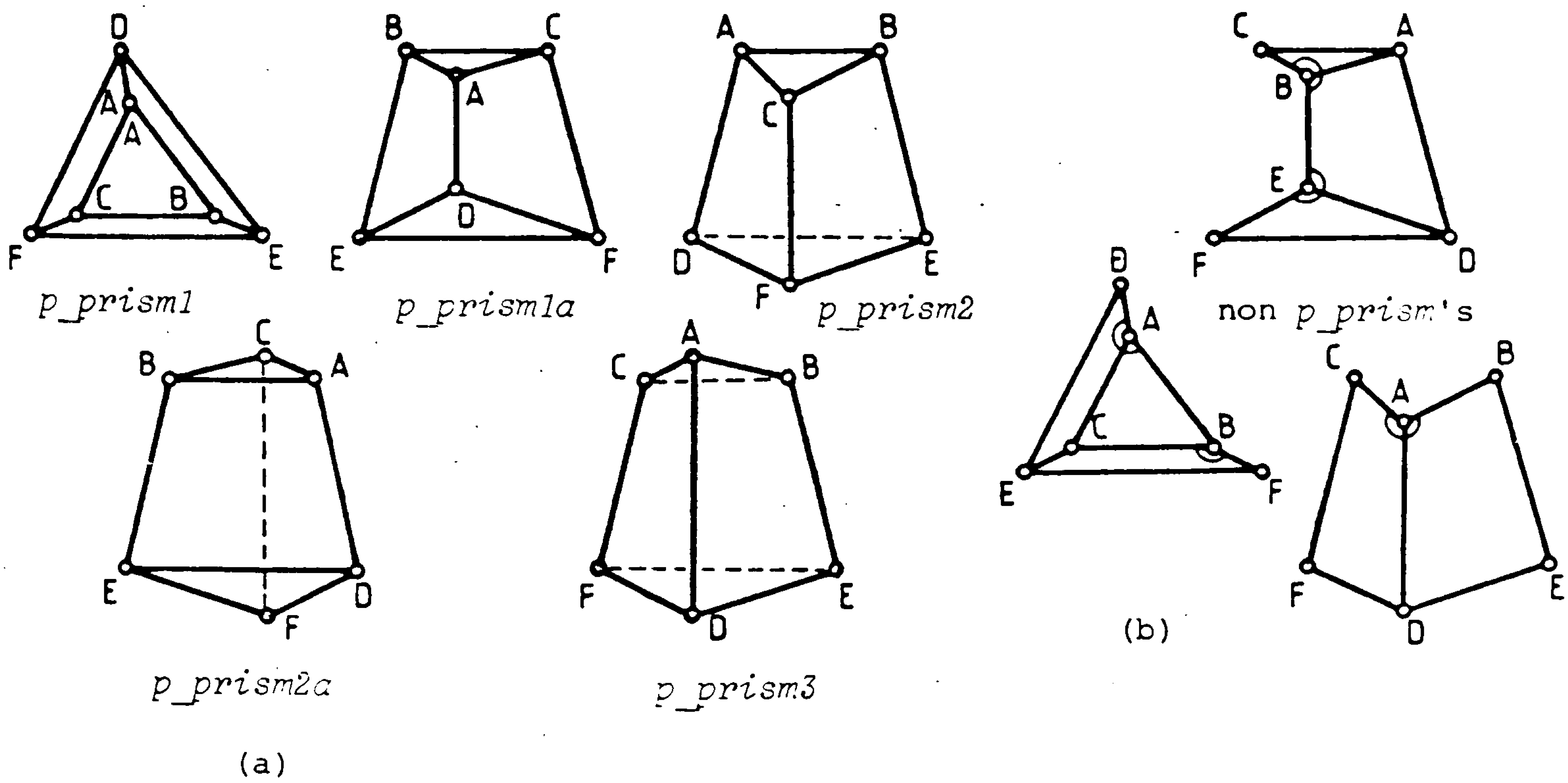


FIGURE 5.7

Figure 5.7a demonstrates the 'positive' *p_prism*'s and 5.7b some 'negative' examples.

```
p_pyram1(A,B,C,D,E):- set_aqu(B,C,D,E),set_atr(A,E,B),set_atr(A,B,C),
                      set_atr(A,D,C),set_ptr(A,D,E).
                                                                (5.31)

p_pyram1a(A,B,C,D,E):- set_atr(A,E,B),set_atr(A,B,C),set_atr(A,D,C),
                       set_atr(A,D,E),set_ptr(B,C,D,E).
                                                                (5.32)

p_pyram2(A,B,C,D,E):- set_aqu(B,C,D,E),set_atr(A,B,C),set_atr(A,D,C),
                      not(non_convex_contour_angle(B)),
                      not(non_convex_contour_angle(D)),
                      A\=E.                                     (5.33)

p_pyram2a(A,B,C,D,E):- set_atr(A,B,C),set_atr(A,D,C),set_atr(A,B,E),
                       not(non_convex_contour_angle(A)),
                       not(non_convex_contour_angle(B)),
                       not(non_convex_contour_angle(C)),
                       B\=D, C\=E,D\=E.                         (5.34)

p_pyram3(A,B,C,D,E):- set_aqu(B,C,D,E),set_atr(A,B,C),
                      not(non_convex_contour_angle(B)),
                      not(non_convex_contour_angle(C)),
                      A\=E,A\=D.                                (5.35)

p_pyram4(A,B,C,D):- set_aqu(A,B,C,D).                           (5.36)
```

The legal cases of *p_pyram* are shown in Figure 5.8a; 5.8b shows two illegal cases.

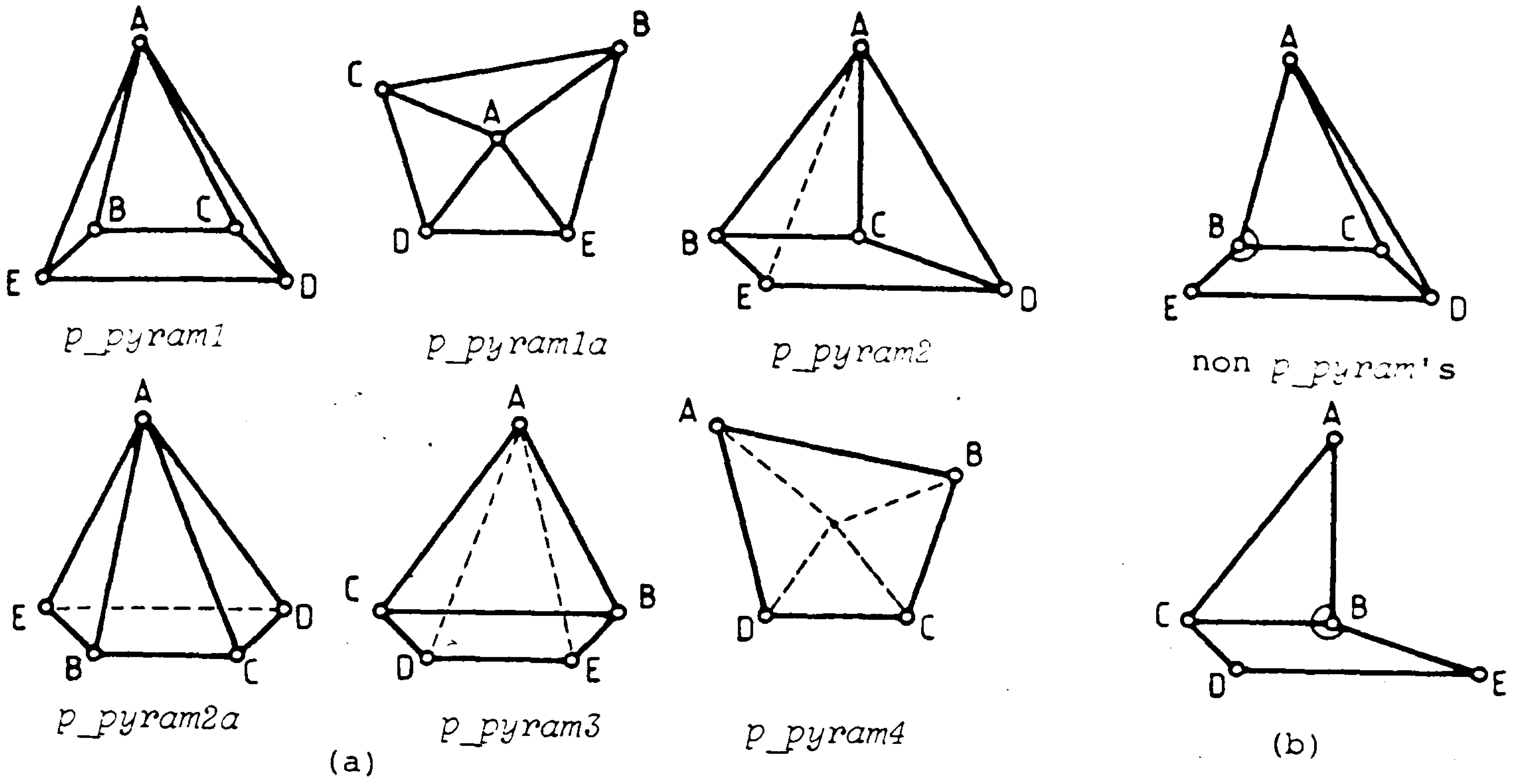


FIGURE 5.8

p_tetra1(A,B,C,D):- set_atr(A,B,C),set_atr(A,C,D),set_atr(C,B,D),
set_ptr(A,B,D). (5.37)

p_tetra2(A,B,C,D):- set_atr(A,B,C),set_atr(A,C,D),
not(non_convex_contour_angle(A)),
not(non_convex_contour_angle(C)),
B\=D. (5.38)

p_tetra3(A,B,C):- set_atr(A,B,C). (5.39)

Figure 5.9a depicts the three legal views of a *p_tetra* and 5.9b an illegal one.

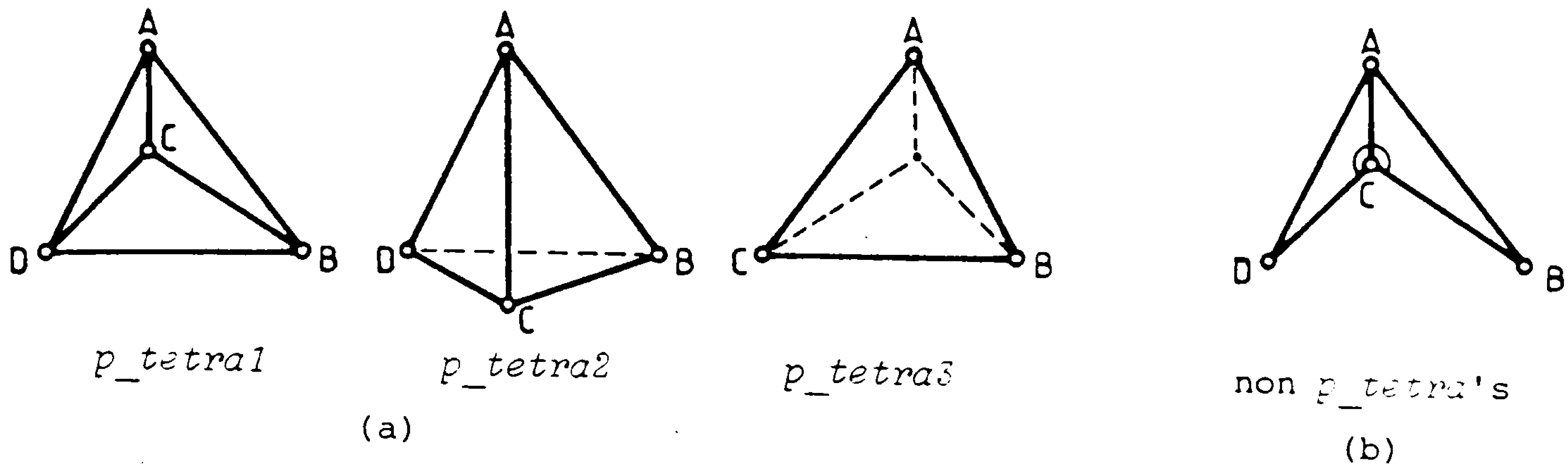


FIGURE 5.9

5.4 THE LEARNER

The *learner* is the program that learns the definitions of the last paragraph and consists of three main routines: the *elementary-concept learner*, the *multiple-view learner* and the *single-view learner*. The first routine learns the definitions of 1-D and 2-D figures which make up the recognizable 3-D figures. The second routine learns the definitions of a recognizable 3-D figure from successive multiple views of it. Finally, the third routine learns to relate single views of recognizable 3-D figures to the objects from which they originate. The method employed by the learner is *learning from examples* that are presented to the system by the user. The examples are, in general, line drawings which represent legitimate figures, or illegitimate figures that differ from the correct ones by a single element - *near misses*. Before examining each routine and its function in detail, a brief outline of the main principles and the basic techniques that are used is attempted. Furthermore, the uniform formalism and terminology of [Bundy & Silver '82] is adopted as well as the PROLOG conventions (Appendix 3).

The learning task can be considered as a procedure that modifies a set of rules of the form:

$$C :- H_1, H_2, \ldots, H_n,$$

where C is the *conclusion* and H_i, $1 \leq i \leq n$ is a condition of the *hypothesis*. The procedure of rule learning consists of the following three stages:

a) A rule is obtained from a correct example by doing a generalization with the method of *turning constants into variables*.

b) The rule is tested by a series of examples until a fault is found. The part of the program responsible for identifying faults is called the *critic*.

c) The rule is modified by another part of the program called the *modifier*, in order to correct the fault.

The last two stages are repeated until the rule is satisfactory. The kind of faults that are involved, are *factual faults* i.e. false answers due to faulty rules, rather than *control faults*.

The critic performs the fault identifying by testing the rules on a certain problem and then analysing the program trace. A correct application of the rules is called a *positive training instance* and is used to generalize them. An incorrect application of the rules is called a *negative training instance* and is used to correct them. There are two types of negative training instances. *Commission errors*, when a rule is used incorrectly because it is insufficiently constrained, and *omission errors*, when a rule fails because it is either incorrectly constrained or does not exist.

The information required for the rule modification is the type of instance, the rule and the *context*. The latter consists of the variable bindings of the rule when applied, and it is called the *selection context* when it refers to positive and *rejection context* when it refers to negative training instances. The rule is modified by adding extra conditions to its hypothesis. The technique used by the modifier is called the *discriminating algorithm* [Langley '81] and its basic idea is to apply the selection and rejection context substitutions to a fixed set of literals, called the *description space*, and then find the difference between the two contexts, called the *discriminating literal*. The latter is added as an extra condition to the rule hypothesis.

5.4.1 The Elementary-Concept Learner

The elementary-concept learner is the first of the three routines comprising the learner, with the task of creating a primary knowledge base that is to be used by the other two routines. The knowledge base consists of elementary 1-D concepts like, *line* (as well as versions: *a_line*, *b_line* and *c_line*), and elementary 2-D concepts like, *trian*, *quadril*, the *point_in* predicates, *non_convex_qul* and the *set_of*-predicates. The routine consists of two main procedures. Procedure [first_rule], that is responsible for forming the initial rule, and procedure [improve], that successively modifies the initial rule, according to a set of examples given by the user, until it takes a final desirable form. The system possesses an internal mechanism that determines the truth value of the offered training instance and tests the validity of the current rule on this example. This mechanism is based on a set of rules called the *r_rule*'s, that are supplied by the user (either manually or by consulting a file) at the beginning of the routine. An r_rule is constituted by an *r_head*, that has the name of the predicate to be learnt prefixed by *r_*, and a *body*, that consists of the conditions which define the predicate according to the user. The set of *r_rule*'s is practically equivalent* to the set of rules that are to be learnt as far as the body of the rules is concerned, while the head of the rules is prefixed by *r_*. Prefix *r_* is used in order to prevent the co-existence of two equivalent sets of rules with identical heads in the system.

Procedure [first_rule] begins with a positive training instance as input (manually or from a file specified by the user), in the following format:

head,[body]

where *r_head* is the name of the concept to be learnt prefixed by *r_*, and *body* is the list of conditions that comprise the body of the rule. Prefix *r_* is used in order to test the truth value of the training instance against the set of the *r_rule*'s. The test is performed by asserting the body-list components into the database. The training instance is *positive* if the rule *r_head* succeeds and *negative* otherwise (the first training instance is always positive). Then, prefix *r_* is removed from *r_head* to obtain the actual head of the rule. The 'new' head, the body-list and the truth value of the training instance are stored in two files, namely <pos_inst> and <train_inst>, for further use. The next step is to create the initial rule by joining up the head and the body of the training instance with the symbol ':-'

**The two bodies are not always identical as the following example shows:*

r_a_line(A,B):- conn(A,B,1);conn(B,A,1).

a_line(A,B):- (conn(A,B,N);conn(B,A,N)),N>0,N<2.

This is due to the way in which the elementary-concept learner learns.

(*if* in PROLOG). The result is an instantiated form of the rule:

$$head\text{:-}\ body. \qquad (1)$$

This is asserted into the database, while the body-list is removed from it. The body of rule (1) is saved in the file <old_rule> in order to be used by procedure [improve]. Rule (1) is generalized by turning the constant arguments of both legs into variables. During the generalization, common arguments in both legs are replaced by common variables. At this point the system takes care of constants in the rule, that must remain instantiated (e.g. $N>0$, in definition 5.1, 0 is a bound and therefore it is not turned into a variable) and specifies them by asserting predicate: *instantiated(ArgList).* into the database. *ArgList* is the list of variables that will remain instantiated (e.g. [0,2] in the case of 5.1). Procedure [first_rule] ends by printing out the message: *initial rule* and the newly created rule directly below it. The system returns to the main routine and continues by calling procedure [improve] if the user wishes to modify the initial rule. Otherwise, the elementary-concept learner ends with the message: *end of rule improvement* .

Procedure [improve] calls procedure [critic] that attempts to identify existing rule errors by testing the current rule against new training instances. If no error can be detected, the system asks for a new training instance to be tried. Otherwise, procedure [modifier] has the task of correcting the rule. The critic begins by comparing two training instances the first one of which is positive and remains the same throughout the rule improvement. The truth value of the second (current) training instance, as well as the behaviour of the current rule, is determined. The result of these tests specify the type of error according to Table 5.1. In cases 1. and 4. of the Table no error can be

	training instance	truth value of new rule	type of error
1.	positive	true	no error
2.	positive	false	omission error
3.	negative	true	commission error
4.	negative	false	no error

TABLE 5.1

detected and therefore a new training instance should be tried. In cases 2. and 3. of Table 5.1 an error is detected and the [critic] proceeds to determine the discriminating element by comparing the bodies of the two rules and their respective contexts. A positive training instance is indicated by asserting the predicate: *select_cont(HeadFunctor, BodyConditionList, ConditionNo)*. The predicate-name stands for 'selection context' and the arguments are: the functor of the head of the rule, the list of condition that comprise the body of the rule in the instantiated form: [Functor| ArgumentList], and the number of conditions in the body. Similarly, predicate: *reject_cont(HeadFunctor, BodyConditionList,ConditionNo)* is asserted to indicate the rejection context of a negative training instance.

The discriminating element that causes a rule to make a commission error is determined by comparing the last two arguments of a selection and a rejection context. Since a near miss has been assumed (§5.4), the difference in the condition-list of the two bodies will be:

an extra condition, that means that one of the two bodies contains a condition more than the other one, which can be checked by comparing the two *ConditionNo*'s of the two contexts, or

a different condition, that means that the two *ConditionNo*'s are equal but two conditions are different, which is checked by calculating the symmetric difference (as in set theory) from each *BodyConditionList* from the other. An extra condition is indicated by asserting predicate: *discr_factor(HeadFunctor,commission_error,[ExtraCondition])* to the database. A different condition may be caused by a *different functor* or by *different arguments*. In the case of different arguments, the two argument lists are treated as ordered sets and their elements are compared in pairs (1st argument of 1st list with 1st argument of 2nd list etc.). A different condition is indicated by asserting predicate: *discr_element(HeadFunctor,commission_error,[DifferentElement])*, where *DifferentElement* contains the two different condition-factors in the form: [Functor|Arguments], or the pairs of the different arguments in the form: [ArgNCondM1,ArgNCondM2]. The same procedure is followed in order to determine the discriminating element that causes an omission error. The only difference is that the latter is determined by comparing two selection contexts and that the second argument of the asserted *discr_element*-predicate is: *omission_error*. Another important point here is that the case of 'extra condition' is only a commission-error case and therefore a *discr_factor(HeadFunctor,omission_error,[DiffElem])* predicate has no sense. This is true, since a commission error is made because of insufficient constraint in the rule body. This means that if an insufficiently constrained rule is presented with a positive training instance that includes an extra condition it will succeed because the extra condition will have no effect at all. In other words, this situation will never give rise to a pair (positive, false) (see Table 5.1), that indicates an omission error.

The next step of the system is to call the [modifier] in order to correct the faulty rule. Procedure [modifier] proceeds by consulting the discriminating element/factor, and creating a factor that attaches the new condition to the old rule-body. The new rule-body is the conjunction of the old body and new condition. The creation of the new condition is considered in the following three cases:

a) *Extra factor:* the existence of a *discr_factor* predicate implies a commission error with extra factor in place of the difference argument. Since the existence of the extra factor has caused the old rule to behave incorrectly, its absence should be attached to the conditions of the old rule-body. This is achieved by setting the new condition equal to the negation of the extra factor.

b) *Different condition with factors that have different functors or <u>more than one</u> different argument:* if a commission error is indicated, the new condition will be the conjunction of the two different condition-factors. If an omission error has been detected, the disjunction of the two different condition-factors will become the new condition. Obviously, the condition-factor that already exists in the old rule-body is removed before the new condition-factor is added to the body (or more precisely it is replaced by the new condition-factor).

c) *Different condition with factors that have the same functor and <u>only one</u> different argument:* first of all, the *type* of the arguments in the two different condition factors is determined (the type-list here is: *atom*, *integer* and *list*). Then each of the two different arguments is compared with the rest of the arguments in the condition factor it belongs, in order to examine its relationship to them. The type of the arguments dictates the kind of the examined relationship (e.g. if one of the arguments is a list, then one of the tried relationships is that of membership). More precisely, the system looks for a relationship between each discriminating argument and another argument in the selection context that is *not* valid between the two corresponding arguments in the rejection context. If such a relationship is found, then this will be the new condition to be added to the old rule-body. Otherwise, the direct relationship between the two different arguments is used as the new condition. In the case of integer discriminating arguments, the discriminating argument of the rejection context is marked as instantiated, because it acts as a boundary value. The bounds of a particular integer argument are calculated by a special procedure called [constrain], which keeps looping until the rule is correctly constrained. The function that is used to add the new condition to the old body-rule is *conjunction* in the case of a commission error, and *disjunction* in the case of an omission error. The task of the [modifier] is accomplished by providing the system with a correct rule, that is announced by the message: *no error can be detected.* This indicates the end of the improving cycle,as well as the end of the learning cycle for a given rule. The following two examples demonstrate more clearly the main functions of the elementary-concept learner.

Example 1: The user inserts the positive training instance:

line(a,b)   :   conn(a,b,2)

into the system providing the head and the body of a first rule instance. The system generalizes this instance by turning the arguments from constants to variables:

initial rule

line(A,B):- conn(A,B,N).

improve rule ?   y.   (for 'yes' by the user)

insert new training instance

The user inserts a new training instance, in this case an anti-example.

consulted training instance

line(5,b)   :   conn(5,b,2)

The system tests the initial rule on the new training instance according to Table 5.1.

training instance   :   negative

new rule   :   true

type of error   :   commission_error

discriminating element

[[conn,5,b,2],[conn,a,b,2]]

At this point the system detects the pair *[5,a]* as discriminating arguments and before proceeding with any comparisons determines their types. It is obvious that *5* is of the type *integer* and *a* of the type *atom* which is strong enough to provide the new condition, namely *atom(a)*, that will be added with a conjunction (commission error) to the old rule-body to obtain: *line(a,b):- conn(a,b,2),atom(a),* which is generalized.

new rule

line(A,B):-conn(A,B,N),atom(A).

improve rule ? y.

The user inserts another negative training instance: *line(a,a) : conn(a,a,b)*, and the system detects again a commission error with discriminating element

[[conn,a,a,2],[conn,a,b,2]]

The discriminating argument pair is *[a,b]*. Now each argument is compared with the rest of the arguments of its context, to determine the discriminating relationship:

a:[a,a,2] (a=a, a\=2), b:[a,b,2] (b\=a,b\=2)

The first part of the two comparisons reveals that relation '=' is valid between the first and second arguments of the rejection context and not valid between the respective arguments in the selection context. That means that the new condition is *b\=a*, and after generalization, the new rule reads:

line(A,B):-conn(A,B,N),atom(A),B\=A.

As next training instance is used:

line(a,b) :conn(a,b,4),

that will cause another commission error to be detected, this time with discriminating element:

[[conn(a,b,4)],[conn(a,b,2)]]

If the above described procedure is repeated, the system will fail to find a discriminating relation between each of the elements of the pair *[4,2]* and the arguments of their respective contexts, and thus will use their direct comparison *4>2* , as the new condition. In this case however, before going to the usual generalization the system will mark 4 as instantiated in order to prevent it from being turned to a variable. The new rule is:

line(A,B):-conn(A,B,N),atom(A),B\=A,N<4.

At this point the system will retain the old rule-body:

conn(a,b,2),atom(a),b\=a,

as basis for the improvement of the rule. This means that the new condition (*N<n*), will not be added to the old rule-body, before the rule behaviour is tested with respect to the constraint of the argument (*N*). Now the user wishes to test the temporary form of the above rule and supplies it with the training instance:

line(a,b) : conn(a,b,3).

The system detects a commission error and following the same procedure obtains a new form of the rule:

line(A,B):-conn(A,B,N),atom(A),B\=A,N<3.

If the new rule is tested with training instance:

line(a,b) : conn(a,b,2)

will be successful, and the new condition *N<3*, as correctly constrained, will be added to the old rule-body. Thus, the new rule-body that is used as a basis for further rule improvement becomes:

conn(a,b,2),atom(a),b\=a,2<3.

The bound *3* is kept in the list *limits([3])*, where the arguments of the rule that remain instantiated after the generalization are stored.

The system is given the training instance:

line(a,b) : conn(b,a,2),

and has the following reaction:

training instance : positive

new rule : false

type of error : *omission error*

discriminating element

[[conn,a,b,2],[conn,b,a,2]]

The context comparison reveals two argument positions with different arguments, namely: *[a,b],[b,a]*. Thus, the new condition is *conn(b,a,2)* and will be added to the rule with a disjunction (omission error) to give:

line(A,B):-(conn(A,B,2);conn(B,A,2)),atom(A),B\=A,N<3.

It is now obvious how the system proceeds to obtain the final form of the rule:

line(A,B):-(conn(A,B,2);conn(B,A,2),atom(A),atom(B),B\=A,
integer(N),N>0,N<3.

Example 2: The user inserts the positive training instance:

trian(a,b,c) : *conn(a,b,2),conn(b,c,2) conn(c,a,2).*

The system performs first a piece of constructive generalization by using the previously learned concept of a *line(A,B)*, and then uses the usual selective generalization to obtain the initial rule:

trian(A,B,C):- line(A,B),line(B,C),line(C,A).

The next training instance is:

trian(a,b,c) : *conn(a,b,2) conn(b,c,2) conn(c,a,2) point_in_trn(a,b,c),*

that causes the rule to make a commission error with discriminating element the factor: *[[point_in_trn(a,b,c)]]*. This is indicated by asserting the predicate:

discrim_factor(trian(a,b,c),commission_error[[point_in_trn(a,b,c)]]

to the system (case of extra factor). Again the system searches its background knowledge in order to see if the above factor can be substituted by a more general concept and succeeds indeed in finding the concept:

point_trn(a,b,c):-point_in_trn(a,b,c);point_in_trn(b,c,a);
point_in_trn(c,a,b).

The system performs first a discrimination by adding the *negation* of that concept to the old rule-body and then generalizes the result to give:

trian(A,B,C):- line(A,B),line(B,C),line(C,A),not(poin_trn(A,B,C)).

A flow-chart of the elementary-concept learner is presented in Figure 5.10.

5.4.2 The Multiple-View Learner

The multiple-view learner is the part of the learner that learns the definitions of the 3-D figures described by 5.19-5.22 (§5.3.3). It also creates a set of *multi_view_a_fig_list*-predicates (one for each definition), which are to be used by the single-view learner.

The multiple-view learner begins by calling the figure simulator to provide it with the first view of a 3-D figure. This first view acts as the *frontal* view of the objects while the simulator produces five more views, namely: *back*, *top*, *bottom*, *left* and *right*, which are stored in the files: <v_bac>,<v_top>,<v_bot>,<v_lef> and <v_rig> respectively. It is obvious that the frontal view does not include some (at least one) of the faces of the current 3-D figure which are occluded and thus invisible. The system is capable of choosing the next optimal view(s) of the object that reveal the invisible faces, in order to complete its definition. The learning principle of the

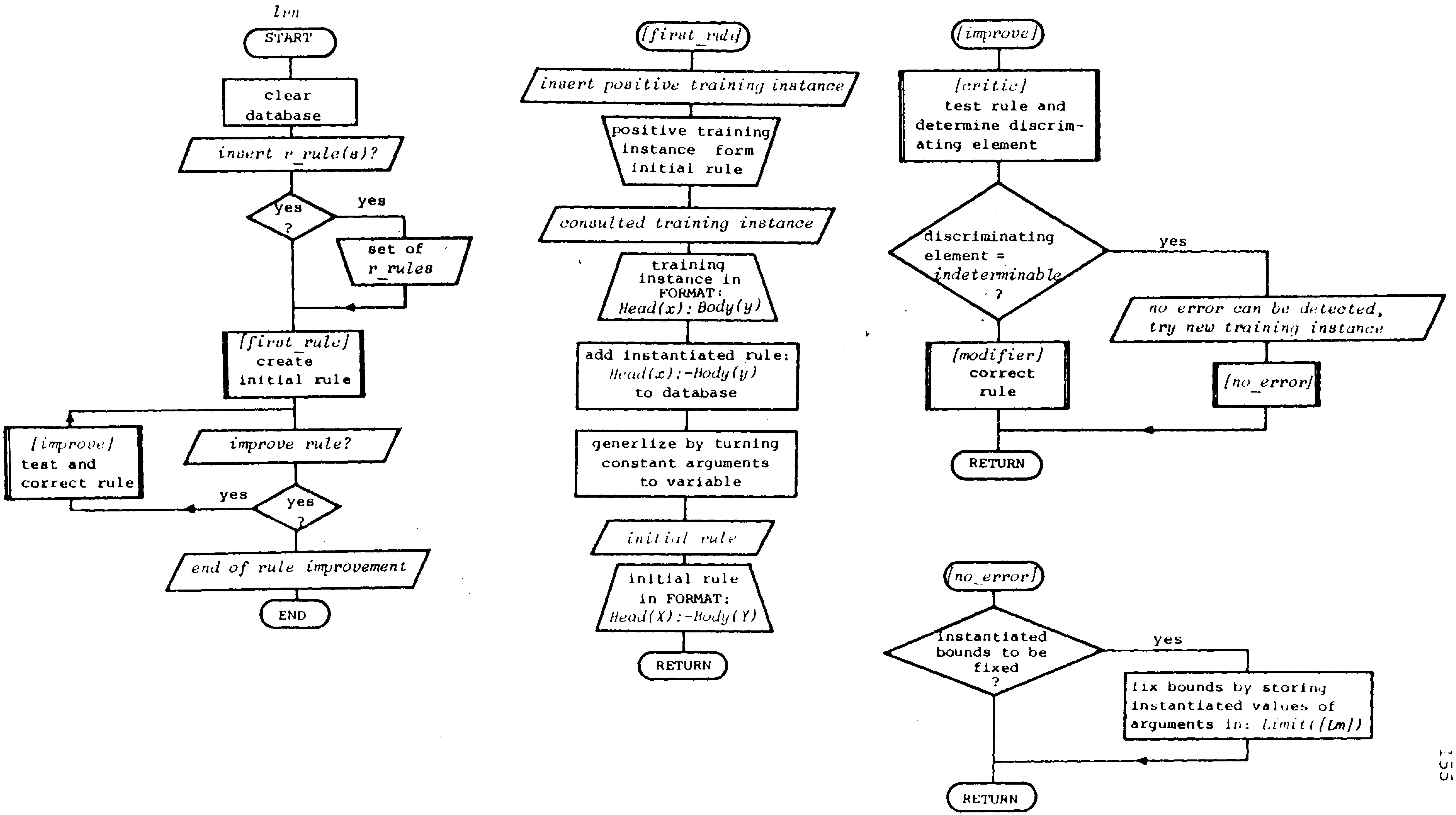
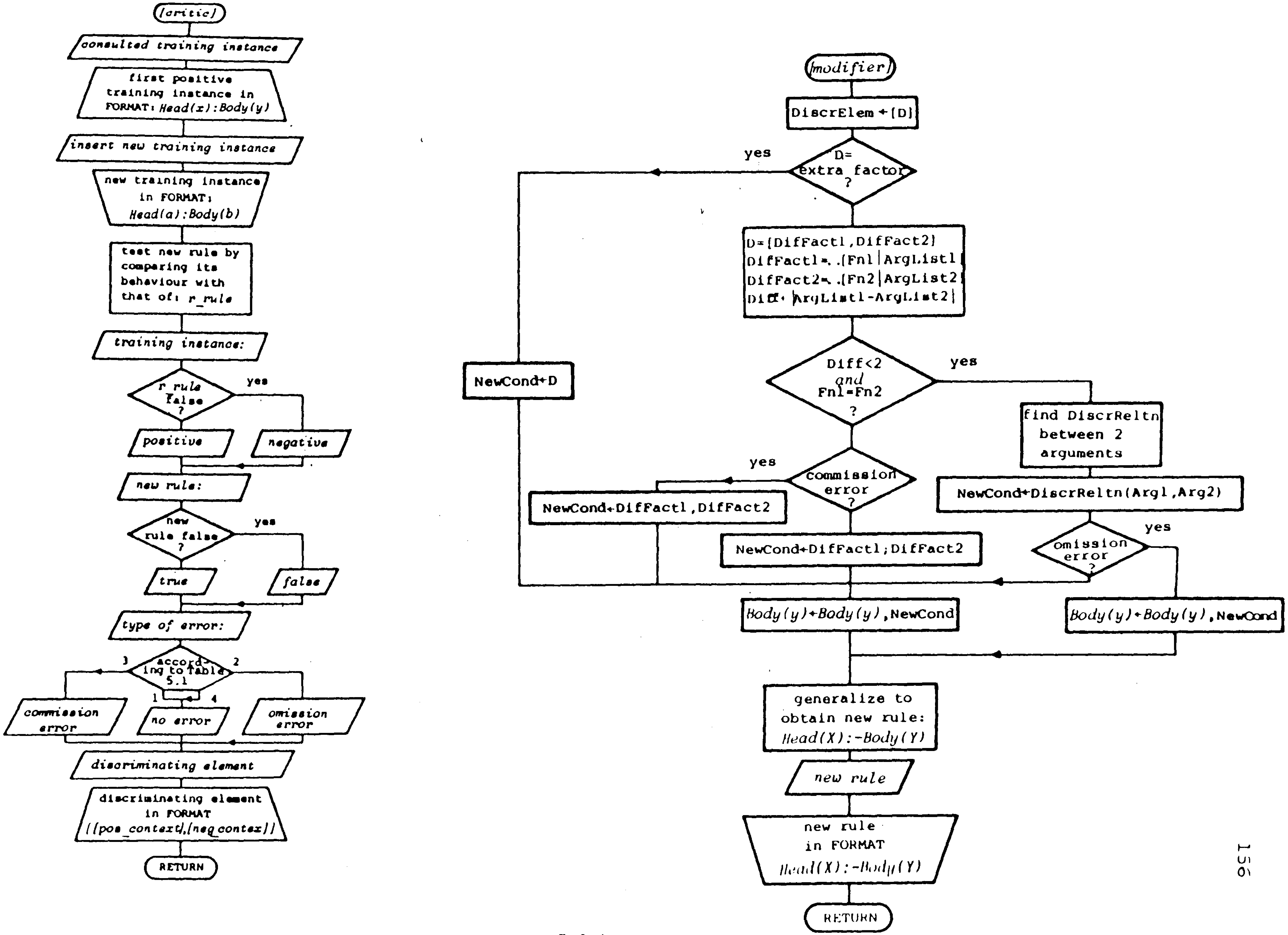

lrn
START
clear database
insert r_rule(s)?
yes ?
yes
set of r_rules
[first_rule] create initial rule
improve rule?
yes ?
yes
[improve] test and correct rule
end of rule improvement
END
[first_rule]
insert positive training instance
positive training instance form initial rule
consulted training instance
training instance in FORMAT: Head(x):Body(y)
add instantiated rule: Head(x):-Body(y) to database
generlize by turning constant arguments to variable
initial rule
initial rule in FORMAT: Head(X):-Body(Y)
RETURN
[improve]
[critic] test rule and determine discriminating element
discriminating element = indeterminable ?
yes
no error can be detected, try new training instance
[no_error]
[modifier] correct rule
RETURN
[no_error]
instantiated bounds to be fixed ?
yes
fix bounds by storing instantiated values of arguments in: Limit([Lm])
RETURN

FIGURE 5.10a



critic
consulted training instance
first positive training instance in FORMAT: Head(x):Body(y)
insert new training instance
new training instance in FORMAT: Head(a):Body(b)
test new rule by comparing its behaviour with that of: r_rule
training instance:
r_rule false ?
yes
positive
negative
new rule:
new rule false ?
yes
true
false
type of error:
according to table 5.1
3
2
1
4
commission error
no error
omission error
discriminating element
discriminating element in FORMAT [[pos_context],[neg_context]]
RETURN
modifier
DiscrElem ← [D]
D= extra factor ?
yes
NewCond←D
D=[DiffAct1,DiffAct2]
DiffAct1=..[Fn1|ArgList1]
DiffAct2=..[Fn2|ArgList2]
Diff←|ArgList1-ArgList2|
Diff<2 and Fn1=Fn2 ?
yes
find DiscrReltn between 2 arguments
NewCond←DiscrReltn(Arg1,Arg2)
commission error ?
yes
NewCond←DiffAct1,DiffAct2
NewCond←DiffAct1;DiffAct2
omission error ?
yes
Body(y)←Body(y),NewCond
Body(y)←Body(y),NewCond
generalize to obtain new rule: Head(X):-Body(Y)
new rule
new rule in FORMAT Head(X):-Body(Y)
RETURN

FIGURE 5.10b

multiple-view learner is the same as in the elementary-concept learner with the following two main differences:

a) The system does not possess an 'oracle' for testing the training instances, which means that the type of these instances must be supplied by the user.

b) The learning process continues until the system has seen enough views in which *all* of the 3-D figure faces are at least *once* visible.

Then the system calls procedure [m_rule] that performs the learning of the multiple-views rules. First the database is cleared and the first view with the format: *Head*, *BodyList*, *ViewValue*, is inserted to the database; where *Head* is the rule-head, *BodyList* the list of predicates on which the forming of the rule-body will be based, and *ViewValue* may be *positive* or *negative*. The same format is used for storing each of the other five views into its respective file. The rule-body consists mainly of *conn*-predicates (depending on the view), as well as *point_in*-predicates and *non_convex*-predicates that are asserted to the database. The system searches the database looking for visible or/and invisible 2-D figures, using the concepts learned in previous phases. Each of these 2-D figures, is a face of the 3-D figure to be learned, and is indicated by asserting a corresponding *a*-predicate (e.g. *atrian(a,b,c)* for a triangular face) to the database. The face-counters of the *conn*'s that make up a face are decremented by 1. As a result of this the new database will consist of *conn(A,B,N)*'s with *N=0* and *N=1*, *special-feature*-predicates, and 2-D *a_figure*-predicates. The same procedure is followed for a second view and the results of the two views are combined to produce a new *augmented* view. The head of this augmented view will have a new argument list equal to the union of argument lists (vertices) of the heads of the two composing views. The body of the augmented view will be similarly the union of the two composing bodies. At this point the sets of *a_figure*'s of the two views are compared, and invisible figures of one view that appear as visible in the other view are removed from the database. At the same time the *point_in*-predicate that indicates this invisibility is removed from the corresponding body since there is no more reason for it to exist. Finally the system forms a union of the *conn*'s that remain after the decrement of their faces-counters. If this list of *conn*'s contains only *conn(A,B,N)*'s, *N=0*, all of the faces in the two views are visible and the systems can create the multiple-view rule for the current object. The instantiated form of the rule will have the head of the augmented view as its head and the disjunction of the 2-D figures that correspond to the augmented view-body as its body. The instantiated form is generalized to obtain the final rule. At the end of the process the system creates the following predicate:

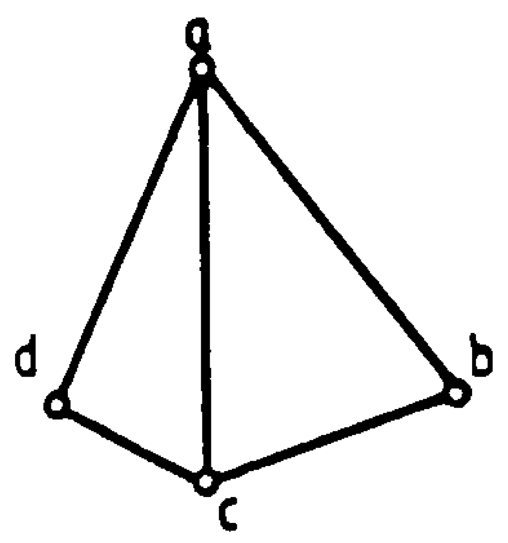
multi_view_a_fig_list(A_Head,A_BodyList,(Faces,Edges,Vertices))

where *A_Head* is the head of the rule prefixed by *a_*, *A_BodyList* is a list of *a_set_2-D_figure*-predicates corresponding to the 2-D figures of the rule-body and the last three arguments are a count of the faces, edges and vertices of the current object respectively. For every new object such a predicate is created in order to be used by the single-view learner.

If the remaining list of *conn*'s contains *conn(A,B,N)*'s with *N=1*, means that another view is needed. The user can choose between a random view supplied by him/herself or the optimal view suggested by the system. The latter guarantees a minimum of views needed to complete the rule. The criterion used for the selection of the optimal next view is quite simple. The system forms three sets of lines i.e. *b_line*'s of the previous new-body before the procedure, and *a_line*'s and *c_line*'s after the procedure. Each of these sets is compared with the body of each of the five other views created by the simulator at the beginning of the learning phase. The next view, in order of decreasing priority, is the one whose view-body contains: the maximum of *b_line*'s that do not belong to the previous view-body, or the maximum of *a_line*'s belonging to the previous view-body, or the minimum of *c_line*'s of the previous view-body. If these priority criteria do not succeed in finding a unique view, the system chooses the one with a *point_in*-predicate in its body. If despite of all that there are still two or more equivalent views, the system proposes all of them. Every time that one of the system-suggested views is used as the next view, the system removes it from the view list, so that it will not be considered again. The function of the multiple-view learner is demonstrated by the following example.

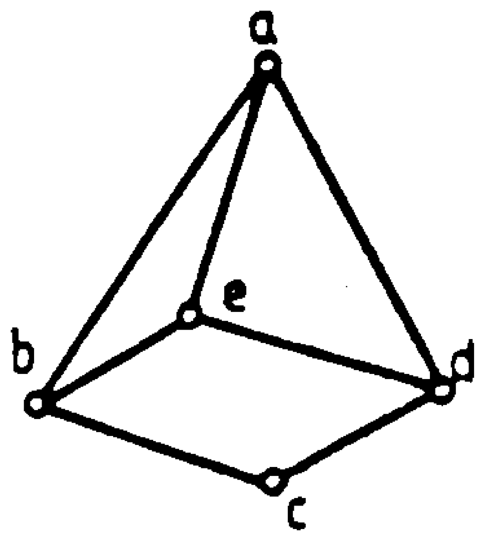
Example: The 3-D figure to be learnt is a *pyramid*. The simulator produces the six basic views of the pyramid and stores them in their respective files as shown in Figure 5.11.

The frontal view is inserted by asserting the predicates composing its body into the database. The system searches for 2-D figures, finds two visible triangles and adds their *a*_predicates to the database:



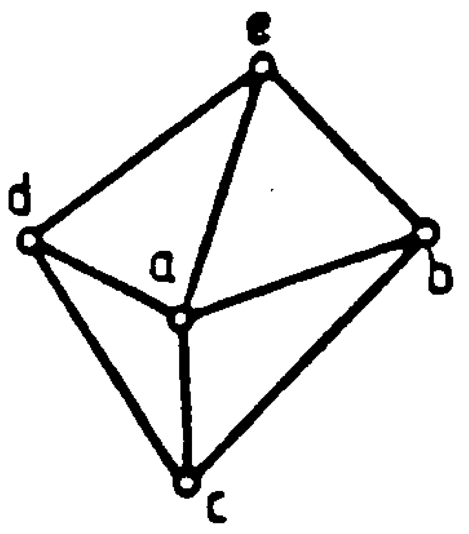
front

```
<pyram>

[pyramid(a,b,c,d),
 [conn(a,b,2),conn(a,c,2),conn(a,d,2),
  conn(b,c,2),conn(c,d,2)],
 positive].
```



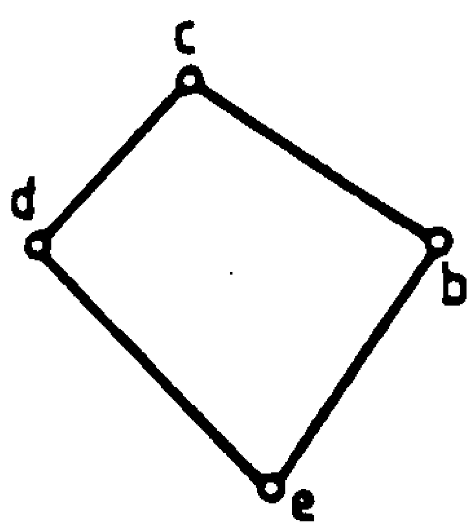
back

```
<v_bac>

[pyramid(a,b,c,d,e),
 [conn(a,b,2),conn(a,d,e),conn(a,e,2),
  conn(b,c,2),conn(b,e,2),conn(c,b,2),
  conn(c,d,2),conn(d,e,2),
   point_in_qul(a,b,c,d)],
 positive].
```



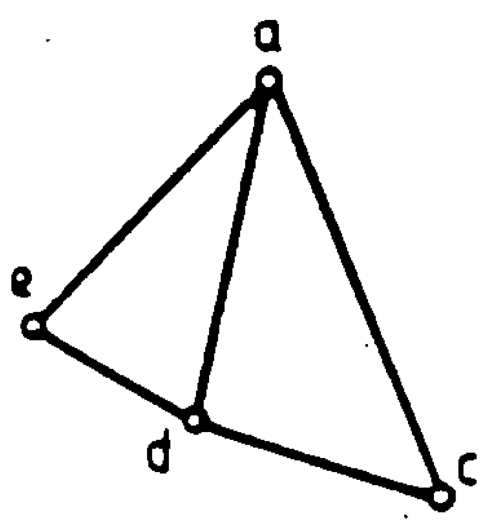
top

```
<v_top>

[pyramid(a,b,c,d,e),
 [conn(a,b,2),conn(a,c,2),conn(a,d,2),
  conn(a,e,2),conn(b,c,2),conn(b,e,2),
  conn(c,d,2),conn(e,d,2),
   point_in_qul(b,c,d,e)],
 positive].
```



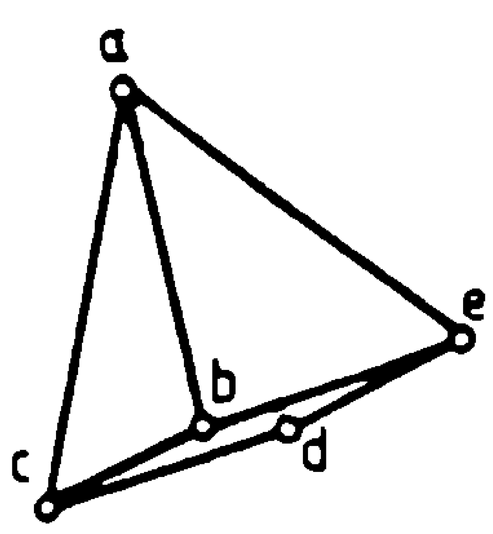
bottom

```
<v_bot>

[pyramid(b,c,d,e),
 [conn(b,c,2),conn(b,e,2),conn(c,d,2),
  conn(d,e,2)],
 positive].
```



left

```
<v_lef>

[pyramid(a,c,e,d),
 [conn(a,c,e),conn(a,d,2),conn(a,e,2),
  conn(c,d,e),conn(d,e,2)],
 positive].
```



right

```
<v_rig>

[pyramid(a,b,c,d,e),
 [conn(a,c,2),conn(a,b,2),conn(d,c,2),
  conn(a,e,2),conn(b,c,2),conn(b,e,2),
  conn(c,d,2),conn(d,e,2),
   point_in_qul(b,c,d,e)],
 positive].
```

FIGURE 5.11

database

conn(a,b,2).
conn(a,d,2).
conn(a,c,2).
conn(b,c,2).
conn(c,d,2).

trian(a,b,c)
trian(a,c,d)

database

$$\left.\begin{array}{l} conn(a,b,1). \\ conn(a,d,1). \\ conn(a,c,0). \\ conn(b,c,1). \\ conn(c,d,1). \end{array}\right\} ConnList1$$

$$\left.\begin{array}{l} atrian(a,b,c). \\ atrian(a,c,d). \end{array}\right\} a\_FigList1$$

The system notices that in the database there are still *conn*'s with facecounter *=1*, which means that there are still invisible faces, and asks for another view. The user decides to ask the system's advice for the selection of the next view. The system forms a list of the *a_line*s (*=conn(A,B,1)*) and *c_line*s (*=conn(A,B,0)*) in the database and gives a first message about the next view:

to include occluded faces, containing lines:

[line(a,b),line(a,d),line(b,c),line(c,d)]

should not contain line(s):

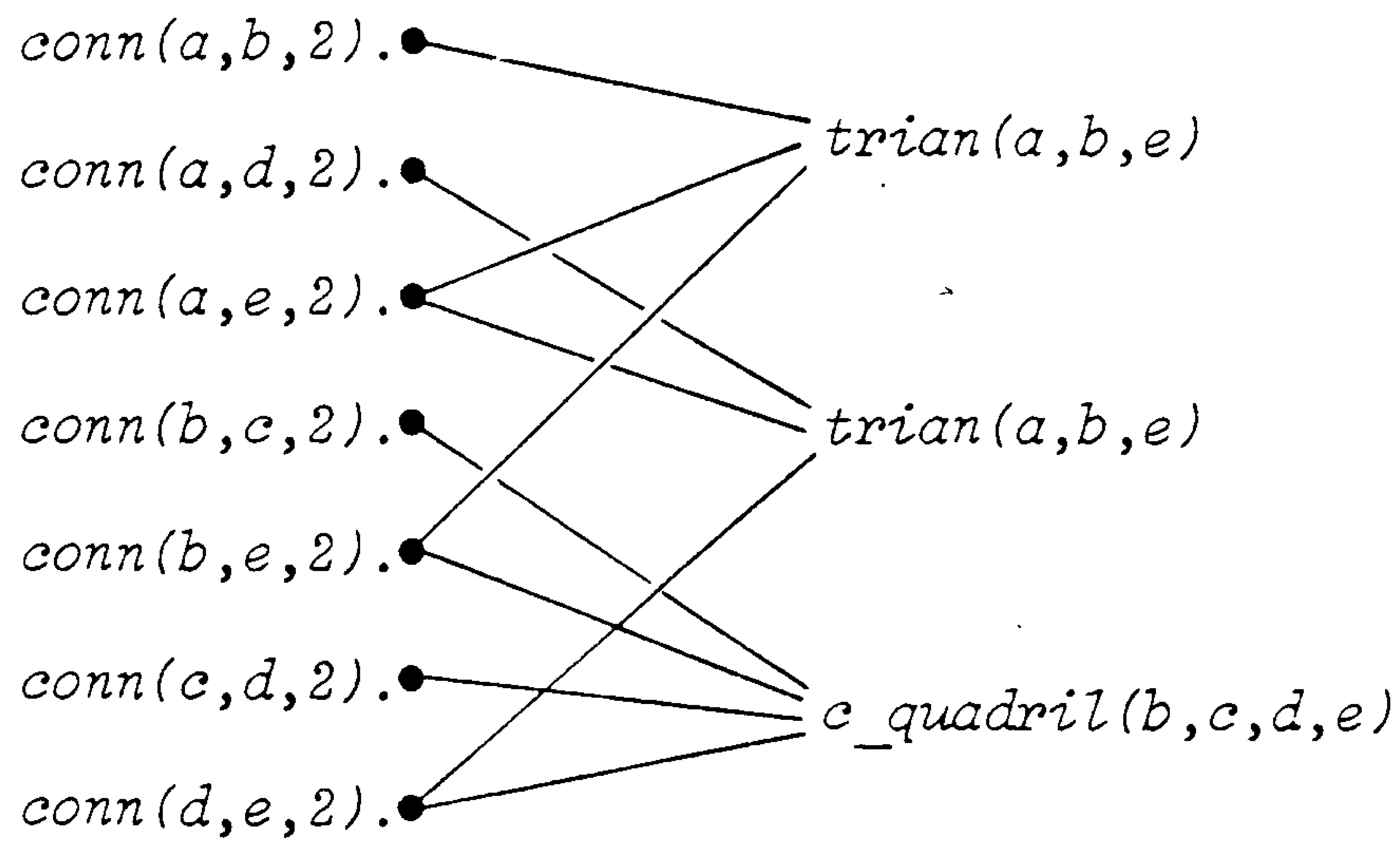
[line(a,c)]

It then searches the bodies of the next five views, looking for the one that has the maximum of new *line*'s in it, which will in this case give the views: *back* and *top*. Since they both contain all of the four *a_line*'s belonging to occluded faces, the containment of *c_line(a,c)* will decide for:

next view from the back

in file : v_bac

The system saves the first view in the file <view2>, the database *ConnList1* and *a_FigList* in the files <conn_list> and <a_fig_list2> respectively, and retracts them from the database. The user decides to insert the next view from the file <v_bac> and tells the system to do so by inputting the name of this file. The system removes this name from the list of next views, inserts the new view as before and stores it in the file <view1>. It then proceeds by looking for 2-D figures:



point_in_qul(a,b,c,d).

conn(a,b,1).
conn(a,d,1).
conn(a,e,0).
conn(b,c,1). } *ConnList2*
conn(b,e,0).
conn(c,d,1).
conn(d,e,0).

atrian(a,b,e).
atrian(a,d,e). } *a_FigList2*
aquadril(b,c,d,e).

It is interesting to notice that, according to the database, there should be one more recognizable 2-D figure, namely a: *p_c_quadril(d,b,c,d)* according to 5.9. The system however eliminates such a possibility, because of the existence of the visible *c_quadril(b,c,d,e)* that shares three points with *p_c_quadril(a,b,c,d)* (they should be coplanar), and removes *poin_in_qul(a,b,c,e)* from the database and from the view-body. At this point the system searches through the *conn*'s in the database in order to find *a_line*'s that belong to two different faces that have already been seen in the last two views. This is done by forming a list of *a_line*'s for each *a_figure* of *a_FigList1* and *a_FigList2*. The set of *a_line*'s that satisfy the above condition are turned to *c_line*'s by setting their face-counter =0.

			database
atrian(a,b,c)	*conn(a,b,1)*	→	*conn(a,b,0)*
atrian(a,c,d)	*conn(a,d,1)*	→	*conn(a,d,0)*
	conn(a,e,0)		
	conn(b,c,1)	→	*conn(b,c,0)*
atrian(a,b,e)	*conn(b,e,0)*		
atrian(a,d,e)	*conn(c,d,1)*	→	*conn(c,d,0)*
aquadril(b,c,d,e)	*conn(d,e,0)*		

ConnList1 is retrieved from <conn_list> and its elements that do not belong to *ConnList2* are asserted to the database. All *a_line*'s of *ConnList2* that appear in the *ConnList1* as *c_line*'s are turned to *c_line*'s. The result is a new set of *conn*'s, that takes the place of the old *ConnList2*. Finally the two views form the augmented view that is saved in <view2>.

Head1(ArgList1)
Head2(ArgList2) *(ArgList=ArgList1* ∪ *ArgList2)* → *head(ArgList)*

in this case: *pyramid(a,b,c,d,e)*

[BodyList1]
[BodyList2] *(BodyList=BodyList1* ∪ *BodyList2)* → *[BodyList]*

here:

[conn(a,b,2),conn(a,c,2),conn(a,d,e),conn(a,e,2),conn(b,c,2), conn(c,d,2),conn(d,e,2)]

and:

ViewValue1=ViewValue2=ViewValue

is always positive since all views are positive training instances. Thus the augmented view is:

view(Head(ArgList),[BodyList],ViewValue)

The system examines *ConnList2* and finds no *a_line*'s in it, which means that the two views left no part of the current 3-D figure invisible, and signals that the system is ready to create the first rule. First the database is cleared from all *conn*'s and *a_figure*'s and the body of the augmented view is input and acts as the new database. The system performs a piece of constructive generalization by looking for all the 2-D figures in the database. This will give:

[trian(a,b,c),trian(a,c,d),trian(a,d,e),trian(a,b,d),c_quadril(b,c,d,e)]

which is the instantiated form of the rule-body. The rule-head is the view-head. By turning the constant to variables:

multiple-view rule

pyramid(A,B,C,D,E):-trian(A,B,C),trian(A,C,D),trian(A,D,E),
trian(A,B,D),c_quadril(B,C,D,E).

The system looks for all the *a_figure*'s in the database, generalizes them to *set_a_figure*'s and creates:

multiple-view a_figure list

multi_view_a_fig_list(a_pyramid(A,B,C,D,E),[set_atr(A,B,C),set_atr(A,C,D),
set_atr(A,D,E),set_atr(A,B,D),set_aqu(B,C,D,E),(5,5,5))

If the user decided to try the bottom-view instead of the proposed back-view, the system would come to the same result, with the only difference that it would need at least another view and one more cycle. This is obvious because the frontal view and the bottom view leave triangles (aed) and (abe) invisible. The system would again suggest back view as the optimal next view. A flow chart of the multiple-view learner is shown in Figure 5.12.

5.4.3 The Single-View Learner

The single-view learner is responsible for learning the definitions of the possible 3-D figures described by 5.23-5.39 (§5.3). Its structure is very similar to that of the elementary-concept learner. It contains the procedure [first_s_rule] that creates the initial rule, and the procedure [s_improve] that improves it. The latter consists of the procedure [s_critic], that detects faults of the rule and uses the procedure [modifier] (same as in [lrn]) to correct them. However, there are certain differences between the two learners with respect to rule formation.

The single-view learner uses the head of the training instance (view) only as a reference to it. The rule-head is formed by finding the best match between the rule-body and the bodies of the four *multi_view_a_fig_list*-predicates that have been learnt by the multiple-view learner. The head of the *multi_view_a_fig_list*-predicate corresponding to the best match will be the head of the rule. Obviously, the body of a single view may match more than one body of a multiple view, which means that there may be more than one candidate for the rule-head. In this case the number of faces, edges and vertices of the views play the deciding role. The head of the multiple view with the minimum number of faces or edges (if same number of faces) or vertices will be the rule-head. In order to avoid the creation of two rules with the same head (two best matches of different bodies may

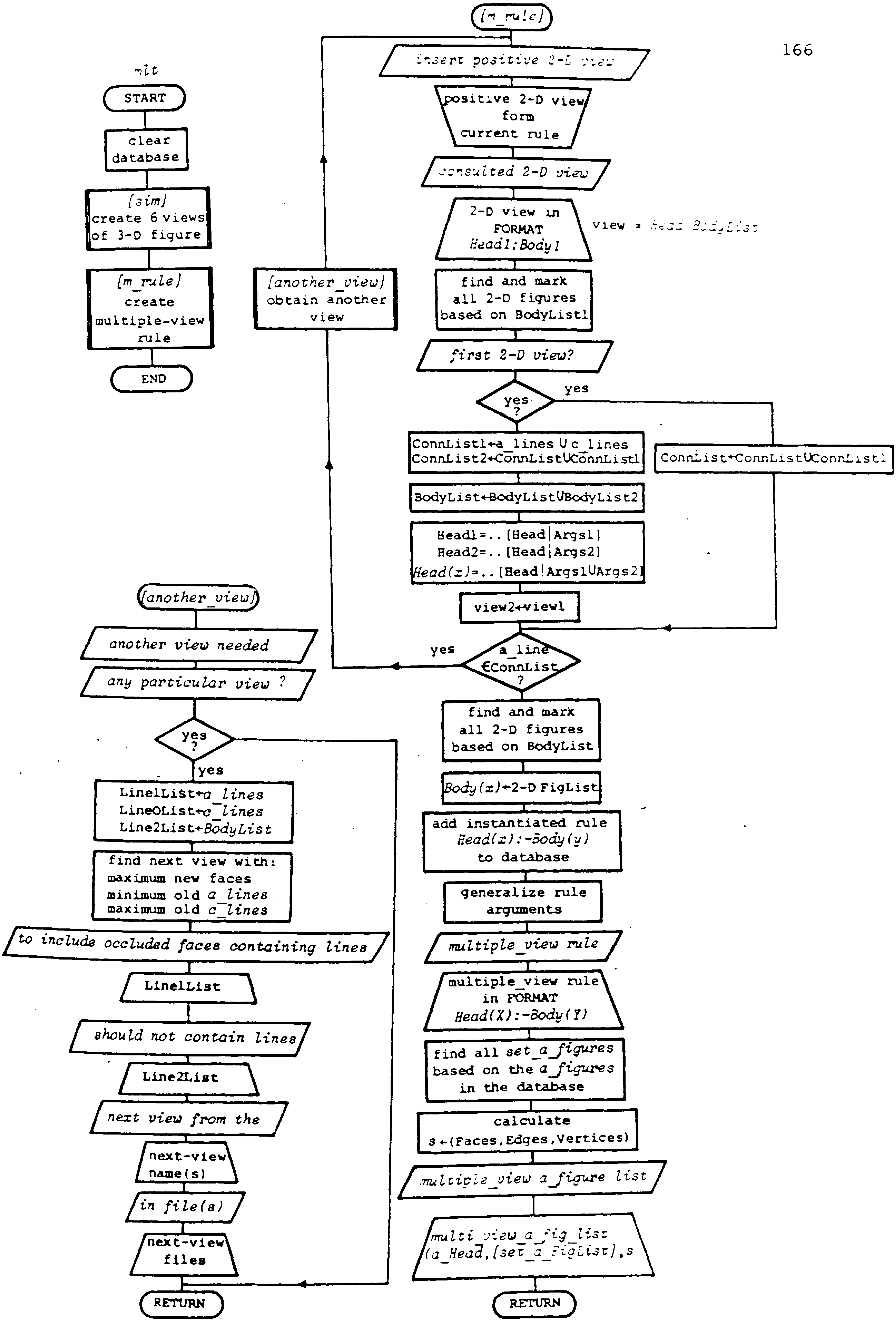
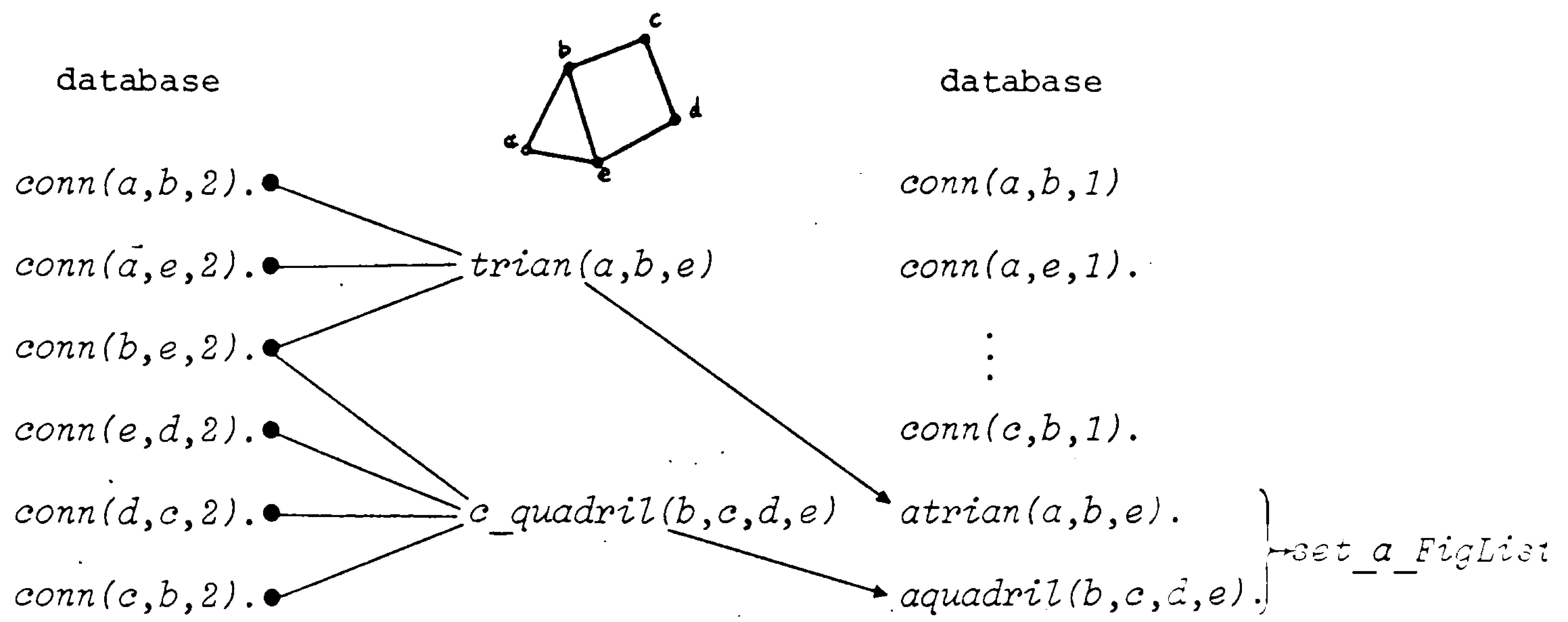

mit
START
clear database
[sim] create 6 views of 3-D figure
[m_rule] create multiple-view rule
END
[m_rule]
insert positive 2-D view
positive 2-D view form current rule
consulted 2-D view
2-D view in FORMAT Head1:Body1
view = Head BodyList
find and mark all 2-D figures based on BodyList1
first 2-D view?
yes ?
yes
ConnList1←a_lines U c_lines
ConnList2←ConnListUConnList1
ConnList←ConnListUConnList1
BodyList←BodyListUBodyList2
Head1=..[Head|Args1]
Head2=..[Head|Args2]
Head(x)=..[Head|Args1UArgs2]
view2←view1
a_line ∈ConnList ?
yes
[another_view] obtain another view
find and mark all 2-D figures based on BodyList
Body(x)←2-D FigList
add instantiated rule Head(x):-Body(y) to database
generalize rule arguments
multiple_view rule
multiple_view rule in FORMAT Head(X):-Body(Y)
find all set_a_figures based on the a_figures in the database
calculate s←(Faces,Edges,Vertices)
multiple_view a_figure list
multi_view_a_fig_list (a_Head,[set_a_FigList],s
RETURN
[another_view]
another view needed
any particular view ?
yes ?
yes
Line1List←a_lines
Line0List←c_lines
Line2List←BodyList
find next view with: maximum new faces minimum old a_lines maximum old c_lines
to include occluded faces containing lines
Line1List
should not contain lines
Line2List
next view from the
next-view name(s)
in file(s)
next-view files
RETURN

FIGURE 5.12

give the same head), an integer equal to the missing faces is added as suffix to the end of the *head functor* (name). If inspite of this there are still two identical heads, the second one will get an extra suffix *a*. The rule-body is formed by looking for *set_a_figure*'s (constructive generalization) based on the body list of the training view. Finally the value of the view (positive/negative) is given by the user because the system does not possess an 'oracle'. The following example clarifies the function of the single-view learner.

Example: The system is given the following view: *(pyramid,[conn(a,b,2), conn(a,e,2),conn(b,e,2),conn(e,d,2),conn(d,c,2),conn(c,b,2)],positive).*

looks at all the 2-D figures, marks them by asserting their respective *a_figure*'s and forms a set of all *set_a_figure*'s.



set_a_FigList=[set_atr(a,b,e),set_aqu(b,c,d,e)].

Then the system forms a list of the *multi_view_a_fig_list*-predicates and instantiates (to a certain extent) before performing the matching.

(a_tetra(a,b,e,c),[set_atr(a,b,e),set_atr(a,b,c),set_atr(a,e,c),

set_atr(b,c,e),(4,6,4)).

```
(a_pyram(a,b,e,c,d),[set_atr(a,b,e),set_atr(a,c,d),set_atr(a,d,e)
                     set_atr(a,b,e),set_aqu(b,c,d,e)],(5,5,5)).
(a_prism(a,b,e,c,d,F),[set_atr(a,b,e),set_atr(e,d,F),set_aqu(b,c,d,e)
                     set_aqu(a,b,e,F),set_aqu(a,c,d,F)],(5,9,6)).
(a_box(a,b,e,c,d,F,G,H),[set_aqu(b,c,d,e),set_aqu(e,d,F,a),
                     set_aqu(d,c,G,F),set_aqu(b,c,G,H),set_aqu(b,H,a,e),
                     set_aqu(a,F,G,H)],(6,12,8))
```

Obviously, the system finds that *a_pyram(a,b,e,c,d)* and *a_prism(a,b,e,c,d)* match, and *a_pyram* is preferred to *a_prism* as rule head because it has fewer edges ($5<9$) since they both have 5 faces. The system prints out both heads in the order of the above priority, in order to enable the user to use two entries *(square-pyramid* and *triangular-prism)*, using the same single-view rule, in the construction of the single view recognizer. Then the system adds suffix *3* (=5-2 missing faces) to *a_pyramid*. The instantiated form of the body is *set_a_FigList*, and after the generalization of arguments the initial rule is formed.

alternative rule heads

a_pyram

a_prism

initial single-view rule

a_pyram3(A,B,E,C,D):- set_atr(A,B,E),set_aqu(B,C,D,E).

The next view is: *(pyramid,[conn(a,b,2),conn(a,e,2),conn(b,e,2), conn(e,d,2),conn(d,a,2),conn(a,b,2)],negative)*. The system calls the [s_critic] and detects a commission error with discriminating element:

[[conn,d,a,2],[conn,d,c,2]],[conn,a,b,2],[conn,c,b,2]]

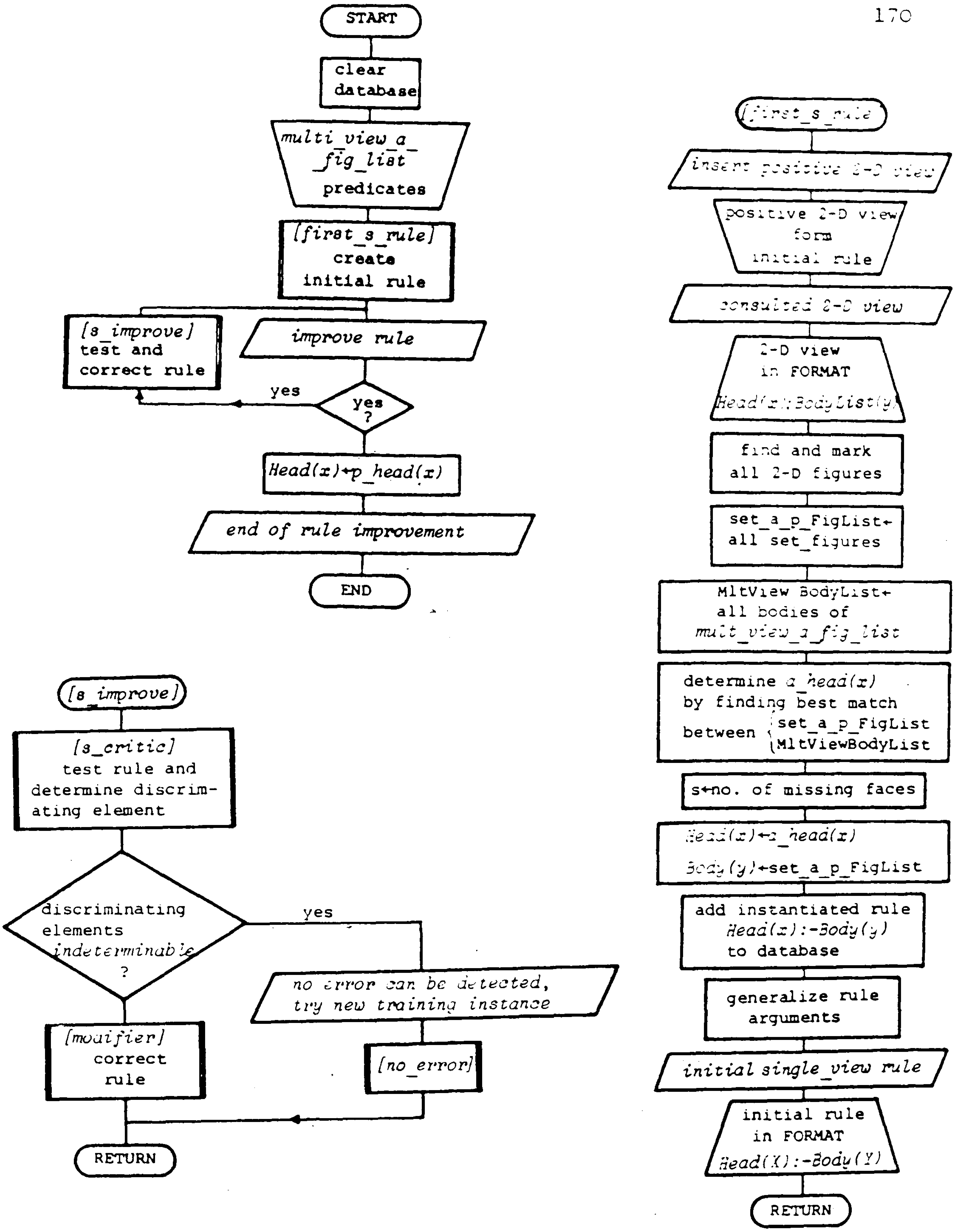
which gives the new condition: *a\=c*. Thus, the new rule becomes:

a_pyram3(A,B,E,C,D):-set_atr(A,B,E),set_aqu(B,C,D,E), A\=C.

Similarly condition *A\=D* can be added to the rule. It is worth noticing that, conditions *A\=B*, *B\=E* and *E\=A* are implied in the triangle definition, and *B\=C*, *B\=D*, *C\=D*, *C\=E* and *D\=E* in the quadrilateral definition. This means that they do not need to be learned. The final view is: *(pyramid,[conn(a,b,2),conn(a,e,2),conn(b,e,2),conn(e,d,2), conn(d,c,2),conn(a,b,2),non_convex_contour_angle(b)],negative)*. This will cause the critic to detect a commission error with the extra factor *non_convex_contour_angle(b)*, as discriminating element. The negation of the new condition (discrimination) is added to the old rule-body to form the new rule:

a_pyram3(A,B,E,C,D):-set_atr(A,B,E),set_aqu(B,C,D,E),A\=C,
not(non_convex_contour_angle(B)).

Finally once the final form of the rule has been reached, the system changes prefix *a_* of the head into *p_* to obtain rule 5.34. A flow chart of the single-view learner is given in Figure 5.13.



[modifier], *[no_error]* are the same as in FIGURE 5.10a

[s_critic] is the same as *[critic]* in FIGURE 5.10b without **test new rule by comparing its behaviour with that of: *r_rule***

FIGURE 5.13

REFERENCES

1. ARBAB, B. and MICHIE, D. 1985: *Generating Rules from Examples*, Proc. 9^{th} IJCAI, pp. 631-633.

2. BUCHANAN, B.G. and FEIGNBAUM, E.A. 1978: *DENDRAL and Meta-DENDRAL: Their Applications Dimension*, Artificial Intelligence, Vol.11, pp.5-24.

3. BUNDY, A. and SILVER, B. 1982: *A Critical Survey of Rule Learning Programs*, D.A.I. Research Paper No. 169, Dept. of AI, Univ. of Edinburgh.

4. DIETTERICH, T. and MICHALSKI, R.S. 1981: *Inductive Learning of Structural Descriptions*, Artificial Intelligence, Vol.16.

5. EMDE, W., HABEL, C.U. and ROLLINGER, C. 1983: *The Discovery of the Equator or Concept Driven Learning*, Proc. 8^{th} IJCAI, pp. 455-458.

6. HAYES-ROTH, F. and McDERMOTT, J. 1977: *Knowledge Acquisition From Structural Descriptions*, Proc. 5^{th} IJCAI, pp. 356-362.

7. IBA, G.A. 1979: *Learning Disjunctive Concepts from Examples*, Master's Thesis, MIT, Cambridge, Mass.

8. KIBLER, D. and PORTER, B. 1983: *Perturbation: A Means for Guiding Generalization*, Proc. 8^{th} IJCAI, pp.415-418.

9. LANGLEY, P. 1981: *Language Acquisition Through Error Recovery*, CIP Working Paper 432, Carnegie-Mellon Univ. June.

10. LOISEL, R. and KODRATOFF, Y. 1981: *Learning (Complex) Structural Descriptions from Examples*, Proc. 7^{th} IJCAI, pp. 141-143.

11. MICHALSKI, R.S. 1980: *Pattern Recognition as Rule-guided Inductive Inference*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No.4, pp. 349-361.

12. MITCHELL, T.M. 1978: *Version Spaces: An Approach to Concept Learning*, Ph.D. Thesis, Stanford Univ. December.

13. NEVES, D. 1985: *Learning from Examples and Doing*, Proc. 9th JICAI, pp. 624-630.

14. SEGEN, J. 1985: *Learning Concept Descriptions from Examples with Errors*, Proc. 9th IJCAI, pp. 634-636.

15. SILVER, B. 1983: *Learning Equation Solving Methods from Examples*, Proc. 8th JICAI, pp. 429-431.

16. VERE, S.A. 1977: *Induction of Relational Productions in the Presence of Background Information*, Proc. 5th IJCAI, pp. 349-355.

17. WINSTON, P.H. 1970: *Learning Structural Descriptions from Examples*, Ph.D. Dissertation, MIT AI Lab., September.

18. WINSTON, P.H. 1975: *Learning Structural Descriptions from Examples:* The Psychology of Computer Vision, Ed. Winston, P.H., McGraw Hill, pp. 157-209.

19. WYSOTZKI, F., KOLBE, W. and SELDIG, J. 1981: *Concept Learning by Structured Examples - An Algebraic Approach*, Proc. 7th IJCAI, pp. 153-158.

CHAPTER 6

THE RECOGNIZER

6.1 INTRODUCTION

The recognizer is the program that attempts to answer the question: *'What do I see?'*, when it is presented with a set of line drawings. Of course the answer will depend strongly on how these line drawings relate to the ones that were 'learnt' in the learning-phase.

Input to the recognizer is a set of *conn*-predicates (defined in §5.3.1) describing a line drawing, which depicts a certain 3-D scene. For the objects comprising the 3-D scene the following assumptions are made:

a) They do not have common elements, i.e. they do not share any sides or vertices.

b) They do not occlude* each other, i.e., they give rise to perfect line drawings which do not intersect.

c) They may be non-convex, but orientated so that no partially occluded faces** occur.

The recognition is performed in three stages. Firstly, all the 2-D 'recognizable' figures are extracted from the set of *conn*-predicates, in what could be potential faces of 3-D figures. Thus, the line drawing is decomposed into n-vertex (here $3 \leq n \leq 4$) closed 2-D figures, representing possible planes. The second stage examines whether these

---

*,** *These two assumptions depend partially on the line drawings created by the simulator and partially on the recognizer. Suggestions to cover these cases are examined in Chapter 7.*

possible planes are visible or not and marks the non-convex ones. At the end of this stage, predicates carrying information about the visibility and the non-convexity of the 2-D figures are passed on to the last stage. The final stage has now enough information to express a firm opinion on what figures make up the scene at first sight. Of course this first sight, one view answer of the recognizer may not be unique. This is to be expected since several 3-D figures can give rise to the same line drawing when viewed from certain angles. A further procedure seeks the possibility of a unique solution after an assumption for the invisible part of the figure has been made.

The basic task of the recognizer, is to 'have a look' at a 'picture' of a 3-D scene and determine whether there are recognizable objects in it (answer to the question: 'What do I see?'). It can also be more specific about certain objects in the scene (answer to question: 'Is there a ... in the scene?') and determine what this object is. The first and third part of the recognizer are written in PROLOG and they require PROLOG-predicates as input. The second part is written in C because it involves several numerical calculations and PROLOG would need much more time to perform; as input it requires normal cartesian coordinates.

6.2 PICTURE SEGMENTATION

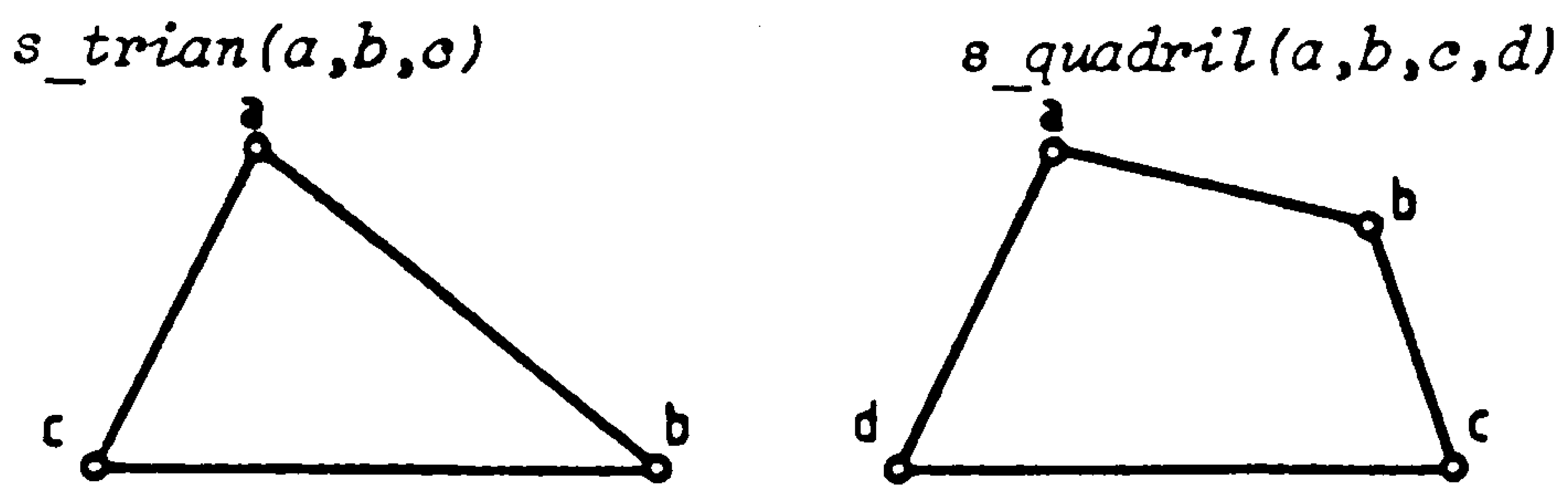
This part of the recognizer, searches for sets of *conn*-predicates connected in a cyclic order. Thus, the data are grouped into groups of three and four, since triangles and quadrilaterals are the only recognizable planes that have been learnt by the previous program, the learner. The goal of the process at this stage is to examine all the combinations of *conn*-predicates, looking for possible planes visible or invisible. Because of the purpose of this part of the routine, it is not necessary to use the strict definitions of triangles and quadrilaterals. Therefore, a triangle is a 2-D figure consisting of three points a,b and c, connected with straight-line segments (Fig. 6.1a). The following statement gives a sufficient definition of a triangle:

$$\mathit{s\_trian}(A,B,C)\text{:-}\mathit{line}(A,B),\mathit{line}(B,C),\mathit{line}(C,A). \tag{6.1}$$

Similarly, a quadrilateral consists of four points a,b,c and d, connected with straight-line segments (Fig. 6.1b). Care should be taken that no diagonally opposite vertices coincide (no degenerate quadrilaterals). The other important condition is, that none of the diagonal lines be present. Presence of a diagonal implies automatically a *non* planar figure. Considering these two conditions, the definition of a quadrilateral is given by:

$$\begin{aligned}&\mathit{s\_quadril}(A,B,C,D)\text{:-}\mathit{line}(A,B),\mathit{line}(B,C),\mathit{line}(C,D),\mathit{line}(D,A)\\ &\qquad A\backslash{=}C,B\backslash{=}D,\mathit{not}(\mathit{line}(A,C)),\mathit{not}(\mathit{line}(B,D)).\end{aligned} \tag{6.2}$$

where *s_* stands for *segmentation*.



(a) (b)

FIGURE 6.1

The segmentation proceeds as follows: The data is searched for *conn*-predicates that satisfy the two definitions 6.1 and 6.2. Thus, a list of triangles and quadrilaterals is formed. Every new 2-D figure is compared with the members of the list, in order to check if it is already there. If it is met for the first time, it becomes a member of the list. Figure 6.2 demonstrates how a tetrahedron and a pyramid are segmented to triangles and quadrilaterals.

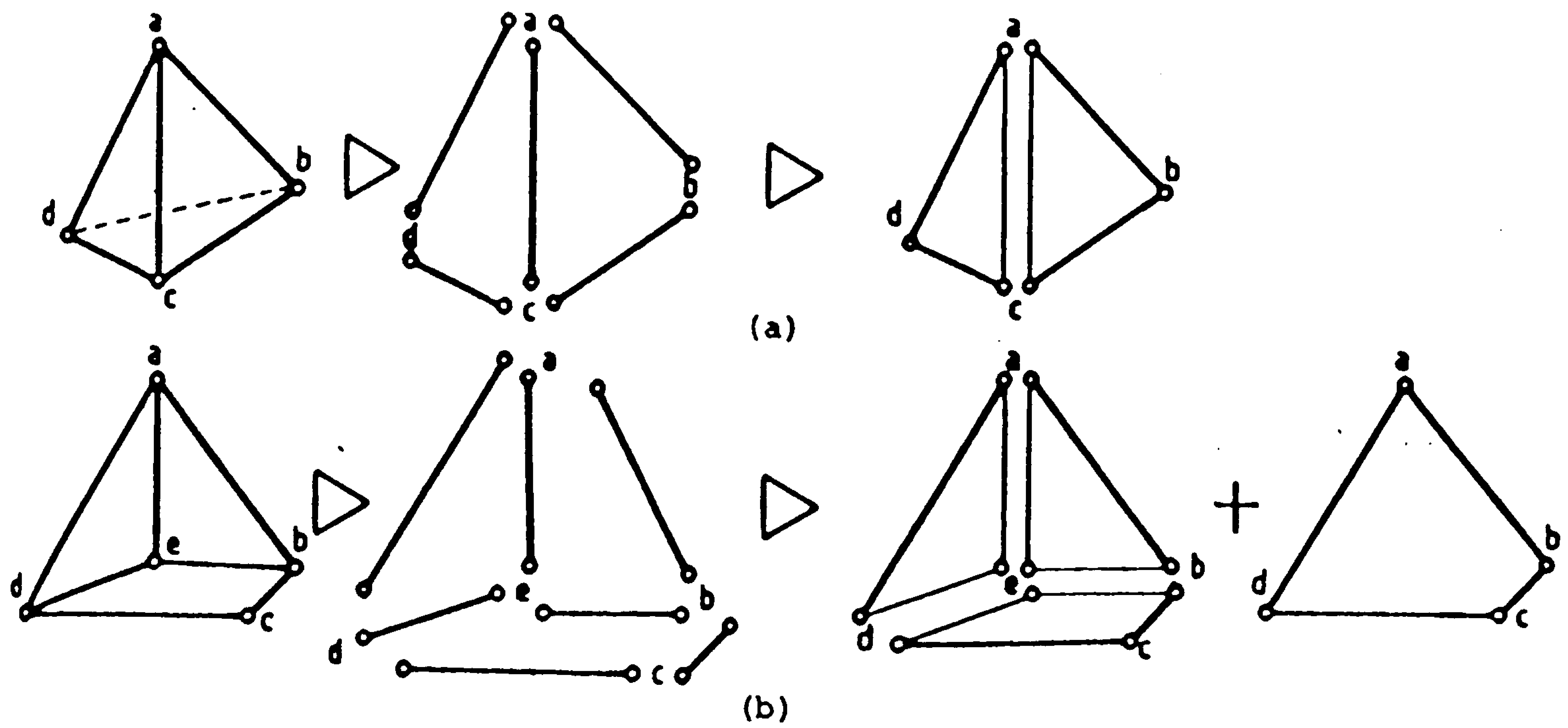


FIGURE 6.2

In Figure 6.2a tetrahedron (abcd) is decomposed to its two visible triangles (abc) and (acd). In Figure 6.2b pyramid (abcde) is split into triangles (abe),(aed) and quadrilaterals (ebcd), (abcd). From these only the first three are visible and therefore planar 2-D figures. The fourth figure, although it has a 'legal' contour, according to the *s_quadril* definition, it is in fact non-planar. This shows, that this first segmentation, is not always correct with respect to the 2-D figures that it considers as planar. The reason for this, is that the 2-D figures that are wrongly considered as planar, are actually invisible, (*s_quadril(a,b,c,d)* is occluded by the other three faces of the pyramid). These simple observations lead to the conclusion, that connectivity of points (i.e. *conn*-predicates only) are not sufficient enough to describe the scene. Some additional predicates are required. These are calculated and supplied by the procedure of the next section.

Before the next procedure is called, the database is converted from PROLOG-predicates into normal C-variables. More specifically, a PROLOG-predicate: *s_trian*(v1,v2,v3), becomes line: 1 2 3 0 in the file <vertex>. This file contains as many lines as there are 2-D figures, each line corresponding to one 2-D figure. The end of every line as well as the end of the file are characteristically marked (0 is here the *end marker*).

The function of the segmentation procedure is summarized in the flowchart of Figure 6.3.

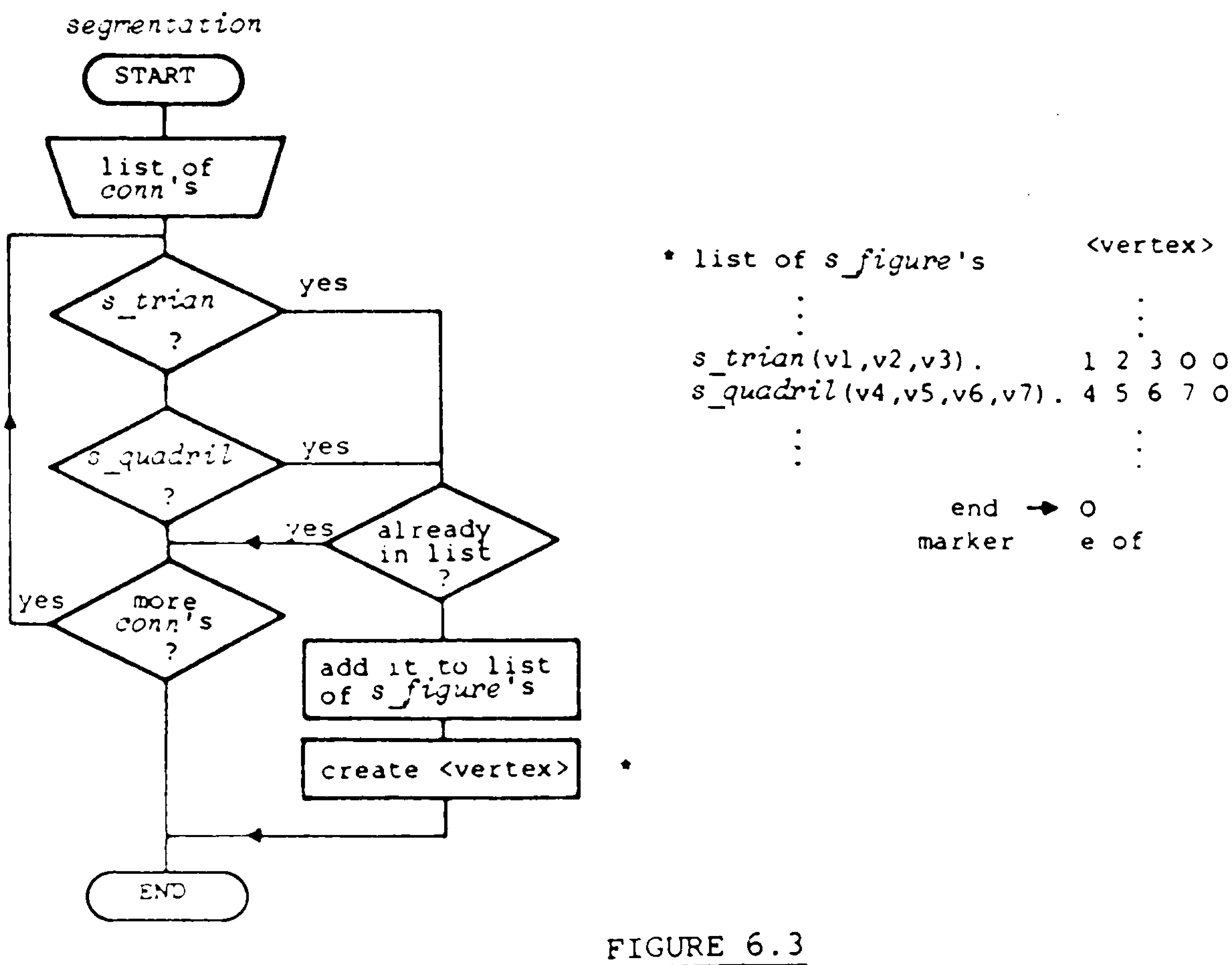


FIGURE 6.3

6.3 SPECIAL FEATURES - LINK

The special features determined by this procedure are: *containment of vertices*, *convexity of faces* and *convexity of contour*. These three features are very important for the recognition phase because they act as supplementary data to the *conn*-predicates carrying crucial cues about the scene. Vertex containment implies that an *s_figure* is invisible and should not be considered as a legal (planar) face, although there may be enough evidence for this (Fig. 6.4a). Face convexity determines whether the object to which it belongs, is recognizable or not (Fig. 6.4b). Objects with non-convex faces are not learnt to be recognized. Finally contour convexity determines whether the object in question is convex (i.e. recognizable) or not (Fig. 6.4c). Without contour convexity the two 2-D figures of Figure 6.4c would be identically recognized.

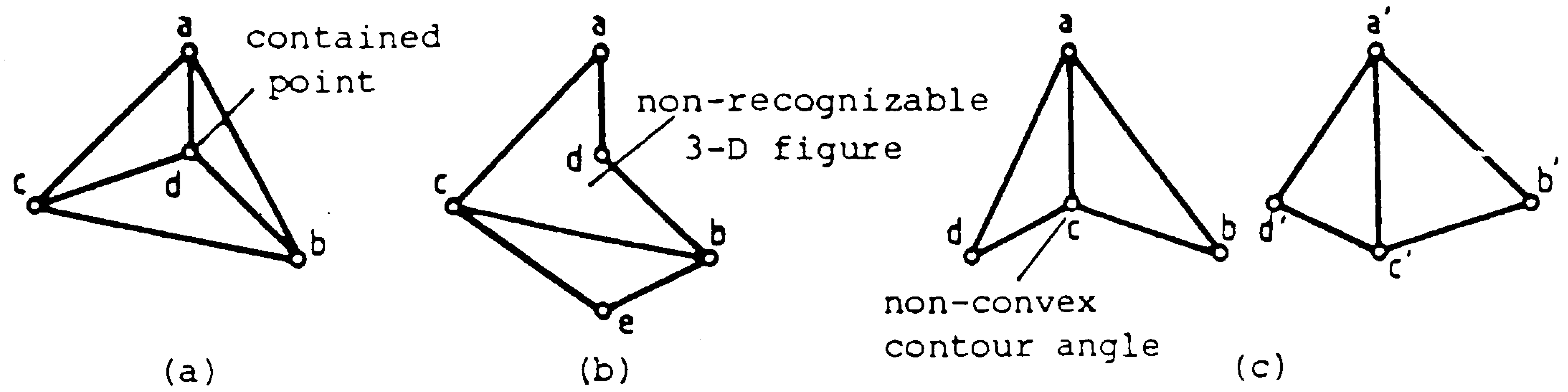


FIGURE 6.4

Before the individual routines for the extraction of the three features are examined, a note should be made about the way the vertex array is reconstructed. A 2-D array, called the *figure array*, [Bennett '83] is created from the data delivered by the segmentation routine. The rows of the figure array contain integers representing the vertices of the 3-D figure which lie on the same face. Every column is a new face. These numbers correspond exactly to the subscripts of the vertex-array that was used by the simulator for the creation of the scene. Thus, the coordinates of the vertices of every face can be retrieved again. In the case of scenes with more than one object, a test separates the faces that belong to different objects. Normally faces with more than two common vertices belong to the same object. The special feature extraction routine is repeated for every object in the scene.

6.3.1 Containment of a Vertex

The principle followed here in order to determine whether a vertex (point) is contained by a polygon of 3 or 4 sides is simple [Harrington '83a]. Firstly, the case of a point contained by a triangle is examined, since every quadrilateral can be split into two triangles.

Every line divides a plane into two half-planes. A point on this plane will be contained by one of the two half-planes, depending on which side of the line it lies (Fig. 6.5a). The containment of a point by a certain half-plane is determined by finding another point that belongs to this half-plane and then comparing the relative positions of the two points with respect to the line that defines the half-plane. If the two points lie on the same side of the line, then they belong to the same half-plane. Each side of a triangle divides the plane determined by its three vertices, into two half-planes. A point that lies on the same half-plane with the third vertex, for all three sides and all three vertices of the triangle, is contained by it (Fig.6.5b).

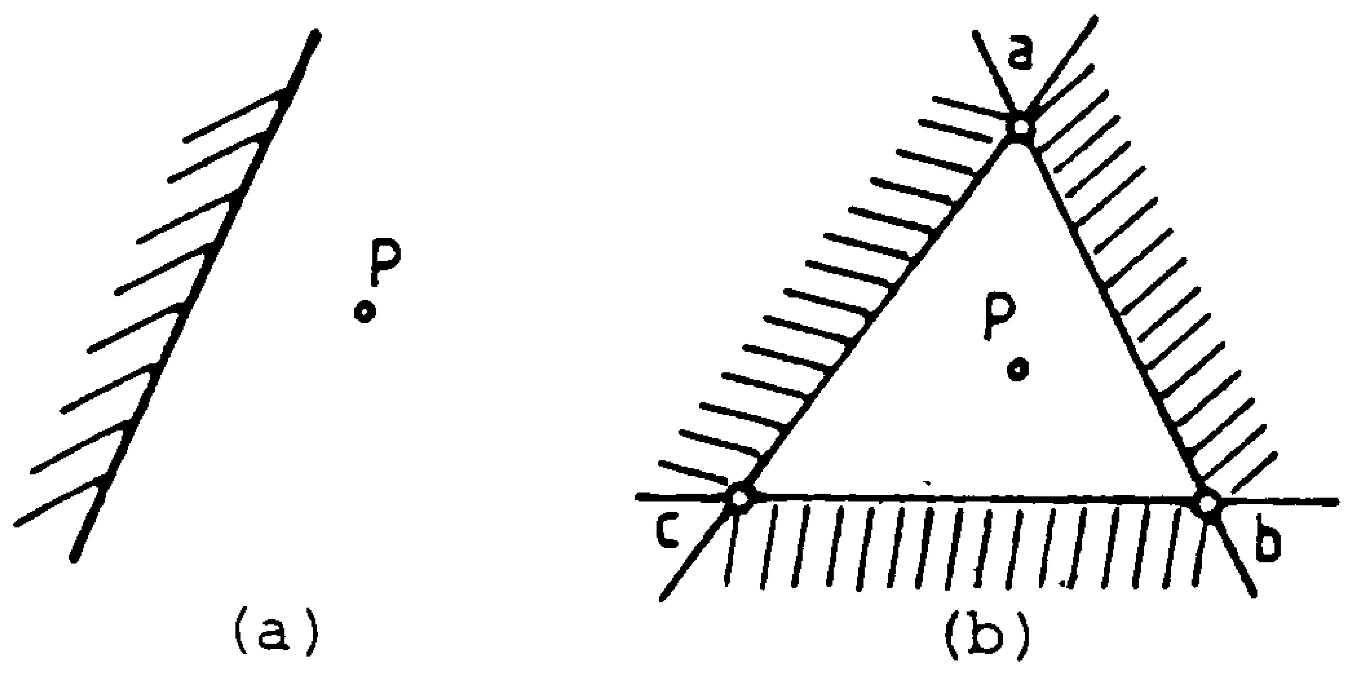


FIGURE 6.5

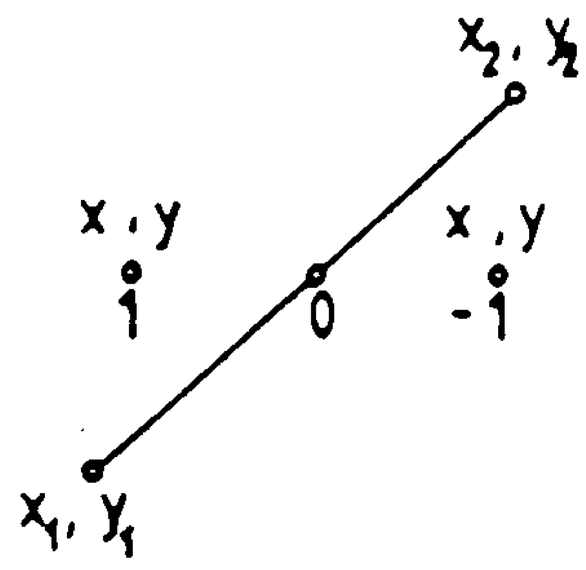


FIGURE 6.6

The equation of a line containing points $(x_1,y_1),(x_2,y_2)$ is:

$$\frac{y-y_1}{x-x_2} = \frac{y_2-y_1}{x_2-x_1} \quad \text{or} \quad (x-x_1)(y_2-y_1)-(x_2-x_1)(y-y_1) = 0 \qquad (6.3)$$

By defining a function, which returns 1,0 or -1 depending on whether a point is in one half-plane, on the line or in the other half plane, it is easy to determine on which half-plane the point lies (Fig. 6.6). This is achieved by comparing the values of the function for two different points. If the values are equal, the two points lie on the same half-plane. In the case of a point lying on the line, a further test is done in order to determine whether the point lies between the two vertices of the triangle side. In that case the point is considered as being 'inside' the triangle.

The containment of a point by a quadrilateral is achieved by testing whether the point is contained by one of the two triangles in which the quadrilateral can be divided. To divide a quadrilateral into two triangles is simple and is done by drawing one of the diagonals. However, if the quadrilateral is non-convex the choice of the diagonal is important (Fig. 6.7 c and d). The following algorithm [Harrington '83,b], determines the correct diagonal. First of all, the left-most vertex (the top-most would be as good) of the quadrilateral is determined and is given the number 1 in a clockwise order. The vertex

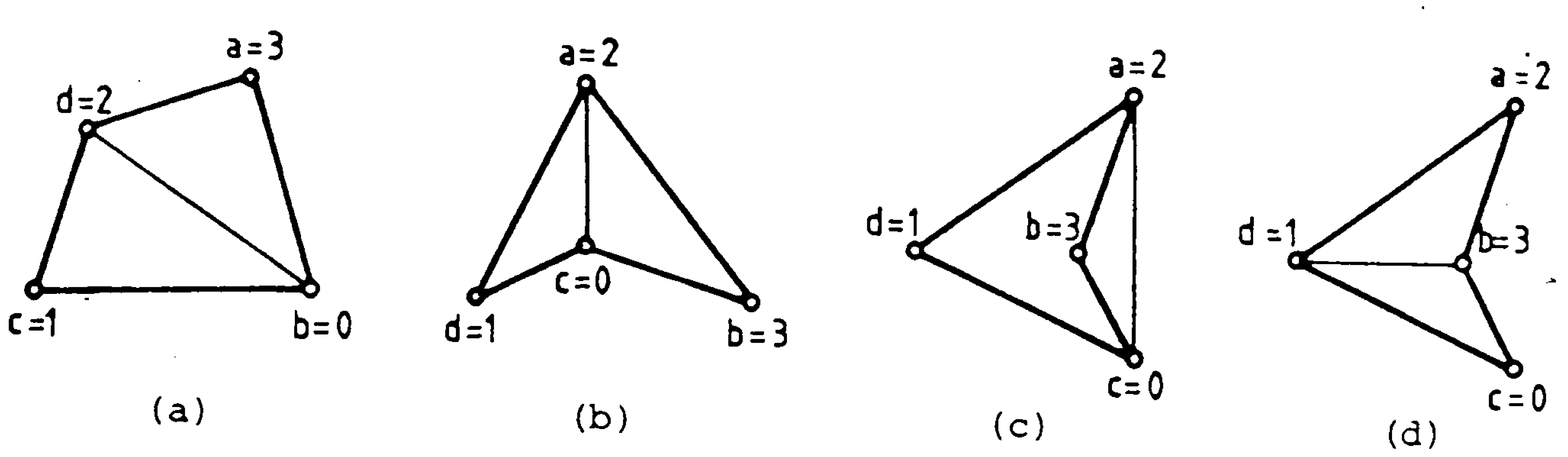


FIGURE 6.7

before it is no. 0 and the one after it, is no. 2. The remaining vertex is no. 3. The quadrilateral is divided into the triangles (012) and (230). If vertex no. 3 is not contained by the triangle (230) the division is correct (Fig. 6.7a and b). If vertex no. 3 is contained by the triangle (230) (Fig. 6.7c), then a new division into triangles (123) and (130) is considered (Fig. 6.7d).

Every *s_figure* of the original segmentation is tested, in order to determine whether it contains any of the vertices that do not belong to it. If such a vertex exists, the *s_figure* is invisible and it is characterised as a *possible figure (p_figure)*. This characterisation is done by means of a predicate called: *point_in_trn(a,b,c)* or *point_in_qul(a,b,c,d)* for triangles and quadrilaterals respectively.

6.3.2 Convexity of Quadrilaterals

A quadrilateral is *convex* if each of its sides extended in both directions, leaves the rest of the sides untouched (does not intersect them), (Fig. 6.8a). Otherwise, the quadrilateral is *non-convex* or *concave* (Fig. 6.8b). A non-convex quadrilateral contains an angle (e.g. b) > 180°. One of the ways to determine the convexity of a quadrilateral is to check whether all its angles are < 180°.

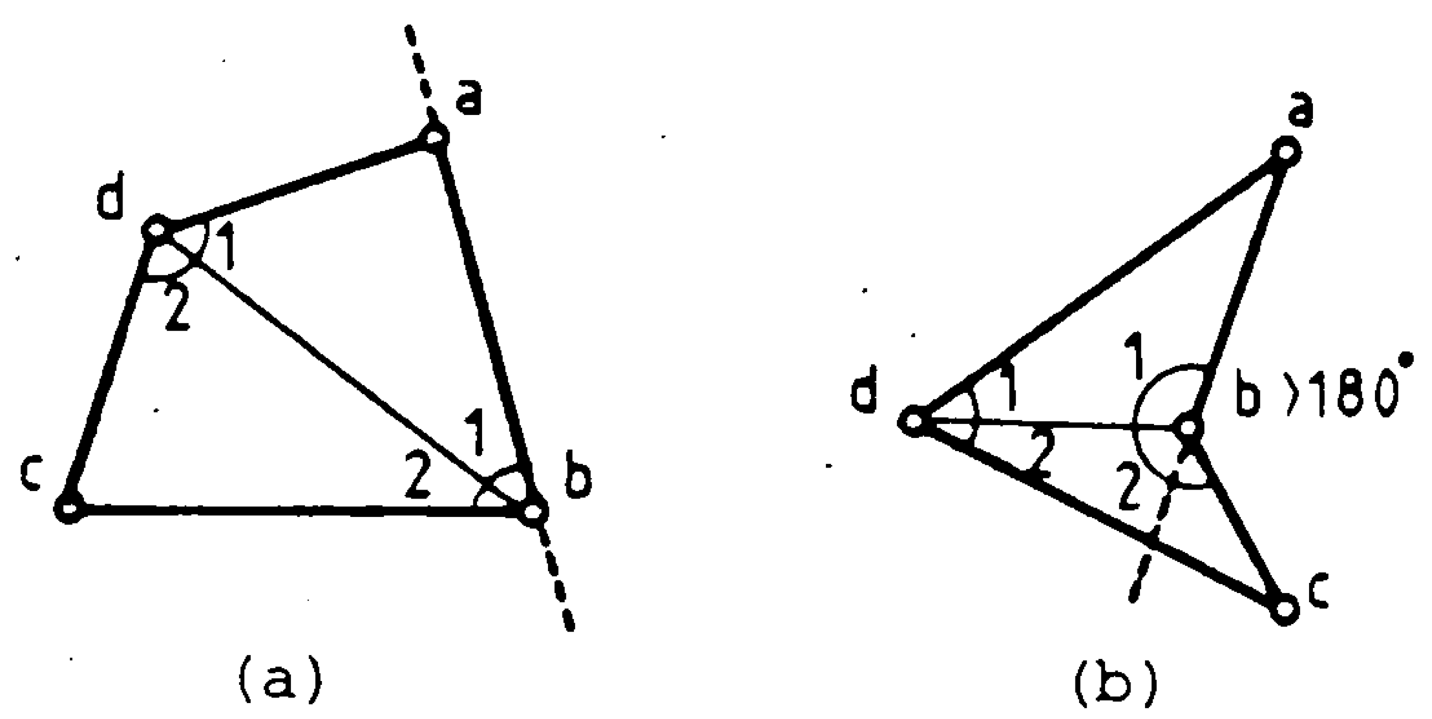


FIGURE 6.8

The angles of the vertices in a quadrilateral are calculated by splitting it first into triangles and calculating the angles of the vertices for every one of the two triangles (Fig. 6.8a). Care should be taken that the splitting of the quadrilateral be done according to §6.2.1. The angles of the triangles that belong to the same quadrilateral vertex are added together. The angles of a triangle are calculated by the following trigonometric and geometric formulae:

$$\cos A = \frac{b^2+c^2-a^2}{2bc} \quad \text{or} \quad \hat{A} = \frac{360°}{2\pi}\,\text{arc-cos}A \qquad (6.4)$$

$$\cos B = \frac{a^2+c^2-b^2}{2ac} \quad \text{or} \quad \hat{B} = \frac{360°}{2\pi}\,\text{arc-cos}B \qquad (6.5)$$

$$\hat{C} = 180° - \hat{A} - \hat{B}\ , \qquad (6.6)$$

where $\hat{A}$, $\hat{B}$, $\hat{C}$ are the angles of a triangle (ABC) in degrees (°) and a, b, c the lengths of the sides lying opposite to each angle respectively.

$$a = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2} \tag{6.7}$$

$$b = \sqrt{(x_0-x_2)^2+(y_0-y_2)^2} \tag{6.8}$$

$$c = \sqrt{(x_2-x_0)^2+(y_2-y_0)^2} \tag{6.9}$$

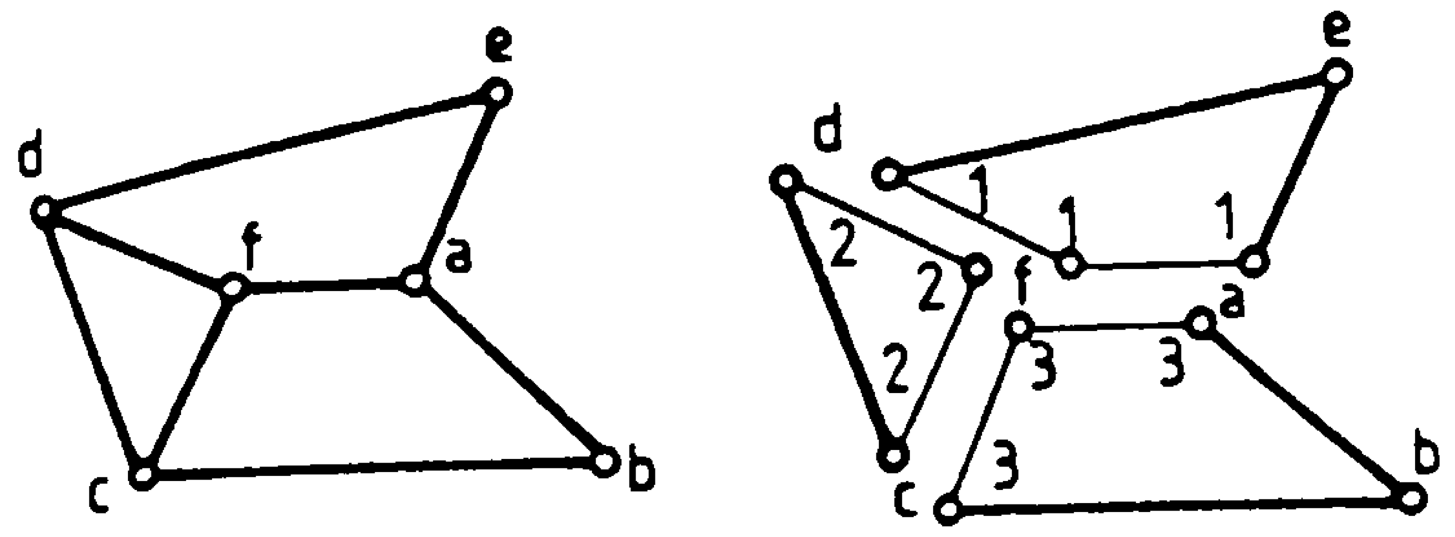
where (x_0,y_0), (x_1,y_1), (x_2,y_2) are the coordinates of the vertices A, B and C respectively.

For every non-convex quadrilateral a predicate: *non_convex_qul(a,b,c,d)* is created.

6.3.3 Convexity of Contour

A 2-D picture of a 3-D figure is confined by a polygon called a *contour*. If this is a non-convex polygon, a special predicate is created, in order to indicate this fact. The convexity of a contour is checked by the following algorithm (Fig. 6.9):

The angles of the faces, in which every picture is segmented, corresponding to the same vertices are added together. If the sum is equal to 360° (within a certain tolerance, depending on the accuracy with which the angles are calculated e.g. angle $\hat{f}$ in Figure 6.9), then the sum does not represent an angle of a contour vertex. If the sum is less than 180° $(\hat{b},\hat{c},\hat{d},\hat{e})$, then the vertex is convex. Otherwise the vertex is non-convex (e.g. $\hat{a}$), and this is indicated by the creation of predicate: *non_convex_contour_angle(a)*.



$\hat{a} = \hat{a}_1 + \hat{a}_3$ > 180° non-convex contour angle

$\hat{b}$

$\hat{c} = \hat{c}_2 + \hat{c}_3$

$\hat{d} = \hat{d}_1 + \hat{d}_2$ < 180° convex contour angles

$\hat{e}$

$\hat{f} = \hat{f}_1 + \hat{f}_2 + \hat{f}_3 = 360°$ no contour angle

FIGURE 6.9

The special feature predicates that may be created in this phase are stored in file <spec_feat>. This file is concatenated with the file <scene> and the new file acts as input to the 3-D recognizer.

The special-feature routine is called [link] and is depicted by the flowchart in Figure 6.10.

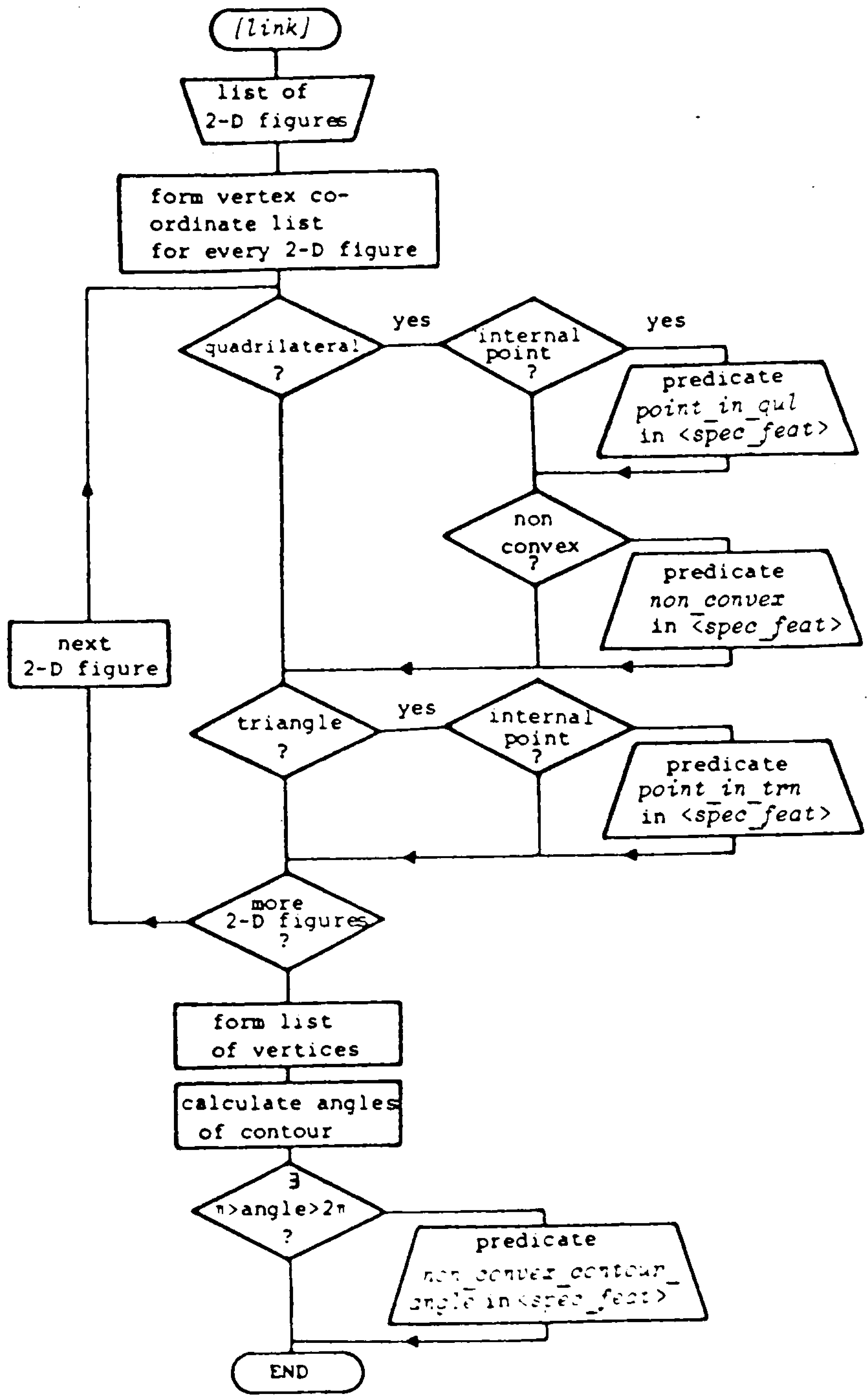


FIGURE 6.10

6.4 RECOGNITION

The recognition phase is divided into two main procedures: the *single-view recognition* and the *multiple-view recognition*. The input to both procedures is the same and consists of a set of *conn*-predicates which describe the structure of the figures, and a set of special features which carry information and crucial cues about the objects composing the scene. These two sets of data are provided by the previous procedure, the purpose of which is to set up the recognizer.

The single-view recognition relies mainly on the rules that were learnt in the single-view learning phase and several other auxiliary rules. Its output is a list of all the recognizable figures in the scene. The procedure also allows certain figures or only figures with a certain visibility of faces, to be looked for.

The multiple-view recognition relies on the definitions of the 3-D figures that were learnt in the multiple-view learning phase. It is obvious from the definitions themselves, that in order for the procedure to succeed more than a single view is required. This is achieved by procedure [assume] which tries to obtain information about hidden points and/or lines of the figures in the scene. The latter is based on the likelihood of some projected 3-D figure representations to be of one type rather than another and depends on assumptions made by the user. Both of these procedures are optional and they come out with a unique answer for each figure.

The recognition phase follows usually the learning phase but it could be also used independently. In this case, the rules and definitions of 1-D, 2-D and 3-D figures, must be supplied by the user.

6.4.1 Single-View Recognition

At the beginning the procedure looks for all the planes that make up the scene. These planes can be divided into two categories: those that can be directly seen, the *visible* planes, and those that are occluded by visible planes, the *possible* planes. By definition a visible plane is a triangle or a quadrilateral (convex or not), with no internal points. Likewise, a possible plane is a triangle or a quadrilateral containing internal points. The search for visible planes uses the definitions of: *trian(A,B,C)*, *c_quadril(A,B,C,D)* and *n_c_quadril(A,B,C,D)*, while that of possible planes uses the definitions of: *p_trian(A,B,C)*, *p_c_quadril(A,B,C,D)* and *p_nc_quadril(A, B,C,D)* (§5.3.2). The presence of every visible plane is marked by asserting a special predicate. These predicates are: *atrian(A,B,C)*, *aquadril(A,B,C,D)* and *nquadril(A,B,C,D)* for a triangle, a convex quadrilateral and a non-convex one respectively. Similar predicates are asserted for every possible plane. These are: *ptrian(A,B,C)*, *pquadril(A,B,C,D)* and *p_nquadril(A,B,C)*. These new predicates represent the faces of the figures in the scene and are the data on which the single-view recognition [Gabrielidis '82] is based. Another feature of the procedure is to mark the *conn*'s that constitute a visible face. This is achieved by decreasing their *face-counter* by 1. *Conn*'s with face-counter =0 are practically non-existent in the database, since they do not represent lines and thus, they can not be sides of a face. This modification of the *conn*-data is used, in order to make assumptions about the occluded faces. The set of rules that looks for planes is given below.

```
plane(convex_quadrilateral(A,B,C,D)):-
    c_quadril(A,B,C,D),
    mark_aqu(A,B,C,D).
plane(non_convex_quadrilateral(A,B,C,D)):-
    nc_quadril(A,B,C,D),
    mark_nqu(A,B,C,D).
plane(triangle(A,B,C)):-
    trian(A,B,C),
    mark_atr(A,B,C).

p_plane((convex_quadrilateral(A,B,C,D,),a1)):-
    p_c_quadril(A,B,C,D),
    mark_pqu(A,B,C,D).
p_plane((non_convex_quadrilateral(A,B,C,D),a1)):-
    p_nc_quadril(A,B,C,D),
    mark_pnqu(A,B,C,D).
p_plane((triangle(A,B,C),a1)):-
    p_trian(A,B,C),
    mark_ptr(A,B,C).
```

The rules consist of the definition of a 2-D figure and a predicate, the function of which is to assert the new plane-predicates. The definitions of the 2-D figures are the basic part of the rule and are learnt in the learning-phase. These perform the actual recognition. The rest is auxiliary and is responsible for minor functions depending chiefly on the structure of language (e.g. prevent infinite recycling loops, etc.). It also converts the data from 1-D to 2-D predicates. Term *a1* that appears in the argument list of the *p_plane*-predicates, is the type of assumption that permits the possible plane to be an actual one. In this case *a1* assumes that no hidden lines or points are behind the above possible plane (see also §6.4.2).

The procedure looks next for recognizable 3-D figures. The latter consist of visible (and sometimes invisible) faces connected in a specific way. The recognizable 3-D figures are those learnt in the single-view-learning phase and are: a *truncated pyramid* or *box*, a *prism*, a *pyramid* or *pyram* and a *tetrahedron* or *tetra*. For every one of these figures there is a number of definitions (see §5.3.3), depending on the number of different possible ways in which the figure can be viewed. It is obvious that a single view of the figure is not enough for a positive answer and therefore all of the definitions used refer to possible 3-D figures. In other words, the several alternatives of the rules correspond to alternative views of the above 3-D figures. In the future such a view will be called a *possible figure* and will be abbreviated by *p_figure* which is also the name used for the head of the rule. The argument list of the *p_figures* contains three entries. The name of the *p_figure*, an integer called *face-visibility* and an atom called *assumption-code number*. Face-visibility is defined as the percentage of the number of visible faces of a *p_figure* to the number of its actual faces:

$$\text{face-visibility} = \frac{\text{no. of visible faces}}{\text{no. of actual faces}} \times 100.$$

The assumption-code number indicates which (of a set) assumptions should be made, in order for the name of the *p_figure* to be correct.

The body of the rule consists of the disjunction of an *a_figure*-predicate and the main body. The main body is a conjunction of the main definition of the *p_figure*, a *not(non_contour_free)* and a *mark*-predicate*. The rule functions in the following way: The procedure examines the database, looking for *a_plane* and *p_plane*-predicates that satisfy the rule. If a successful combination is found, the *a_plane* and *p_plane* predicates that satisfy it are retracted from the database and a new predicate *a_figure* replaces them. This is done by the *mark*-

The several versions of mark-predicates are contained in Appendix 2.

predicate. The logic behind this substitution of 2-D figure data by 3-D figure data is to speed up the procedure in its search for alternative *p_figures*. This is performed by the first part of the disjunction. The definition of the *p_figure* is learnt by the single-view learner and is the main part of the rule. The data replacement is done by the *mark*-predicate. The *non_contour_free* predicate is defined as:

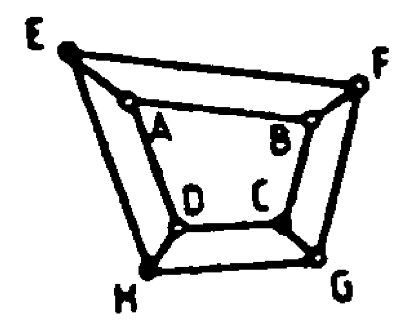
non_contour_free(List):-line(X,Y),(member(X,List),
not(member(Y,List))).

where *List* is the list of vertices of each *p_figure*. Its negation makes sure that the current *p_figure* is not part of another figure, according to the initial assumption a) of §6.1. It also prevents 3-D figures containing non-recognizable 2-D faces from being split into recognizable parts. The last two predicates constitute the auxiliary part of the rule.

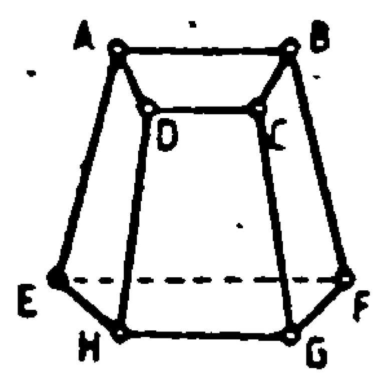
The rules are applied in order of decreasing complexity. If two *p_figure* definitions consist of the same type of faces (i.e. all triangles or all quadrilaterals) then the one with the greater number of faces is tried first. The same principle is followed for *p_figure*'s consisting of faces of different types (i.e. triangles and quadrilaterals). In the case of equal numbers of faces, those with greater number of quadrilaterals are tried first. Finally, *p_figure*'s consisting of faces with a greater number of vertices (quadrilaterals) have priority over *p_figure*'s consisting of faces with a smaller number of vertices (triangles).

Every recognizable 3-D figure consists of a number of *p_figure* definitions corresponding to its different possible views. If a certain view can be interpreted as two or more different 3-D figures, then these 3-D figures 'share' the same *p_figure* definition. In this case the procedure inserts the *a_figure* predicate, that corresponds to the 3-D figure with the highest priority. The set of rules that perform the single-view-recognition is given below and Figure 6.11 illustrates the corresponding views.

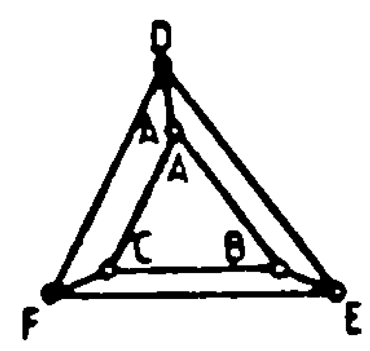
```
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),86,a1)):-
   a_box1(A,B,C,D,E,F,G,H);
   (p_box1(A,B,C,D,E,F,G,H),
   VrtxList=[A,B,C,D,E,F,G,H],
   not(non_contour_free(VrtxList)),
   mark_bx1(A,B,C,D,E,F,G,H)).
```



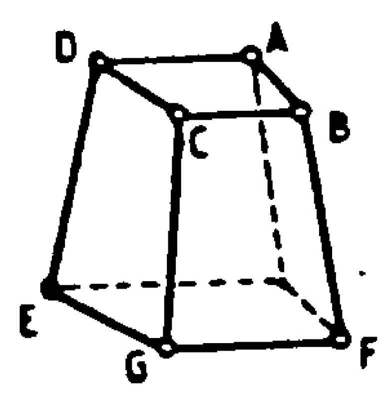

```
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),67,a3)):-
   a_box2(A,B,C,D,E,F,G,H);
   (p_box2(A,B,C,D,E,F,G,H),
   VrtxList=[A,B,C,D,E,F,G,H],
   not(non_contour_free(VrtxList)),
   mark_bx2(A,B,C,D,E,F,G,H)).
```



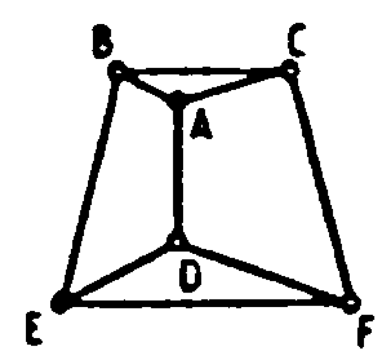

```
p_figure((prism(A,B,C,D,E,F),80,a1)):-
   a_prism1(A,B,C,D,E,F);
   (p_prism1(A,B,C,D,E,F),
   VrtxList=[A,B,C,D,E,F],
   not(non_contour_free(VrtxList)),
   mark_pr1(A,B,C,D,E,F)).
```




```
p_figure((truncated-pyramid(A,B,C,D,E,F,G,H),50,a4)):-
   a_box 3(A,B,C,D,E,F,G);
   (p_box 3(A,B,C,D,E,F,G),
   VrtxList=[A,B,C,D,E,F,G],
   not(non_contour_free(VrtxList)),
   mark_bx 3(A,B,C,D,E,F,G)).
```




```
p_figure((prism(A,B,C,D,E,F),80,a1)):-
   a_prism1a(A,B,C,D,E,F);
   (p_prism1a(A,B,C,D,E,F),
   VrtxList=[A,B,C,D,E,F],
   not(non_contour_free(VrtxList)),
   mark_pr1a(A,B,C,D,E,F)).
```




```
p_figure((prism(A,B,C,D,E,F),60,a3)):-
   a_prism2(A,B,C,D,E,F);
   (p_prism2(A,B,C,D,E,F),
   VrtxList=[A,B,C,D,E,F],
   not(non_contour_free(VrtxList)),
   mark_pr2(A,B,C,D,E,F)).
```

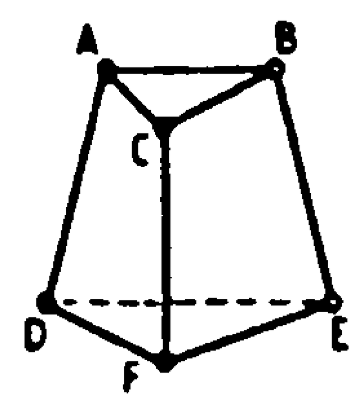
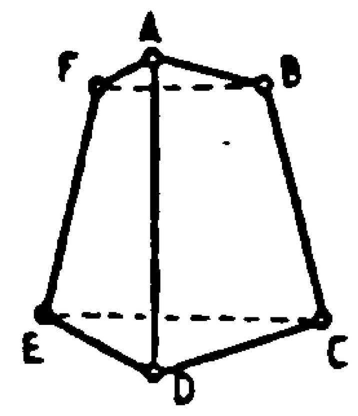
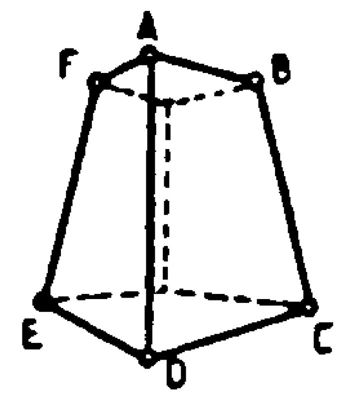


FIGURE 6.11

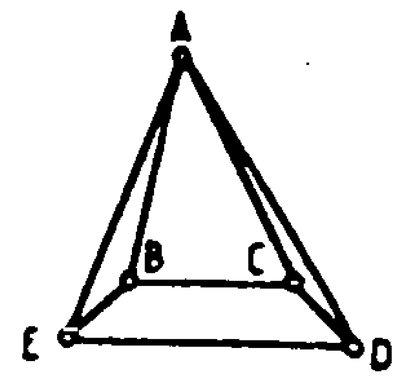
```
p_figure((prism(A,B,C,D,E,F),60,a2)):-
   a_prism3(A,B,C,D,E,F);
   (p_prism3(A,B,C,D,E,F),
   VrtxList=[A,B,C,D,E,F),
   not(non_contour_free(VrtxList)),
   mark_pr3(A,B,C,D,E,F)).
```



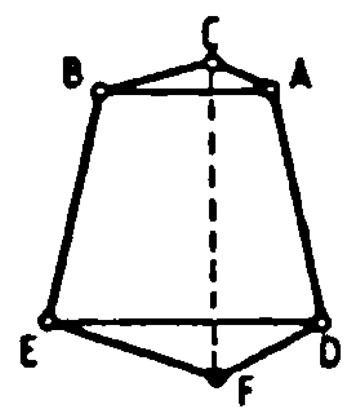

```
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),33,a3)):-
   a_prism3(A,B,C,D,E,F);
   (p_prism3(A,B,C,D,E,F);
   VrtxList=[A,B,C,D,E,F],
   not(non_contour_free(VrtxList)),
   mark_pr3(A,B,C,D,E,F)).
```



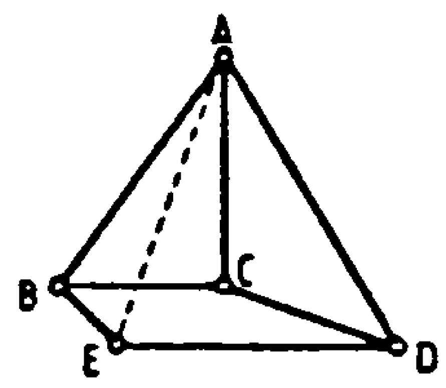

```
p_figure((pyramid(A,B,C,D,E),80,a1)):-
   a_pyram1(A,B,C,D,E);
   (p_pyram1(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py1(A,B,C,D,E)).
```



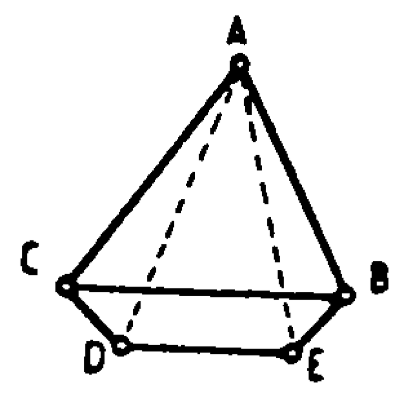

```
p_figure((triangular_prism(A,B,C,D,E,F),40,a2)):-
   a_prism2a(A,B,C,D,E,F);
   (p_prism2a(A,B,C,D,E,F);
   VrtxList=[A,B,C,D,E,F],
   not(non_contour_free(VrtxList)),
   mark_pr2a(A,B,C,D,E,F)).
```



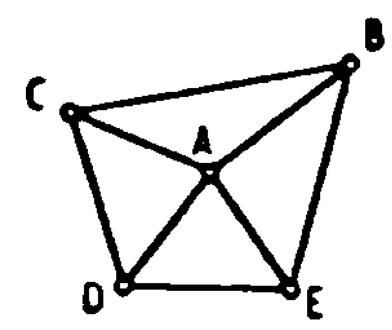

```
p_figure((pyramid(A,B,C,D,E),60,a2)):-
   a_pyram2(A,B,C,D,E);
   (p_pyram2(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py2(A,B,C,D,E)).
```



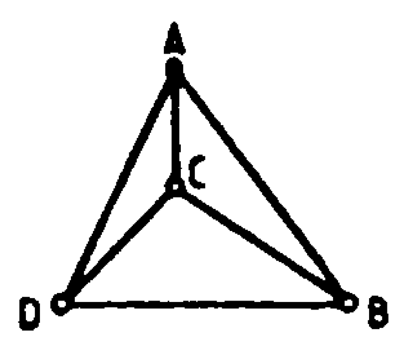

```
p_figure((pyramid(A,B,C,D,E),40,a2)):-
   a_pyram3(A,B,C,D,E);
   (p_pyram3(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py3(A,B,C,D,E)).
```




```
p_figure((pyramid(A,B,C,D,E),80,a1)):-
   a_pyram1a(A,B,C,D,E);
   (p_pyram1a(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py1a(A,B,C,D,E)).
```




```
p_figure((tetrahedron(A,B,C,D),75,a1):-
   a_tetra1(A,B,C,D);
   (p_tetra1(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_te1(A,B,C,D)).
```




```
p_figure((pyramid(A,B,C,D,E),60,a3)):-
   a_pyram2a(A,B,C,D,E);
   (p_pyram2a(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py2a(A,B,C,D,E)).
```

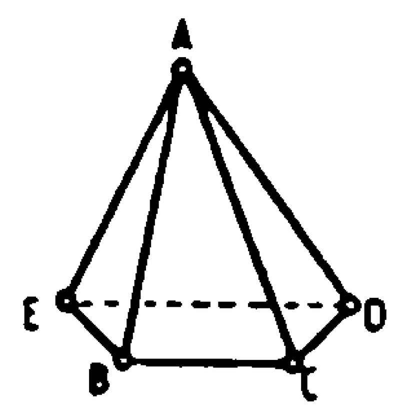
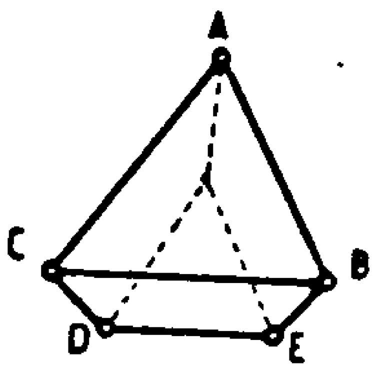
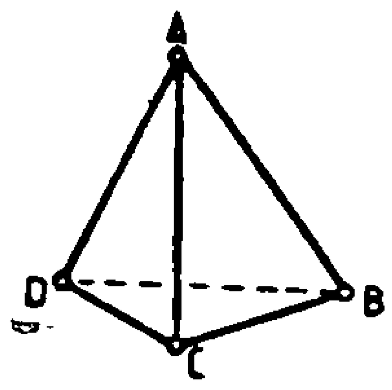


FIGURE 6.11 (cont.)

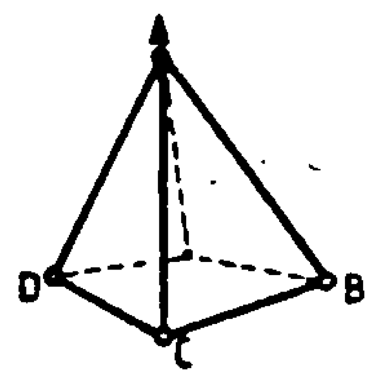
```
p_figure((prism(A,B,C,D,E,F),33, a4)):-
   a_pyram3(A,B,C,D,E);
   (p_pyram3(A,B,C,D,E),
   VrtxList=[A,B,C,D,E],
   not(non_contour_free(VrtxList)),
   mark_py3(A,B,C,D,E)).
```



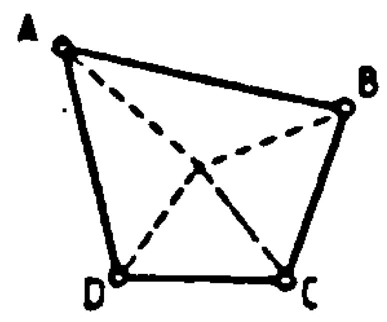

```
p_figure((tetrahedron(A,B,C,D),50,a2)):-
   a_tetra2(A,B,C,D);
   (p_tetra2(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_te2(A,B,C,D)).
```



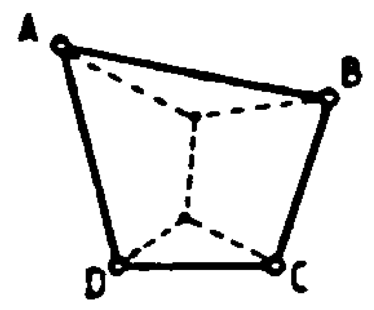

```
p_figure((pyramid(A,B,C,D,E),20,a4)):-
   a_tetra2(A,B,C,D);
   (p_tetra2(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_te2(A,B,C,D)).
```



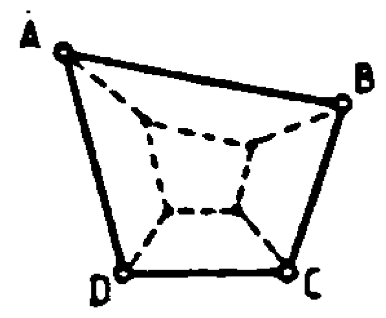

```
p_figure((pyramid(A,B,C,D,E),25,a5)):-
   a_pyram4(A,B,C,D);
   (p_pyram4(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_py4(A,B,C,D)).
```



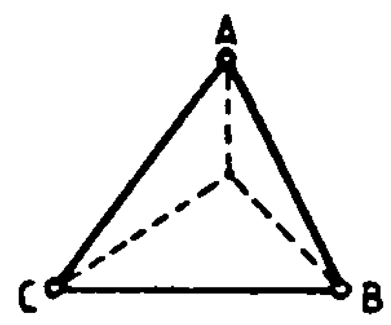

```
p_figure((prism(A,B,C,D,E,F),20,a6)):-
   a_pyram4(A,B,C,D);
   (p_pyram4(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_py4(A,B,C,D)).
```



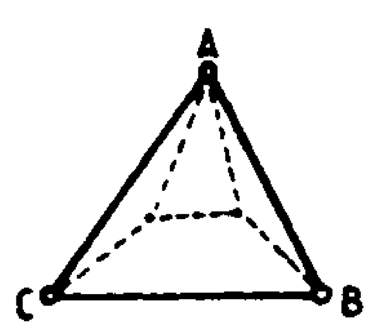

```
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),17,a7)):-
   a_pyram4(A,B,C,D);
   (p_pyram(A,B,C,D),
   VrtxList=[A,B,C,D],
   not(non_contour_free(VrtxList)),
   mark_py4(A,B,C,D)).
```




```
p_figure((tetrahedron(A,B,C,D),25,a5)):-
   a_tetra3(A,B,C);
   (p_tetra3(A,B,D),
   VrtxList=[A,B,D],
   not(non contour_free(VrtxList)),
   mark_te3(A,B,C)).
```




```
p_figure((pyramid(A,B,C,D,E),20,a6)):-
   a_tetra3(A,B,D);
   (p_tetra3(A,B,D),
   VrtxList=[A,B,C]
   not(non_contour_free(VrtxList)),
   mark_te3(A,B,D)).
```




```
p_figure((prism(A,B,C,D,E,F),17,a7)):-
   a_tetra3(A,B,C);
   (p_tetra3(A,B,C),
   VrtxList=[A,B,D],
   not(non_contour_free(VrtxList)),
   mark_te3(A,B,D)).
```

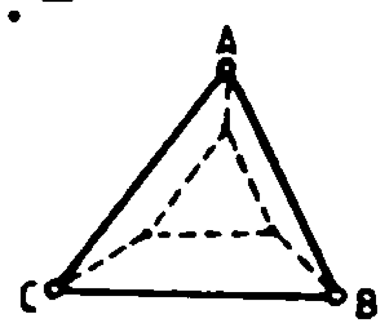


FIGURE 6.11 (cont).

6.4.2 Multiple-View Recognition

The main parts of the multi-view recognizer rules are the definitions that were learnt by the multi-view learner. The definitions cover the same group of 3-D figures, that are recognizable by the previous procedure. They are also unique and precise, because they have been constructed from more than one different view of the 3-D figure they represent. The head of the rules contains only one argument, namely the name of a 3-D figure. The body of the rules consists of a 3-D figure definition as its main part and a *mark*-predicate as its auxiliary part. The definition consists of a set of 2-D figures which are the faces of the 3-D figure it defines. The multiple-view recognizer uses *conn*-predicates as its data. Every *conn*-predicate represents an edge of the defined 3-D figure and has the form: *conn(x,y,2)*, which means that: vertices x and y are connected with an edge that belongs to two faces. The function of the *mark*-predicate is to decrease the face-counter by 1, every time it meets a face containing the above edge. This means, that if two faces that contain an edge are found, the face-counter of its corresponding *conn*-predicate will take the value 0 and thus will be virtually removed from the database. When all *conn*'s are removed, the procedure succeeds. If, at the end of the procedure, there are *conn*'s with face-counters $\neq 0$, it means that there are faces of the 3-D figure that are invisible. These 'non-zero' *conn*-predicates are the cues to what is missing. The order of the rules is not important, since the definitions are unique. The rules of the multiple-view recognizer are given below.

```
m_figure(truncated_pyramid(A,B,C,D,E,F,G,H)):-
    box(A,B,C,D,E,F,G,H),
    mark_box(A,B,C,D,E,F,G,H).
```

```
m_figure(prism(A,B,C,D,E,F)):-
    prism(A,B,C,D,E,F),
    mark_prism(A,B,C,D,E,F).
m_figure(pyramid(A,B,C,D,E)):-
    pyram(A,B,C,D,E),
    mark_pyram(A,B,C,D,E).
m_figure(tetrahedron(A,B,C,D)):-
    tetra(A,B,C,D),
    mark_tetra(A,B,C,D).
```

An example is given at the end of §6.5.

6.4.3 Assumptions

The multi-view recognizer is used in combination with a set of assumptions, that attempt to add to the database *conn*-predicates of invisible faces. At the beginning of the single-view recognition, a routine looks for all visible and possible planes of a 3-D figure. The procedure marks all the *conn*'s that belong to a visible face, by decreasing their face-counter by 1. It is obvious that *conn*'s that belong to two visible faces will have a face-counter =0 (or *c_line*'s) and *conn*'s that belong partly to a visible and partly to an invisible face will have a face-counter =1 (or *a_line*'s), after the end of the routine. The following assumptions make use of the modified database, in order to deduce information about hidden points, lines or faces. The definitions of the recognizable 3-D figures can be divided into three major categories. In the first one belong the definitions which contain a *p_plane*, which means that there is *only one face missing* from the corresponding multiple-view definition. The second category covers all the definitions with *more than one missing face*. Finally, in the third category belong definitions where *only one face is visible*.

The first category deals with the assumption indicated by code:

a1: no hidden points or lines behind possible face

This assumption (as the comment suggests) eliminates practically all the elements that could prevent the *p_face* from being a visible one. For example the 3-D figure shown in Figure 6.12a could be any of the 3-D figures depicted by 6.12b,c and d, from which only d is a recognizable one. What assumption *a1* does, is to eliminate all other cases. Now the only thing that remains to be done, is to remove from

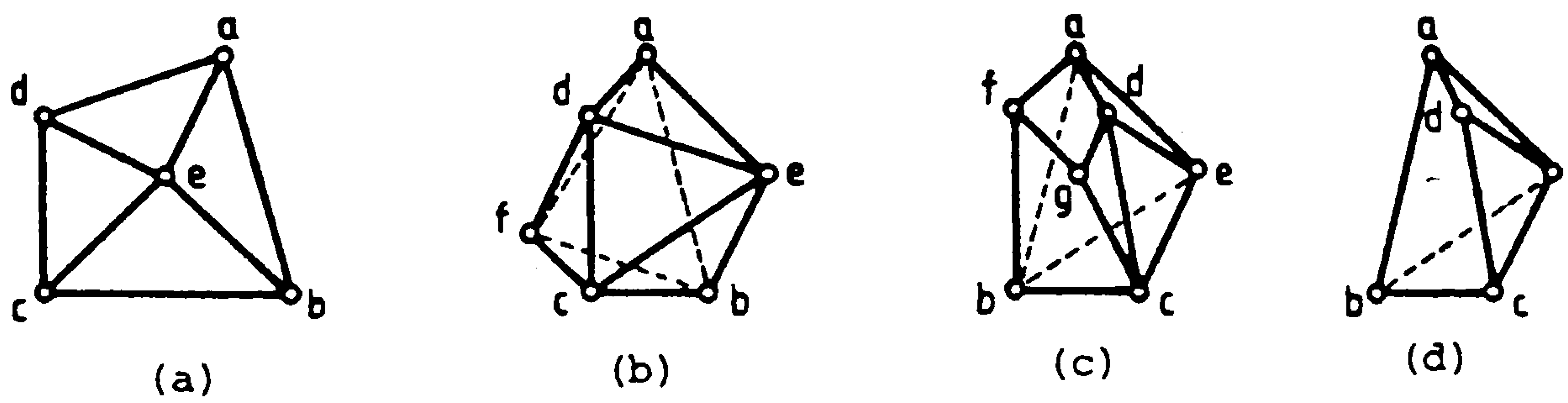


FIGURE 6.12

the database the element that is evidence that the face is invisible. This is done by retracting the *point_in*-predicate. The multiple-view recognizer applied on the new database will succeed in finding the name of the 3-D figure.

The second category covers the following three cases:

a2: hidden (triangle) line

a3: hidden (quadrilateral) line

a4: hidden (quadrilateral) point

These three assumptions are based on a common main idea. They look through the modified data, in order to find important cues about invisible faces and their hidden lines. Such cues are primarily *a_line*'s and *c_line*'s in the second place.

The first assumption *a2*, looks for occluded triangular faces with one side missing. It starts by seeking two *a_line*'s with one common end. It then makes sure that the non-connected ends of the *a_line*'s are really 'open' (i.e. no *a_*, *b_*, or *c_line* connecting them) and that no other line stems from the common vertex. If all the above conditions are satisfied, then the two open ends of the *a_line*'s are connected with a *b_line* (*conn(x,y,2)*) and the three lines form a triangle. Case *a2* is demonstrated in Figure 6.13. Planes (*bac*) and (*cad*) are two visible faces with a common edge (*ac*). After the first plane search

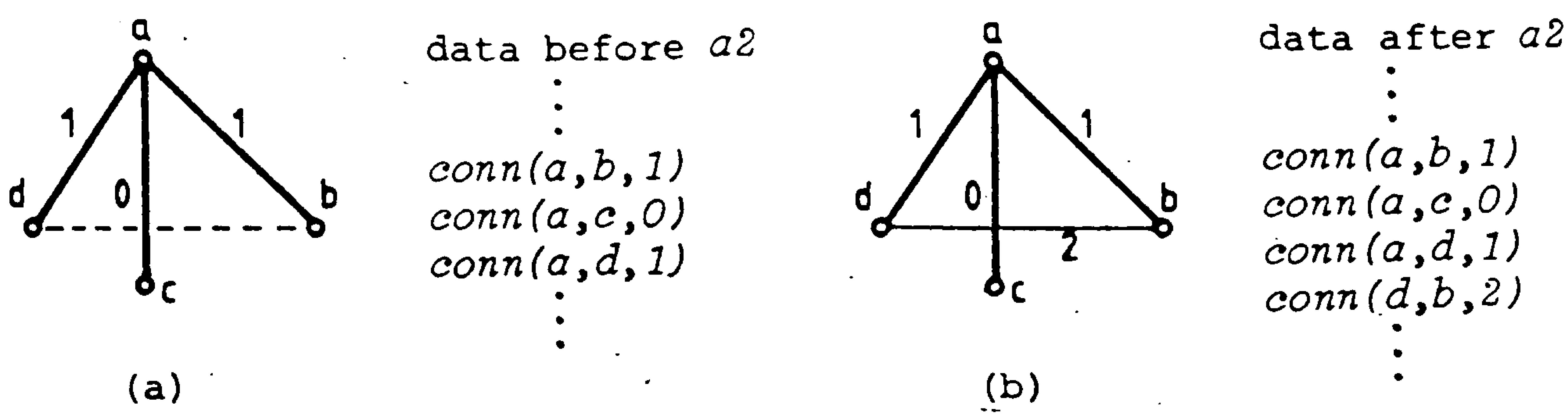


FIGURE 6.13

the three lines will look like: *conn(a,b,1)*, *conn(a,c,0)* and *conn(d,a,1)*. Thus, lines (*ab*) and (*ad*) are good candidates for assumption *a2*. Vertices *d* and *b* are connected by means of asserting a *conn(d,b,2)* predicate to the database.

Assumption *a3* works on a very similar basis. This time it looks for three *a_lines* forming a quadrilateral with an open fourth side. If none of the diagonals exists, then the open vertices are connected with a *b_line*. Figure 6.14 shows the function of this case. Assumption *a3* is made whenever assumption *a2* does not apply. In Figure 6.14c for example lines *c_line(a,d)* and *c_line(a,c)* prevent lines *a_line(a,e)*, *a_line(e,d)* and *a_line(a,b)*, *a_line(b,c)* respectively from satisfying *a2*. Lines *a_line(c,d)* and *a_line(d,c)*, for example, satisfy *a2* but lead to an unrecognizable 3-D figure (Fig. 6.14d).

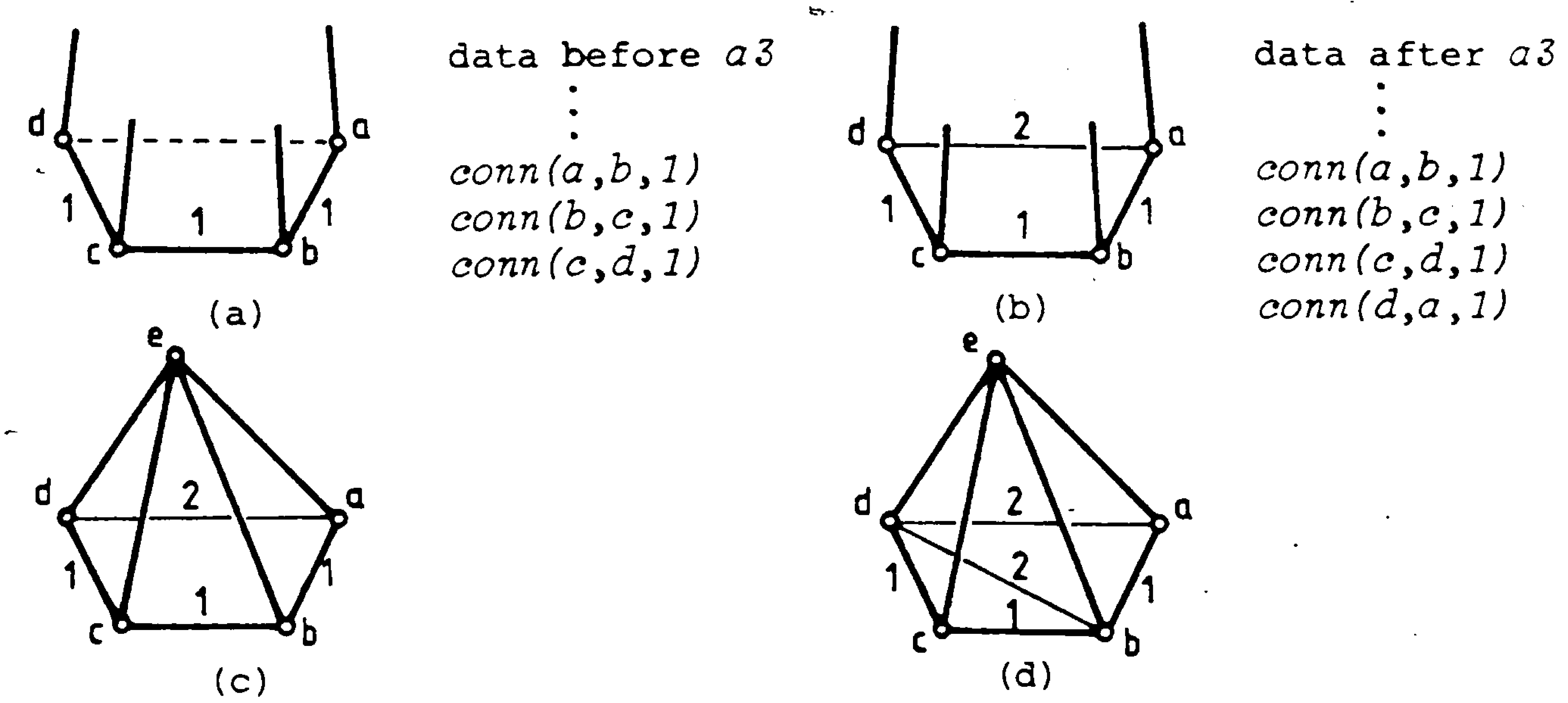


FIGURE 6.14

Assumption *a4* can be seen as an extension of *a2*. It looks again for two connected *a_line*'s with the other end open. This time however, instead of connecting the two open ends with a *b_line*, it introduces a third point *hp* (hidden point) and connects it to the two open ends of the *a_line*'s, so that a quadrilateral is formed. Figure 6.15 illustrates case *a4* and shows how different results can be obtained if different assumptions are made. The 3-D figure in 6.15d is an impossible case because faces (*abcd*) and (*bcde*) are actually coplanar since they contain three common vertices *b*, *c* and *d*.

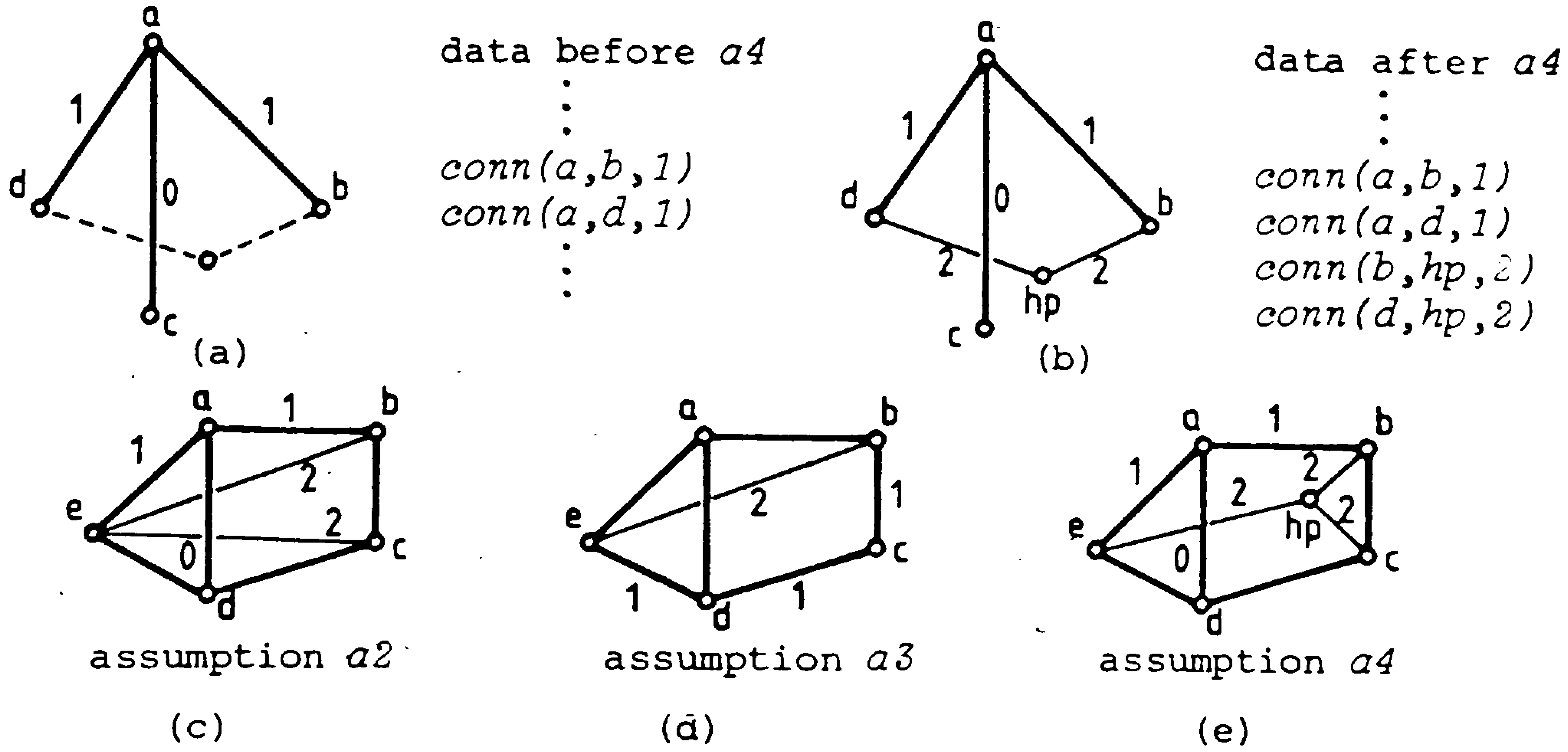


FIGURE 6.15

The third category deals with the following cases:

a5: entirely hidden point

a6: entirely hidden line

a7: entirely hidden face

The hidden lines covered by the last category are characterized by the fact that they are directly connected to existing *a_line*'s of a certain 3-D figure. In other words, the existence of the lines is, more or less, dictated by the structure of the 3-D figure. This last category contains cases, where the procedure inserts hidden lines that are not directly connected to the original 3-D figure. These elements will be called *entirely hidden elements*, because no part of them is visible.

In the first case, *a5*, every *a_line* of the face is thought of, as side of a potential triangular hidden face. Another way of looking at it, is to place a hidden point in the middle of the original face and connect it to all the vertices of the face. This case is similar to *a4*, with the only difference of looking for single *a_line*'s instead of pairs. Figure 6.16 shows case *a5*.

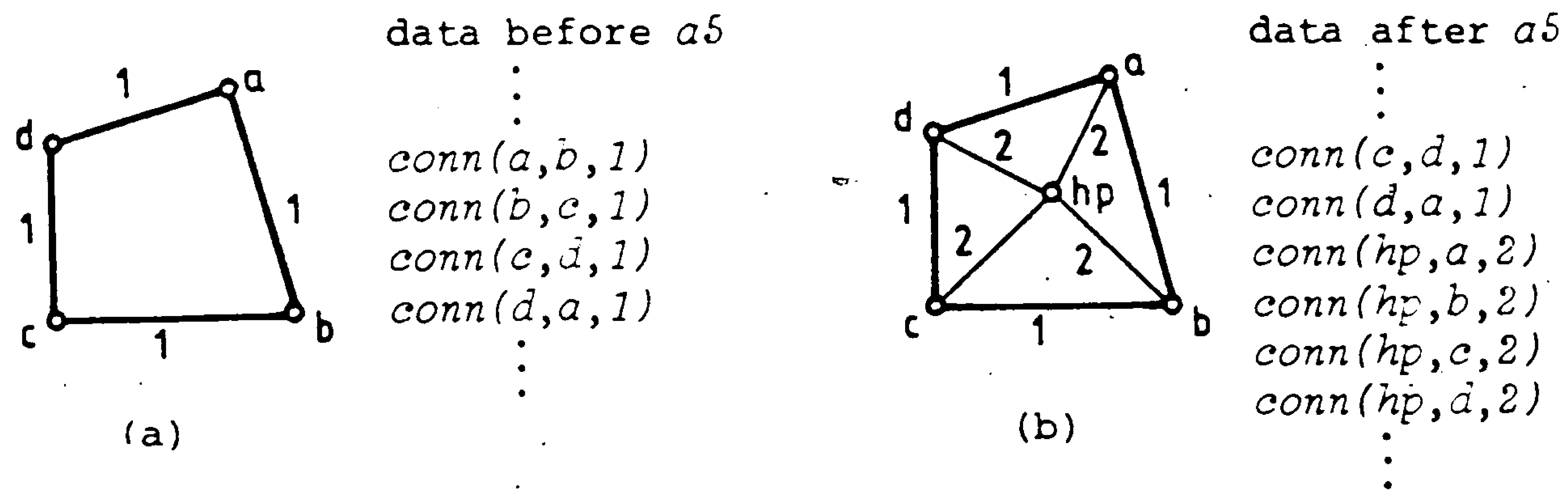


FIGURE 6.16

The second case *a6*, is an extension of *a5*, in the sense that two hidden points connected with a *b_line* (i.e. a hidden line) are inserted. This case is more specific than the previous ones, with respect to the way the hidden line is connected to the vertices of the visible face. Thus, if the visible face is a triangle the inserted *b_line*'s form a pyramid, and if the visible face is a quadrilateral the result of the inserted *b_line*'s is a prism (Fig. 6.17).

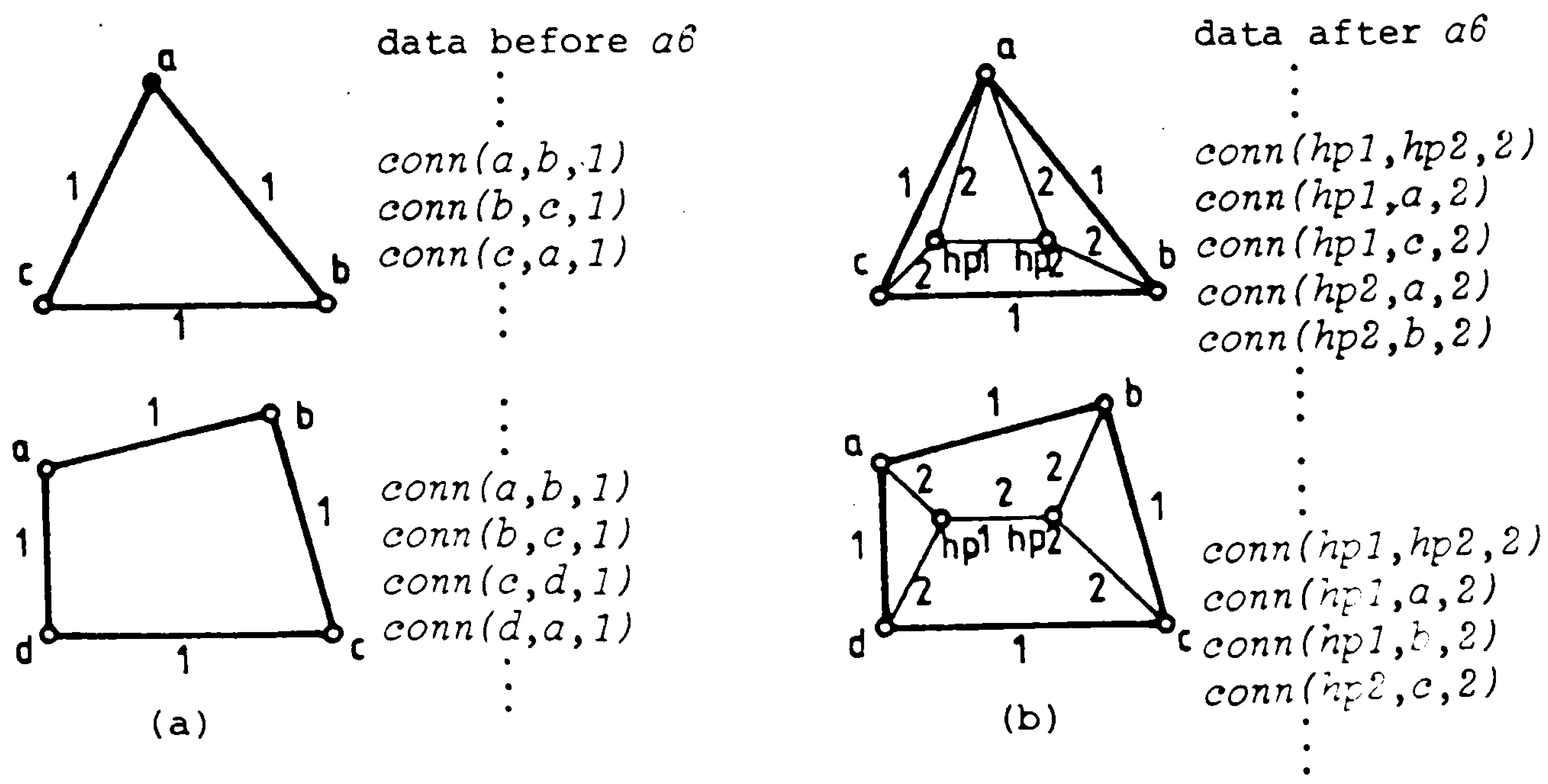
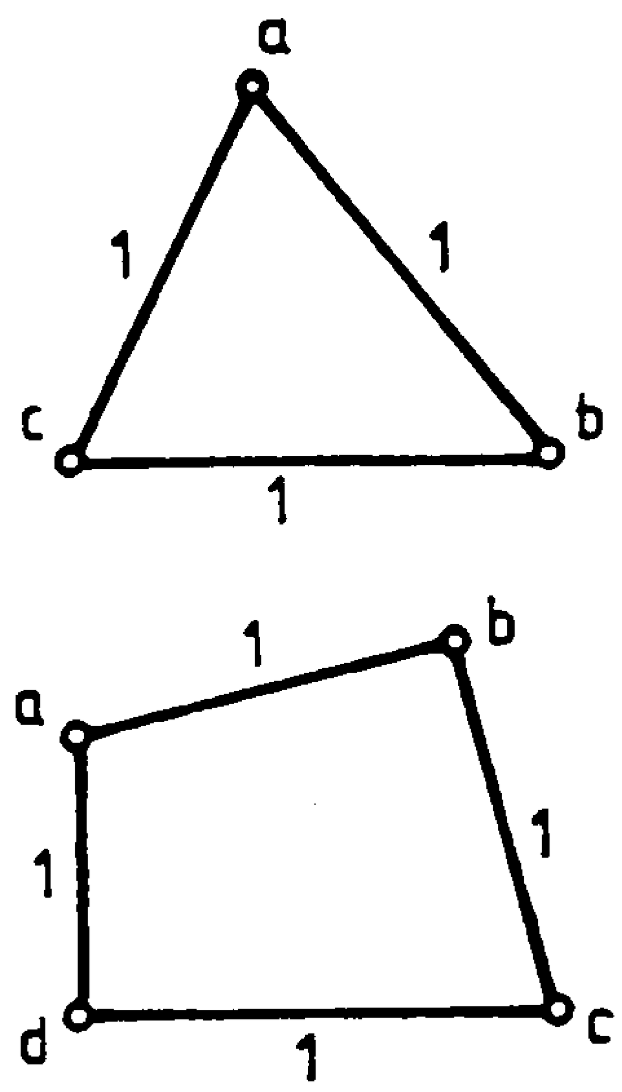


FIGURE 6.17

The third case *a7* is also specific and it considers a hidden triangle behind a triangular visible face,and a hidden quadrilateral if the visible face is a quadrilateral. It covers the cases of a truncated pyramid and that of a triangular prism respectively (Fig.6.18).



data before *a7*

⋮

conn(a,b,1)
conn(b,c,1)
conn(c,a,1)

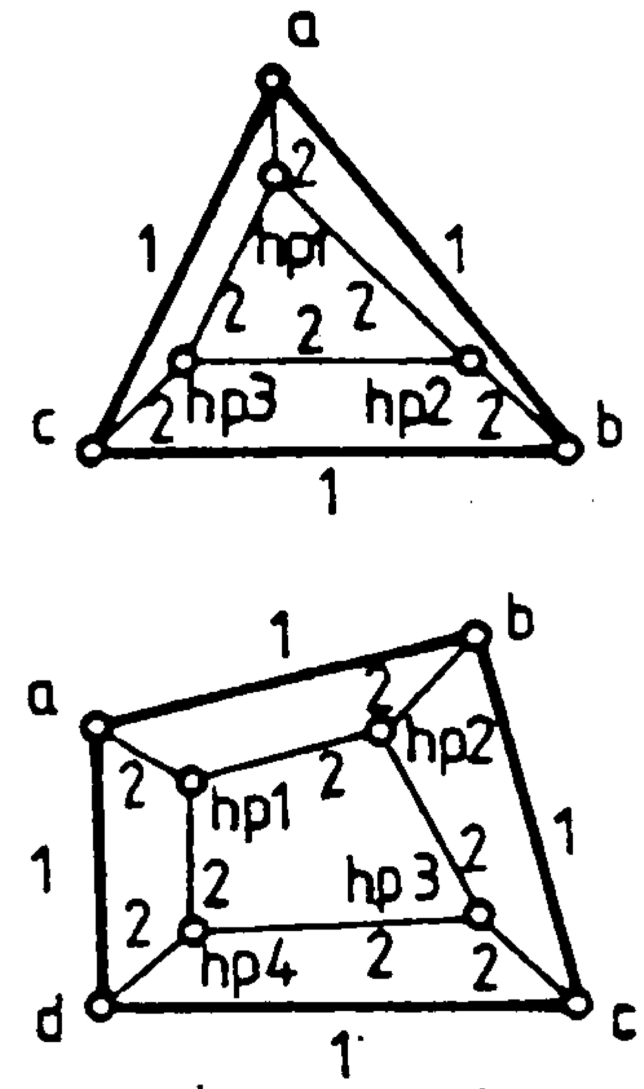
⋮

⋮

conn(a,b,1)
conn(b,c,1)
conn(c,d,1)
conn(d,a,1)

⋮

(a)



data after *a7*

⋮

conn(hp1,hp2,2)
conn(hp2,hp3,2)
conn(hp3,hp1,2)
conn(hp1,a,2)
conn(hp2,b,2)
conn(hp3,c,2)

⋮

conn(hp1,hp2,2)
conn(hp2,hp3,2)
conn(hp3,hp4,2)
conn(hp4,hp1,2)
conn(hp1,a,2)
conn(hp2,b,2)
conn(hp3,c,2)
conn(hp4,d,2)

(b)

FIGURE 6.18

The above described assumptions are subprocedures of a procedure called [assume] which is part of the single-view recognizer. The function of this procedure is the following:

If the user decides to proceed to an assumption, she/he must provide the system with the desirable assumption-code. This assumption-code can be the one suggested by the single-view recognizer or anyone of the list. In the second case, no successful result by the multiple-view recognizer is guaranteed. She/he must also specify the particular *p_figure*, for which the assumption will be made by inserting the list of its vertices. This is necessary, because [assume] works with one figure at a time. The procedure carries out the required assumption for the specified figure, and forms a list of the hidden lines (if any at all) that have been added to the database. The *conn*'s that represent hidden lines are saved in special file called <hidden>. Finally, the list of the hidden lines is printed out.

In the case of assumption *a4*, procedure [assume] performs an extra test in order to cover a special case which is described below and is illustrated by Figure 6.19.

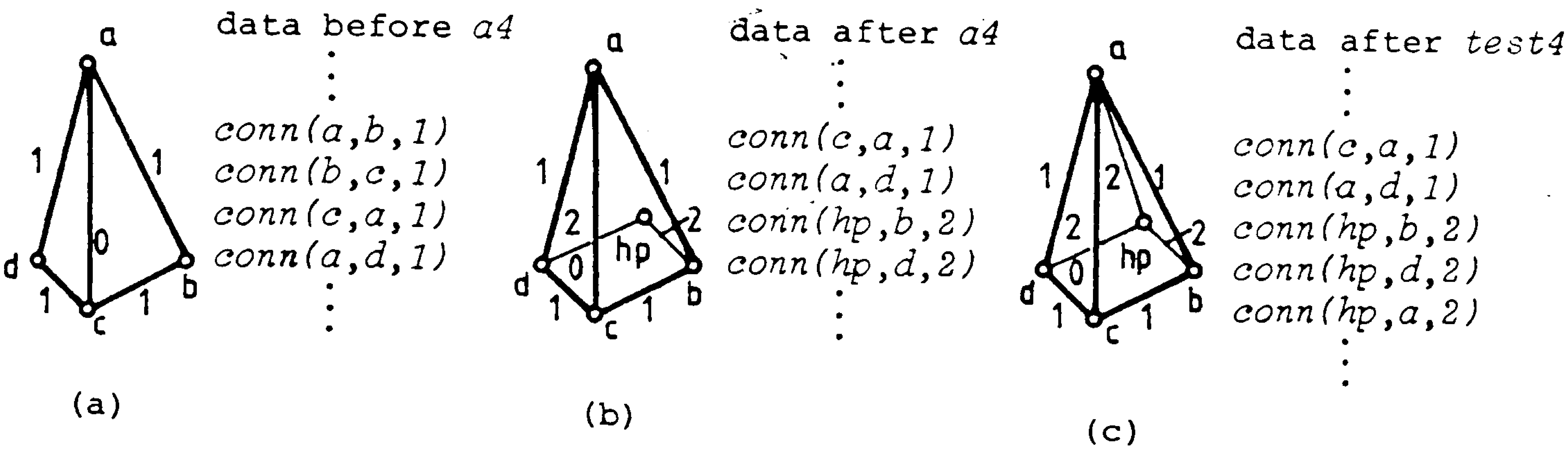


FIGURE 6.19

In the case of the 3-D figure of Figure 6.19a, assumption *a4* would have the effect of adding hidden lines *b_line(d,hp)* and *b_line(b,hp)* to the database. This would result in an unrealistic situation (Fig. 6.18b) of faces (*abhpd*) and (*dhpbc*) being co-planar. To prevent this awkward situation, the procedure records all *hidden quadrilaterals** that are formed after assumption *a4* has been carried out. If two of them share the same three vertices, then the second is an invalid one, and as such it is split into two triangles by asserting its diagonal (i.e. *b_line(a,hp)* in Fig.6.19c).

** hidden quadrilateral is a quadrilateral that contains hidden points hp, i.e. c_quadril(A,B,C,hp).*

6.5 FUNCTION OF THE RECOGNIZER

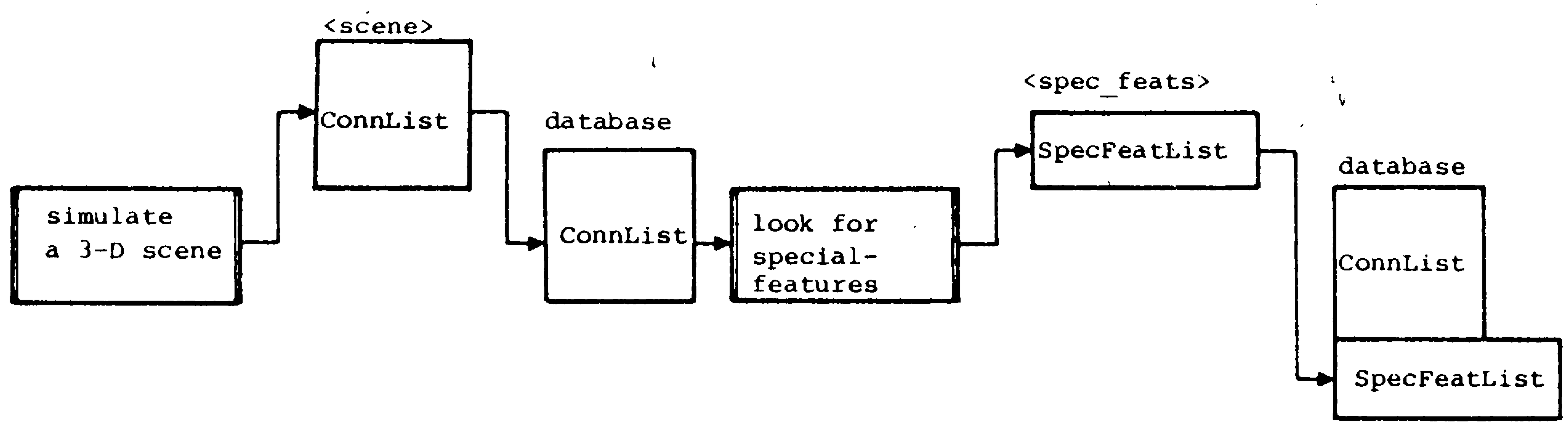
Until now, the individual procedures of the recognizer have been examined and their functions on several 3-D figures have been described in detail. The purpose of this section is to regard the procedure as a whole and examine how it applies to a scene of 3-D figures. The routine of the recognizer is basically the single-view recognizer and the other two routines (i.e. assumptions and multiple-view recognizer) are considered as two optional subroutines called by it. The scene to be recognized is made up of a number of 3-D figures supplied by the simulator.

The recognition phase is completed in two parts. The first part is called [see] and its goal is to create a scene of 3-D figures for the recognizer. It begins by initializing the database i.e. retracting all predicates that may exist from previous scenes or recognitions. Then, it calls the [simulator] to build up a scene and stores it in file <scene>. It inserts the 'scene' into the database, displays it on the screen, and segments it into a set of possible faces. Next, it calls procedure [link], that searches the scene for special features which are added to the original database. The set of special features (if there are any) is displayed, and the procedure ends with the message: *ready for recognition.*

The second part is called [rec] and attempts to recognize the 3-D figures that make up the scene. The procedure starts by offering to the user the possibility to look for a certain set of figures or to seek all figures with face-visibility over a certain limit. The default option will look for all recognizable 3-D figures regardless of their face-visibility. The next step is to form a list of all the planes in the scene, based on the new database. The planes, which represent the faces of the 3-D figures, are subdivided in visible ones and possible ones. These two lists are printed out by the system. At this point the procedure is ready to start the recognition. It looks for the first *p_figure*, forms a list of the 3-D figures that it may represent, and prints out the list. Then, the different alternatives are printed out in order of decreasing face-visibility (i.e. the *p_figure* with the highest face-visibility will come out first). The *conn*'s that constitute this figure will be modified (will become *a_conn*'s or *b_conn*'s), while the rest of the *conn*'s will remain unaffected. Next, the procedure calls [assume], that asks the user for an assumption-code, and the vertex list of the *p_figure* for which the assumption is meant. The corresponding assumption is carried out and introduces a list of hidden lines that is saved in the file <hidden>. The procedure continues by calling the [mlt_rec] (which is optional), to perform multiple-view recognition. If the user decides to proceed to multi-view recognition, [mlt_rec] composes a new database consisting of the *b_conn*'s that constitute the <scene>, and the hidden-line list that is stored in the file <hidden>. The system will come up with the name of the 3-D figure, if it is a recognizable one or the answer: *non-recognizable 3-D figure*. The above sequence, constitutes a *cycle of recognition*. If the user wishes to try an alternative assumption, types *alt*. Procedure [alt] carries out the new assumption and calls the [mlt_rec] automatically. If the procedure [rec] fails to find a certain 3-D figure or a 3-D figure with a certain face-visibility, it returns the answer: *no such 3-D figure*. If it can not recognize a 3-D figure, it prints out the message: *non-recognizable 3-D figure*. The

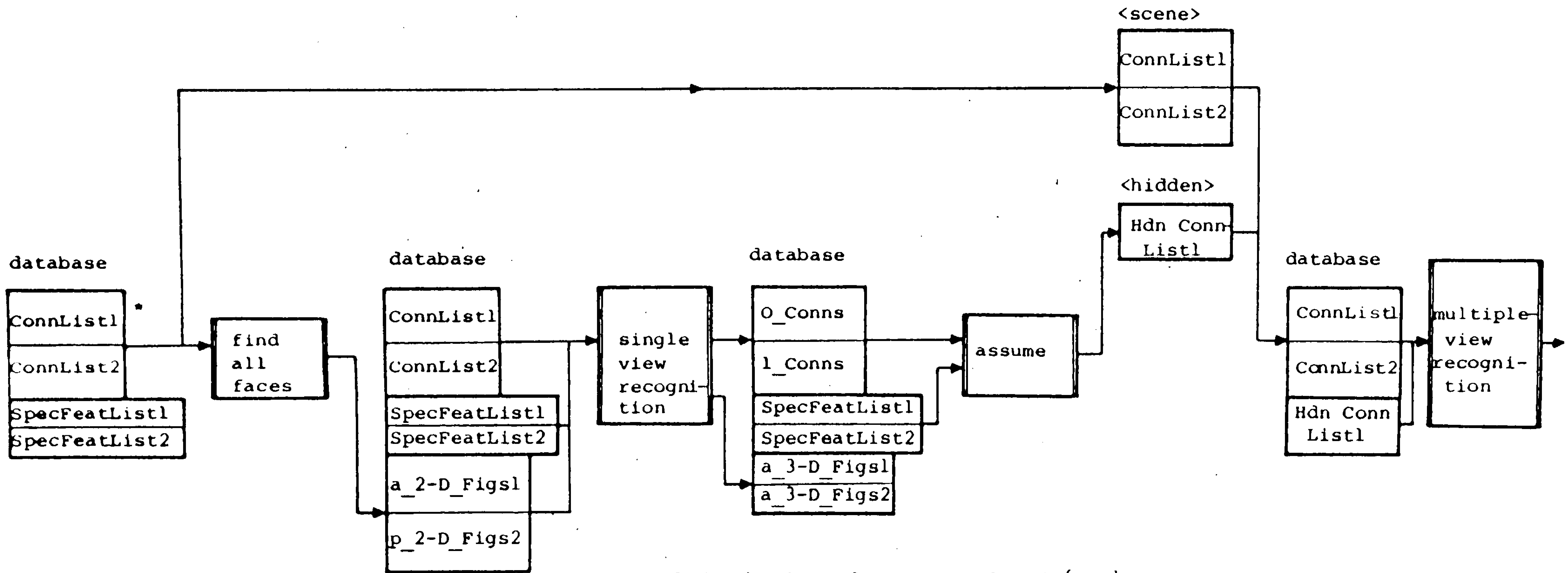
clear
database
[simulator]
create a 3-D scene
express it in
terms of conn's
conn's
to <scene>
conn's
in
database
segment
scene into
2-D figures
[link]
look for special-
features in scene
SpecFeatList
to <spec_feat>
SpecFeatList
in database
database
scene
conn's
special-features
any
?
yes
SpecFeatList
none
ready for recognition
END
[mlt_rec]
consult
<scene>
consult
<hidden>
look for a
multiple-view
3-D figure
any
?
yes
multiple-view 3-D figure
non-recognizable
3-D figure
name of
3-D figure
RETURN
START
find all
visible faces
any
?
yes
visible faces
visible
faces
find all
invisible faces
any
?
yes
invisible faces
invisible
faces
looking for any particular 3-D figures?
yes
?
yes
insert list of 3-D figures
FigList=[ ]
list of
3-D figures
any particular face-visibility?
yes
?
yes
insert % of face visibility
FaceVis=0
% of face-
visibility
find all
p_figure's
any
?
yes
Update conn's
of p_figure's
FigList=[ ]
& FaceVis<=17
?
yes
no such
3-D figure
non-recognizable
3-D figure
END
insert assumption-code
assumption
code
insert vertex-list to
specify 3-D figure
vertex-
list
carry out
assumption for
specified figure
hidden conn's
in <hidden>
assumed hidden lines
hidden lines
clear
database
RETURN
single_view 3-D figure
name of
3-D figure
or
alternatives
?
yes
assumption?
yes
?
yes
[assume]
make an
assumption
multiple-view recognition
yes
?
yes
[mlt_rec]
multiple-view
recognition

FIGURE 6.20



Change of the database during *[see]*

(a)



Change of the database during a cycle of *[rec]*

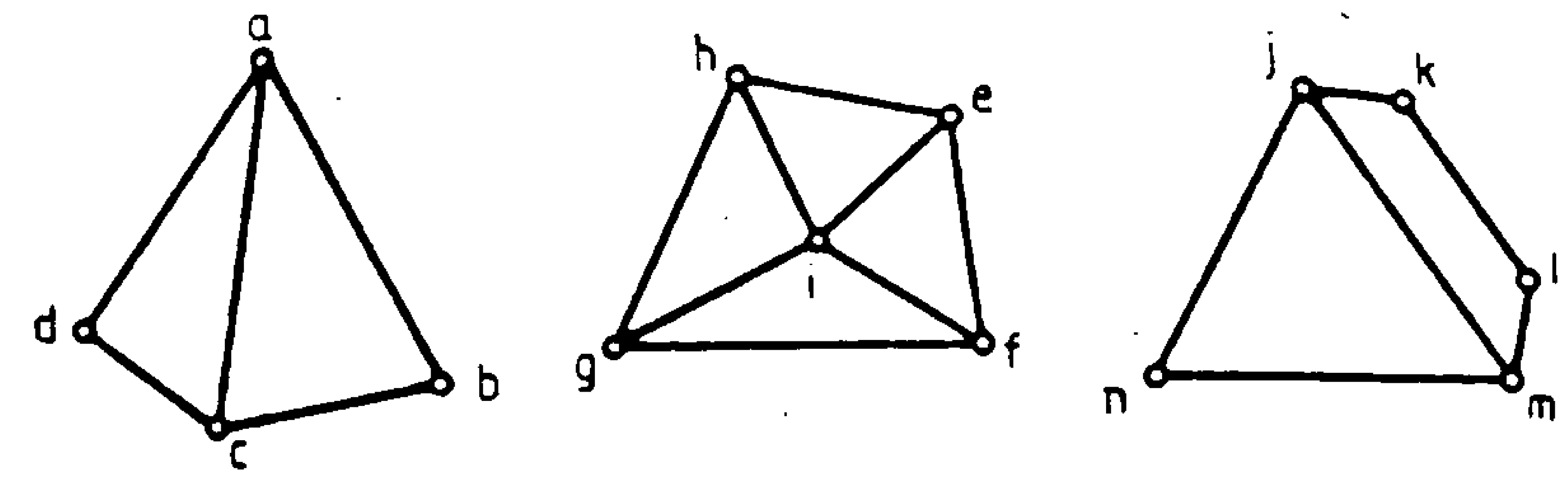
(b)

*ConnList1 and 2 correspond to two not related views of *conn*'s (i.e. there are two different 3-D figures in the scene)

FIGURE 6.21

end of a recognition cycle is marked with the message: *end of recognition*. The flowcharts of Figure 6.20 illustrate the two main routines [see] and [rec] of the recognition phase. The change of the database is illustrated in Figure 6.21.

Example: The system is presented with the following scene:



The *conn*'s that make up the scene are asserted to the system's database by consulting the file <scene>. Segmentation will yield the following list of *s_figure*'s:

[s_trian(a,b,c),s_trian(a,c,d),s_trian(e,i,f),s_trian(i,h,g),
s_trian(i,g,f),s_trian(j,m,n),s_quadril(e,f,g,h),s_quadril(j,k,l,m)].

and [link] stores *point_in_qul(e,f,g,h)* in <spec_feat>. The system consults <spec feat> and prints out the database:

database

scene

conn(a,b,2).
conn(b,c,2).
⋮
conn(n,j,2).

special feature(s)

point_in_qul(e,f,g,h)

ready for recognition

The user calls [rec] and the system replies:

looking for any particular 3-D figure(s)?   n. (for 'no')

The user decides to have a look at all the faces in the scene

any particular face-visibility?   n.

Again, the user does not restrict the system in its search for 3-D figures.

visible 2-D face(s)

triangle(a,b,c)	*(1. atrian(a,b,c).*
triangle(a,c,d)	*2. atrian(a,c,d).*
triangle(e,i,f)	*3. atrian(e,i,f).*
triangle(e,i,h)	*4. atrian(e,i,h).*
triangle(i,h,g)	*5. atrian(i,h,g).*
triangle(i,g,f)	*6. atrian(i,g,f).*
triangle(j,m,n)	*7. atrian(j,m,n).*
convex_quadrilateral(j,k,l,m)	*8. aquadril(j,k,l,m).*

possible 2-D face(s)

convex_quadrilateral(e,f,g,h)	*9. pquadril(e,f,g,h).)*

The second column in brackets shows the assertions to the database that mark that the existence of the 2-D figures of the first column. The *p_figure* rules are tried next and the following predicates are asserted to the database:

a_pyram1(e,f,g,h,i), *a_pyram3(j,k,l,m,n)* and *a_tetra2(a,b,c,d)*.

which cause the following answer by the system:

single-view figure(s)

pyramid(e,f,g,h,i),80,a1

pyramid(j,k,l,m,n),40,a2
or
prism(j,k,l,m,n,F),33,a4

tetrahedron(a,b,c,d),50,a2
or
pyramid(a,b,c,d,E),20,a4

assumption? y. (for 'yes')

insert assumption-code

a1: no hidden lines or points behind possible face
a2: hidden(triangle)line
⋮
a4: hidden(quadrilateral)point
⋮
a7: entirely hidden face

a4.

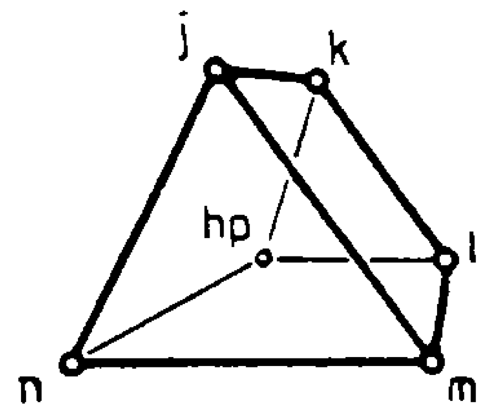
The user decides assumption code *a4*

insert vertex-list to specify 3-D figure
with format [v1,v2,...,vN]
[j,k,l,m,n].

The user picks the second 3-D figure.

assumed hidden line(s)

line(k,hp)
line(l,hp)
line(n,hp)



The system assumes a hidden vertex *hp* and asserts *conn(k,hp,2)*, *conn(l,hp,2)*, and *conn(n,hp,2)* to the database (shown by the figure).

multiple-view recognition? y.

prism(j,k,l,m,n,hp)

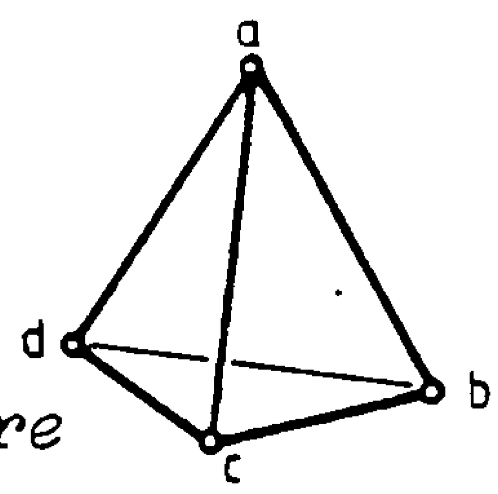
end of recognition

The user wishes an alternative assumption and calls [alt]. The system reacts as before asking for assumption-code and vertex-list and receives: a2 and [a,b,c,d], in which case it replies:

assumed hidden line(s)

line(b,d)

multiple-view 3-D figure



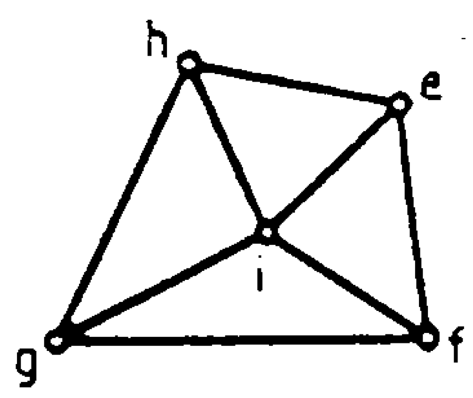
tetrahedron(a,b,c,d)

end of scene recognition

If the user types in the pair (a1,[e,f,g,h,i]) the system will assume:

assumed hidden line(s)

none, also no hidden points



and will recognize the 3-D figure as:

pyramid(e,f,g,h,i)

REFERENCES

1. BENNETT, W.S. 1983: *A Review of Some of the Better Known Techniques Applied to Simulation*, Computer Image Generation, Ed: Schachter, Wiley-Interscience Pub., pp. 18-26.
2. GABRIELIDIS, G. 1982: *Recognition of Simple 3-D Objects by the Use of Syntactic Pattern Recognition*, M.Phil. Thesis, Dept. Comp. Science, Loughborough Univ.Tech., June, pp. 95-126.
3. HARRINGTON, S. 1983: *Computer Graphics: A Programming Approach*, McGraw-Hill; a: pp. 314-316; b: pp. 347-348.

CHAPTER 7

DISCUSSION

7.1 INTRODUCTION

The objective of this project is to develop a system that learns to recognize scenes of 3-D objects, from line drawings generated by a 3-D figure simulator. The system is divided into three main parts:

A 3-D figure *simulator*, that generates line drawings representing 3-D objects, which are used as input to the system.

A 3-D figure *learner*, that has the task of learning a list of 3-D figures consisting of triangular and quadrilateral faces, from single or multiple views.

A 3-D figure *recognizer*, that attempts to recognize the 3-D figures that make up a scene created by the simulator.

A detailed description of these parts, their function and the necessary background information was presented in the previous chapters. The next paragraphs discuss the performance of each individual part, analyse its advantages and disadvantages, and make several suggestions for further improvement. The conclusion at the end of the chapter contains the main points of the discussion.

7.2 THE SIMULATOR

The simulator uses as input a *vertex-array* of 3-D cartesian co-ordinates defining the vertex positions of 3-D figures, and a *face-array* that defines their connectivity. Its output consists of a list of *conn*-predicates, that correspond to the visible edges of the 3-D figures it generates. These structurally represented 3-D figures are the input to the learner and the recognizer.

The simulator is capable of generating 3-D figures (or 3-D scenes) with a maximum of *119 vertices* and *119 faces*. The faces are *closed polygons*, each with a maximum of *8 sides*. It includes a *back-face*, and *hidden-line removal* mechanism designed mainly for *convex polyhedra*. This mechanism can also be used for *non-convex* polyhedra provided that no *partial face-occlusion* occurs. The removal of a partially occluded face depends on the particular vertex that is chosen for the visibility test. The original input 3-D figure(s) can be subjected to transformations (*rotation*, *scaling*, *translation*). Each transformation can be repeated several times until a wished 'view' is obtained. This allows the user to compose his/her own 3-D scene for recognition. A visual output of the simulated scene consists of a *central* and/or *orthographic projection* of the scene on a *VDU* and/or on *paper*. The centre of projection and the viewing pyramid can be defined by the user. Finally, an extra facility allows the hidden lines of the scene to be displayed on the final plot. If the simulator is used by the learner, it generates (and plots) *6 alternative views* of the 3-D figure to be learnt, which are needed for the *multiple-view learning*.

The 3-D figure simulator is a *fast* and *reliable* method for producing *perfect* line drawings and their respective database. It has no problem in generating 3-D scenes made up by the range of 3-D figures used in this project (see §7.4) following the assumptions of §6.1. On the other hand, the simulator has a number of weak points, which can also serve as goals for futher improvement. Some of these are: It does not allow object occlusion by other objects. As a result of this, alternative views of scenes where no occlusion of objects occurs can not practically be obtained. Partial occlusion by the same or by other objects is also not considered. A further improvement of the simulator could involve *curved* objects, two *stereoscopic* images, and *time varying* images.

Using a simulator as an alternative to normal input device (e.g. a camera) has the following advantages:

a) it delivers perfect line drawings without the need of low level processing,

b) it is independent of the objects composing the scene and the conditions of illumination,

c) it provides an almost direct input to high-level vision procedures,

d) it is cheaper because it does not need any expensive equipment (e.g. digitizer).

Some of its disadvantages are:

a) back face removal is a computer-time costly process,

b) its input is sometimes 'too ideal' to represent real-world scenes.

7.3 THE LEARNER

The learner consists of three main parts, the *elementary-concept learner*, the *multiple-view learner*, and the *single-view learner*, that learn the definitions (5.1-5.39) that are used by the recognizer. It uses the method of *learning from examples*, which are provided by the user, and each of its three parts demonstrates several techniques of *inductive learning*.

The elementary-concept learner uses an input of *conn*'s, *special-feature* predicates, and *a_figure* predicates, in order to build the *background knowledge base* that is used by the other two parts. It works in a *bottom-up* fashion, using a *structural representation*. It possesses an 'oracle' *(r_rules)* that determines the *truth value* of the *training instances* supplied by the user. The rules are obtained by the *generalization* technique of *turning constants to variables*. For elementary concepts (e.g. *line*) a small *description space* is used. In order to express relations between arguments, this contains the following functors: (=, \=, *atom*, *integer*, >,<, *member*). At this level only *selective generalization* is used. As the concepts become more complex the use of *constructive generalization* is introduced. As the system learns new concepts, it builds up a *description space* from which it can choose new conditions. These are created from the *discriminating element* that is extracted from *positive* and *negative training instances*. This is in agreement with the general learning system proposed by Bundy & Silver (see §5.4), saying that 'the description space must always be supplied by the user' . An interesting idea on this point is given at the end of the section. The learning process depends on the kind of examples that it receives. Basically, elementary concepts after the first example, require at least as many examples as there are conditions in the corresponding *r_rules*. The process continues until no more errors can be detected.

The *multiple-view learner* uses the same database (except the *a_figure*'s) as the previous part. It learns the multiple-view definitions of the 3-D figures considered in this project, from alternative views (positive examples) determined by itself. This is based on the very important element of the *conn*'s, the *face counter* (see 5.3.1). The face counters of the *conn*'s involved in a certain view are modified for the visible sides. Those of them $\neq 0$ are used by a set of criteria to decide on the next view. The system requires at least two different views of the same 3-D figure, in order to learn a multiple-view definition. The process is continued until there are no more invisible lines in the database. The system uses *constructive generalization* based on the *background knowledge base* created by the elementary-concept learner. The multiple-view learner demonstrates the importance of the face counter, and the power of alternative views. The latter is possible from the simulator's point of view, because it involves only single objects.

The *single-view recognizer* takes *a_figures* as input, and generates the single-view definitions based on the results of the previous process. It works very much like the elementary-concept learner, except that it does not possess an 'oracle' and the user provides the truth value of the training instances.

The learner is a *special purpose* system that uses *induction* to generate its rules. Its performance as a whole is more than satisfactory. On the other hand, it has two weak points. It considers only *near miss* examples, and its control depends very much on the user. Further system improvement could include the investigation of the above two points, as well as the capability of learning more 'clever' rules like:

```
non_cnvx_cntr_angl(A,VertexList):-non_convex_contour_angle(A),
                                  member(A,VertexList).
```

All current learning systems, leave the choice of the description space to the user. This means, that the user is responsible for providing a set of literals on which the buildup of the rule is based. It would be very useful if the system itself could 'have an opinion' on its need for concepts, that represent intermediate stages between the original database and the concept that is about to learn. For example, supposing that the system is given a set of *conn*'s and is going to learn about *tetrahedron*'s without knowing the rules about *line*'s and *trian*'s. It is worth developing a mechanism that gives the system the capability to create one or two intermediate concepts (e.g. *conn_level1*, *conn_level2*), which the user can rename afterwards, using more familiar names, (e.g. *line* and *trian* respectively). Figure 7.1 illustrates this idea.

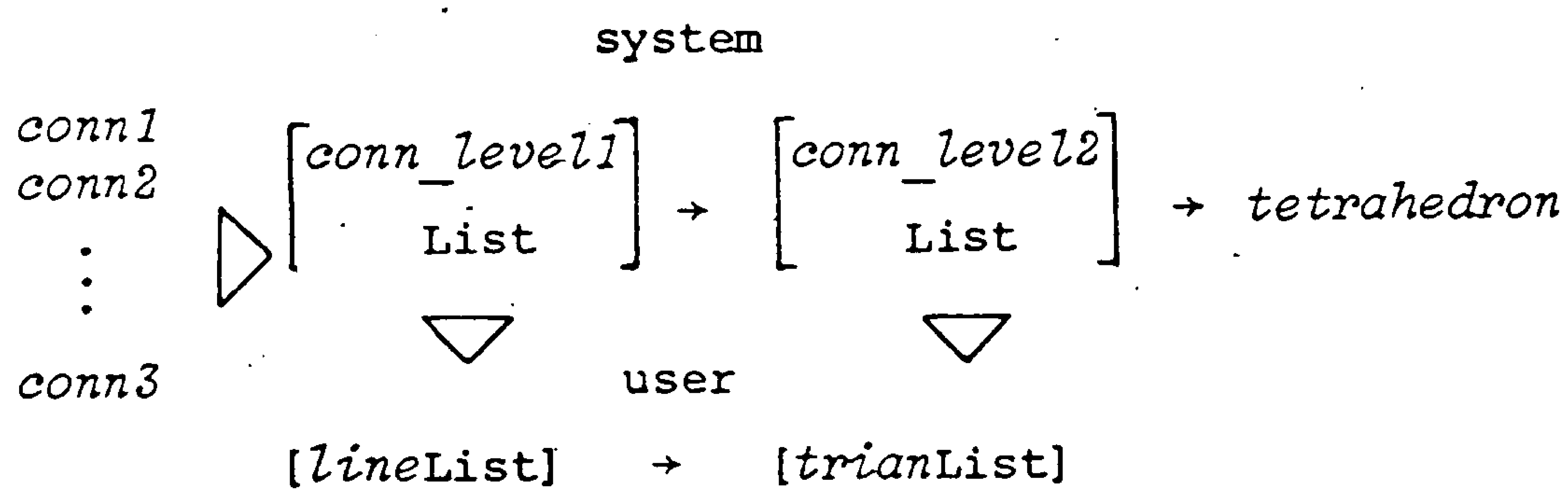
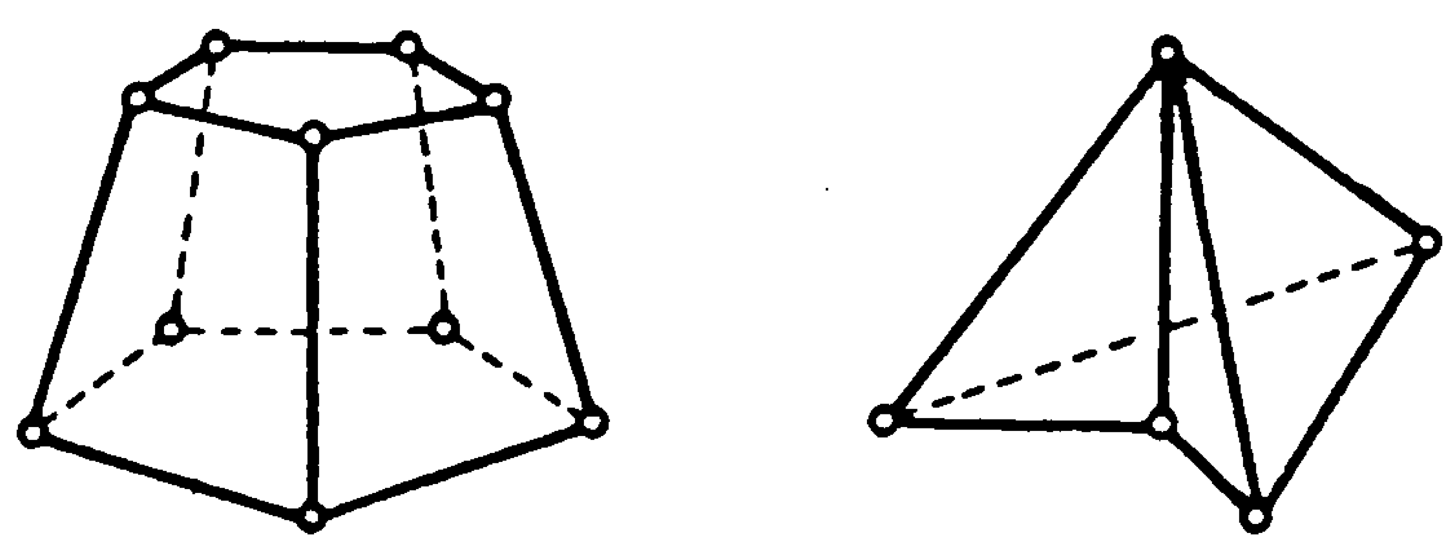


FIGURE 7.1

7.4 THE RECOGNIZER

The input to the recognizer consists primarily of the *conn*-predicates created by the simulator. The goal of the recognizer is to interpret a list of connected points as 3-D objects, using the minimum possible information about them. It begins by *segmenting* the scene into *triangular* and *quadrilateral* polygons. Segmentation is followed by a number of analytical calculations that determine whether the segmented polygons correspond to polyhedral faces or not. This is indicated by adding predicates *point_in_*, *non_convex_qul* and *non_convex_contour_angle* to the original database. The *single-view recognizer* uses the new database to recognize the 3-D objects that compose the input scene. An (optional) *assumption* on individual objects looks for *hidden lines*, and the (also optional) *multiple-view recognizer* gives a unique answer about the above object, based on the assumed hidden lines.

The recognizer was tested with a number of scenes. The set of objects that was used to make up the scenes included, apart from the learnt ones, a *pentagonal prism*, and a *non-convex pyramid* (Fig. 7.2).



pentagonal prism non-convex pyramid

FIGURE 7.2

The lengthiest procedures are the segmentation and the single-view recognition. The recognition time varies from a minimum of a few seconds, for single-object scenes, to several minutes, for scenes with more than one object. It also depends on whether non-recognizable objects are included in the scene (longer), and whether certain objects are looked for (shorter). The assumption is virtually a way round the simulator's weakness to obtain several alternative views of a scene. It uses the fact that lines with a face counter = 1 belong to occluded faces, to insert hidden lines into the database. Its advantage over another view is, that no matching of objects between the two views is needed. However it has the drawback of being ad-hoc, and it takes several attempts to find a correct assumption if the assumption code is not known beforehand. The multiple-view recognizer verifies the result of a correct assumption. Both procedures are applied to single objects, and therefore, need a relatively short time (seconds) to be completed.

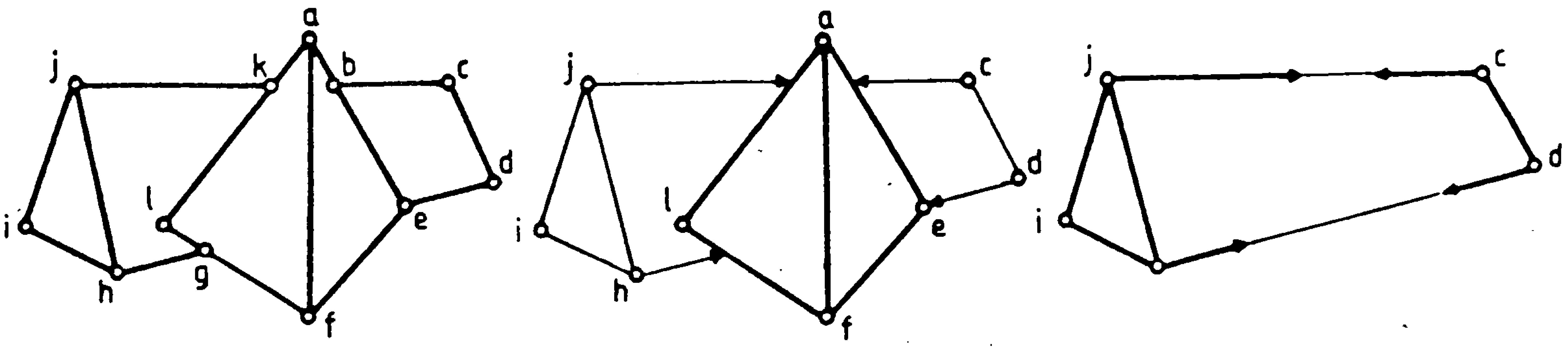
Recognition of polyhedra with polygonal faces with number of faces greater than four, and partial occlusion of objects, are two of the features not included in the capabilities of the system. However, the recognizer has an extensible structure and can cope with such problems. Some raw ideas and suggestions for tackling the above problems are given below.

Polygons with more than four sides can be easily added to the definitions of §5.3.2. For example the definition of a *pentangle* could look like:

```
pentan(A,B,C,D,E):- line(A,B),line(B,C),line(C,D),line(D,E),line(E,A),
                    not(line(A,C)),not(line(A,D)),not(line(B,D)),
                    not(line(B,E)),not(line(C,E)),
                    A\=C,A\=D,B\=D,B\=E,C\=E,
                    not(point_in_pnt(A,B,C,D,E).
```

Of course, the increase in the number of sides affects the number of conditions in the rule, and obviously this slows the process down.

Now to the problem of partial occlusion. Supposing that the scene of Figure 7.3a is given for recognition. At first, the system looks for



(a) (b) (c)

FIGURE 7.3

closed polygonal lines. Each of these lines is tested to see if it includes points (vertices) that lie on the same side. This eliminates points *b*, *g*, and *k* from the scene (Fig. 7.3b). *Conn*'s like *conn(j,k,2)* are replaced with *open-end conn*'s *open(j,oe,2)*, and the new scene is searched for meaningful 3-D figures according to the definitions of §5.3.3. This succeeds in finding *tetrahedron(a,e,f,l)*[*], which is removed from the scene (Fig. 7.3c). A test checks whether any *open-end conn*'s are co-linear or intersect, (within certain limits) and inserts the missing line parts or the points of intersection respectively to the database. This leaves only one possibility, *prism(c,d,h,i,j)*[*]. This method is a natural continuation of the original single-view recognizer and it has the advantage over the classical line labelling techniques (§3.2.3), that it does not need to memorize tables of line labels. Nevertheless, this basic idea needs more elaboration if it is to cope with non-convex polyhedra. A natural consequence to this, would be the

[*] *The choice of names is random.*

replacement of the assumption with another view. Finally, the system could be further improved to cope with *curved* objects.

7.5 CONCLUSION

A system that uses inductive learning in order to create a set of rules that recognize 3-D objects has been presented. A simulator of line drawings generates a database that introduces a structural representation, which plays an important role in both the learning and the recognizing process. Learning involves the generation of two sets of rules. The first is used to recognize 3-D objects from a single view, and is obtained from examples supplied by the user. The second recognizes the same objects from multiple views, and is obtained from alternative views selected by the system.

The system has been tested on scenes of non-occluding polyhedra with triangular and quadrilateral faces with good results. The capabilities of the system can be extended to include more complex polyhedra and to cope with object occlusion. Finally, it would be interesting to incorporate fuzzy logic into the system, in order to express the uncertainty in the single-view recognition.

APPENDIX 1

1.1 THE 'C' PROGRAMS

The programs written in the programming language C are the simulator and the link. C is a general-purpose language [Kerninghan & Ritchie '78], which is used in this project as a support to PROLOG for two main reasons:

a) It is faster at calculations involving real numbers.

b) It can use graphic facilities.

The simulator (see §3.3) is stored in file <sim.c>, and its executable form in file <simulator>. It uses the standard C library, a mathematical library. It also uses the PH-PLOT/21 Plotter Software Package [HP-7221T '81] in order to obtain plots of scenes on paper, and the GINO-F Graphics Package [Gino-F '83] in order to perform plotting of scenes on a graphics VDU.

The link (see §6.3) is stored in the file <lnk.c> and its executable form in file <link>. It uses only the standard C library and a mathematical library.

Paragraph 1.2 contains a listing of the C programs.

REFERENCES

1. GINO-F GRAPHICS PACKAGE 1983: *Programmer Manual.*
2. HP-7221T 1981: *Graphics Software User's Manual for Use with Hewlett-Packard 7221 Series Graphic Plotters.*
3. KERNIGHAM, B.W. and RITCHIE, D.M. 1978: *The C Programming Language*, Prentice-Hall.

```
                    /* simulator */

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

/* end markers */

#define End  100.0
#define LAST   0
#define END   -1
#define space     printf("\n")
#define manually  printf("             manually ?    ")
#define positive  fprintf(prlg,"positive.\n")

/* pen colours */

#define BLACK   1
#define RED     2
#define ORANGE  3
#define YELLOW  4
#define GREEN   5
#define BROWN   6
#define BLUE    7
#define PURPLE  8
#define VIOLET 10

/* symbol */

#define CIRCLE  7
FILE *inpt,*inpf,*prlg,*oupt,*in3,*fopen(),*fclose();
double sin(),cos();
double a,b,c,l,m,n,Zo,Zi,Zu,Pw,Ph,V[120][2],T[120][4],Vc[120][2],
      vf[120][3],Rt[4][4],Sc[4][4],Tr[4][4],Pr[4][4],d[120][4],
       Az,By,Cx;
int N,M,L,f[120][10],VIS[120],INV[120],vis[20],inv[20],
    s[10],t[20][10],R,P,Vln[120][2],Iln[120][2],pen,vln[20][2],
    Rl,Rm,Rr,Rlc,Rmc,Prv,Prvc,Prf,Prfc,Fc[120][10],view[120];
char *title1=" orthographic projection ";
char *title2=" central projection ";
char *bac="back";
char *top="top";
char *bot="bottom";
char *lef="left";
char *rig="right";
char ttl[100],fnm[30],objname[30],lrn[4],mnl[4],rep[4],Ch[6];

main()
{  int i,j;
   char par[4],mor[4],plo[4];
space;
printf("      define viewing-pyramid parameters\n");
space;
printf("            standard values ?    ");
scanf("%s",par);
        if(par[0]=='n')
        {
space;
printf("      insert viewing-pyramid parameters :\n");
space;
printf("         Zo : centre of projection\n");
printf("         Zi : position of projection plane on z axis\n");
printf("         Zu : user coordinate origin\n");
printf("         Ph : projection plane height from Zi\n");
printf("         Pw : projection plane width from Zi\n");
space;
printf("               Zo Zi Zu Ph Pw\n");
space;
scanf("%f%f%f%f%f",&Zo,&Zi,&Zu,&Ph,&Pw);
          }
          else
            def_view_pyr();
space;
printf("      viewing-pyramid parameters\n");
space;
printf("Zo = %5.2f Zi = %5.2f Zu = %5.2f\n",Zo,Zi,Zu);
printf("   Ph = %5.2f Pw = %5.2f\n",Ph,Pw);
space;
space;
printf("            learning ?   ");
scanf("%s",lrn);
     if(lrn[0]=='n')
     {
space;
printf("      insert scene\n");
space;
          init();
manually;
scanf("%s",mnl);

/* scene file input */

        if(mnl[0]=='n')
        {
space;
printf("      insert file name of scene\n");
space;
scanf("%s",fnm);
             inpt=fopen(fnm,"r");
             for(i=0;(i<120&&(fscanf(inpt,"%f%f%f",&T[i][0],&T[i][1],&T[i][2])&&

                   (T[i][0]!=End)));i++);
             N=i;
             for(i=0;(i<120&&fscanf(inpt,"%d",&f[i][0])&&f[i][0]!=END);i++)

                for(j=1;(j<6&&fscanf(inpt,"%d",&f[i][j])&&f[i][j]!=LAST);j++);

             M=i;
             fclose(inpt);
             loadf();
             homocoord();
             store();
             scr_plot();
             transform();
             clip();
             hidden(&Rl,&Rm,VIS,INV,&Prfc);
             loadv();
             c_project();
             c_clip();
             loadvc();
        }

/* manual scene input */

     else if(mnl[0]=='y')
          {   mor[0]='y';
              while(mor[0]=='y')
              {   object_in();
                  homocoord();
                  loadf();
                  store();
                  scr_plot();
                  transform();
```

```
                    clip();
                    hidden(&Rl,&Rm,VIS,INV,&Prfc);
                    loadv();
                    c_project();
                    c_clip();
                    loadvc();
space;
printf("            new 3-D figure ?     ");
scanf("%s",mor);
space;
                }
        }
        mark_end();
        vis_lines(Vln,VIS,Fc);
        inv_lines(Iln,Vln,INV,Fc);
        prolog();
        coord();
space;
printf("            plot scene ?     ");
scanf("%s",plo);
space;

/* plot scene */

        if(plo[0]=='y')
           plotout();
space;
printf("      end of scene build-up\n");
space;
    }

/* create alternatine views (learning)*/

    else
       all_views();
    exit(0);
}

/* define viewing pyramid */

def_view_pyr()
{   Zo= -40.0;
    Zi= -20.0;
    Zu=   0.0;
    Ph=  10.0;
    Pw=  15.0;
    return;
}

/* initialize pointers */

init()
{   M=0;
    N=0;
    Prfc=0;
    Prf=0;
    Prv=1;
    Rl=0;
    Rm=0;
    Rr=0;
    Prvc=1;
    Rlc=0;
    Rmc=0;
    return;
 }

/* input 3-D figure */

object_in()
{   int i,j;
space;
printf("      insert 3-D figure\n");
space;
manually;
scanf("%s",mnl);

/* 3-D figure file input */

    if(mnl[0]=='n')
    {
space;
printf("      insert filename of 3-D figure\n");
space;
scanf("%s",fnm);
        inpt=fopen(fnm,"r");
        for(i=0;(i<10&&(fscanf(inpt,"%f%f%f",&T[i][0],&T[i][1],&T[i][2])
            &&(T[i][0]!=End)));i++);
        N=i;
        for(i=0;(i<120&&fscanf(inpt,"%d",&f[i][0])&&f[i][0]!=END);i++)
           for(j=1;(j<6&&fscanf(inpt,"%d",&f[i][j])&&f[i][j]!=LAST);j++)

        M=i;
        fclose(inpt);
    }

/* 3-D figure manual input */

else if(mnl[0]=='y')
    {
space;
printf("       insert coordinates Vx,Vy,Vz of vertex array V[i][3]\n");
space;
       for(i=0;(i<10&&scanf("%f%f%f",&T[i][0],&T[i][1],
              &T[i][2])&&T[i][0]!=End);i++);
space;
       N=i;
space;
printf("       insert  elements of face array f[120][6]\n");
space;
       for(i=0;(i<120&&scanf("%d",&f[i][0])&&f[i][0]!=END);i++)
          for(j=1;(j<6&&scanf("%d",&f[i][j])&&f[i][j]!=LAST);j++);
       M=i;
    }
    return;
}

/* set end markers */

mark_end()
{   int i,j;
    for(i=1;VIS[i]!=LAST;i++);
    VIS[i]= END;
    for(i=1;INV[i]!=LAST;i++);
    INV[i]= END;
    Fc[Prf][0]= END;
    Vc[Prvc][0]= End;
    V[Prv][0]= End;
    return;
}

/* files with alternative views (learning) */
```

```
all_views()
{   char plo[4];
    int i,K;
    init();
    object_in();
    front();
    for(i=0;vln[i][0]!=END;i++)
        view[i]=vln[i][0];
    K=i;
    new_view(180.0,  0.0,-10.0,-5.0,"v_bac");
    for(i=0;vln[i][0]!=END;i++)
        view[K+i]=vln[i][0];
    K=K+i;
    new_view(  0.0,-90.0,  0.0, 5.0,"v_top");
    for(i=0;vln[i][0]!=END;i++)
        view[K+i]=vln[i][0];
    K=K+i;
    new_view(  0.0, 90.0,  0.0,-5.0,"v_bot");
    for(i=0;vln[i][0]!=END;i++)
        view[K+i]=vln[i][0];
    K=K+i;
    new_view(-90.0,  0.0, 10.0, 5.0,"v_lef");
    for(i=0;vln[i][0]!=END;i++)
        view[K+i]=vln[i][0];
    K=K+i;
    new_view( 90.0,  0.0, 10.0,-5.0,"v_rig");
    for(i=0;vln[i][0]!=END;i++)
        view[K+i]=vln[i][0];
    K=K+i;
    mark_end();
    vis_lines(Vln,VIS,Fc);
    inv_lines(Iln,Vln,INV,Fc);
space;
printf("            plot other views ?    ");
scanf("%s",plo);
space;

/* plot alternative views */

    if(plo[0]=='y')
        plotout();
    return;
}

/* create frontal view (learning) */

front()
{   int i;
    char fname[32];
    loadf();
    homocoord();
    store();
    scr_plot();
    transform();
    clip();
    store();
    hidden(&Rl,&Rm,VIS,INV,&Prfc);
    l = -10.0;
    m =  5.0;
    n =  0.0;
    translate();
    loadv();
    insert();
    c_project();
    c_clip();
    l = -10.0;
    m =  5.0;
    n =  0.0;
    homocoord();
    translate();
    loadvc();
space;
printf("       insert name of 3-D object\n");
space;
scanf("%s",objname);
space;
    fname[0]='p';
    fname[1]='_';
    for(i=0;i<30;i++)
        fname[i+2]=objname[i];
    vis_lines(vln,vis,f);
    verElist();
    s[P]= END;
    v_prolog(fname);
    return;
}

/* create a single alternative view (learning) */

new_view(AngY,AngX,QuaX,QuaY,View)
    double AngX,AngY,QuaX,QuaY;
    char View[5];
{   insert();
    homocoord();
    loadf();
    Az = 0.0;
    By = AngY;
    Cx = AngX;
    rotate();
    hidden(&Rl,&Rm,VIS,INV,&Prfc);
    l = QuaX;
    m = QuaY;
    n = 0.0;
    translate();
    clip();
    loadv();
    insert();
    Az = 0.0;
    By = AngY;
    Cx = AngX;
    rotate();
    c_project();
    c_clip();
    l = QuaX;
    m = QuaY;
    n = 0.0;
    homocoord();
    translate();
    loadvc();
    vis_lines(vln,vis,f);
    verElist();
    s[P]= END;
    v_prolog(View);
    return;
}

/* save vertex-array */

store()
{   int i,j;
    for(i=0;i<120;i++)
        for(j=0;j<3;j++)
```

```
            vf[i][j]=T[i][j];
      return;
 }

/* retrieve vertex-array */

insert()
{   int i,j;
    for(i=0;i<120;i++)
       for(j=0;j<3;j++)
          T[i][j]=Vf[i][j];
    return;
}

/* form scene face-array */

loadf()
{   int i,j;
    Prfc=Prf;
    for(i=0;i<=M-1;i++)
       for(j=0;(j<10&&f[i][j]!=LAST);j++)
          Fc[i+Prf][j]=f[i][j]+Prv-1;
    Prf=i+Prf;
    return;
}

/* form scene vertex-array for central projection */

loadvc()
{   int i,j;
    for(i=0;i<=N-1;i++)
       for(j=0;j<=1;j++)
          Vc[i+Prvc][j]=T[i][j];
    Prvc=Prvc+i;
    return;
}

/* form scene vertex-array for orthog. projection */

loadv()
{   int i,j;
    for(i=0;i<=N-1;i++)
       for(j=0;j<=1;j++)
          V[i+Prv][j]=T[i][j];
    Prv=Prv+i;
    return;
}

/* transformations */

transform()
{  int i,j;
   char trn[4],rot[4],sca[4],tra[4];
space;
printf("            transform ?    ");
scanf("%s",trn);
space;
   if(trn[0]=='y')
   {
space;
printf("            rotation ?    ");
scanf("%s",rot);
      if(rot[0]=='y')
      {   rep[0]='y';
          while(rep[0]=='y')
             {
space;
printf("       insert angles of rotation\n");
printf("             Az , By , Cx\n");
space;
scanf("%f%f%f",&Az,&By,&Cx);
          rotate();
space;
space;
printf("      matrix of rotated figure(s) co-ordinates");
space;
space;
             for(i=0;i<=N-1;i++)
printf("T[%d][0]=%10.4f T[%d][1]=%10.4f T[%d][2]=%10.4f T[%d][3]=%10.4f\
n",
          i,T[i][0],i,T[i][1],i,T[i][2],i,T[i][3]);
             scr_plot();
             repeat();
          }
          store();
      }
space;
printf("           scaling ?    ");
scanf("%s",sca);
      if(sca[0]=='y')
      {   rep[0]='y';
          while(rep[0]=='y')
          {
space;
printf("       insert scaling factors\n");
printf("            ax , by , cz\n");
space;
scanf("%f%f%f",&a,&b,&c);
          sscale();
space;
space;
printf("      matrix of scaled figure(s) co-ordinates");
space;
space;
          for(i=0;i<=N-1;i++)
printf("T[%d][0]=%10.4f T[%d][1]=%10.4f T[%d][2]=%10.4f T[%d][3]=%10.4f\
n",
          i,T[i][0],i,T[i][1],i,T[i][2],i,T[i][3]);
             scr_plot();
             repeat();
          }
          store();
      }
space;
     if(lrn[0]=='n')
     {
printf("           translation ?    ");
scanf("%s",tra);
          if(tra[0]=='y')
          {   rep[0]='y';
              while(rep[0]=='y')
              {
space;
printf("       insert translating quantities\n");
printf("             lx , my , nz\n");
space;
scanf("%f%f%f",&l,&m,&n);
              translate();
space;
space;
printf("      matrix of translated figure(s) co-ordinates");
space;
```

```
space;
                for(i=0;i<=N-1;i++)
printf("T[%d][0]=%10.4f T[%d][1]=%10.4f T[%d][2]=%10.4f T[%d][3]=%10.4f\
n",
          i,T[i][0],i,T[i][1],i,T[i][2],i,T[i][3]);
                    scr_plot();
                    repeat();
                }
            }
        }
    }
    return;
}

/* matrix multiplication */

mtrxmlt(a,b,m,n,p)
    int m,n,p;
    double a[120][4],b[4][4];
    {   int i,j,k;
        for(i=0;i<=m;i++)
        {   for(j=0;j<=n;j++)
            {   d[i][j]=0;
                for(k=0;k<=p;k++)
                    d[i][j]=d[i][j]+a[i][k]*b[k][j];
            }
        }
        move();
        return;
    }

/* matrix of homogeneous coordinates */

homocoord()
{   int i;
    for(i=0;i<=N-1;i++)
       T[i][3]=1;
    return;
}

/* translation matrix */

translate()
{   int i;
printf("l=%10.4f,m=%10.4f,n=%10.4f\n",l,m,n);
space;
    for(i=0;i<=3;i++)
       Tr[i][i]=1;
    Tr[3][0]=l;
    Tr[3][1]=m;
    Tr[3][2]=n;
space;
printf("      translation matrix");
space;
space;
    for(i=0;i<=3;i++)
       printf("%    10.4f    %10.4f    %10.4f    %10.4f\n",
            Tr[i][0],Tr[i][1],Tr[i][2],Tr[i][3]);
    mtrxmlt(T,Tr,N-1,3,3);
    return;
}

/* scaling matrix */

sscale()
{   int i,j;
printf("a=%10.4f,b=%10.4f,c=%10.4f\n",a,b,c);
space;
    for(i=0;i<=3;i++)
    {   for(j=0;j<=3;j++)
          Sc[i][j]=0;
    }
    Sc[0][0]=a;
    Sc[1][1]=b;
    Sc[2][2]=c;
    Sc[3][3]=1;
space;
printf("      scaling matrix");
space;
space;
    for(i=0;i<=3;i++)
       printf("%    10.4f    %10.4f    %10.4f    %10.4f\n",
             Sc[i][0],Sc[i][1],Sc[i][2],Sc[i][3]);
    mtrxmlt(T,Sc,N-1,3,3);
    return;
}

/* rotation matrix */

rotate()
{   int i;
    double A1,A2,A3;
printf("Az=%10.4f,By=%10.4f,Cx=%10.4f\n",Az,By,Cx);
space;
    A1=Az*3.1415/180;
    A2=By*3.1415/180;
    A3=Cx*3.1415/180;
    for(i=0;i<=2;i++)
    {   Rt[i][3]=0;
        Rt[3][i]=0;
    }
        Rt[0][0]=cos(A1)*cos(A2);
        Rt[0][1]=sin(A1)*cos(A2);
        Rt[0][2]= -sin(A2);
        Rt[1][0]= -sin(A1)*cos(A3)+cos(A1)*sin(A2)*sin(A3);
        Rt[1][1]=cos(A1)*cos(A3)+sin(A1)*sin(A2)*sin(A3);
        Rt[1][2]=cos(A2)*sin(A3);
        Rt[2][0]=sin(A1)*sin(A3)+cos(A1)*sin(A2)*cos(A3);
        Rt[2][1]= -cos(A1)*sin(A3)+sin(A1)*sin(A2)*cos(A3);
        Rt[2][2]=cos(A2)*cos(A3);
        Rt[3][3]=1;
space;
printf("      rotation matrix");
space;
space;
    for(i=0;i<=3;i++)
       printf("    %10.4f    %10.4f    %10.4f    %10.4f\n",
             Rt[i][0],Rt[i][1],Rt[i][2],Rt[i][3]);
   mtrxmlt(T,Rt,N-1,3,3);
   return;
}

/* central projection matrix */

c_project()
{   int i,j;
space;
    for(i=0;i<=3;i++)
       for(j=0;j<=3;j++)
          Pr[i][j]=0;
    Pr[0][0]=1;
    Pr[1][1]=1;
```

```
     Pr[2][2]=1;
     Pr[2][3]= -1/(Zo-Zi);
     Pr[3][3]= Zo/(Zo-Zi);
space;
printf("      central projection matrix");
space;
space;
    for(i=0;i<=3;i++)
printf("Pr[%d][0]=%10.4f Pr[%d][1]=%10.4f Pr[%d][2]=%10.4f Pr[%d][3]=%10.4f\n",
        i,Pr[i][0],i,Pr[i][1],i,Pr[i][2],i,Pr[i][3]);
    mtrxmlt(T,Pr,N-1,3,3);
    move();
space;
space;
printf("      matrix of projected figure(s) co-ordinates");
space;
space;
    for(i=0;i<=N-1;i++)
    {  for(j=0;j<=1;j++)
          T[i][j]=T[i][j]/T[i][3];
printf("T[%d][0]=%10.4f T[%d][1]=%10.4f\n",
        i,T[i][0],i,T[i][1]);
    }
    return;
}

/* form transformed vertex-array */

move()
{   int i,j;
    for(i=0;i<=N-1;i++)
    {   for(j=0;j<=3;j++)
           T[i][j]=d[i][j];
    }
    return;
}

/* orthographic projection front window clipping */

clip()
{   int i;
    double Xvp,Yvp,Zvp;
    Zvp=2*Zu-Zi;
    for(i=0;i<=N-1;i++)
    {   Xvp=Pw;
        Yvp=Ph;
        if(T[i][0]< -Xvp||T[i][0]>Xvp)
        {
printf("x coordinate of vertex V[%d] =%5.2f outside viewing parallelepiped\n",
        i,T[i][0]);
           space;
        }
        if(T[i][1]< -Yvp||T[i][1]>Yvp)
        {
printf("y coordinate of vertex V[%d] =%5.2f outside viewing parallelepiped\n",
        i,T[i][1]);
           space;
        }
        if(T[i][2]<Zi||T[i][2]>Zvp)
        {
printf("z coordinate of vertex V[%d] =%5.2f outside viewing parallelepiped\n",
        i,T[i][2]);
           space;
        }
    }
    return;
}

/* central projection front window clipping */

c_clip()
{   int i;
    double Xvp,Yvp,Zvp;
    Zvp=2*Zu-Zi;
    for(i=0;i<=N-1;i++)
    {   Xvp=(Zu-Zo+T[i][2])*Pw/(Zi-Zo);
        Yvp=(Zu-Zo+T[i][2])*Ph/(Zi-Zo);
        if(T[i][0]< -Xvp||T[i][0]>Xvp)
        {
printf("x coordinate of vertex V[%d] =%5.2f outside the viewing pyramid\n",
        i,T[i][0]);
           space;
        }
        if(T[i][1]< -Yvp||T[i][1]>Yvp)
        {
printf("y coordinate of vertex V[%d] =%5.2f outside the viewing pyramid\n",
        i,T[i][1]);
           space;
        }
        if(T[i][2]<Zi||T[i][2]>Zvp)
        {
printf("z coordinate of vertex V[%d] =%5.2f outside the viewing pyramid\n",
        i,T[i][2]);
           space;
        }
    }
    return;
}

/* find hidden faces */

hidden(Hl,Hm,Hvis,Hinv,Hprfc)
    int *Hl,*Hm,*Hprfc,Hvis[120],Hinv[120];
{   int i,j,k,l,m,p,r,I,J,K;
    double A[3][3],C[3],COSA;
    l= *Hl;
    m= *Hm;
    p=0;
    r=0;
    for(i=0;f[i][0]!=END;i++)
    {   K=f[i][1]-1;
        for(j=0;j<2;j++)
        {   I=f[i][j]-1;
            J=f[i][j+1]-1;
            for(k=0;k<=2;k++)
               A[j][k]=T[J][k]-T[I][k];
        }
        C[0]=T[K][0]-0.0;
        C[1]=T[K][1]-0.0;
        C[2]=T[K][2]-Zo;
        COSA=(A[0][1]*A[1][2]-A[0][2]*A[1][1])*C[0]+
             (A[0][2]*A[1][0]-A[0][0]*A[1][2])*C[1]+
             (A[0][0]*A[1][1]-A[0][1]*A[1][0])*C[2];
 printf("COSA = %12.8f\n",COSA);
        if(COSA<0)
```

```
        {   Hvis[l]=i+*Hprfc;
            l=l+1;
            vis[p]=i;
            p=p+1;
        }
        else
        {   Hinv[m]=i+*Hprfc;
            m=m+1;
            inv[r]=i;
            r=r+1;
        }
    }
    *Hl=l;
    *Hm=m;
    vis[p]= END;
    inv[r]= END;
space;
printf("       visible faces\n");
space;
    for(i=0;i<=l-1;i++)
printf("VIS[%d] = %d\n",i,Hvis[i]);
space;
printf("       invisible faces\n");
space;
    for(i=0;i<=m-1;i++)
printf("INV[%d] = %d\n",i,Hinv[i]);
    return;
}

/* form single-vertx list */

vertlist()
{   int i,j,k,Q,S,T;
    S=vis[0];
    for(i=0;f[S][i+1]>LAST;i++)
        s[i]=f[S][i];
    P=i;
    for(i=0;vis[i]>END;i++)
        Q=i;
    for(i=1;i<=Q;i++)
    {   R=vis[i];
        for(j=0;f[R][j+1]>LAST;j++)
        {   T=0;
            for(k=0;k<P;k++)
            {   if(s[k]==f[R][j])
                T=T+1;
            }
            if(T==0)
            {   s[P]=f[R][j];
                P=P+1;
            }
        }
    }
    return;
}

/* create prolog view-files (learning) */

v_prolog(file)
    char file[40];
{   int i,j,I,J;
    prlg=fopen(file,"w");
fprintf(prlg,"%s(",objname);
for(i=0;s[i+1]!=END;i++)
    fprintf(prlg,"v%d,",s[i]);
fprintf(prlg,"v%d),\n",s[i]);
fprintf(prlg,"[conn(v%d,v%d,2),\n",vln[0][0],vln[0][1]);
    for(i=1;vln[i+1][0]>END;i++)
    {   I=vln[i][0];
        J=vln[i][1];
fprintf(prlg," conn(v%d,v%d,2),\n",I,J);
    }
fprintf(prlg," conn(v%d,v%d,2)],\n",vln[i][0],vln[i][1]);
    positive;
    fclose(prlg);
    return;
}

/* create prolog file <scene> with conn's */

prolog()
{   int i,j,I,J;
    FILE *prlg;
    prlg=fopen("scene","w");
    for(i=0;Vln[i][0]>END;i++)
    {   I=Vln[i][0];
        J=Vln[i][1];
fprintf(prlg,"conn(v%d,v%d,2).\n",I,J);
    }
    fclose(prlg);
    return;
}

/* store vertex coordinates in file <coord> */

coord()
{   int i;
    oupt=fopen("coord","w");
    for(i=1;(i<120&&Vc[i-1][0]!=End);i++)
fprintf(oupt,"%f %f\n",Vc[i][0],Vc[i][1]);
    fclose(oupt);
    return;
}

/* form list of invisible lines */

inv_lines(giln,gvln,ginv,gf)
    int giln[120][2],gvln[120][2],ginv[120],gf[120][10];
{   int i,j,k,I,J,X,Y,Z;
    giln[i][0]= LAST;
    Y=0;
    for(i=0;ginv[i]>END;i++)
    {   Z=ginv[i];
        for(j=0;gf[Z][j+1]>LAST;j++)
        {   X=0;
            for(k=0;gvln[k][0]>END;k++)
                if(gvln[k][0]==gf[Z][j])
                    if(gvln[k][1]==gf[Z][j+1])
                        X=X+1;
            if(X==0)
            {   for(k=0;gvln[k][0]>END;k++)
                    if(gvln[k][1]==gf[Z][j])
                        if(gvln[k][0]==gf[Z][j+1])
                            X=X+1;
                if(X==0)
                {   if(giln[0][0]==LAST)
                    {   giln[Y][0]=gf[Z][j];
                        giln[Y][1]=gf[Z][j+1];
                        Y=Y+1;
                        giln[Y][0]= END;
                    }
                    for(k=0;giln[k][0]>END;k++)
```

```
                    if(giln[k][0]==gf[Z][j])
                       if(giln[k][1]==gf[Z][j+1])
                          X=X+1;
                 if(X==0)
                 {   for(k=0;giln[k][0]>END;k++)
                        if(giln[k][1]==gf[Z][j])
                           if(giln[k][0]==gf[Z][j+1])
                              X=X+1;
                     if(X==0)
                     {   giln[Y][0]=gf[Z][j];
                         giln[Y][1]=gf[Z][j+1];
                         Y=Y+1;
                         giln[Y][0]= END;
                     }
                 }
              }
           }
        }
    }
    if(giln[0][0]==LAST)
       giln[0][0]=END;
space;
printf("      invisible lines\n");
space;
    for(i=0;i<=Y;i++)
printf("Iln[%d][0] = %d  ,  Iln[%d][1] = %d\n",i,giln[i][0],i,giln[i][1]
);
    return;
}

/* form list of visible lines */

vis_lines(gvln,gvis,gf)
    int gvln[120][2],gvis[120],gf[120][10];
{   int i,j,I,J,k,X,Y,Z;
    Z=gvis[0];
    for(i=0;gf[Z][i+1]>LAST;i++)
    {   gvln[i][0]=gf[Z][i];
        gvln[i][1]=gf[Z][i+1];
    }
    gvln[i][0]=END;
    Y=i;
    for(i=1;gvis[i]>END;i++)
    {   Z=gvis[i];
        for(j=0;gf[Z][j+1]>LAST;j++)
        {   X=0;
            for(k=0;gvln[k][0]>END;k++)
               if(gvln[k][0]==gf[Z][j])
                  if(gvln[k][1]==gf[Z][j+1])
                     X=X+1;
            if(X==0)
            {   for(k=0;gvln[k][0]>END;k++)
                   if(gvln[k][1]==gf[Z][j])
                      if(gvln[k][0]==gf[Z][j+1])
                         X=X+1;
                if(X==0)
                {   gvln[Y][0]=gf[Z][j];
                    gvln[Y][1]=gf[Z][j+1];
                    Y=Y+1;
                    gvln[Y][0]= END;
                }
            }
        }
    }
 space;
 printf("      visible lines\n");
space;
    for(i=0;i<=Y;i++)
printf("Vln[%d][0] = %d  ,  Vln[%d][1] = %d\n",i,gvln[i][0],i,gvln[i][1]
);
    return;
}

/* plot scene */

plotout()
{   char scr[4],pap[4];
space;
printf("            on screen ?    ");
scanf("%s",scr);
space;
    if(scr[0]=='y')
       screen_plot();
space;
printf("            on paper ?    ");
scanf("%s",pap);
space;
    if(pap[0]=='y')
       paper_plot();
    return;
}

/* plot on paper */

paper_plot()
{   char opr[4],cpr[4],hdn[4];
    int nstat;
    ploti();
space;
printf("            orthographic projection ?    ");
scanf("%s",opr);
space;
    if(opr[0]=='y')
    {   set_pap_plot();
        newpen(BLACK);
        penspd(2);
        symbol(-6.5,9.0,0.5,title1,0.0,25);
        if(lrn[0]=='y')
        {   newpen(BLACK);
            penspd(2);
            pap_view_symbol();
            paper_views_plot(BLUE,Vln,V);
        }
        else
        {
            paper_scene_plot(BLUE,Vln,V);
        }
space;
printf("            hidden lines ?    ");
scanf("%s",hdn);
space;
        if(hdn[0]=='y')
        {   daslna(3,0.25);
            if(lrn[0]=='y')
               paper_views_plot(RED,Iln,Vc);
            else
               paper_scene_plot(RED,Iln,Vc);
        }
    }
space;
printf("            central projection ?    ");
scanf("%s",cpr);
```

```
space;
    if(cpr[0]=='y')
    {   if(opr[0]=='y')
            advf(nstat);
        set_pap_plot();
        newpen(BLACK);
        penspd(2);
        symbol(-5.0,9.0,0.5,title2,0.0,20);
        if(lrn[0]=='y')
        {   newpen(BLACK);
            penspd(2);
            pap_view_symbol();
            paper_views_plot(RED,Vln,Vc);
        }
        else
            paper_scene_plot(RED,Vln,Vc);
space;
printf("            hidden lines ?    ");
scanf("%s",hdn);
space;
        if(hdn[0]=='y')
        {   daslna(3,0.25);
            if(lrn[0]=='y')
                paper_views_plot(RED,Iln,Vc);
            else
                paper_scene_plot(RED,Iln,Vc);
        }
space;
    }
    plote();
    return;
}

/* set up  hp plotter */

set_pap_plot()
{   limit(1.0,13.0,1.0,9.0);
    mscale(50.0,30.0);
    locate(0.0,210.0,0.0,140.0);
    mapuu(-15.0,15.0,-10.0,10.0);
    pap_ax_pl();
    return;
}

/* plot a frame and a pair of axes on paper */

pap_ax_pl()
{   char axe[4];
    penspd(10);
    newpen(BLACK);
    cframe();
space;
printf("            axes ?    ");
scanf("%s",axe);
space;
    if(axe[0]=='y')
    {   newpen(BLACK);
        laxes(0.5,0.5,-15.0,-10.0,-4,4,1.0);
    }
}

/* plot alternative views on paper (learning) */

paper_views_plot(pen,K,Sv)
   int K[120][2],pen;
   double Sv[120][2];
{   int i,j,I,J,p[2];
    newpen(pen);
    pehspd(2);
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        plot(Sv[I][0],Sv[I][1],3);
        plot(Sv[J][0],Sv[J][1],2);
    }
    newpen(BLACK);
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        p[0]=view[i];
        int_to_char(p);
        for(j=0;Ch[j]!='*';j++);
        Ch[j]=' ';
        Ch[j+1]=' ';
        symbol(Sv[I][0]+0.5,Sv[I][1]+0.5,0.3, Ch ,0.0,3);
    }
    return;
}

/* plot scene on paper */

paper_scene_plot(pen,K,Sv)
   int K[120][2],pen;
   double Sv[120][2];
{   int i,j,I,J,p[2];
    newpen(pen);
    penspd(2);
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        plot(Sv[I][0],Sv[I][1],3);
        plot(Sv[J][0],Sv[J][1],2);
    }
    newpen(BLACK);
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        p[0]=K[i][0];
        int_to_char(p);
        for(j=0;Ch[j]!='*';j++);
        Ch[j]=' ';
        Ch[j+1]=' ';
        symbol(Sv[I][0]+0.5,Sv[I][1]+0.5,0.3, Ch ,0.0,3);
    }
    return;
}

/* print alternative  view-names on paper (learning) */

pap_view_symbol()
{   symbol(-11.00,-0.3,0.3,objname,0.0,20);
    symbol(-10.75,-9.6,0.3,  bac  ,0.0,4);
    symbol( -0.40,-0.3,0.3,  top  ,0.0,3);
    symbol( -1.00,-9.6,0.3,  bot  ,0.0,6);
    symbol(  9.50,-0.3,0.3,  lef  ,0.0,4);
    symbol(  9.20,-9.6,0.3,  rig  ,0.0,5);
}

/* repeat transformation */

repeat()
{   space;
```

```
 printf("            repeat ?     ");
 scanf("%s",rep);
     if(rep[0]=='y')
        insert();
     return;
 }

/* plot 3-D figure  on screen during transformaton */

scr_plot()
{   int i,j,l,m;
    int Srfc,Siln[120][2],Svln[120][2],Svis[120],Sinv[120];
    double Sv[120][2];
    l=0;
    m=0;
    Srfc=0;
    for(i=0;i<120;i++)
    {   Svis[i]=0;
        Sinv[i]=0;
        for(j=0;j<2;j++)
        {   Svln[i][j]=0;
            Siln[i][j]=0;
        }
    }
    hidden(&l,&m,Svis,Sinv,&Srfc);
    Svis[l]=END;
    Sinv[m]=END;
    vis_lines(Svln,Svis,f);
    inv_lines(Siln,Svln,Sinv,f);
    dtx5a();
    piccle();
    scr_frame();
    for(i=0;i<120;i++)
       for(j=0;j<2;j++)
          Sv[i+1][j]=T[i][j];
    screen_invis_plot(VIOLET,Siln,Sv);
    movto2(7.2*4+109, 8.8*4+95);
    chahol("hidden lines*.");
    screen_scene_plot(BROWN,Svln,Sv);
    chamod();
    devend();
    return;
}

/* plot on screen */

screen_plot()
{   char opr[4],cpr[4],hdn[4];
    dtx5a();
    piccle();
space;
printf("            orthographic projection ?    ");
scanf("%s",opr);
space;
    if(opr[0]=='y')
    {
        scr_frame();
        movto2(-6.5*4+109,10.3*4+95);
        pensel(GREEN);
        chahol("orthographic projection*.");
space;
printf("            hidden lines ?    ");
scanf("%s",hdn);
space;
            if(hdn[0]=='y')
        {   screen_invis_plot(BLUE,Iln,V);
            movto2(-14.5*4+109, -11.3*4+95);
            chahol("hidden lines*.");
        }
        if(lrn[0]=='y')
        {   screen_views_plot(GREEN,Vln,V);
            chamod();
            scr_view_symbol();
        }
        else
        {   screen_scene_plot(GREEN,Vln,V);
            chamod();
         }
    }
space;
printf("            central projection ?    ");
scanf("%s",cpr);
space;
    if(cpr[0]=='y')
    {   if(opr[0]=='y')
           piccle();
        scr_frame();
        movto2(-5.0*4+105,10.3*4+95);
        pensel(BROWN);
        chahol("central projection*.");
space;
printf("            hidden lines ?    ");
scanf("%s",hdn);
space;
        if(hdn[0]=='y')
        {   screen_invis_plot(BLUE,Iln,Vc);
            movto2(-14.5*4+109, -11.3*4+95);
            chahol("hidden lines*.");
        }
        if(lrn[0]=='y')
        {   screen_views_plot(BROWN,Vln,Vc);
            chamod();
            scr_view_symbol();
        }
        else
        {   screen_scene_plot(BROWN,Vln,Vc);
            chamod();
         }
    }
    devend();
    return;
}

/* plot frame on screen */

scr_frame()
{   movto2(49.0, 55.0);
    linto2( 169.0,55.0);
    linto2( 169.0, 135.0);
    linto2(49.0, 135.0);
    linto2( 49.0, 55.0);
    return;
}

/* plot altrnative views on screen (learning) */

screen_views_plot(pen,K,Sv)
   int K[120][2],pen;
   double Sv[120][2];
{   int i,j,I,J,p[2];
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
```

```
        J=K[i][1];
        pensel(VIOLET);
        p[0]=view[i];
        int_to_char(p);
        movto2(4*Sv[I][0]+108.7,4*Sv[I][1]+95.3);
        chahol(Ch);
        pensel(pen);
        movto2(4*Sv[I][0]+109.0,4*Sv[I][1]+95.0);
        symbol(CIRCLE);
        linto2(4*Sv[J][0]+109.0,4*Sv[J][1]+95.0);
    }
    return;
}

/* plot scene on screen */

screen_scene_plot(pen,K,Sv)
   int K[120][2],pen;
   double Sv[120][2];
{   int i,j,I,J,p[2];
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        pensel(VIOLET);
        p[0]=K[i][0];
        int_to_char(p);
        movto2(4*Sv[I][0]+108.7,4*Sv[I][1]+95.3);
        chahol(Ch);
        pensel(pen);
        movto2(4*Sv[I][0]+109.0,4*Sv[I][1]+95.0);
        symbol(CIRCLE);
        linto2(4*Sv[J][0]+109.0,4*Sv[J][1]+95.0);
    }
    return;
}

/* plot scene invisible lines on screen */

screen_invis_plot(pen,K,Sv)
   int K[120][2],pen;
   double Sv[120][2];
{   int i,I,J;
    pensel(pen);
    for(i=0;K[i][0]!=END;i++)
    {   I=K[i][0];
        J=K[i][1];
        movto2(4*Sv[I][0]+109.0,4*Sv[I][1]+95.0);
        linto2(4*Sv[J][0]+109.0,4*Sv[J][1]+95.0);
    }
    return;
}

/* print names of alternative views (learning) */

scr_view_symbol()
{   pensel(VIOLET);
    movto2(-11.00*4+109,-0.60*4+95);
    chahol("front*.");
    movto2(-10.75*4+109,-9.80*4+95);
    chahol("back*.");
    movto2( -0.40*4+109,-0.60*4+95);
    chahol("top*.");
    movto2 (-1.00*4+109,-9.80*4+95);
    chahol("bottom*.");
    movto2(  9.50*4+109,-0.60*4+95);
    chahol("left*.");
    movto2(  9.20*4+109,-9.80*4+95);
    chahol("right*.");
    return;
}

/* convert integer to string */

int_to_char(intg)
   int intg[2];
{  int i,h,d,u;
   if(intg[0]>99)
   {
      h=intg[0]/100;
      Ch[0]=h+'0';
      d=(intg[0]-h*100)/10;
      Ch[1]=d+'0';
      u=(intg[0]-h*100)-d*10;
      Ch[2]=u+'0';
      Ch[3]='*';
      Ch[4]='.';
   }
   else if(intg[0]>9&&intg[0]<100)
   {
      d=intg[0]/10;
      Ch[0]=d+'0';
      u=intg[0]-d*10;
      Ch[1]=u+'0';
      Ch[2]='*';
      Ch[3]='.';
   }
   else
   {
      u=intg[0];
      Ch[0]=u+'0';
      Ch[1]='*';
      Ch[2]='.';
   }
   return;
}
```

```
                    /* link */

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

/* end markers */

#define LAST   0
#define END   -1
#define End  100.0
#define space  printf("\n")
int Pl[120][10],Fg[120][10],P[10];
char CH[5];
double Vc[120][2];
double A,B,C,X[120][4],Y[120][4],Qa[4],Ta[3],Ag[120],An[120][4];
double sqrt(),acos(),fabs();
int Vrx,Hafpln,Insd,Poin,Yes;
FILE *inpt1,*inpt2,*prlg;

main()
{   int i,j,k,l,m,n,r,p,Two,K,N,Newfig[10];
    double Xq[4],Yq[4];

/* create file <spec_feat> */

    prlg=fopen("spec_feat","w");
    fclose(prlg);
    inpt2=fopen("coord","r");
    for(i=1;i<=120;i++)
       for(j=0;(j<2&&fscanf(inpt2,"%f",&Vc[i][j])&&Vc[i][j]!=End);j++);
    fclose(inpt2);
    inpt1=fopen("vertex","r");
    for(i=0;(fscanf(inpt1,"%d",&Fg[i][0]))&&(Fg[i][0]!=END);i++)
    {
printf("j = 0\n");
printf("Fg[%d][0] = v%d\n",i,Fg[i][0]);
         for(j=1;(fscanf(inpt1,"%d",&Fg[i][j]))&&(Fg[i][j]!=LAST);j++)
         {
printf("j = %d\n",j);
printf("Fg[%d][%d] = v%d\n",i,j,Fg[i][j]);
         }
    }

/* form list of 3-D figures */

    Fg[i][0]=END;
    fclose(inpt1);
    for(k=0;k<120;k++)
       for(m=0;m<5;m++)
          Pl[k][m]=0;
    N=l=0;
    for(r=0;Fg[r][0]!=END;r++)
    {   if(Fg[r][0]!=LAST)
        {   for(j=0;Fg[r][j]!=LAST;j++)
            {   Pl[l][j]=Fg[r][j];
                Fg[r][j]=0;
            }
            p=l;
            Newfig[N]=p;
            l=l+1;
            N=N+1;
            for(k=p;Pl[k][0]!=LAST;k++)
            {   for(i=0;Fg[i][0]!=END;i++)
                {   Two=0;
                    for(j=0;Fg[i][j]!=LAST;j++)
                    {   for(m=0;Pl[k][m]!=LAST&&Fg[i][j]!=Pl[k][m];m++);
                            if(Pl[k][m]!=LAST)
                                Two=Two+1;
                    }
                    if(Two>=2)
                    {    for(n=0;Fg[i][n]!=LAST;n++)
                         {    Pl[l][n]=Fg[i][n];
                              Fg[i][n]=0;
                         }
                         l=l+1;
                    }

                }
            }
            l=l+1;
        }
    }
    Newfig[N]=END;

/* for each figure relate coordinates to vertices */

    for(r=0;Newfig[r]!=END;r++)
    {   p=Newfig[r];
        for(i=0;i<120;i++)
           for(j=0;j<5;j++)
              An[i][j]=0;
        for(i=p;Pl[i][0]!=LAST;i++)
        {    for(j=0;Pl[i][j]!=LAST;j++)
             {
printf("Pl[%d][%d] = v%d\n",i,j,Pl[i][j]);
                 for(k=1;(Pl[i][j]!=k)&&(k<=120);k++);
                 X[i][j]=Vc[k][0];
                 Y[i][j]=Vc[k][1];
printf("X[%d][%d] = %5.2f  Y[%d][%d] = %5.2f\n",i,j,X[i][j],i,j,Y[i][j]);
             }
space;
        }
        for(i=p;Pl[i][0]!=LAST;i++)
        {    for(j=0;(Pl[i][j]!=LAST&&j<10);j++);
             if(j>4)
             {
printf("figure(");
                for(j=0;(Pl[i][j]!=LAST&&j<10);j++)
printf("v%d",Pl[i][j]);
printf(")   :   face with more than 4 vertices\n");
space;
             }

/* solve quadrilateral */

           else if(j==4)
                {    for(k=0;k<4;k++)
                     {    Xq[k]=X[i][k];
                          Yq[k]=Y[i][k];
                     }
                     quadrila(Xq,Yq);
                     point_in_fig(i);
                     if(PoIn==0)
                     {
printf("quadrilateral(v%d,v%d,v%d,v%d)\n",Pl[i][0],Pl[i][1],Pl[i][2],Pl[i][3]);
space;
                          for(k=0;k<4;k++)
                          {    An[i][k]=Qa[k];
                               if(Qa[k]>180)
                               {    prol_non_convex(i);
```

```
printf("v%d = %5.2f",Pl[i][k],Qa[k]);
printf("  :  non-convex\n");
                    }
                    else
printf("v%d = %5.2f\n",Pl[i][k],Qa[k]);
                }
            }
        }

/* solve triangle */

      else if(j==3)
          {   triangle(X[i][0],Y[i][0],X[i][1],Y[i][1],X[i][2],Y[i][2]
);
              point_in_fig(i);
              if(PoIn==0)
              {
printf("triangle(v%d,v%d,v%d)\n",Pl[i][0],Pl[i][1],Pl[i][2]);
space;
                  for(k=0;k<3;k++)
                  {   An[i][k]=Ta[k];
printf("v%d = %5.2f\n",Pl[i][k],Ta[k]);
                  }
              }
          }
space;
        }
        single_vrtx(p);
        cntr_angl(p);
    }
space;
exit(0);
}

/* calculate angles of triangle */

triangle(x0,y0,x1,y1,x2,y2)
    double x0,x1,x2,y0,y1,y2;
{   double ax,bx,cx,ay,by,cy;
    float CosA,CosB,a,b,c;
    int i;
    ax=x2-x1;ay=y2-y1;
    bx=x0-x2;by=y0-y2;
    cx=x1-x0;cy=y1-y0;
    a=sqrt(ax*ax+ay*ay);
    b=sqrt(bx*bx+by*by);
    c=sqrt(cx*cx+cy*cy);
    CosA=(b*b+c*c-a*a)/(2*b*c);
    CosB=(a*a+c*c-b*b)/(2*a*c);
    Ta[0]=A=(180/3.1416)*acos(CosA)+0.005;
    Ta[1]=B=(180/3.1416)*acos(CosB)+0.005;
    Ta[2]=C=180-A-B;
    return;
}

/* 'half-plane' test */

half_plane(x,y,x1,y1,x2,y2)
    double x,y,x1,y1,x2,y2;
{   double a,b;
    a=(x-x1)*(y2-y1);
    b=(x2-x1)*(y-y1);
    if(a>b)
       Hafpln=1;
    else if(a<b)
       Hafpln= -1;
    else Hafpln=0;
    return;
}

/* 'point inside triangle' test */

inside(x,y,x1,y1,x2,y2,x3,y3)
    double x,y,x1,y1,x2,y2,x3,y3;
{   int Hp1,Hp2;
    Insd=0;
    half_plane(x,y,x1,y1,x2,y2);
    Hp1=Hafpln;
    if(Hp1==0)
    {   in_line(x,y,x1,y1,x2,y2);
        if(Yes==1)
        {   Insd=1;
            return;
        }
    }
    half_plane(x3,y3,x1,y1,x2,y2);
    Hp2=Hafpln;
    if(Hp1!=Hp2)
       return;
    half_plane(x,y,x2,y2,x3,y3);
    Hp1=Hafpln;
    if(Hp1==0)
    {   in_line(x,y,x2,y2,x3,y3);
        if(Yes==1)
        {   Insd=1;
            return;
        }
    }
    half_plane(x1,y1,x2,y2,x3,y3);
    Hp2=Hafpln;
    if(Hp1!=Hp2)
       return;
    half_plane(x,y,x3,y3,x1,y1);
    Hp1=Hafpln;
    if(Hp1==0)
    {   in_line(x,y,x3,y3,x1,y1);
        if(Yes==1)
        {   Insd=1;
            return;
        }
    }
    half_plane(x2,y2,x3,y3,x1,y1);
    Hp2=Hafpln;
    if(Hp1==Hp2)
       Insd=1;
    return;
}

/* calculate angles of quadrilateral */

quadrila(x,y)
     double x[],y[];
  {  double min1,min2,min,Q1[4],Q2[4];
     int i,K,L,M,N;
     for(i=0;i<4;i++)
        Q1[i]=Q2[i]=0;
     min1=(x[0]<x[1]?x[0]:x[1]);
     min2=(x[2]<x[3]?x[2]:x[3]);
     min=(min1<min2?min1:min2);
     for(K=0;(K<4&&x[K]!=min);K++);
     L=(K<3?K+1:0);
     M=(K<1?  3:K-1);
```

```
        N=(K<2?K+2:K-2);
        inside(x[N],y[N],x[L],y[L],x[K],y[K],x[M],y[M]);
        if(Insd==0)
        {   triangle(x[M],y[M],x[K],y[K],x[L],y[L]);
            Q1[M]=A; Q1[K]=B; Q1[L]=C;
            triangle(x[M],y[M],x[N],y[N],x[L],y[L]);
            Q2[M]=A; Q2[N]=B; Q2[L]=C;
        }
        else
        {   triangle(x[N],y[N],x[M],y[M],x[K],y[K]);
            Q1[N]=A; Q1[M]=B; Q1[K]=C;
            triangle(x[N],y[N],x[L],y[L],x[K],y[K]);
            Q2[N]=A; Q2[L]=B; Q2[K]=C;
        }
        for(i=0;i<4;i++)
            Qa[i]=Q1[i]+Q2[i];
        return;
    }

    /* form single-vertex list */

    single_vrtx(s)
        int s;
    {   int i,j,k,p,t;
        for(i=0;Pl[s][i]!=LAST;i++)
            P[i]=Pl[s][i];
        p=i;
        for(i=s+1;Pl[i][0]!=LAST;i++)
        {   for(j=0;Pl[i][j]!=LAST;j++)
            {   t=0;
                for(k=0;k<p;k++)
                {   if(P[k]==Pl[i][j])
                        t=t+1;
                }
                if(t==0)
                {   P[p]=Pl[i][j];
                    p=p+1;
                }
            }
        }
        P[p]=LAST;
        return;
    }

/* calcuate contour angles of 3-D figure */

    cntr_angl(s)
        int s;
    {   int i,j,k,Cv,Cntr;
        for(i=0;i<10;i++)
            Ag[i]=0;
        Cntr=0;
        for(i=0;P[i]!=LAST;i++)
        {   for(j=s;Pl[j][0]!=LAST;j++)
            {   for(k=0;Pl[j][k]!=LAST;k++)
                {   if(P[i]==Pl[j][k])
                        Ag[i]=Ag[i]+An[j][k];
                }
            }
        }
printf("angles of 2-D figure contour\n");
space;
        for(i=0;Ag[i]!=0;i++)
        {
            if(Ag[i]>180&&fabs(Ag[i]-360)>0.05)
            {
printf("v%d = %5.2f",P[i],Ag[i]);
printf("  :  concave 3-D figure\n");
                Cv=i;
                Cntr=1;
                }
        else if(fabs(Ag[i]-360)>0.05)
printf("v%d = %5.2f\n",P[i],Ag[i]);
        }
        if(Cntr==1)
            prlg_concave(Cv);
        return;
    }

/* 'point inside triangle/quadrilateral' test */

point_in_fig(l)
    int l;
{   int j,k,m,K,L,M,N;
    double min1,min2,min;
    Poin=0;
    for(k=1;Pl[k][0]!=LAST;k++)
    {   for(m=0;(k!=l)&&Pl[k][m]!=LAST;m++)
        {   for(j=0;Pl[l][j]!=LAST&&Pl[l][j]!=Pl[k][m];j++);
            if(Pl[l][j]==LAST)
            {   if(Pl[l][3]==LAST)
                {   inside(X[k][m],Y[k][m],X[l][0],Y[l][0],
                           X[l][1],Y[l][1],X[l][2],Y[l][2]);
                    if(Insd==1)
                    {   prol_poin_fig(l);
printf("point inside triangle(v%d,v%d,v%d)\n",Pl[l][0],
                           Pl[l][1],Pl[l][2]);
                        Poin=1;
                        return;
                    }
                }

/* find left most vertex */

                else
                {   min1=(X[l][0]<X[l][1]?X[l][0]:X[l][1]);
                    min2=(X[l][2]<X[l][3]?X[l][2]:X[l][3]);
                    min=(min1<min2?min1:min2);
                    for(K=0;(K<4&&X[l][K]!=min);K++);
                    L=(K<3?K+1:0);
                    M=(K<1?  3:K-1);
                    N=(K<2?K+2:K-2);

/* non-convex quadril */

                    inside(X[l][N],Y[l][N],X[l][M],Y[l][M],
                           X[l][K],Y[l][K],X[l][L],Y[l][L]);
                    if(Insd==1)
                    {   inside(X[k][m],Y[k][m],X[l][K],Y[l][K],
                               X[l][L],Y[l][L],X[l][N],Y[l][N]);
                        if(Insd==1)
                        {   prol_poin_fig(l);
printf("point inside  quadrilateral(v%d,v%d,v%d,v%d)\n",Pl[l][0],
                                   Pl[l][1],Pl[l][2],Pl[l][3]);
                            Poin=1;
                            return;
                        }
                        else
                        {   inside(X[k][m],Y[k][m],X[l][N],Y[l][N],
                                   X[l][M],Y[l][M],X[l][K],Y[l][K]);
                            if(Insd==1)
                            {   prol_poin_fig(l);
```

```
printf("point inide quadrilateral(v%d,v%d,v%d,v%d)\n",Pl[l][0],
                                  Pl[l][1],Pl[l][2],Pl[l][3]);
                                  Poin=1;
                                  return;
                              }
                          }
                      }

/* convex quadril */

                      else
                      {   inside(X[k][m],Y[k][m],X[l][M],Y[l][M],
                                  X[l][K],Y[l][K],X[l][L],Y[l][L]);
                          if(Insd==1)
                          {   prol_poin_fig(l);
printf("point inside quadrilateral(v%d,v%d,v%d,v%d)\n",Pl[l][0],
                                  Pl[l][1],Pl[l][2],Pl[l][3]);
                              Poin=1;
                              return;
                          }
                          else
                          {   inside(X[k][m],Y[k][m],X[l][M],Y[l][M],
                                  X[l][L],Y[l][L],X[l][N],Y[l][N]);
                              if(Insd==1)
                              {   prol_poin_fig(l);
printf("point inside quadrilateral(v%d,v%d,v%d,v%d)\n",Pl[l][0],
                              Pl[l][1],Pl[l][2],Pl[l][3]);
                                  Poin=1;
                                  return;
                              }
                          }
                      }
                  }
              }
          }
      }
      return;
}

/* insert non-convexity predicate in file <spec_feat> */

prol_non_convex(k)
    Int k;
{   prlg=fopen("spec_feat","a");
fprintf(prlg,"non_convex(v%d,v%d,v%d,v%d).\n",
                Pl[k][0],Pl[k][1],Pl[k][2],Pl[k][3]);
    fclose(prlg);
    return;
}

/* insert non-conv-contour predicate in file <spec_feat> */

prlg_concave(k)
    Int k;
{   prlg=fopen("spec_feat","a");
fprintf(prlg,"non_convex_contour_angle(v%d).\n",P[k]);
    fclose(prlg);
    return;
}

/* insert point-in-figure predicate in file <spec_feat> */

prol_poin_fig(k)
    Int k;
{   prlg=fopen("spec_feat","a");
    if(Pl[k][3]!=LAST)
fprintf(prlg,"point_in_qul(v%d,v%d,v%d,v%d).\n",
        Pl[k][0],Pl[k][1],Pl[k][2],Pl[k][3]);
    else
fprintf(prlg,"point_in_trn(v%d,v%d,v%d).\n",
        Pl[k][0],Pl[k][1],Pl[k][2]);
    fclose(prlg);
    return;
}

/* 'point on line' test */

in_line(a,b,a1,b1,a2,b2)
    double a,b,a1,b1,a2,b2;
{   double mina,minb,maxa,maxb;
    Yes=0;
    mina=(a1<a2?a1:a2);
    minb=(b1<b2?b1:b2);
    maxa=(a1>a2?a1:a2);
    maxb=(b1>b2?b1:b2);
    if((a<maxa&&a>mina)||(b<maxb&&b>minb))
       Yes=1;
    return;
}
```

APPENDIX 2

2.1 THE 'PROLOG' PROGRAMS

The version of PROLOG used in this project is CPROLOG 1.5a/UNIX, implemented on a VAX 11/750 computer system. The programs written in PROLOG are the elementary-concept learner stored in the file <p_lrn>, the multiple-view learner stored in the <p_mtl>, the single-view learner stored in the file <p_sng>, the recognizer stored in the file <p_rec>. The above main routines use a number of procedures which are stored in the following files:

<p_cmn> containing common procedures used by all the main routines.

<p_gnl> containing general basic procedures [Clocksin & Mellish '81, Burnham & Hall '85, Bratko '86, Giannesini et al '86]. Some of the original procedures found in the above references have been modified and/or extended to suit the purposes of this project.

<p_fig> containing procedures concerning the involved figures. Some of these procedures use *retract(X)*, which is indicated by _*r*_ in their heads.

<p_df2>, and <p_dif3> containing the definitions of the involved 1-D, 2-D and 3-D figures (see §5.3) respectively.

<p_rdf> containing the *r_rule*'s used by the elementary concept learner.

Paragraph 2.2 contains a listing of the PROLOG programs.

REFERENCES

1. BRATKO, I. 1986: *PROLOG Programming for Artificial Intelligence*, Addison-Wesley.

2. BURNHAM, W.D. and HALL, A.R. 1985: *PROLOG Programming and Applications*, MacMillan.

3. CLOCKSIN, W.F. and MELLISH, C.S. 1981: *Programming in Prolog*, Springer-Verlag.

4. GIANNESINI, F., KANOUI, H. PASERO, R. and van CANEGHEM, M. 1986: *PROLOG*, Addison-Wesley.

```

                   /* elementary-concept learner */

lrn:-
    m_clr,
    nl,
    write('              insert r_rule(s) ?    '),
    read(Insert),
    ((Insert=y,
      manl_data_insert(YesNo),
      ((YesNo=y,
        nl,
        [user]);
       (YesNo=n,
        file_data_insert(FileName),
        consult(FileName),
        nl,
        write('      consulted r_rule(s)'),
        nl)));
     Insert=n),
    first_rule(Example,Rule),
    Example=(Hd:-Bd),
    Rule=(Head:-Body),
    repeat,
    nl,
    write('              improve rule ?    '),
    read(NoYes),
    ((NoYes=y,
      HdList=[Hd],
      improve(HdList));
     true),
    NoYes=n,
    nl,
    write('      end of rule improvement'),
    nl.

/* generate initial rule */

first_rule(Example,Rule):-
    insert_train_inst(positive,TrainInst),
    print_database,
    TrainInst=(R_Head,BodyList),
    train(TrainInst,TrInVal),
    prefix_cut(R_Head,r_,Head),
    store(pos_inst,(Head,BodyList,TrInVal)),
    head=..[HdFn|HdArgs],
    list_count(0,HdArgs,N),
    ((N>2,
      all_r_1D_figs([],FiglList),
      form_set(FiglList,Body),
      Example=(Head:-Body));
     (form_set(BodyList,Body),
      Example=(Head:-Body),
      list_retract(BodyList))),
    store(old_rule,Body),
    assert(Example),
    assert(limits([])),

/* generalize converting constants to variables */

    rule(Head,Rule),
    nl,
    write('      initial rule'),
    nl,
    nl,
    rl_write(Rule),
    nl.

/* improve incorrect rule */

improve([Head|Heads]):-
    Head=..[HeadFn|HeadArgs],
    print_database,
    critic(HeadFn,DiscrElem),
     ((DiscrElem=indeterminable,
      ((constrain,
        no_error([Head|Heads],NewBody));
       not(constrain)),
     nl,
     write('      no error can be detected'),
     nl,
     write('      try new training instance'),
     nl);
     modifier([Head|Heads],NewBody)),
     !.

/* input training instance */

insert_train_inst(KeyWord,TrainInst):-
    nl,
    write('      insert '),
    write(KeyWord),
    write(' training instance with format'),
    nl,
    write('           R_Head,[Body]'),
    nl,

/* manually */

    manl_data_insert(YesNo),
    ((YesNo=y,
      nl,
      read(TrainInst));

/* read from a file */

     (YesNo=n,
      file_data_insert(FileName),
      nl,
      write('      consulted training instance'),
      nl,
      retrieve(FileName,TrainInst),
      TrainInst=(R_Head,BodyList),
      nl,

/* output training instance */

      write(R_Head),
      nl,
      tab(6),
      write('  :  '),
      nl,
      list_write(BodyList),
      nl)).

/* test rule and detect error */

critic(HdFn,DiscrElem):-
    retrieve(pos_inst,(Head1,BodyList1,positive)),
    store(train_Inst,(Head1,BodyList1,positive)),
    nl,
    write('     consulted training instance'),
```

```
    nl,
    write(Head1),
    nl,
    tab(6),
    write('  :  '),
    nl,
    list_write(BodyList1),
    nl,
    context(NewBodyList1,Context1),
    insert_train_inst(new,TrainInst),
    TrainInst=(R_Head,BodyList2),
    train(TrainInst,TrInVal),
    prefix_cut(R_Head,r_,Head),
    store(train_inst,(Head,BodyList2,TrInVal)),
    Head=..[HeadFn|HeadArgs],
    test_new_rule(HeadFn,Fault),
    list_retract(BodyList2),
    ((Fault=no_error,
      DiscrElem=indeterminable);
     (Fault=omission_error,
      o_compare(HeadFn,Fault,DiscrElem));
     (Fault=commission_error,
      c_compare(HeadFn,Fault,DiscrElem))),
    nl,
    write('      discriminating element'),
    nl,
    nl,
    write(DiscrElem),
    nl.

/* check r_rules to test new rule */

train((R_Head,BodyList),TrInVal):-
    list_assert(BodyList),
    !,
    ((R_Head,
      TrInVal=positive);
     (not(R_Head),
      TrInVal=negative)),
    prefix_cut(R_Head,r_,Head),
    store(train_inst,(Head,BodyList,TrInVal)).

/* output results of test */

test_new_rule(HeadFn,Fault):-
    retrieve(train_inst,(Head,BodyList,TrInVal)),
    context(NewBodyList,Context),
    nl,
    write('      training instance : '),
    ((TrInVal=positive,
      write(TrInVal),
      nl,
      ((Head,
        NewRuleVal=true,
        Fault=no_error);
       (not(Head),
        NewRuleVal=false,
        Fault=omission_error)));
     (TrInVal=negative,
      write(TrInVal),
      nl,
      ((Head,
        NewRuleVal=true,
        Fault=commission_error);
       (not(Head),
        NewRuleVal=false,
        Fault=no_error)))),
    nl,
    write('      new rule          : '),
    write(NewRuleVal),
    nl,
    nl,
    write('      type of error     : '),
    write(Fault),
    nl.

/* determine context of new training instance */

context(NewBodyList,Context):-
   retrieve(train_inst,(Head,BodyList,TrInVal)),
   body_list(BodyList,NewBodyList),
   list_count(0,NewBodyList,N),
   Head=..[HeadFn|HeadArg],
   ((TrInVal=positive,
     Context=selection_context,
     form_list(Body,BodyList),
     assert(select_cont(HeadFn,NewBodyList,N)));
    (TrInVal=negative,
     Context=rejection_context,
     form_list(Body,BodyList),
     assert(reject_cont(HeadFn,NewBodyList,N)))).

/* compare contexts in case of commission error */

c_compare(H,F,D):-
   select_cont(H,B1,N1),
   reject_cont(H,B2,N2),
   ((N1\=N2,
     difference(B2,B1,D2),
     difference(B1,B2,D1),
     union(D1,D2,Du),
     D=[Du]);
    diff_pair(B1,B2,D)),
   context_compare_list(H,F,D,Diff),
   retract(select_cont(H,B1,N1)),
   retract(reject_cont(H,B2,N2)).

/* compare contexts in case of omission error */

o_compare(H,F,D):-
   select_cont(H,B1,N1),
   select_cont(H,B2,N2),
   ((N1\=N2,
     difference(B2,B1,D2),
     difference(B1,B2,D1),
     union(D1,D2,Du),
     D=[Du]);
    diff_pair(B1,B2,D)),
   context_compare_list(H,F,D,Diff),
   retract(select_cont(H,B1,N1)),
   retract(select_cont(H,B2,N2)).

/* compare a context with list of contexts */

context_compare_list(H,F,[D|Ds],[Diff|Diffs]):-
   context_compare(H,F,D,Diff),
   context_compare_list(H,F,Ds,Diffs).
context_compare_list(H,F,[],[]).

/* determine discriminating element */

context_compare(H,F,[D1,D2],Diff):-
```

```
    D1=[Fn1|Arg1],
    D2=[Fn2|Arg2],
    assert(true_context(Arg1)),
    (Fn1=Fn2,
      assert(cn(0)),
      differ(Arg1,Arg2,Dif,Index),
      retract(cn(X)),
      ((list_count(0,Indx,N),
       ((N=1,
         Diff=Dif);
        (N>1,
         Diff=[D1,D2])));
    Diff=[D1,D2]),
    (discr_element(H,F,Diff);
     assert(discr_element(H,F,Diff)))).

/* determine discriminating factor */

context_compare(H,F,[D],[D]):-
   discr_factor(H,F,[D]);
   assert(discr_factor(H,F,[D])).

/* correct faulty rule , case : [Fn1=Fn2 & 1<discr.literals] or Fn1\=Fn2
*/

modifier([Hd|Hds],NewBd):-
   Hd=..[H|A],
   discr_element(H,ErrorType,[Diff]),
   Diff=[D1,D2],
   rule_list([Hd|Hds],Body,OldRls),
   list_retract(OldRls),
   true_context(Cnxt),
   retract(true_context(Cnxt)),
   type_list([atom,integer,list],Diff,DiffTypeList),
   s_difference(Cnxt,[D1],RestCnxt),
   type_list([atom,integer,list],RestCnxt,RestCnxtTypeList),
   discrim(DiffTypeList,RestCnxtTypeList,DiscrElmList),
   form_set(NewCond,DiscrElmList),
   retrieve(old_rule,Bd),
   form_list(Bd,BdList),
   ((ErrorType=commission_error,
     ((subst_stronger_constr(NewCond,BdList,NewBdList);
       form_list(NewBd,NewBdList));
      body_set(Bd,DiscrElmList,NewBd));
       form_list(NewBd,NewBdList));
    (ErrorType=omission_error,
     NewBd=(Bd;NewCond))),
   ((not(bounds),
     store(new_rule,NewBd));
    bounds),
   rulel_ist([Hd|Hds],NewBd,NewRlList),
   list_assert(NewRlList),
   new_rule(Hds),
   set_rules([Hd|Hds]).

/* ...case : single discriminating literal */

modifier([Hd|Hds],NewBd):-
   Hd=..[H|A],
   discr_elelment(H,ErrorType,Diff),
   Diff=[D1,D2],
   Fcat1=..D1,
   Fcat2=..D2,
   rule_list([Hd|Hds],Body,OldRls),
   list_retract(OldRls),
   retrieve(old_body,Bd),
   OldRl=(Hd:-Bd),
   ((Hd\=line(A,B),
     ((r_1D_fig(Fig1),
       Factor1=Fig1);
       (((special_feature(SpecFeat1),
          Factor1=SpecFeat1);
         Factor1=Fact1)),
       retract(Fact1)),
     assert(Fact2),
     ((r_1D_fig(Fig2),
       Factor2=Fig2);
       (((special_feature(SpecFeat2),
          Factor2=SpecFeat2);
         Factor2=Fact2)),
       retract(Fact2)),
   Hd=line(A,B))),
   ((ErrorType=commission_error,
     NewCond=(Factor1,Factor2));
    (ErrorType=omission_error,
     NewCond=(Factor1;Factor2))),
   substitute(NewCond,Factor1,OldRl,NewRl),
   NewRl=(Hd:-NewBd),
   store(new_rule,NewBd),
   store(old_rule,NewBd),
   rule_list([Hd|Hds],NewBd,NewRlList),
   list_assert(NewRlList),
   new_rule(Hds),
   set_rules([Hd|Hds]).

/* ...case : extra factor */

modifier([Hd|Hds],NewBd):-
   Hd=..[H|A],
   discr_factor(H,commission_error,[D]),
   Fact=..D,
   rule_list([Hd|Hds],Body,OldRls),
   list_retract(OldRls),
   retrieve(old_body,Bd),
   OldRl=(Hd:-Bd),
   ((Hd\=line(A,B),
     assert(Fact),
     ((r_1D_fig(Fig),
       Factor=not(Fig));
       (((special_feature(SpecFeat),
          Factor=not(SpecFeat));
         Factor=not(Fig)),
       retract(Fact))));
    Hd=line(A,B)),
   body_set(Bd,[Factor],NewBd),
   store(new_rule,NewBd),
   rule_list([Hd|Hds],NewBd,NewRlList),
   list_assert(NewRlList),
   new_rule(Hds),
   set_rules([Hd|Hds]),
   retract(discr_factor(H,F,[D])).

/* fix bounds of varibles */

no_error([Hd|Hds],NewBd):-
   (instantiated(Elem),
    bounds([Hd|Hds]),
    clause(Hd,NewBd),
    store(old_rule,NewBd));
    clause(Hd,NewBd).

bounds([Hd|Hds]):-
```

```
    instantiated(Elem),
    retract(instantiated(Elem)),
    limits(List),
    append(List,Elem,Inst),
    retract(limits(List)),
    assert(limits(Inst)).

/* find pair of differing elements */

discrim([D1,D2],[A|As],[DiscrElem|X]):-
   D1=[Elm1,Type1],
   D2=[Elm2,Type2],
   ((Type1\=Type2,
     DiscrElem=..[Type1,Elm1]);
    (A=[ElmA,TypeA],
     ((Type1\=TypeA,
       (Type1=list,
        reltn_list([s_member],[A,Elm1,A,Elm2],[Rltn]),
        DiscrElem=..[Rltn,A,Elm1]);
       (TypeA=list,
        reltn_list([s_member],[Elm1,A,Elm2,A],[Rltn]),
        DiscrElem=..[Rltn,Elm1,A]);
      (reltn([=,\=],[A,Elm1,Elm2],Relat),
       (Relat=[Rltn],
        DiscrElem=..[Rltn,A,Elm1]);
       (Relat=[],
        ((Type1=atom,
          Rltn='\=',
          assert(instantiated(Elm2));
         (Type1=integer,
          ((Elm1>Elm2,
            Rltn='>');
           Rltn='<')),
          set_limits(Rltn,Elm2)),
        DiscrElem=..[Rltn,Elm1,Elm2]))))))),
   discrim([D1,D2],As,X).
discrim(D,[],[]).

/* determine type of arguments */

type_list(TypeList,[X|Xs],[TypeX|TypeXs]):-
   type(TypeList,X,TypeX).
   type_list(TypeList,Xs,TypeXs).
type_list(TypeList,[],[]).

type([Type|Types],X,[X|TypeX]):-
   T1=..[Type|X],
   not(T1),
   type(Types,X,[X|TypeX]).
type([Type|Types],X,[X|Type]).

/* determine relationship between arguments */

reltn_list([Rlt|Rlts],[A1,A2,B1,B2],[Relation|X]):-
   R1=..[Rlt,A1,B1],
   R2=..[Rlt,A2,B2],
   ((R1,
     not(R2),
     Relation=R1);
    (not(R1),
     R2,
     Relation=R2)),
reltn_list(Rlts,[A1,A2,B1,B2],X).
reltn_list([],[A1,A2,B1,B2],[]).

reltn([Rlt|Rlts],[A1,A2,B],[Rlt|X]):-
   R1=..[Rlt,A1,B],
   R2=..[Rlt,A2,B],
   R1,
   not(R2),
   reltn(Rlts,[A1,A2,B],X).
reltn([],[A1,A2,B],[]).

/* sustitute stronger constraint in rule-body */

subst_stronger_constr(Cns,[L|Ls],NewBdList):-
   Cns=..[CnsFn,D1,D2],
   L=..[LFn,D1,D3],
   LFn\=CnsFn,
   subst_stronger_constr(Cns,Ls,NewBdList).
subst_stronger_constr(Cns,[L|Ls],NewBdList):-
   Cns=..[CnsFn,D1,D2],
   L=..[CnsFn,D1,D3],
   ((CnsFn='>',
     D2<D3);
    (CnsFn='<',
     D2>D3)),
   substitute(Cns,L,BdList,NewBodyList),
   retract(limits(Lmts)),
   substitute(D3,D2,Lmts,NewLmts),
   assert(NewLmts).
subst_stronger_constr(Cns,[],BdList).

rule_list([Hd|Hds],Bd,[R1|X]):-
   R1=(Hd:-Bd),
   rule_list(Hds,Bd,X),
   !.
rule_list([],Bd,[]).

special_feature(Feature):-
   (poin_trn(A,B,C),
    Feature=poin_trn(A,B,C));
   (poin_qul(A,B,C,D),
    Feature=poin_qul(A,B,C,D));
   (nocnvx(A,B,C,D),
    Feature=nocnvx(A,B,C,D));
   (non_convex_contour_angle(A),
    Feature=non_convex_contour_angle(A)).

new_condition(FactList,NewCond):-
   disjunct_set(NewCond,FactList).

symmetric_new_cond(NewCond,SymNewCond):-
   (NewCond=..[Fn,A,B],
    (Fn=(=);
     Fn=(\=)),
    not(integer(A)),
    not(integer(B)),
    SymNewCond=..[Fn,B,A]);
   SymNewCond=..[Fn,A,B].

/* if argument correctly constrained fix bound and rule-body */

constrain([Hd|Hds]):-
   ((not(constrain),
     assert(constrain));
    constrain),
   move_new_old(NoYes),
   ((NoYes=n,
     nl,
     write('          argument correctly constrained ?    '),
     read(YesNo),
```

```
      ((YesNo=y,
        retract(move_new_old(n)),
        assert(move_new_old(y)));
       YesNo=n));
     NoYes=y),
    ((move_new_old(y),
      ((instantiated(Elem),
        limits(List),
        append(List,Elem,Inst),
        retract(instantiated(Elem)),
        retract(limits(List)),
        assert(limits(Inst)));
       not(instantiated(Elem))),
      retrieve(new_rule,NewBd),
      rule_list([Hd|Hds],NewBd,NewRlList),
      list_assert(NewRlList),
      store(old_rule,NewBd);
     move_new_old(n))),
    new_rule(Hds),
    set_rules([Hd|Hds]).

/* retain appropriate bound */

set_limits(DiscRel,D2):-
   DiscRel=(>),
   ((instantiated([D0]),
     maximum([D0,D2],Dmax),
     retract(instantiated([D0])),
     assert(instantiated([Dmax])));
    assert(instantiated([D2]))).
set_limits(DiscRel,D2):-
   DiscRel=(=),
   (instantiated([D2]);
    assert(instantiated([D2]))).
set_limits(DiscRel,D2):-
   DiscRel=(<),
   ((instantiated([D0]),
     minimum([D0,D2],Dmin),
     retract(instantiated([D0])),
     assert(instantiated([Dmin])));
    assert(instantiated([D2]))).
```

```
            /* single-view learner */

sng:-
   clr_data,
   clr_a_p_figs,
   nl,
   write('           consult file <p_mvr> ?    '),
   read(Cnslt),
   ((Cnslt=y,
     consult(p_mvr));
    Cnslt=n),
   nl,
   first_s_rule(A_HeadList,Body),
   repeat,
   nl,
   write('            improve rule ?'),
   nl,
   nl,
   read(NoYes),
   ((NoYes=y,
     s_improve(A_HeadList,NewBody));
    true),
   NoYes=n,
   nl,
   write('      end of rule improvement'),
   nl,
   p_rule(A_HeadList,NewBody).

/* generare initial rule */

first_s_rule(A_HeadList,Body):-
   insert_s_2D_view(positive,View),
   View=(Head,BodyList,ViewVal),
   prefix_add(Head,a_,A_Head),
   store(pos_inst,(A_Head,BodyList,positive)),
   store(train_inst,(A_Head,BodyList,positive)),
   A_Head=..[A_HdFn|A_HdArgs],
   list_count(0,A_HdArgs,Vertices),
   all_conns([],ConnList),
   list_count(0,ConnList,Edges),
   all_2D_figs([],Fig2List),
   print_a_figs,
   all_a_figs([],A_FigList),
   list_count(0,A_FigList,Faces),
   all_r_set_a_p_figs([],SetA_P_FigList),
   body_arg(SetA_P_FigList,AllArgLists),
   body_arg_list(AllArgLists,SngArgList),
   all_r_mlt_view_a_figlist(SngArgList,[],MltViewA_FigList),
   SngViewA_Fig=(SetA_P_FigList,(Faces,Edges,Vertices)),

/* find rule-head by matching body to multi_view_a_fig_list */

   set_match(SngViewA_Fig,MltViewA_FigList,PosMltViewA_FigList),

/* order alternative rule heads */

   set_best_match_order(PosMltViewA_FigList,PosMltViewA_FigList,SngView
_Fig,FnA_HdList),
   FnA_HdList=[FnA_Hd|FnA_Hds],
   FnA_HeadList=[FnA_Hd],
   nl,
   ((FnA_Hds=[],
     write('      rule head'));
    write('      alternative rule heads'),
   nl,
   nl,
```

```
   body_write(FnA_HdList),
   nl,
   form_set(SetA_P_FigList,Body),
   nl,
   write('       initial single_view rule'),
   nl,
   assert(limits([])),
   set_rules(FnA_HeadList,A_HdArgs,A_HdList,Body).

/* improve incorrect rule */

s_improve([Head|Heads],NewBody):-
   Head=..[HeadFn|HeadArgs],
   s_critic(HeadFn,DiscrElem),
   ((DiscrElem=no_discriminating_element,
    ((constrain,
      no_error([Head|Heads],NewBody));
     true),
     nl,
     write('       no error can be detected'),
     nl,
     write(' , try new training instance'),
     nl);
    modifier([Head|Heads],NewBody)),
   !.

/* detect error of rule */

s_critic(HdFn,DiscrElem):-
   retrieve(pos_inst,(A_Head1,BodyList1,positive)),
   nl,
   write('        consulted 2-D view'),
   nl,
   nl,
   write(A_Head1),
   nl,
   tab(6),
   write('  :  '),
   nl,
   list_write(BodyList1),
   nl,
   clr_data,
   context(NewBodyList1,Context1),
   insert_s_2D_view(ViewVal,View),
   View=(Head2,BodyList2,ViewVal),
   prefix_add(Head2,a_,A_Head2),
   View2=(A_Head2,BodyList2,ViewVal),
   store(train_inst,View2),
   all_2D_figs([],FigList),
   clr_data,
   A_Head2=..[A_HeadFn|A_HeadArgs],
   test_new_rule(A_HeadFn,Fault),
   ((Fault=no_error,
     DiscrElem=no_discriminating_element);
    (Fault=omission_error,
     o_compare(A_HeadFn,Fault,DiscrElem));
    (Fault=commission_error,
     c_compare(A_HeadFn,Fault,DiscrElem))),
   nl,
   write('       discriminating element'),
   nl,
   nl,
   write(DiscrElem),
   nl.

/* form list of multi_view_a_fig_list predicates */

all_r_mlt_view_a_figlist(SngArgList,_,[W|X]):-
   multi_view_a_fig_list((A_Hd,A_BdList),(Fc,Eg,Vc)),
   instantiate(A_Hd,SngArgList),
   W=((A_Hd,A_BdList),(Fc,Eg,Vc)),
   retract(multi_view_a_fig_list((A_Hd,A_BdList),(Fc,Eg,Vc))),
   all_r_mlt_view_a_figlist(SngArgList,[W|_],X),
   !.
all_r_mlt_view_a_figlist(SngArgList,W,[]):-
   not(multi_view_a_fig_list((A_Hd,Hd,A_BdList),(Fc,Eg,Vc))).

/* instantiate head arguments of single-view rule */

instantiate(A_Hd,SngArgList):-
   A_Hd=..[A_HdFn|A_HdArgs],
   list_count(0,SngArgList,N1),
   list_count(0,A_HdArgs,N2),
   MaxArgList=[v1,v2,v3,v4,v5,v6,v7,v8],
   difference(MaxArgList,SngArgList,DifArgList),
   ((N1=N2;
     N1>N2);
    uninst(SngArgList,A_HdArgs,DifArgList)).

/* leave excess head arguments of multiple-view rule uninstatiated */

uninst([L|Ls],[M|Ms],R):-
   uninst(Ls,Ms,R).
uninst([],M,R):-
   inst(M,R).

inst([L|Ls],[M|Ms]):-
   L=M,
   inst(Ls,Ms),
   !.
inst([],M).

/* generate single-view rule p_rule */

p_rule([A_Hd|A_Hds],Body):-
   retract((A_Hd:-Body)),
   prefix_change(A_Hd,p,P_Hd),
   form_head(P_Hd,Head),
   Rule=(Head:-Body),
   assert(Rule),
   rule(Head,NewRule),
   Head=..[HeadFn|HeadArgs],
   listing(HeadFn),
   p_rule(A_Hds,Body),
   !.
p_rule([],Body).

/* add suffix 'a' to the 2nd of two identical heads */

form_head(Hd,HewHd):-
   (clause(Hd,Bd),
    suffix_add(Hd,a,NewHd));
   NewHd=Hd.
   Rule=(Head:-Body),

/* generate rule(s) by converting constant arguments to variable */

set_rules([FnHd|FnHds],ArgsHd,[Hd|X],Body):-
   Hd=..[FnHd|ArgsHd],
   Example=(Hd:-Body),
   store(old_rule,Body),
   assert(Example),
```

```
    rule(Hd,Rule),
    nl,
    rl_write(Rule),
    nl,
    clr_discr_elem,
    set_rules(FnHds,ArgsHd,X,Body),
    !.
set_rules([],ArgsHd,[],Body).

/* form list of matching multi-view-figures to single-view a_figures */

set_match(SngViewA_Fig,[MltViewA_Fig|MltViewA_Figs],[MltViewA_Fig|W]):-
    SngViewA_Fig=(A_P_FigList,(Faces1,Edges1,Vertices1)),
    MltViewA_Fig=((A_Head,A_BodyList),(Faces2,Edges2,Vertices2)),
    match(A_P_FigList,A_BodyList),
    set_match(SngViewA_Fig,MltViewA_Figs,W),
    !.
set_match(SngViewA_Fig,[MltViewA_Fig|MltViewA_Figs],W):-
    set_match(SngViewA_Fig,MltViewA_Figs,W),
    !.
set_match(S,[],[]).

/* match bodylist of single_view to a part of multiple-view */

match([SngViewFig|SngViewFigs],MltViewFigList):-
    S0=SngViewFig,
    (s_belong(S0,SngViewFig,MltViewFigList);
     p_belong(S0,SngViewFig,MltViewFigList)),
    match(SngViewFigs,MltViewFigList),
    !.
match([],X).

/* match occurs when functors and argument lists are equal */

s_belong(X0,X,[X|_]).
s_belong(X0,X,[R|_]):-
    X=..[FnX|ArgsX],
    R=..[FnR|ArgsR],
    FnX=FnR,
    s_equal(ArgsX,ArgsR).
s_belong(X0,X,[R|Y]):-
    X=..[FnX|ArgsX],
    R=..[FnR|ArgsR],
    FnX=FnR,

/* otherwise commutate argumens */

    commutate(X,NewX),
    X0\=NewX,
    s_belong(X0,NewX,[R|Y]).
s_belong(X0,X,[_|Y]):-
    s_belong(X0,X,Y),
    !.

/* functors can be visible or possible */

p_belong(X0,X,[X|_]).
p_belong(X0,X,[R|_]):-
    X=..[FnX|ArgX],
    R=..[FnR|ArgR],
    ((FnX=set_ptr,
      FnR=set_atr);
     (FnX=set_pqu,
      FnR=set_aqu);
     (FnX=set_n_pqu,
      FnR=set_nqu)),
    s_equal(ArgX,ArgR).
p_belong(X0,X,[R|Y]):-
    X=..[FnX|ArgX],
    R=..[FnR|ArgR],
    ((FnX=set_ptr,
      FnR=set_atr);
     (FnX=set_pqu,
      FnR=set_aqu);
     (FnX=set_n_pqu,
      FnR=set_nqu)),
    commutate(X,NewX),
    X0\=NewX,
    p_belong(X0,NewX,[R|Y]).
p_belong(X0,X,[_|Y]):-
    p_belong(X0,X,Y).

/* criteria for ordering alternative rule heads */

best_match_order(M,[L|Ls],S,R):-
    S=(SetA_P_Figs,(Fc,Eg,Vc)),
    M=((A_Hd1,BdList1),(Fc1,Eg1,Vc1)),
    L=((A_Hd2,BdList2),(Fc2,Eg2,Vc2)),
    ((M\=L,

/* first figure with maximum no. of faces */

     (Fc1<Fc2;

/*then figure with maximum no. of edges */

        Eg1<Eg2;

/*last figure with maximum no. of vertices */

        Vc1<Vc2)),
      (M=L,
       best_match_order(M,Ls,S,R));
     best_match_order(M,Ls,S,R)).
     !.
best_match_order(M,[],S,R):-
    M=((A_Hd,BdList),(Fc,Eg,Vc)),
    S=(SetA_P_Figs,(Fc1,Eg1,Vc1)),

/* use no. of hidden planes as suffix to rule head */

    N is Fc-Fc1,
    A_Hd=..[FnA_Hd|ArgsA_Hd],
    suffix_add(FnA_Hd,N,NewFnA_Hd),
    R=FnNewA_Hd.

set_best_match_order([M|Ms],L,S,[R|X]):-
    best_match_order(M,L,S,R),
    exclude(M,[M|Ms],NewM),
    exclude(M,L,NewL),
    set_best_match_order(NewM,NewL,S,X),
    !.
set_best_match_order([M|Ms],L,S,X):-
    set_best_match_order(Ms,L,S,X),
    !.
set_best_match_order([],L,S,[]).
```

```
                /* multiple-view learner */

mlt:-
    m_clr,
    slm,
    FileList=[v_bac,v_top,v_bot,v_lef,v_rig],
    ViewList=['back_side','top_side','bottom_side','lefthand_side',
              'righthand_side'],
    store(other_views,(FileList,ViewList)),
    m_rule(Example,Rule).

/* create multiple-view rule */

m_rule(Example,Rule):-
    repeat,
    clr_data,
    insert_m_2D_view(positive,View),
    View=(Head,BodyList,ViewVal),
    all_2D_figs([],Fig2List),
    all_r_a_c_lines([],A_C_LineList1),
    (not(point_in_trn(A1,B1,C1));
     retract(point_in_trn(A1,B1,C1))),
    (not(point_in_qul(Aa1,Ba1,Ca1,Da1));
     retract(point_in_qul(Aa1,Ba1,Ca1,Da1))),
    all_r_a_p_figs([],A_P_FigList),
    list_assert(BodyList),
    lines(2,BodyList,Line2List),
    all_2D_figs([],Fig1List),
    print_a_p_figs,
    all_a_figs([],A_FigList),
    store(a_fig_list1,A_FigList),
    all_r_a_p_figs([],A_P_FgList),
    (not(point_in_trn(A1,B1,C1));
     retract(point_in_trn(A1,B1,C1))),
    (not(point_in_qul(Aa1,Ba1,Ca1,Da1));
     retract(point_in_qul(Aa1,Ba1,Ca1,Da1))),
    list_assert(A_P_FigList),
    nl,
    write('              first 2-D view ?    '),
    read(First),
    ((First=y,
      store(a_fig_list2,A_FigList),
      all_r_a_lines(Conn1List),
      all_r_c_lines(Conn0List),
      equ_union(Conn1List,Conn0List,ConnList),
      store(conn_list,ConnList),
      lines(0,Conn0List,Lin0List),
      equ_intersection(Lin0List,Line2List,NewLin0List),
      store(line0,NewLin0List),
      store(line2,Line2List),
      store(view2,View));
     (First=n,
      retrieve(a_fig_list1,A_FigList1),
      retrieve(a_fig_list2,A_FigList2),
      all_set_same_face_a_lines(A_FigList1,A_FigLineList1),
      all_set_same_face_a_lines(A_FigList2,A_FigLineList2),
      all_set_common_a_line_discard(A_FigLineList1,A_FigLineList2,
                                    NewA_FigLineList1,NewA_FigLineList2),
      all_a_figs([],NewA_FigList),
      store(a_fig_list2,NewA_FigList),
      retrieve(conn_list,OldConnList),
      conn_assert(OldConnList),
      all_r_a_lines(Conn1List),
      all_r_c_lines(Conn0List),
      equ_union(Conn1List,Conn0List,ConnList),
      store(conn_list,ConnList),
      retrieve(view1,(Head1,BodyList1,ViewVal1)),
      retrieve(view2,(Head2,BodyList2,ViewVal2)),
      equ_union(BodyList1,BodyList2,NewBodyList),
      adjust_head(Head1,Head2,NewHead),
      NewView=(NewHead,NewBodyList,NewViewVal),
      store(view2,NewView))),
    ((not(belong(conn(S,T,1),ConnList)),
      all_r_a_c_lines([],A_C_LineList2),
      (not(point_in_trn(A2,B2,C2));
       retract(point_in_trn(A2,B2,C2))),
      (not(point_in_qul(Aa2,Ba2,Ca2,Da2));
       retract(point_in_qul(Aa2,Ba2,Ca2,Da2))),
      retrieve(view2,(Hd,BdList,VwVl)),
      list_assert(BdList),
      all_conns([],Conns),
      list_count(0,Conns,Edges),
      all_2D_figs([],FigList),
      print_a_p_figs,
      form_set(FigList,Body),
      Example=(Hd:-Body),
      store(old_rule,Example),
      assert(Example),
      assert(limits([])),
      rule(Hd,Rule),
      nl,
      write('      multiple_view rule'),
      nl,
      nl,
      rl_write(Rule),
      nl,
      all_r_set_a_p_figs([],SetA_FigList),
      list_count(0,SetA_FigList,Faces),
      form_set(SetA_FigList,A_Body),
      prefix_add(Hd,a_,A_Hd),
      A_Hd=..[FnA_Hd|ArgsA_Hd],
      list_count(0,ArgsA_Hd,Vertices),
      A_Example=(A_Hd:-A_Body),
      store(old_p_rule,A_Example),
      assert(A_Example),
      assert(limits([])),
      rule(A_Hd,A_Rule),
      A_Rule=(NewA_Hd:-NewA_Body),
      single_factor_list(NewA_Body,NewA_BodyList),
      assert(multi_view_a_fig_list((NewA_Hd,NewA_BodyList),
                                   (Faces,Edges,Vertices))),
      nl,
      write('      multiple-view a_figure list'),
      nl,
      listing(multi_view_a_fig_list));
     (another_view(Conn1List,Conn0List,Line2List),
      fail)).

/* ask for another view */

another_view(Conn1List,Conn0List,Line2List):-
    nl,
    write('      another 2-D view needed'),
    nl,
    nl,
    write('              any particular 2-D view ?    '),
    read(YesNo),
    ((YesNo=y,
      lines(1,Conn1List,Line1List),
      lines(0,Conn0List,Ln0List),
      equ_intersection(Ln0List,Line2List,NewLine0List),
      retrieve(line0,OldLine0List),
```

```
      equ_union(OldLine0List,NewLine0List,Line0List),
      store(line0,Line0List),
      retrieve(line2,OldLine2List),
      equ_union(Line2List,OldLine2List,NewLine2List),
      store(line2,NewLine2List),
      retrieve(other_views,(FileList,ViewList)),
      next_view(FileList,NewLine2List,Line1List,Line0List,NextFileList),
      find_file(FileList,ViewList,NextFileList,Views),
      NextFileList=[NextFile|NextFiles],
      Views=[Vw|Vws],
      nl,
      write('      to include occluded faces , containing lines :'),
      nl,
      write(Line1List),
      nl,
      nl,
      write('     should not contain line(s) :'),
      nl,
      write(Line0List),
      nl,
      nl,
      write('      next view from the  '),
      write(Vw),
      view_write(Vws),
      nl,
      nl,
      write('      in file      :      '),
      write(NextFile),
      view_write(NextFiles),
      nl);
     YesNo=n).

insert_m_2D_view(KeyWord,View):-
   nl,
   write('      insert '),
   write(KeyWord),
   write(' 2-D view with format'),
   nl,
   write('            Head,[Body],ViewVal'),
   nl,
   manl_data_insert(YesNo),
   ((YesNo=y,
     nl,
     read(View),
     View=(Head,BodyList,ViewVal),
     list_assert(BodyList));
    (YesNo=n,
     file_data_insert(FileName),
     retrieve(other_views,(FileList,ViewList)),
     exclude(FileName,FileList,NewFileList),
     find_file(FileList,ViewList,[FileName],ViewName),
     exclude_list(ViewName,ViewList,NewViewList),
     store(other_views,(NewFileList,NewViewList)),
     retrieve(FileName,View),
     View=(Head,BodyList,ViewVal),
     list_assert(BodyList),
     nl,
     write('      consulted 2-D view'),
     nl,
     listing(conn),
     print_spec_feat,
   store(view1,View).

/* remove common a_lines between two faces */

common_a_line_discard([Face1,A_LineList1],[Face2,A_LineList2],[Face1,New
A_LineList1],[Face2,NewA_LineList2]):-
   equ_intersection(A_LineList1,A_LineList2,ComA_LineList),
   ((same(Face1,Face2),
     NewA_LineList1=ComA_LineList,
     NewA_LineList2=ComA_LineList);
    (equ_difference(A_LineList1,ComA_LineList,NewA_LineList1),
     equ_difference(A_LineList2,ComA_LineList,NewA_LineList2),
     update_conn_list(ComA_LineList))).

/* remove common a_lines between a face a list of faces */

set_common_a_line_discard(F,[L|Ls],NewF,[NewL|NewLs]):-
   common_a_line_discard(F,L,NewF1,NewL),
   set_common_a_line_discard(NewF1,Ls,NewF2,NewLs),
   !.
set_common_a_line_discard(F,[],NewF,[]).

/* remove common a_lines between two lists of faces */

all_set_common_a_line_discard([F|Fs],L,[NewF|NewFs],NewL):-
   set_common_a_line_discard(F,L,NewF,NewL1),
   all_set_common_a_line_discard(Fs,NewL1,NewFs,NewL2),
   !.
all_set_common_a_line_discard([],L,[],NewL).

/* decrease face counter of a_lines by 1 */

update_conn_list([C|Cs]):-
   ((C=a_line(A,B),
     rtr_a_line(A,B));
    (C=a_line(B,A),
     rtr_a_line(B,A))),
   assert(conn(A,B,0)),
   update_conn_list(Cs),
   !.
update_conn_list([]).

/* form argument list of rule-head */

adjust_head(Hd1,Hd2,NewHd):-
   (not(same(Hd1,Hd2)),
    Hd1=..[FnHd1|ArgHd1],
    Hd2=..[FnHd2|ArgHd2],
    union(ArgHd1,ArgHd2,NewArgHd),
    NewHd=..[FnHd1|NewArgHd]);
   (same(Hd1,Hd2),
    NewHd=Hd1).

/* from a list of 6 views determine next view(s) */

next_view(ViewList,Ln2List,Ln1List,Ln0List,NextViewList):-
   score_list(Ln2List,Ln1List,Ln0List,ViewList,NewViewList,
              ScorList,Scr2List,Scr1List,Scr0List),
   set_list_max([Scr2List,Scr1List],MaxScrList),
   set_list_min([Scr0List],MinScrList),
   append(MaxScrList,MinScrList,BestScorList),
   diflist(BestScorList,ScorList,DifList,Dif2List,Dif1List,Dif0List),
   set_list_min([Dif2List,Dif1List,Dif0List],MinDifList),
   best_diflist(2,MinDifList,NewViewList,DifList,NewView2List,NewDif2Lis
t),
   NewView2List=[NewView2|NewView2s],
   ((NewView2List=[],
     best_dif(Ln1List,MinDifList,NewViewList,DifList,
              NewView2List,NewDif2List,NextViewList));
    ((NewView2List\=[],
      ((NewView2s=[],
```

```
            NextViewList=NewView2List);
           (NewView2s\=[],
            best_dif(LnlList,MinDifList,NewViewList,DifList,
                    NewView2List,NewDif2List,NextViewList)))))).

/* criteria for selection of next view */

best_dif(LnlList,MinDifList,NewViewList,DifList,
        NewView2List,NewDif2List,NextViewList):-

/* that with maximum of a_lines */

   best_diflist(1,MinDifList,NewView2List,NewDif2List,NewView1List,
               NewDif1List),
   NewView1List=[NewView1|NewView1s],
   ((NewView1s=[],
     NextViewList=NewView1List);
    (NewView1s\=[],

/* that with minimum of c_lines */

     best_diflist(0,MinDifList,NewView1List,NewDif1List,NewView0List,
                 NewDif0List),
     NewView0List=[NewView0|NewView0s],
     ((NewView0s=[],
       NextViewList=NewView0List);
      (NewView0s\=[],
       vertex_list(LnlList,[],VrtxList),
       list_count(0,VrtxList,VrtxNo),
       ((VrtxNo>=5,
         NextViewList=NewView0List);
        (VrtxNo<5,
         next_view_list(VrtxList,NewView0List,NextViews)),
         ((NextViews=[],
           NextViewList=NewView0List);
          (NextViews\=[],
           NextViewList=NextViews))))))).

/* that with point inside 2-D figure */

next_view_list([A,B,C],[V|Vs],[V|X]):-
   retrieve(V,(Hd,BdList,ViewVal)),
   belong(point_in_trn(A,B,C),BdList),
   next_view_list([A,B,C],Vs,X),
   !.
next_view_list([A,B,C],[V|Vs],X):-
   retrieve(V,(Hd,BdList,ViewVal)),
   not(belong(point_in_trn(A,B,C),BdList)),
   next_view_list([A,B,C],Vs,X),
   !.
next_view_list([A,B,C],[],[]).
next_view_list([A,B,C,D],[V|Vs],[V|X]):-
   retrieve(V,(Hd,BdList,ViewVal)),
   belong(point_in_qul(A,B,C,D),BdList),
   next_view_list([A,B,C,D],Vs,X),
   !.
next_view_list([A,B,C,D],[V|Vs],X):-
   retrieve(V,(Hd,BdList,ViewVal)),
   not(belong(point_in_qul(A,B,C,D),BdList)),
   next_view_list([A,B,C,D],Vs,X),
   !.
next_view_list([A,B,C,D],[],[]).


/* find no. of a_ ,b_ ,and c_lines in 2-D view */

score_list(Ln2List,LnlList,Ln0List,[V|Vs],[V|X],[S|Y],[S2|W2],[S1|W1],
0|W0]):-
   retrieve(V,(Hd,BdList,ViewVal)),
   lines(2,BdList,LineList),
   n_score(LineList,Ln2List,0,Ln2Scor),
   S2=Ln2Scor,
   p_score(LnlList,LineList,0,LnlScor),
   S1=LnlScor,
   p_score(Ln0List,LineList,0,Ln0Scor),
   S0=Ln0Scor,
   S=[Ln2Scor,LnlScor,Ln0Scor],
   score_list(Ln2List,LnlList,Ln0List,Vs,X,Y,W2,W1,W0),
   !.
score_list(Ln2List,LnlList,Ln0List,[],[],[],[],[],[]).

/* positive score : counter of lines incremented if line in view */

p_score([C|Cs],DataList,S,FnlS):-
   not(belong(C,DataList)),
   p_score(Cs,DataList,S,FnlS),
   !.
p_score([C|Cs],DataList,S,FnlS):-
   belong(C,DataList),
   NewS is S+1,
   p_score(Cs,DataList,NewS,FnlS),
   !.
p_score([],DataList,S,S).

/* negative score : counter of lines incremented if line not in view *

n_score([C|Cs],DataList,S,FnlS):-
   belong(C,DataList),
   n_score(Cs,DataList,S,FnlS),
   !.
n_score([C|Cs],DataList,S,FnlS):-
   not(belong(C,DataList)),
   NewS is S+1,
   n_score(Cs,DataList,NewS,FnlS),
   !.
n_score([],DataList,S,S).

/* form list of maxima of list of lists */

set_list_max([S|Ss],[Max|X]):-
   assert(max(0)),
   list_max(S,Max),
   set_list_max(Ss,X),
   !.
set_list_max([],[]).

/* find maximum of a list of values */

list_max([L|Ls],MaxL):-
   max(Max),
   maximum(L,Max,NewMax),
   retract(max(Max)),
   assert(max(NewMax)),
   list_max(Ls,MaxL),
   !.
list_max([],MaxL):-
   retract(max(MaxL)).

/* form list of minima of list of lists */

set_list_min([S|Ss],[Min|X]):-
   assert(min(100)),
```

```
    list_min(S,Min),
    set_list_min(Ss,X),
    !.
set_list_min([],[]).

/* find minimum of a list of values */

list_min([L|Ls],MinL):-
   min(Min),
   minimum(L,Min,NewMin),
   retract(min(Min)),
   assert(min(NewMin)),
   list_min(Ls,MinL),
   !.
list_min([],MinL):-
   retract(min(MinL)).

/* form differences of a value from a list of valus */

diflist([B2,B1,B0],[S|Ss],[D|X],[D2|X2],[D1|X1],[D0|X0]):-
   S=[S2,S1,S0],
   D2 is B2-S2,
   D1 is B1-S1,
   D0 is S0-B0,
   D=[D2,D1,D0],
   diflist([B2,B1,B0],Ss,X,X2,X1,X0),
   !.
diflist([B2,B1,B0],[],[],[],[],[]).

/* find optimum difference list */

best_diflist(N,M,[V|Vs],[D|Ds],[V|X],[D|Y]):-
   D=[D2,D1,D0],
   M=[M2,M1,M0],
   ((N=2,
     M2=D2,
     best_diflist(N,M,Vs,Ds,X,Y));
    (N=1,
     M1=D1,
     best_diflist(N,M,Vs,Ds,X,Y));
    (N=0,
     M0=D0,
     best_diflist(N,M,Vs,Ds,X,Y))),
   !.
best_diflist(N,M,[V|Vs],[D|Ds],X,Y):-
   D=[D2,D1,D0],
   M=[M2,M1,M0],
   ((N=2,
     M2\=D2,
     best_diflist(N,M,Vs,Ds,X,Y));
    (N=1,
     M1\=D1,
     best_diflist(N,M,Vs,Ds,X,Y));
    (N=0,
     M0\=D0,
     best_diflist(N,M,Vs,Ds,X,Y))),
   !.
best_diflist(N,M,[],[],[],[]).

/* find file-name of view */

find_file([V|Vs],[W|Ws],NextViewList,[W|X]):-
   belong(V,NextViewList),
   find_file(Vs,Ws,NextViewList,X),
   !.
find_file([V|Vs],[W|Ws],NextViewList,X):-
   not(belong(V,NextViewList)),
   find_file(Vs,Ws,NextViewList,X),
   !.
find_file([],[],NextViewList,[]).

/* output alternative views */

view_write([W|Ws]):-
   nl,
   ((W\=[],
     tab(24),
     write('or'),
     nl,
     tab(26),
     write(W));
    (Ws=[],
     nl)),
   nl,
   view_write(Ws),
   !.
view_write([]).

/* form vertex-list for rule head */

vertex_list([L|Ls],X,V):-
   L=line(A,B),
   union(X,[A,B],U),
   vertex_list(Ls,U,V),
   !.
vertex_list([],V,V).
```

```
                /* common procedures */


/* generate set of rules */

set_rules([Hd|Hds]):-
   rule(Hd,Rule),
   nl,
   rl_write(Rule),
   nl,
   clr_discr_elem,
   set_rules(Hds),
   !.
set_rules([]).

/* generate rule */

rule(Head,Rule):-
   rule_args(Head,Body,ArgUn),
   rl_sublist(Var,ArgUn,(Head:-Body),Rule),
   retract((Head:-Body)),
   asserta(Rule).

/* form list of arguments for rule_body */

rule_args(Head,Body,ArgUn):-
   clause(Head,Body),
   Head=..[Fn|Args],
   single_factor_list(Body,BodyList),
   body_arg(BodyList,BodyArgs),
   body_arg_list(BodyArgs,BodyArgUn),
   union(Args,BodyArgUn,ArgUn).

body_arg_list(BodyArgs,BodyArgUn):-
   assert(un([])),
   all_args(BodyArgs),
   un(BodyArgUn),
   retract(un(BodyArgUn)).

all_args([L|Ls]):-
   un(X),
   union(L,X,ArgUnion),
   retract(un(X)),
   assert(un(ArgUnion)),
   all_args(Ls).
all_args([]).

body_arg([Bd|Bds],[ArgArg|Args]):-
   Bd=..[not|Arg],
   [Ar]=Arg,
   Ar=..[ArgFn|ArgArg],
   body_arg(Bds,Args).
body_arg([Bd|Bds],[Arg|Args]):-
   Bd=..[Fn|Arg],
   body_arg(Bds,Args).
body_arg([],[]).

/* convert constant arguments to variables except 'instantiated' */

rl_sublist([],[],X,X):-
   !.
rl_sublist([New|News],[Old|Olds],X,Z):-
   ((not(instantiated([U])),
     limits(W),
     not(member(Old,W)));
    (instantiated([U]),
     Old\=U,
     limits(W),
     not(member(Old,W)))),
   substitute(New,Old,X,Y),
   rl_sublist(News,Olds,Y,Z).
rl_sublist([New|News],[Old|Olds],X,Z):-
   ((not(instantiated([U])),
     limits(W),
     member(Old,W));
    (instantiated([U]),
     Old=U);
    (instantiated([U]),
     Old\=U,
     limits(W),
     member(Old,W))),
   rl_sublist(News,Olds,X,Z).

/* change prefix of name */

prefix_add(Pred,Prefix,NewPred):-
   Pred=..[PredFn|PredArgs],
   name(PredFn,PredLtrList),
   name(Prefix,PrefixList),
   append(PrefixList,PredLtrList,NewPredLtrList),
   name(NewPredFn,NewPredLtrList),
   NewPred=..[NewPredFn|PredArgs].

prefix_cut(Pred,Prefix,NewPred):-
   Pred=..[PredFn|PredArgs],
   name(PredFn,PredLtrList),
   name(Prefix,PrefixList),
   append(PrefixList,NewPredLtrList,PredLtrList),
   name(NewPredFn,NewPredLtrList),
   NewPred=..[NewPredFn|PredArgs].

prefix_change(Pred,NewPrefix,NewPred):-
   Pred=..[PredFn|PredArgs],
   name(PredFn,[PredLtr|PredLtrs]),
   name(NewPrefix,NewPrefixList),
   append(NewPrefixList,PredLtrs,NewPredList),
   name(NewPredFn,NewPredList),
   NewPred=..[NewPredFn|PredArgs].

/* change suffix of name */

suffix_add(Pred,Suffix,NewPred):-
   Pred=..[PredFn|PredArgs],
   name(PredFn,PredLtrList),
   name(Suffix,SuffixList),
   append(PredLtrList,SuffixList,NewPredLtrList),
   name(NewPredFn,NewPredLtrList),
   NewPred=..[NewPredFn|PredArgs].

factor_list([D|Ds],Arg,[Factor|Factors]):-
   Factor=..[D,Arg],
   factor_list(Ds,Arg,Factors).
factor_list([],Arg,[]).

/* form new rule-body by conjunction of new factor */

body_set(Body,NewFact,NewBody):-
   form_list(Body,BodyList),
   append(BodyList,NewFact,NewBodyList),
   form_list(NewBody,NewBodyList).

/* convert bodies into lists : [BodyFunctor|BodyArguments] */
```

```
body_list([Bd|Bds],[NewBd|NewBds]):-
   Bd=..[Fn|Args],
   NewBd=[Fn|Args],
   body_list(Bds,NewBds).
body_list([],[]).

/* form list of body functors */

body_fn([Bd|Bds],[Fn|Fns]):-
   Bd=..[Fn|Arg],
   body_fn(Bds,Fns).
body_fn([],[]).

/* find position of argument */

arg_index(N,[D|Ds],Arg):-
   N>1,
   N1 is N-1,
   arg_index(N1,Ds,Arg).
arg_index(1,D,Arg):-
   arg(1,D,Arg).

/* for single-view recognizer */

insert_s_2D_view(KeyWord,View):-
   nl,
   write('      insert '),
   write(KeyWord),
   write(' 2-D view with format'),
   nl,
   write('          Head,[Body],ViewVal'),
   nl,
   manl_data_insert(YesNo),
   ((YesNo=y,
     nl,
     read(View),
     View=(Head,BodyList,ViewVal),
     list_assert(BodyList));
    (YesNo=n,
     file_data_insert(FileName),
     retrieve(FileName,View),
     View=(Head,BodyList,ViewVal),
     list_assert(BodyList),
     nl,
     write('      consulted 2-D view'),
     nl,
     listing(conn),
     print_spec_feats)),
   store(view1,View).

manl_data_insert(YesNo):-
   nl,
   write('            manually ?    '),
   read(YesNo).

file_data_insert(FileName):-
   nl,
   write('      insert name of input-file'),
   nl,
   nl,
   read(FileName).

store(FileName,Data):-
   tell(FileName),
   write(Data),
   write(.),
   nl,
   told.

retrieve(FileName,Data):-
   see(FileName),
   read(Data),
   seen.

/* input-output procedures */

rl_write(Rule):-
   Rule=(Head:-Body),
   Head=..[HdFn|HdArg],
   listing(HdFn).

new_rule_assert([Hd|Hds],Bd,NewBd):-
   retract((Hd:-Bd)),
   assert((Hd:-NewBd)),
   new_rule_assert(Hds,Bd,NewBd),
   !.
new_rule_assert([],Bd,NewBd).

new_rule(Hds):-
   nl,
   write('      new '),
   ((Hds=[],
     write('rule'));
    write('rules')),
   nl.

print_a_p_figs:-
   listing(atrian),
   listing(aquadril),
   listing(nquadril),
   listing(ptrian),
   listing(pquadril),
   listing(p_nquadril).

print_database:-
   listing(conn),
   print_a_p_figs,
   print_spec_feats.

conn_assert([C|Cs]):-
   ((C=conn(A,B,1),
     not(c_line(A,B)),
     not(a_line(A,B)),
     assert(C));
    (C=conn(A,B,0),
     ((a_line(A,B),
       rtr_a_line(A,B),
       assert(C));
      (not(c_line(A,B)),
       assert(conn(A,B,0))));
      true)),
   conn_assert(Cs),
   !.
conn_assert([]).

list_write([L|Ls]):-
   write(L),
   ((Ls\=[],
     write(','),
     nl);
    (write(':'),
```

```
        nl)),
      list_write(Ls).
  list_write([]).

body_write([L|Ls]):-
    write(L),
    write('.'),
    nl,
    body_write(Ls).
body_write([]).

print_list([L|Ls]):-
    write(L),
    nl,
    print_list(Ls).
print_list([]).

/* output special features */

spec_feats:-
    (not(point_in_trn(A1,B1,C1)),
     not(point_in_qul(A2,B2,C2,D2)),
     not(non_convex(A3,B3,C3,D3)),
     not(non_convex_contour_angle(D1)),
     write('none'),
     nl);
    print_spec_feats.

print_spec_feats:-
    listing(point_in_trn),
    listing(point_in_qul),
    listing(non_convex),
    listing(non_convex_contour_angle).

functor_list(Head,[Fn|Fns]):-
    clause(Head,Body),
    form_list(Body,BodyList),
    body_fn(BodyList,[Fn|Fns]),
    assert(predic([Fn|Fns])).

/* clear-database procedures */

m_clr:-
    (not(select_cont(A1,B1,C1));
      retractall(select_cont(Aa1,Ba1,Ca1))),
    (not(reject_cont(A2,B2,C2));
      retractall(reject_cont(Aa2,Ba2,Ca2))),
    (not(discr_factor(A3,B3,C3));
     retractall(discr_factor(Aa3,Ba3,Ca3))),
    (not(un(A4));
     retractall(un(Aa4))),
    (not(limits(A5));
     retractall(limits(Aa5))),
    (not(discr_elem_list(A6));
     retractall(discr_elem_list(Aa6))),
    (not(conn(A7,B7,C7));
     retractall(conn(Aa7,Ba7,Ca7))),
     clr_discr_elem,
     clr_data,
     clr_a_p_figs.

 clr_discr_elem:-
    ((discr_element(A,B,C,D),
      retract(discr_element(A,B,C,D)));
     not(discr_element(A,B,C,D))),
     ((move_new_old(U),
      retract(move_new_old(U)));
     not(move_new_old(U))).

clr_data:-
    retractall(conn(Aa1,Ba1,Na1)),
    retractall(point_in_trn(Aa2,Ba2,Ca2)),
    retractall(point_in_qul(Aa3,Ba3,Ca3,Da3)),
    retractall(non_convex_contour_angle(Aa4)).

clr_a_p_figs:-
    (not(atrian(A1,B1,C1));
     retractall(atrian(Aa1,Ba1,Ca1))),
    (not(ptrian(A2,B2,C2));
     retractall(ptrian(Aa2,Ba2,Ca2))),
    (not(aquadril(A3,B3,C3,D3));
     retractall(aquadril(Aa3,Ba3,Ca3,Da3))),
    (not(pquadril(A4,B4,C4,D4));
     retractall(pquadril(Aa4,Ba4,Ca4,Da4))),
    (not(nquadril(A5,B5,C5,D5));
     retractall(nquadril(Aa5,Ba5,Ca5,Da5))),
    (not(p_nquadril(A6,B6,C6,D6));
     retractall(p_nquadril(Aa6,Ba6,Ca6,Da6))).
```

```
            /* general procedures */

/* substitute term1 by term2 */

substitute(New,Old,Old1,New):-
   Old==Old1,
   !.
substitute(New,Old,Old1,Old1):-
   var(Old1),
   !.
substitute(New,Old,Val,Val):-
   atomic(Val),
   !.
substitute(New,Old,Val,NewVal):-
   Val=..[Fn|Args],
   subst_fn(New,Old,Fn,NewFn),
   subst_args(New,Old,Args,NewArgs),
   NewVal=..[NewFn|NewArgs].
subst_args(_,_,[],[]):-
   !.
subst_args(New,Old,[Arg|Args],[NewArg|NewArgs]):-
   substitute(New,Old,Arg,NewArg),
   subst_args(New,Old,Args,NewArgs).
subst_fn(New,Old,Fn,Fn):-
   Old\==Fn.
subst_fn(New,Old,Fn,NewFn):-
   Old==Fn,
   substitute(New,Old,Fn,NewFn).

/* substitute all terms of a list by term2 */

subst_all(New,Old,[Val|Vals],[NewVal|NewVals]):-
   substitute(New,Old,Val,NewVal),
   subst_all(New,Old,Vals,NewVals).
subst_all(_,_,[],[]):-
!.

/* substitute all terms of list1 by all terms of list2 */

sublist_all([New|News],[Old|Olds],Val,NewVal):-
   subst_all(New,Old,Val,V1New),
   sublist_all(News,Olds,V1New,NewVal).
sublist_all([],[],X,X):-
   !.

/* substitute list1 of terms by list2 of terms */

sublist([New|News],[Old|Olds],Val,NewVal):-
   substitute(New,Old,Val,V1New),
   sublist(News,Olds,V1New,NewVal).
sublist([],[],X,X):-
!.

/* form a list of factors separated by ';' or ',' */

single_factor_list((X,Y),L):-
    nonvar(X),
    nonvar(Y),
    X=(X1;X2),
   single_factor_list((Y,X),L).
single_factor_list((X,Y),[X|L]):-
   single_factor_list(Y,L).
single_factor_list((X;Y),[X|L]):-
   single_factor_list(Y,L).
single_factor_list(X,[X]).

list(List):-
   List=[L|Ls].

/* transform (A,B,...,Z) to [A,B,...,Z] */

form_list((X,Y),[X|L]):-
   form_list(Y,L),
   !.
form_list(X,[X]).

/* transform [A,B,...,Z] to (A,B,...,Z) */

form_set(List,Set):-
   form_list(Set,List).

disjunct_set((X;Y),[X|L]):-
   disjunct_set(Y,L),
   !.
disjunct_set(X,[X]).

append([A|B],C,[A|D]):-
   append(B,C,D),
   !.
append([],X,X).

/* general member */

member(X,[R|_]):-
   nonvar(X),
   nonvar(R),
   X==R.
member(X,[R|_]):-
   var(X),
   var(R),
   X==R.
member(X,[_|Y]):-
   member(X,Y).

/* simple member */

s_member(X,[X|_]).
s_member(X,[_|Y]):-
   s_member(X,Y).

/* equivalent member */

belong(X,[X|_]).
belong(X,[R|_]):-
   same(X,R).
belong(X,[_|Y]):-
   belong(X,Y).

subset([A|X],Y):-
   member(A,Y),
   subset(X,Y).
subset([],Y).

union([],X,X).
union([X|R],Y,Z):-
   member(X,Y),
   !,
   union(R,Y,Z).
union([X|R],Y,[X|Z]):-
   union(R,Y,Z).
```

```
/* union of elements of alist */

union_list([L|Ls],Union):-
   un(X),
   union(X,L,U),
   retract(un(X)),
   assert(un(U)),
   union_list(Ls,Union).
union_list([],Union):-
   un(Union),
   retract(un(Union)).

/* equivalent union */

equ_union([],X,X).
equ_union([X|R],Y,Z):-
   belong(X,Y),
   !,
   equ_union(R,Y,Z).
equ_union([X|R],Y,[X|Z]):-
   equ_union(R,Y,Z).

intersection([],X,[]).
intersection([X|R],Y,[X|Z]):-
   member(X,Y),
   !,
   intersection(R,Y,Z).
intersection([X|R],Y,Z):-
   intersection(R,Y,Z).

/* equivalent intersection */

equ_intersection([],X,[]).
equ_intersection([X|R],Y,[X|Z]):-
   belong(X,Y),
   !,
   equ_intersection(R,Y,Z).
equ_intersection([X|R],Y,Z):-
   equ_intersection(R,Y,Z).

difference([X|A],Y,[X|Z]):-
   not(member(X,Y)),
   !,
   difference(A,Y,Z).
difference([X|A],Y,Z):-
   difference(A,Y,Z).
difference([],X,[]).

s_difference([X|A],Y,[X|Z]):-
   not(s_member(X,Y)),
   !,
   s_difference(A,Y,Z).
s_difference([X|A],Y,Z):-
   s_difference(A,Y,Z).
s_difference([],X,[]).

/* equivalent difference */

equ_difference([X|A],Y,[X|Z]):-
   not(belong(X,Y)),
   !,
   equ_difference(A,Y,Z).
equ_difference([X|A],Y,Z):-
   equ_difference(A,Y,Z).
equ_difference([],X,[]).

equal(X,Y):-
   difference(X,Y,D1),
   difference(Y,X,D2),
   D1=[],
   D2=[].

s_equal([X|Xs],List):-
   s_member(X,List),
   s_equal(Xs,List),
   !.
s_equal([],L).

/* A equivalent B */

same(A,B):-
   A=..[FnA|ArgsA],
   B=..[FnB|ArgsB],
   FnA=FnB,
   equal(ArgsA,ArgsB).

/* commutate arguments of X */

commutate(X,NewX):-
   X=..[FnX|ArgsX],
   ArgsX=[F|Rs],
   append(Rs,[F],NewArgsX),
   NewX=..[FnX|NewArgsX],
   !.

split((A,B),A,B).
split(A,A).

find_all(W,[X|Xs]):-
   X=rline(A,B),
   not(member(X,W)),
   find_all([X|W],Xs).
find_all(W,[]).

/* form pair of different elements */

ddiffer(L1,L2,[L1,L2],Index):-
   L1\=L2,
   count(Index0,Index).
ddiffer(L1,L2,[],Index0):-
   count(Index0,Index).

/* form lists of pairs of different elements */

ddiffer_list([L1|Ls1],[L2|Ls2],[DifList|DifLists],[Index|Indices]):-
   assert(cn(0)),
   ddiffer(L1,L2,DifList,Index),
   retract(cn(N)),
   ddiffer_list(Ls1,Ls2,DifLists,Indices).
ddiffer_list([],[],[],[]).

differ([L1|Ls1],[L2|Ls2],[[L1,L2]|DifList],[Index|Indices]):-
   L1\=L2,
   count(Index0,Index),
   differ(Ls1,Ls2,DifList,Indices).
differ([L1|Ls1],[L2|Ls2],DifList,Indices):-
   count(Index0,Index),
   differ(Ls1,Ls2,DifList,Indices).
differ([],[],[],[]).

differ_list([L1|Ls1],[L2|Ls2],[DifList|DifLists],[Index|Indices]):-
   assert(cn(0)),
```

```
    differ(L1,L2,DifList,Index),
    retract(cn(N)),
    differ_list(Ls1,Ls2,DifLists,Indices).
differ_list([],[],[],[]).

diff_pair([D1|Ds1],[D2|Ds2],[[D1,D2]|D]):-
   D1\=D2,
   diff_pair(Ds1,Ds2,D).
diff_pair([D1|Ds1],[D2|Ds2],D):-
   diff_pair(Ds1,Ds2,D).
diff_pair([],[],[]).

minimum(Old,New,New):-
   New<Old.
minimum(Old,New,Old).

minimum([Old,New],New):-
   New<Old.
minimum([Old,New],Old).

maximum(Old,New,New):-
   New>Old.
maximum(Old,New,Old).

maximum([Old,New],New):-
   New>Old.
maximum([Old,New],Old).

/* find no. of list elements */

list_count(0,[],0).
list_count(0,[L|Ls],N):-
   ((Ls=[],
     N=1);
    Ls\=[]),
   list_count(1,Ls,N),
   !.
list_count(N1,[L|Ls],N):-
   N2 is N1+1,
   ((Ls=[],
     N=N2);
    Ls\=[]),
   list_count(N2,Ls,N).
list_count(N1,[],N).

count(N,N1):-
   cn(N),
   N1 is N+1,
   assert(cn(N1)),
   retract(cn(N)).

list_assert(S):-
   S\=[H|B],
   S\=[],
   form_list(S,L),
   list_assert(L).
list_assert([L|Ls]):-
   assert(L),
   list_assert(Ls).
list_assert([]).

list_retract(S):-
   S\=[H|B],
   S\=[],
   form_list(S,L),
   list_retract(L).
list_retract([L|Ls]):-
   retract(L),
   list_retract(Ls).
list_retract([]).

/* remove an element from a list */

exclude(P,[L|Ls],[L|X]):-
   P\=L,
   exclude(P,Ls,X),
   !.
exclude(P,[L|Ls],X):-
   P=L,
   exclude(P,Ls,X),
   !.
exclude(P,[],[]):-
   !.

excl_pred(PredName,[L|Ls],[L|X]):-
   L=..[FnL|ArgL],
   PredName\=FnL,
   excl_pred(PredName,Ls,X),
   !.
excl_pred(PredName,[L|Ls],X):-
   excl_pred(PredName,Ls,X),
   !.
excl_pred(PredName,[],[]):-
   !.

/* remove a list of elements from a list */

exclude_list([P|Ps],L,X):-
   exclude(P,L,Y),
   exclude_list(Ps,Y,X),
   !.
exclude_list(P,L,L).
```

```
    /* r_rules for elementary-concept learner */

r_line(A,B):-
   (conn(A,B,N);
    conn(B,A,N)),
   atom(A),
   atom(B),
   integer(N),
   A\=B,
   N>0,
   N=<2.

r_a_line(X,Y):-
   conn(X,Y,1);
   conn(Y,X,1).

r_b_line(X,Y):-
   conn(X,Y,2);
   conn(Y,X,2).

r_c_line(X,Y):-
   conn(X,Y,0);
   conn(Y,X,0).

r_set_atr(A,B,C):-
   atrian(A,B,C);
   atrian(A,C,B);
   atrian(B,A,C);
   atrian(B,C,A);
   atrian(C,A,B);
   atrian(C,B,A).


r_p_trian(A,B,C):-
   line(A,B),
   line(B,C),
   line(C,A),
   poin_trn(A,B,C).

r_set_ptr(A,B,C):-
   ptrian(A,B,C);
   ptrian(A,C,B);
   ptrian(B,A,C);
   ptrian(B,C,A);
   ptrian(C,A,B);
   ptrian(C,B,A).

r_poin_trn(A,B,C):-
   point_in_trn(A,B,C);
   point_in_trn(A,C,B);
   point_in_trn(B,A,C);
   point_in_trn(B,C,A);
   point_in_trn(C,A,B);
   point_in_trn(C,B,A).

r_c_quadril(A,B,C,D):-
   line(A,B),
   line(B,C),
   line(C,D),
   line(D,A),
   A\=C,
   B\=D,
   not(line(A,C)),
   not(line(B,D)),
   not(poin_qul(A,B,C,D)),
   not(nocnvx(A,B,C,D)).

r_set_aqu(W,X,Y,Z):-
   aquadril(W,X,Y,Z);
   aquadril(W,Z,Y,X);
   aquadril(X,W,Z,Y);
   aquadril(X,Y,Z,W);
   aquadril(Y,X,W,Z);
   aquadril(Y,Z,W,X);
   aquadril(Z,W,X,Y);
   aquadril(Z,Y,X,W).

r_p_c_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   poin_qul(W,X,Y,Z),
   not(nocnvx(W,X,Y,Z)).

r_set_pqu(W,X,Y,Z):-
   pquadril(W,X,Y,Z);
   pquadril(W,Z,Y,X);
   pquadril(X,W,Z,Y);
   pquadril(X,Y,Z,W);
   pquadril(Y,X,W,Z);
   pquadril(Y,Z,W,X);
   pquadril(Z,W,X,Y);
   pquadril(Z,Y,X,W).

r_nc_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   not(poin_qul(W,X,Y,Z)),
   nocnvx(W,X,Y,Z).

r_set_nqu(W,X,Y,Z):-
   nquadril(W,X,Y,Z);
   nquadril(W,Z,Y,X);
   nquadril(X,W,Z,Y);
   nquadril(X,Y,Z,W);
   nquadril(Y,X,W,Z);
   nquadril(Y,Z,W,X);
   nquadril(Z,W,X,Y);
   nquadril(Z,Y,X,W).

r_p_nc_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   poin_qul(W,X,Y,Z),
   nocnvx(W,X,Y,Z).

r_set_p_nqu(W,X,Y,Z):-
   p_nquadril(W,X,Y,Z);
   p_nquadril(W,Z,Y,X);
   p_nquadril(X,W,Z,Y);
   p_nquadril(X,Y,Z,W);
   p_nquadril(Y,X,W,Z);
   p_nquadril(Y,Z,W,X);
   p_nquadril(Z,W,X,Y);
   p_nquadril(Z,Y,X,W).

r_poin_qul(W,X,Y,Z):-
   point_in_qul(W,X,Y,Z);
   point_in_qul(W,Z,Y,X);
   point_in_qul(X,W,Z,Y);
   point_in_qul(X,Y,Z,W);
   point_in_qul(Y,X,W,Z);
   point_in_qul(Y,Z,W,X);
   point_in_qul(Z,W,X,Y);
   point_in_qul(Z,Y,X,W).

r_nocnvx(W,X,Y,Z):-
   non_convex(W,X,Y,Z);
   non_convex(W,Z,Y,X);
   non_convex(X,W,Z,Y);
   non_convex(X,Y,Z,W);
   non_convex(Y,X,W,Z);
   non_convex(Y,Z,W,X);
   non_convex(Z,W,X,Y);
   non_convex(Z,Y,X,W).
```

```
            /* recognizer */


/* consider a scene of 3-D figures */

see:-
   init,
   sim,
   consult(scene),
   nl,
   write('      segmenting scene . . .'),
   nl,
   all_s_planes([],S_PlnList),
   nl,
   write('      segmented plane(s)'),
   nl,
   nl,
   print_list(S_PlnList),
   nl,
   tell(vertex),
   d_vrtx_list(S_PlnList),
   told,
   link,
   consult(spec_feat),
   nl,
   write('      batabase'),
   nl,
   nl,
   write('      scene'),
   listing(conn),
   nl,
   write('      special feature(s)'),
   nl,
   spec_feats,
   nl,
   write('      ready for recognition'),
   nl.

/* reconsider same scene */

resee:-
   clr_data,
   clr_a_figures,
   retrieve(scene,ConnList),
   list_assert(ConnList),
   consult(spec_feat),
   nl,
   write('      batabase'),
   nl,
   nl,
   write('      scene'),
   listing(conn),
   nl,
   write('      special feature(s)'),
   nl,
   spec_feats,
   nl,
   write('      ready for recognition'),
   nl.

/* create scene */

sim:-
   shell("simulator").

/* calculate special features */

link:-
   shell("link").

/* single-view recognition */

rec:-
   all_r_lines(ConnList),
   store(scene,ConnList),
   list_assert(ConnList),
   fig_choice(FigChoList),
   visibility(VisFcPc),
   nl,
   write('      recognizing 2-D figure(s) . . .'),
   nl,
   all_planes([],PlnList),
   nl,
   ((PlnList\=[],
     nl,
     write('      visible  face(s)'),
     nl,
     nl,
     print_list(PlnList),
     nl);
    PlnList=[]),
   all_p_planes([],P_PlnList),
   ((P_PlnList\=[],
     nl,
     write('      possible  face(s)'),
     nl,
     nl,
     print_list(P_PlnList),
     nl);
    P_PlnList=[]),
   nl,
   write('      recognizing 3-D figure(s) . . .'),
   nl,
   all_s_3D_figures(FigChoList,VisFcPc,[],FigList),
   ((FigList\=[],
     nl,
     write('      single_view figure(s)'),
     nl,
     nl,
     fig_write(FigList),
     nl,
     nl,
     write('            assumption ?    '),
     read(Asm),
     nl,
     ((Asm=y,
       assume(VrtxList),
       nl,
       write('            multiple-view recognition ?    '),
       read(Mlt),
       nl,
       ((Mlt=y,
         mlt_rec(VrtxList));
        Mlt=n));
      Asm=n));
    (FigList=[],
     nl,
     nl,
     ((FigChoList=[],
       VisFcPo=<17,
       write('non-recognizable'));
       write('no such')),
```

```
        write(' 3-D figure(s)'),
        nl)),
    nl,
    write('      end of recognition'),
    nl.

/* make alternative assumpion */

alt:-
    clr_data,
    nl,
    write('      new assumption'),
    nl,
    retrieve(conns,Conns),
    list_assert(Conns),
    assume(VrtxList),
    nl,
    write('      multiple-view recognition'),
    nl,
    mlt_rec(VrtxList).

/* multiple-view recognition */

mlt_rec(VrtxList):-
    all_r_conns([],Conns),
    ((retrieve(scene,ConnList),
      fig_conn_assert(VrtxList,ConnList),
      retrieve(hidden,HdnLnList),
      list_assert(HdnLnList),
      m_fig);
     (nl,
      write('non-recognizable 3-D figure'),
      nl)).

/* remove from database */

intl:-
    clr_a_figures,
    all_r_conns([],Conns).

/* add conn's of certain figure to database */

fig_conn_assert(VxList,[C|Cs]):-
    C=..[conn,A,B,2],
    ((s_member(A,VxList),
      s_member(B,VxList),
      assert(C),
      fig_conn_assert(VxList,Cs));
     fig_conn_assert(VxList,Cs)).
fig_conn_assert(VxList,[]).

visibility(VisFacePcnt):-
    nl,
    write('              any particular face-visibility ?    '),
    read(Vis),
    nl,
    ((Vis=y,
      nl,
      write('      insert percentage of face-visibility'),
      nl,
      nl,
      read(VisFacePcnt),
      nl);
     VisFacePcnt=0).

fig_choice(FigChoList):-
    nl,
    write('              looking for any particular 3-D figure(s) ?    '),
    read(Fig),
    nl,
    ((Fig=y,
      nl,
      write('      insert list of 3-D figures'),
      nl,
      nl,
      write(' with format [FigFn1,FigFn2,...,FigFnN]'),
      nl,
      nl,
      read(FigChoList),
      nl,
      nl);
     FigChoList=[]).

/* segmentation definitions */

s_plane(quadrilateral(W,X,Y,Z)):-
    s_quadril(W,X,Y,Z).
s_plane(triangle(X,Y,Z)):-
    s_trian(X,Y,Z).

s_quadril(W,X,Y,Z):-
    line(W,X),
    line(X,Y),
    line(Y,Z),
    line(Z,W),
    W\=Y,
    X\=Z,
    not(line(W,Y)),
    not(line(X,Z)).

s_trian(X,Y,Z):-
    line(X,Y),
    line(Y,Z),
    line(Z,X).

/* convert vertex list [vN,vM,...] to integer list [N M ... 0] */

d_vrtx([X|Xs]):-
    name(X,[X1|X2]),
    string(X2),
    tab(1),
    d_vrtx(Xs),
    !.
d_vrtx([]):-
    write('0'),
    nl.

string([X|Xs]):-
    name(N,[X]),
    write(N),
    string(Xs),
    !.
string([]).

d_vrtx_list([L|Ls]):-
    L=..[FnL|ArgsL],
    d_vrtx(ArgsL),
    d_vrtx_list(Ls),
    !.
d_vrtx_list([]):-
    write('-1'),
    nl.
```

```
/* remove from database */

init:-
    clr_data,
    clr_a_p_figs,
    clr_a_figures.

clr_aplane:-
    retractall(atrian(A,B,G)),
    retractall(aquadril(C,D,E,F)),
    retractall(nquadril(S,T,U,V)).

clr_a_figures:-
    ((not(a_box1(A1,B1,C1,D1,E1,F1,G1,H1));
     retractall(a_box1(A1a,B1a,C1a,D1a,E1a,F1a,G1a,H1a))),
    (not( a_box2(A2,B2,C2,D2,E2,F2,G2,H2));
     retractall( a_box2(A2a,B2a,C2a,D2a,E2a,F2a,G2a,H2a))),
    (not(a_box3(A3,B3,C3,D3,E3,F3,G3));
     retractall(a_box3(A3a,B3a,C3a,D3a,E3a,F3a,G3a))),
    (not(a_prism1(A4,B4,C4,D4,E4,F4));
     retractall(a_prism1(A4a,B4a,C4a,D4a,E4a,F4a))),
    (not(a_prism1a(A5,B5,C5,D5,E5,F5));
     retractall(a_prism1a(A5a,B5a,C5a,D5a,E5a,F5a))),
    (not(a_prism2(A6,B6,C6,D6,E6,F6));
     retractall(a_prism2(A6a,B6a,C6a,D6a,E6a,F6a))),
    (not(a_prism2a(A7,B7,C7,D7,E7,F7));
     retractall(a_prism2a(A7a,B7a,C7a,D7a,E7a,F7a))),
    (not(a_prism3(A8,B8,C8,D8,E8,F8));
     retractall(a_prism3(A8a,B8a,C8a,D8a,E8a,F8a))),
    (not(a_pyram1(A9,B9,C9,D9,E9));
     retractall(a_pyram1(A9a,B9a,C9a,D9a,E9a))),
    (not(a_pyram1a(A10,B10,C10,D10,E10));
     retractall(a_pyram1a(A10a,B10a,C10a,D10a,E10a))),
    (not(a_pyram2(A11,B11,C11,D11,E11));
     retractall(a_pyram2(A11a,B11a,C11a,D11a,E11a))),
    (not(a_pyram2a(A12,B12,C12,D12,E12));
     retractall(a_pyram2a(A12a,B12a,C12a,D12a,E12a))),
    (not(a_pyram3(A13,B13,C13,D13,E13));
     retractall(a_pyram3(A13a,B13a,C13a,D13a,E13a))),
    (not(a_pyram5(A14a,B14a,C14a,D14a));
     retractall(a_pyram5(A14,B14,C14,D14))),
    (not(a_tetra1(A15,B15,C15,D15));
     retractall(a_tetra1(A15a,B15a,C15a,D15a))),
    (not(a_tetra2(A16,B16,C16,D16));
     retractall(a_tetra2(A16a,B16a,C16a,D16a))),
    (not(a_tetra3(A17,B17,C17));
     retractall(a_tetra3(A17a,B17a,C17a))).

/* multiple-view recognizer */

m_fig:-
    nl,
    write('      recognizing . . .'),
    nl,
    all_m_figures([],FigList),
    nl,
    nl,
    write('      multiple-view figure'),
    nl,
    nl,
    ((FigList=[],
      write('non-recognizable 3-D figure'),
      nl,
      nl);
      print_list(FigList)).

m_figure(truncated_pyramid(A,B,C,D,E,F,G,H)):-
    box(A,B,C,D,E,F,G,H),
    mark_box(A,B,C,D,E,F,G,H).
m_figure(prism(A,B,C,D,E,F)):-
    prism(A,B,C,D,E,F),
    mark_prism(A,B,C,D,E,F).
m_figure(pyramid(A,B,C,D,E)):-
    pyram(A,B,C,D,E),
    mark_pyram(A,B,C,D,E).
m_figure(tertahedron(A,B,C,D)):-
    tetra(A,B,C,D),
    mark_tetra(A,B,C,D).

/* unconnected-figure condition */

non_contour_free(List):-
    line(X,Y),
    (member(X,List),
     not(member(Y,List))).

/* single-view recognizer */

p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),86,a1)):-
    a_box1(A,B,C,D,E,F,G,H);
    (p_box1(A,B,C,D,E,F,G,H),
     VrtxList=[A,B,C,D,E,F,G,H],
     not(non_contour_free(VrtxList)),
     mark_bx1(A,B,C,D,E,F,G,H)).
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),67,a3)):-
    a_box2(A,B,C,D,E,F,G,H);
    (p_box2(A,B,C,D,E,F,G,H),
     VrtxList=[A,B,C,D,E,F,G,H],
     not(non_contour_free(VrtxList)),
     mark_bx2(A,B,C,D,E,F,G,H)).
p_figure((prism(A,B,C,D,E,F),80,a1)):-
    a_prism1(A,B,C,D,E,F);
    (p_prism1(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr1(A,B,C,D,E,F)).
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),50,a4)):-
    a_box3(A,B,C,D,E,F,G);
    (p_box3(A,B,C,D,E,F,G),
     VrtxList=[A,B,C,D,E,F,G],
     not(non_contour_free(VrtxList)),
     mark_bx3(A,B,C,D,E,F,G)).
p_figure((prism(A,B,C,D,E,F),80,a1)):-
    a_prism1a(A,B,C,D,E,F);
    (p_prism1a(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr1a(A,B,C,D,E,F)).
p_figure((prism(A,B,C,D,E,F),60,a3)):-
    a_prism2(A,B,C,D,E,F);
    (p_prism2(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr2(A,B,C,D,E,F)).
p_figure((prism(A,B,C,D,E,F),60,a2)):-
    a_prism3(A,B,C,D,E,F);
    (p_prism3(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr3(A,B,C,D,E,F)).
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),33,a3)):-
```

```
    a_prism3(A,B,C,D,E,F);
    (p_prism3(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr3(A,B,C,D,E,F)).
p_figure((pyramid(A,B,C,D,E),80,a1)):-
    a_pyram1(A,B,C,D,E);
    (p_pyram1(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py1(A,B,C,D,E)).
p_figure((triangular_prism(A,B,C,D,E,F),40,a2)):-
    a_prism2a(A,B,C,D,E,F);
    (p_prism2a(A,B,C,D,E,F),
     VrtxList=[A,B,C,D,E,F],
     not(non_contour_free(VrtxList)),
     mark_pr2a(A,B,C,D,E,F)).
p_figure((pyramid(A,B,C,D,E),60,a2)):-
    a_pyram2(A,B,C,D,E);
    (p_pyram2(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py2(A,B,C,D,E)).
p_figure((pyramid(A,B,C,D,E),40,a2)):-
    a_pyram3(A,B,C,D,E);
    (p_pyram3(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py3(A,B,C,D,E)).
p_figure((pyramid(A,B,C,D,E),80,a1)):-
    a_pyram1a(A,B,C,D,E);
    (p_pyram1a(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py1a(A,B,C,D,E)).
p_figure((tetrahedron(A,B,C,D),75,a1)):-
    a_tetra1(A,B,C,D);
    (p_tetra1(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_te1(A,B,C,D)).
p_figure((pyramid(A,B,C,D,E),60,a3)):-
    a_pyram2a(A,B,C,D,E);
    (p_pyram2a(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py2a(A,B,C,D,E)).
p_figure((prism(A,B,C,D,E,F),33,a4)):-
    a_pyram3(A,B,C,D,E);
    (p_pyram3(A,B,C,D,E),
     VrtxList=[A,B,C,D,E],
     not(non_contour_free(VrtxList)),
     mark_py3(A,B,C,D,E)).
p_figure((tetrahedron(A,B,C,D),50,a2)):-
    a_tetra2(A,B,C,D);
    (p_tetra2(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_te2(A,B,C,D)).
p_figure((pyramid(A,B,C,D,E),20,a4)):-
    a_tetra2(A,B,C,D);
    (p_tetra2(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_te2(A,B,C,D)).
p_figure((pyramid(A,B,C,D,E),25,a5)):-
    a_pyram4(A,B,C,D);
    (p_pyram4(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_py4(A,B,C,D)).
p_figure((prism(A,B,C,D,E,F),20,a6)):-
    a_pyram4(A,B,C,D);
    (p_pyram4(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_py4(A,B,C,D)).
p_figure((truncated_pyramid(A,B,C,D,E,F,G,H),17,a7)):-
    a_pyram4(A,B,C,D);
    (p_pyram4(A,B,C,D),
     VrtxList=[A,B,C,D],
     not(non_contour_free(VrtxList)),
     mark_py4(A,B,C,D)).
p_figure((tetrahedron(A,B,C,D),25,a5)):-
    a_tetra3(A,B,C);
    (p_tetra3(A,B,C),
     VrtxList=[A,B,C],
     not(non_contour_free(VrtxList)),
     mark_te3(A,B,C)).
p_figure((pyramid(A,B,C,D,E),20,a6)):-
    a_tetra3(A,B,C);
    (p_tetra3(A,B,C),
     VrtxList=[A,B,C],
     not(non_contour_free(VrtxList)),
     mark_te3(A,B,C)).
p_figure((prism(A,B,C,D,E,F),17,a7)):-
    a_tetra3(A,B,C);
    (p_tetra3(A,B,C),
     VrtxList=[A,B,C],
     not(non_contour_free(VrtxList)),
     mark_te3(A,B,C)).

/* visible-face definitions */

plane(convex_quadrilateral(A,B,C,D)):-
    c_quadril(A,B,C,D),
    mark_aqu(A,B,C,D).
plane(non_convex_quadrilateral(A,B,C,D)):-
    nc_quadril(A,B,C,D),
    mark_nqu(A,B,C,D).
plane(triangle(A,B,C)):-
    trian(A,B,C),
    mark_atr(A,B,C).

/* possible-face definitions */

p_plane((convex_quadrilateral(A,B,C,D),a1)):-
    p_c_quadril(A,B,C,D),
    mark_pqu(A,B,C,D).
p_plane((non_convex_quadrilateral(A,B,C,D),a1)):-
    p_nc_quadril(A,B,C,D),
    mark_pnqu(A,B,C,D).
p_plane((triangle(A,B,C),a1)):-
    p_trian(A,B,C),
    mark_ptr(A,B,C).

/* output alternative names of possible 3-D figure */

fig_write([W1|W1s]):-
    write(W1),
    nl,
    ((W1s\=[],
```

```
      tab(8),
      W1=(F1,P1,A1),
      W1s=[W2|W2s],
      W2=(F2,P2,A2),
      F1=..[F1fn|F1args],
      F2=..[F2fn|F2args],
      intersection(F1args,F2args,F12args),
      ((F12args\=[],
        write('or'));
       (nl,
        nl)));
    W1s=[]),
   nl,
   fig_write(W1s),
   !.
fig_write([]).

/* make an assumpion about current figure */

assume(VrtxList):-
   nl,
   write('      insert assumption_code'),
   nl,
   nl,
   write('         a1 : no hidden lines or points behind possible face'),
   nl,
   write('         a2 : hidden (triangle) line'),
   nl,
   write('         a3 : hidden (qudrilateral) line'),
   nl,
   write('         a4 : hidden (qudrilateral) point'),
   nl,
   write('         a5 : entirely hidden  point'),
   nl,
   write('         a6 : entirely hidden  line'),
   nl,
   write('         a7 : entirely hidden  face'),
   nl,
   nl,
   nl,
   read(AsmCode),
   nl,
   nl,
   write('      insert vertex-list to specify 3-D figure'),
   nl,
   write('            with format [v1,v2,...,vN]'),
   nl,
   nl,
   read(VrtxList),
   nl,
   nl,
   write('      assuming . . .'),
   nl,
   ((AsmCode\=a1,
     all_conns(0,[],Conn0List),
     all_conns(1,[],Conn1List),
     union(Conn0List,Conn1List,ConnList),
     store(conns,ConnList),
     all_hdn_lines(AsmCode,[],HdnLines),
     ((AsmCode=a4,
       all_hdn_quadrils([],HdnQuList),
       HdnQuList=[HdnQu|HdnQus],
       check_hqu(HdnQus),
       list_retract(HdnQuList));
      (AsmCode=a5,
       entirely_hdn_point);
      (AsmCode=a6,
       entirely_hdn_line);
      (AsmCode=a7,
       entirely_hdn_face)),
     all_r_hdn_lines([],AllHdnLines),
     fig_hdn_lines(VrtxList,AllHdnLines,FigHdnLines),
     lines(2,FigHdnLines,HdnLnList),
     AsmHdnLines=HdnLnList,
     store(hidden,FigHdnLines));
    AsmHdnLines=['none,also no hidden points']),
   nl,
   write('      assumed hidden line(s)'),
   nl,
   nl,
   print_list(AsmHdnLines),
   nl,
   clr_data.

/* form hidden-line list */

hidden_lines(W,[R|Y]):-
   conn(A,B,2),
   R=hidden_line(A,B),
   not(belong(R,W)),
   hidden_lines([R|W],Y),
   !.
hidden_lines(W,[]).

/* form hidden-line list for certain figure */

fig_hdn_lines(F,[A|As],[H|W]):-
   A=..[conn,X,Y,2],
   (not(s_member(X,[h1,h2,h3,h4]));
    not(s_member(Y,[h1,h2,h3,h4]))),
   intersection(F,[X,Y],FAargs),
   FAargs\=[],
   H=A,
   fig_hdn_lines(F,As,W).
fig_hdn_lines(F,[A|As],W):-
   A=..[conn,X,Y,2],
   (not(s_member(X,[h1,h2,h3,h4]));
    not(s_member(Y,[h1,h2,h3,h4]))),
   intersection(F,[X,Y],FAargs),
   FAargs=[],
   fig_hdn_lines(F,As,W).
fig_hdn_lines(F,[A|As],[A|W]):-
   A=..[conn,X,Y,2],
   s_member(X,[h1,h2,h3,h4]),
   s_member(Y,[h1,h2,h3,h4]),
   fig_hdn_lines(F,As,W).
fig_hdn_lines(F,[],[]).
fig_hdn_lines(F,[A|As],[H|W]):-
   (A=..[conn,X,Y,2],
    (((not(s_member(X,[h1,h2,h3,h4]));
       not(s_member(Y,[h1,h2,h3,h4]))),
      intersection(F,[X,Y],FAargs),
      ((FAargs\=[],
        H=A,
        fig_hdn_lines(F,As,W));
       (FAargs=[],
        fig_hdn_lines(F,As,[H|W]))));
     fig_hdn_lines(F,As,[H|W])))).
fig_hdn_lines(F,[],[]).

/* for assumptions a2,a3,a4 add all hidden lines to database */
```

```
all_hdn_lines(C,W,[R|X]):-
   ((C=a2,
     htr_line(A,B));
    (C=a3,
     hqu_line(A,B));
    (C=a4,
     B=h1,
     h_point(A,B))),
   R=line(A,B),
   not(belong(R,W)),
   all_hdn_lines(C,[R|W],X),
   !.
all_hdn_lines(C,W,[]).

/* form hidden-quadrilateral list */

all_hdn_quadrils(_,[R|X]):-
   c_quadril(A,B,C,h1),
   not(set_hqu(A,B,C,h1)),
   R=hquadril(A,B,C,h1),
   assert(R),
   all_hdn_quadrils([R|_],X),
   !.
all_hdn_quadrils(R,[]).

/* perform impossible-hidden-quadrilateral test in case a4 */

check_hqu([L|Ls]):-
   L=hquadril(A,B,C,h1),
   set_hqu(A,B,C,h1),
   set_non_hqu(A,B,C,h1,X),
   assert(conn(B,h1,2)),
   check_hqu(Ls),
   !.
check_hqu([L|Ls]):-
   L=hquadril(A,B,C,h1),
   set_hqu(A,B,C,h1),
   not(set_non_hqu(A,B,C,h1,X)),
   check_hqu(Ls),
   !.
check_hqu([]).

/* hidden-quadrilateral set */

set_hqu(W,X,Y,Z):-
   hquadril(W,X,Y,Z);
   hquadril(W,Z,Y,X);
   hquadril(X,W,Z,Y);
   hquadril(X,Y,Z,W);
   hquadril(Y,X,W,Z);
   hquadril(Y,Z,W,X);
   hquadril(Z,W,X,Y);
   hquadril(Z,Y,X,W).

/* impossible hidden-quadrilateral set */

set_non_hqu(A,B,C,D,X):-
   (set_hqu(X,B,C,D);
    set_hqu(A,X,C,D);
    set_hqu(A,B,X,D);
    set_hqu(A,B,C,X)),
   X\=A,
   X\=B,
   X\=C,
   X\=D.

/* assumption a2 */

htr_line(B,D):-
   a_line(A,B),
   a_line(A,D),
   A\=D,
   B\=D,
   no_line(B,D),
   not(b_line(A,E)),
   assert(conn(B,D,2)).

/* assumption a3 */

hqu_line(A,D):-
   a_line(A,B),
   a_line(B,C),
   a_line(C,D),
   A\=C,
   A\=D,
   B\=D,
   no_line(A,D),
   no_line(B,D),
   no_line(A,C),
   assert(conn(A,D,2)).

/* assumption a4 */

h_point(X,h1):-
   a_line(A,B),
   a_line(A,D),
   A\=D,
   B\=D,
   B\=h1,
   D\=h1,
   not(a_line(B,D)),
   not(c_line(B,D)),
   ((not(conn(B,h1,2)),
     X=B,
     assert(conn(B,h1,2)));
    (not(conn(D,h1,2)),
     X=D,
     assert(conn(D,h1,2)))).

/* assumption a5 */

entirely_hdn_point:-
   ((a_tetra3(A,B,C),
     VrtxList=[A,B,C]);
    (a_pyram4(A,B,C,D),
     VrtxList=[A,B,C,D])),
   hdn_line_assetrt(VrtxList).

/* add hidden-conn list to database */

hdn_line_assetrt([L|Ls]):-
   assert(conn(L,h1,2)),
   !,
   hdn_line_assetrt(Ls).
hdn_line_assetrt([]).

/* assumption a6 */

entirely_hdn_line:-
   assert(conn(h1,h2,2)),
   ((a_tetra3(A,B,C),
     assert(conn(A,h1,2)),
```

```
      assert(conn(A,h2,2)),
      assert(conn(B,h1,2)),
      assert(conn(C,h1,2)));
     (a_pyram4(A,B,C,D),
      assert(conn(A,h1,2)),
      assert(conn(B,h1,2)),
      assert(conn(C,h2,2)),
      assert(conn(D,h2,2)))).

/* assumption a7 */

entirely_hdn_face:-
   assert(conn(h1,h2,2)),
   assert(conn(h2,h3,2)),
   ((a_tetra3(A,B,C),
     assert(conn(h3,h1,2)));
    (a_pyram4(A,B,C,D),
     assert(conn(h3,h4,2)),
     assert(conn(h4,h1,2)),
     assert(conn(D,h4,2)))),
   assert(conn(A,h1,2)),
   assert(conn(B,h2,2)),
   assert(conn(C,h3,2)).

/* remove all hidden lines from database */

all_r_hdn_lines(_,[W|X]):-
   conn(A,B,2),
   W=conn(A,B,2),
   retract(conn(A,B,2)),
   all_r_hdn_lines([W|_],X),
   !.
all_r_hdn_lines(W,[]).

/* add conn list to database */

conn_assert([L|Ls]):-
   L=conn(A,B,2),
   assert(L),
   conn_assert(Ls),
   !.
conn_assert([L|Ls]):-
   conn_assert(Ls),
   !.
conn_assert([]).
```

```
          /* procedures for figure manipulation */


 /* generalize to line */

 r_1D_fig(line(A,B)):-
    line(A,B),
    retract(conn(A,B,N)).

 /* form list of visible faces */

 v_2D_fig(Fig,FigList):-
    (trian(A,B,C),
     Fig=trian(A,B,C),
     not(belong(trian(A,B,C),FigList)),
     rtr_trn(A,B,C),
     mark_atr(A,B,C));
    (c_quadril(A,B,C,D),
     Fig=c_quadril(A,B,C,D),
     !,
     not(belong(c_quadril(A,B,C,D),FigList)),
     mark_c_line(A,B,C,D),
     rtr_qul(A,B,C,D),
     mark_aqu(A,B,C,D));
    (nc_quadril(A,B,C,D),
     Fig=nc_quadril(A,B,C,D),
     !,
     not(belong(nc_quadril(A,B,C,D),FigList)),
     rtr_nqu(A,B,C,D),
     mark_aqu(A,B,C,D)).

 /* indicate presence of visible triangle by atrian */

 mark_atr(A,B,C):-
    (not(set_atr(A,B,C)),
     assert(atrian(A,B,C)),

 /* remove ptrian , point_in_trn from previews view */

     ((set_ptr(A,B,C),
       rtr_ptr(A,B,C),
       retrieve(view2,(Head,BodyList,ViewVal)),
       excl_pred(point_in_trn,BodyList,NewBodyList),
       store(view2,(Head,NewBodyList,ViewVal)));
      true));
     true.

 /* indicate presence of visible quadrilateral by aquadril */

 mark_aqu(A,B,C,D):-
    (not(set_aqu(A,B,C,D)),
     assert(aquadril(A,B,C,D)),

 /* remove pquadril , point_in_qul from previews view */

     (((set_pqu(A,B,C,D);
        set_non_pqu(A,B,C,D,X)),
       rtr_pqu(A,B,C,D),
       retrieve(view2,(Head,BodyList,ViewVal)),
       excl_pred(point_in_qul,BodyList,NewBodyList),
       store(view2,(Head,NewBodyList,ViewVal)));
      true));
     true.

 /* mark diagonals of convex quadrilateral with c_lines */
```

```
cross_qul(W,X,Y,Z):-
   assert(conn(W,Y,0)),
   assert(conn(X,Z,0)).

/* form list of possible faces */

p_2D_fig(Fig,FigList):-
  (p_trian(A,B,C),
   Fig=p_trian(A,B,C),
   !,
   not(belong(p_trian(A,B,C),FigList)),
   mark_ptr(A,B,C));
  (p_c_quadril(A,B,C,D),
   Fig=p_c_quadril(A,B,C,D),
   !,
   not(belong(p_c_quadril(A,B,C,D),FigList)),
   mark_pqu(A,B,C,D));
  (p_nc_quadril(A,B,C,D),
   Fig=p_nc_quadril(A,B,C,D),
   !,
   not(belong(p_nc_quadril(A,B,C,D),FigList)),
   mark_p_nqu(A,B,C,D)).

/* indicate presence of possible quadrilateral by pquadril */

mark_ptr(A,B,C):-
  ((set_atr(A,B,C);
   (set_ptr(A,B,C)),
   retrieve(view1,(Head,BodyList,ViewVal)),
   excl_pred(point_in_trn,BodyList,NewBodyList),
   rtr_poin_trn(A,B,C),
   store(view1,(Head,NewBodyList,ViewVal)));
   (not(set_ptr(A,B,C)),
    assert(ptrian(A,B,C))).

/*  indicate presenceof possible quadrilateral by pquadril */

mark_pqu(A,B,C,D):-
  (((set_aqu(A,B,C,D);
    set_pqu(A,B,C,D));
    ((set_atr(A,B,C),
     set_ptr(A,B,C));
     (set_atr(A,B,C),
      set_ptr(A,B,C)))),
    retrieve(view1,(Head,BodyList,ViewVal)),
    rtr_poin_qul(A,B,C,D),
    excl_pred(point_in_qul,BodyList,NewBodyList),
    store(view1,(Head,NewBodyList,ViewVal)));
    (not(set_pqu(A,B,C,D)),
     not(set_non_pqu(A,B,C,D,X)),
     assert(pquadril(A,B,C,D))).

/* indicate presence of visible non-convex quadrilateral */

mark_nqu(W,X,Y,Z):-
   not(set_nqu(W,X,Y,Z)),
   mark_c_line(W,X,Y,Z),
   assert(nquadril(W,X,Y,Z))).

mark_c_line(W,X,Y,Z):-
   not(c_line(W,Y)),
   not(c_line(X,Z)),
   cross_qul(W,X,Y,Z).

/* indicate presence of possible non-convex quadrilateral */

mark_p_nqu(W,X,Y,Z):-
   not(set_p_nqu(W,X,Y,Z)),
   assert(p_nquadril(W,X,Y,Z))).

/* form list of ( _r_ : and retract) all ... */

all_r_1D_figs(_,[W|X]):-
   r_1D_fig(W),
   all_r_1D_figs([W|_],X),
   !.
all_r_1D_figs(W,[]).

/* ... 2-D figures */

all_2D_figs(R,[W|X]):-
   (v_2D_fig(W,R);
    p_2D_fig(W,R)),
   all_2D_figs([W|R],X),
   !.
all_2D_figs(W,[]).

all_conns(R,[W|X]):-
   conn(A,B,2),
   W=conn(A,B,2),
   not(belong(W,R)),
   all_conns([W|R],X),
   !.
all_conns(W,[]).

/* ...   a_lines and c_lines making up a figure */

all_r_fig_conns(VrtList,R,[W|X]):-
   ((retract(conn(A,B,1)),
     W=conn(A,B,1),
     not(belong(W,R)),
     member(A,VrtList),
     member(B,VrtList));
    (retract(conn(A,B,0)),
     W=conn(A,B,0),
     not(belong(W,R)),
     member(A,VrtList),
     member(B,VrtList))),
   all_r_fig_conns(VrtList,[W|R],X),
   !.
all_r_fig_conns(VrtList,W,[]).

/* ... set_of_a_figures , set_of_p_figures */

all_r_set_a_p_figs(_,[W|X]):-
   ((W=set_atr(A,B,C),
     rtr_atr(A,B,C));
    (W=set_aqu(A,B,C,D),
     rtr_aqu(A,B,C,D));
    (W=set_nqu(A,B,C,D),
     rtr_nqu(A,B,C,D));
    (W=set_ptr(A,B,C),
     rtr_ptr(A,B,C));
    (W=set_pqu(A,B,C,D),
     rtr_pqu(A,B,C,D));
    (W=set_p_nqu(A,B,C,D),
     rtr_p_nqu(A,B,C,D))),
   all_r_set_a_p_figs([W|_],X),
   !.
all_r_set_a_p_figs(W,[]):-
   not(set_atr(A,B,C)),
   not(set_aqu(A,B,C,D)),
```

```
    not(set_nqu(A,B,C,D)),
    not(set_ptr(A,B,C)),
    not(set_pqu(A,B,C,D)),
    not(set_p_nqu(A,B,C,D)).

/* ... a_figures , p_figures */

all_r_a_p_figs(_,[W|X]):-
   ((W=atrian(A,B,C),
      retract(atrian(A,B,C)));
    (W=aquadril(A,B,C,D),
      retract(aquadril(A,B,C,D)));
    (W=nquadril(A,B,C,D),
      retract(nquadril(A,B,C,D)));
    (W=ptrian(A,B,C),
      retract(ptrian(A,B,C)));
    (W=pquadril(A,B,C,D),
      retract(pquadril(A,B,C,D)));
    (W=p_nquadril(A,B,C,D),
      retract(p_nquadril(A,B,C,D)))),
   all_r_a_p_figs([W|_],X),
   !.
all_r_a_p_figs(W,[]):-
   not(atrian(A,B,C)),
   not(aquadril(A,B,C,D)),
   not(nquadril(A,B,C,D)),
   not(ptrian(A,B,C)),
   not(pquadril(A,B,C,D)),
   not(p_nquadril(A,B,C,D)).

all_a_figs(R,[W|X]):-
   ((atrian(A,B,C),
      W=atrian(A,B,C));
    (aquadril(A,B,C,D),
      W=aquadril(A,B,C,D))),
   not(belong(W,R)),
   all_a_figs([W|R],X),
   !.
all_a_figs(W,[]).

/* ... a_lines , c_lines */

all_r_a_c_lines(_,[W|X]):-
   ((a_line(A,B),
      W=a_line(A,B),
      rtr_a_line(A,B));
    (c_line(A,B),
      W=c_line(A,B),
      rtr_c_line(A,B))),
   all_r_a_c_lines([W|_],X),
   !.
all_r_a_c_lines(W,[]):-
   not(a_line(A,B)),
   not(c_line(A,B)).

/* ... b_lines */

all_r_lines([W|X]):-
   retract(conn(A,B,2)),
   W=conn(A,B,2),
   all_r_lines(X),
   !.
all_r_lines([]):-
   not(conn(A,B,2)).

all_r_a_lines([W|X]):-
   W=conn(A,B,1),
   retract(conn(A,B,1)),
   all_r_a_lines(X),
   !.
all_r_a_lines([]):-
   not(conn(A,B,1)).

all_r_c_lines([W|X]):-
   W=conn(A,B,0),
   retract(conn(A,B,0)),
   all_r_c_lines(X),
   !.
all_r_c_lines([]):-
   not(conn(A,B,0)).

all_r_conns(_,[W|X]):-
   ((a_line(A,B),
      W=conn(A,B,1),
      rtr_a_line(A,B));
    (c_line(A,B),
      W=conn(A,B,0),
      rtr_c_line(A,B))),
   all_r_conns([W|_],X),
   !.
all_r_conns(W,[]):-
   not(a_line(A,B)),
   not(c_line(A,B)).

all_conns(N,W,[R|X]):-
   conn(A,B,N),
   R=conn(A,B,N),
   not(belong(R,W)),
   all_conns(N,[R|W],X).
all_conns(N,W,[]).

all_points_in(W,[R|X]):-
   ((point_in_trn(A,B,C),
      R=point_in_trn(A,B,C));
    (point_in_qul(A,B,C,D),
      R=point_in_qul(A,B,C,D))),
   not(belong(R,W)),
   all_points_in([R|W],X).
all_points_in(W,[]).

all_non_cvx_angles(W,[R|X]):-
   non_convex_angle(A),
   R=non_convex_angle(A),
   not(belong(R,W)),
   all_non_cvx_angles([R|W],X).
all_non_cvx_angles(W,[]).

/* form list of a_lines belonging to the same face */

set_same_face_a_lines(A_Fig,P,[W|R]):-
   a_line(X,Y),
   W=a_line(X,Y),
   not(belong(W,P)),
   ((A_Fig=atrian(A,B,C),
      belong([X,Y],[[A,B],[B,C],[C,A]]));
    (A_Fig=aquadril(A,B,C,D),
      belong([X,Y],[[A,B],[B,C],[C,D],[D,A]]));
    (A_Fig=nquadril(A,B,C,D),
      belong([X,Y],[[A,B],[B,C],[C,D],[D,A]]))),
   set_same_face_a_lines(A_Fig,[W|P],R),
   !.
set_same_face_a_lines(A_Fig,W,[]).
```

```
/* form list of a_lines belonging to the same face , in a list of faces
*/

all_set_same_face_a_lines([F|Fs],[(F,[W|X])|Y]):-
    set_same_face_a_lines(F,[],[W|X]),
    all_set_same_face_a_lines(Fs,Y),
    !.
all_set_same_face_a_lines([F|Fs],X):-
    set_same_face_a_lines(F,[],[]),
    all_set_same_face_a_lines(Fs,X),
    !.
all_set_same_face_a_lines([],[]).

/* form list of conns with face counter N */

lines(N,[C|Cs],[L|X]):-
    C=conn(A,B,N),
    L=line(A,B),
    lines(N,Cs,X),
    !.
lines(N,[C|Cs],L):-
    C\=conn(A,B,N),
    lines(N,Cs,L),
    !.
lines(N,[],[]).

no_line(X,Y):-
    not(a_line(X,Y)),
    not(b_line(X,Y)),
    not(c_line(X,Y)).

notline(X,Y):-
    not(a_line(X,Y)),
    not(b_line(X,Y)).

rtr_a_line(A,B):-
    retract(conn(A,B,1));
    retract(conn(B,A,1)).

/*  rtr : remove from database ... */

rtr_all_a_lines:-
    all_a_lines(Conn1List).

rtr_c_line(A,B):-
    retract(conn(A,B,0));
    retract(conn(B,A,0)).

rtr_all_c_lines:-
    all_c_lines(Conn0List).

rtr_all_a_p_figs:-
    all_set_a_p_figs([],APFigList).

/* ... visible triangle replacement */

rtr_atr(A,B,C):-
    retract(atrian(A,B,C));
    retract(atrian(A,C,B));
    retract(atrian(B,A,C));
    retract(atrian(B,C,A));
    retract(atrian(C,A,B));
    retract(atrian(C,B,A)).

/* decrease face counter of conns of a triangle by 1 */

rtr_trn(A,B,C):-
    (retract(conn(A,B,L));
     retract(conn(B,A,L))),
    L1 is L-1,
    assert(conn(A,B,L1)),
    (retract(conn(B,C,M));
     retract(conn(C,B,M))),
    M1 is M-1,
    assert(conn(B,C,M1)),
    (retract(conn(C,A,N));
     retract(conn(A,C,N))),
    N1 is N-1,
    assert(conn(C,A,N1)),
    !.

/* ... possible triangle replacement */

rtr_ptr(A,B,C):-
    retract(ptrian(A,B,C));
    retract(ptrian(A,C,B));
    retract(ptrian(B,A,C));
    retract(ptrian(B,C,A));
    retract(ptrian(C,A,B));
    retract(ptrian(C,B,A)).

rtr_poin_trn(X,Y,Z):-
    retract(point_in_trn(X,Y,Z));
    retract(point_in_trn(X,Z,Y));
    retract(point_in_trn(Y,X,Z));
    retract(point_in_trn(Y,Z,X));
    retract(point_in_trn(Z,X,Y));
    retract(point_in_trn(Z,Y,X)).

/* decrease face counter of conns of a quadrilateral by 1 */

rtr_qul(A,B,C,D):-
    (retract(conn(A,B,K));
     retract(conn(B,A,K))),
    K1 is K-1,
    assert(conn(A,B,K1)),
    (retract(conn(B,C,L));
     retract(conn(C,B,L))),
    L1 is L-1,
    assert(conn(B,C,L1)),
    (retract(conn(C,D,M));
     retract(conn(D,C,M))),
    M1 is M-1,
    assert(conn(C,D,M1)),
    (retract(conn(D,A,N));
     retract(conn(A,D,N))),
    N1 is N-1,
    assert(conn(D,A,N1)),
    !.

/* ... visible convex-qudrilateral replacement */

rtr_aqu(W,X,Y,Z):-
    retract(aquadril(W,X,Y,Z));
    retract(aquadril(W,Z,Y,X));
    retract(aquadril(X,W,Z,Y));
    retract(aquadril(X,Y,Z,W));
    retract(aquadril(Y,X,W,Z));
    retract(aquadril(Y,Z,W,X));
    retract(aquadril(Z,W,X,Y));
    retract(aquadril(Z,Y,X,W)).
```

```
/* non-possible quadrilateral */

set_non_pqu(A,B,C,D,X):-
    (set_pqu(X,B,C,D);
     set_pqu(A,X,C,D);
     set_pqu(A,B,X,D);
     set_pqu(A,B,C,X)),
    X\=A,
    X\=B,
    X\=C,
    X\=D.

/* ... possible convex-qudrilateral replacement */

rtr_pqu(W,X,Y,Z):-
    retract(pquadril(W,X,Y,Z));
    retract(pquadril(W,Z,Y,X));
    retract(pquadril(X,W,Z,Y));
    retract(pquadril(X,Y,Z,W));
    retract(pquadril(Y,X,W,Z));
    retract(pquadril(Y,Z,W,X));
    retract(pquadril(Z,W,X,Y));
    retract(pquadril(Z,Y,X,W)).

/* ... visible non-convex-qudrilateral replacement */

rtr_nqu(W,X,Y,Z):-
    retract(nquadril(W,X,Y,Z));
    retract(nquadril(W,Z,Y,X));
    retract(nquadril(X,W,Z,Y));
    retract(nquadril(X,Y,Z,W));
    retract(nquadril(Y,X,W,Z));
    retract(nquadril(Y,Z,W,X));
    retract(nquadril(Z,W,X,Y));
    retract(nquadril(Z,Y,X,W)).

/* ... possible non-convex-qudrilateral replacement */

rtr_p_nqu(W,X,Y,Z):-
    retract(p_nquadril(W,X,Y,Z));
    retract(p_nquadril(W,Z,Y,X));
    retract(p_nquadril(X,W,Z,Y));
    retract(p_nquadril(X,Y,Z,W));
    retract(p_nquadril(Y,X,W,Z));
    retract(p_nquadril(Y,Z,W,X));
    retract(p_nquadril(Z,W,X,Y));
    retract(p_nquadril(Z,Y,X,W)).

rtr_poin_qul(W,X,Y,Z):-
    retract(point_in_qul(W,X,Y,Z));
    retract(point_in_qul(W,Z,Y,X));
    retract(point_in_qul(X,W,Z,Y));
    retract(point_in_qul(X,Y,Z,W));
    retract(point_in_qul(Y,X,W,Z));
    retract(point_in_qul(Y,Z,W,X));
    retract(point_in_qul(Z,W,X,Y));
    retract(point_in_qul(Z,Y,X,W)).

/* form list of visible planes */

all_planes(W,[R|X]):-
    plane(R),
    not(belong(R,W)),
    ((R=triangle(A,B,C),
      rtr_trn(A,B,C));
     (R=convex_quadrilateral(A,B,C,D),
      mark_c_line(A,B,C,D),
      rtr_qul(A,B,C,D));
     (R=non_convex_quadrilateral(A,B,C,D),
      mark_c_line(A,B,C,D),
      rtr_nqu(A,B,C,D))),
    all_planes([R|W],X),
    !.
all_planes(W,[]).

/* ... possible planes */

all_p_planes(W,[R|X]):-
    p_plane(R),
    not(belong(R,W)),
    all_p_planes([R|W],X),
    !.
all_p_planes(W,[]).

all_s_3D_figures(FigChoList,VisFcPc,W,[R|X]):-
    p_figure(R),
    not(belong(R,W)),
    R=(Fig,Vis,Asm),
    ((FigChoList\=[],
      Fig=..[FigHd|FigArgs],
      belong(FigHd,FigChoList));
     FigChoList=[]),
    Vis>=VisFcPc,
    all_s_3D_figures(FigChoList,VisFcPc,[R|W],X),
    !.
all_s_3D_figures(FigChoList,VisFcPc,W,[]).

/* ...replacements for single-view 3-D figures */

all_a_3D_figs(R,[W|X]):-
    ((a_box1(A,B,C,D,E,F,G,H),
      W=a_box1(A,B,C,D,E,F,G,H));
     (a_box2(A,B,C,D,E,F,G,H),
      W=a_box2(A,B,C,D,E,F,G,H));
     (a_box3(A,B,C,D,E,F),
      W=a_box3(A,B,C,D,E,F));
     (a_prism1(A,B,C,D,E,F),
      W=a_prism1(A,B,C,D,E,F));
     (a_prism1a(A,B,C,D,E,F),
      W=a_prism1a(A,B,C,D,E,F));
     (a_prism2(A,B,C,D,E,F),
      W=a_prism2(A,B,C,D,E,F));
     (a_prism2a(A,B,C,D,E,F),
      W=a_prism2a(A,B,C,D,E,F));
     (a_prism3(A,B,C,D,E,F),
      W=a_prism3(A,B,C,D,E,F));
     (a_pyram1(A,B,C,D,E),
      W=a_pyram1(A,B,C,D,E));
     (a_pyram1a(A,B,C,D,E),
      W=a_pyram1a(A,B,C,D,E));
     (a_pyram2(A,B,C,D,E),
      W=a_pyram2(A,B,C,D,E));
     (a_pyram2a(A,B,C,D,E),
      W=a_pyram2a(A,B,C,D,E));
     (a_pyram3(A,B,C,D,E),
      W=a_pyram3(A,B,C,D,E));
     (a_pyram4(A,B,C,D),
      W=a_pyram4(A,B,C,D));
     (a_tetra1(A,B,C,D),
      W=a_tetra1(A,B,C,D));
     (a_tetra2(A,B,C,D),
```

```
       W=a_tetra2(A,B,C,D));
      (a_tetra3(A,B,C),
       W=a_tetra3(A,B,C))),
      not(belong(W,R)),
    all_a_3D_figs([W|R],X).
all_a_3D_figs(R,[]).

/* ... segmentation planes */

all_s_planes(W,[R|X]):-
    s_plane(R),
    not(belong(R,W)),
    all_s_planes([R|W],X),
    !.
all_s_planes(W,[]).

/* ... multiple-view 3-D figures */

all_m_figures(W,[R|X]):-
    m_figure(R),
    not(belong(R,W)),
    all_m_figures([R|W],X),
    !.
all_m_figures(W,[]).

/* adjust face counters of face-conns in multiple view figures */

mark_tetra(A,B,C,D):-
    rtr_trn(A,B,C),
    rtr_trn(A,C,D),
    rtr_trn(A,B,D),
    rtr_trn(B,C,D).

mark_pyram(A,B,C,D,E):-
    rtr_qul(A,D,E,B),
    rtr_trn(A,B,C),
    rtr_trn(A,D,C),
    rtr_trn(C,D,E),
    rtr_trn(B,E,C).

mark_prism(A,B,C,D,E,F):-
    rtr_qul(E,A,F,D),
    rtr_qul(A,B,C,F),
    rtr_qul(E,B,C,D),
    rtr_trn(A,B,E),
    rtr_trn(F,C,D).

mark_box(A,B,C,D,E,F,G,H):-
    rtr_qul(A,B,C,G),
    rtr_qul(G,C,D,E),
    rtr_qul(A,G,E,F),
    rtr_qul(B,A,F,H),
    rtr_qul(C,B,H,D),
    rtr_qul(H,F,E,D).

/* replace a_2D_figures by a_3D_figures , for singl-view box */

mark_bx1(S,T,U,V,W,X,Y,Z):-
    not(a_box1(S,T,U,V,W,X,Y,Z)),
    rtr_aqu(S,T,X,W),
    rtr_aqu(S,W,Z,V),
    rtr_aqu(T,U,Y,X),
    rtr_aqu(Z,Y,U,V),
    rtr_aqu(W,X,Y,Z),
    rtr_pqu(S,T,U,V),
    assert(a_box1(S,T,U,V,W,X,Y,Z)).

mark_bx2(S,T,U,V,W,X,Y,Z):-
    not(a_box2(S,T,U,V,W,X,Y,Z)),
    rtr_aqu(S,T,Z,Y),
    rtr_aqu(T,Z,V,U),
    rtr_aqu(Y,Z,V,W),
    rtr_aqu(S,Y,W,X),
    assert(a_box2(S,T,U,V,W,X,Y,Z)).

mark_bx3(T,U,V,W,X,Y,Z):-
    not(a_box3(T,U,V,W,X,Y,Z)),
    rtr_aqu(T,U,Z,Y),
    rtr_aqu(Y,Z,W,X),
    rtr_aqu(Z,U,V,W),
    assert(a_box3(T,U,V,W,X,Y,Z)).

/* ... prism */

mark_pr1(U,V,W,X,Y,Z):-
    not(a_prism1(U,V,W,X,Y,Z)),
    rtr_aqu(U,W,Y,X),
    rtr_aqu(Z,Y,W,V),
    rtr_aqu(V,U,X,Z),
    rtr_atr(X,Y,Z),
    rtr_ptr(V,U,W),
    assert(a_prism1(U,V,W,X,Y,Z)).

mark_pr1a(U,V,W,X,Y,Z):-
    not(a_prism1a(U,V,W,X,Y,Z)),
    rtr_aqu(Y,Z,U,W),
    rtr_aqu(X,Y,Z,V),
    rtr_atr(Z,V,U),
    rtr_atr(X,Y,W),
    rtr_pqu(V,U,W,X).

mark_pr2(U,V,W,X,Y,Z):-
    not(a_prism2(U,V,W,X,Y,Z)),
    rtr_aqu(U,V,W,X),
    rtr_aqu(U,X,Y,Z),
    rtr_atr(X,W,Y),
    assert(a_prism2(U,V,W,X,Y,Z)).

mark_pr2a(U,V,W,X,Y,Z):-
    not(a_prism2a(U,V,W,X,Y,Z)),
    rtr_aqu(U,V,X,Y),
    rtr_atr(V,W,X),
    rtr_atr(U,Y,Z),
    assert(a_prism2a(U,V,W,X,Y,Z)).

mark_pr3(U,V,W,X,Y,Z):-
    not(a_prism3(U,V,W,X,Y,Z)),
    rtr_aqu(U,V,W,X),
    rtr_aqu(X,Y,Z,U),
    assert(a_prism3(U,V,W,X,Y,Z)).

/* ... pyramid */

mark_py1(V,W,X,Y,Z):-
    not(a_pyram1(V,W,X,Y,Z)),
    rtr_aqu(W,X,Y,Z),
    rtr_atr(V,W,Z),
    rtr_atr(V,Y,Z),
    rtr_atr(V,Y,X),
    rtr_ptr(V,X,W),
    assert(a_pyram1(V,W,X,Y,Z)).
```

```
mark_py1a(V,W,X,Y,Z):-
   not(a_pyramla(V,W,X,Y,Z)),
   rtr_atr(V,W,Z),
   rtr_atr(V,Z,Y),
   rtr_atr(Y,Z,X),
   rtr_atr(Z,X,W),
   rtr_pqu(V,W,X,Y),
   assert(a_pyramla(V,W,X,Y,Z)).

mark_py2(V,W,X,Y,Z):-
   not(a_pyram2(V,W,X,Y,Z)),
   rtr_aqu(Z,W,X,Y),
   rtr_atr(V,Z,Y),
   rtr_atr(V,W,Z),
   assert(a_pyram2(V,W,X,Y,Z)).

mark_py2a(V,W,X,Y,Z):-
   not(a_pyram2a(V,W,X,Y,Z)),
   rtr_atr(V,W,X),
   rtr_atr(V,X,Y),
   rtr_atr(V,Y,Z),
   assert(a_pyram2a(V,W,X,Y,Z)).

mark_py3(V,W,X,Y,Z):-
   not(a_pyram3(V,W,X,Y,Z)),
   rtr_aqu(V,Y,Z,W),
   rtr_atr(V,W,X),
   assert(a_pyram3(V,W,X,Y,Z)).

mark_py4(W,X,Y,Z):-
   not(a_pyram4(W,X,Y,Z)),
   rtr_aqu(W,X,Y,Z),
   assert(a_pyram4(W,X,Y,Z)).

/* ... tetrahedron */

mark_te1(W,X,Y,Z):-
   not(a_tetra1(W,X,Y,Z)),
   rtr_atr(W,X,Y),
   rtr_atr(W,Y,Z),
   rtr_atr(W,Z,X),
   rtr_ptr(X,Y,Z),
   assert(a_tetra1(W,X,Y,Z)).

mark_te2(W,X,Y,Z):-
   not(a_tetra2(W,X,Y,Z)),
   rtr_atr(W,X,Y),
   rtr_atr(W,Y,Z),
   assert(a_tetra2(W,X,Y,Z)).

mark_te3(X,Y,Z):-
   not(a_tetra3(X,Y,Z)),
   rtr_atr(X,Y,Z),
   assert(a_tetra3(X,Y,Z)).
```

```
                /* definitions of 3-D figures */


/* multiple-view definitions */

box(A,B,C,D,E,F,G,H):-
   c_quadril(A,B,C,G),
   c_quadril(G,C,D,E),
   c_quadril(A,G,E,F),
   c_quadril(B,A,F,H),
   c_quadril(C,B,H,D),
   c_quadril(H,F,E,D).

prism(A,B,C,D,E,F):-
   c_quadril(E,A,F,D),
   c_quadril(A,B,C,F),
   c_quadril(E,B,C,D),
   trian(A,B,E),
   trian(F,C,D).

pyram(A,B,C,D,E):-
   c_quadril(A,D,E,B),
   trian(A,B,C),
   trian(A,D,C),
   trian(C,D,E),
   trian(B,E,C).

tetra(A,B,C,D):-
   trian(A,B,C),
   trian(A,C,D),
   trian(A,B,D),
   trian(B,C,D).

/* single-view definitions */

p_box1(A,B,C,D,E,F,G,H):-
   set_aqu(A,B,C,D),
   set_aqu(A,B,F,E),
   set_aqu(B,C,G,F),
   set_aqu(D,C,G,H),
   set_aqu(A,D,H,E),
   set_pqu(E,F,G,H),
   E\=G,
   E\=C,
   F\=D,
   G\=A,
   H\=F,
   H\=B.

p_box2(A,B,C,D,E,F,G,H):-
   set_aqu(A,B,C,D),
   set_aqu(B,C,G,F),
   set_aqu(D,C,G,H),
   set_aqu(A,D,H,E),
   not(non_convex_contour_angle(A)),
   not(non_convex_contour_angle(B)),
   not(non_convex_contour_angle(G)),
   not(non_convex_contour_angle(H)),
   B\=E,
   F\=E,
   F\=A,
   G\=A,
   G\=E,
   H\=B,
   H\=F.
```

```
p_box3(A,B,C,D,E,F,G):-
    set_aqu(A,B,C,D),
    set_aqu(B,C,G,F),
    set_aqu(D,C,G,E),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(D)),
    not(non_convex_contour_angle(G)),
    A\=E,
    A\=F,
    A\=G,
    B\=E,
    D\=F.

p_prism1(A,B,C,D,E,F):-
    set_aqu(A,B,E,D),
    set_aqu(B,E,F,C),
    set_aqu(C,F,D,A),
    set_atr(A,B,C),
    set_ptr(D,E,F).

p_prism1a(A,B,C,D,E,F):-
    set_aqu(C,F,D,A),
    set_aqu(A,B,E,D),
    set_atr(A,B,C),
    set_atr(D,E,F),
    set_pqu(B,E,F,C),
    B\=F,
    C\=E.

p_prism2(A,B,C,D,E,F):-
    set_aqu(C,F,D,A),
    set_aqu(B,E,F,C),
    set_atr(A,B,C),
    not(non_convex_contour_angle(A)),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(F)),
    A\=E,
    B\=D.

p_prism2a(A,B,C,D,E,F):-
    set_aqu(A,B,E,D),
    set_atr(A,B,C),
    set_atr(D,E,F),
    not(non_convex_contour_angle(A)),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(D)),
    not(non_convex_contour_angle(E)),
    A\=F,
    B\=F,
    C\=D,
    C\=E,
    C\=F.

p_prism3(A,B,C,D,E,F):-
    set_aqu(A,B,C,D),
    set_aqu(A,D,E,F),
    not(non_convex_contour_angle(A)),
    not(non_convex_contour_angle(D)),
    B\=F,
    B\=C,
    C\=E,
    E\=F.

p_pyram1(A,B,C,D,E):-
    set_aqu(B,C,D,E),
    set_atr(A,E,B),
    set_atr(A,B,C),
    set_atr(A,D,C),
    set_ptr(A,D,E).

p_pyram1a(A,B,C,D,E):-
    set_atr(A,E,B),
    set_atr(A,B,C),
    set_atr(A,D,C),
    set_atr(A,D,E),
    set_pqu(B,C,D,E).

p_pyram2(A,B,C,D,E):-
    set_aqu(B,C,D,E),
    set_atr(A,B,C),
    set_atr(A,D,C),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(D)),
    A\=E.

p_pyram2a(A,B,C,D,E):-
    set_atr(A,B,C),
    set_atr(A,C,D),
    set_atr(A,B,E),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(C)),
    B\=D,
    C\=E,
    E\=D.

p_pyram3(A,B,C,D,E):-
    set_aqu(B,C,D,E),
    set_atr(A,B,C),
    not(non_convex_contour_angle(B)),
    not(non_convex_contour_angle(C)),
    A\=D,
    A\=E.

p_pyram4(A,B,C,D):-
    set_aqu(A,B,C,D).

p_tetra1(A,B,C,D):-
    set_atr(A,B,C),
    set_atr(A,C,D),
    set_atr(C,B,D),
    set_ptr(A,B,D).

p_tetra2(A,B,C,D):-
    set_atr(A,B,C),
    set_atr(A,C,D),
    not(non_convex_contour_angle(A)),
    not(non_convex_contour_angle(C)),
    B\=D.

p_tetra3(A,B,C):-
    set_atr(A,B,C).
```

```
    /* definitions of 1-D and 2-d figures */


/* lines */

line(A,B):-
   (conn(A,B,N);
    conn(B,A,N)),
   atom(A),
   atom(B),
   integer(N),
   A\=B,
   N>0,
   N=<2.

a_line(X,Y):-
   conn(X,Y,1);
   conn(Y,X,1).

b_line(X,Y):-
   conn(X,Y,2);
   conn(Y,X,2).

c_line(X,Y):-
   conn(X,Y,0);
   conn(Y,X,0).

/* visible triangle */

trian(A,B,C):-
   line(A,B),
   line(B,C),
   line(C,A),
   not(poin_trn(A,B,C)).

set_atr(A,B,C):-
   atrian(A,B,C);
   atrian(A,C,B);
   atrian(B,A,C);
   atrian(B,C,A);
   atrian(C,A,B);
   atrian(C,B,A).

/* possible triangle */

p_trian(A,B,C):-
   line(A,B),
   line(B,C),
   line(C,A),
   poin_trn(A,B,C).

set_ptr(A,B,C):-
   ptrian(A,B,C);
   ptrian(A,C,B);
   ptrian(B,A,C);
   ptrian(B,C,A);
   ptrian(C,A,B);
   ptrian(C,B,A).

/* point inside triangle */

poin_trn(A,B,C):-
   point_in_trn(A,B,C);
   point_in_trn(A,C,B);
   point_in_trn(B,A,C);
   point_in_trn(B,C,A);
   point_in_trn(C,A,B);
   point_in_trn(C,B,A).

/* visible convex quadrilateral */

c_quadril(A,B,C,D):-
   line(A,B),
   line(B,C),
   line(C,D),
   line(D,A),
   A\=C,
   B\=D,
   not(line(A,C)),
   not(line(B,D)),
   not(poin_qul(A,B,C,D)),
   not(nocnvx(A,B,C,D)).

set_aqu(W,X,Y,Z):-
   aquadril(W,X,Y,Z);
   aquadril(W,Z,Y,X);
   aquadril(X,W,Z,Y);
   aquadril(X,Y,Z,W);
   aquadril(Y,X,W,Z);
   aquadril(Y,Z,W,X);
   aquadril(Z,W,X,Y);
   aquadril(Z,Y,X,W).

/* possible convex quadrilateral */

p_c_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   poin_qul(W,X,Y,Z),
   not(nocnvx(W,X,Y,Z)).

set_pqu(W,X,Y,Z):-
   pquadril(W,X,Y,Z);
   pquadril(W,Z,Y,X);
   pquadril(X,W,Z,Y);
   pquadril(X,Y,Z,W);
   pquadril(Y,X,W,Z);
   pquadril(Y,Z,W,X);
   pquadril(Z,W,X,Y);
   pquadril(Z,Y,X,W).

/* visible non-convex quadrilateral */

nc_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   not(poin_qul(W,X,Y,Z)),
   nocnvx(W,X,Y,Z).

set_nqu(W,X,Y,Z):-
   nquadril(W,X,Y,Z);
   nquadril(W,Z,Y,X);
   nquadril(X,W,Z,Y);
   nquadril(X,Y,Z,W);
   nquadril(Y,X,W,Z);
   nquadril(Y,Z,W,X);
   nquadril(Z,W,X,Y);
   nquadril(Z,Y,X,W).

/* possible non-convex quadrilateral

p_nc_quadril(W,X,Y,Z):-
   line(W,X),
   line(X,Y),
   line(Y,Z),
   line(Z,W),
   W\=Y,
   X\=Z,
   not(line(W,Y)),
   not(line(X,Z)),
   poin_qul(W,X,Y,Z),
   nocnvx(W,X,Y,Z).

set_p_nqu(W,X,Y,Z):-
   p_nquadril(W,X,Y,Z);
   p_nquadril(W,Z,Y,X);
   p_nquadril(X,W,Z,Y);
   p_nquadril(X,Y,Z,W);
   p_nquadril(Y,X,W,Z);
   p_nquadril(Y,Z,W,X);
   p_nquadril(Z,W,X,Y);
   p_nquadril(Z,Y,X,W).

/* point inside convex quadrilateral

poin_qul(W,X,Y,Z):-
   point_in_qul(W,X,Y,Z);
   point_in_qul(W,Z,Y,X);
   point_in_qul(X,W,Z,Y);
   point_in_qul(X,Y,Z,W);
   point_in_qul(Y,X,W,Z);
   point_in_qul(Y,Z,W,X);
   point_in_qul(Z,W,X,Y);
   point_in_qul(Z,Y,X,W).

/* non-convex angle among (W,X,Y,Z)

nocnvx(W,X,Y,Z):-
   non_convex(W,X,Y,Z);
   non_convex(W,Z,Y,X);
   non_convex(X,W,Z,Y);
   non_convex(X,Y,Z,W);
   non_convex(Y,X,W,Z);
   non_convex(Y,Z,W,X);
   non_convex(Z,W,X,Y);
   non_convex(Z,Y,X,W).
```

APPENDIX 3

3.1 INTRODUCTION TO PREDICATE CALCULUS

Predicate calculus [Ballard & Brown '82, Charniak et al '85] is a useful and attractive knowledge representation and inference system. It consists of a language for expressing propositions, and rules for inferring new facts from those that already exist. The language uses a *clausal syntax* and a *nonclausal* one. In the clausal syntax a *sentence* is a set of *clauses*. A clause is an ordered pair of sets of *atomic formulae* or *atoms*. The two sets are separated with an implication symbol that points from the *conditions* or *hypotheses* of the clause to its *conclusion:*

$$A_1, A_2, \ldots, A_n \rightarrow C_1, C_2, \ldots, C_m$$

where the A's and C's are atoms. An atom is an expression of the form:

$$p(a_1, a_2, \ldots, a_k)$$

where p is a predicate symbol with k arguments. An argument can be a *variable*, a *constant symbol*, or a *term*. A term is an expression of the form:

$$t(a_1, a_2, \ldots, a_\ell)$$

where t is a *function symbol* with ℓ arguments each of which may also be a term. Constant symbols may be treated as terms too.

In the nonclausal syntax a set of *logical connectives* is used to combine atoms to form *well-formed formulae* (wffs). The logical connectives are the following unary and binary operators:

$\bar{A}$('not A'), $A \Rightarrow B$ ('A implies B', or 'if A then B'), $A \vee B$ ('A or B')

$A \wedge B$ ('A and B') and $A \Leftrightarrow B$ ('A is equivalent to B', or 'A if and only if B')

where A and B are atoms. More general things can be expressed using variables and a pair of quantifiers. The *universal quantifer* $\forall$ ('for every element') is interpreted as a conjunction of all domain elements, and the *existential quantifier* $\exists$ ('there is an element') as a disjunction of all domain elements. A quantifier quantifies its 'dummy' variable and the variable is said to be *bound* by the quantifier within its *scope*. Clauses and wffs are assigned a *truth value* according to a *truth table* (Table 3.1), by determining the truth value of each of their atoms.

A	B	$\overline{A}$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

t=True
f=False

TABLE 3.1

The conversion [Nilsson '71] of nonclausal to clausal form is done by applying straight-forward rewriting rules, based on logic identities. The basic idea is to remove all existential quantifiers by replacing the occurrence of their variables by newly created functions (called *skolem*) of all the universally quantified variables whose scope includes the existential quantifiers being eliminated. For example,

$$(\forall x)(\exists y)(bigger(x,y))$$

will become

$$(\forall x)(bigger(something\ bigger(x),x)$$

where *something bigger(x)* is the skolem function.

Predicate calculus as a knowledge representation system has the following advantages:

a) It is a language with a machine-independent semantics.

b) Clauses with only one conclusion atom (*Horn clauses* [Horn '51]) may be considered as 'procedures', and thus lead to the development of predicate logic-based programming languages (e.g. PROLOG).

c) Resolutions performed on the left or right clauses result in derivation trees corresponding to top-down and bottom-up versions of problem solving.

d) It contains uniform proof procedures for logic that can prove any true theorem in finite time.

Some disadvantages of predicate calculus are:

a) Its implementation of common concepts is not immediately obvious.

b) Its 'first order' version does not allow clauses with variables ranging over an infinite number of predicates, functions, assertions and sentences.

c) It accumulates a large number of axioms that remain true after their actions have been performed causing problems in maintaining the state of the world state *(frame problem)*.

d) It does not address certain sorts of human reasoning, like the ability to describe its own formulae, the notation of defaults, or a mechanism for plausible reasoning.

Predicate calculus is successfully used to represent semantic networks, and in theorem proving [Kowalski '79].

3.2 INTRODUCTION TO PROLOG

PROLOG is a class of languages designed for logic programming based on first-order predicate calculus expressed in Horn clause format. It originated in 1972 by Colmerauer and Roussel at Luminy (Marseille), to implement logic as a tool for Artificial Intelligence. At first, it was a theorem prover and passed through several stages of development before taking its latest form in 1981, brought forward by several projects in Artificial Intelligence.

3.2.1 Syntax [Clocksin & Mellish '81a]

PROLOG programs are built from *terms* that can be either *constants*, *variables*, or *structures*. A constant is either an *atom*, represented by a symbol made up of letters and digits normally beginning with a lower-case letter, or an *integer* used to represent numbers. A variable looks like an atom, except it begins with a *capital letter* or an *underscore sign '_'* and stands for objects of unknown names. Variables can be either *instantiated* or *uninstantiated*. A variable is instantiated when there is certain object that the variable stands for. Uninstantiated is a variable when what it stands for is not yet known. A structure is a single object consisting of a number of components grouped together in order to be handled conveniently. An example of an atom, a variable and a structure is given below:

banana. , *Object.* , *eat(monkeys,bananas).*

eat is the *functor* of the structure, and *monkey,bananas* its *arguments* enclosed in round brackets and separated by commas. Two important data structures are *trees* and *lists*. A tree is a complicated structure with the functor being its root and the arguments being its branches, (Fig.3.1).

sentence(noun(monkeys),verb_phrase(verb(eat),noun(bananas)))

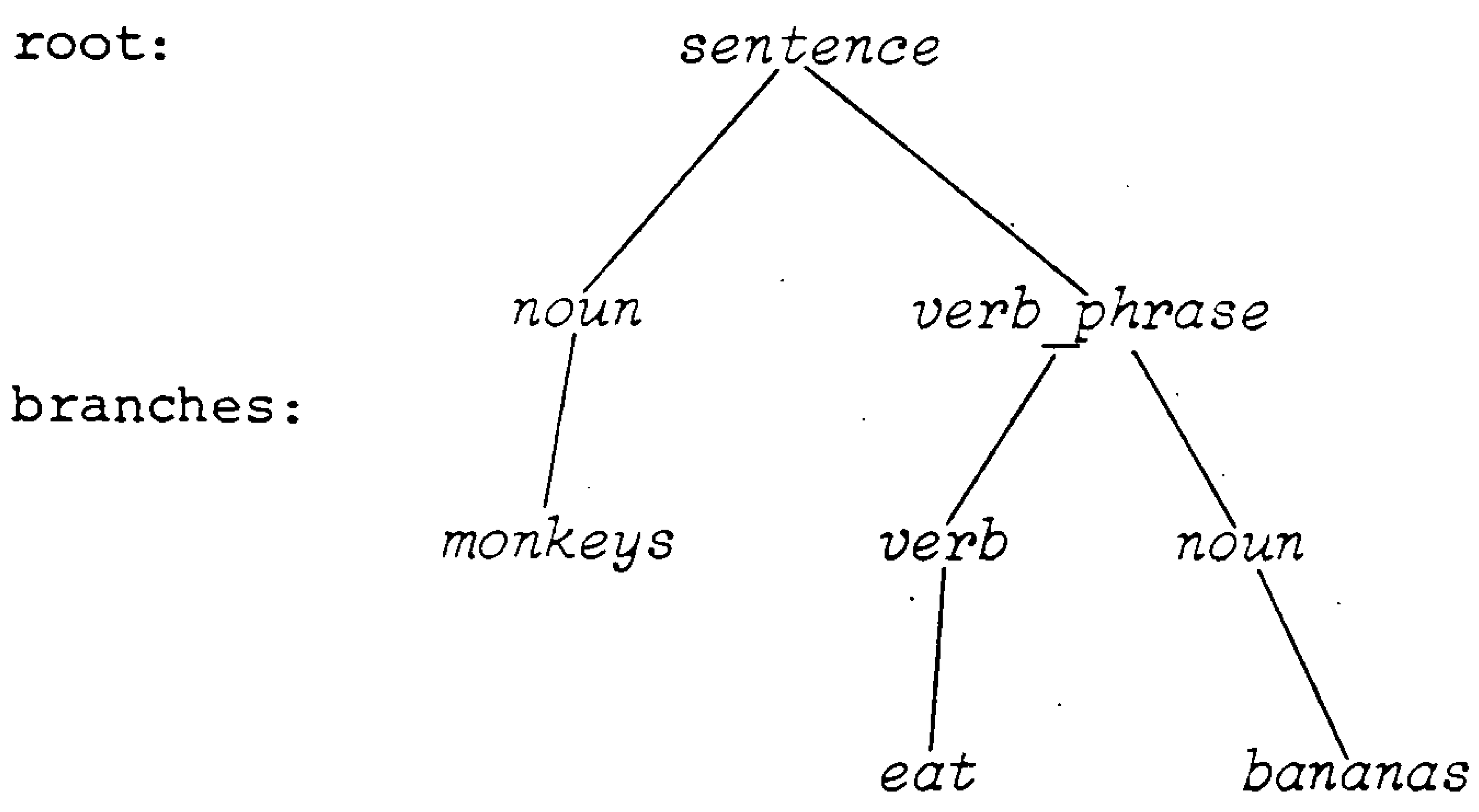


FIGURE 3.1

A list is an ordered sequence of *elements* that can be any kind of terms or other lists, enclosed in square brackets and separated by commas, e.g.

[vertex,triangle(x,y,z),[a,b,c]]

The empty list is denoted by [ ]. A very useful list representation is:

[L|Ls]

where *L* is the first element of the list, and *Ls* is a list of the rest of elements. This applied to the previous example would give:

L=vertex,Ls=[triangle(x,y,z),[a,b,c]]

3.2.2 Programming

A PROLOG program consists of a sequence of statements that can be either rules or comments. A rule is a general statement about objects and their relationships and consists of a left-hand part *(rule-head)* and a right-hand part *(rule-body)* separated by a ':-' (=if) and ended by a (.) e.g.

triangle(A,B,C):-line(A,B),line(B,C),line(C,A).

The commas separating the conditions of the rule-body are special symbols denoting *conjunction*. *Disjunction* is denoted by a ';'. A program execution consists of responding to a *question* about the relationships expressed by a rule. A question is a sequence of terms, or *goals* representing a conjunction of relationships to be satisfied. PROLOG responds to a question by attempting to *satisfy* its goals one by one in the order in which they appear. Satisfying a goal begins by searching the database from the top, looking for a matching fact or rule-head.

a) If a goal matches a fact, the system marks this place in the database and instantiates any previously uninstantiated variables that have matched. If the match is with a rule-head, PROLOG attempts first to satisfy the rule. If the goal *succeeds*, an attempt to satisfy the next goal on the right is made.

b) If no matching predicate is found, the goal is said to have *failed*. In this case PROLOG goes back to *re-satisfy* a previous goal. This is known as *backtracking* and involves the following steps:

   b1) The variables that became instantiated when this goal was previously satisfied are made again uninstantiated.

b2) The search for alternative matches is resumed at the point previously marked.

Backtracking will lead to an alternative goal that either succeeds or fails, i.e. either case (a) or case (b) will occur. PROLOG carries on until either the original goal succeeds, or there are no more alternative goals, in which case the initial goal fails.

PROLOG contains a number of *built-in predicates* that may provide convenient facilities to the programmer. A complete set of the built-in predicates with explanations of their use is given in [Clocksin & Mellish '81b]. A subset of this is given below:

assert(X): Adds clause *X* to the database.

clause(X,Y): There is a clause in the database with head *X* and body *Y*.

consult(X): Reads clauses and goals from file *X*.

listing(X): Lists all clauses with atom *X* as predicate.

name(X,Y): *Y* is the list of the characters of *X*'s name.

not(X): A goal succeeds if and only if *X* fails.

read(X): Reads term *X* terminated by '.' and a new line from the current input.

repeat: A goal that succeeds in infinitely many ways.

retract(X): Removes clause *X* from the database.

see(X): Switches current input to be file *X*. *X* is an atom.

seen: Closes the current input file and switches back to standard input.

tell(X): Switches current output to be file *X*. *X* is an atom.

told: Closes the current output file and switches back to standard output.

write(X): Writes term *X* on the current output, taking account of current operator declarations.

X=..Y: *Y* is the list made up of the functor of *X* followed by the arguments of *X*.

!: The 'cut' operator. It commits the system to all choices made since the invocation of the most recent ancestor goal that is not a conjunction, disjunction, or a use of 'call(_)'. It causes later alternatives to be ignored.

3.2.3 PROLOG Versions

The above presented PROLOG version is general and it does not correspond exactly to any existing operating system. However, several versions have been developed according to the capabilities and special features of the operating systems on which they are implemented [Campbell '84]. Therefore, some differences in the *syntax*, *arbitrary bounds of some numbers*, *environmental features*, *option of compilation*, *built-in predicates*, and *debugging facilities* may occur. Some of the most common PROLOG versions are: PROLOGII [Colmerauer et al '82, Colmerauer '85], based on the original version developed by [Roussel '75]. The EDINBURG PROLOG [Pereira et al '79], implemented on the DEC-10 operating system. The UNIX PROLOG [Clocksin & Mellish '79] developed for the UNIX operating system. The RT-11 PROLOG [Clocksin et al '80] developed for the DEC LSI-11 machine (running the RT-11 operating system), which has been translated from the PDP-11 UNIX version. POPLOG [Hardy '82], that creates a multi-language programming environment by combining PROLOG, a logic language, with POP-2 [Burstall et al '71], a procedural language. The MICRO-PROLOG [McCabe '80] that works on inexpensive microprocessor systems and is used for modelling expert systems.

In recent years, methods representing uncertainty and fuzzy logic are incorporated into the PROLOG language to produce some fuzzy versions. PROLOG-ELF [Ishizuka & Kanai '85] is a PROLOG implementation as a basic language for building knowledge systems. It has all the preferable basic features of PROLOG in addition to assertions with a truth-value 1.0-0.5, and it can manipulate fuzzy set very easily to a certain extent. FUZZY PROLOG [Hinde '86] introduces a natural method of controlling the search, making it tree admissible. It also discusses the use of variable functors in the specification of bidirectional logic, including several examples and suggested applications. FRIL (Fuzzy Programming Interface Language) [Baldwin et al '87], is a new language that combines a dialect of PROLOG with a Support Logic Programming System for representation and reasoning with uncertainty in a Logic Programming Environment. It is designed for applications in Artificial Intelligence and Expert Systems.

Although PROLOG has been adopted as basic language for the *Fifth Generation* project, it has also taken some negative criticism [Forsyth '84] for having little error-protection, being memory greedy, being unsuitable for efficient parallel execution,and in general questioning its being a 'logic programming language'.

REFERENCES

1. BALDWIN, J.F., MARTIN, T.P. and PILSWORTH, B.W. 1987: *FRIL Manual*, Equipu-AIR Ltd.
2. BALLARD, D.H. and BROWN, C.N. 1982: *Computer Vision*, Prentice-Hall, pp.384-395.

3. BURSTALL, R.M., COLLINS, J.S. and POPPLESTONE, R. 1971: *Programming in POP-2*, Edinburgh Univ. Press.

4. CAMPBELL, J.A. 1984: *Implementation of PROLOG*, Ellis Horwood Series AI.

5. CHARNIAK, E. and McDERMOTT, D. 1985: *Introduction to AI*, Addison-Wesley, pp.14-21.

6. CLOCKSIN, W.F. and MELLISH, C.S. 1979: *The Unix Prolog System*, Software Report 5, Dept. of AI, Univ. of Edinburgh, Edinburgh, Scotland.

7. CLOCKSIN, W.F., MELLISH, C.S. and FISHER, R. 1980: *The RT-11 Prolog System*, Software Report 5a (revised), Dept. of AI, Univ. of Edinburgh, Edinburgh, Scotland.

8. CLOCKSIN, W.F. and MELLISH, C.S. 1981: *Programming in Prolog*, Springer-Verlag, a: pp.22-57, b: pp. 94-129.

9. COLMERAUER, A. 1985: *Prolog in 10 Figures*, Proc. 9th IJCAI, pp. 487-499.

10. COLMERAUER, A. KANOUI, H. and VAN CANEGHEM, M. 1982: *Prolog, Theoretical Principles and Current Trends*, Technology and Science of Informatics, Vol.2 No. 4, Faculté des Sciences de Marseille-Luminy.

11. FORSYTH, R. 1984: *Expert Systems: Principles and Case Studies*, Chapman-Hall Computing, pp. 15-16.

12. HARDY, S. 1982: *The POPLOG Programming Environment*, Cognitive Studies Memo 82-05 Univ. of Sussex.

13. HINDE, C.J. 1986: *Fuzzy Prolog*, Int.J. Man-Machine Studies 24, pp. 569-595.

14. HORN, A. 1951: *On Sentences That are True of Direct Unions of Algebras*, Journal of Symbolic Logic, 16, pp. 14-21.

15. ISHIZUKA, M. and KANAI, N. 1985: *Prolog-ELF Incorporating Fuzzy Logic*, Proc. 9th IJCAI, pp. 701-710.

16. KOWALSKI, R.A. 1979: *Logic for Problem Solving*, N. York, Elsevier North Holland (AI Series).

17. McGABE, F.G. 1980: *Micro-PROLOG*, Programmers' Reference Manual, Logic Programming Associates.

18. NILSSON, N.J. 1971: *Problem-Solving Methods in AI*, N.York, McGraw-Hill.

19. PEREIRA, L.M., PEREIRA, F. and WARREN, D. 1979: *User's Guide to DEC System-10 Prolog*, DAI Occasional Paper 15, Dept. of AI, Univ. of Edinburgh, Edinburgh, Scotland.

20. ROUSSEL, P. 1975: *Prolog Manuel de Reférênce et d'Utilisation*, GIA, Faculté des Sciences de Marseille-Luminy.