TWO-DIMENSIONAL COMPUTER VISION
FOR INSPECTION AND ROBOTICS

By

DAVID W. CAPSON, B.Sc.Eng., M.Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Doctor of Philosophy

McMaster University

TWO-DIMENSIONAL COMPUTER VISION

DOCTOR OF PHILOSOPHY (1984)
(Electrical and Computer Engineering)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: Two-dimensional Computer Vision for Inspection and Robotics

AUTHOR: David W. Capson, B.Sc.Eng. (University of New Brunswick)
M.Eng. (McMaster University)

SUPERVISOR: Dr. R. Kitai

NUMBER OF PAGES: xv, 221

ABSTRACT

The thesis concerns binary vision for non-contact inspection and robotics applications in flexible manufacturing. Two-dimensional silhouettes of three-dimensional objects are processed to measure a number of features including area, perimeter, circularity, maximum radius, moments, and number of holes. This information is then used to identify and locate randomly positioned objects to provide visual feedback for part acquisition by an industrial robot or inspection tasks that might include checking dimensional tolerances, verification of hole placement, and the like.

Silhouettes are encoded in the system as a linked list of "vertex points" representing changes of direction on the contour. An algorithm and data structures for extraction of vertex points from a raster-scanned binary image have been developed. The method operates sequentially and no restrictions are imposed on the number or topology of silhouettes in each frame. Several existing contour tracing algorithms are reviewed; the new algorithm is shown to offer considerable improvement in execution time at the expense of a small increase in memory. It is also demonstrated that vertex point approximations use significantly fewer points than run-length segment representations.

System implementation is based on a 232 x 240 CID camera, an 8086/8087 single-board microcomputer, and two custom-built boards. The architecture features a multiple bus configuration designed for high speed, parallel operation of dedicated modules: grey-level histogram generation and binary image acquisition are at video-rates. The complete software is in EPROM, and includes feature extraction for recognition and location of silhouettes encoded as vertex point lists, a data base for prototype objects and a comprehensive command set.

The software also includes communication with a PUMA 600 robot via a serial interface as well as extensions to its VAL language; in this way the robot accesses visual information under program control. An example is given of parts sorting using the extended instruction set.

The results of a large number of statistical measurements are used to establish overall system performance.

ACKNOWLEDGEMENTS

The author is indebted to Dr. Reuven Kitai for guidance and encouragement throughout his supervision of this work. Gratitude is also extended to Dr. J. Tlusty and Dr. D.P. Taylor as members of the supervisory committee and to the Natural Sciences and Engineering Research Council and McMaster University for financial assistance. The typing skills of Mrs. Debbie Harris in the preparation of the manuscript are greatly appreciated.

Finally, a very special thank you to my wife for whose patience and love this work is dedicated.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure		Page

Figure		Page

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

m	–	Number of bits needed to represent horizontal resolution
n	–	Number of bits needed to represent vertical resolution
VSYNC	–	Vertical Synchronizing Pulse
HSYNC	–	Horizontal Synchronizing Pulse
ERC	–	Element Rate Clock
X_L	–	Left binary video transition
X_R	–	Right binary video transition
V_i	–	i^{th} vertex point
x_i	–	x-coordinate of i^{th} vertex point
y_i	–	y-coordinate of i^{th} vertex point
N	–	Number of vertex points on a contour
X_c	–	x-coordinate of contour centroid
Y_c	–	y-coordinate of contour centroid
ℓ_i	–	Distance between the i^{th} vertex point and the previous vertex point
P2A	–	Circularity of contour
$\bar{X}_i$	–	x-coordinate of centroid of link ℓ_i
$\bar{Y}_i$	–	y-coordinate of centroid of link ℓ_i
CFD	–	Current Frame Description
SRA	–	Stored Record Address

k	–	Number of entries in data base index
M	–	Number of features in feature set
θ	–	Contour angular orientation
M_{xx}	–	Second moment of inertia with respect to x-axis
M_{yy}	–	Second moment of inertia with respect to y-axis
M_{xy}	–	product of inertia
n	–	Number of binary transitions in one raster scan line
$P_{i,j}$	–	Value of binary pixel with Cartesian coordinates (i,j)
PUMA	–	Programmable Universal Machine for Assembly
μC	–	Microcomputer
SBC	–	Single Board Computer
p_i	–	i^{th} grey-scale pixel
b_i	–	i^{th} binary pixel
T_ℓ	–	Lower threshold
T_h	–	Upper threshold
@	–	Carriage return
V	–	Vertex point encoding ratio
S.D.	–	Standard Deviation
σ_x	–	Standard deviation of centroid x-coordinate
σ_y	–	Standard deviation of centroid y-coordinate
P_x	–	Spatial resolution on x-axis
P_y	–	Spatial resolution on y-axis
h	–	Length of maximum radius from centroid to a sharp corner
σ_θ	–	Standard deviation of orientation

CHAPTER 1

INTRODUCTION

Manufacturing industriés, contending with demands for increased productivity and the escalating costs of human resources, are turning to automation to remain competitive. The last decade of evolution in microelectronics technology and the more recent economic difficulties have led to a trend towards flexible automation systems in which robotics promises to play an important role. Widespread activity in this area has shown that the capabilities of industrial robots can be greatly enhanced with the development of robot sensory systems, and with vision in particular. Machine vision also finds application in a wide range of inspection tasks; recent examples include hot steel slabs [1.1], surface defects in wood [1.2] and instrument gauges [1.3].

Human visual abilities are impressive in terms of resolution and speed as well as in the perception of depth, colour and complex scenes. However, machine vision tailored to very specific applications can outperform human vision in highly quantitative and repetitive tasks. Costs and accuracy have improved markedly with the availability of inexpensive, powerful microcomputer equipment and solid state imager devices.

Systems which process full grey-scaled images have the capability of handling complex scenes under unpredictable lighting conditions, but have difficulties in meeting the speed requirements of a production environment. Of note is the research carried out at the University of Rhode Island [1.4] [1.5] in picking overlapping workpieces from a bin, and that of General Motors Research Laboratories [1.6] [1.7] in detecting overlapping parts on a cluttered background.

If the brightness of each pixel is restricted to one of two values corresponding to black or white, images are reduced to silhouettes and a considerable saving in storage and processing times can be achieved. The range of images which can then be analyzed is obviously restricted, but the techniques for processing silhouette (binary) images are generally well-established, faster, and use far less memory than comparable resolution grey-scaled pictures [1.8], [1.9]. Of course, some image forms are inherently binary including those of handwriting, printed text, printed circuit traces and engineering drawings.

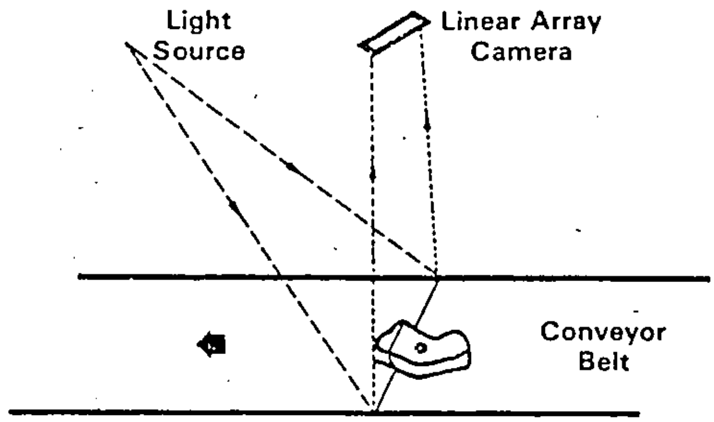
Early work including that at SRI International [1.10] has spawned a large number of binary image processing systems both commercially and at research institutions [1.11]. Measurements are made of the silhouettes of randomly positioned workpieces to characterize their shape and to determine their position and orientation within the image. These typically include area, perimeter, number of holes, moments of inertia and the like. This information can then be used for analyses such as shape identification, checking dimensional tolerances and verifying hole placement.

Frequently, location and orientation are used to guide a manipulator for part acquisition.

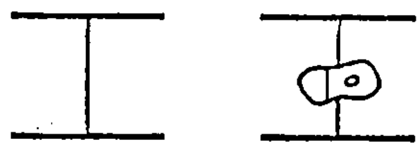
Binary image processing methods are also used in the measurement and location of shape in grey-scaled images which have undergone segmentation. The process of segmenting a grey-scaled (or colour) image involves isolating the contours of regions of interest from background information, resulting in binary representations [1.12]. An example is described by Birk, Kelley and Martins [1.13] in which the binary images of the surfaces of workpieces are detected in a grey-scaled image of a bin of overlapping workpieces.

An important application of binary systems is in the processing of images formed by the projection of points, lines or grids onto three-dimensional objects. Shape descriptions can be inferred from the binary images of the light patterns, distorted by the curvature of the object. A well known example of this technique is the CONSIGHT-I system developed at General Motors Research Laboratories [1.14]. A line of light is projected from an angle across a conveyor belt carrying workpieces and a linear array camera is positioned perpendicular to the belt so as to view the line [figure 1.1 (a)]. When objects intersect the line, they block a portion of the light before it reflects on the belt surface and the camera can easily detect the presence of the part as shown in figure 1.1 (b).

A more recent example is the Real-Time Photogrammetry System (RPS) developed at the National Research Council of Canada and described in [1.15]. In this system, the relative positions of the silhouettes of targets projected on a surface are used to determine



(a) Position of light source and camera.



(b) Camera view of conveyor belt.

Figure 1.1 General Motors CONSIGHT-I Lighting Arrangement.
(reproduced from [1.14])

its x,y,z and yaw, pitch and roll coordinates in a three-dimensional space, as shown in figure 1.2.

These few examples have been selected to illustrate the versatility of binary vision. Many systems and techniques have been described: see for example, references [1.11], [1.16], [1.17].

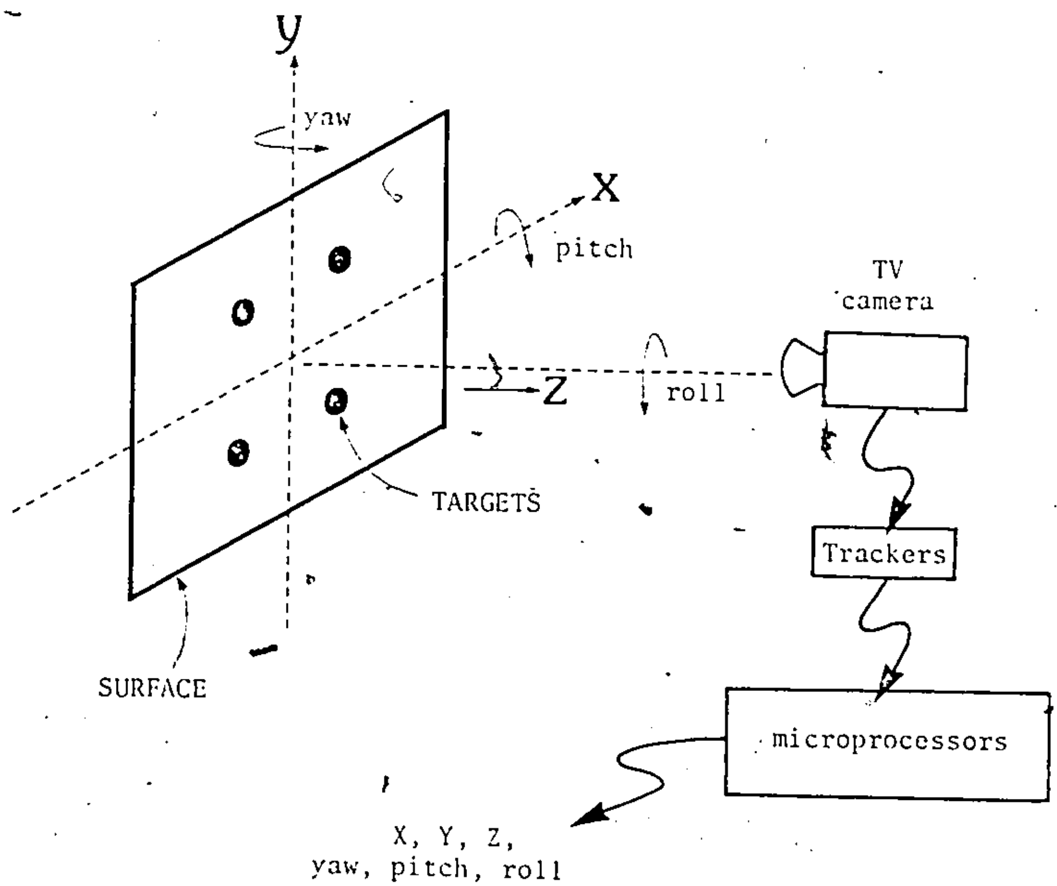


Figure 1.2 Real-Time Photogrammetry System.

This thesis is devoted to 2-D binary vision that is particularly suited to non-contact inspection and robotics applications and to its linking with an industrial robot. Two-dimensional projections of three-dimensional objects (silhouettes) are used to characterize workpieces by measuring a number of features including position, orientation, area, perimeter, circularity, maximum radius and number of holes. An identical set of features are measured for each hole. In inspection applications these measurements may be used for rapid checks on component presence or absence, workpiece alignment, dimensional tolerances, hole placement, and the like. For robot guidance they provide visual feedback for workpiece acquisition and for automated assembly. The primary requirements are processing speed, cost, and flexibility. Industry imposes two additional requirements of a vision system: one is that the system should leave the working area uncluttered, while the other concern is that of the user interface since the operator will invariably not be expert in computer vision.

Within this context, contributions have been made in the following areas:

1. <u>Vertex Point Extraction Algorithm</u>

A distinguishing feature of our system is the use of the "vertex point" encoding scheme for representing the contours of regions in a binary image. An efficient algorithm and data structures for the sequential extraction of vertex points from a raster scan have been developed. The new method generates polygonal approximations to the contours of silhouettes and associated holes in time independent of region complexity: any number of objects may exist in the frame and each may contain any number of holes. Objects can be intertwined or be situated within the holes of larger objects and this may occur to any depth. Arithmetic operations are restricted to comparisons, additions and subtractions, and with the use of dynamic data structures this technique is well suited to high speed implementations.

2. Implementation

A full hardware/software implementation of the vision system has been completed using state-of-the-art technology and software development tools. The objectives have been the following:

(a) To subject the new vertex point extraction algorithm and data structures to comprehensive tests using a wide range of realistic test objects. These have established the reliability and robustness of the new process.

(b) To obtain measures of the error characteristics of a complete vision system based on the vertex point technique. A variety of statistical tests have established the accuracy in discrimination and position measurement for a number of objects of varying size and shape.

(c) To link the vision system with an industrial robot. A software package was developed which allowed for the incorporation of the vision system in the control of a PUMA 600 robot. Through a serial interface, the VAL language has been extended to include high level commands for interaction with the vision. Extensive tests confirm that the vision system meets the accuracy requirements for reliable part acquisition.

(d) To establish the hardware and software requirements for an economical, stand-alone vision system including substantial consideration of the user interface. This has been achieved with three 6"x12" Multibus boards which include all of the processing and memory requirements. Software includes a data base for object prototypes, a dynamic data structure for efficient representation of frame contents and feature extraction from contours encoded with vertex points.

3. System Architecture

A unique multiple bus structure has been developed which supports parallel operation of dedicated modules to perform video-rate functions including histogram generation, binary image acquisition and the collection of grey-scaled images. The architecture also accomodates programmable thresholding for converting grey-scaled images to binary and programmable selection of background colour. A notable feature of this design is its flexibility, allowing for additional hardware to be inserted with minimal system reconfiguration.

The apparatus is configured as shown in figure 1.3. A solid state TV camera produces images of objects placed randomly on a back-lit table which has been positioned within the working envelope of the PUMA robot. The vision system, based on the 8086/8087 microprocessor, determines the identity, location and orientation of the silhouettes of each object in view. This information is then made available via the serial link to the robot controller which directs the arm to move and orient its gripper to pick up the desired objects. The system learns prototype objects in a "teach-by-showing" session with the user issuing commands through the CRT terminal. A complete specification of the system is given in Appendix A.

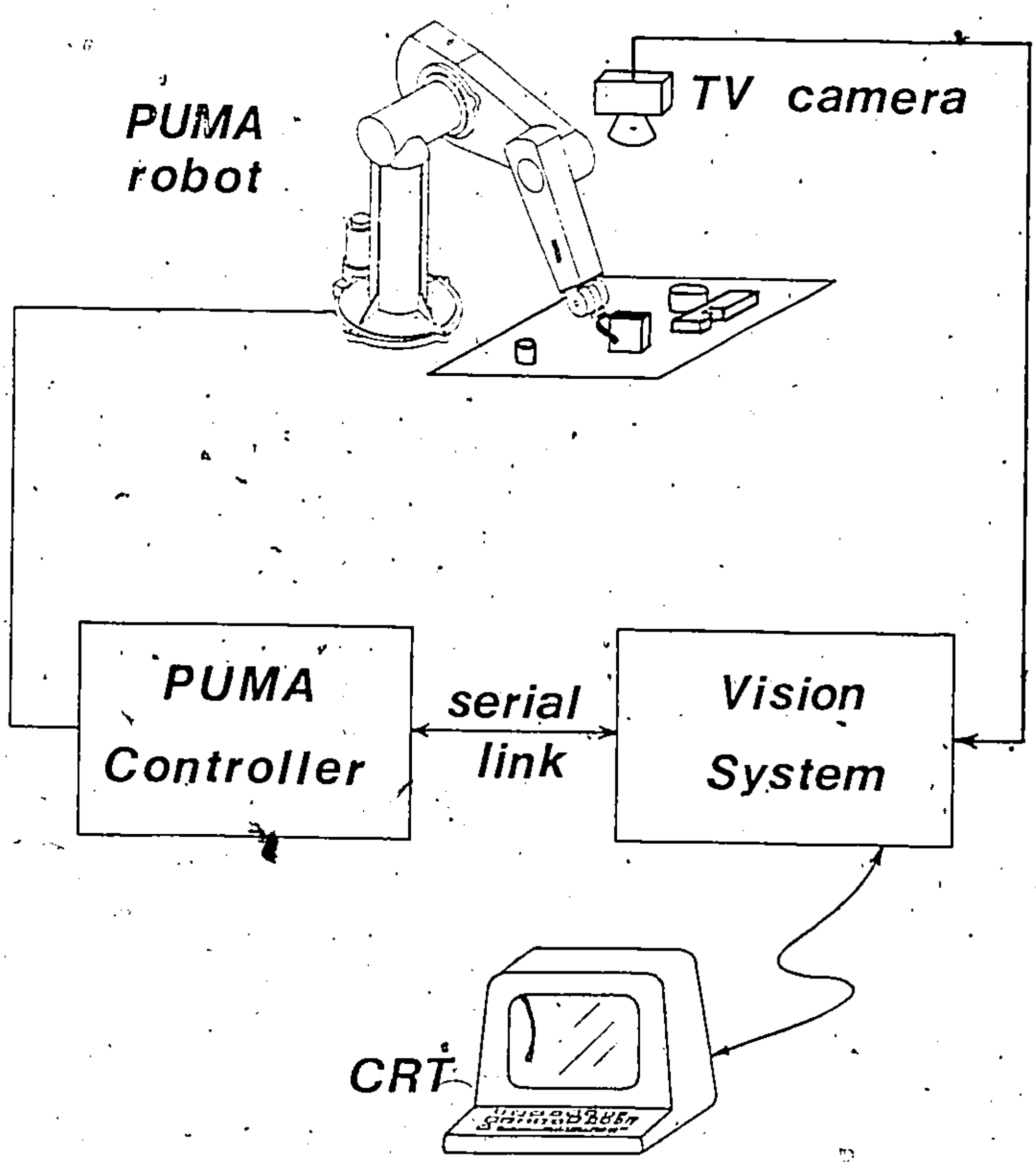


Figure 1.3 Configuration of Robot and Vision System.

Chapter 2 of the thesis constitutes a brief description of the well-known techniques for 2-D binary vision. Contour encoding methods are introduced as well as calculations used for feature extraction. Methods for silhouette recognition and the determination of location and angular orientation are also discussed. Chapter 3 begins with a brief overview of contour connectivity and chain coding as an introduction to boundary extraction algorithms. Those which are not suited to high speed implementation are discarded, leaving a selection of three sequential techniques, all of which are described in detail. Two of these, in turn, are found to require excessive post-processing leaving the method of Aggrawala and Kulkarni. The new algorithm and data structures are then described in detail and compared with those of Aggrawala and Kulkarni in terms of number of edge point comparisons, memory requirements, Object Array and *Active_ List* Structures, and representation of frame contents.

Implementation details are contained in Chapter 4. The system architecture is described, followed by functional descriptions of the binary pixel packer, the histogram generator, the threshold unit and the system status register. Software descriptions include that of the major data structures, data base design and the command set. The chapter concludes with a discussion of the robot/vision interface package, an outline of the robot/vision coordinate calibration procedure, and an example of parts sorting by the robot utilizing visual feedback.

A set of objects designed to test the connectivity capability of the system is presented in Chapter 5 together with examples of their processing. The efficiency of the boundary encoding scheme is examined, as are the effects of resolution. Performance of the system is evaluated from the point of view of execution time and accuracy in discrimination and in the determination of location and orientation of vertex point-encoded contours. Chapter 6 contains conclusions and comments and concludes with suggestions for further research.

CHAPTER 2

TECHNIQUES FOR 2-D SILHOUETTE VISION

The operation of current machine vision is functionally analogous to that of the human vision process. In terms of a silhouette vision system, five principal areas have been identified by Gonzalez [2.1]:

(i) Image Sensing - Visual information exists as a two-dimensional continuous function and must be spatially sampled and quantized to facilitate processing by a digital computer. This is typically achieved by an integrated array of photosensitive elements arranged as a matrix of 128x128 or 256x256 pixels with eight bits of grey-scale and scanning at a frame rate of 30 Hz.

(ii) Segmentation - Grey-scaled images are easily segmented into silhouettes if the scene illumination provides high contrast between background and objects. The histogram of such an arrangement is bimodal as shown in figure 2.1 and a threshold value between black and white is readily selected between the two peaks. Thresholds can be manually set or determined automatically with a variety of algorithms [2.2]. Each pixel in the image is assigned a 1 or 0 depending on its grey-scale magnitude. Storage requirements for binary images can be greatly reduced with the use of run-length encoding or vertex points.

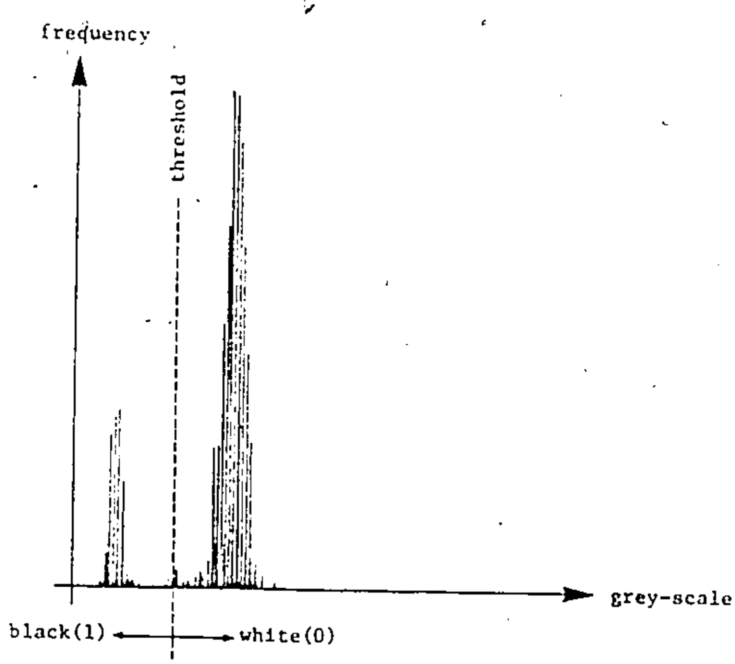


Figure 2.1 Histogram of High Contrast Scene.

(iii) Description - Silhouettes are characterized by measuring several features: area, perimeter, number of holes and others are typical. In addition, the number of separate objects in a frame and object/hole containment relationships must be detected and represented in an appropriate data structure.

(iv) Recognition - The identity of silhouettes is determined by comparing their features with those of a data base of prototype, or model objects.

(v) Interpretation - The interpretation of the contents of a binary image is very much dependent on application. In a robot control application, for example, the required information may be the identity, location and orientation of objects in the image. In an inspection task on the other hand, information about the presence (or absence) of components, their relative positions, and dimensional tolerances may be desired.

From the large canvass of the literature concerned with these areas, a small number of specific basic techniques have been selected for review, primarily in order to establish terminology used in this work. The references which have been cited also serve as an introduction to the extensive publications in this subject.

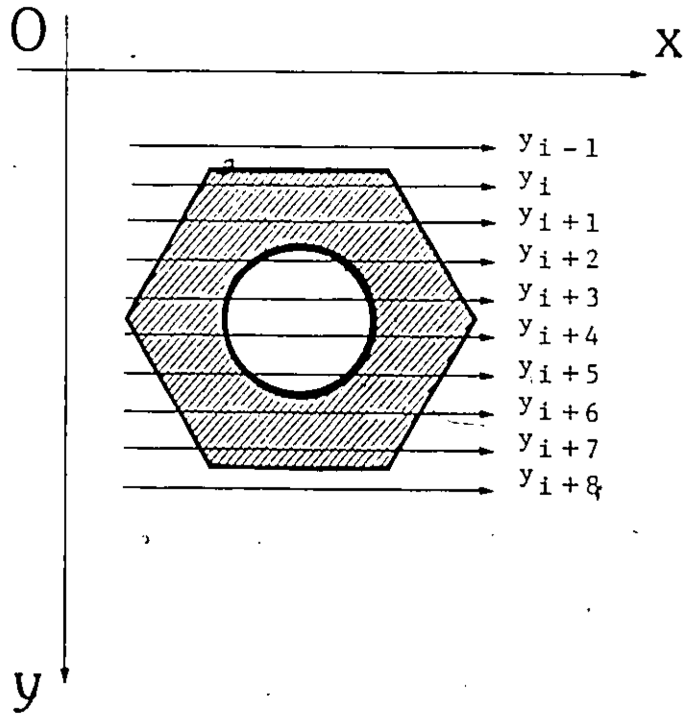
2.1 Run-length Encoding

The techniques of run-length encoding are widely used in the processing and transmission of images to achieve data compression [2.3]. The method is most effective for pictures composed of large regions of constant (or nearly constant) grey-level. During the raster scan, the lengths or coordinates of runs of uniform grey-level for each line can be specified.

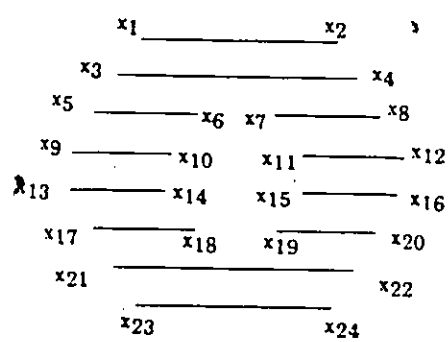
Binary images are particularly well suited to run-length encoding and are used in virtually all commercially available binary vision systems. Run-length encoding yields the x-coordinates of transition points between silhouette and background in each scan line of the frame. The original binary image is thus approximated line-by-line as pairs of transition points as illustrated in figure 2.2. For lines in which no portion of an object appears, no points need be stored (y_{i-1} and y_{i+8} in figure 2.2 a). For any line which intersects the object, the x-coordinates of each transition are recorded. For the example in figure 2.2, the run-length codes are:

line	transitions
y_i	x_1, x_2
y_{i+1}	x_3, x_4
y_{i+2}	x_5, x_6, x_7, x_8
y_{i+3}	x_9, x_{10}, x_{11}, x_{12}
y_{i+4}	x_{13}, x_{14}, x_{15}, x_{16}
y_{i+5}	x_{17}, x_{18}, x_{19}, x_{20}
y_{i+6}	x_{21}, x_{22}
y_{i+7}	x_{23}, x_{24}

Transitions in the binary video are readily detected at video rates using standard digital hardware. A simplified design is shown in figure 2.3. Two counters provide the m-bit y-coordinate (line number) and the n-bit x-coordinate of each run-length code. The vertical synchronizing pulse (VSYNC) clears the y-counter which



(a) raster scan of original binary image



(b) segments approximation

Figure 2.2 Run-Length Encoding of a Binary Image.

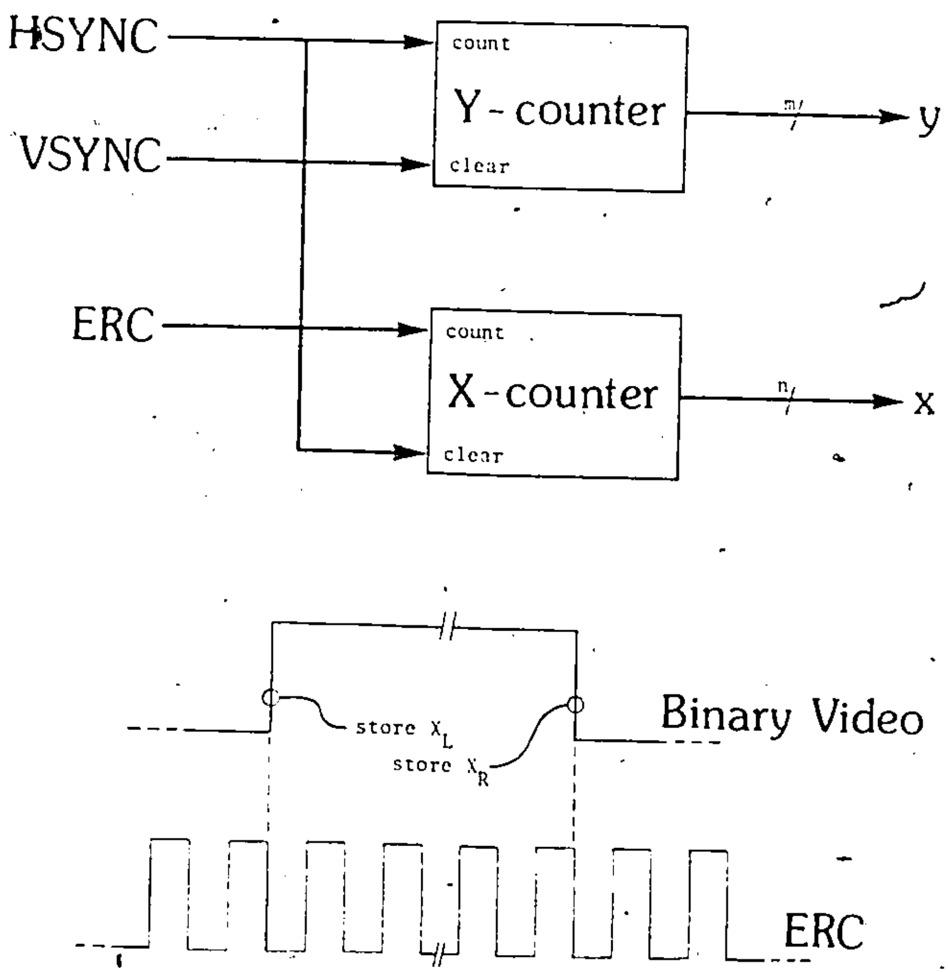


Figure 2.3 Simplified Run-Length Code Extractor

increments at each new line indicated by the horizontal synchronizing pulse (HSYNC). HSYNC is also used to clear the x-counter which is incremented by the element rate clock (ERC) pulse indicating the occurrence of a new pixel on the line. For each line therefore, the y-coordinate of the run-length codes is available from the y-counter and the left (X_L) and right (X_R) x-coordinates are read from the x-counter at the indicated transitions of the binary video.

2.2 Vertex Points

A pair of successive transition points on the same line define a *segment* and the (x,y) coordinates of these points will be referred to as *edge points*. A *left edge point* has the lesser x-coordinate of the pair; a *right edge point* has the greater. For example, in Figure 2.2 (b) valid segments are designated as $[x_1,x_2]$, $[x_3,x_4]$, $[x_5,x_6]$, $[x_6,x_7]$, $[x_7,x_8]$, etc. The corresponding edge points for segment $[x_1,x_2]$ are (x_1,y_i) and (x_2,y_i). For segment $[x_3,x_4]$, they are (x_3, y_{i+1}) and (x_4,y_{i+1}) and so on. An *object region* is a set of connected segments which are of opposite 'colour' from the background. *Hole regions* are a set of connected segments of the same colour as the background but entirely surrounded by an object region. Segments are considered to be connected if they occur on consecutive lines and there is an overlap in the intervals which they span on the x-axis. Whenever any portion of an object region appears in the current scan line of the raster, that object region is said to be an *active object*. At the first occurrence of a line which contains no part of an object which has previously been active, that object is said to be a *complete object*.

The contour of the object region of our example is defined by the ordered list of edge points (x_1,y_i), (x_3, y_{i+1}), ... (x_2,y_i). Similarly the contour of the hole region is (x_7,y_{i+2}), (x_{11},y_{i+3}), ... (y_6,y_{i+2}). These points can be regarded as the vertices of a polygonal approximation to a region and obviously any collinear points can be eliminated from the edge point list without loss of shape information. The remaining set of points have been designated as *vertex* *points* and are shown in figure 2.4. The number of vertex points to represent a contour is always less than or equal to the number of edge points needed. In practice, however, the number of vertex points is often considerably less than the number of edge points. It is convenient to represent vertex points in a linked list of (x,y) coordinates as shown. This structure will be referred to as the *vertex_list*. Vertex points are listed according to the following standards:

- Object region vertices are linked counterclockwise and one vertex point is specified as a starting point. This point is that of maximum y-coordinate with corresponding left-most x-coordinate. (V_{11} in figure 2.4)

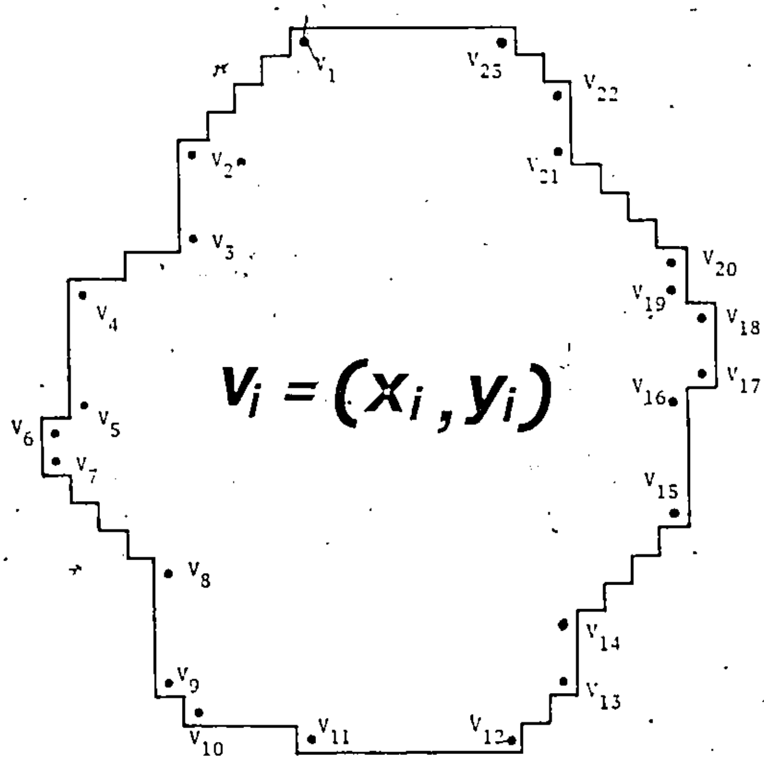
- Hole region vertices are linked clockwise. The starting point for a hole vertex list is that point of maximum y-coordinate with corresponding right-most x-coordinate. (V_{12} in figure 2.4)

The process of constructing a vertex list for each object region and any associated hole region in a frame is a form of connectivity analysis [2.4] [2.5].

2.3 Contour Navigation

Object and hole regions are characterized by measuring a number of properties of each contour. This 'feature' set is used to obtain the following information:

a. Vertex Points :



b. Data Structure :

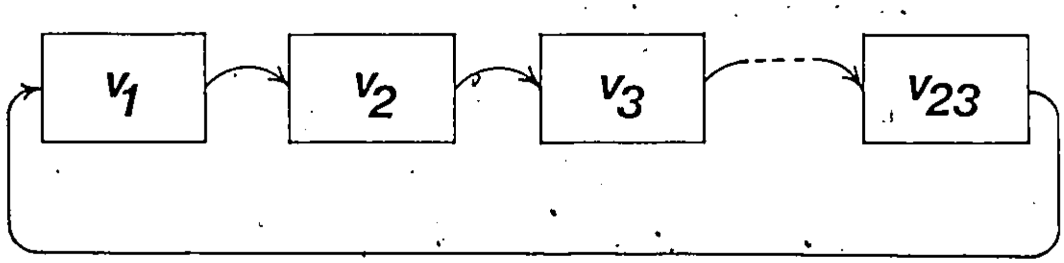


Figure 2.4 Vertex Point Representation of a Region.

(a) vertex points, (b) data structure

(i) identification: the identity of each silhouette is determined by comparing its feature set to those of prototype objects which have previously been stored in a data base.

(ii) position: the (x,y) coordinates and angular orientation (with respect to the camera coordinate axes) are measured for each contour.

Features are calculated for each object and hole contour by 'visiting' each point in the vertex list and are classified according to the time at which they can be determined:

(i) first navigation: features which are found by accumulating quantities which depend only on the coordinates of each vertex point.

(ii) second navigation: features which are found by accumulating quantities which depend on the coordinates of each vertex point and on features found on a first navigation.

(iii) post-navigation: features which are derived from those found on a first or second navigation.

The choice of feature set is highly dependent on the class of silhouettes which are to be encountered. Table 2.1 summarizes a number of popular features but is not exhaustive. The features which have been implemented are readily supplemented or changed as the application warrants.

Given the contour consisting of the set of vertex points $[(x_i,y_i), i = 1,N]$, the following first navigation calculations have been implemented:

Area – the area enclosed by a contour is given by:

$$1/2 \sum_{i=1}^{N} [x_i \, (y_{i+1} - y_{i-1})]$$

where subscripts are reduced modulo N. For counter-clockwise traversal, the area will be positive.

Perimeter – the perimeter of the contour is given by:

$$\sum_{i=1}^{N} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

Centroid – the coordinates of the center of area of a contour (X_c,Y_c) are evaluated by viewing the contour as a set of connected links with endpoints at consecutive vertex points. Consider the contour of figure 2.5 where (x_{i-1}, y_{i-1}) and (x_i,y_i) are any two consecutive vertex points separated by a distance ℓ_i.

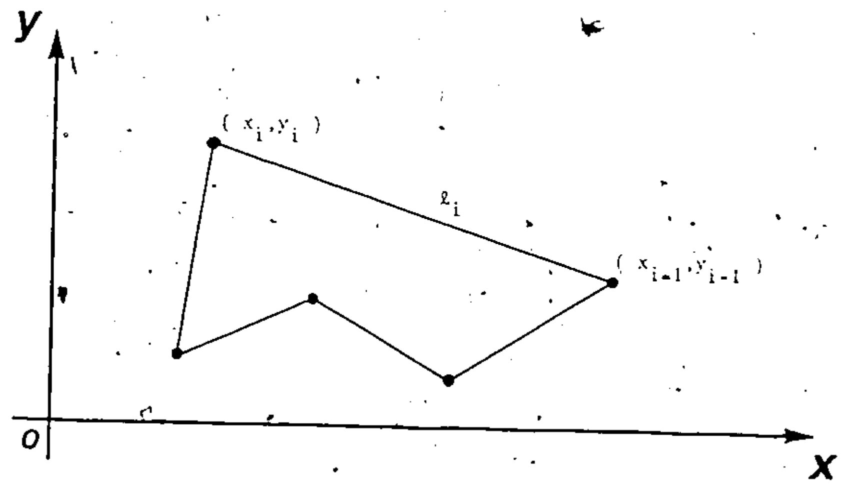


Figure 2.5 Centroid Determination.

The moment about the x-axis is given by the product of the abscissa of the centroid and the total length of the contour. This quantity is equal to the sum of the moments about the x-axis of each link with centroid coordinates $(\bar{X}_i, \bar{Y}_i)$ and length ℓ_i [2.6]. That is,

$$X_c \, \Sigma \ell_i = \Sigma \, \bar{X}_i \ell_i$$

Similarly, moments about the y-axis can be equated:

$$Y_c \, \Sigma \ell_i = \Sigma \, \bar{Y}_i \ell_i$$

The centroid coordinates of the composite structure are therefore,

$$X_c = \frac{\Sigma \, \bar{X}_i \ell_i}{\Sigma \ell_i} \quad \text{and} \quad Y_c = \frac{\Sigma \, \bar{Y}_i \ell_i}{\Sigma \ell_i}$$

The centroid of each link is located at

$$\bar{X}_i = \frac{x_i + x_{i-1}}{2} \quad \text{and} \quad \bar{Y}_i = \frac{y_i + y_{i-1}}{2}$$

and the distances ℓ_i are given by

$$\ell_i = \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}$$

The summation $\Sigma\ell_i$ is the perimeter as defined previously.

On the second navigation, we obtain:

Maximum radius – the maximum of the set of radii defined by the contour centroid (X_c, Y_c) and each of its vertex points (x_i, y_i):

$$\underset{i=1,N}{\mathrm{MAX}} \left| \sqrt{(x_i - X_c)^2 + (y_i - Y_c)^2} \right.$$

In addition, the 'circularity' (P2A) of a contour is also calculated after the first navigation:

$$\mathrm{P2A} = (\mathrm{Perimeter})^2/(4\pi \times \mathrm{Area})$$

Feature	Used for Object Contours	Used for Hole Contours	When Obtained
Area	X	X	1
Perimeter	X	X	1
Circularity (P2A)	X	X	P
Number of holes	X		P
Total hole area	X		P
Total hole area/Area	X		P
Maximum radius from centroid	X	X	2
Minimum radius from centroid	X	X	2
Average radius from centroid	X	X	2
Standard deviation of radius	X	X	2
Width of Bounding Box	X	X	1
Height of Bounding Box	X	X	1
Position of hole within object	X	X	P

1 = First navigation

2 = Second navigation

P = Post-navigation

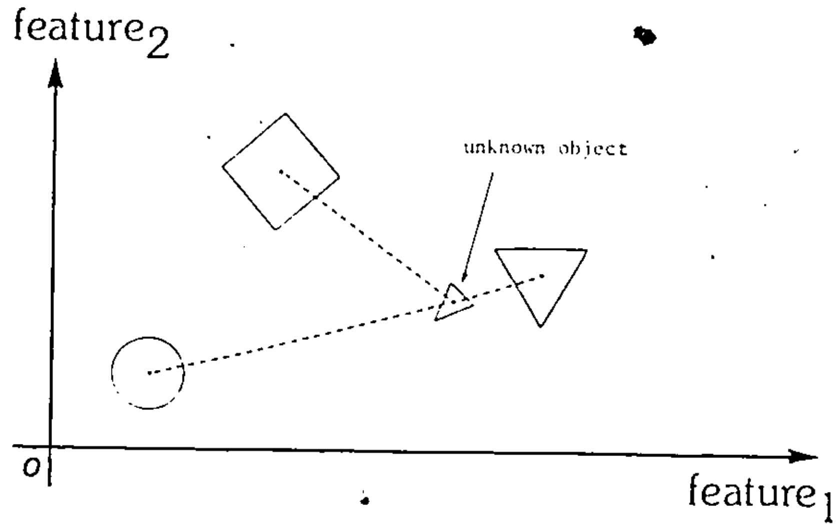
Table 2.1 Some Silhouette Features

2.4 Object Recognition

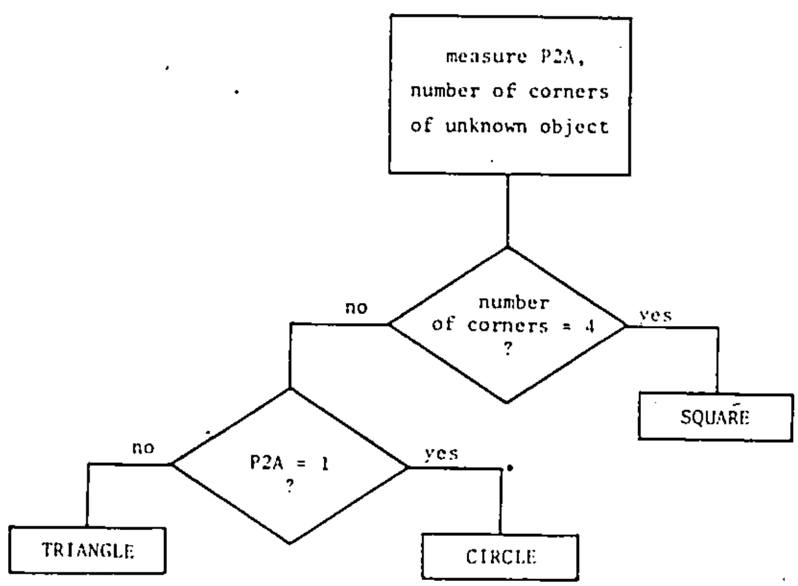
Given a data base of prototype objects, the objective is to decide which prototype feature set is the best match to an unknown object. Two popular methods are described by Agin [2.7] and outlined here:

(a) Nearest Neighbour - With a set of 'M' features, the prototypes can be viewed as points in an M-dimensional space. The distance from the unknown silhouette to each prototype is calculated as the sum of the squares of the differences of each measured feature. Identification is established as the prototype for which this distance is minimum. Figure 2.6 (a) illustrates an example for two features and prototypes of a circle, square and triangle. The distance from an unknown object (another triangle) is calculated to each prototype and the minimum is found to be for that of the triangle prototype.

(b) Binary Decision Tree - Recognition of unknown objects based on a decision tree is illustrated in figure 2.6 (b). From the feature set, one is selected which divides the known prototypes into two distinct classifications (number of corners in this case). Each of these is similarly divided by other features until each prototype appears as a leaf in the tree. To identify an unknown object, tests are made on the selected features sequentially, until a leaf is encountered.



(a) two-dimensional nearest neighbour



(b) a simple binary decision tree

Figure 2.6 Classification Strategies.

The nearest neighbour method provides accurate discrimination but has the disadvantage of requiring a large number of calculations. Classification with a binary decision tree is fast but it is often difficult to choose the appropriate features for constructing the tree. In our vision system, object identification is based on a simple feature by feature comparison of unknown objects with prototypes. If each feature of the unknown silhouette is equal to the corresponding feature of a prototype object, then identification is established. The tolerance with which each feature can differ from that of the prototype and still be considered equal is set by the user in the tolerance vector. The tolerance vector contains corresponding elements to the feature vector and are specified as percentages.

The feature vector currently in use consists of:

Area	-	the area enclosed by the object contour
Perimeter	-	the perimeter of the object contour
P2A	-	$(\text{Perimeter})^2/(4\pi \times \text{Area})$
Max_rad	-	the maximum of the set of radii from the centroid of the object to each of its vertex points
No_holes	-	the number of holes contained in the object (the tolerance for this feature is not specified as a percentage, but as an absolute integer value)

The advantage of this technique is that the relative importance of each feature can be adjusted for a particular application by setting its tolerance value. If the area of a workpiece is required to be within a certain range of the prototype for example, then its tolerance can be set to detect that range. Features which are of little importance can be effectively ignored by choosing large tolerance values. In practice, the size of an object within the frame and its position will introduce errors in feature measurement, typically of the order of 1 to 10 percent. Some examples are given in section 5.5. To eliminate a feature from the recognition process, its tolerance would be set so that the corresponding feature for every object in the data base will be considered equal.

2.5 Location and Orientation

The location of objects or holes within a field of view are reported as the coordinates X_c,Y_c (in mm) of the centroid of the contour (figure 2.7). Centroid locations for any holes may be used for measuring the location of the hole(s) within the object.

The orientation of a silhouette is currently specified as the angle which the maximum radius vector makes with the x-axis of the camera frame within the range -180° to $+180^{\circ}$ (see figure 2.7). This is one of several methods available for determining angular orientation but is not necessarily unique for some objects. An example is shown in figure 2.8 in which the orientation of a rectangle cannot be discerned with the maximum radius vector. In general, the method is appropriate for objects with a unique maximum radius vector or those that have greater than two axes of symmetry (circles, squares, the isoceles triangle and regular pentagons, hexagons, etc.)

For objects which contain identifiable holes, orientation can be determined from the location of the holes as shown in figure 2.9. The orientation is specified as the angle of the vector between two hole centroids (figure 2.9 a) or between the object centroid and a hole centroid (figure 2.9 b). The method fails if an object contains multiple holes from which one or two cannot be uniquely identified.

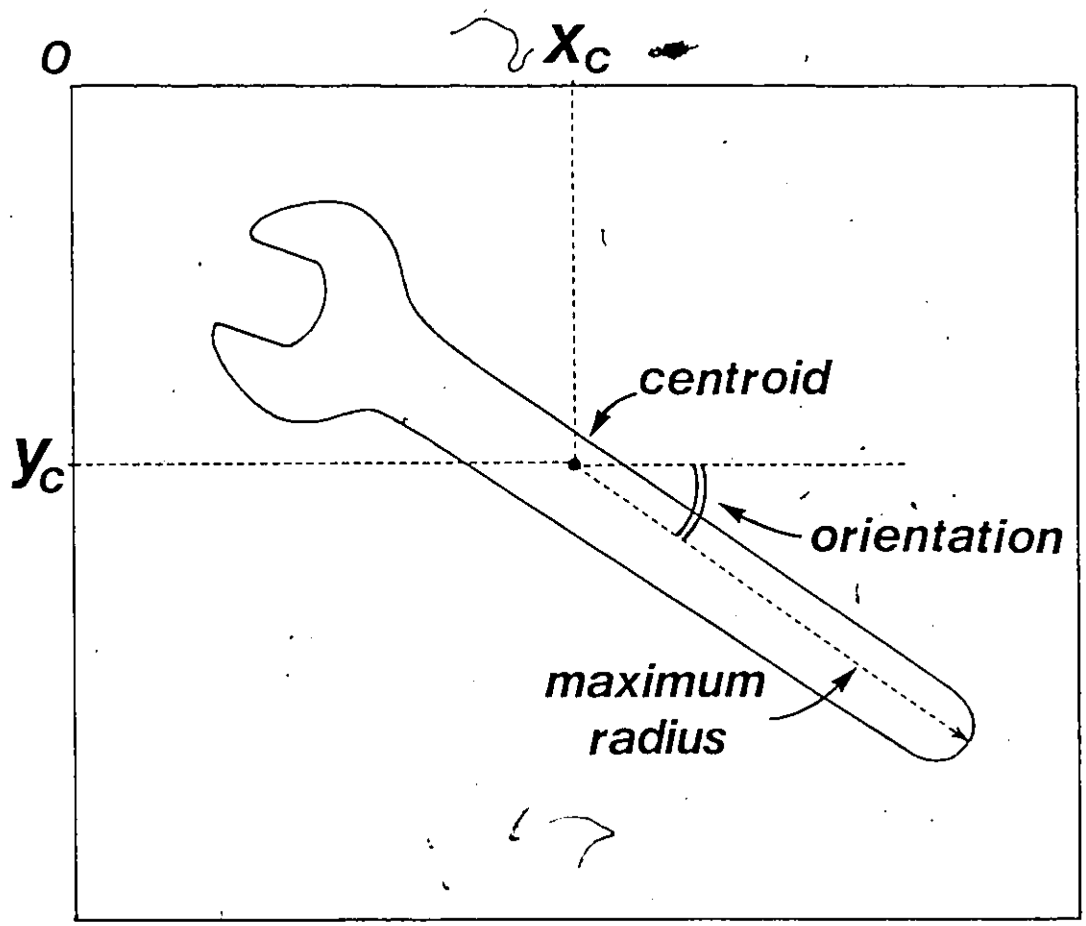


Figure 2.7 Object Location and Orientation.

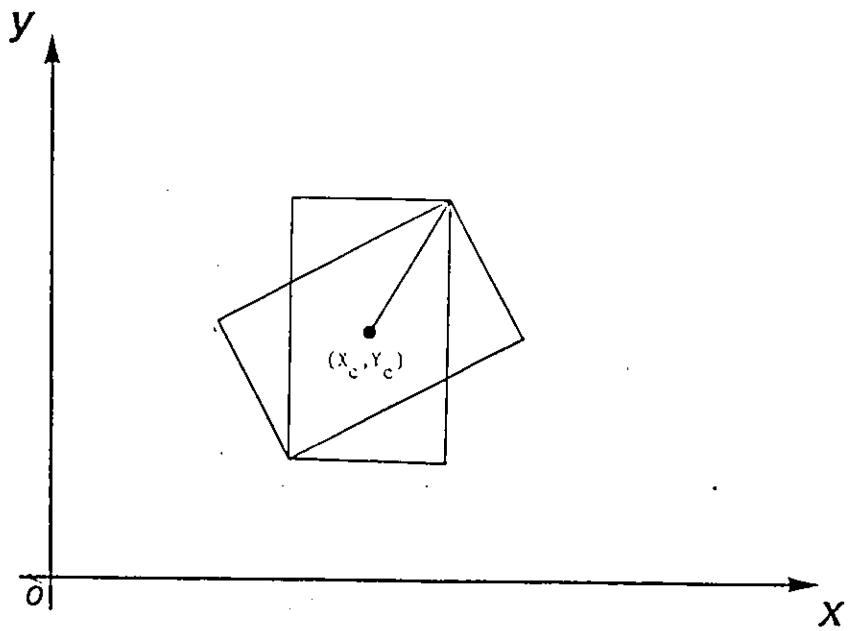


Figure 2.8 Example of Object for which Orientation Cannot be Determined from Maximum Radius.

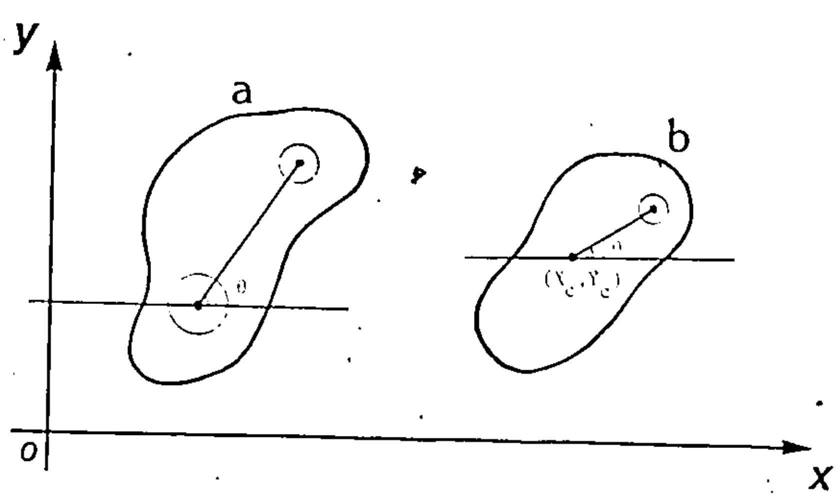


Figure 2.9 Determining Orientation (θ) by Hole Location.

A popular technique for establishing orientation is by way of the second moment. The orientation is estimated by the angle (θ) of the principal axes:

$$\theta = \frac{1}{2} \tan^{-1}\left(\frac{2M_{xy}}{M_{yy} - M_{xx}}\right)$$

where M_{xx}, M_{yy} are the second moments about the centroid (X_c, Y_c) given by

$$M_{xx} = \sum_{i=1}^{N} (X_i - X_c)^2 , \qquad M_{yy} = \sum_{i=1}^{N} (Y_i - Y_c)^2$$

and $M_{xy} = \sum_{i=1}^{N} (X_i - X_c)(Y_i - Y_c)$ is the product of inertia.

For objects in which the quantities M_{xx}, $M_{yy} = 0$ (for example, a square), the equation for θ becomes indeterminate. This condition should be checked before the division is carried out. An example of this method is shown in figure 2.10.

Further descriptions of techniques for determining orientation can be found in [2.8] [2.9].

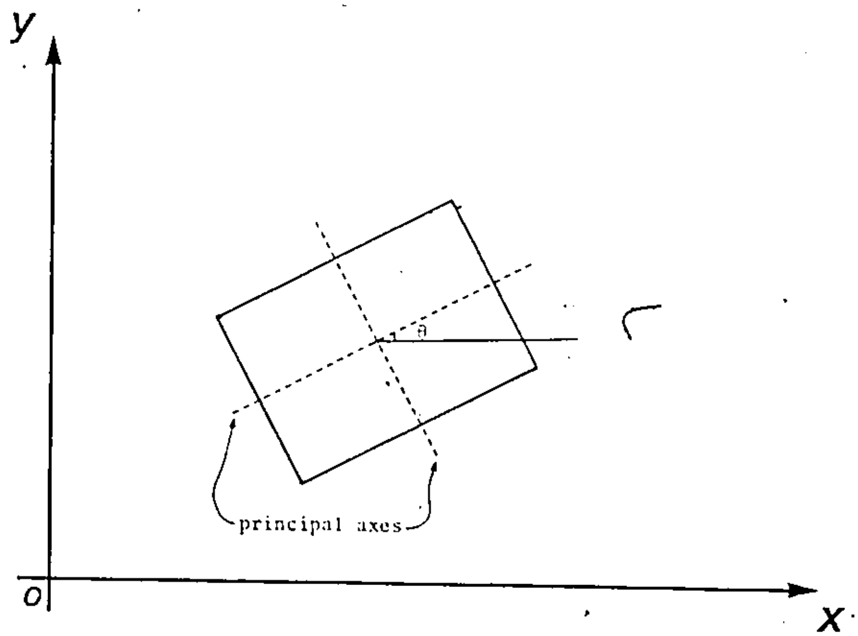


Figure 2.10 Determining Orientation by Angle of Principal Axes.

CHAPTER 3

BOUNDARY EXTRACTION ALGORITHMS

Binary image processing is largely concerned with the measurement of shape, location and orientation of connected regions in a given scene. Although a number of features used to determine these quantities can be extracted from examination of the entire region (e.g. area counts, moment accumulation), this information can be calculated from knowledge of their boundaries alone. The objective has been to design an algorithm which would be suitable for video-rate processing (boundary extraction during the horizontal retraces of a single frame scan) so that boundary point data is available immediately at the conclusion of the scan. Such a procedure would greatly enhance the visual processing task in high speed inspection or robot guidance applications.

A suitable algorithm must be sequential in operation, that is, the segments are processed line-by-line as they appear from the raster to avoid the large storage requirements of a complete frame. Furthermore, the processing speed for any segment should not depend on the shape or number of contours previous to that segment so that a maximum execution time can be determined. With this requirement, the minimum number of segments per line for which processing is guaranteed before the occurence of the next line can be specified. It is also desirable to eliminate collinear points on the boundary to minimize storage and subsequent analysis time and to maintain the containment relationships of holes to objects.

This chapter begins with a brief introduction to the connectivity characteristics and chain code representation of binary regions. An elementary contour tracing procedure is presented to illustrate the drawbacks of such techniques followed by an outline of several border tracing strategies which are shown to be inappropriate for high speed implementation. In section 3.1, three sequential border extraction methods are reviewed; the first two methods utilize chain codes and require additional procedures to complete the processing at the conclusion of the frame and are therefore not suitable. An improved algorithm is described in section 3.2 and this is compared in terms of execution efficiency and memory requirements to the third method (of section 3.1) in section 3.3.

A border point $P_{i,j}$ in a binary image is readily detected by checking to find if any of its neighbours are different in value from that of $P_{i,j}$. Neighbours may be considered in the 8-connected or 4-connected sense as illustrated in figure 3.1. If all eight neighbours of a pixel $P_{i,j}$ in a rectangular array are considered adjacent to $P_{i,j}$, then the region is said to be 8-connected. If only the four horizontal and vertical neighbours of $P_{i,j}$ are taken, then the region is considered to be 4-connected. The distinction becomes important when the proximity of two regions fall within a single pixel width. For example, in figure 3.2 (a) one or two regions are detected depending on the selection of the connectivity sense; in the 8-

connected sense cell P is adjacent to Q whereas in a 4-connected sense it is not. In figure 3.2 (b), the hole region would merge with the background in 8-connectedness by virtue of the adjacency of cells R and S but is isolated if 4-connectivity is assumed. A popular strategy is to consider object regions to be 8-connected and hole regions as 4-connected. This would result in figure 3.2 (b) being interpreted as an object containing a single hole. The connectivity of digital pictures has been formally treated by Rosenfeld [3.1].

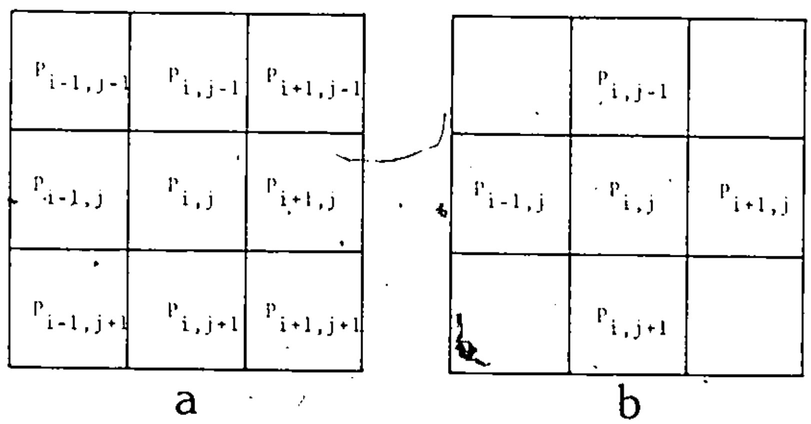


Figure 3.1 The Neighbours of Pixel $P_{i,j}$.

(a) 8-connected, (b) 4-connected

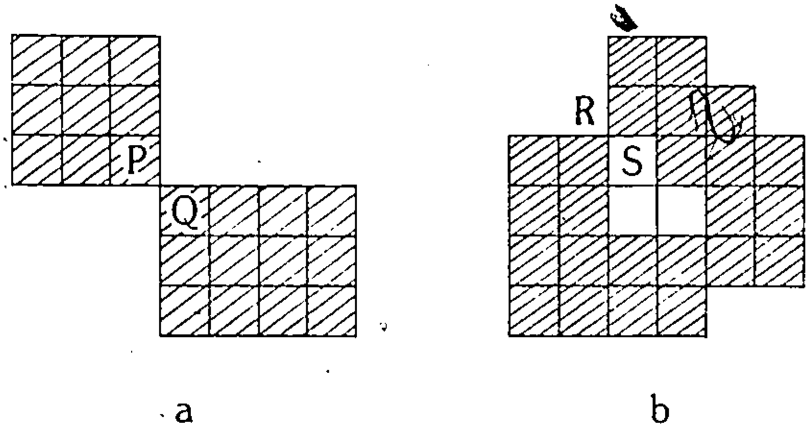


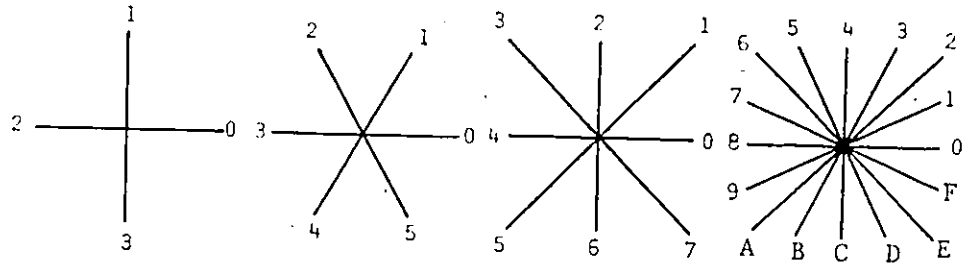
Figure 3.2 Connectivity of Regions.

(a) object, (b) hole

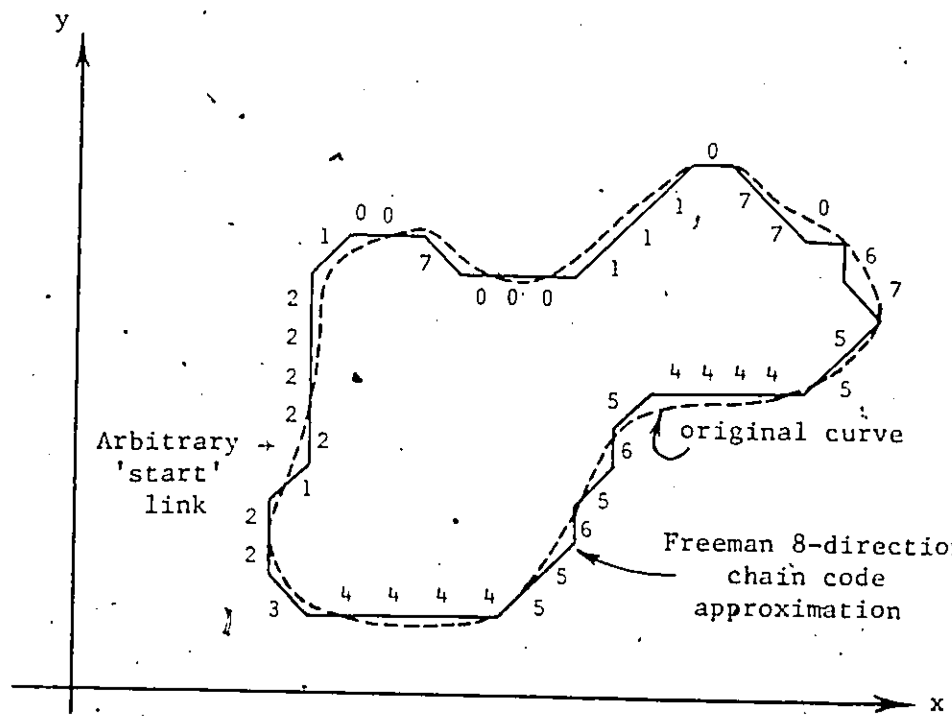
A complete border representation must retain not only the Cartesian coordinates of each border point, but the sequence of points which will yield a clockwise or counterclockwise traversal of each region. A widely known method is that of chain coding as introduced by Freeman [3.2]. The chain code of a contour consists of the coordinates of a start point followed by a series of directed links indicating the position of each boundary point relative to the previous boundary point (figure 3.3 (a)). The process is illustrated in figure 3.3 (b) (c) for an 8-connected contour. An 8-element chain code can be extracted by tracing a contour by way of the simple procedure described in [3.3]:

```
trace: procedure;
       scan the image until a pixel within the region is found
       (this is taken as the start point);
       repeat until the start point is reencountered;
          if this pixel is in the region then turn left;
                                          else turn right;
          move forward one pixel;
       end;
end;
```

An example of the path traced by this method is shown in figure 3.4. For the example shown, the initial trace first intersects the region at pixel p. Since p is in the region, we turn left and step to q. Pixel q is outside the region, so we turn right and step to r. The procedure is repeated, resulting in visiting pixels p, q, r, s, t, u, v, w, t, x, etc. It is evident that many more pixels are visited than



(a) Left to Right: Chain code links for 4, 6, 8 and 16 directions.



(b) An arbitrary curve and its chain-code approximation.

22222 10070 00111 07706 75544 44565 65544 44432 21

(c) 8-direction chain code of the above boundary.

Figure 3.3 Freeman Chain Code.

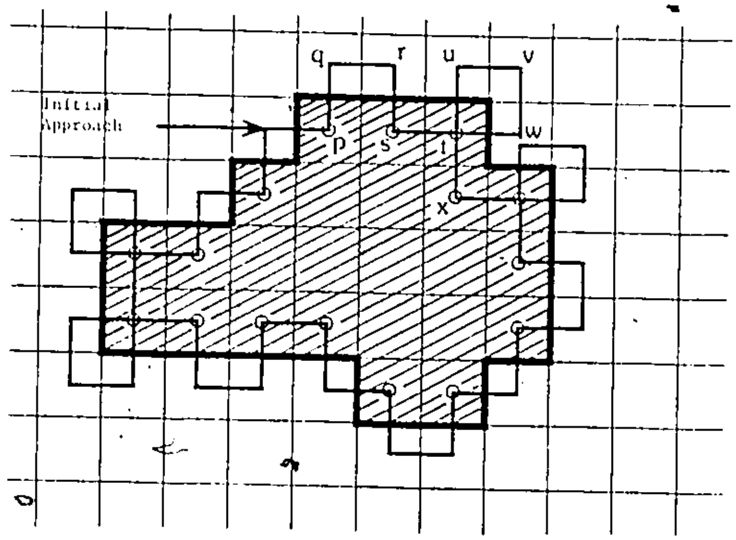
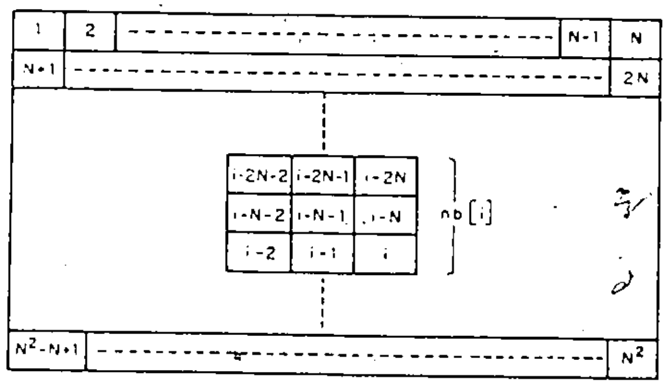


Figure 3.4 Simplistic Contour Tracing.

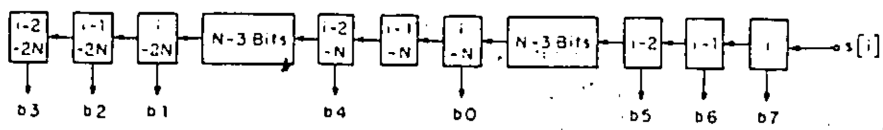
are actually on the border and in several instances, border points are examined twice (pixel t, for example). Assuming that we wish to follow all of the borders in a given image, we might systematically scan the picture executing the trace procedure each time a border point is encountered. However, to avoid retracing the same contour several times, border points must be marked during the initial trace. The trace procedure is then subsequently invoked only for unmarked borders. Additional labels must be applied if it is required to distinguish between object and hole contours. Although these problems have been considered in more sophisticated versions of this scheme such as those described by Pavlidis [3.4] and Dudani [3.5], the approach clearly suffers from a number of inefficiencies.

The literature includes numerous examples of boundary extraction algorithms. An approach described by Sobel [3.6] makes use of a neighbourhood encoding technique. The method utilizes a hardware tapped shift register for assembling the eight neighbours of each pixel into a single byte during the raster scan. The raster scan sequence is illustrated in figure 3.5 (a) with the neighbourhood nb(i) of pixel i-N-1. The serial bit stream s(i) from the raster is fed to a shift register of length 2N+3 bits as shown in figure 3.5 (b). At the occurrence of each clock pulse (i), the eight neighbours of pixel i-N-1 are read in parallel and stored in bit positions b_0 to b_7 of a single byte as indicated. There is a delay of N pixels before the first neighbourhood is available and the top and bottom rows and left- and right-most columns of pixels are not encoded since these positions do not have a complete neighbourhood. The 8-bit quantities stored for each pixel then represent the values of the eight neighbours of each pixel in the original image. An example of the encoder output is shown in figure 3.5 (c) for a 7x7 section of an image. The advantage of the technique is the capability of accessing all eight neighbours of any pixel in a single operation. Using simple table look-up procedures with the encoded pixels, successive points on the boundary are quickly identified. A fast contour follower is described which generates an 8-direction chain code boundary representation.

The methods that have been outlined thus far have the disadvantage that they require storage of a full frame of data to permit access to points in the image in an arbitrary fashion (the neighbourhood encoding procedure requires the equivalent of eight frames of binary image data). An algorithm presented by Pavlidis [3.7] does not require storage of an entire image; however an intermediate data structure known as the line adjacency graph (LAG) must be constructed from which contour points are then determined.



(a)



(b)

00	80	C0	40	20	00	00
80	C1	71	F0	F0	60	20
81	C3	F7	7D	39	10	10
83	C7	FF	7E	3E	0C	08
03	87	5F	3C	18	00	00
02	07	0E	1C	08	00	00
00	02	04	08	00	00	00

(c)

Figure 3.5 Neighbourhood Encoding for Contour Following.
(a) hardware encoder,
(b) tapped shift register,
(c) example encoder output (hexadecimal) for 7x7 section of image.

As shown in figure 3.6 (b) nodes are entered into the structure corresponding to segments of the binary image. Node 1 for example, represents segment [a,b] (of figure 3.6(a)) and this is divided into two nodes (2 and 3) representing segments [c,d] and [e,f] which are known to overlap with segment [a,b]. Similarly, when a segment joins two segments from the previous line (segment [k,l] connects segments [g,h] and [i,j] for example), the corresponding entries in the LAG are connected (nodes 9, 10 to node 12 in the example). The exterior boundary of a region is then traced from the exterior nodes of the LAG (nodes 1, 2, 4, 6, 9, 12, 14, 12, 10, 13, 11, 8, 5, and 3) and hole boundaries are determined from nodes on interior loops. For this example these are nodes 4, 6, 9, 12, 10 and 7 for the left hole and nodes 1, 2, 4, 7, 10, 13, 11, 8, 5, and 3 for the right hole.

A similar drawback exists in the extraction of boundary descriptions by way of quadtree representations of binary images as described by Dyer, Rosenfeld and Samet [3.8]. The quadtree structure is demonstrated in figure 3.7. The original binary image is divided into four quadrants of equal area, designated as north-west (NW), north-east (NE), south-west (SW) and south-east (SE). Each of these quadrants are similarly divided until the smallest quadrant consists of four individual pixels. These divisions are stored in the structure of figure 3.7 (b). The central node (CN) representing the entire image, contains pointers to nodes representing the four sub-quadrants. Each of these is further divided as long as the corresponding quadrant is not of uniform color. When a quadrant is

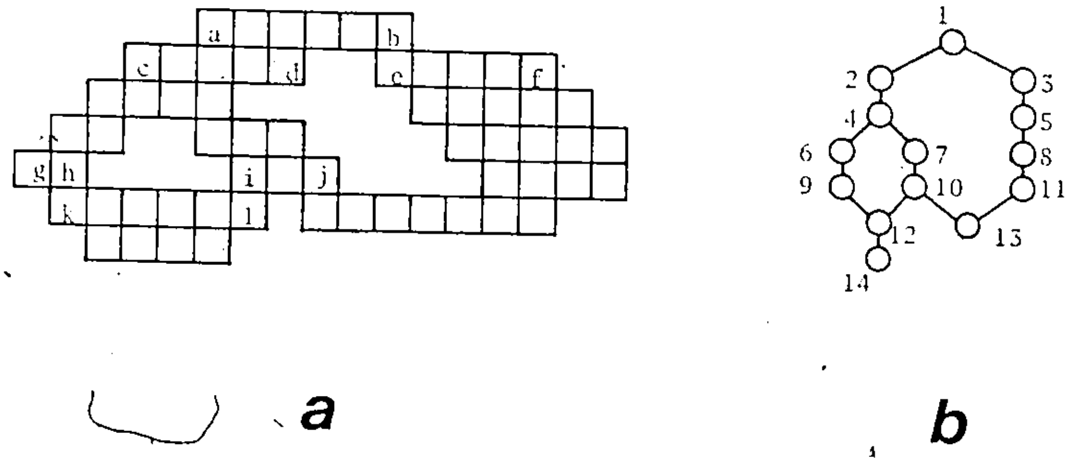


Figure 3.6 Line Adjacency Graph Representation.

(a) segments, (b) LAG.

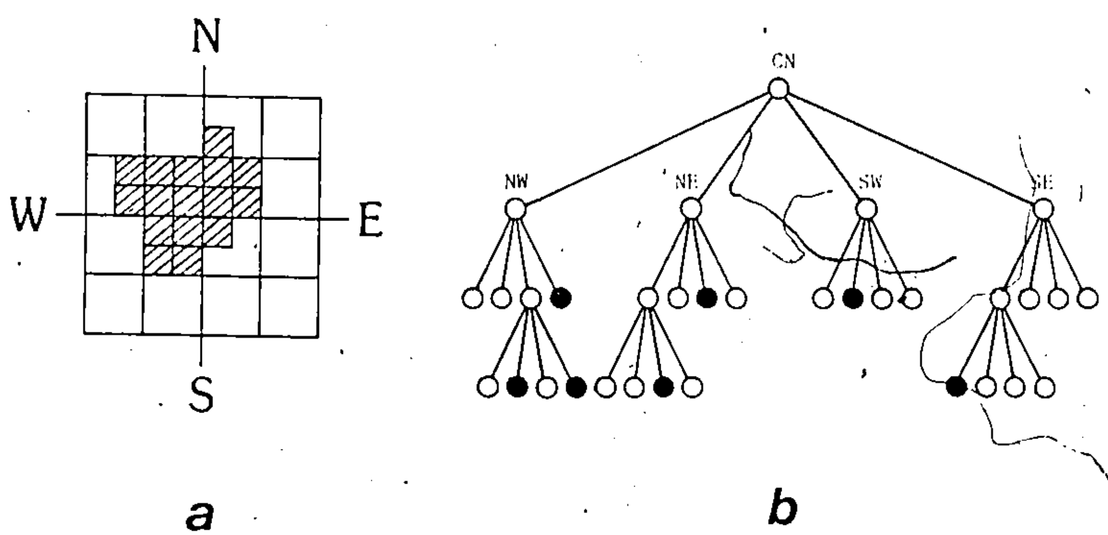


Figure 3.7 Quadtree Representation.

(a) original binary image divided into quadrants,

(b) quadtree.

homogeneous in color, a leaf is entered in the tree. For this example, the SW quadrant comprises four sub-quadrants, all of which are uniform. Three white leaves and one black leaf are then appended to this node. The NE quadrant on the other hand, contains three regions of uniform color (two white, one black) but the fourth region (NW) must be further divided. Given a complete quadtree structure, an algorithm is then described for tracing the region boundary by traversing the tree. Obviously neither this method nor that based on the LAG would be suitable for high speed implementations.

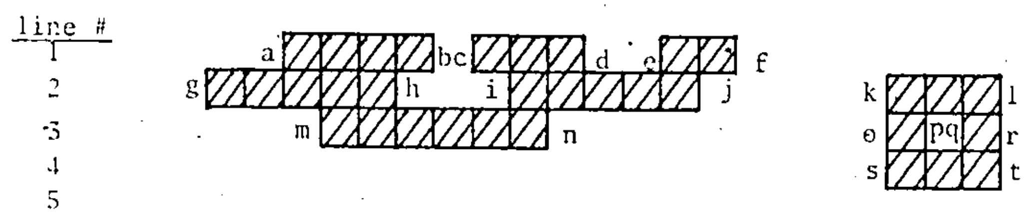
3.1 Sequential Techniques

A great deal of storage and processing time can be saved if border extraction is carried out sequentially. Border representations are better constructed from row-by-row presentation of the image (as in a raster scan), obviating the need for large frame buffers or conversions to intermediate data structures. The only image storage requirements should be that for two consecutive lines of segment data. For example, the algorithm of Grant and Reid [3.9] processes run-length segments in raster sequence however, additional storage is required to access segments on earlier lines in a non-sequential order. In this section we review in detail three methods which may be classified as sequential in operation.

Method 1

A method described by Rosenfeld and Kak [3.10] sequentially converts run-length segments into 4-connected chain codes for each boundary in a binary image. The algorithm results in a counter-clockwise representation of object contours and clockwise for hole borders. Its operation is illustrated by way of the example of figure 3.8.

(a)



(b)

line #

1 122223, 12223, 1223
2 10122223223, 101223221222303, 12223
3 1000010122322122230322210122223223000 3, 1122233, 103
4 10000101223221222303222101222232230003000000, 111222333, 1032
5 10000101223221222303222101222232230003000000, 111222333000, 1032

(c)

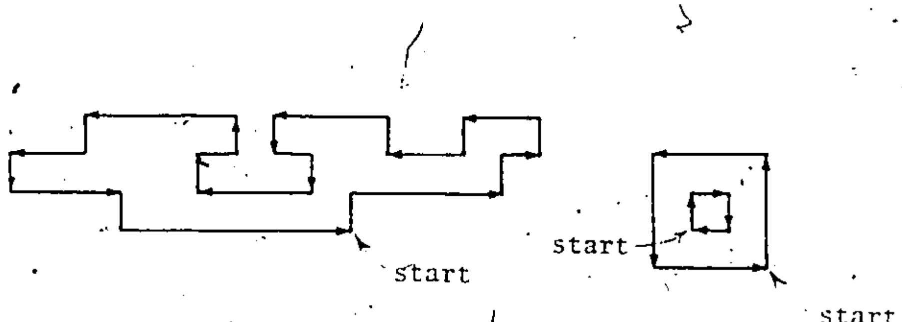


Figure 3.8 Sequential Boundary Extraction: Method 1.
(a) original input image, (b) 4-direction chain code at each line, (c) paths traced.

For clarity, the segments have been labelled [a,b] ... [s,t] in their raster order and the new elements of the strings added at each line are underlined. The procedure is outlined below:

(1) For each segment on the first line, a string of the form 12...23 is created with the number of 2's equal to the length of the segment. The beginning of the string represents the right end of the corresponding segment; the end of the string represents the left.

(2) When a segment on a new line overlaps with a single segment on the previous line, additions to the string are made as follows. If the segment on the new line extends to the left of the segment on the previous line (eg. [g,h] and [a,b] of the example), then append a string of the form 2...23 to the end with the number of 2's equal to the length of overlap (a-g). This can be seen in the first string of line #2 in figure 3.8 (b). If the left edges were perfectly aligned (a=g) then a single 3 would be added. A string of the form 0...03 is added at the end if the segment on the previous line extends to the left of that on the new line. Similarly, a string of the form 12...2 is inserted at the beginning if the new segment extends to the right of that on the previous line and a single 1 if the right edges are perfectly aligned (eg. [q,r] with [k,l]). Finally, a string of the form 10...0 is added if the segment on the previous line extends to the right of the segment on the new line.

(3) When more than one segment on the new line overlaps with a single segment on the previous line, a new string of the form 10...03 is created for each successive pair of new segments. For example in figure 3.8 (a) segments [o,p] and [q,r] overlap with [k,l]. The new string 103 is formed with a number of zero's equal to (q-p) and represents the top of a potential hole contour.

(4) When a segment on the new line overlaps with more than one segment of the previous line, the corresponding strings must be merged. For example, in line #3 of figure 3.8 (a) we find segment [m,n] overlapping segments [g,h] and [i,j]. The end of the string representing [i,j] is connected to the beginning of the string representing [g,h] with a string on the form 2...2. The number of connecting 2's is given by (i-h+1). The merged string is then extended as in (2) above.

(5) If a segment on the previous line is not overlapped by any segment of the new line, a string of the form 0...0 is added to the end of its string to close the contour. In our example this occurs at line 4 for segment [m,n] and at line 5 for segment [s,t].

At the completion of the scan, each region contour is represented in a closed string of 4-connected chain codes. The starting point for contour traversal will be the lowest right extremity for object boundaries and the lowest left extremity for hole boundaries, taking the first element of each string at its beginning. The paths traced by this procedure are shown in figure 3.8 (c).

Method 2

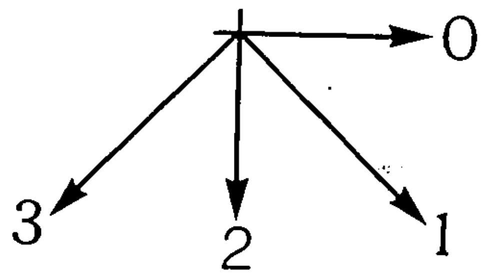
A strategy for constructing a chain code boundary description from a raster scan device has been proposed by Cederberg [3.11]. In this method, boundaries are represented by a number of coordinate pairs (max-points) from each of which originate two chains (R-chains) as shown in figure 3.9 (b). Each R-chain comprises elements of the 4-direction chain code of figure 3.9 (a). Since the raster scan moves from left to right and top to bottom, only four links are required to represent an 8-connected contour. A min-point is defined at the termination of two R-chains.

A max-point is detected by examination of the 3x3 neighbourhood of each pixel (a hardware encoder for assembling the eight neighbours of each pixel is described and is similar to the design of figure 3.5 (a)). Max-points occur for each of the neighbourhoods shown below (0 = background pixel, 1 = object pixel, x = don't care):

1	2	3	4	5	6
000 011 1xx	000 010 1x1	000 011 01x	000 010 110	000 010 011	000 011 001

7	8	9	10	11	12
000	000	000	000	xxx	xxx
010	010	010	011	x11	x1x
100	010	001	000	100	101

(a)



(b)

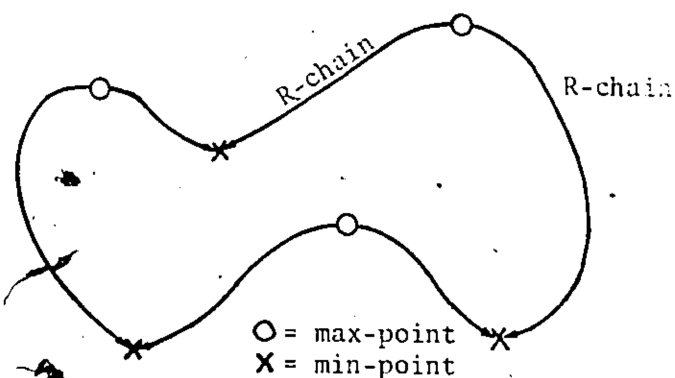


Figure 3.9 Sequential Boundary Extraction: Method 2.

(a) raster scan chain code directions,

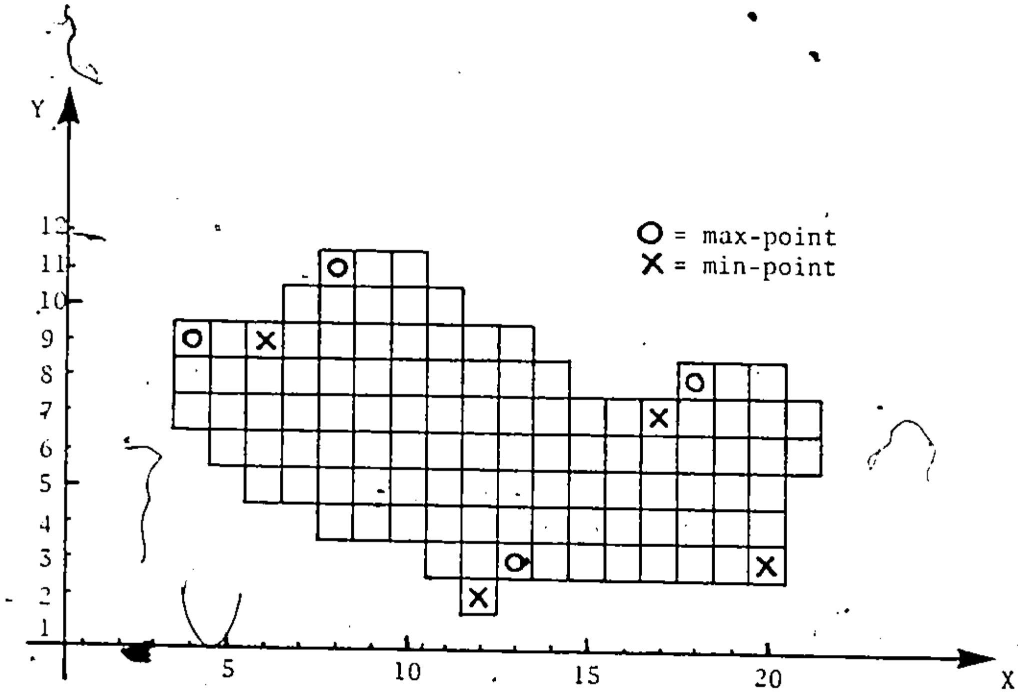
(b) object boundary encoded with max-points and R-chains.

The neighbourhoods for detection of min-points are obtained by the same set of templates, rotated by 180°. A second set of neighbourhood templates define the R-chain elements to be assigned to each border pixel:

13	15	17	19
00x 11x xxx	10x x1x xxx	01x 01x xxx	x01 x1x xxx
14	**16**	**18**	**20**
xxx 11x 00x	1xx 01x xxx	x10 x10 xxx	xx1 x10 xxx
link 0	link 1	link 2	link 3

The algorithm is demonstrated by way of the example in figure 3.10. As the input image is scanned in the raster, the appropriate neighbourhood patterns are recognized for each pixel; templates 1 to 12 will detect the max- and min-points as marked in figure 3.10 (a) and templates 13 to 20 indicate the chain direction for each border pixel. Chain elements are appended to the R-chains in the same sequence for each row of the image. Each time a match is found in the border following templates (13 to 20), that link is added to the current R-chain, the next link is added to the succeeding R-chain, and

(a)



(b)

Max-point (x,y)	Left R-Chain	Right R-Chain
(8,11)	33	001101100
(4,9)	2211010011	00
(18,8)	3	0012322
(13,3)	3	0000000

(c) 001101100 3 0012322 0000000 3 1100101122 00 33

(d) 007707700100765664444444533443433220011

Figure 3.10 Raster Scan Chain Code Example: Method 2.

(a) input image with max- and min-points indicated,
(b) R-Chains for each max-point,
(c) complete R-Chain for the contour,
(d) conversion to 8-direction chain code.

so on. As max-points are found, two new R-chains are inserted in the list of R-chains; when min-points are detected, the two connecting R-chains are removed from the list and no new links are added. The processing of the example procedes as follows:

line #	List of R-Chains				[max-point coordinates]			
					(left R-chain)	(right R-chain)		
	[8,11]		[4,9]		[18,8]		[13,3]	
11	( )	(00)						
10	(3)	(001)						
9	(33)	(00110)	( )	(00)				
8		(001101)	(2)		( )	(00)		
7		(001101100)	(22)		(3)	(001)		
6			(221)			(0012)		
5			(22110)			(00123)		
4			(22110100)			(001232)		
3			(221101001)			(0012322)	( )	(0000000)
2			(2211010011)				(3)	

The resulting R-chains for each max-point are listed in figure 3.10 (b). To represent the contour as a continuous clockwise or counterclockwise traversal, the end points of each R-chain must be connected. This is achieved when a min-point is found. For our example, a min-point is detected on line 9 at the end of the left R-chain of max-point (8,11) and at the end of the right R-chain of max-point (4,9). These two R-chains are then known to be connected.

Similar connections occur on lines 7, 3 and 2. The complete contour can be followed by reading one of the R-chains for the first max-point coordinate until it points to the end of the connecting R-chain which is then read in reverse order. This procedure is continued until the initial coordinate is found. The resulting contour traversal for our example is given in Figure 3.10 (c). R-chain representations are readily converted to Freeman 8-direction chain codes by substituting:

7 for 1, 6 for 2 and 5 for 3 in forward-read R-chains

and

4 for 0, 3 for 1, and 1 for 3 in reverse-read R-chains.

The Freeman code for our example is given in figure 3.10 (d).

Object/hole relationships are ordered in a hierarchy as shown in figure 3.11. This image contains a large object (A) with two holes (B,C). A third object (D) is situated in hole C and a fourth object (E) is outside of A. The interpretation of the contents of the image is changed at each max-point and ambiguities are resolved at min-points during the raster scan. For example, new objects are detected at max-points a and b and found to merge at min-point c. Max-points d and e indicate the start of potential holes in A, but this is not confirmed until min-points h and i are encountered.

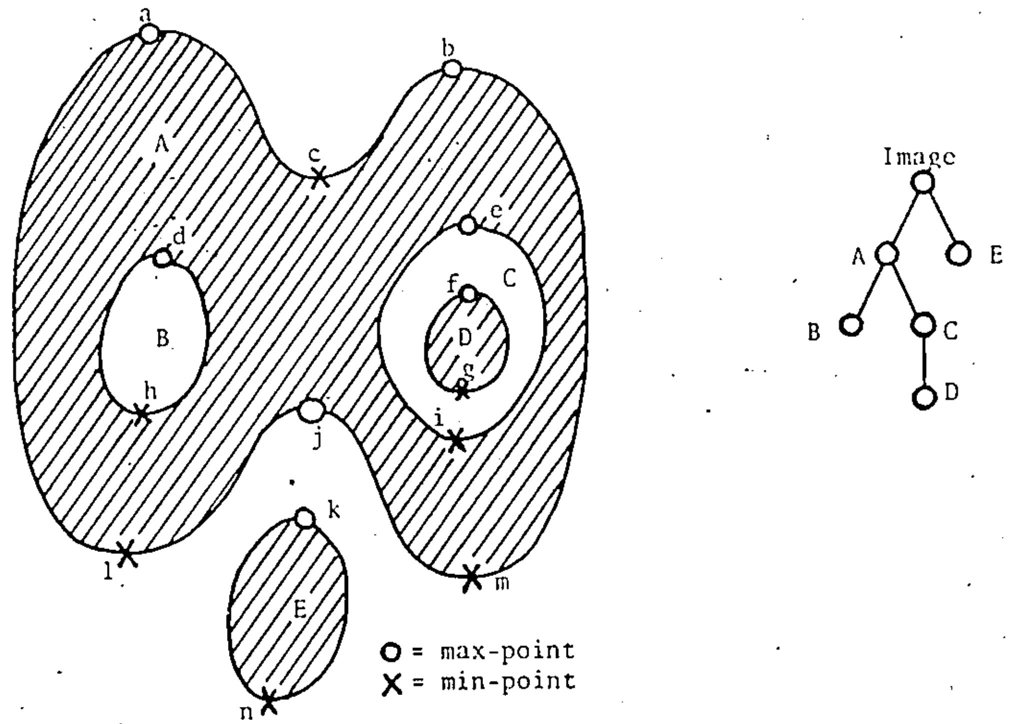


Figure 3.11 Interpretation of Image Contents: Method 2.

Method 3

A sequential tracking algorithm is described by Agrawala and Kulkarni [3.12]. Although the technique has been generalized to deal with multicolour images in [3.13], we describe here the method as applied only to binary images.

The Agrawala and Kulkarni procedure processes the edge point data (run-length codes) as they are sequentially generated during the raster scan. The edge points from two successive lines are stored and overlapping segments are detected to establish the connections required to generate the boundaries for each distinct region (object or hole) in the image.

The algorithm utilizes two data structures as shown in figure 3.12. The edge point data from the new line (K) and the old line (K-1) are stored in two rows of the Scan Line Array. For each edge point in the new line, an object number OL1(K), OR1(K), OL2(K), OR2(K) ... OLn(K), ORn(K) is assigned to the corresponding element in the structure as the boundary to which each point belongs is determined. A second structure, the object array, contains the following fields for each object number:

type - this field indicates the type of boundary, either object or hole.

parent - for hole boundaries, this field indicates the object in which the hole exists.

status - at any point in the frame, the current status of each boundary is labelled as continuing (con), merged, or terminated (term). A merged boundary is one which has been found to be connected to another object or hole.

Additional fields (feature 1, feature 2 ...) are added as needed to keep track of a variety of features during scanning. These may include area, perimeter and the like. Since these do not enter into the boundary extraction process, we do not consider them further.

K-1	OLD LINE	Edge Point	XL1(K-1)	XR1(K-1)	XL2(K-1)	XR2(K-1)
		Object No.	OL1(K-1)	OR1(K-1)	OL2(K-1)	OR2(K-1)
K	NEW LINE	Edge Point	XL1(K)	XR1(K)	XL2(K)	XR2(K)
		Object No.	OL1(K)	OR1(K)	OL2(K)	OR2(K)

Scan Line Array

Object #	Type	Parent	Status	Feature 1	Feature 2 ...
A					
B					
C					
D					
.					
.					
.					

Object Array

Figure 3.12 Sequential Boundary Extraction: Method 3.

Connections between the edge points of segments on successive lines are determined from overlap conditions. The segments are considered as in figure 3.13.

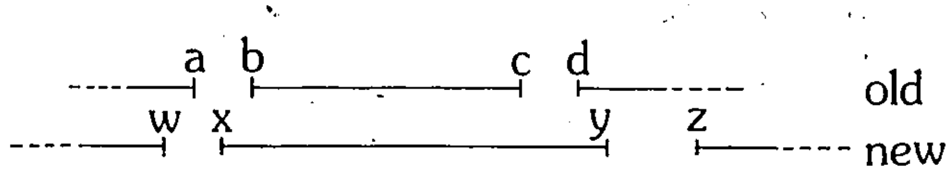


Figure 3.13 Segment Overlap Determination: Method 3.

The current segment on the old line is designated as [b,c], corresponding to [XLn(K-1), XRn(K-1)] in the Scan Line Array. The right edge point of the previous segment is labelled "a" (XRn-1(K-1)) and the left edge point of the following segment is labelled "d" (XLn+1(K-1)). Similarly, labels w, x, y, z are assigned to edge points on the new line. The appropriate boundary connections between the old and new lines are detected by comparing edge point coordinates according to the following conditions:

(i) continuation of left boundary (connect b to x)

$(w<b)$ and $(b<y)$ and $(a<x)$ and $(x<c)$

(ii) continuation of right boundary (connect y to c)

$(x<c)$ and $(c<z)$ and $(b<y)$ and $(y<d)$

When conditions (i) or (ii) are not true, it must be that a boundary on the old line has terminated or that additional boundaries on the new line have occurred. These are detected by:

(iii) creation of hole boundary (connect y to z)

$z<c$

(iv) creation of object boundary (connect y to x)

$y<d$

(v) termination of hole boundary (connect d to c)

$b<y$

(vi) termination of object boundary (connect b to c)

$c<x$

Creation or termination of either object or hole boundaries necessitate a horizontal connection between edge points as indicated above. Whenever an object or hole is created, its type is stored in the Object Array, along with a new object number and a status of 'continuing'. For hole boundaries, the object number of its parent is recorded. Terminating regions are marked by a 'terminating' [illegible] in the object array. However, if the object numbers of the edge points being connected in a termination are different, then a 'merge' condition has been detected. In this situation, the status of one boundary is set to 'merged' in the Object Array while the other remains as 'continuing'. A merged boundary need not be considered in subsequent processing. Examples of the three cases of merge conditions are given in figure 3.14.

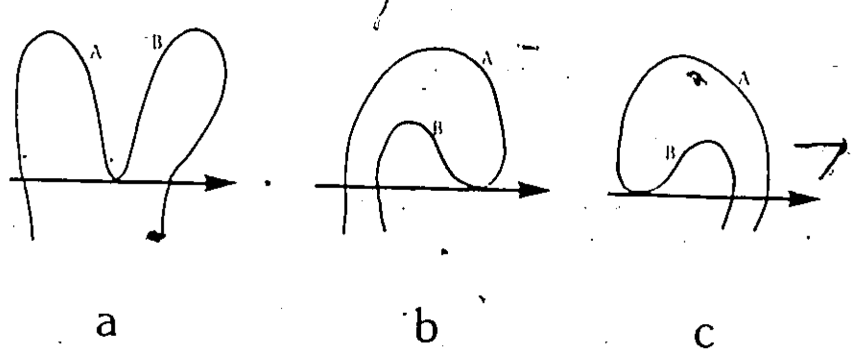


Figure 3.14 Three Cases of Merge: Method 3.

(a) Boundaries A and B are both objects,

(b) Boundary B is merged and A continues,

(c) Boundary A is merged and B continues.

To demonstrate the operation of the algorithm, consider the example shown in figure 3.15 (taken from [3.12]). The input image consists of a single object (body) spanning lines 1 through 9 and contains one hole which spans lines 3 to 5. The edge point coordinates for each line are indicated on the diagram. To begin, the Scan Line and Object Arrays are initialized as follows:

0	Edge Pt.	0	0	0	0	0	0
	Object #	0	0	0	0	0	0
1	Edge Pt.	10	25	0	0	0	0
	Object #	0	0	0	0	0	0

Object #	Type	Parent	Status
0	0	0	0

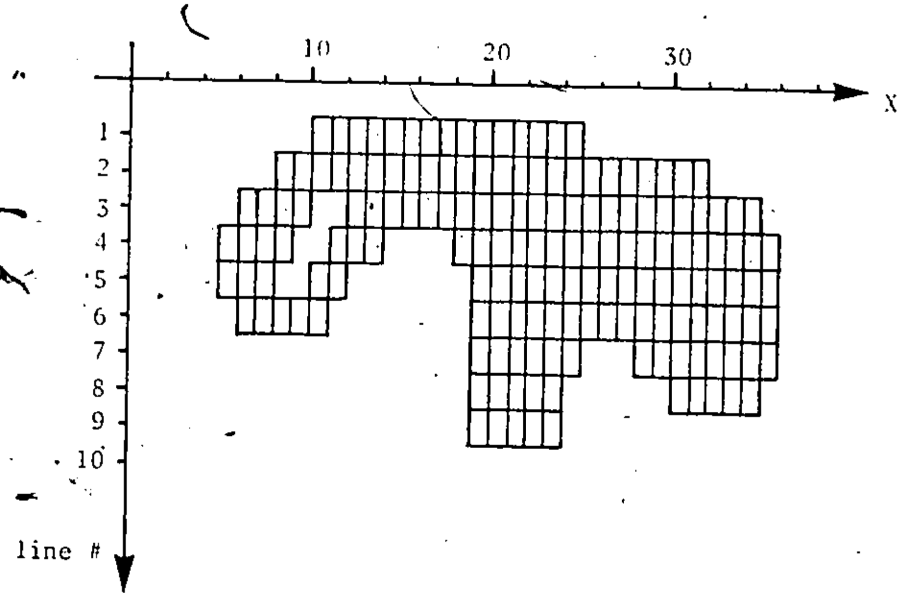


Figure 3.15 Processing Example: Method 3.

At the end of the first line, the beginning of the object is detected and assigned object # "A". The type is set to "Body" and status to "continuing":

								Object #	Type	Parent	Status
0	Edge Pt.	0	0	0	0	0	0				
	Object #	0	0	0	0	0	0				
1	Edge Pt.	10	25	0	0	0	0	A	Body	0	Con
	Object #	A	A	0	0	0	0				

At the end of the second line, the left and right boundaries are found to continue:

								Object #	Type	Parent	Status
1	Edge Pt.	10	25	0	0	0	0				
	Object #	A	A	0	0	0	0				
2	Edge Pt.	8	32	0	0	0	0	A	Body	0	Con
	Object #	A	A	0	0	0	0				

On completion of line 3, the beginning of a potential hole boundary is detected. A new row is added to the Object Array and its 'parent' field is set to the enclosing object "A". The hole is assigned object "B":

								Object #	Type	Parent	Status
2	Edge Pt.	8	32	0	0	0	0				
	Object #	A	A	0	0	0	0				
3	Edge Pt.	6	10	12	35	0	0	A	Body	0	Con
	Object #	A	B	B	A	0	0	B	Hole	A	Con

A second potential hole is found at the end of line 4 and is entered in the Object Array as object "C". Its parent is also "A":

								Object #	Type	Parent	Status
3	Edge Pt.	6	10	12	35	0	0				
	Object #	A	B	B	A	0	0				
4	Edge Pt.	5	9	11	14	18	36	A	Body	0	Con
	Object #	A	B	B	C	C	A	B	Hole	A	Con
								C	Hole	A	Con

The processing of line 5 does not result in the creation of any new objects or holes and no update is necessary on the Object Array. The Scan Line Array appears as:

4	Edge Pt.	5	9	11	14	18	36
	Object #	A	B	B	C	C	A
5	Edge Pt.	5	8	10	12	19	36
	Object #	A	B	B	C	C	A

At the end of line 6, hole "B" is completed. Its status in the Object Array is changed to "terminated":

5	Edge Pt.	5	8	10	12	19	36
	Object #	A	B	B	C	C	A
6	Edge Pt.	6	11	19	36	0	0
	Object #	A	C	C	A	0	0

Object #	Type	Parent	Status
A	Body	0	Con
B	Hole	A	Term
C	Hole	A	Con

The second potential hole (C) is found not to be a hole at the end of line 7. Its status in the Object Array is now changed to "merged". A third potential hole "D" also appears:

6	Edge Pt.	6	11	19	36	0	0
	Object #	A	C	C	A	0	0
7	Edge Pt.	19	25	28	36	0	0
	Object #	A	D	D	A	0	0

Object #	Type	Parent	Status
A	Body	0	Con
B	Hole	A	Term
C	Hole	A	Merged
D	Hole	A	Con

The processing of lines 8 and 9 are similar to the above situations. No change takes place in the Object Array for line 8 and in line 9, potential hole "D" is changed to a status of 'merged'. No segments are detected at line 10, indicating the termination of object "A". The resulting arrays are then:

9	Edge Pt.	19	24	0	0	0	0
	Object #	A	A	0	0	0	0
10	Edge Pt.	0	0	0	0	0	0
	Object #	0	0	0	0	0	0

Object #	Type	Parent	Status
A	Body	0	Term
B	Hole	A	Term
C	Hole	A	Merged
D	Hole	A	Merged

The contents of the frame can be interpreted from the Object Array by examination of the 'status' fields. At the end of scanning the number of regions (objects plus holes) can be determined by counting the number of objects which have terminated. Object types are available from the 'type' field and object/hole relationships can be read from the 'parent' field. The contours of object regions are linked counterclockwise and those of holes are connected in clockwise sequence as shown in figure 3.16.

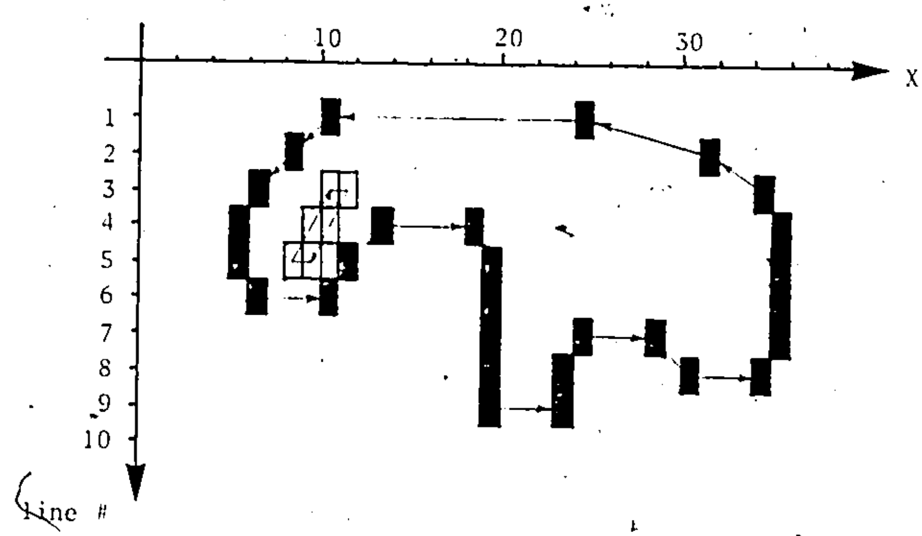


Figure 3.16 Boundary Points Extracted: Method 3.

3.2 An Improved Sequential Algorithm

In practice, binary images of workpieces in visual inspection or manipulator guidance applications often consist of a large number of objects each of which may contain many holes. Consider, for instance the inspection of the printed circuit traces of figure 3.17.

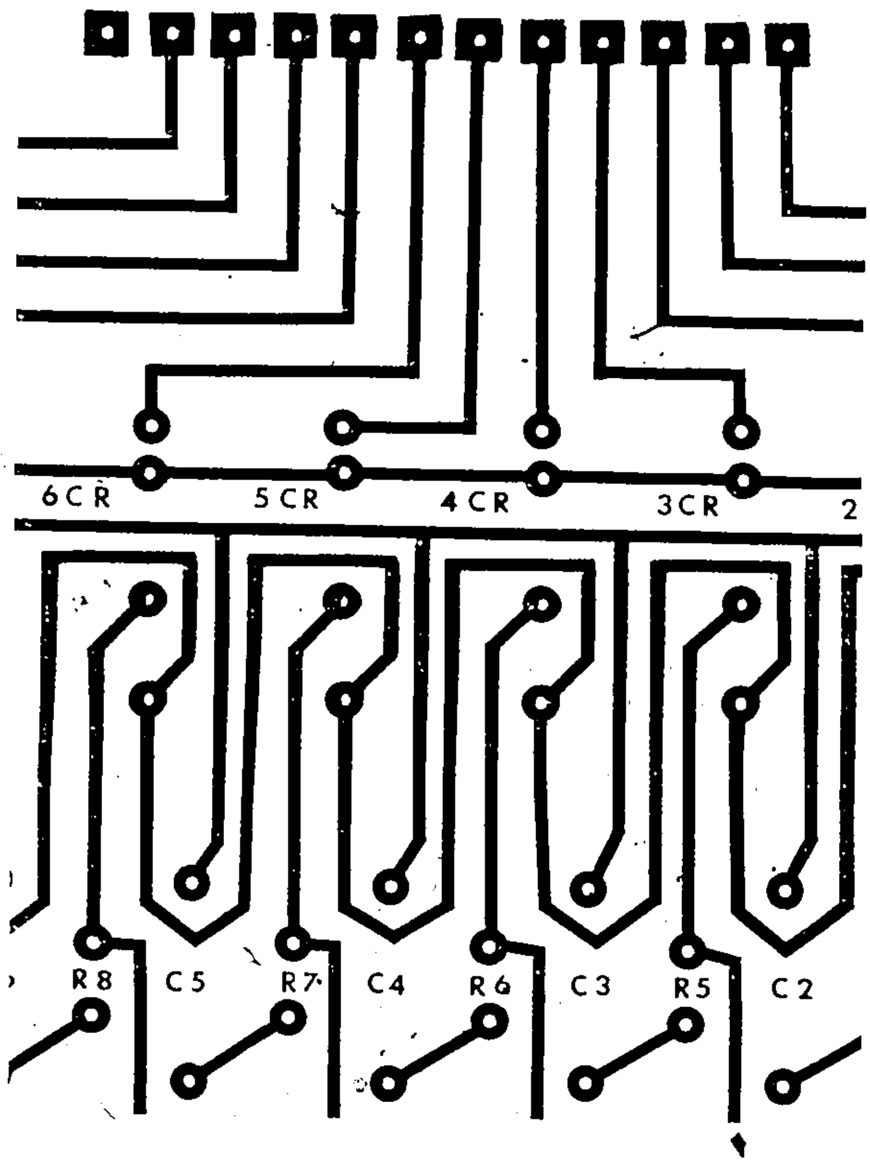


Figure 3.17 Binary Image of Printed Circuit Board Traces.

A large number of the traces consist of long, straight sections on the contour and embody several holes for component lead insertion. Defects may include broken or missing traces, inadvertant bridges between traces, and non-existant or plugged holes. In addition, component labels (5CR, 4CR ... in the example) are frequently applied to the board and these must be distinguished from actual traces. In a production environment inspection speed is very often of prime importance. It is therefore desirable to select a contour extraction algorithm which will encode the boundaries as quickly as possible and with the minimum number of points. Subsequent processing can be greatly enhanced if there is a data structure for representing the type of each contour (object or hole) and the relation of each hole to objects.

An extensive variety of procedures have been established for processing chain code representations as would be generated by Methods 1 and 2 [3.14] [3.15]. However, chain coded boundaries have the disadvantage of containing many redundant links for representing straight sections of a contour. The encoding technique of Method 3 will use fewer boundary points for horizontally aligned traces, but may contain collinear points for regions in other orientations (see figure 3.16).

In Method 1, no relationship between objects and holes are maintained during the boundary extraction process. The type of contour is determined only after a traversal of each chain; object boundaries are linked counterclockwise, and hole boundaries clockwise. With Method 2, object/hole relationships are retained from the max-and min-point information, but additional processing is required to join pairs of R-chains to form closed contours.

The algorithm of Method 3 does not suffer from these drawbacks. At the completion of a frame scan, object/hole relationships are available by way of the 'parent' field in the Object Array structure. Object contours are closed in the counterclockwise sense and hole contours clockwise, as each region terminates. A disadvantage of this technique is that execution time is not completely independent of the number of regions in the frame. The problem arises in situations similar to that shown in figure 3.18. Previous to line j, traces A and B are treated as separate objects, each with a number of holes. When it is determined that A and B are in fact connected at line j, the 'parent' field of each of the holes in B must be changed to point to object A. The time required for processing line j is therefore dependent on the number of holes in B. Given a fixed maximum time to perform boundary extraction (say, during the horizontal retrace of the raster), we must be able to specify the minimum number of edge points for which processing is guaranteed.

The remainder of this section describes an improved algorithm and data structures for the sequential extraction of region boundaries from a raster scan. The problem of figure 3.18 is avoided by the use of a dynamic data structure for organizing object and hole boundaries at any line in the raster. Updates to the structure are accomplished by simply changing a fixed number of pointers. Contours are encoded as a set of vertex points arranged in counterclockwise order for object borders and clockwise for holes (the storage economies achieved with vertex point representations are demonstrated in Chapter 5).

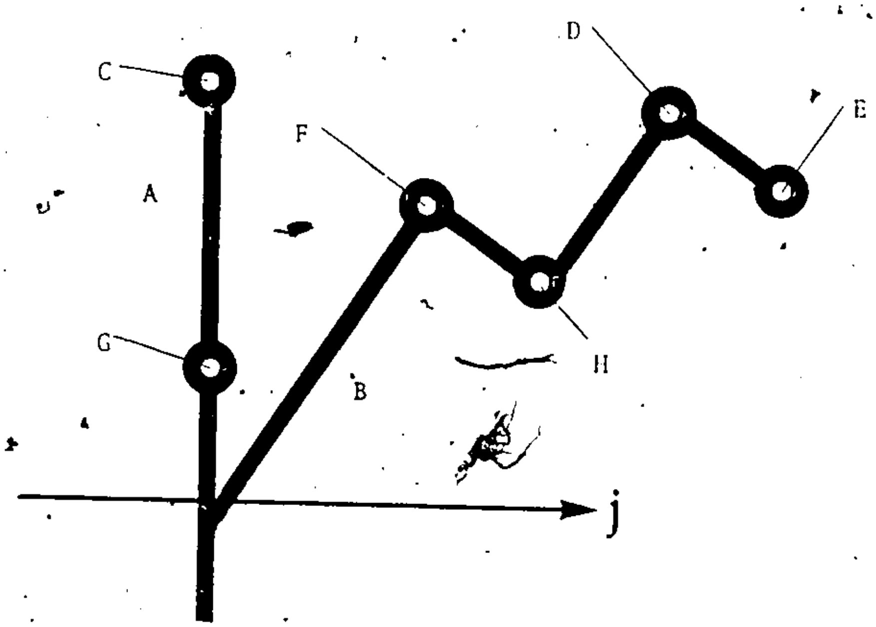


Figure 3.18 Example of Objects for which Processing Time is Dependent on the Number of Holes.

Edge point connections between segments on successive raster lines are determined with comparisons similar to those of Method 3, however fewer calculations are required. An additional data structure is described which offers an improved representation of object/hole relationships. There is no restriction on complexity: objects may intertwine, intersect frame edges or appear as 'islands' within the holes of larger objects and this may occur to any depth. The high-level language implementation of the algorithm occupies less than 2 k bytes of memory and is shown in Appendix B.

3.2.1 Edge Point Connections

The algorithm accepts as input the set of segments which have been detected for a single new line in the frame, having already processed segments from the preceeding line. The basic task is then to decide the connection to be made between edge points for both lines. This can be achieved by considering the segments from the previous line sequentially from left to right (as they occurred in the raster) and those from the new line in the same order. We refer to the segments shown in figure 3.19: for the previous line (y_{k-1}), the segment currently being considered is designated by its left-edge point (*a*), its right-edge point (*b*), and the left-edge point of the following segment (*c*). Similarly, the current segment on the new line (y_k) is specified by the edge points *d*, *e* and *f*. Since all connections for *a*, *b*, *c*, *d*, *e* and *f* are determined before advancing to subsequent segments on each line, there is never any need to consider the edge points of segments previous to [*a*, *b*] or [*d*, *e*]. If a segment

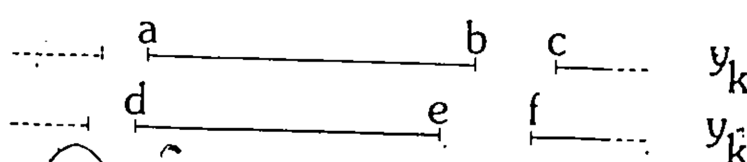


Figure 3.19 Current Segments on Previous Line [*a*, *b*] and New Line [*d*, *e*].

is the last on either line, then c (or f) is set to a value which is greater than the largest (right-most) coordinate in a line. If the previous line contains no segments, then a, b and c are each set to the maximum right value. For a line with 2^n pixels, therefore, valid edge points may occur from 0 to 2^n-2 with the value 2^n-1 reserved for this case.

The appropriate connections for a, b, c, d, e and f are decided as follows:

Condition	Interpretation
$e < a$	segment $[d,e]$ is the beginning of a new object in the frame. Connect e to d.
$d > b$	segment $[a,b]$ is the termination of an existing object in the frame. Connect a to b.
neither of the above	segment $[d,e]$ overlaps with segment $[a,b]$. Connect a to d.
$f < b$	segment $[a,b]$ has split and $[e,f]$ could be the beginning of a hole. Connect e to f.
$e > c$	segment $[d,e]$ is merging with segment $[a,b]$ and the following segment, and $[b,c]$ could be the termination of a hole. Connect c to b.
neither of the above	connect e to b.

Connections are made by establishing the *link* field between coordinate points stored in the *vertex_list* data structure shown in figure 3.20. The values of *a*, *b* and *c* are obtained from the *x* field and are accessed via corresponding pointers *a_ptr*, *b_ptr*, and *c_ptr*. To connect *a* to *b* for example, the *link* field of the location where *a* is stored is set to the location where *b* is stored.

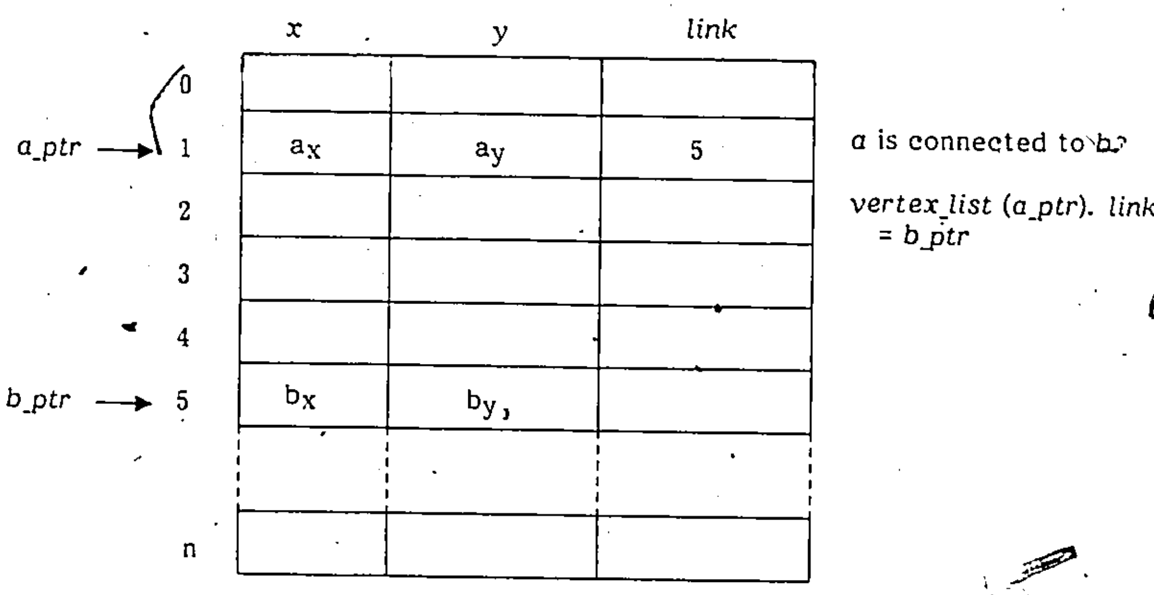


Figure 3.20 *Vertex_list* Data Structure.

The *line_buffer* structure is used to hold the edge points for the line in the frame which is about to be analysed (figure 3.21). It consists of an array whose maximum required number of elements is equal to the number of pixels in a single line and a variable (y_k) to hold the corresponding line number within the frame. Since edge points always exist as pairs, j is even.

The segment currently being analysed is defined by the pointers *d_ptr* and *e_ptr* and has edge point coordinates (x,y) of:

$$(line\_buffer(d\_ptr),\ y_k) \text{ and } (line\_buffer(e\_ptr),\ y_k)$$

The *f_ptr* pointer defines the left edge point of the next segment following the current segment.

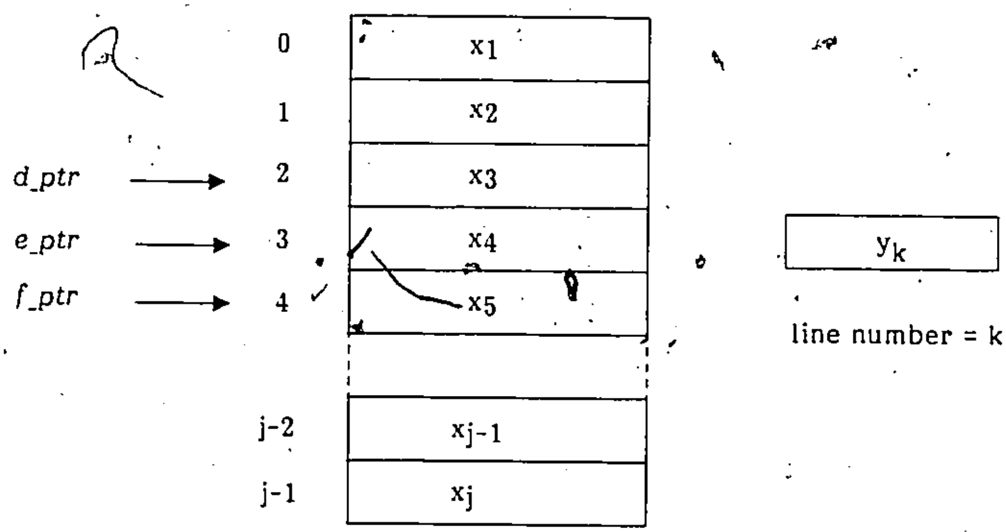


Figure 3.21 *Line_buffer* Structure.

The highest level of the algorithm can now be written:

```
Get_vertex_points:  procedure;
    Initialize;
    Do while there are still more lines in the frame;
        Reset a, b, c to the beginning of the segments of the
        previous line;
        Read segments from new line;
        Reset d, e, f to the beginning of the new segments;

           Do while there are still segments on previous line
                    or new line;
              if  e < a  then   create_a_new_object;
                         else
              if  d > b  then   terminate_an_object;
                         else
                           begin;
                             connect a to d;
                             do while (f < b) or (e > c)
                                if (f < b) then split_an_object;
                                           else merge_two_objects;

                             end;
                             connect e to b;
                             advance to next segment on previous line;
                             advance to next segment on new line;

                       end;
           end;
    end;
end Get_vertex_points;
```

```
create_a_new object:  procedure;
        connect e to d;
        update the active list;
        advance to next segment on new line;
end;

terminate_an_object:  procedure;
        connect a to b;
        update the active list;
        advance to next segment on previous line;
end;

split_an_object:  procedure;
        connect e to f;
        advance to next segment on new line;
        update the active list;
end;

merge_two_objects:  procedure;
        connect c to b;
        update the active list;
        advance to next segment on previous line;
end;
```

3.2.2 Active Objects Data Structure

The *active_list* data structure, shown in figure 3.22, dynamically represents the objects in the frame which are currently active and the relationships among them. There are five link fields in each node:

next — is the pointer to the next node in the active list. The left to right order preserves the order in which portions of objects appeared on the previous line.

hole_ptr — points to the node representing the first hole in the hole list of an active object. In hole nodes, *hole_ptr* points to subsequent holes in the list.

last_hole_ptr — is used to point to the last hole node in the hole list for an active object. It is used to facilitate the merging of two hole lists.

forward, back — are used to connect nodes which are known to be of the same object region. When a node is split, its *forward* pointer is set to the new node created and the new node's *back* pointer is set to the original node.
The *forward* and *back* links form a doubly-linked circular list of nodes within the list defined by the *next* links.

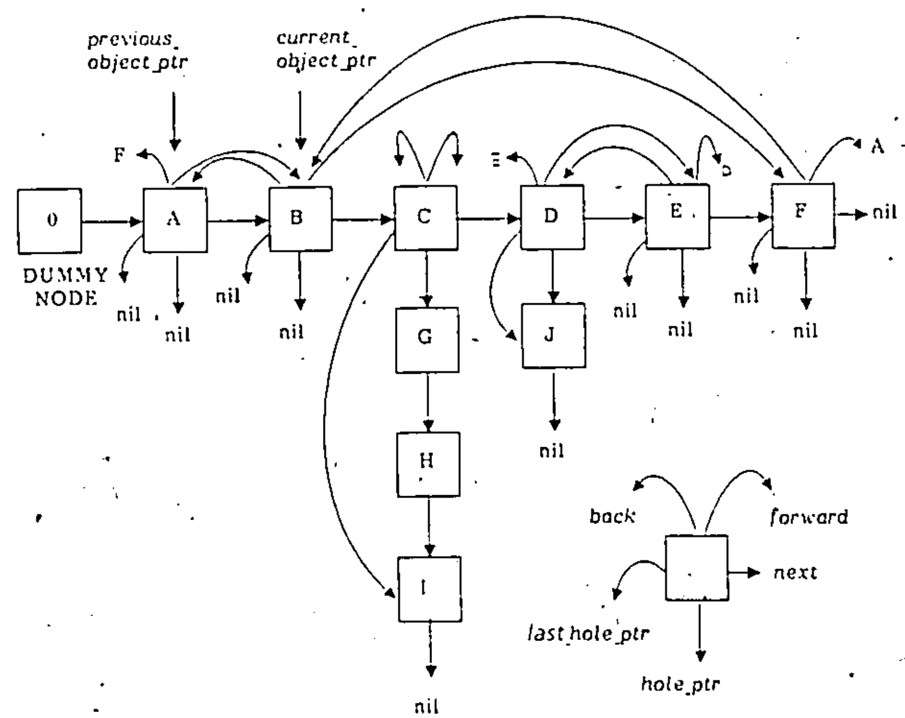


Figure 3.22 *Active_list* Data Structure.

In addition to the links, other fields are defined for each node:

left, right — are pointers into the *vertex_list* to the co-ordinates of the left- and right-edge points of that portion of an active object which occurred on the previous line. When object nodes are added to the list of complete objects, the *left* field is used as the starting point for the contour.

delta_XL, delta_XR the change in the x-coordinate of the left- and right-edge point between two successive lines is retained in these fields. When these changes are determined for the current line, they are compared to the previous *delta_XL* and *delta_XR*. If either is identical, then a col-linear point has been found and need not be entered in the *vertex_list*.

The pointers *current_object_ptr* and *previous_object_ptr* keep track of the position of the nodes in *active_list* which represent the current object and previous object respectively.

In the example shown (figure 3.22), there are six objects currently active (A,B,C,D,E,F). Three holes have been detected for object C (G,H,I) and one hole in object D (J). Hole lists are linked via the *hole_ptr* field. Object A has been split, resulting in object B which in turn has been split to form object F. Similarly object D has split, giving object E. Objects B and F represent potential holes in object A, as does object E within D. Furthermore, objects C and D are located between objects B and F.

The first node in the *active_list* (0) is a dummy node which allows simplified insertion and deletion at the beginning of the list. The dummy node's *next* field is considered to point to the first active object. The procedures for manipulating the *active_list* data structure are outlined in the next section.

3.2.3 Operations on the *Active list*

Inserting a node

This routine inserts a node in the *active_list* to the right of the node pointed to by "PTR". The parameter "*new_object_ptr*" is a pointer to the next available unused node in the cache.

```
insert_node_to_right:  procedure (PTR);
    active_list (new_object_ptr). next = active_list (PTR). next;
    active_list (PTR). next = new_object_ptr;
    initialize the new node fields;
end;
```

Deleting a node

This routine deletes a node from the *active_list*. 'PTR' points to the node whose *'next'* field points to the node to be deleted.

```
delete_next_node:  procedure (PTR);
    active_list (PTR). next = active_list (active_list (PTR).
    next). next;
end;
```

Concatenating two hole lists

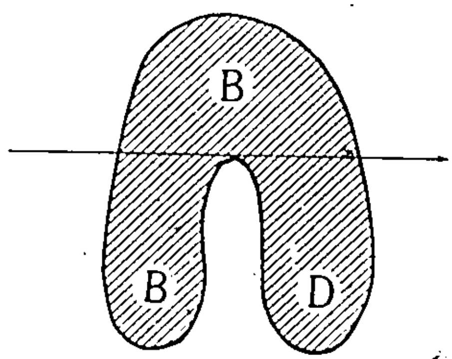
This routine joins a hole list defined by FIRST_PTR, LAST_PTR to the hole list of the object node pointed to by NODE_PTR.

```
join_to_hole_list:  procedure (NODE_PTR, FIRST_PTR, LAST_PTR);
     active_list (active_list (NODE_PTR). last_hole).
     hole_ptr = FIRST_PTR;
     active_list (NODE_PTR). last_hole = LAST_PTR;
end;
```

Updating for Split

When an object region splits (f < b), a node is inserted into the *active_list* immediately to the right of the current node. *Forward* and *back* pointers are set (as shown in figure 3.23) to indicate that the two nodes are known to be part of the same object region. It is unknown at this time if the split is the beginning of a hole region or not. Until such time as this is determined, the two nodes are treated as separate objects.

```
update_for_split:  procedure;
     insert_node_to_right (current_object_ptr);
     set forward and back pointers;
     previous_object_ptr = current_object_ptr;
     current_object_ptr = active_list (current_object_ptr). next;
end;
```



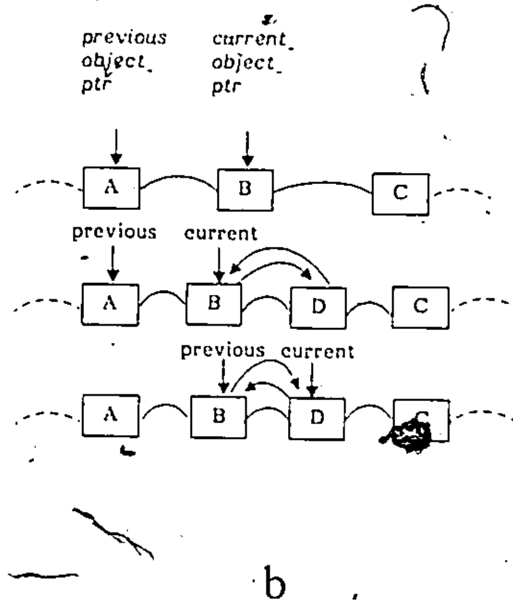


Figure 3.23 Split.

(a) object splits, (b) *active_list* updates

Updating for Merge

A merge operation is always done with the current object node and its immediate right-neighbour in the *active_list*. There are four cases to be considered for a merge and these are illustrated in figure 3.24.

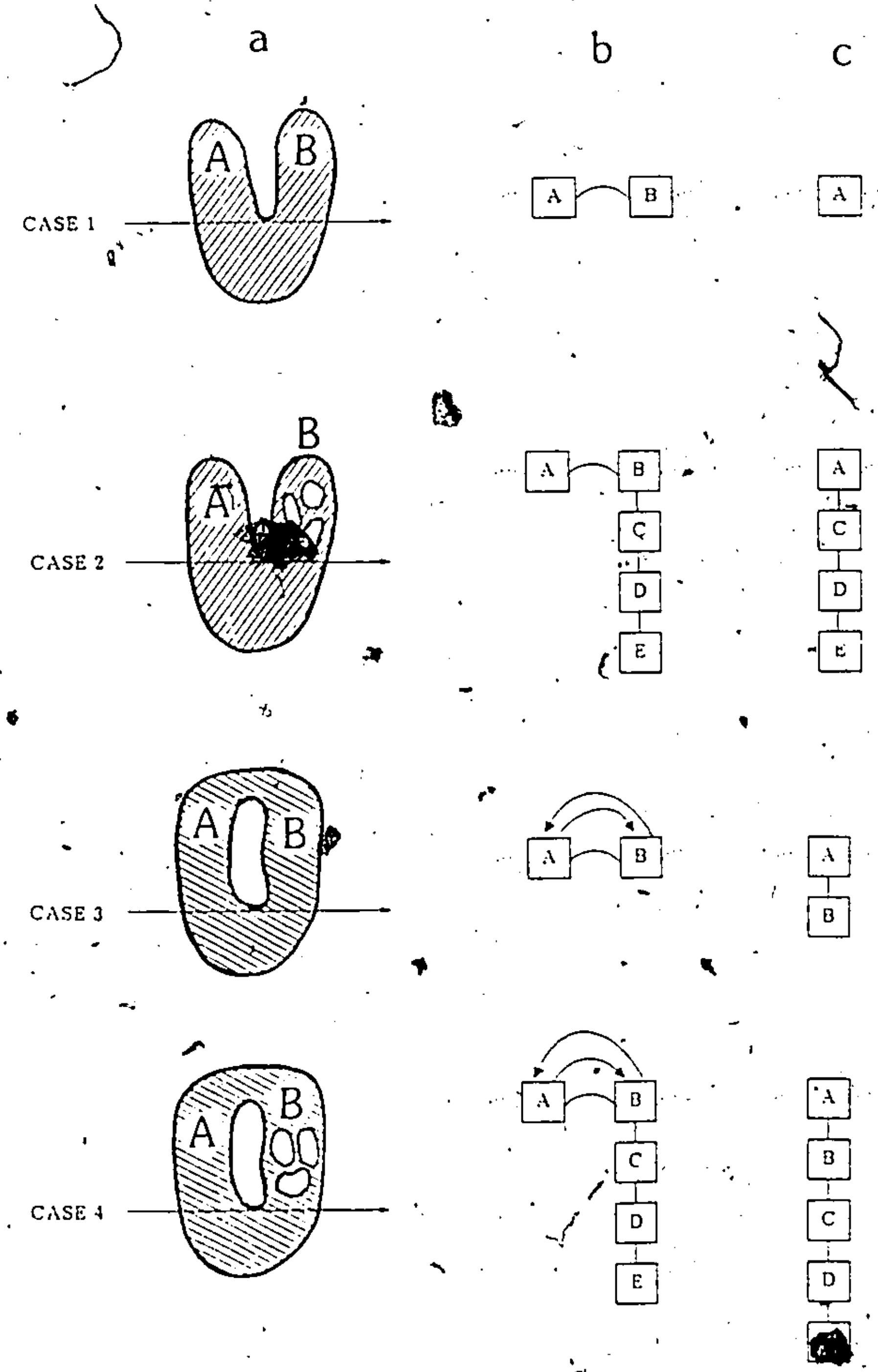


Figure 3.24 Four Cases of Merge.
(a) sample objects,
(b) *active_list* before Merge,
(c) *active_list* after Merge.

Case 1 the right node was a distinct object region.

Case 2 the right node was a distinct object region with holes (C,D,E).

Case 3 the right node was a result of a split from the left node. Therefore the right node represents a hole region contained in the left node.

Case 4 the right node was a result of a split from the left node, but it also contains a hole list (C,D,E).

```
update_for_merge:  procedure;
     if next node was split from current node (case 3 or case 4)
          then  join next node and its hole list to the hole list
                of the current node;
          else  if the next node has a hole list (case 2)
                      then join next node's hole list to hole list
                           of current node;
     remove next node from forward/back linked chain;
     set current node's right field to next node's right field;
     delete_next_node (current_object_ptr);
end;
```

Updating for creation of an object node

When an object is first encountered in the frame, a node in the *active_list* is created. The condition for detecting the start of a new object (e < a) implies that the new object lies to the left of the current object; that is, it is to be inserted immediately to the right of the node which is immediately to the left of the current object. In this manner, the same routine can be utilized for right or left insertion. Once the node is inserted, the pointer to the previous object must be advanced (figure 3.25).

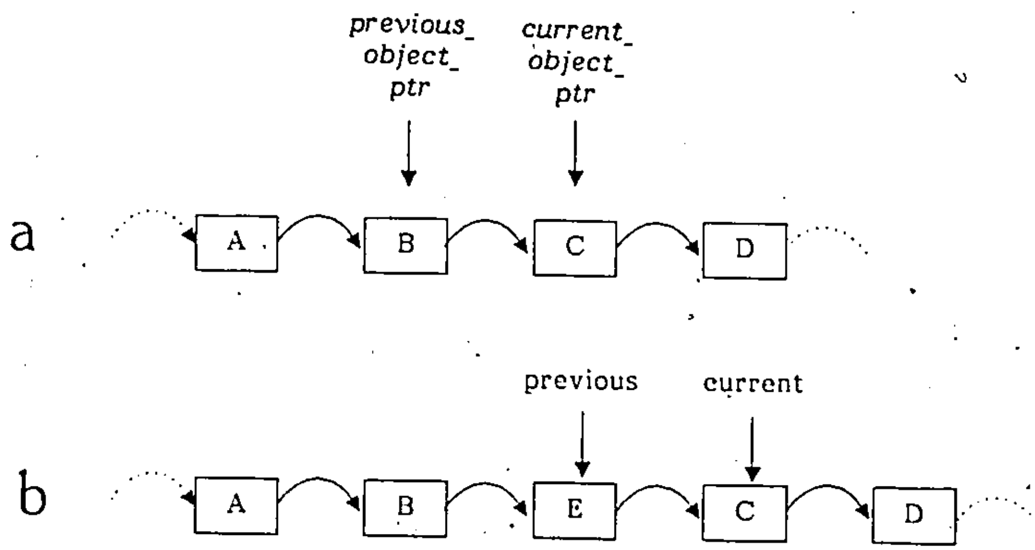


Figure 3.25 Creation.

(a) *Active_list* before creation,
(b) *Active_list* after creating a new node (E) and advancing the previous object pointer.

update for create: *procedure*;

insert_node_to_right (*previous_object_ptr*);

previous_object_ptr = *active_list* (*previous_object_ptr*).

next;

end;

Updating for termination of a node

The condition d > b indicates the termination of an active node. In general, there are two cases to be considered:

Case 1 (figure 3.26 a)

Although a node has terminated, the object is still active. This is detected by checking if the terminating node has been split or is the result of a split; that is, by checking the *forward* and *back* fields. If this is the case, any hole list must also be appended to a continuing node.

Case 2 (figure 3.26 b)

The object region is terminating. The *active_list* node representing the object should be removed from the *active list* and an entry is made in the *complete_list* structure pointing to this node (see section 3.2.4).

```
update_for_terminate:  procedure;
     remove the node from the forward/back linked chain;
     if the object region is complete (Case 2)
     then put a pointer to the node in the complete_list array;
     else if the node contains any holes
          then append its hole list to the hole list of the
               continuing node;
     delete_next_node (previous_object_ptr);
     current object ptr = active_list (previous_object_ptr). next;
end;
```

3.2.4 Complete Objects Data Structure

The *complete_list* is an array which holds a list of pointers to *active_list* nodes which were found to represent complete objects. Each object node in turn may point to any number of hole nodes via the *hole_ptr* field. An example is shown in figure 3.27. The starting point (SP) is the final value set for the *left* field in the *active list* node.

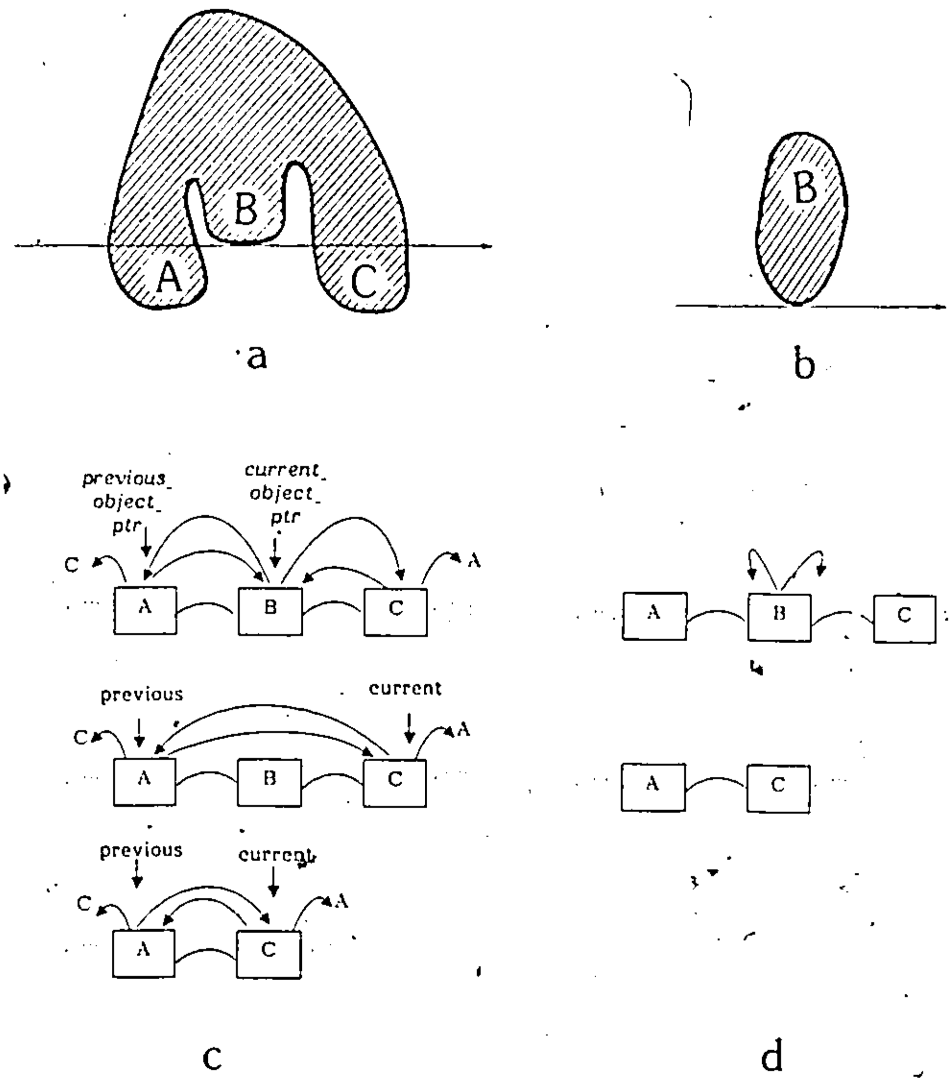


Figure 3.26 Two Cases of Termination.

(a) object continues in other nodes,
(b) object is complete,
(c) *active_list* update for (a),
(d) *active_list* update for (b).

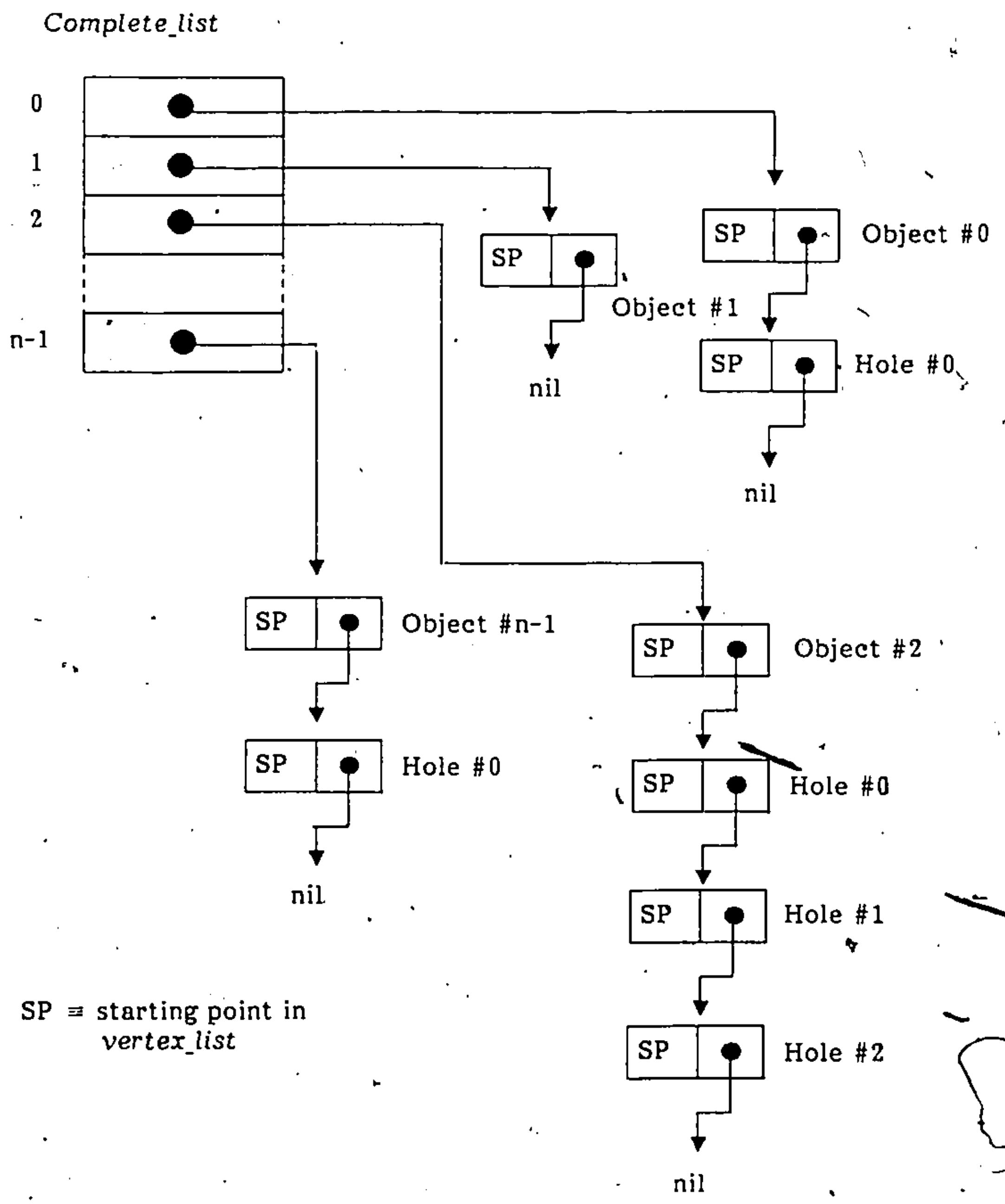


Figure 3.27 Data Structure Used to Represent Complete Objects in a Frame.

3.2.5 Example

To illustrate the operation of the algorithm, consider the *active_list* data structure as it would exist at each of the lines indicated in the frame of figure 3.28. At the end of processing, the *complete_list* array contains two entries, one for each object region, and the second object node contains a pointer to a third node representing its hole. This is shown in figure 3.29. Each of the three nodes contains a starting pointer (SP) into the corresponding *vertex_list*.

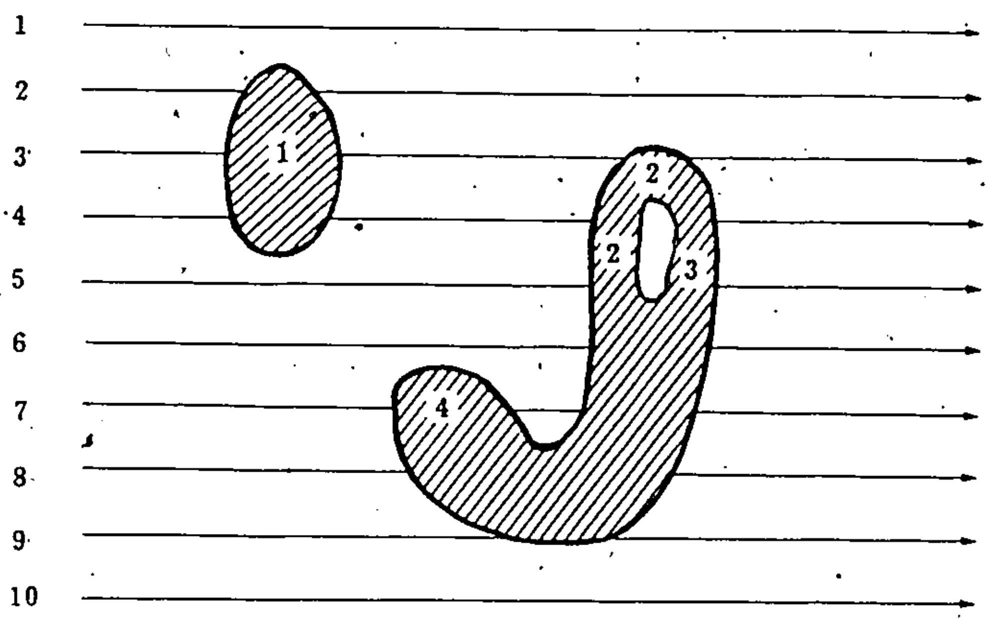
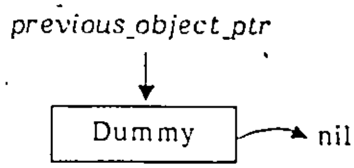


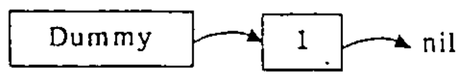
Figure 3.28 Sample 10 Line Frame.

line #1



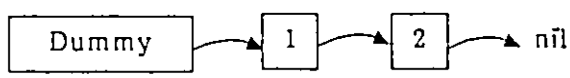
The *active_list* is initialized as shown. Since line #1 contains no segments, there is no change to the structure.
Current_object_ptr = nil.

line #2



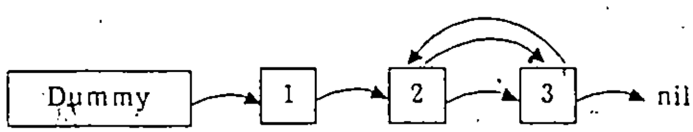
Object 1 is created.

line #3



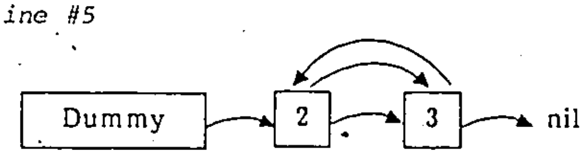
Object 1 is updated.
Object 2 is created.

line #4



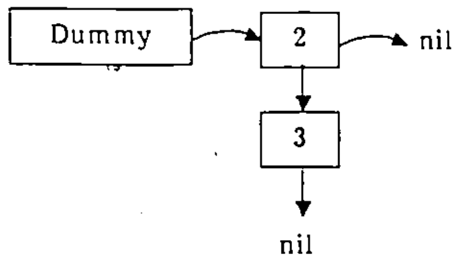
Object 1 is updated.
Object 2 is split giving object 3 (a potential hole).

line #5



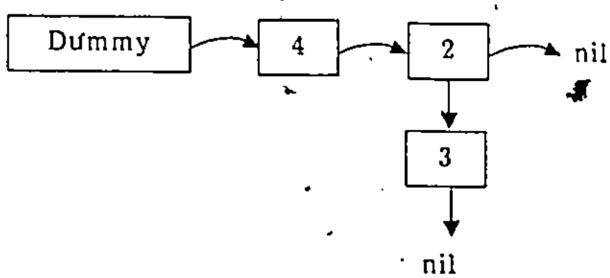
Object 1 is terminated and added to the *complete_list*.
Objects 2, 3 are updated.

line #6



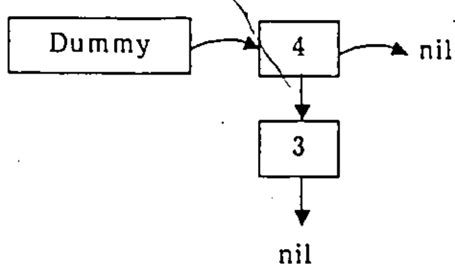
Objects 2, 3 merge. Since object 3 was a potential hole in object 2, it now becomes a hole node in object 2.

line #7



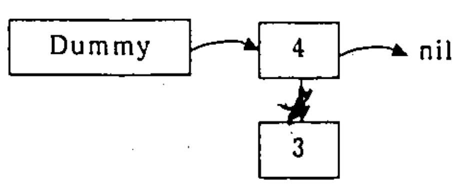
Object 4 is created.
Object 2 is updated.

line #8



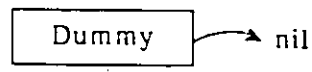
Objects 4, 2 merge. The hole list of object 2 is added to that of object 4 and object 2 is terminated and discarded.

line #9



Object 4 is updated.

line #10



Object 4 is terminated and added to the *complete_list*. The *complete_list* now contains pointers to nodes 1 and 4. Node 4 contains a pointer to its hole.

Additional examples are demonstrated in Chapter 5.

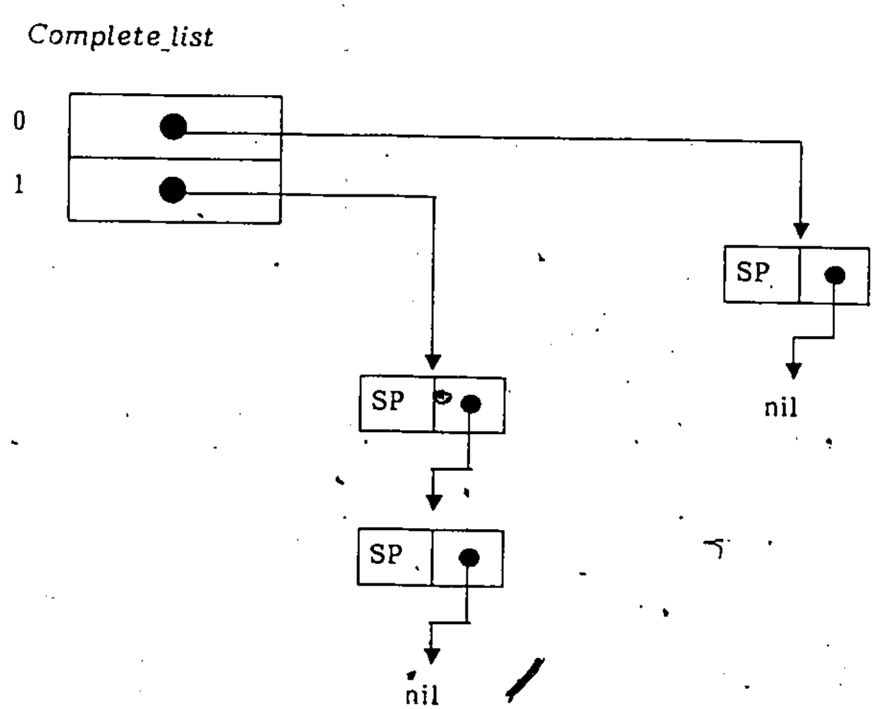


Figure 3.29 *Complete_list* Array for Sample Frame.

3.3 Algorithm Comparisons

3.3.1 Number of Edge Point Comparisons

Boundary connections are established by checking for overlap conditions of segments on successive lines of the image. In method 3, this is achieved by comparing the coordinates of four edge points on each line: the left and right edge points of the current segment, the right edge point of the previous segment and the left edge point of the following segment (figure 3.13). In the new algorithm, the same connections can be determined by examination of only three edge points from each line. These are the left and right edge points of the current segment and the left edge point of the following segment as shown in figure 3.19. Execution is sequential not only in the line-by-line sense, but from left to right along each line as well. The resulting number of comparisons required to detect each of the possible overlap conditions is greatly reduced from that needed in method 3. These are summarized in table 3.1.

The procedure in method 3 begins the processing of new segments by checking for boundary continuation from the previous line to the new line. Four comparisons are made for each of the left and right boundaries for a total of eight operations to detect a region continuation. If continuation conditions are violated, then tests are made for object creation, hole creation, hole termination and finally object termination. Each of these require one comparison operation. The detection of object creation therefore, requires a total of nine comparisons. The split condition requires four comparisons to

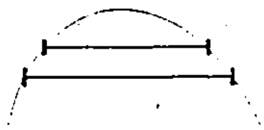
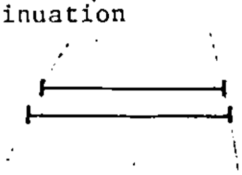
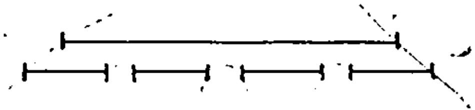
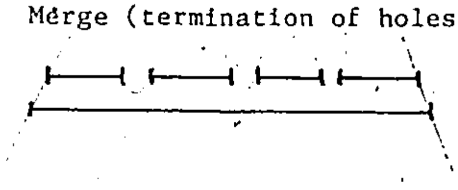
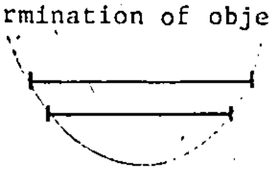
Condition	Number of edge point comparisons required	
	Method 3	New Algorithm
Creation of object 	9	1
Continuation 	8	4
Splits (creation of holes) 	5(n-1) + 8	3(n-1) + 4
	(n = number of segments on line k)	
Merge (termination of holes) 	7(n-1) + 8	3(n-1) + 4
	(n = number of segments on line k-1)	
Termination of object 	12	2

Table 3.1 Number of Edge Point Comparisons.

establish the left boundary continuation, four comparisons to find that the right boundary does not continue, and then five comparisons for each of the remaining segments on the new line until the right boundary is finally continued. Similarly, a merge situation uses four comparisons to detect the left boundary condition, four to find that the right boundary does not continue and then seven comparisons for each of the remaining segments on the previous line. Terminations are discovered when none of the above conditions hold, for a total of twelve edge point coordinate comparisons.

In contrast, the new algorithm begins the processing of new segments by immediately checking for object creation and termination, each requiring only one comparison. If neither condition holds, then segments must overlap and the left boundary continuation is established immediately. Three comparisons only are required for each segment involved in a split or merge situation.

Summary

The total number of edge point comparisons required for the example in figure 3.15 are shown in table 3.2. The new algorithm can be seen to require fewer than 50% of the number of operations used in the algorithm of Method 3.

3.3.2 Memory Requirements

Assuming an image resolution of 256 x 256 elements, a maximum of 256 objects or hole regions (including those eventually found to merge), and no more than 10 segments (20 edge points) per line, the storage required to hold the data structures of Method 3 and the new algorithm are calculated below. Images having the complexity of figure 3.17 would approach these limit specifications.

Line #	Number of Edge Point Comparisons: Method 3	New Algorithm
1 (creation)	9	1
2 (continuation)	8	4
3 (split)	13	7
4 (continuation + split)	21	11
5. (3x continuation)	24	12
6 (merge + continuation)	23	11
7 (termination + split)	25	9
8 (2x continuation)	16	8
9 (continuation + termination)	20	6
Totals	159	69

Table 3.2 Number of Comparisons Required for Example of Figure 3.15.

Method 3

The boundary extraction information is organized in a pair of two-dimensional tables shown in figure 3.12. The Scan Line Array contains storage for each of the edge point coordinates for the previous and new lines and for the object numbers as they are assigned to each edge point. The maximum edge point coordinate can be represented with 8 bits as can the object numbers. The dimensions of the array are then 4 x 20 x 8 bits for a total of 640 bits.

The Object Array structure contains four fields. The 'object #' and 'parent' fields each require 8 bits, given a maximum of 256 objects or holes. The 'type' field needs only 1 bit of storage (body or hole) and the 'status' field requires 2 bits to indicate one of (continuation, termination, merged). The array must contain as many rows as there are objects and holes. The dimensions are therefore 256 x 19 bits for a total of 4864 bits.

New Algorithm

The corresponding structures to the Scan Line and Object Arrays of Method 3 are the *line_buffer* and *active_list* structures, shown in figures 3.21 and 3.22 respectively. The *line_buffer* contains the edge point data for the new line only and therefore requires 20 x 8 bits = 160 bits of storage.

Each node of the *active_list* consists of five pointers; *forward*, *back*, *next*, *last_hole_ptr* and *hole_ptr* each of which require an 8-bit representation for 256 objects and holes. The maximum storage needed is therefore 256 nodes x 40 bits for a total of 10240 bits.

In addition, the *complete_list* array is used to retain pointers to the *active_list* nodes of complete objects in the frame. This will require 256 x 8 bits = 2048 bits.

Summary

The overall storage required by the two methods are:

Method 3: 640 + 4864 = 5504 bits

New Algorithm: 160 + 10240 + 2048 = 12448 bits

For the conditions assumed above, Method 3 will require less than 50% of the storage needed by the new algorithm.

3.3.3 Object Array vs. *Active list* Structure

We again refer to the example of figure 3.18. At each line of the image, the relationships between object and hole regions are represented by the Object Array of Method 3 and by the *active_list* of the new algorithm. The two structures are shown in figure 3.30 as they would exist after line j-1 and after line j of the example. These are now compared:

Method 3

When bodies A and B are found to merge at line j, the 'type' field of the remainder of the Object Array is searched and each hole with 'parent' field 'B' has its 'parent' field changed to 'A'. This requires a time proportional to the number of holes in the entire array. Thus in figure 3.30 (a) and (c), when body B is found to merge with body A, the four holes with parent B require to be converted to the single parent A.

New Algorithm

When objects A and B are found to be connected at line j, the hole list of A is appended to that of B by way of the *hole_ptr* and *last_hole_ptr* links. This is achieved in two operations regardless of the number of holes in either A or B. Thus, in figure 3.30 (b) and (d), when object B is found to merge with object A, the four holes with parent B are converted to parent A in a single operation.

Summary

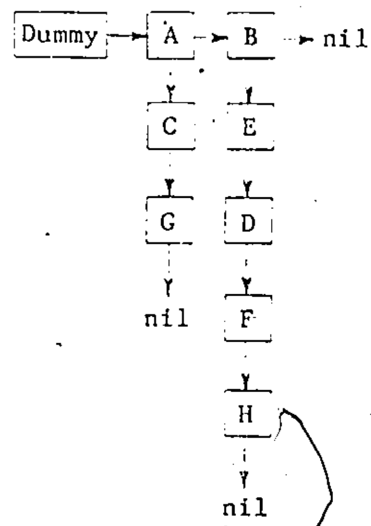
For the simple image of figure 3.18 (one object, six holes) the Object Array search involves just six objects. However, if this configuration were embedded in a complex image containing a large number of hole regions (such as in figure 3.17) the searching process would introduce a significant increase in the processing time.

The initialization of the Object Array before the start of a frame scan requires setting each of the fields to zero. (Object numbers in the Scan Line Array must also be zeroed.) The *active_list* is initialized to contain a single dummy node with a 'nil' pointer ~~to~~ represent an empty list. During processing the *active_list* is constructed and depleted of all nodes before the end of the frame (see the example in section 3.2.5), returning the structure to its initialized state. Processing of subsequent frames does not require line # | Object Array | *Active list*

line j-1

Object #	Type	Parent	Status
A	Body	0	Con
B	Body	0	Con
C	Hole	A	Term
D	Hole	B	Term
E	Hole	B	Term
F	Hole	B	Term
G	Hole	A	Term
H	Hole	B	Term

(a)

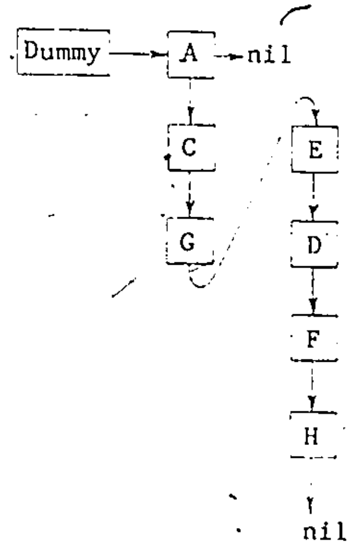


(b)

line j

Object #	Type	Parent	Status
A	Body	0	Con
B	Body	0	Merged
C	Hole	A	Term
D	Hole	A	Term
E	Hole	A	Term
F	Hole	A	Term
G	Hole	A	Term
H	Hole	A	Term

(c)



(d)

Figure 3.30 Processing of Example in Figure 3.18.

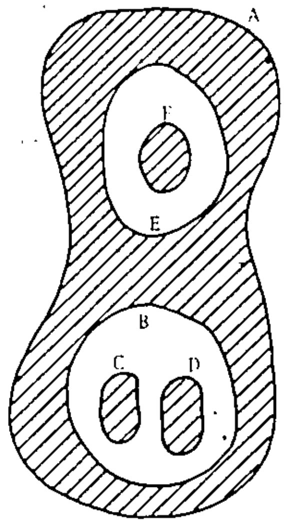
reinitialization. For the Object Array of Method 3, initialization must be performed at the beginning of every frame.

When objects are merged, the 'status' field of the object array is set to 'merged' and the object need no longer be considered. However, its space in the Object Array remains occupied. In the *active_list* structure, merged object nodes (B in our example) are deleted and made available for future objects or holes.

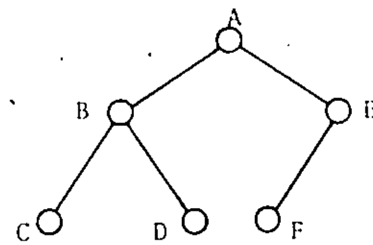
3.3.4 Representation of Frame Contents

A hierarchical ordering of objects and holes contained in a scene has been described by Kulkarni [3.13] and in Cederberg [3.11]. The technique is illustrated in figure 3.31 (a) (b). Alternate levels of the tree represent object and hole regions with the first level indicating the larger objects, and lower levels depicting nested regions. The structure is generated by examination of the fields in the Object Array of Method 3 (figure 3.31 (c)). The corresponding *complete_list* constructed by the new algorithm is shown in figure 3.31 (d).

The hierarchy can be established only at the completion of a frame scan when the parent relationships have been determined and the Object Array is available. In contrast, the *complete_list* structure is generated during the border extraction process. As each object region is completed in the frame, the corresponding node is deleted from the *active_list* and is retained by way of a pointer in the next element of the *complete_list* array. To find, say, the total number of



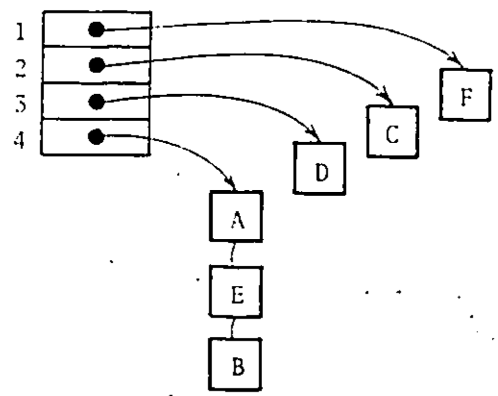
(a)



(b)

Object #	Type	Parent	Status
A	Body	0	Term
B	Hole	A	Term
C	Body	B	Term
D	Body	B	Term
E	Hole	A	Term
F	Body	E	Term

(c)



(d)

Figure 3.31 Representation of Frame Contents.

(a) original image,

(b) object heirarchy,

(c) Object Array representation,

(d) *complete_list* representation.

object regions in the frame, the entries are merely counted. The equivalent operation for Method 3 requires counting nodes on alternate levels of the hierarchy or scanning the Object Array and counting objects with (type = body) and (status = term). Counting the number of holes for a particular object is readily achieved from either the *complete_list* or hierarchy tree, but would require a search of the Object Array to find entries with (type = hole), (status = term) and (parent = desired object #).

The *complete_list* structure has the added advantage that it utilizes the same nodes as were in the *active_list*. The only supplemental storage requirement is the list of pointers to each object node. The hierarchical structure on the other hand, contains nodes which must have a variable number of children. In the example of figure 3.31, the configuration is that of a binary tree. However, if the input image were more complex (e.g. multiple objects with several holes each) the corresponding hierarchical data structure could become computationally unwieldy. The *complete_list* scheme does not represent object containment information as does the hierarchy; this would have to be determined from further processing of the contours.

CHAPTER 4

IMPLEMENTATION

This chapter is devoted to the description of the hardware and software implementations. In section 4.1, the various hardware components of the system are outlined including the architecture which has been designed for flexibility in expansion and high-speed operation. In section 4.2, we describe aspects of the software and major data structures used in the representation of the contents of a frame, the design of the prototype objects database and the command set. In the final section, the robot/vision interface package is described including the extensions made to the VAL programming language [4.1], calibration between the vision and robot coordinate frames and a sample program for parts sorting by the PUMA under visual control.

A photograph of the apparatus is shown in figure 4.1. A solid state TV camera is fixed so as to view a portion of the working area of the PUMA robot. Backlighting provides bimodal images of randomly positioned workpieces which are fed to the vision system where they are converted to binary. The vertex points for each contour are found from which the identity, location and orientation of each object is determined. This information is transferred to the PUMA controller as

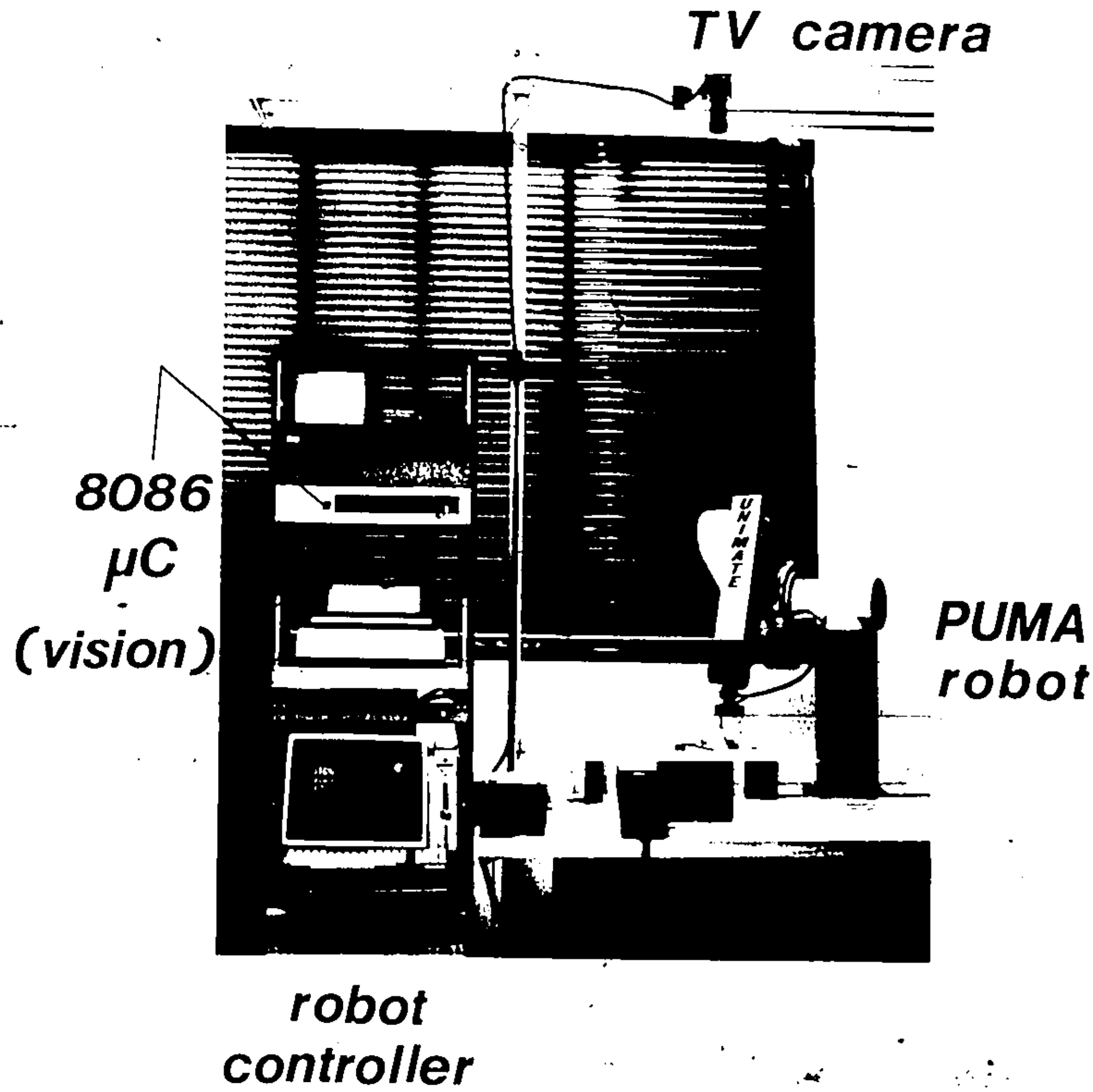


Figure 4.1
Photograph of Robot and Vision System.

a VAL location variable and the robot is then able to move and orient its gripper to pick up the workpiece. The system is initialized with a calibration between the vision and robot coordinate frames, teaching prototypes of objects which are to be recognized and setting the tolerances for each feature. These procedures are outlined in the description of the command set.

4.1 Hardware

The master processor of the vision system is an Intel iAPX 86/20 microcomputer consisting of the 8086-based iSBC 86/12A single board computer [4.2] and the 8087-based iSBC 337 floating point coprocessor [4.3]. Additional RAM and EPROM have been added in the form of the iSBC 300 and iSBC 340 for totals of 64K bytes and 32K bytes respectively. The board also supports a CRT/keyboard and a graphics line printer and is shown in figure 4.2 (a). Image and histogram data are dumped to the iAPX 86/20 by way of a DMA controller over the Multibus [4.4] at a maximum rate of 500K bytes/s. The remainder of the hardware is contained on two additional custom-built cards; the video-rate histogram generator for automatic threshold selection (figure 4.2 (b)) and a utility board containing the dual threshold unit, the system status register, the binary pixel packer and the serial robot communications interface (figure 4.2 (c)).

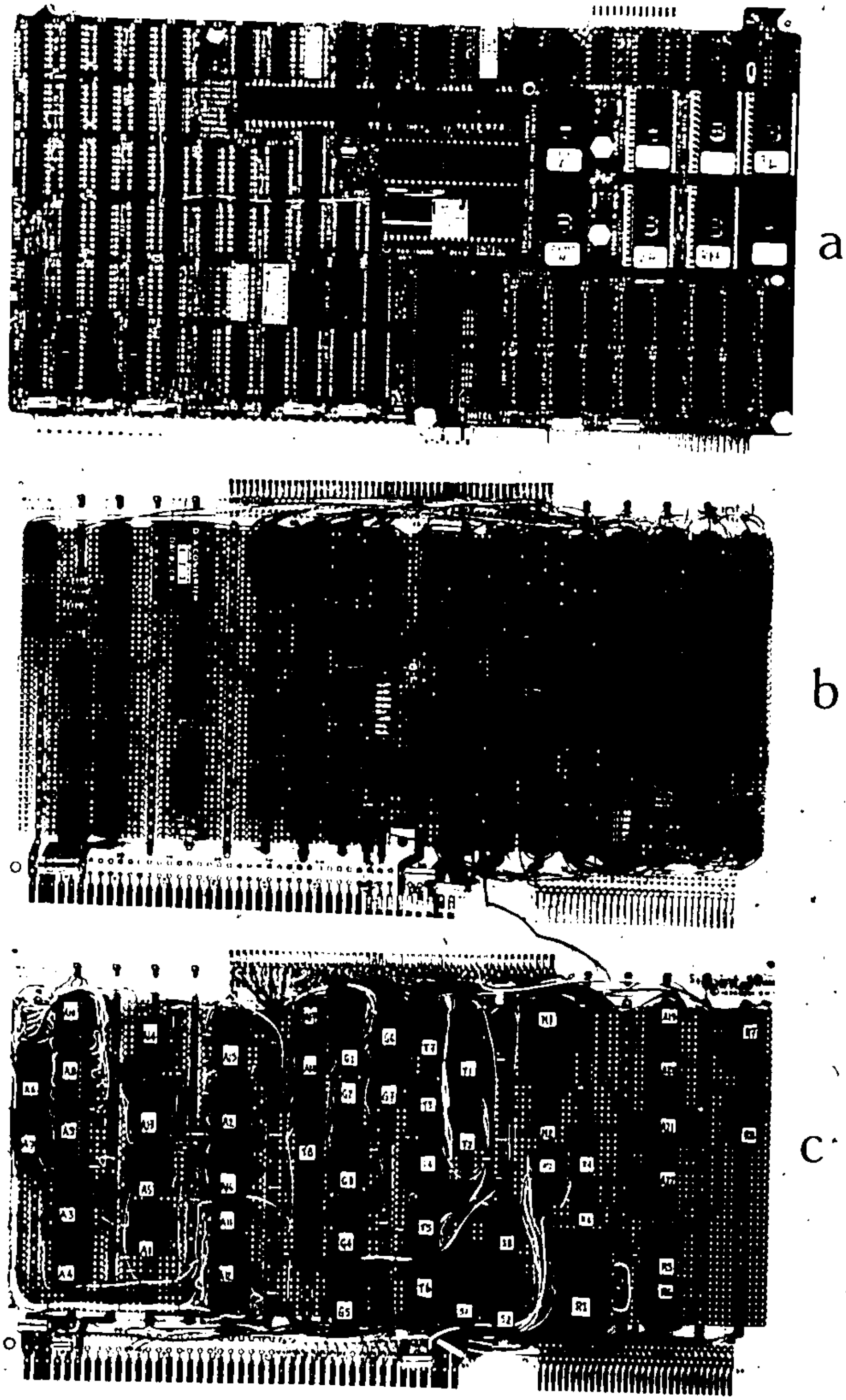


Figure 4.2
Photograph of Circuit Boards.

Images are provided by the General Electric TN2500 digital television camera [4.5]. The camera produces an 8-bit grey-scale quantization for each of 248 x 244 pixels at a 30 Hz frame rate. The scan mode can be selected as interlaced for viewing the image on a standard TV monitor or sequential for the collection of the image for analysis. In sequential scanning, each of the 244 lines are output twice and in sequence as shown in figure 4.3. The camera is fitted with a 75 mm, f1.8 lens and is mounted approximately 1.5 m above the viewing surface. This results in a field of view of approximately 162 mm x 128 mm.

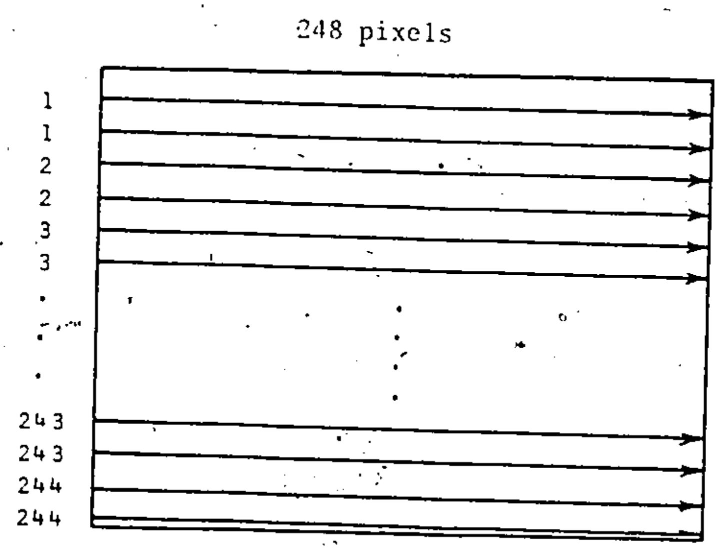


Figure 4.3 Sequential Mode Frame Scan.

4.1.1 Architecture

The hardware architecture of the vision system is depicted in figure 4.4. The design features a 4-bus structure interconnected by dedicated units which have been designed to operate at high speeds.

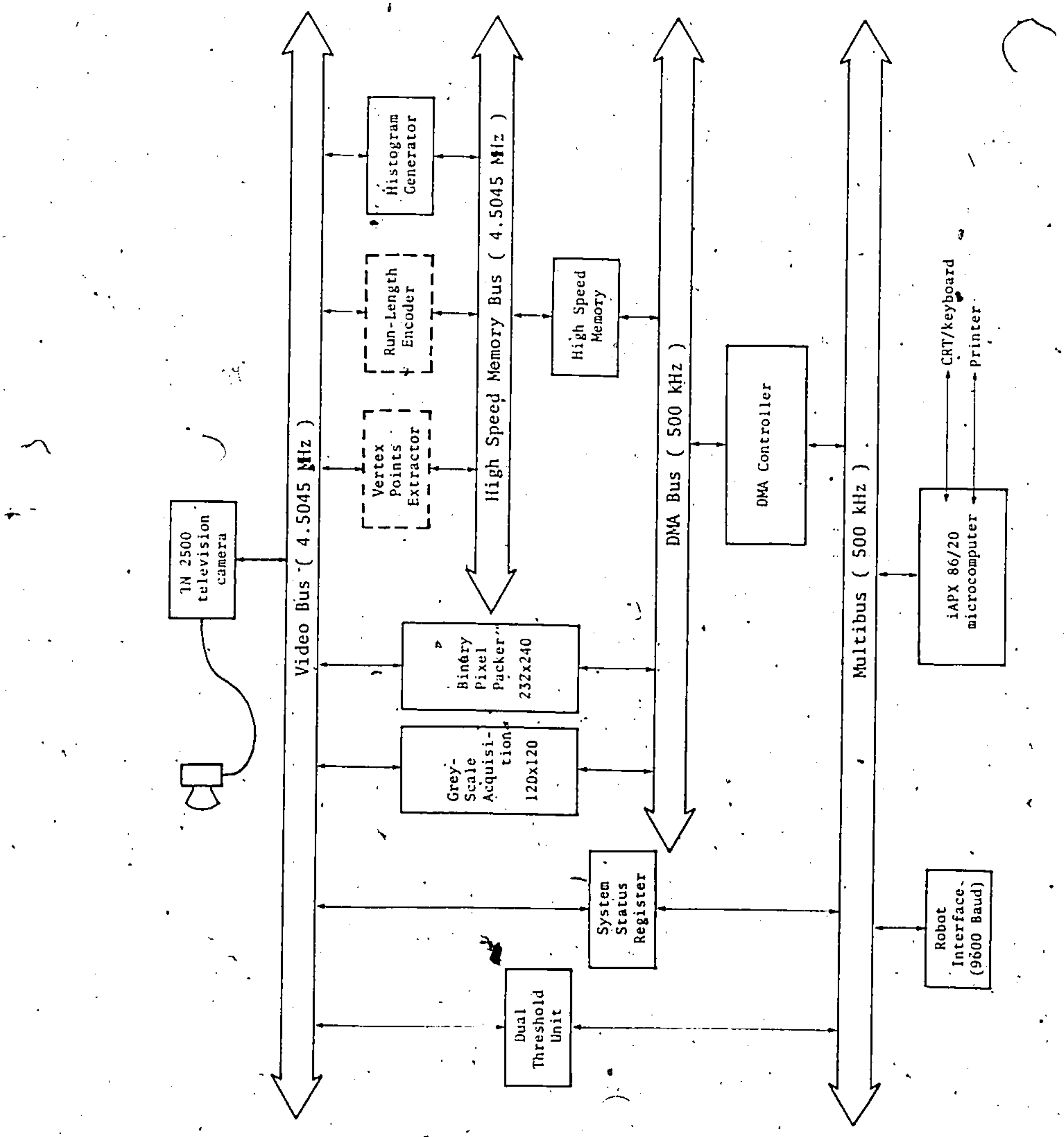


Figure 4.4 System Hardware Architecture.

The position of each unit is dictated by its input and output speed requirements. With this organization, many low-level operations can be performed at video-rates (30 frames/s) and some degree of parallelism can be achieved. For example, histograms are generated simultaneously with binary or grey-scale image acquisition. The design is flexible in that the capabilities of the system can be extended when new hardware modules are available by inserting them between the appropriate buses. This is the case for the 'Vertex points extractor' and 'run-length encoder' shown in broken lines on the diagram. These functions are currently carried out in software for development purposes but are now being converted to video-rate hardware units.

The Intel Multibus connects the iAPX 86/20 microcomputer, the DMA controller, the robot communications interface, the threshold unit and the system status register. The highest frequency required is the 500K bytes/s transfer rate from the DMA controller. The Multibus supports 16-bit data and 20-bit address lines.

The Video Bus comprises the Optomation Interface signals (connector J4) of the General Electric TV camera and the binary video output from the threshold unit. The highest clock rate on this bus is the element rate clock (ERC) at 4.5045 MHz which indicates the presence of a new pixel in the raster. Other lines include 8-bits of video data, horizontal and vertical synchronization pulses and the selection of camera scan mode.

Data to be transferred to the microcomputer RAM is placed on the DMA bus (8 bits wide) and a transfer request is generated. In this system, data is provided by the binary pixel packer for transfer of binary images or by the high speed memory for the transfer of histograms (a grey-scale image can also be acquired at the expense of decreasing the spatial resolution to 120 x 120, but is not used in this version of the vision system).

Modules which are too fast to interface directly to the DMA Bus are handled by the High Speed Memory Bus. The histogram generator, for example, requires storage which can keep pace with the video signals. This is provided in the high speed memory unit.

In the following sections, we outline the designs of the individual units in the form of functional descriptions using block diagrams.

4.1.2 Binary Pixel Packer

Binary iamges are collected by way of the Binary Pixel Packer module located between the Video Bus and the DMA Bus. A full 232 x 240 binary image is acquired in a single frame time (33 ms) and stored in the RAM of the iAPX 86/20 via the DMA controller. This module achieves a reduction in speed by packing 8 binary pixels into one byte before each DMA transfer. The block diagram is shown in figure 4.5.

When requested to capture a binary image, the frame detection logic awaits the occurrence of a vertical sync pulse indicating the beginning of a new frame. The MSB of the counter is used to enable one of the serial in/parallel out shift registers and disable the other. When the horizontal sync pulse is detected, the counter is

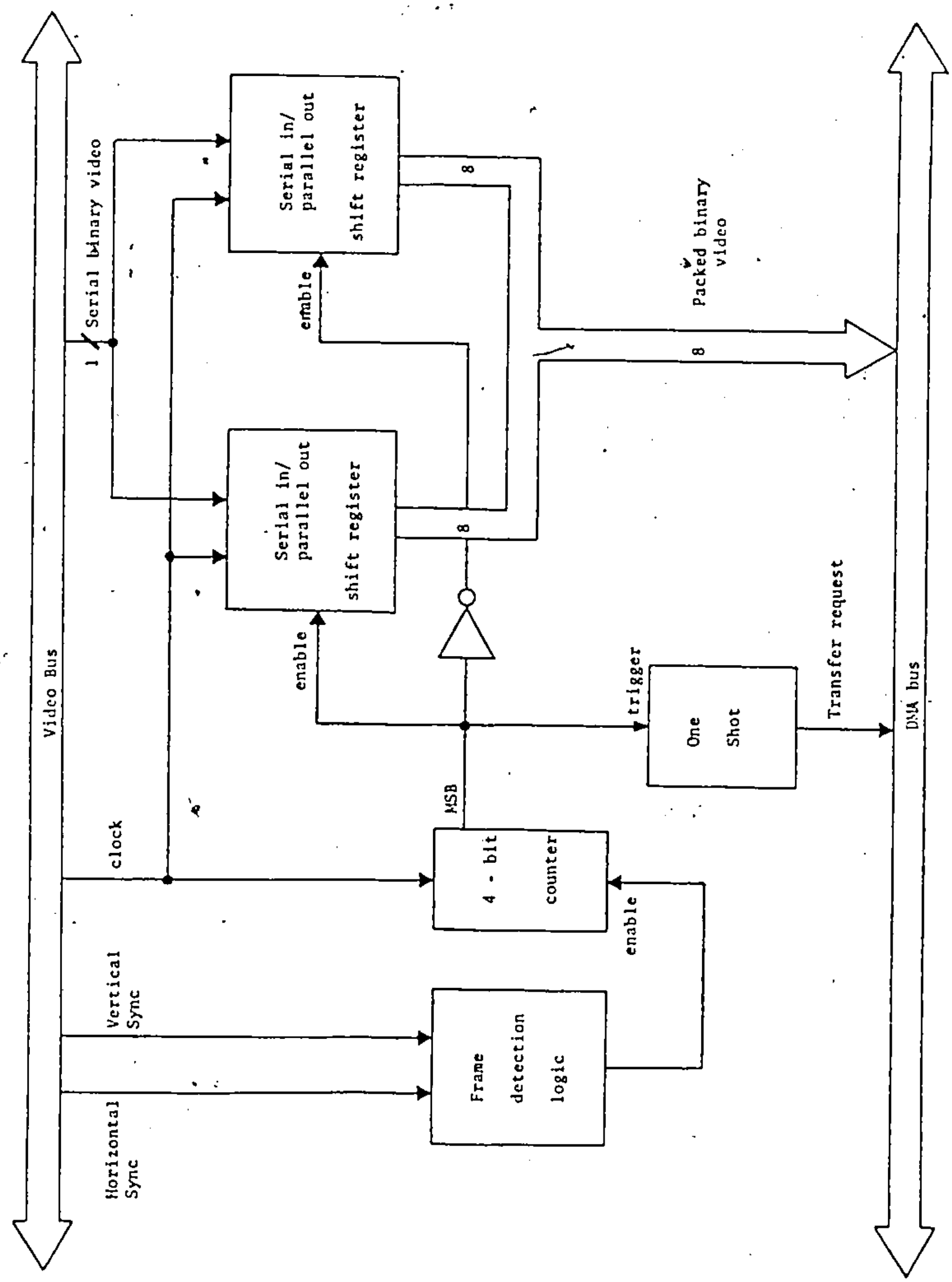


Figure 4.5 Binary Pixel Packer.

cleared and begins counting the element rate clock pulses. Simultaneously, binary pixel values are shifted into the serial input of the enabled shift register. After eight clocks, the shift register is full and the MSB of the counter toggles disabling this shift register and enabling the other. The next eight pixels will then be collected in the second shift register. At the same time, the previous eight pixels appear on the DMA bus in parallel and a monostable (one-shot) is triggered from the MSB of the counter to generate a transfer request to the DMA controller. After the next eight clock pulses the MSB of the counter again toggles and the process is repeated until an entire frame of binary image data has been transferred.

4.1.3 Histogram Generator

Modules which operate on video signals but provide data at rates too high for direct connection to the DMA Bus are inserted between the Video Bus and the High Speed Memory Bus. The histogram generator is in this category. Each pixel in the raster, quantized to 8 bits, is used to address 256 16-bit memory locations each of which are read, incremented and written back into the high speed RAM before the occurrence of the next pixel. If at the beginning of a frame each memory location is cleared, then the RAM will contain the histogram of grey-levels at the end of the frame. This unit is used for automatically adjusting the threshold unit for conversion of grey-scaled images to binary. A functional diagram of the histogram generator is shown in figure 4.6.

The high speed memory (histogram buffer) is constructed from four Intel 2148-3 RAM's organized as 1024 x 16 bits. These were selected for their fast access times (50 ns) to keep pace with the pixel rate of 4.5045 MHz. Upon the presentation of each pixel, the memory must be read, the contents incremented and rewritten before the next pixel appears 220 ns later. The memory is organized as four pages of 256 x 16 bits allowing for the storage of four histograms at any one time. Two bits of the system status register are used to select the page. Ths histogram generator is operated in one of three modes:

Clear

The clear mode initializes the high speed memory by setting the contents of all memory locations to zero. This is achieved by addressing the memory from the sequencer which provides a count from 00 to FF (hexadecimal) by way of the function selector. Simultaneously, the eraser unit places TTL LO values on the data bus. The write line is then pulsed for each address from the timing and control unit. At the conclusion of the count, the selected page of the memory will contain zeros and is ready to begin accumulation of a histogram.

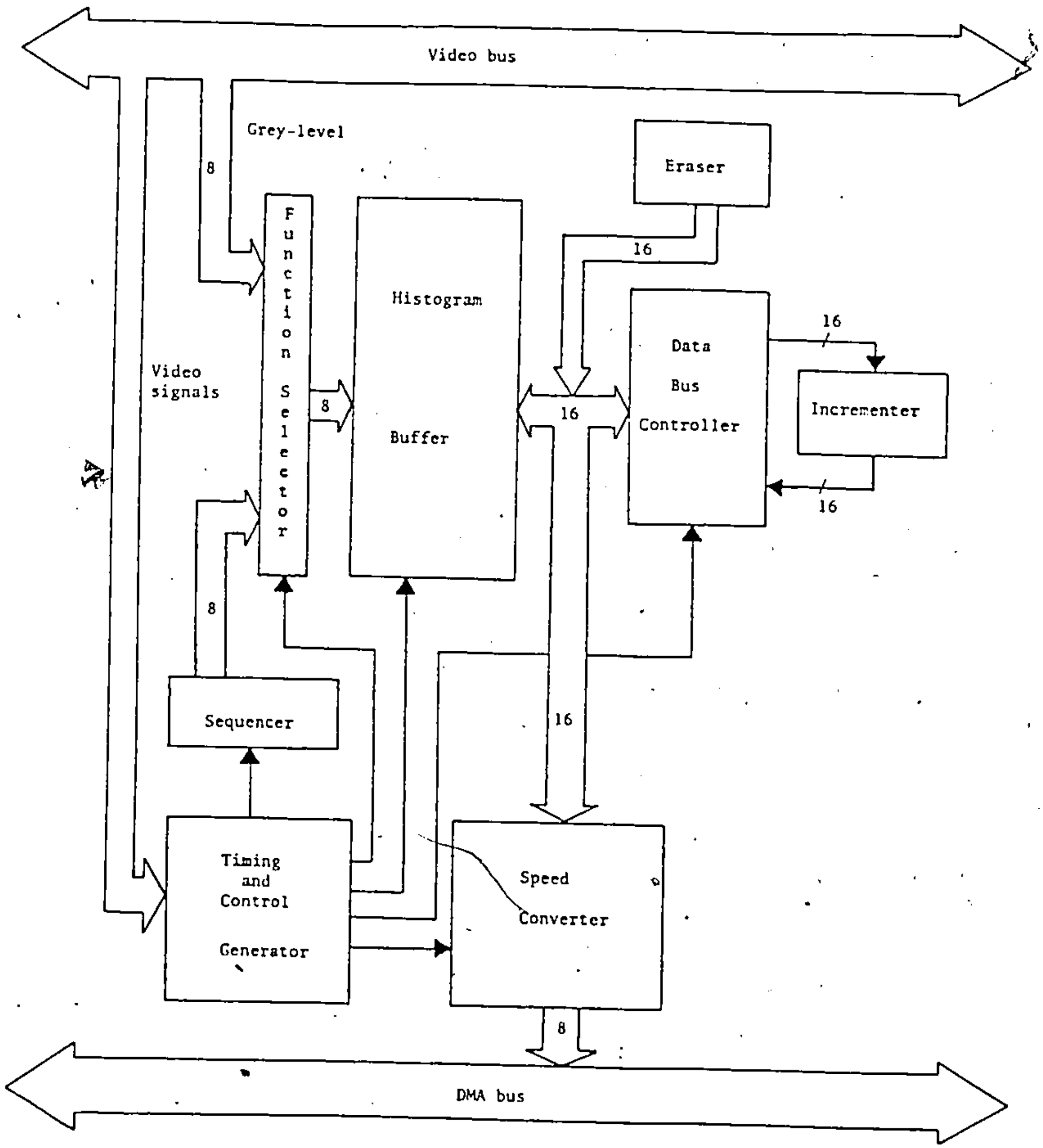


Figure 4.6 Video-rate Histogram Generator.

Increment

In increment mode, a histogram is generated in the high speed memory (assuming that a clear operation has previously been performed). Memory addresses are provided by the 8-bit grey-level of each pixel selected via the function selector from the Video Bus. As each location is addressed, the memory contents are read and placed on the data bus. The data bus controller directs this value to the incrementor and returns the updated value to the data bus where it is written into the same location. The incrementor unit consists of two 8-bit binary adders to increment the incoming 16-bit word within 43 ns.

Read

At the end of a frame, the complete histogram is contained in the histogram buffer. Using the sequencer to generate addresses, each location is read and the corresponding contents are placed on the DMA Bus. The timing and control unit then issues transfer request pulses at a rate of 500 kHz until the entire histogram is transferred to the main storage of the iAPX 86/20 board for processing.

4.1.4 Threshold Unit

The dual threshold unit converts 8-bit grey-scale pixels (p_i) to binary pixels (b_i) as shown in figure 4.7. Two 8-bit registers (Intel 8282) hold the lower (T_ℓ) and upper (T_h) thresholds which may be set manually or under program control from the analysis of histogram data [4.6]. The value of b_i is set as follows:

$T_\ell < p_i < T_h$	Background Color	b_i
0	0 (white)	0
1	0	1
0	1 (black)	1
1	1	0

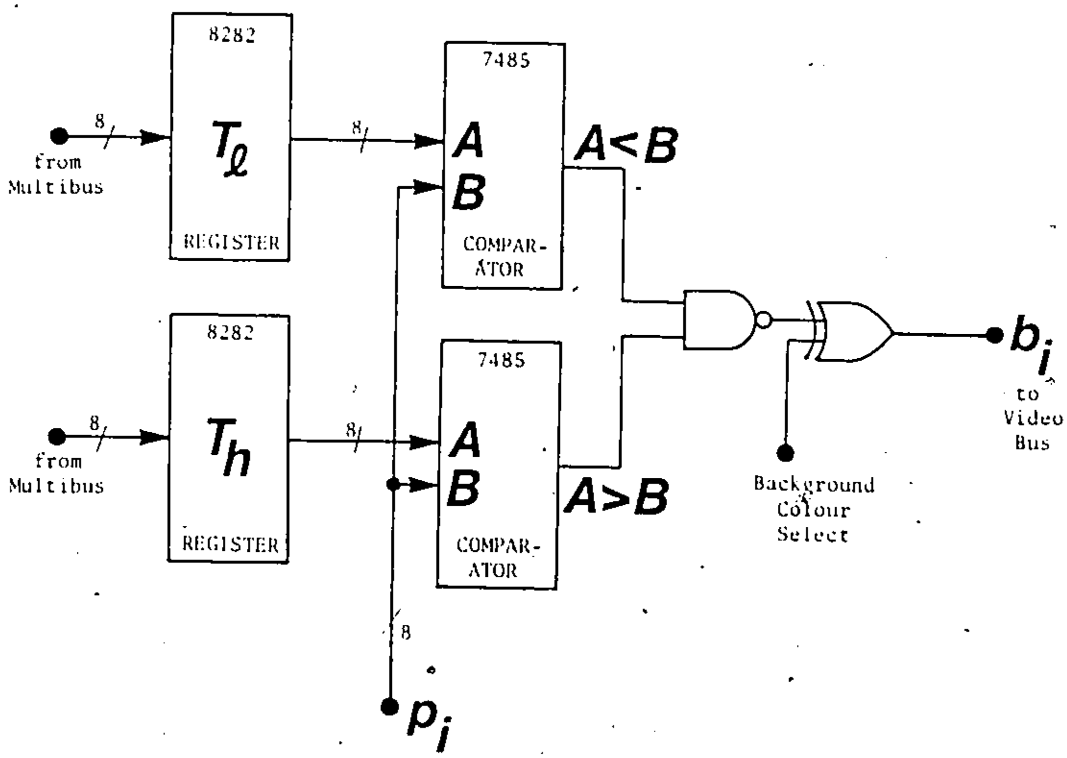


Figure 4.7 Threshold Unit.

4.1.5 System Status Register

The system status register is situated between the Multibus and the Video bus. Constructed from an Intel 8282 8-bit register, bits are provided for selection of image background color (white or black), camera scan mode (interlaced or sequential) and histogram pages. As shown in figure 4.8, the remaining four bits are unused and available for future expansion. This may include, for example, the selection of one of multiple cameras connected to the vision system.

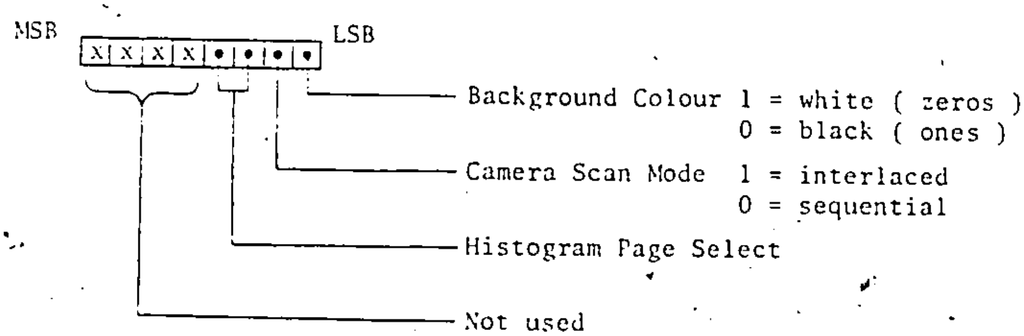


Figure 4.8 System Status Register.

4.2 Software

The system software resides on eight 2732A EPROM's on the iSBC 86/12A board. All programming is in the Intel PL/M-86 [4.7] language and developed with the aid of MCS-86 software development utilities [4.8] which run on an MDS-230 microprocessor development system. The 8087 support libraries [4.9] were used to access the floating point facility.

The 16-bit, segmented architecture of the 8086 CPU is well suited to high level language implementations. PL/M-86 allows for complex data structures and is amenable to a modular, top-down design, the organization of which is shown in figure 4.9. On power-up the "Main" module is executed which performs various low-level hardware initialization. These include programming the Baud rate generators, the parallel interface, the 8087 coprocessor and setting the default grey to binary threshold levels. Control is passed to the "VAL interface" module which provides the link between the vision system software and the VAL operating system of the PUMA robot. At this point, the vision software is transparent and the environment is that of the VAL system.

Access to the vision software is gained from the VAL monitor by way of the "Vision Command Dispatch" module. From here, all vision commands typed by the user are interpreted and the appropriate function is dispatched from the "Vision Commands" module.

Lower level functions are organized as libraries and are available to any of the high level modules. The "Vision Library" consists of four modules whose functions are briefly described:

display routines: for displaying vision system information on the CRT or line printer.

get CFD: performs the vertex point contour navigations and generates the CFD.

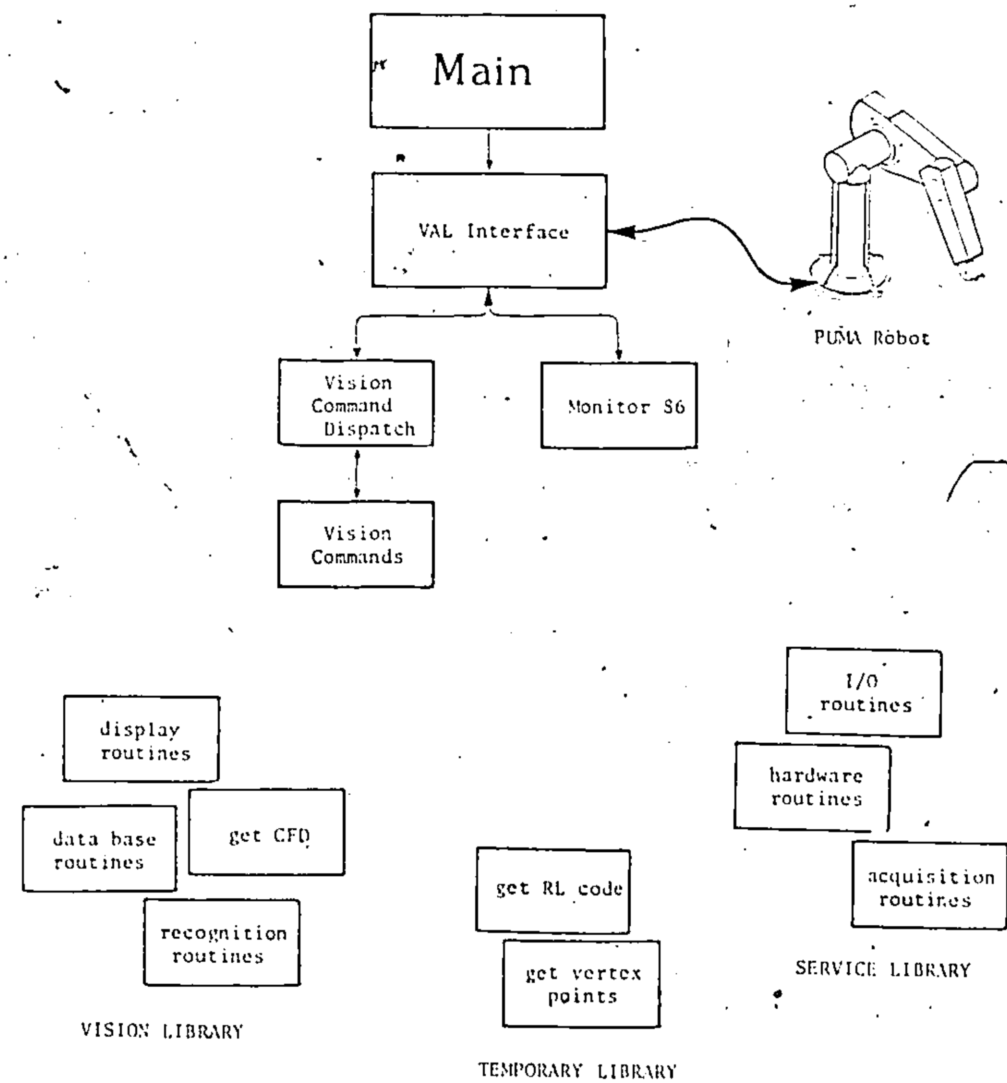


Figure 4.9 System Software Architecture.

data base routines: contains the software for managing the data base of prototype objects.

recognition routines:	extracts feature vectors from prototype objects in the data base and those in the CFD and performs the recognition of objects based on the specified tolerances.

The lowest level functions are arranged in the "Service Library" consisting of three modules:

I/O routines:	contains a variety of routines for input and output from the CRT/keyboard, the line printer, and the robot.
hardware routines:	a collection of procedures for accessing the system status register, the threshold unit and the histogram generator.
acquisition routines:	configure the DMA controller for collection of histograms, packed binary images or grey-level images.

Two additional modules are contained in a "Temporary Library". These are functions which are currently achieved in software but are currently being converted to hardware units (see figure 4.4):

get RL codes:	generate the set of run-length codes from the packed binary representation ~~of~~ an image.

get vertex points: implements the sequential boundary extraction algorithm which extracts the vertex points from the set of run-length codes.

4.2.1 Frame Description Data Structure

Each time an image is analyzed by the system, a description of the contents of the frame is generated, based on a data structure named Current Frame Description (CFD). This is shown in figure 4.10:

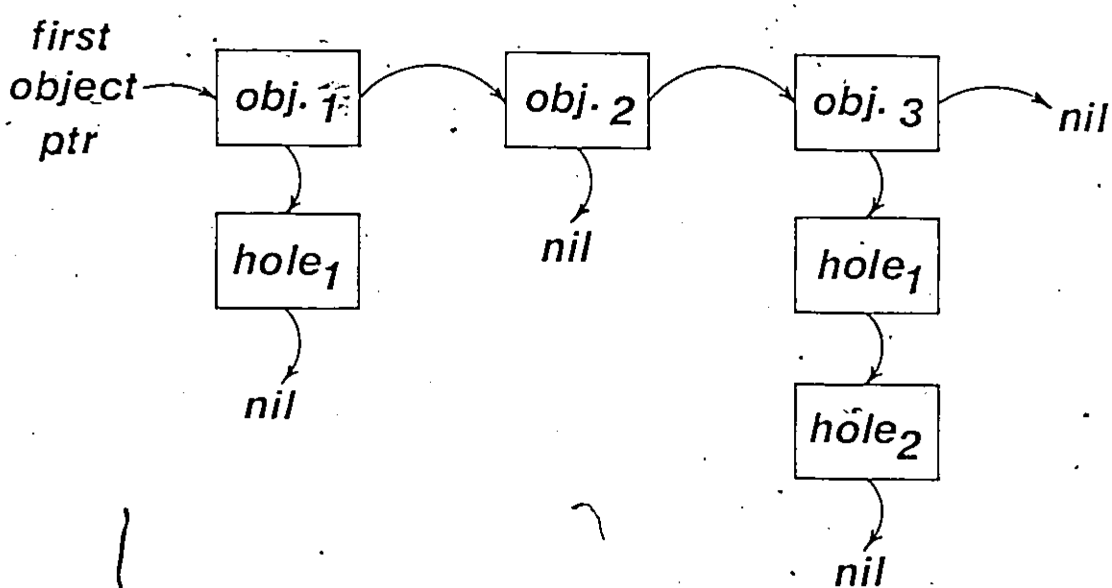
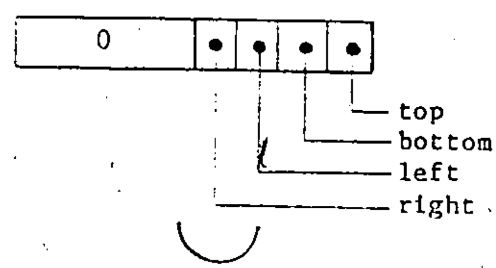


Figure 4.10 Current Frame Description Data Structure.

The structure is a singly-linked list of nodes representing each object contour found in the frame. The beginning of the list is indicated by a pointer (*first object ptr*) and the holes associated with each object contour are represented by a second singly-linked list of nodes headed by the corresponding object node. The fields of each node in the structure are:

Field	Data Type	Description
X_c	REAL	centroid coordinates
Y_c	REAL	
Orientation	REAL	angle of maximum radius
Area	REAL	enclosed area
Perimeter	REAL	perimeter length
Max_radius	REAL	maximum radius from (X_c, Y_c) to perimeter
Hole_ptr	WORD	next hole pointer
Next	WORD	next object pointer
Start	WORD	pointer to first vertex point
Edge	BYTE	frame edge intersection flags



Notes:

1. All dimensions are in units of pixels.
2. Frame edge intersections are indicated by a logical HI in the corresponding bit.
3. NIL is indicated by the value (OFFFFH).

The advantage of this type of structure is very fast access to descriptions of the frame. For the example shown by figure 4.10, it is easy to establish that the frame contains three object regions. Of these, object 1 contains one hole, object 2 has none, and object 3 contains two. To find, say, the largest hole in object 3, it is a simple matter to step through its hole list and retain the node of greatest area.

The dimensions of the field quantities are stored in the CFD in units of pixels and must be multiplied by a scale factor to convert to actual lengths. This scale factor is determined during a calibration procedure described later. If any part of an object region intersects an edge of the frame, the corresponding bit in the *Edge* field is set HI as an indication that the feature measurement will not be accurate. Feature measurement is still carried out for the visible portions of the object. The *start* field is a pointer to the starting point in the *vertex_list* of the contour.

4.2.2 Data Base Design

Prototype objects are taught to the system by placing a sample of the object in view and storing its feature measurements and name (specified by the user) in a data base. A library of routines has been developed for managing addition, deletion and searching operations. The data base is of the relational model [4.10] and consists of two primary data structures as shown in figure 4.11.

INDEX:

	name BYTE(10)	*SRA* WORD
0		
1		
2		
:		
74		

FEATURE:

	Area REAL	*Perimeter* REAL	*Max_rad* REAL	*hole_ptr* WORD	*next_free_SRA* WORD
0					
1					
2					
:					
149					

Figure 4.11 Data Base Structure.

The *INDEX* relation is used to hold the names of all prototype objects which are taught to the system. A maximum of 10 ASCII characters in the *name* field is allowed. The corresponding *SRA* (stored record address) field is a pointer into the relation *FEATURE* where the actual object feature measurements are stored. All entries into *INDEX* are placed in alphabetical order. Since *INDEX* is used as the key to the data base, this allows a fast binary search to be implemented. The worst case access time to find any object name is $O(\log_2 k)$, where k is the number of entries (the current maximum has been arbitrarily set at k = 75). For additions to *INDEX*, a search is made to find the correct position for the new object name and all entries below this position are moved down by one. On deletion of an entry, any entries below that which is deleted will be moved up by one position. In this way, there is never an empty line in *INDEX*.

The relation *FEATURE* holds the values of the features of objects in the index. This is a dynamic structure in that whenever an object is deleted from *INDEX*, the corresponding space in *FEATURE* is made available to future addition operations so that there are always 150 (arbitrary maximum) spaces available for object (or hole) features. The *hole_ptr* field is used to point to the *SRA* of the feature values of any holes which may occur in an object. If this field is *NIL*, then the object contains no holes; otherwise this field forms a linked list of holes contained in the object. The *next_free_SRA* field (together with a variable *free_SRA*) are used to determine the next available space in *FEATURE*. *Area*, *perimeter* and *max_ rad* are the respective feature values (in units of millimeters) and additional features can be stored by adding columns to the structure. The number of objects in the data base is stored in the variable *No_objects_in data_base*.

As in the CFD data structure, much information about the objects is implicitly stored in the *FEATURE* relation. For example, the number of holes in an object is never stored directly, but can easily be counted by stepping through the linked list formed by the *hole_ptr* field. Also, since the information is stored in millimeters, the data base need not be changed if the camera lens or position are changed. The calibration procedure compensates for this.

4.2.3 Command Set

The human interface is an important design consideration for machine vision systems. A high level command set facilitates system programming such as adaptation to different lighting conditions, teaching of prototype objects and system calibration. A command set was developed for our system and is summarized below. The descriptions omit error recovery messages and other user aids to which considerable attention has also been given. The instructions are listed alphabetically.

BACKGROUND

The background colour is selected to be either black or white. This capability is useful, for example, where it may be desired to measure the features of small holes in large objects which cannot be contained within the field of view.

```
example:        BACKGROUND @

                Background colour:  WHITE(O)
                Change (Y/N)?  Y
                Enter background colour (B/W):  B
```

CALIBRATE

This establishes the scale factor between linear dimensions and those measured by the vision system in pixels. The coordinates of the centroid of an object are recorded in two positions within the frame and the user then enters the actual distance between these points in millimeters. For best accuracy, the calibration object should be one for which the user can precisely determine the centroid point (a circular disk with a small hole in the center is appropriate) and the two points should be as far apart in the frame as possible.

The calibrate procedure is aborted if more than one object appears in the frame or if the calibration object intersects any edge of the frame.

example: CALIBRATE @

Position object in frame and hit CR: @
Position object in frame and hit CR: @
Enter actual distance in mm: 238 @

CAMERA

The GE TN2500 camera is capable of operating either in the interlaced mode (compatible with a standard TV monitor) or in the sequential mode which must be used for collecting an image.

example: CAMERA @

Type camera scan mode (I/S): I

CFD

The number of objects in the scene is reported and followed by the values of the features measured for each object, including any holes. The coordinates of the centroid of each object and each of any holes are included. If any object is intersecting an edge of the frame, this is reported as a warning to indicate that the measurements are only of that portion of the object which appears in the frame.

example:

```
CURRENT FRAME DESCRIPTION:
----------------------------------------------------------------------

Frame contains 3 objects
----------------------------------------------------------------------

OBJECT #1    *** WARNING *** RIGHT  Edge Intersections

    Area              = -592.9  sq.mm.
    Perimeter         = 121.0  mm.
    P2A               = -1.965
    Max. Radius       = 26.4  mm.
    # of Holes        = 0

    Centroid (X.Y) = 198.3 , 60.2  mm.
    Orientation       = 41.2  degrees

----------------------------------------------------------------------

OBJECT #2

    Area              = -753.2  sq.mm.
    Perimeter         = 125.3  mm.
    P2A               = -1.660
    Max. Radius       = 22.0  mm.
    # of Holes        = 2

    Centroid (X.Y) = 137.3 , 84.4  mm.
    Orientation       = -65.6  degrees

  HOLE #1

    Area              = 97.7  sq.mm.
    Perimeter         = 37.0  mm.
    P2A               = 1.114
    Max. Radius       = 6.3  mm.

    Centroid (X.Y) = 139.9 , 72.5  mm.
    Orientation       = -28.3  degrees

  HOLE #2

    Area              = 157.4  sq.mm.
    Perimeter         = 46.8  mm.
    P2A               = 1.110
    Max. Radius       = 7.6  mm.

    Centroid (X.Y) = 134.6 , 93.2  mm.
    Orientation       = -23.5  degrees

----------------------------------------------------------------------

OBJECT #3

    Area              = -639.4  sq.mm.
    Perimeter         = 157.8  mm.
    P2A               = -3.099
    Max. Radius       = 29.3  mm.
    # of Holes        = 0

    Centroid (X.Y) = 63.4 , 118.5  mm.
    Orientation       = -97.6  degrees
```

CLEAR

All objects which have been learned are deleted from the data base, and counters for objects in the data base, objects recognized and objects rejected are set to zero. (see STATUS command)

```
example:        CLEAR @

                Are you sure (Y/N)?  Y
                Data base cleared ...
```

DELETE

Any specified object may be deleted and the space in the data base occupied by the features of this object is made available for future "LEARN" commands.

```
example:        DELETE @

                Enter object name:  WRENCH @
                ok ...
```

DISPLAY

This instruction will display the current binary image in its packed-binary form. Pixels are printed in hexadecimal characters (58 to a line, representing 58 x 4 = 232 pixels) in 240 lines. A sample display for a hexagonal nut is shown below. This may be used to check the quality of the image in order to adjust the binary thresholds and/or lighting.

example:

```
0000000000000000000000000000000000000000000000000000000000000000   0
0000000000000000000000000000000000000000000000000000000400444554   1
0000000000000000000000000000000000000000000000000000000000000000   2
0000000000000000000000000000000000000000000000000000000000000000   3
0000000000000000000000000000000000004000000000000000000000000000   4
000000000000000000000000000000000001F000000000000000000000000000   5
000000000000000000000000000000000003F800000000000000000000000000   6
00000000000000000000000000000000000FFC00000000000000000000000000   7
00000000000000000000000000000000001FFF00000000000000000000000000   8
00000000000000000000000000000000007FFF80000000000000000000000000   9
00000000000000000000000000000000007FFFC0000000000000000000000000   10
0000000000000000000000000000000001FFFFF0000000000000000000000000   11
0000000000000000000000000000000003FFFFF8000000000000000000000000   12
0000000000000000000000000000000007FFFFFC000000000000000000000000   13
000000000000000000000000000000000FFFFFFF000000000000000000000000   14
000000000000000000000000000000001FFFFFFF000000000000000000000000   15
000000000000000000000000000000003FFE1FFF800000000000000000000000   16
000000000000000000000000000000003FF807FF800000000000000000000000   17
000000000000000000000000000000003FF003FF800000000000000000000000   18
000000000000000000000000000000003FE001FF800000000000000000000000   19
000000000000000000000000000000003FC000FF800000000000000000000000   20
000000000000000000000000000000003FC000FF800000000000000000000000   21
000000000000000000000000000000003F80007F800000000000000000000000   22
000000000000000000000000000000003F80007F800000000000000000000000   23
000000000000000000000000000000003F90007F800000000000000000000000   24
000000000000000000000000000000003F80007F800000000000000000000000   25
000000000000000000000000000000003F90003F800000000000000000000000   26
000000000000000000000000000000003F80007F800000000000000000000000   27
000000000000000000000000000000003FC0003F800000000000000000000000   28
000000000000000000000000000000003F80007F800000000000000000000000   29
000000000000000000000000000000003F80007F800000000000000000000000   30
000000000000000000000000000000003FC0007F800000000000000000000000   31
000000000000000000000000000000003FC0007F800000000000000000000000   32
000000000000000000000000000000003FE000FF800000000000000000000000   33
000000000000000000000000000000003FE000FF800000000000000000000000   34
000000000000000000000000000000003FF001FF800000000000000000000000   35
000000000000000000000000000000003FF803FF800000000000000000000000   36
000000000000000000000000000000003FFE0FFF800000000000000000000000   37
000000000000000000000000000000001FFFFFFF000000000000000000000000   38
000000000000000000000000000000001FFFFFFE000000000000000000000000   39
00000000000000000000000000000000007FFFFF800000000000000000000000   40
00000000000000000000000000000000007FFFFF000000000000000000000000   41
00000000000000000000000000000000001FFFFC000000000000000000000000   42
0000000000000000000000000000000000FFFF80000000000000000000000000   43
00000000000000000000000000000000003FFF00000000000000000000000000   44
00000000000000000000000000000000001FFC00000000000000000000000000   45
00000000000000000000000000000000007F8000000000000000000000000000   46
00000000000000000000000000000000000003F0000000000000000000000000   47
00000000000000000000000000000000C000E0000000C0000000000000000000   48
0000000000000000000000000000000000000000000000000000000000000000   49
0000000000000000000000000000000000000000000000000000000000000000   50
0000000000000000000000000000000000000000000000000000000000000000   51
0000000000000000000000000000000000000000000000000000000000000000   52
0000000000000000000000000000000000000000000000000000000000000000   53
0000000000000000000000000000000000000000000000000000000000000000   54
0000000000000000000000000000000000000000000000000000000000000000   55
0000000000000000000000000000000000000000000000000000000000000000   56
0000000000000000000000000000000000000000000000000000000000000000   57
0000000000000000000000000000000000000000000000000000000000000000   58
0000000000000000000000000000000000000000000000000000000000000000   59
```

DESCRIBE

The values of the features of any specified object in the data base are displayed. Similar information about any hole in an object is also provided.

example:

Data Base: Features of object: NUT

Area	=	-455.3 sq.mm.
Perimeter	=	81.9 mm.
P2A	=	-1.173
Max. Radius	=	13.2 mm.
# of Holes	=	1

HOLE #1

Area	=	160.9 sq.mm.
Perimeter	=	46.9 mm.
P2A	=	1.089
Max. Radius	=	7.7 mm.

FIND

The name of the desired object is prompted for and if it exists in the current frame, its centroid coordinates are reported on the CRT. Multiple occurrences are denoted. A distinction is made between a FIND command and an IDENTIFY command (described later). In the FIND command, the user specifies the name of an object to be located. Its features are then extracted from the data base and a comparison is made to the feature vector of every object in the current frame. Any matches (determined by the tolerance settings) are indicated by displaying the centroid coordinates. The time required is proportional only to the number of objects in the current frame.

example:

```
FIND @
Enter object name:  WRENCH @

Found at:
  (Xc,Yc) =  93.6,  129.7 mm.
  (Xc,Yc) = 172.6,  185.1 mm.
```

HELP

This command prints a brief summary of all the commands on the CRT.

IDENTIFY

The identity from the data base index of every object in the frame is reported. If the tolerance settings are such that more than one prototype in the data base matches a particular object, then each possible identity is indicated. If no matches are found for an object its identity is shown as unknown, and if an object is intersecting any edge of the frame, this is also designated. Identities are exhibited with corresponding centroid coordinates. Since the feature vector for every object in the frame must be checked against that of each feature vector in the data base, the time required for this command is proportional to the product of the number of objects in the frame and the number of objects in the data base.

```
example:     IDENTIFY @

             Object #1
               at (Xc,Yc) =  27.5, 212.6 mm. is NUT
             Object #2
               at (Xc,Yc) =  56.2,  91.8 mm. is WRENCH
             Object #3
               at (Xc,Yc) =  75.8, 107.2 mm. is NUT
             Object #4
               at (Xc,Yc) = 126.2, 125.0 mm. is unknown ...
             Object #5
               at (Xc,Yc) = 142.1, 181.7 mm. is NUT or WASHER
```

LEARN

New object prototypes are taught to the system using this command. The desired name is prompted for, and added to the data base index in alphabetical position and the feature measurements are added to the data base. When an object is taught to the system, it must be the only object in the frame and must not be intersecting any edges. A separate LEARN operation is performed for each stable state of a single object.

```
example:       LEARN @
               Enter object name:  WING_NUT @
               ok ...
```

LIST

This command generates a list of the data base index which is the alphabetical inventory of the names of all the objects taught to the system using LEARN commands.

example:

Data Base: Index	
1	BLOCK_1
2	BLOCK_2
3	BLOCK_3
4	GEAR
5	GEAR-8mm
6	KEY
7	NUT
8	NUT.SIDE
9	WASHER
10	WRENCH

LOOK

The vision system is instructed to acquire a binary image and to generate the vertex list for each object in the frame. The current frame description is obtained by navigating through the vertex list of each object twice. A LOOK command is automatically executed wherever needed in the commands:

CALIBRATE, FIND, IDENTIFY and LEARN.

STATUS

A sample output of the STATUS command is shown. This allows the user to examine a summary of the current settings of the vision system variables. These are:

No. objects learned: the number of objects currently in the database.

No. objects recognized: each time an IDENTIFY command is issued, the number of objects successfully identified is added to this count.

No. objects rejected: each time an IDENTIFY command is issued, the number of objects not recognized is added to this count. Together with the number of objects recognized, these two counters indicate the success rate of the system so that tolerances can be adjusted.

Lower, Upper threshold: the thresholds used for conversion from grey-scaled to binary images.

Background: the current setting of the background colour (see BACKGROUND command).

Scale factor: the conversion from pixel dimensions to linear dimensions in mm/pixel. This value is established by a CALIBRATE command.

example: STATUS @

```
System Status
- - - - - - - - - - - - - - - - - - - - -
No. objects learned:       9
No. objects recognized:   27
No. objects rejected:      2

Lower threshold:   63
Upper threshold:  255

Background:    WHITE(0)

Scale factor:  1.0416 mm/pixel
```

TOLERANCE

The amount by which a feature of any object can differ from the value of the corresponding feature stored in the data base is set by the user.

```
example:        TOLERANCE @

                Data Base:  Tolerance Vector
                ----------------------------

                Area            =   10.0%
                Perimeter       =   16.7%
                P2A             =   20.0%
                Max_Radius      =    7.5%
                No_holes        =     0

                Change (Y/N)?  Y

                Area = 10.0%
                Change (Y/N/CR)?  Y
                Enter new value:  15.5 @

                Perimeter = 16.7%
                Change (Y/N/CR)?  @
```

THRESHOLD

This command is used for manual selection of the lower and upper thresholds.

```
example:        THRESHOLD @

                Lower threshold:     63
                Upper threshold:    255
                Change (Y/N)?         Y

                Type new values     (0 to 255) ...
                Lower threshold:     66 @
                Upper threshold:    198 @
```

VAL

Control is returned to the interface software between the vision and VAL system. The environment is once again that of the VAL system.

VERTEX

The vertex list of any specified object (or hole) in the current frame description is displayed on the line printer.

example: VERTEX @

```
Enter object #:  1 @
Enter hole #:    1 @
```

```
Vertex list for object #1
  Contains 20 points

0 207 200 2
2 206 201 4
4 202 203 5              CFD(1).start = 0
5 200 205 10
10 200 217 31
31 201 218 33
33 201 219 35
35 203 220 37
37 206 223 39
39 211 223 38
38 212 222 36
36 214 221 34
34 216 219 30
30 216 218 27
27 216 216 15
15 217 215 9
9 217 206 6
6 215 204 3
3 213 203 1
1 210 200 0
```

next vertex point address

y - coordinate

x - coordinate

vertex point address

```
Vertex list for hole #1 in object #1
  Contains 20 points

7 206 205 8                CFD( CFD(1).hole_ptr ).start = 7
8 210 205 12
12 212 206 14
14 213 207 17
17 213 208 19
19 214 209 21
21 214 214 25
25 213 215 26
26 213 216 29
29 212 217 32
32 210 218 28
28 205 218 24
24 203 216 23
23 203 214 22
22 202 213 20
20 202 210 18
18 203 209 16
16 203 208 13
13 204 207 11
11 204 206 7
```

4.3 Robot/Vision Interface Package

Although a vision system is commercially available for use with the PUMA robot [4.11], this is not compatible with our version of the VAL operating system. The objective has been to implement an interface between our vision system and the robot without alteration to the existing VAL software. This has been achieved with a communication link over the serial (9600 Baud, RS-232) line which is normally used for the CRT connection directly to the PUMA controller. As shown in figure 1.3, the vision system computer is inserted in this line, and the robot/vision interface software is used to "complete" the circuit. On power-up, this software link is completely transparent to the user and all the VAL system commands are available as described in [4.1]. However, since the vision CPU now handles every character being transmitted in both directions between the CRT/Keyboard and the PUMA controller, it is possible to extend the instruction set of the VAL language by recognizing new sequences of ASCII symbols. Extensions to both the VAL monitor and VAL program instructions have been implemented; these are described briefly, followed by the calibration procedure between vision and robot coordinate frames and an illustrative parts sorting example.

New VAL Monitor Commands:

MON86 – gives access to a monitor debugging package which is used for manipulating the memory, ports and registers of the 86/12A microcomputer. This was included for diagnostic and development work.

VISION – gives access to the command set of the vision system.

PRINT – allows the output of any VAL command to be displayed on the line printer connected to the vision computer. This facility can be used for obtaining hardcopy of VAL programs, location variables, disk directories, etc. When prompted, any valid VAL command is entered in the print command and any output which would normally appear on the CRT is now displayed on the line printer.

```
example:    .PRINT  @
             VAL Command?  LISTP PALLET @
             END PRINT
```

The listing of the program "PALLET" is then printed.

↑ - (the upward arrow) is used to automatically issue a "DO READY" sequence to the PUMA to return it to the "ready" position. Since this action is required often, it is convenient to accomplish this with a single keystroke. Similarly, any other keys could be pre-programmed for specific VAL commands.

New VAL Program Instructions:

The vision system is treated as a peripheral device which supplies gripper location and orientation information to the robot. As such, the VAL system must be capable of controlling the vision system functions and requesting information under program control. This is accomplished by placing certain commands in a VAL "TYPE" statement, and preceding them with a "$" symbol to indicate that it is a vision command rather than a character string to be output to the CRT (a "PAUSE" command could also be used). All commands automatically stop the robot, execute the specified vision function, and finally restart the interrupted robot motion. For this reason a dummy statement must be placed after each vision command at which the VAL program can halt (DELAY 0.0 is suitable).

$LOOK - instructs the vision system to take a new picture and produce the current frame description.

$FIND object$_x$, n - requests that the vision system find a silhouette which matches the description of "object$_x$" in the current frame description. If one is found control passes to the next statement following the "FIND" command. If none is found, control passes to the VAL statement numbered "n". An "object$_x$" prototype must have been previously taught to the vision system.

$LOCPILE n - determines the location of the largest object in the current frame description. If the frame is empty, execution passes to the VAL statement numbered "n".

Robot/Vision Coordinate Calibration

The two-dimensional visual coordinate system with origin 0_v and axes X_v and Y_v is embedded in the three-dimensional frame of the robot system as shown in figure 4.12. The coordinate system of the robot is defined by origin 0_r and axes X_r, Y_r and Z_r. Location and orientation of objects are reported in the X_v, Y_v frame and must therefore be transformed to the X_r, Y_r, Z_r system for object acquisition by the robot. A procedure for calibration between the two coordinate systems is outlined below:

1. With a pointer in the gripper, the robot is moved to 0_v. This point is recorded using the 'HERE OV' command.

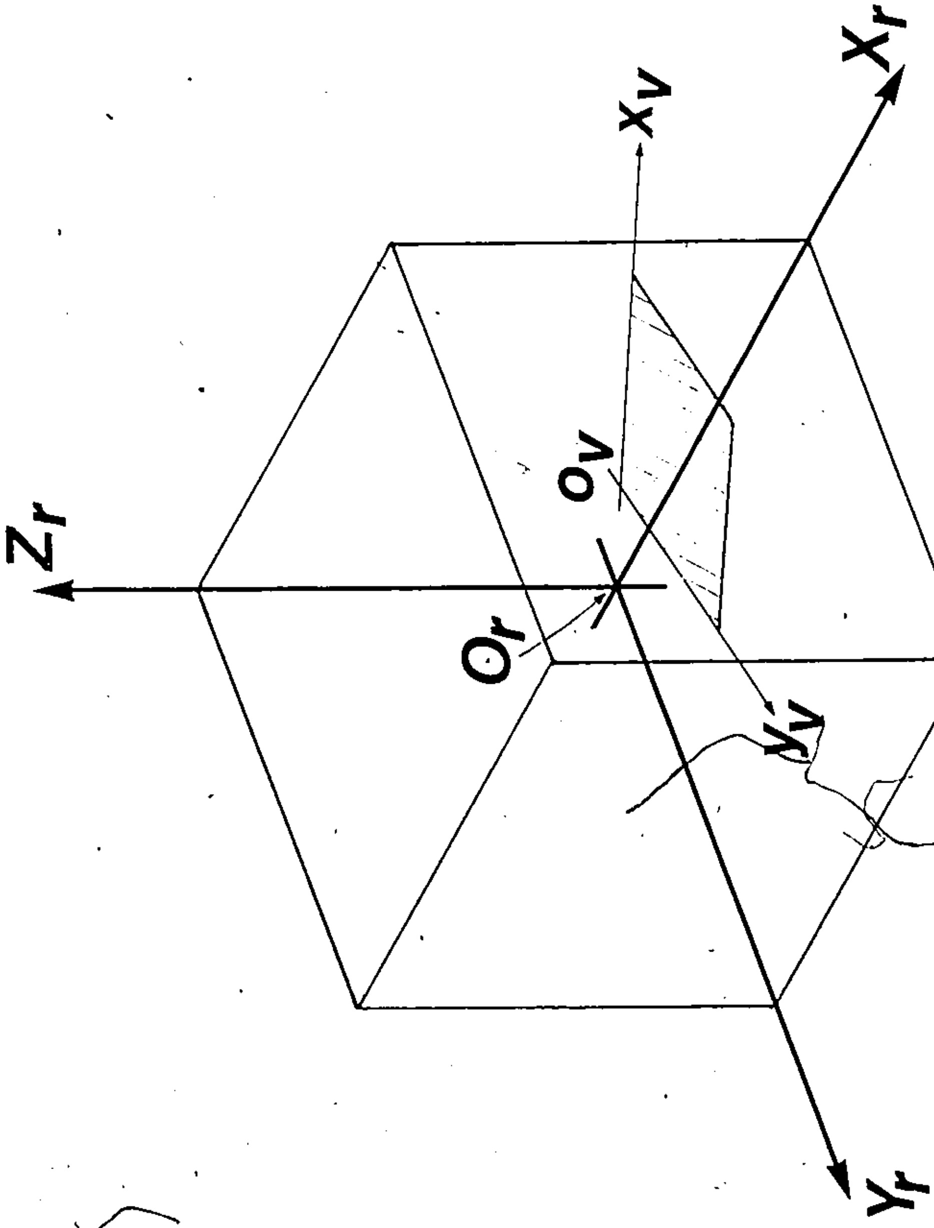


Figure 4.12 Robot and Vision Coordinate Frames.

Similarly one point on X_v (XAXIS), and another point on Y_v (YAXIS) are determined. The plane of the camera coordinate system is then established in robot coordinates by:

DO FRAME CAMERA: OV, XAXIS, YAXIS @

2. Object location and angular orientation of objects in the camera system are reported to VAL in the location variable CENTROID. Therefore, to move the robot to the centroid of an object and to align the gripper with its orientation, the following compound transformation is used:

DO MOVE CAMERA: CENTROID @

3. Pickup points on workpieces are not necessarily the centroidal point and must be determined relative to CAMERA:CENTROID. The gripper is moved manually to the pickup location and recorded using:

HERE CAMERA: CENTROID: ppoint @

The name "ppoint" is the transformation from the centroid to the pickup point and must be unique for each workpiece (or each stable position of one workpiece).

4. When an object is now placed in any random position and orientation within the camera field of view, the vision system determines the variable CENTROID and the object is picked up with the sequence:

```
APPRO CAMERA : CENTROID : ppoint, d
MOVES CAMERA : CENTROID : ppoint
CLOSEI
DEPARTS  d
```

where d is some distance above the object which is large enough to clear it.

As an example of the capability of the system, consider the sorting of two objects (PART1 and PART2) from a random placement of a mixture of the two. The silhouettes of both parts will have been taught to the system previously. The VAL program containing vision commands shown in figure 4.13 will accomplish this task. The program begins at line 4 with the $LOOK command instructing the vision system to acquire a binary image and to construct the current frame description. At line 9, the $FIND command requests the vision system to return the angular orientation and location of the centroid of a PART1 in the location variable CENTROID. When this ~~is~~ completed, the PUMA moves to a location 75 mm above the pickup point of the part which is achieved with the compound transformation on line 14. The pickup point of PART1 relative to its centroid and orientation is specified by the variable PICKUP1 which also has been previously taught. The speed of the robot is then reduced (line 15), the gripper moves to the part along a straight path (line 16), the gripper is closed (line 17) and the object is removed (line 18). Assuming the existence of a suitable subroutine, the part is then dropped in a bin for PART1's (line 22). At line 23, execution is returned to line 4 where a new picture is taken and the process is repeated.

When no more PART1's are visible, the $FIND command on line 9 will cause a branch to line 27 of the program where a similar loop is executed for all visible PART2's. These are sorted into a second bin. When no more PART2's can be seen, the $FIND statement on line 29 causes a branch to line 44. The first executable statement is the $LOCATE command which returns the location of the largest object in view. On the assumption that the unknown silhouette is formed from overlapping PART1's and PART2's, the subroutine BREAKUP of line 49 initiates a sweep of the gripper to attempt to separate the objects. On line 53, control is returned to line 4 where the entire process is repeated. When the frame is empty, the $LOCATE statement of line 47 then causes a branch to line 54 and the procedure halts.

Although this is a rather simple application, it serves to illustrate the flexibility achieved with visual input provided to the robot. Since a new picture is taken before each acquisition, parts can be moved during the operation (the robot itself frequently moves neighbouring parts when approaching the desired object). Of course, the program is easily extended to handle a larger number of parts.

```
.PROGRAM SORT
        REMARK
        REMARK      ACQUIRE AN IMAGE
        REMARK
    100 TYPE $LOOK
        DELAY 0.00
        REMARK
        REMARK      LOOK FOR A "PART1" ...
        REMARK
        TYPE $FIND PART1,300
        DELAY 0.00
        REMARK
        REMARK      PICK UP THE PART ...
        REMARK
        APPRO CAMERA:CENTROID:PICKUP1, 75.00
        SPEED 20.00
        MOVES CAMERA:CENTROID:PICKUP1
        CLOSEI 0.00
        DEPARTS 75.00
        REMARK
        REMARK      CALL ROUTINE FOR DROPPING PART IN FIRST BOX ...
        REMARK
        GOSUB PUTINBOX1
        GOTO 100
        REMARK
        REMARK      NO MORE "PART1" ... LOOK FOR A "PART2" ...
        REMARK
    300 TYPE $LOOK
        DELAY 0.00
        TYPE $FIND PART2,600
        DELAY 0.00
        REMARK
        REMARK      PICK UP THE PART ...
        REMARK
        APPRO CAMERA:CENTROID:PICKUP2, 75.00
        SPEED 20.00
        MOVES CAMERA:CENTROID:PICKUP2
        CLOSEI 0.00
        DEPARTS 75.00
        REMARK
        REMARK        CALL ROUTINE FOR PUTTING PART IN A SECOND BOX ...
        REMARK
        GOSUB PUTINBOX2
        GOTO 300
    600 REMARK
        REMARK       NO MORE KNOWN OBJECTS ... TRY TO BREAK UP PILE ...
        REMARK
        TYPE $LOCATE 400
        DELAY 0.00
        GOSUB BREAKUP
        REMARK
        REMARK       GO BACK AND LOOK FOR MORE "PART1" AND "PART2"
        REMARK
        GOTO 100
    400 REMARK
        REMARK       FRAME EMPTY ... END OF SORTING ...
        REMARK
        READY
.END
```

PART1

PART2

BREAKUP

Figure 4.13   VAL Program for Sorting Operation.

CHAPTER 5

EVALUATION AND TESTING

The performance of a silhouette vision system may be evaluated by considering its capability in three major areas:

a) scene complexity - workpieces are required to have a small number of stable states and must be either stationary or moving in well-defined directions at known speed. A binary image must be readily extracted by enhancing the contrast between object and background and workpieces must be separated and not overlapping. Individual objects are segregated from the scene by connectivity analysis routines and these must be capable of handling a large variety of combinations of intertwined and nested objects and holes.

b) processing time - the time required for the system to perform the connectivity analysis and to determine workpiece identity and/or location is dependent on the scene complexity and the choice of feature set.

c) accuracy - the accuracy of the system in determining the identity and location of workpieces must be measured. Sources of error include:

1. motion of camera or objects
2. parallax
3. lighting and threshold selection
4. lens focus and distortion
5. sensor alignment and uniformity
6. quantization error

The influence of the first three factors has been greatly reduced by ensuring proper fixture of the camera and objects, using low-profile test objects, providing uniform back lighting for high contrast images and maintaining a constant threshold throughout the testing. The dominant sources of error can then be attributed to the final three factors.

The non-uniformity which exists in the frame is illustrated in figure 5.1. The curves were obtained by systematically moving a small disc occupying approximately 2% of the frame area throughout the field of view and plotting the area measurements at over 300 locations.

To facilitate performance comparison of binary vision systems, Rosen and Gleason [5.1] have proposed the two sets of standard two-dimensional figures shown in figures 5.2 and 5.3. These are used to measure the processing speed and accuracy of the system ((b) and (c) above) which are then compared to that of a commercial system.

The first set consists of eight discs, 51 mm (2 inches) in diameter with a number of 6.35 mm (1/4 inch) diameter holes ranging from zero to seven. Since only the number of holes varies, they are used to measure the processing times of a system as a function of the number of "blobs" to be analyzed. The second set is a collection of four subsets of objects of different shapes to indicate discrimination capabilities; each figure has a total area of 2027 mm^2 (π square inches) arranged as subsets consisting of one object with a 6.35 mm (1/4 inch) diameter hole and one without.

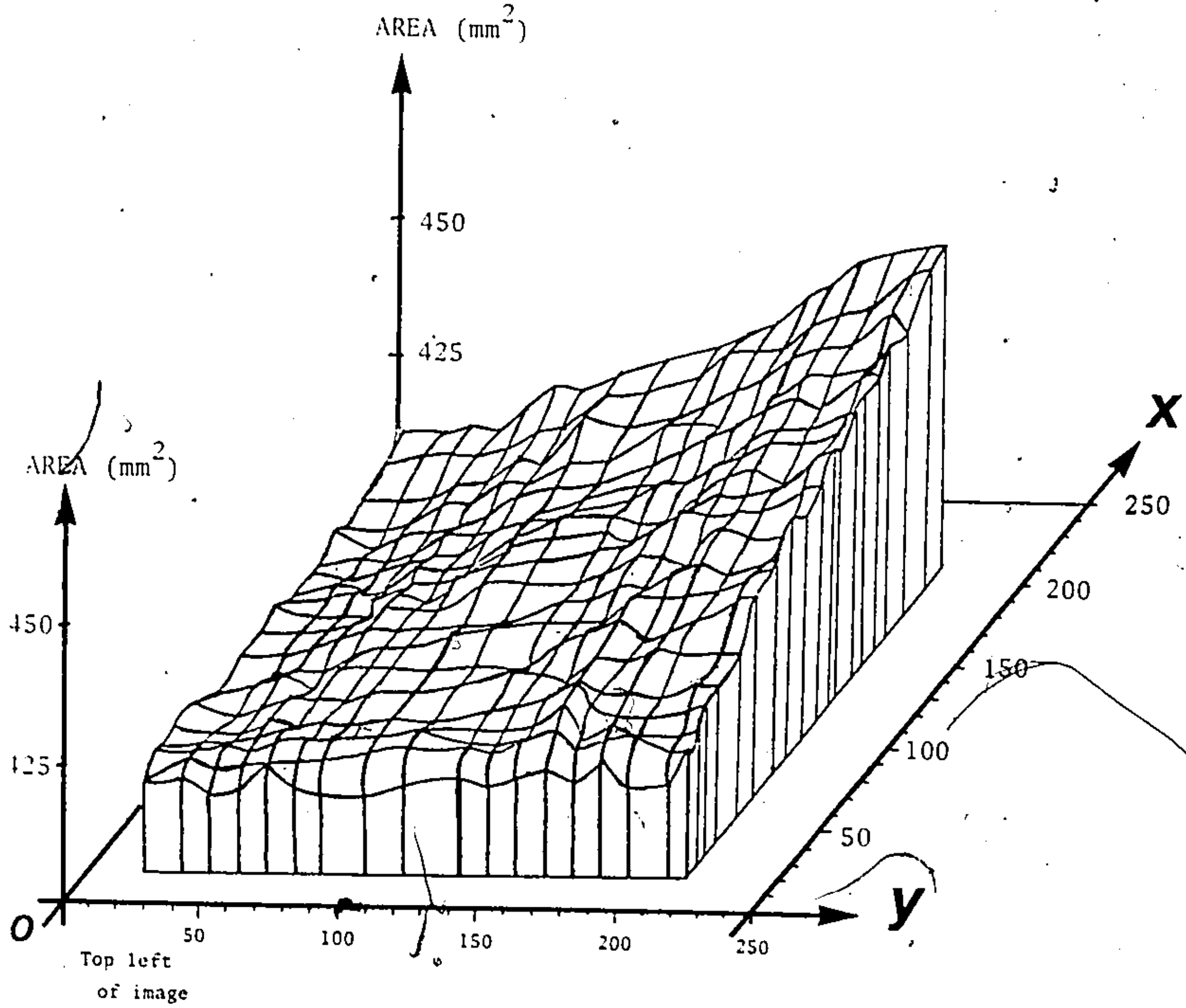


Figure 5.1 Non-uniformity of Area Measurement.

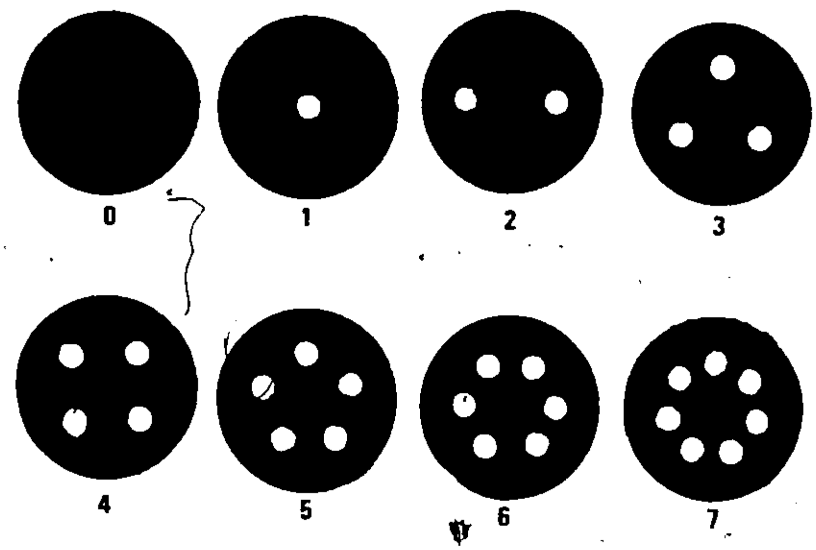


Figure 5.2 Standard Figures for Measuring Execution Times.

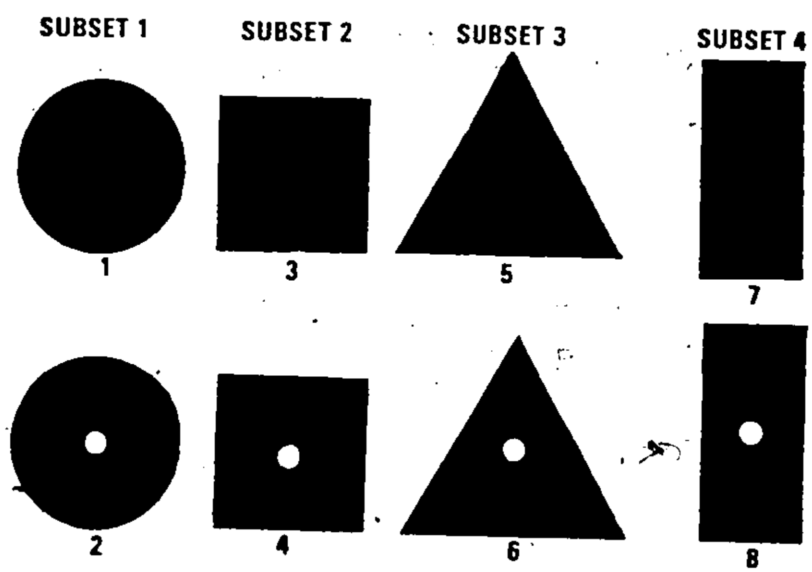


Figure 5.3 Standard Figures for Measuring Discrimination.

A third set of test figures is presented here, as shown in figure 5.4. These have been devised to give an indication of the capability of the vertex points extraction algorithm to handle the connectivity of complex scenes (item (a) above).

The experimental measurements in [5.1] were obtained using a field of view of 102 mm x 102 mm (4 inches x 4 inches). The field size in all cases described here was set to 162 mm x 128 mm and the size of the standard objects of figures 5.1 and 5.2 were increased proportionately. The sensor resolution is 232 x 240 pixels.

In section 5.1, examples of the processing of each of the objects of figure 5.4 are given followed by an examination of the efficiency of the vertex point encoding scheme in section 5.2 and the effects of resolution in 5.3. The overall system performance is measured in the final sections including recognition time and discrimination, centroid and orientation errors.

5.1 Boundary Extraction Examples

The sequential boundary extraction algorithm is capable of handling a wide range of silhouette topologies: objects may be intertwined or convoluted to any degree (as can holes within objects) and nesting is permitted. The seven silhouettes of figure 5.4 include different examples of the various situations that could be encountered. In each of the following examples, a 'creation' is executed at the first occurrence of a line containing a portion of the object; at the first subsequent line containing no portion of the

1. Outer Ring: Object Area = 9744 mm^2

Hole Area = 5735 mm^2

Middle Ring: Object Area = 3536 mm^2

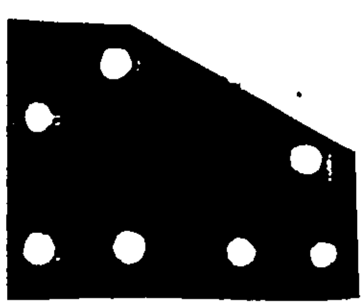
Hole Area = 1535 mm^2

Inner Ring: Object Area = 477 mm^2

Hole Area = 110 mm^2

2. Spiral (i): Object Area = 1807 mm^2

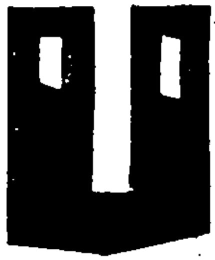
Spiral (ii): Object Area = 1913 mm^2



3. Object Area = 3328 mm^2

Hole Area = 7 x 41 mm^2

Figure 5.4 Test Figures for Demonstrating Connectivity Analysis Capability.



4. Object Area: 1679 mm^2
   Left Hole Area: 28 mm^2
   Right Hole Area: 51 mm^2

5. Object Area: 2184 mm^2
   Hole Area: 495 mm^2

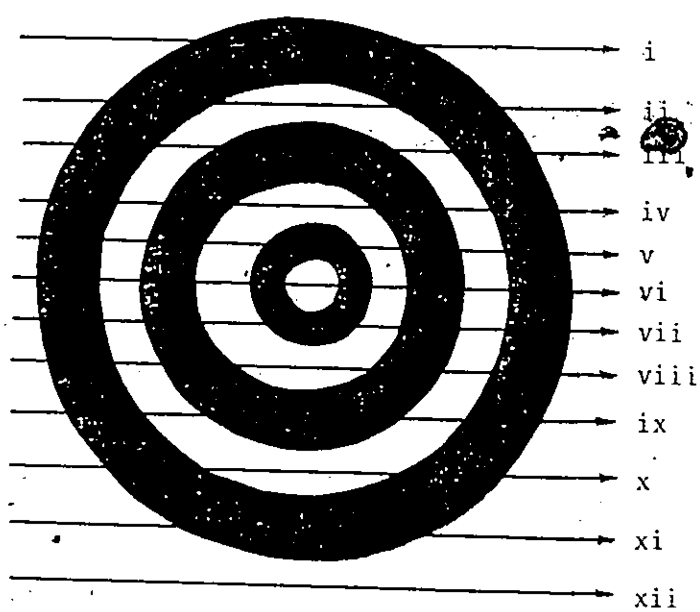
6. Object Area: 1695 mm^2
   Outer Hole Area: 450 mm^2
   Inner Hole Area: 96 mm^2

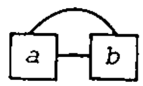
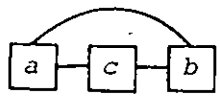
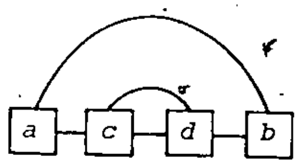
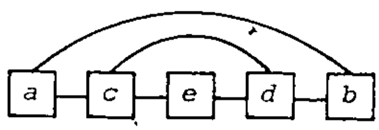
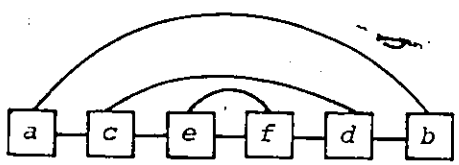
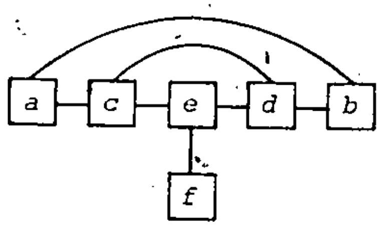
7. Object Area: 1416 mm^2
   Left Hole Area: 41 mm^2
   Center Hole Area: 303 mm^2
   Right Hole Area: 51 mm^2

Figure 5.4 (continued)

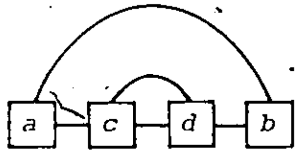
object, processing is completed with a Case 2 termination. At this point, the object node (together with any associated hole nodes in the hole list) is removed from the *active_list* and an entry is made in the *complete_list* structure to point to this node. The progression of the *active_list* data structure is shown for each object at the indicated raster lines. To simplify the diagrams, only the *hole_ptr*, *next* and *forward* links are shown. The *forward/back* links actually form a doubly-linked circular list of connected nodes. For instance, the *forward* link of node *d* (silhouette 5, line iii) points to node *a* and the *back* link of *a* points to *d*.

Silhouette 1: Nested Objects



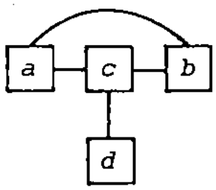
Line	Active list	Comment
i		*a* created.
ii		*a* splits, giving *b*.
iii		*c* created.
iv		*c* splits, giving *d*.
v		*e* created.
vi		*e* splits, giving *f*.
vii		*f* merges with *e*. Case 3 Merge: *f* is a hole in *e*.

viii



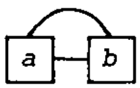
e terminates.
Case 2 Terminate: *e* moved to *complete_list*.

ix



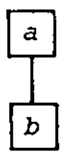
d merges with *c*.
Case 3 Merge: *d* is a hole in *c*.

x



c terminates.
Case 2 Terminate: *c* moved to *complete_list*.

xi

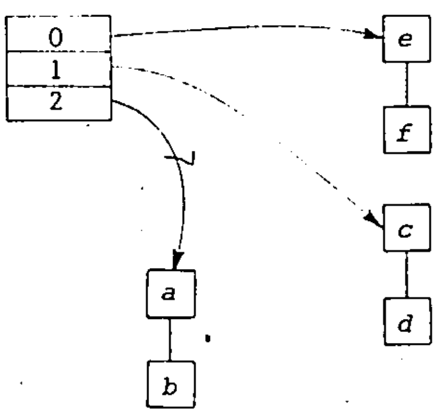


b merges with *a*.
Case 3 Merge: *b* is a hole in *a*.

xii (empty)

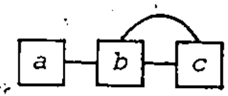
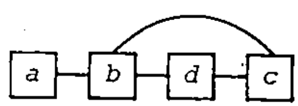
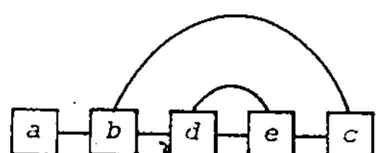
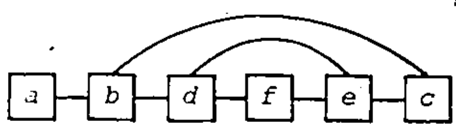
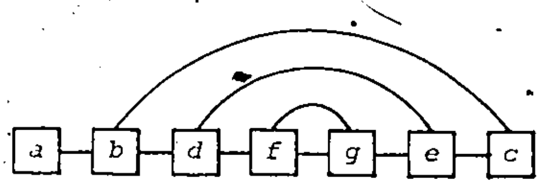
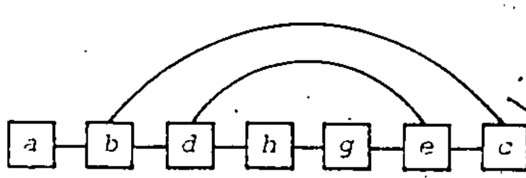
a terminates.
Case 2 Terminate: *a* moved to *complete_list*.

Complete_list:

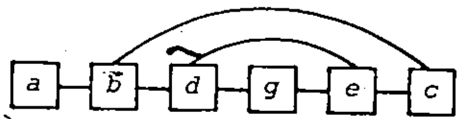


3 objects,
each with 1 hole.

Silhouette 2: Intertwined Objects

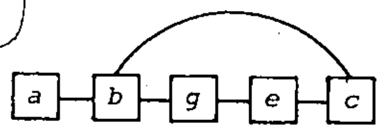
Li	*Active list*	Comment
i		*a* and *b* created.
ii		*b* splits, giving *c*.
iii		*d* created.
iv		*d* splits, giving *e*.
v		*f* created.
vi		*f* splits, giving *g*.
vii		*f* terminates, Case 1 Terminate: object continues in node *g*. *h* created.

viii



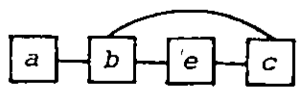
h merges with *d*.
Case 1 Merge: object continues in node *d*.

ix



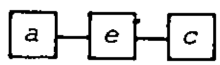
d terminates.
Case 1 Terminate: object continues in node *e*.

x



g merges with *b*.
Case 1 Merge: object continues in node *b*.

xi



b terminates.
Case 1 Terminate: object continues in node *c*.

xii

e merges with *a*.
Case 1 Merge: object continues in node *a*.

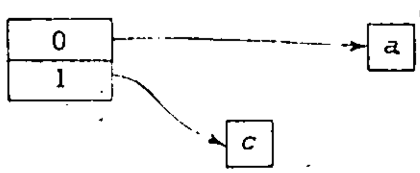
xiii

a terminates.
Case 2 Terminate: *a* moved to *complete_list*.

xiv (empty)

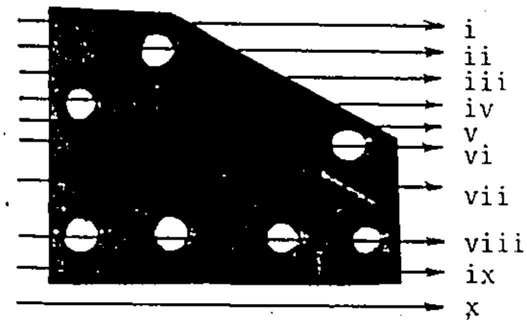
c terminates.
Case 2 Terminate: *c* moved to *complete_list*.

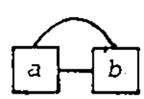
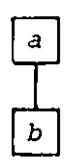
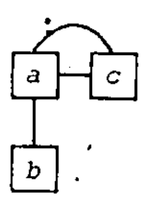
Complete_list



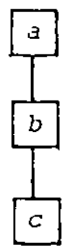
2 objects.

Silhouètte 3: Object with Several Holes



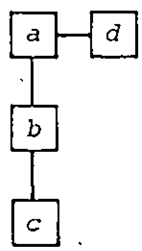
Line	Active list	Comment
i		*a* created.
ii		*a* splits, giving *b*.
iii		*b* merges with *a*. Case 3 Merge: *b* is a hole in *a*.
iv		*a* splits, giving *c*.

v



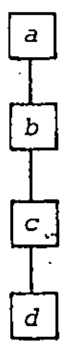
c merges with *a*.
Case 3 Merge:
c is a hole in *a*.

vi



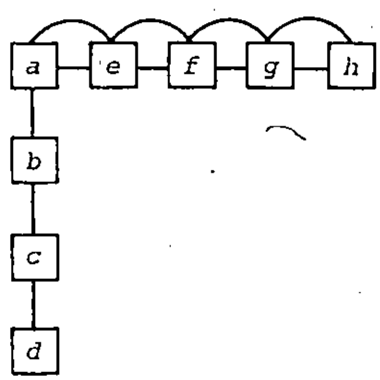
a splits, giving *d*.

vii



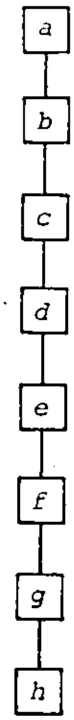
d merges with *a*.
Case 3 Merge:
d is a hole in *a*.

viii



object *a* splits,
giving *e*, *f*, *g*, *h*.

ix



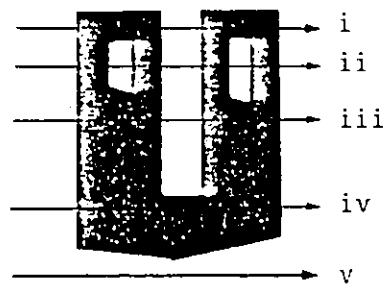
e, *f*, *g*, *h*, merge with *a*.
Case 3 Merges:
e, *f*, *g*, *h* are holes in *a*.

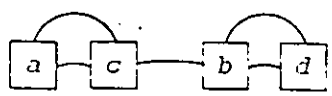
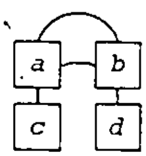
x (empty)

a terminates.
Case 2 Terminate:
a moved to *complete_list*.

Complete_list contains 1 object with 7 holes.

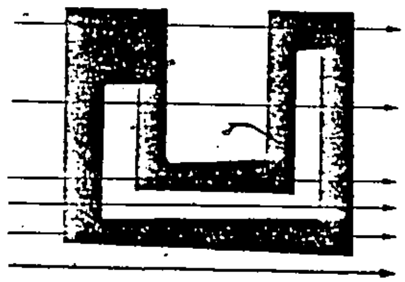
Silhouette 4: Merging of Regions with Holes (Case 2 Merge)



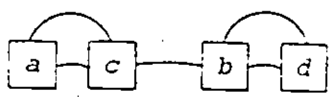
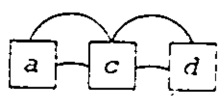
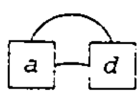
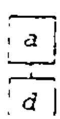
Line	Active list	Comment
i		a, b created
ii		a, b split, giving c, d
iii		c merges with a (Case 3) d merges with b (Case 3)
iv		b merges with a. Case 2 Merge: d is a hole in a.
v	(empty)	a terminates. Case 2 Terminate: a is moved to *complete_list*.

Complete_list contains 1 object with 2 holes.

Silhouette 5: Merging of Regions with Common Hole (Case 1, 3 Merge)

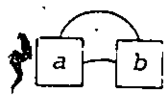
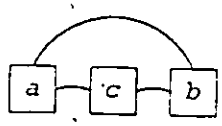
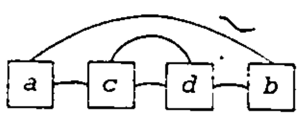
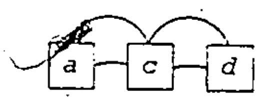
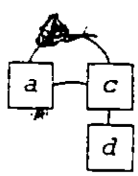


i
ii
iii
iv
v
vi

Line	Active list	Comment
i		*a*, *b* created.
ii		*a*, *b* split, giving *c*, *d*.
iii		*b* merges with *c* Case 1 Merge: object continues in node *c*.
iv		*c* terminates. Case 1 Terminate: object continues in other nodes.
v		*d* merges with *a*. Case 3 Merge: *d* is a hole in *a*.
vi	(empty)	*a* terminates. Case 2 Terminate: *a* is moved to *complete_list*.

Complete_list contains 1 object with 1 hole.

Silhouette 6: Intertwined Holes (Case 1, 3, 4 Merge)

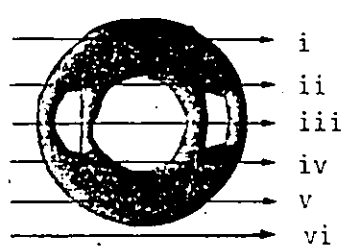
Line	Active list	Comment
i		*a* created.
ii		*a* splits, giving *b*.
iii		*c* created.
iv		*c* splits, giving *d*.
v		*b* merges with *d*. Case 1 Merge: object continues in node *d*.
vi		*d* merges with *c*. Case 3 Merge: *d* is a hole in *c*.

vii	$\boxed{a}$ $\boxed{c}$ $\boxed{d}$	*c* merges with *a*. Case 4 Merge: *c*, *d* are both holes in *a*.
viii	(empty)	*a* terminates. Case 2 Terminate: *a* is moved to *complete_list*.

Complete_list contains 1 object with 2 holes.

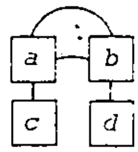
Silhouette 7: Merging of Regions with Holes to Form a Hole

(Case 3, 4 Merges)



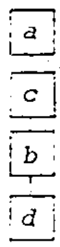
Line	*Active list*	Comment
i	$\boxed{a}$	*a* created.
ii	$\boxed{a}$ — $\boxed{b}$	*a* splits, giving *b*.
iii	$\boxed{a}$ — $\boxed{c}$ — $\boxed{b}$ — $\boxed{d}$	*a*, *b* split, giving *c*, *d*.

iv



c merges with *a*.
Case 3 Merge: *c* is a hole in *a*.
d merges with *b*.
Case 3 Merge: *d* is a hole in *b*.

v



b merges with *a*.
Case 4 Merge: *b* and *d* are holes in *a*.

vi (empty)

a terminates

Case 2 Terminate: *a* is moved to *complete_list*.

Complete_list contains 1 object with 3 holes.

5.2 Vertex Point Encoding Efficiency

The sequential boundary extraction algorithm converts run-length codes (edge points) into the set of vertex points comprising a region. The reduction in the number of points required to represent any contour obtained by the use of vertex point encoding over run-length encoding is expressed by the ratio V:

$$V = \frac{\text{number of vertex points}}{\text{number of edge points}}$$

An example of the vertex point encoding technique is shown in figure 5.5 for the actual binary image of a small circular washer.

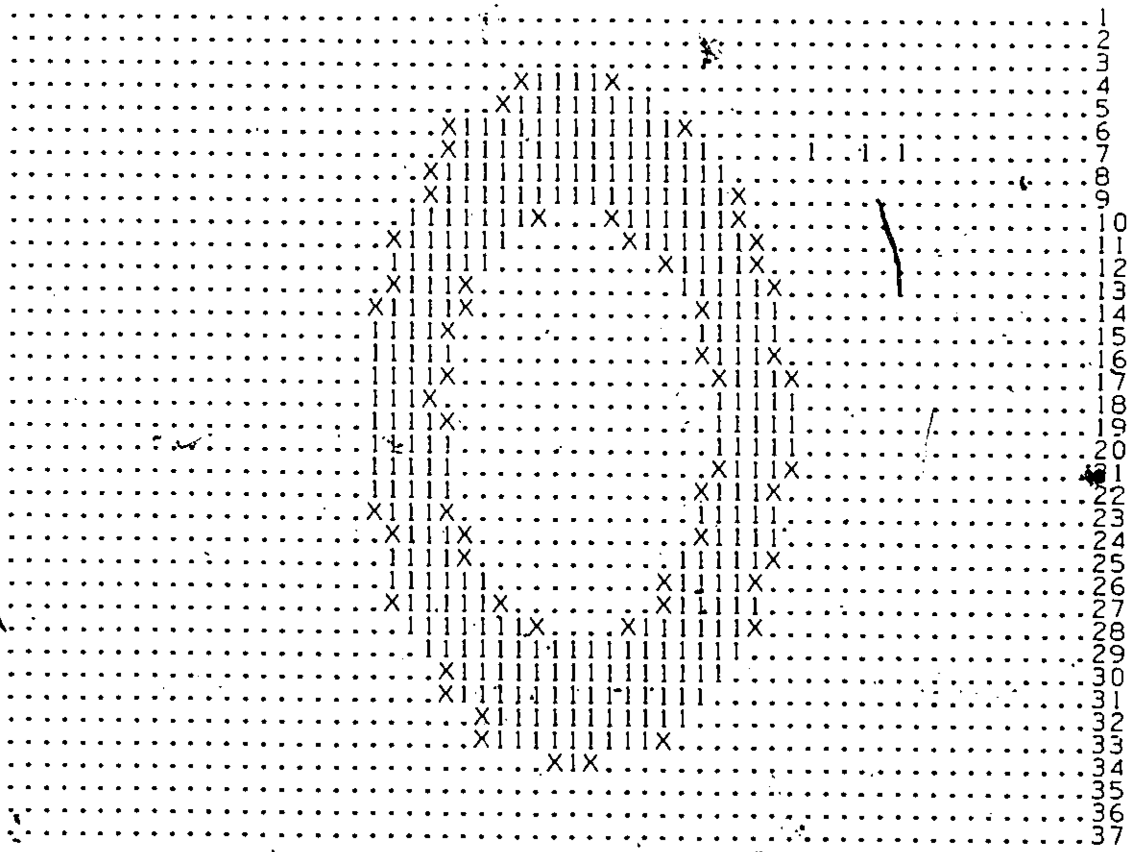


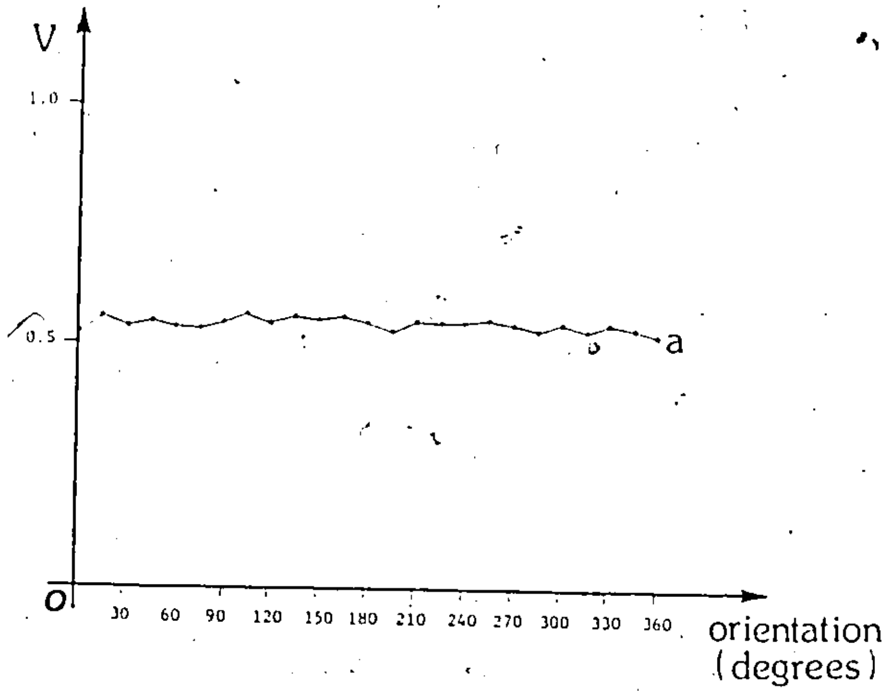
Figure 5.5 Vertex Point Encoding Example.

Background pixels are shown as '.', object pixels as '1' and the vertex points are indicated with an 'x'. The radius of the washer is approximately 12 mm; the radius of the hole is approximately 7 mm (the display is somewhat distorted in the y-direction due to the geometry of the printed characters).

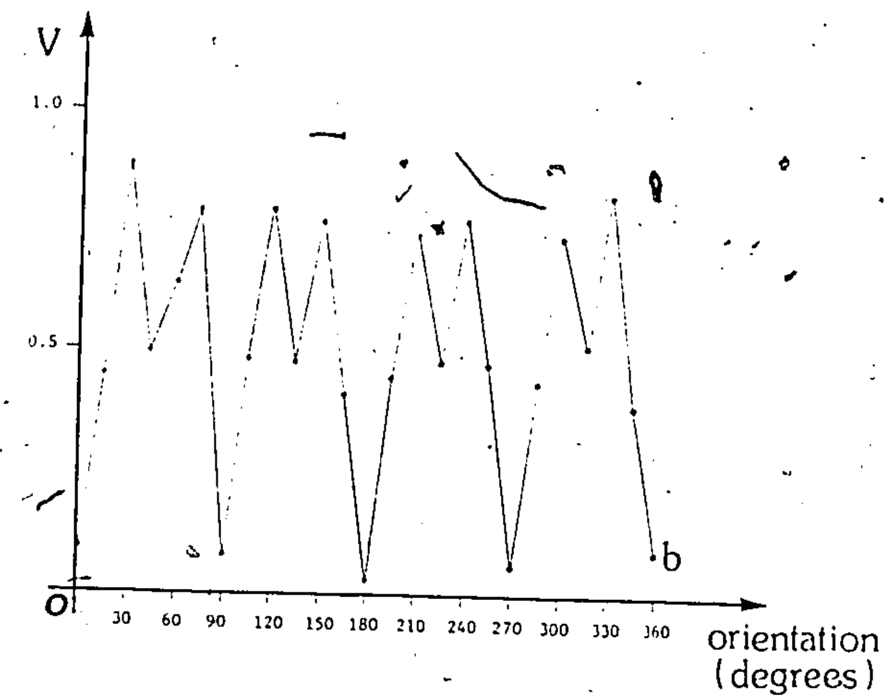
The washer occupies 31 lines of the frame. The external contour therefore is approximated by 62 edge points. Similarly, the hole, which spans 19 lines, comprises 38 edge points. In contrast, vertex point encoding requires 33 points on the external contour and 24 points on the hole contour. The ratio V, for each region is then:

$$V_{external} = \frac{33}{62} = (53\%), \qquad V_{hole} = \frac{24}{38} = (63\%).$$

Smaller values of V are obtained for contours consisting of long straight-line segments such as the triangle, rectangle and square. Ideally the triangle can be represented by three vertex points and the rectangle and square each by four. In practice however, quantization of the image sensor causes the encoding efficiency to be sensitive to angular orientation. This is illustrated in the plots of figure 5.6. The circle, triangle, square and rectangle (objects 1,3,5 and 7 of figure 5.3) were rotated about their centroids in 15° increments and the ratio V was plotted against angle. Reduction of contour points for the circle is nearly independent of its angular orientation and has a mean value of 0.54. For the square and rectangle it is clear that the greatest reduction occurs when they are rectilinearly oriented and vertex point encoding is least efficient for orientations at multiples of 30°. Overall, the mean values of V were found to be 0.56 and 0.55 respectively. The triangle is encoded with least efficiency at multiples of 60° when no side is aligned with one of the frame axes. The mean value of V was found to be 0.56 for this case.

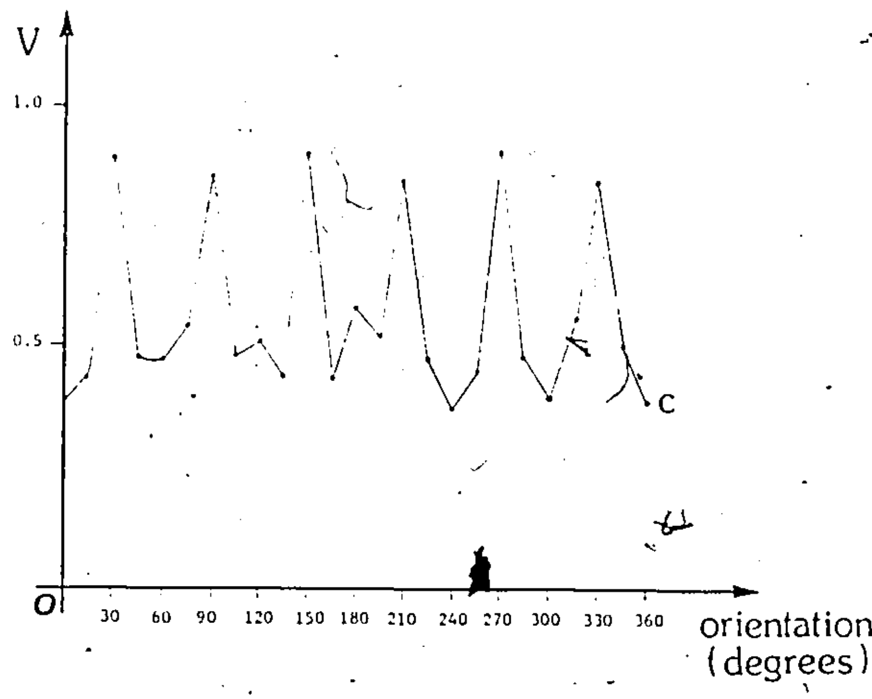


(a) circle (mean = 0.54)

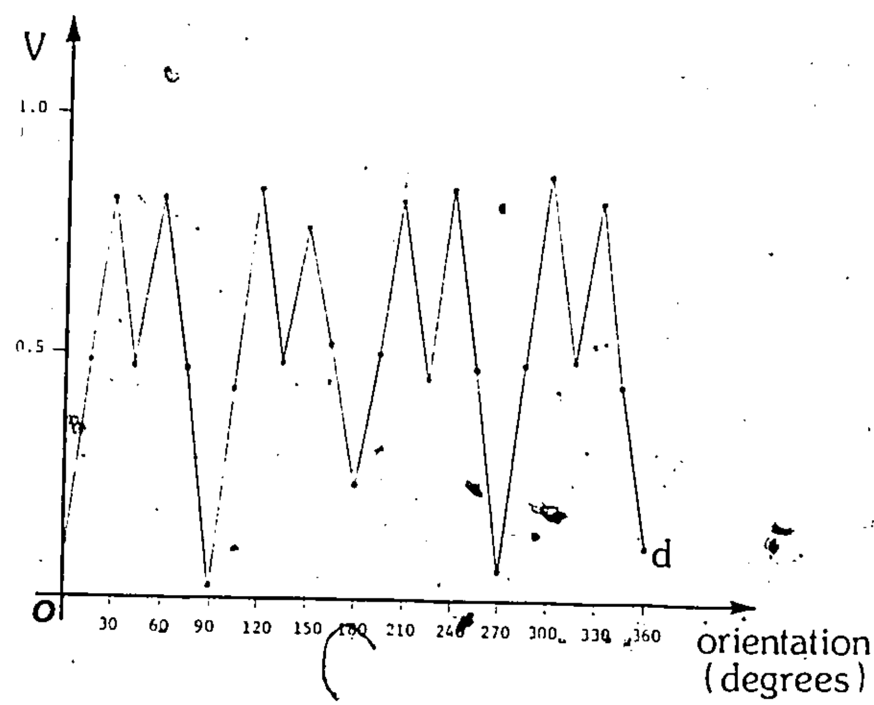


(b) square (mean = 0.56)

Figure 5.6 Vertex Point Encoding Efficiency for Standard Figures.



(c) triangle (mean = 0.56)



(d) 2:1 rectangle (mean = 0.55)

Figure 5.6 (continued)

The values of V were similarly obtained for several circular and square objects occupying various portions of the frame area. In figure 5.7 (a), the mean V for a circle is seen to be nearly constant in the range 0.50 to 0.60 for all sizes. The square is nearly constant in the range 0.45 to 0.50 for large areas (greater than 10% of the frame area), but encoding is less efficient for the smaller sizes where the mean V approaches 0.70. This is depicted in figure 5.7 (b). In figure 5.8, the mean V was obtained for a number of objects of widely different shape as measured by P2A; this is also seen to be entirely within a range of 0.50 to 0.60. On average then, the vertex point representation can be expected to require in the vicinity of 50-60% of the number of edge points.

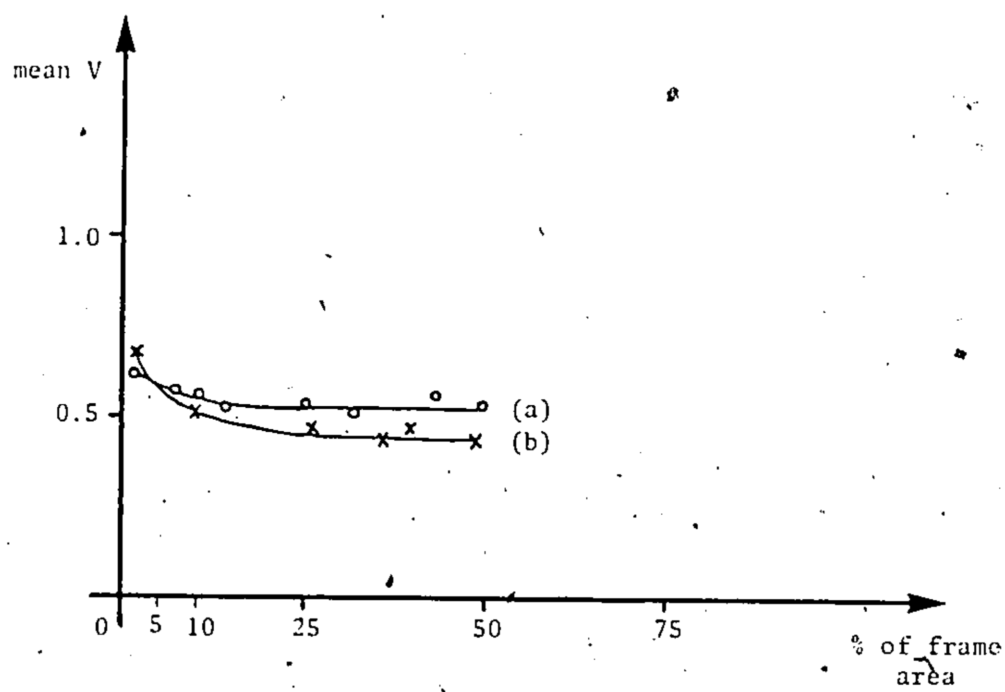


Figure 5.7 Mean V vs. Area.

(a) circles, (b) squares

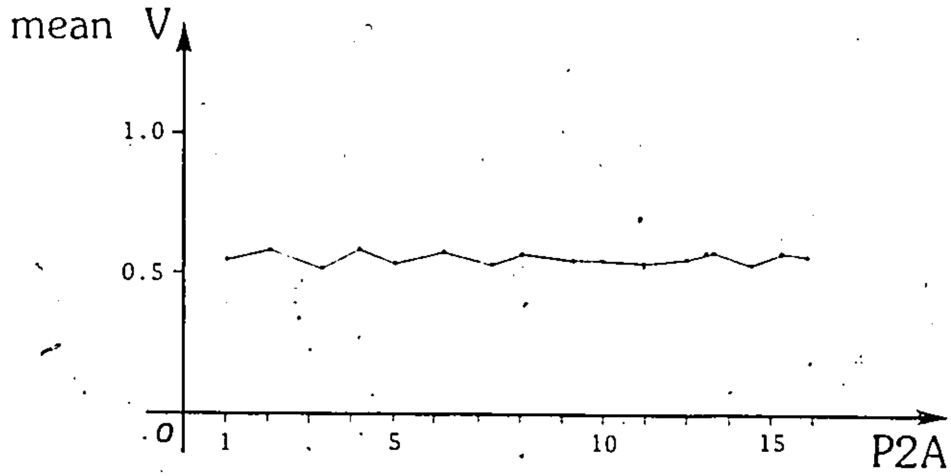


Figure 5.8 Mean V vs. Shape as Measured by P2A.

5.3 Effects of Resolution

In this section, we consider the effect of the image sensor resolution on the contour approximations as created by the vertex point extraction algorithm. Previously, we have shown that the number of vertex points required to represent a contour is sensitive to angular orientation. Although the use of backlighting eliminates problems with shadows, images are still subject to degradation due to dirt particles and reflections. These frequently result in apparent object or hole regions which approach the size of individual pixels and must be handled by the system for reliable operation. In addition, portions of larger objects may contain narrow protrusions, necks or concavities with dimensions near to the image resolution. The resolution can also affect the representation of contours of touching (or nearly touching) objects and these are discussed below. In the accompanying diagrams, vertex points are marked 'x'.

Minimum Resolvable Regions

An isolated object which is quantized as a single pixel is represented as a single entry in the vertex list with a pointer to itself. Similarly, objects quantized as a line of single pixels are approximated by a two-element vertex list representing the end points of the line. These are shown in figure 5.9 (a)and (b). As described in section 2.3, features are obtained from navigation of the vertex lists. For the case of the single pixel object, the resulting area and perimeter are zero and for the line of pixels, the area is found to be zero with a perimeter equal to twice its length. Since calculation of the circularity and centroid require division by these quantities, these cases must be given special consideration. This could be achieved by checking the length of the vertex list; for a single element list the corresponding area and perimeter would be set to one. For a two-element vertex list, the area would be set to the length of the line. In practice however, images can contain 'noise' pixels due to dirt contamination or lighting (see figure 5.5, line 7).

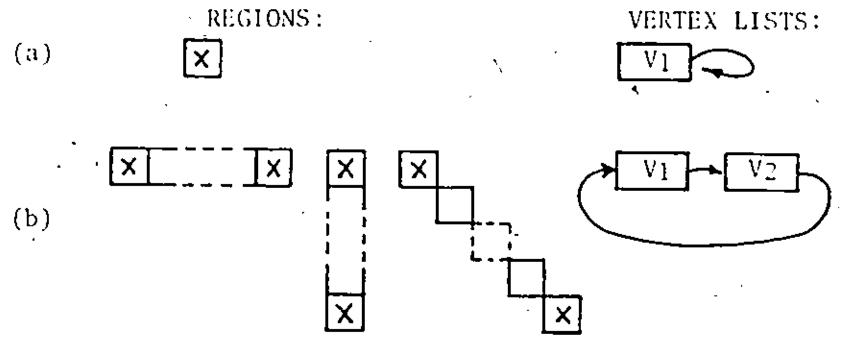


Figure 5.9 Special Cases of Vertex Encoding.
(a) single pixel object region,
(b) line of single pixels.

To avoid the time required to detect and process these cases, the following strategy has been adopted:

(i) object segments of single pixel width are not permitted during the run-length encoding. That is, only segments within an object region which contain two or more pixels are considered valid.

(ii) object or hole regions must contain greater than two vertex points to be included in the Current Frame Description data structure. With (i) above, this means that a contour of less than four vertex points is not possible.

With conditions (i) and (ii) above, the minimum resolvable object and hole regions are shown in figure 5.10 (a) (b). In (b), it is seen that since hole contours are encoded with points from within the surrounding object, a hole with a width of only one pixel can be tolerated. Condition (ii) above, however, does not permit a single pixel or a horizontally oriented line of pixels as a valid hole region.

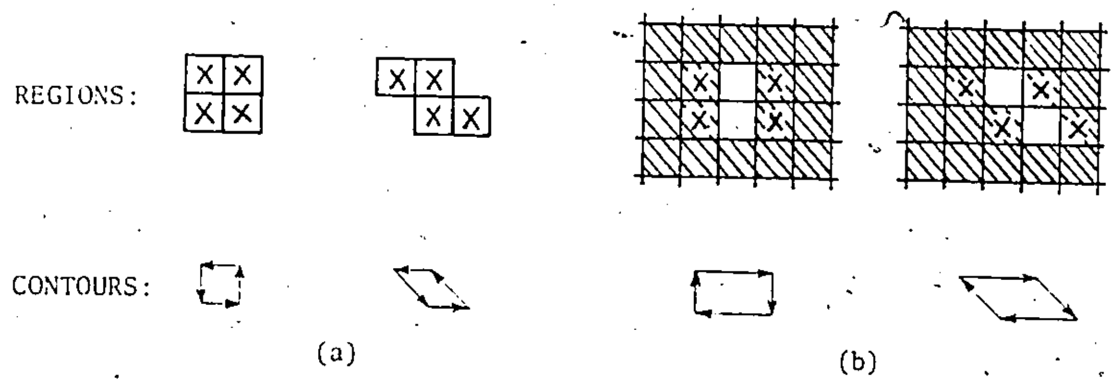


Figure 5.10 Minimum Resolvable Regions (Four Vertex Points).
(a) objects, (b) holes.

Connectivity

The vertex point extraction algorithm generates lists of vertex points by considering segment overlap in the 4-connected sense for object regions and in the 8-connected sense for hole regions. These are illustrated in figure 5.11. In (a) separate vertex lists are formed for objects 1 and 2 which neighbour only at diagonally adjacent pixels, A and B. In figure 5.11 (b), pixels A and B are vertical neighbours resulting in a single vertex list for both objects. In figure 5.11 (c), the vertex points for an 8-connected hole region are shown.

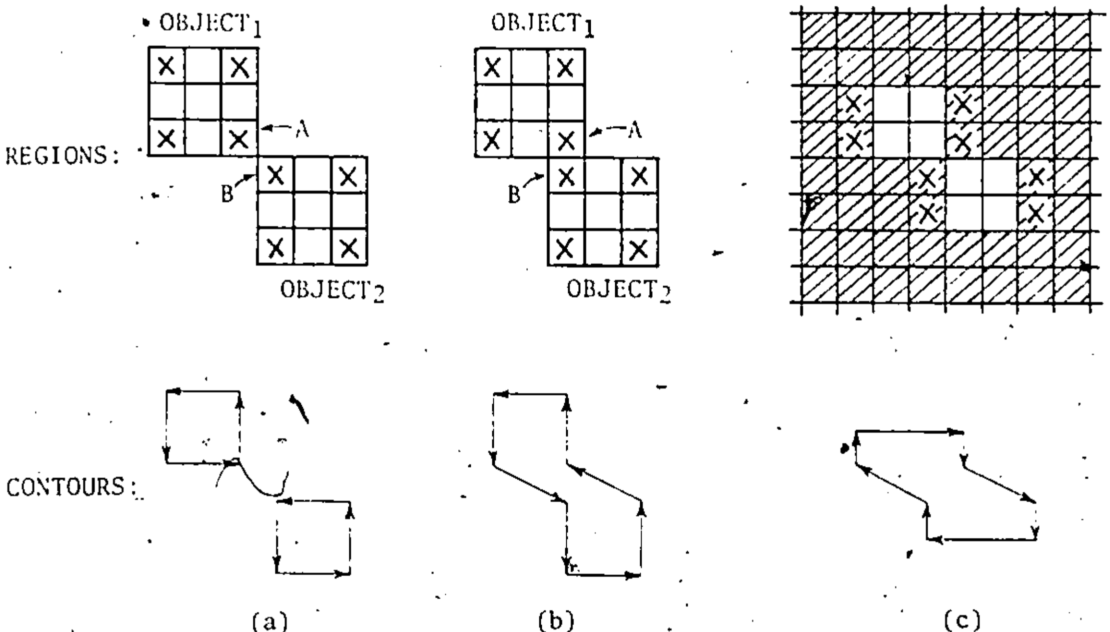


Figure 5.11 Vertex Point Connectivity.
(a) two distinct object regions,
(b) connected object regions,
(c) connected hole regions.

Protrusions/Indentations

The vertex point encoding of single pixel protrusions or indentations on a boundary are sensitive to angular orientation. The end point of a protrusion is included in the vertex list only for horizontal orientation (figure 5.12 (a)). For other orientations, the protrusion will comprise single pixel segments which are eliminated according to (i) above and none of these points will appear in the vertex list (figure 5.12 (b)(c)). Single pixel width indentations are encoded in the vertex list for rectilinear orientations as demonstrated in figure 5.13. The approximation is triangular in shape for a horizontal orientation (figure 5.13 (a)) and rectangular for an exact vertical alignment (figure 5.13 (b)). For other orientations (figure 5.13 (c)) the quantizing effect will result in different representations; in some instances the object may close around the indentation forming one or several narrow hole regions, depending upon the length of the indentation and the orientation angle.

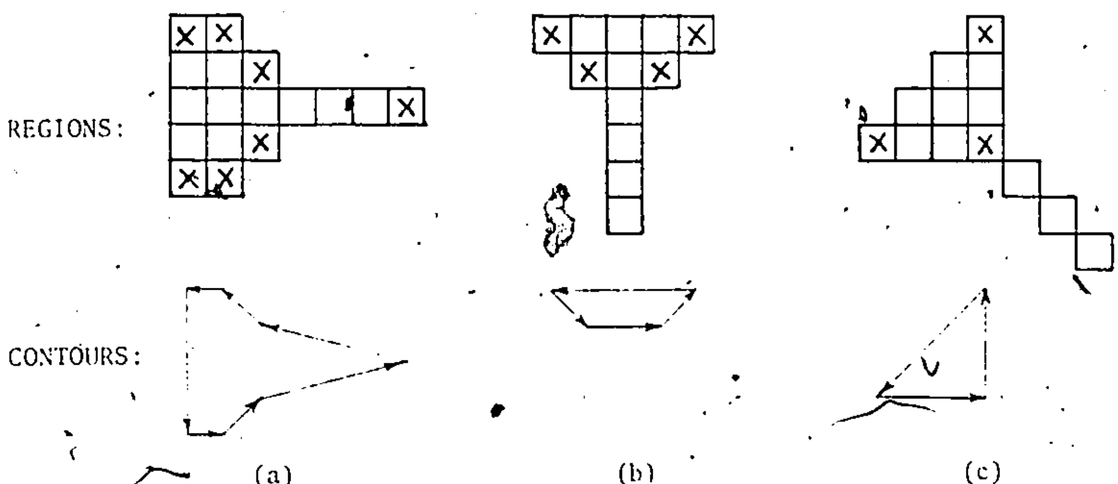


Figure 5.12 Protrusion.

(a) horizontal, (b) vertical, (c) 45 degrees.

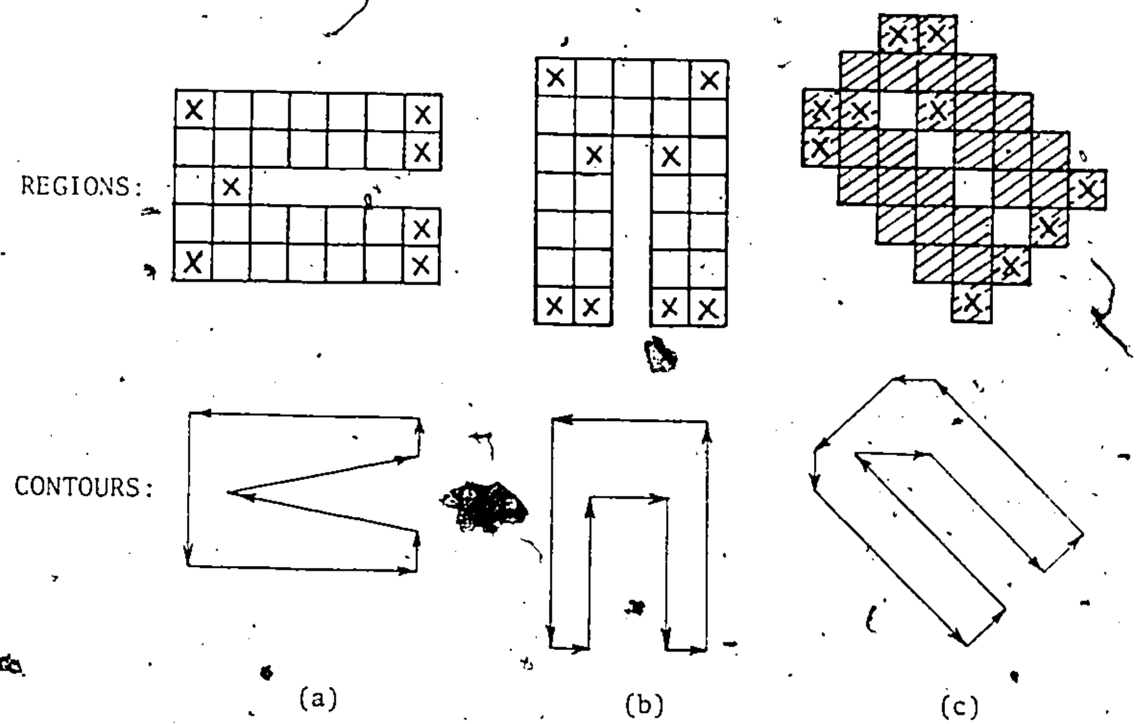


Figure 5.13 Indentation.

(a) horizontal, (b) vertical, (c) 45 degrees.

Necks

The resolution of the sensor can similarly affect objects with small necks connecting two larger areas. The effect is the same as for protrusions described above. For the example of figure 5.14, a single vertex list is formed only for a horizontal orientation (figure 5.14 (a)). In other orientations, the connecting neck is quantized by a single pixel (and therefore ignored) resulting in two separate contours (figure 5.14 (b)).

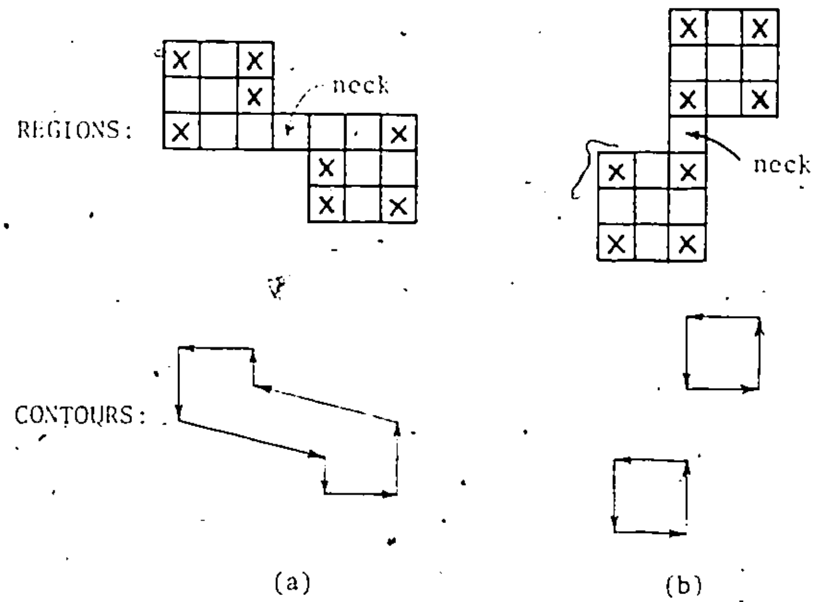


Figure 5.14 Connecting Necks.
(a) horizontal, (b) vertical.

Touching Objects

Overlapping or touching objects generally result in a single vertex list representing one contour. However, in some instances touching objects can be segregated if the point at which they touch is quantized by a single pixel. This can occur when a sharp corner of one object is resting against a straight section of the contour of a second object located above the first. This is demonstrated in figure 5.15. Pixel A, belonging to object 2, is the result of quantizing the sharp corner of object 2 which touches the bottom of object 1. Since this results in a segment of single pixel width, it is not seen by the run-length encoder, and two distinct vertex lists are generated.

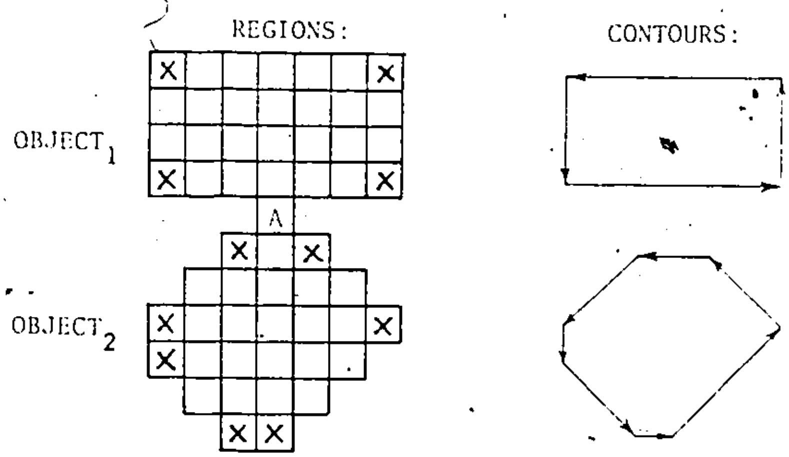
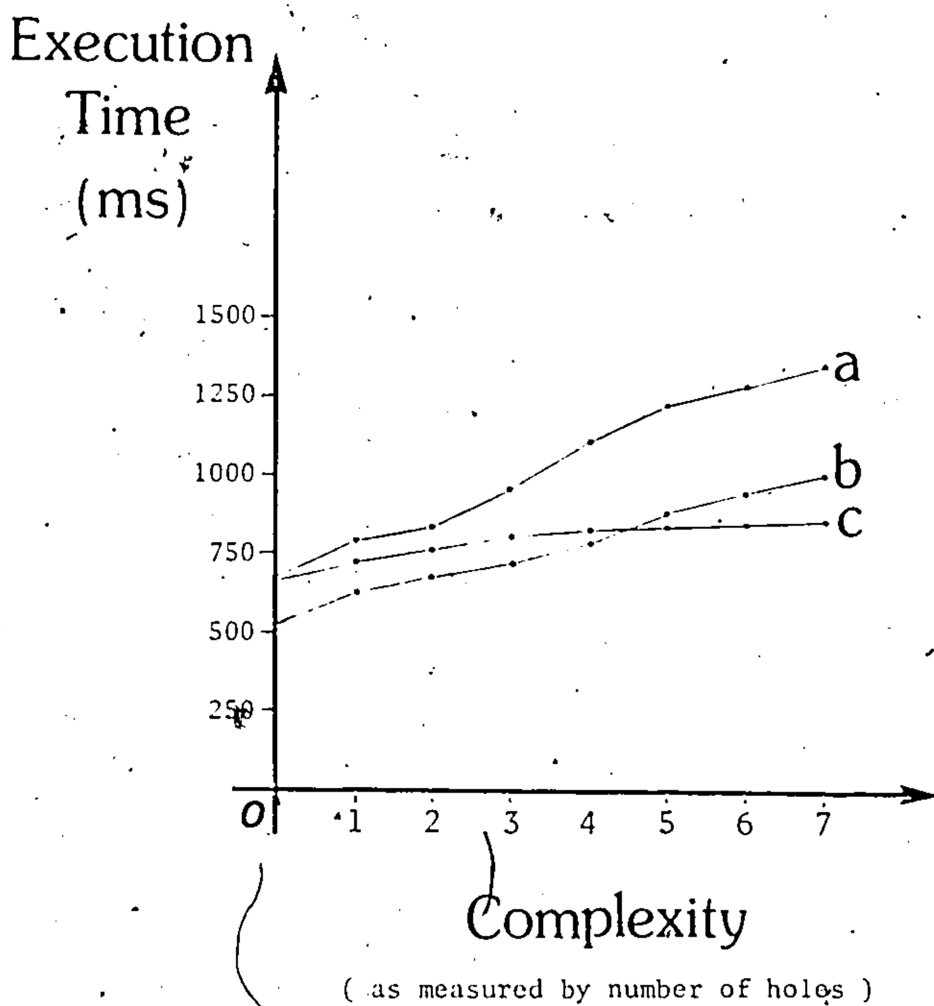


Figure 5.15 Touching Objects which can be Segregated.

5.4 Recognition Time

The processing time for each of the objects of figure 5.2 are plotted in figure 5.16 against complexity, as measured by the number of holes. Curve (a) is the total time required to recognize each disc among the eight in the set and indicates times ranging from approximately 700 ms to 1350 ms. Curve (c) shows the execution times for the same objects for the Machine Intelligence Corporation VS-100 vision system [5.1]. These times, however, are based on the analysis of a 128 x 128 element image; in our system, images contain more than three times as many pixels (the resolution being 232 x 240). The greatest portion of the processing time is that required for contour navigation (Curve (b)) which is directly related to the number of vertex points.



(a) total recognition time

(b) total contour navigation time

(c) Machine Intelligence Corporation VS-100 execution times as reported in [5.1]

Figure 5.16 Processing Time vs. Complexity as Measured by the Number of Holes.

5.5 Discrimination Measurements

An experimental procedure for gauging discriminatory capabilities is proposed in [5.1] and duplicated here. Each of the eight objects of figure 5.3 were processed by the vision system in 40 random positions and orientations within the frame. The measurements of area, perimeter, P2A, maximum radius and hole area at each location were recorded and the means and standard deviations (S.D.) are summarized in table 5.1. By design, the areas of the objects are nearly equal, so that area is not useful in discrimination.

As reported in [5.1], no single feature can be used to characterize individual objects. If a Gaussian distribution is assumed for each feature then this can be graphically displayed as in figure 5.17, using a 99.7% confidence level (3 times the standard deviation).

The maximum radius is clearly able to discriminate among the four subsets of the objects. Together with the hole area (or number of holes) any of the eight objects are readily recognized. The perimeter can distinguish three of the subsets but is not reliable for discriminating between the squares and rectangles. However, the P2A ratio does improve the discrimination between these shapes.

Object	Area (mm^2)		Perimeter (mm)		P2A		Maximum Radius (mm)		Hole Area (mm^2)	
No.	Mean	S.D.	Mean	S.D.	Mean	S.D.	Mean	S.D.	Mean	S.D.
1	4023.1	7.7	227.1	0.8	1.109	0.006	35.6	0.1	0	0
2	4020.4	9.4	226.9	1.4	1.107	0.011	35.8	0.1	71.2	3.1
3	4009.7	12.9	262.5	5.7	1.368	0.060	44.8	0.3	0	0
4	3999.0	11.7	262.0	5.9	1.366	0.062	44.7	0.2	72.8	2.4
5	3977.9	16.9	299.5	6.3	1.795	0.076	55.2	0.2	0	0
6	3974.4	13.6	300.3	5.9	1.820	0.065	55.2	0.3	66.9	3.3
7	4004.1	22.1	276.3	4.2	1.518	0.049	50.4	0.4	0	0
8	4001.8	20.8	275.7	5.6	1.512	0.063	50.3	0.5	70.8	3.8

Table 5.1 Discrimination Measurements

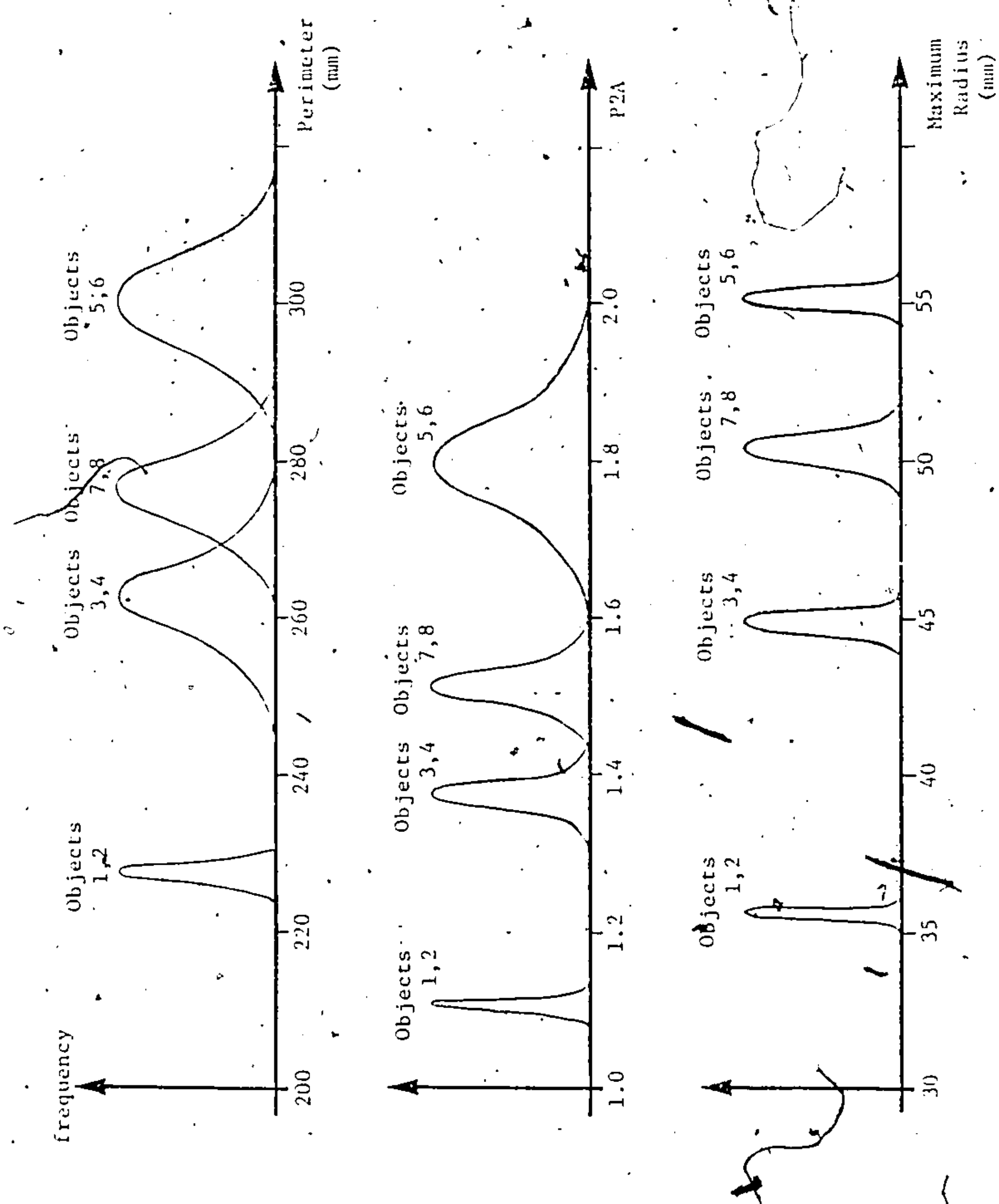


Figure 5.17 Distribution of Features.

In another test using a Monte Carlo procedure, several objects were positioned at 100 random locations within the frame to give a distribution of area measurements [5.2]. A typical histogram is shown in figure 5.18. Four sets of test objects were measured, each set consisting of various size objects of constant shape:

Object	P2A	% of Frame Area Occupied
Circle	1.0	20.5
		10.5
		5.0
		2.2
4:3 Rectangle	1.3	27.0
		19.0
		12.4
		7.4
4:1 Rectangle	2.0	10.7
		6.8
		4.0
		1.7
8:1 Rectangle	3.2	5.4
		1.9

If a Gaussian distribution is assumed in each case, the error in area measurement will fall within ± 2.58 standard deviations with 99% confidence. These errors for the above objects are plotted in figure 5.19. The curves indicate that the accuracy of the area measurement improves as the object occupies larger portions of the frame area. This of course, is expected. Also, the error is seen to be insensitive to variation in shape factor. To illustrate this characteristic further, the experiment was repeated using a triangular-shaped object with P2A = 1.86. The resulting error is included in figure 5.19. The curves therefore, have value in estimating the maximum error in area measurement of objects of random area and shape. For workpiece discrimination based on area alone, the curves can be used to assess the minimum differences in area that are required for reliable operation.

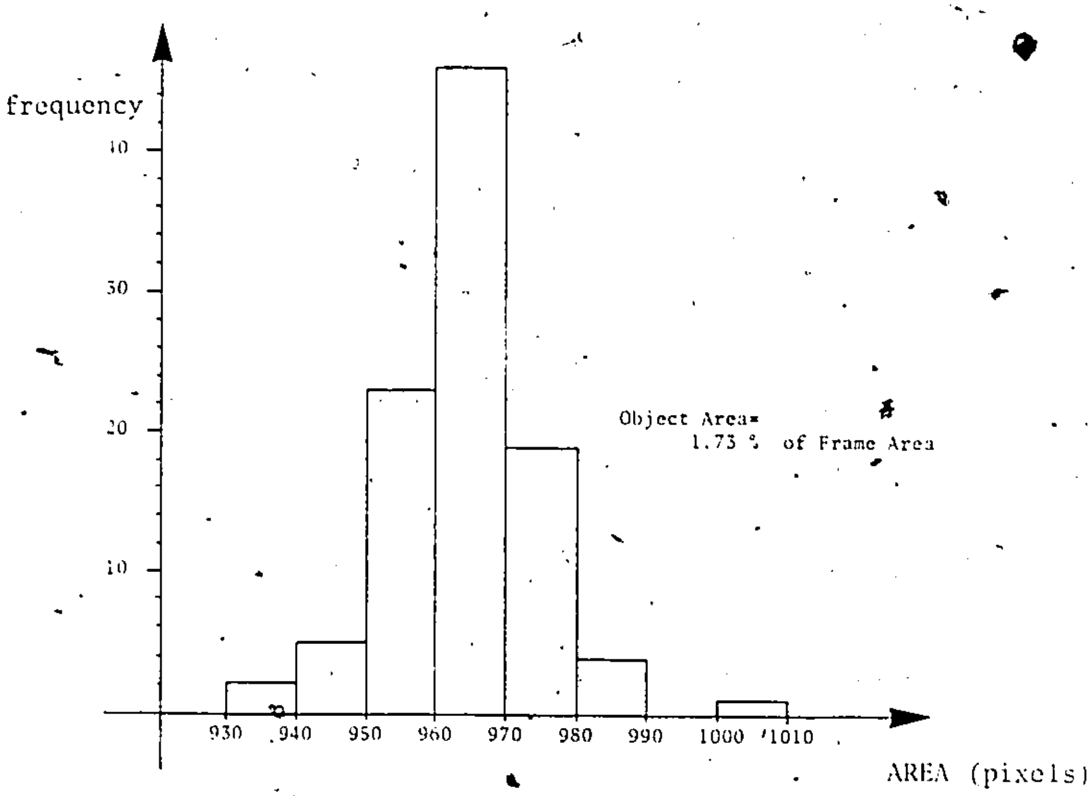


Figure 5.18 Distribution of Area Measurements.

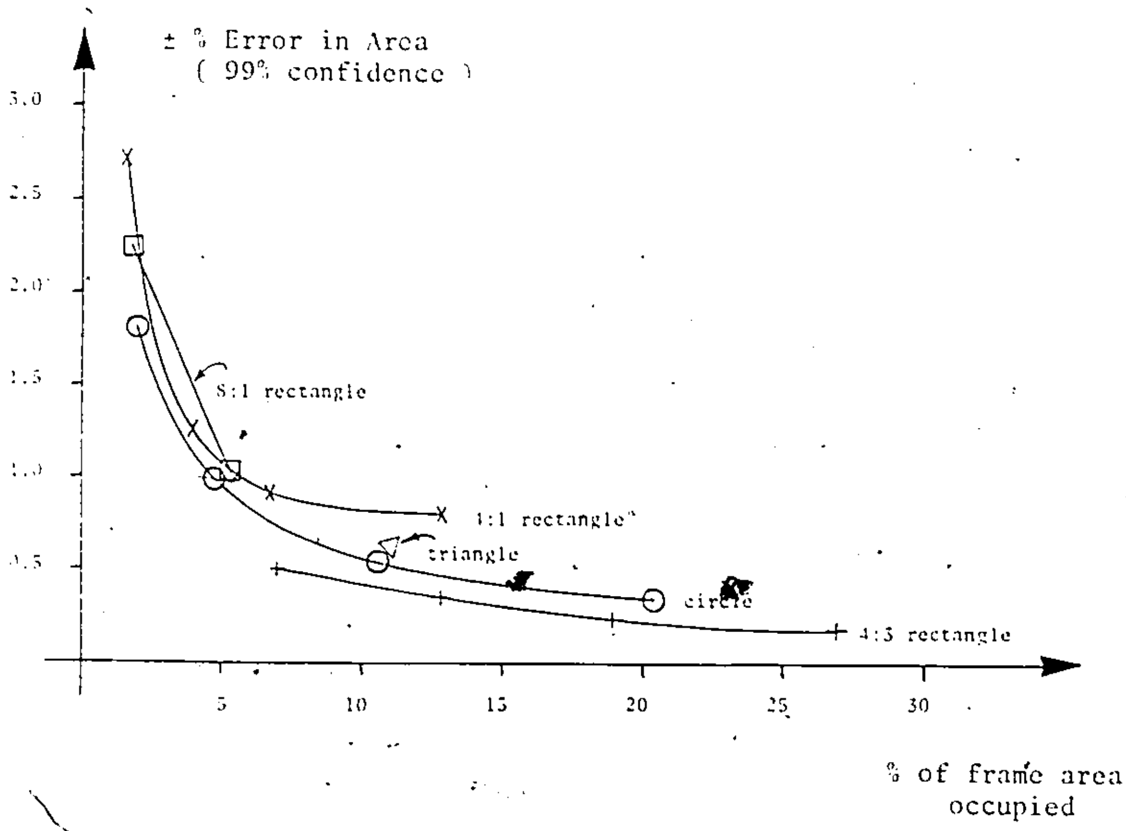


Figure 5.19 Error Curves for Test Objects.

5.6 Centroid Accuracy Measurements

To measure the accuracy of centroid determination of the vision system, 40 readings were made for each of three circular discs. The diameters were 25.5 mm, 44.7 mm and 72.0 mm in the 162 mm x 128 mm field of view. The centroid of each disc was found manually and marked with a pinhole. By positioning the pinhole at several known locations within the frame, the (x,y) coordinates were recorded as reported by the vision system. The mean errors in x and y were calculated and are shown in figure 5.20 with 99.7% confidence limits.

The measured standard deviations are as follows:

Disc diameter	% of frame area	Measured σ_x	Measured σ_y
25.5 mm	2.5	0.55 mm	0.65 mm
44.7 mm	7.6	0.46 mm	0.58 mm
72.0 mm	19.6	0.32 mm	0.36 mm

Errors in the determination of X_c and Y_c improve as the discs occupy larger areas of the frame and the contours therefore comprise a greater number of vertex points.

The error analysis for the accuracy of centroid (X_c,Y_c) determination for run-length encoded circular holes is described by Rosen et. al. [5.3]. They have shown that the worst case errors in X_c and Y_c are given by:

$$\sigma_x = P_x/\sqrt{12} \qquad\qquad \sigma_y = P_y/\sqrt{12}$$

where P_x = spatial resolution on x axis (mm),

and P_y = spatial resolution on y axis (mm).

The quantities P_x and P_y for our system are:

$$P_x = 162 \text{ mm}/232 \text{ pixels} = 0.698 \text{ mm}$$

$$P_y = 128 \text{ mm}/240 \text{ pixels} = 0.533 \text{ mm}$$

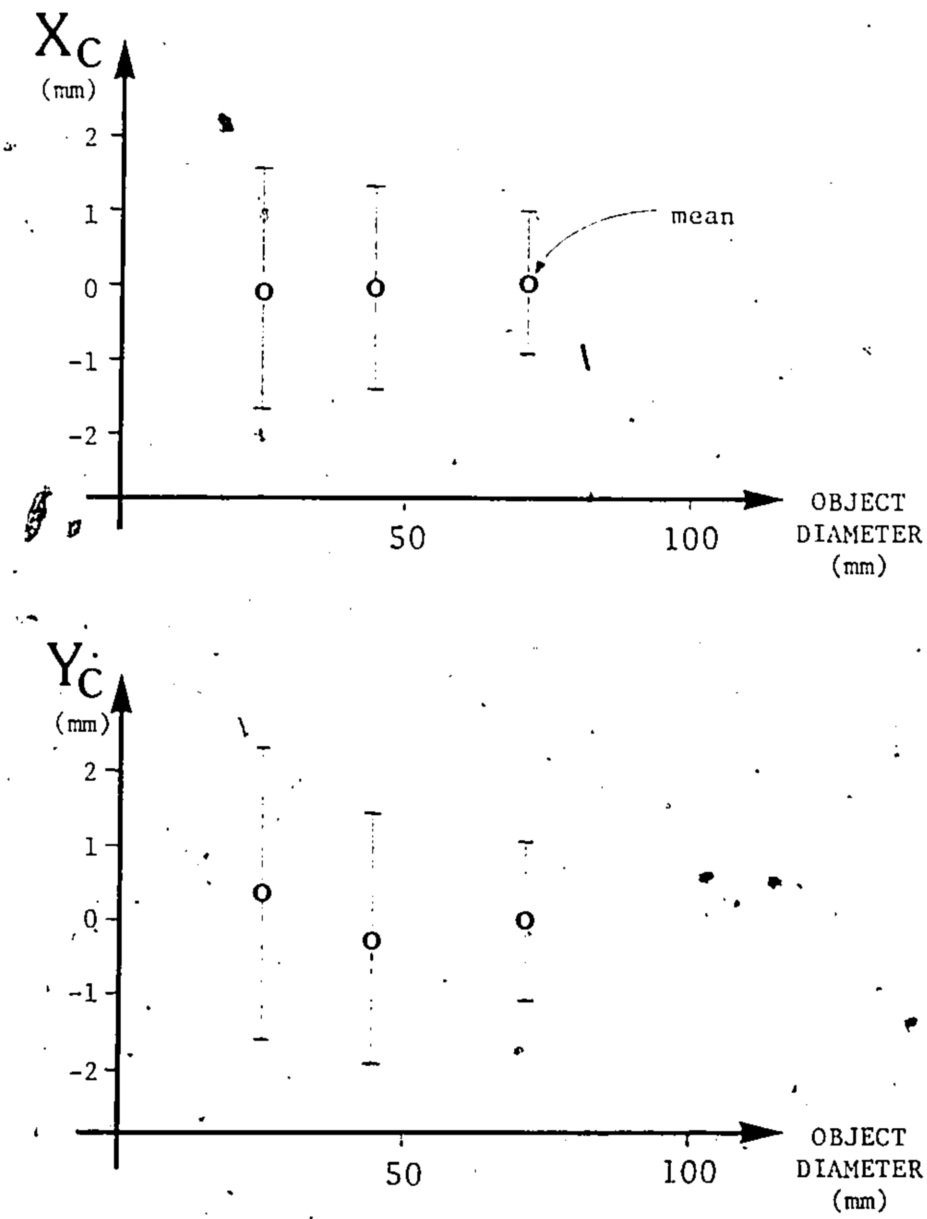


Figure 5.20 Error in Centroid Measurement.

(a) X_c, (b) Y_c

Hence,

$$\sigma_x = 0.201 \text{ mm} \quad \text{and} \quad \sigma_y = 0.154 \text{ mm}.$$

Actual measured errors in X_c and Y_c determined from vertex point approximations are seen to be similar to these values.

In another test, a 50 mm x 50 mm square was rotated about its centroid for five different locations in the frame to indicate the sensitivity of centroid measurement to angular orientation for a non-circular object. The mean X_c and Y_c of 25 readings were calculated for each position shown in figure 5.21. The results are summarized below. The results indicate a maximum error in X_c of 3 x 0.27 = 0.81 mm and in Y_c of 3 x 0.29 = 0.87 mm due to orientation at position 1.

Position	X_c (mm)		Y_c (mm)	
	Mean	S.D.	Mean	S.D.
1	43.45	0.27	37.70	0.29
2	120.67	0.21	38.12	0.17
3	42.57	0.16	94.74	0.20
4	117.21	0.17	93.13	0.18
5	84.84	0.18	65.36	0.18

5.7 Orientation Accuracy Measurements

The triangular object of figure 5.22 was used to give an indication of the accuracy of the angular orientation measurement. The object was rotated about the axis of the maximum radius and 50 measurements recorded.

The mean difference between the angle reported by the vision system and the measured angle was found to be 0.13 degrees with a standard deviation of 0.84 degrees.

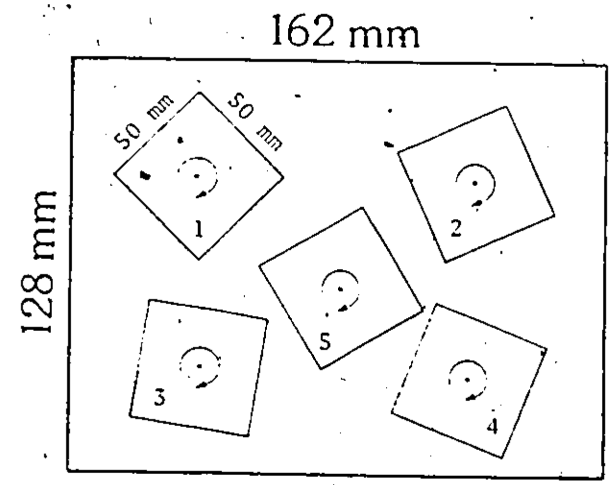


Figure 5.21 Testing Centroid Sensitivity to Angular Orientation.

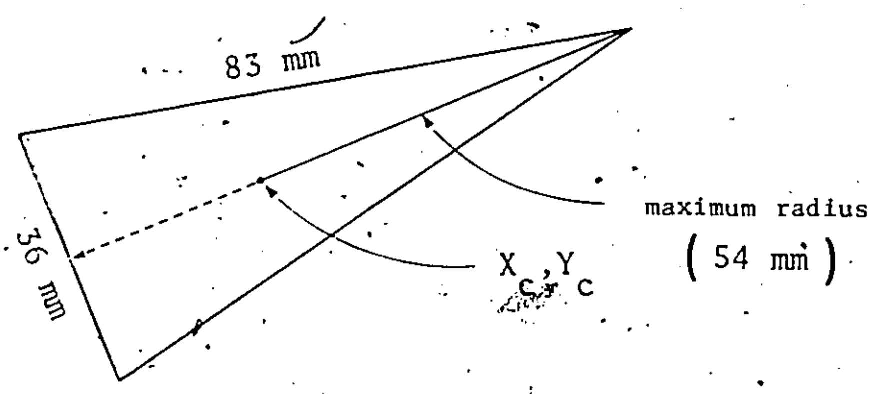


Figure 5.22 Triangle Used for Determining Orientation Accuracy.

Ho [5.4] has shown that the error in orientation determined by the maximum radius vector of length 'h' from the centroid to a sharp corner on the contour is estimated by:

$$\sigma_\theta = (1/\sqrt{3}) \tan^{-1} (1/h)$$

For this example h = 54 mm, giving a value of $\sigma_\theta = 0.61$ degrees which is very near the experimentally measured error.

CHAPTER 6

DISCUSSION

6.1 Conclusions

Machine vision will surely play an increasing role in automated manufacturing with applications identified in two principal categories: inspection and visually-controlled manipulation. Computer vision is appropriate for workpiece sorting operations, identifying or measuring shape, parts fitting or mating, measurement of critical dimensions and tool wear, checking for integrity or completeness and similar quantitative processes. Silhouette vision is particularly well-suited to the processing of inherently binary images (including those of printed text, handwriting, printed circuit traces, etc.), the interpretation of structured light patterns and the analysis of isolated regions of grey-scaled pictures. Processing speeds however, remain a serious limitation in the development of general purpose vision systems. The real-time extraction and interpretation of information contained in large quantities of visual data requires advances in preprocessing hardware, highly parallel machine architectures and efficient algorithms and software.

Towards this end, a new algorithm and data structures have been developed for the sequential extraction of region contours from raster-scanned binary images. Existing methods have been surveyed and three sequential methods were selected for detailed description. The first two of these are based on the well-known chain coding technique and were found to require excessive post processing, precluding their use in a high speed implementation. The third method, due to Aggrawala and Kulkarni, was then compared with our new procedure. At the expense of an increase in memory, the new algorithm is found to require significantly fewer edge point comparisons, and the dynamic data structures can be manipulated more efficiently. In particular, processing time is not dependent on the topology of the contour, no initialization is required at the beginning of every frame and the storage utilized by merged regions is freed for subsequent use. The *complete_list* structure used for representing the contents of a completed frame is constructed 'on the fly'; the hierarchy tree of Aggrawala and Kulkarni is available only after additional processing.

The new algorithm approximates contours as a linked list of vertex points indicating changes of direction on the boundary. Operation is completely sequential and no restrictions are imposed on the configuration of the regions: an image may contain any number of objects and holes which may be intertwined, nested or intersecting frame edges. Scan lines are processed individually so that a linear array sensor could also be used. The vertex point encoding scheme is shown to offer considerable savings over run-length codes in the number of points used for boundary encoding. Measurements indicate that on average, savings of 50 to 60% can be achieved. A set of test silhouettes of widely different shapes was used for demonstrating the operation and robustness of the algorithm and data structures. Removing single pixel width segments and eliminating contours of fewer than three points have proven effective in reducing image contamination due to small dirt particles and reflections. These strategies, however, can affect the vertex point representation of regions containing narrow protrusions, indentations and connecting necks.

A complete implementation of a vision system based on the vertex point technique was undertaken. The architecture, designed with an 8086/8087 single board computer, features a flexible multiple bus structure which supports high speed, parallel operation of hardware modules: these include video rate histogram generation and binary image acquisition. The system is compact, being completely contained on three printed circuit boards including software which has been confined to on-board EPROM. Software has been developed for vertex list navigation for feature extraction, object recognition and the determination of location and angular orientation. A relational data base is used to store the features of prototype objects, entered in a 'teach-by-showing' session. The contents of each frame are efficiently represented in a dynamic data structure (CFD). This structure maintains object/hole relationships and provides an easy method for determining information such as the number of objects in the frame, the largest object (or hole), the object with the most holes, etc. Calibration between pixel and linear dimensions is accomplished by a simple procedure: an object for which the centroid is known is shown to the system in two positions and the actual distance between the centroids is entered. All prototype feature measurements are stored in linear units so that if the camera position were to be changed, the prototypes need not be retaught.

A software communications package has been developed for interfacing the vision system with a PUMA 600 industrial robot. By inserting the vision processor in the serial line normally used for a CRT terminal, the instruction set of the VAL control language has been extended to include commands for accessing visual information under program control. The vision system provides workpiece identification, location and orientation without modification to the robot controller. High reliability and accuracy have been demonstrated in repeated operation of a parts sorting application by the robot under visual control. The 9600 Baud serial communication link was not found to be a limiting factor in the overall speed of the robot/vision system.

The vision system has been evaluated in terms of execution speed and discriminatory capabilities, and for comparison purposes two sets of standard shapes have been used. In execution time measurement, it was found that for increasingly complex objects (as measured by the number of holes), the system required from 700 to 1350 ms as compared to Machine Intelligence Corporation's VS-100 system using 700 to 850 ms over the same range. However, the VS-100 system is processing fewer than one third as many pixels. In discriminatory measurements, nearly identical results were found to those reported for the VS-100 system.

Measurement of the errors in centroid and orientation determination indicate that vertex point approximation of contours does not significantly affect these values. In both cases, measured errors were comparable to theoretical limits. Centroid accuracy was found to improve as objects occupied larger portions of the frame area and therefore comprised a greater number of vertex points.

6.2 Future Work

Preliminary work has already begun in the design of an implementation of the sequential boundary extraction algorithm during horizontal retrace intervals of the raster scan. The use of the Texas Instruments TMS 320 and the AMD 29116 processors are being considered to attain video-rate execution. Additional hardware modules will be incorporated in the system architecture for high-speed, parallel extraction of contour features from vertex lists. Replacing the 8086-based computer with the recently introduced Intel 80286 processor will also offer improvements in performance.

Further algorithms could be developed for processing vertex points to optimize recognition and orientation determination for shapes in particular applications. These include the recognition of overlapping or touching workpieces, or the detection of workpieces in close proximity which might result in a robot crashing into surrounding objects while attempting to grasp another. Additional measurements of system accuracy could lead to methods of compensating for focus, alignment and quantization errors.

Multiple-camera arrangements can be utilized for extending the two-dimensional techniques to three dimensions. For example, one camera would give a plan view of workpieces with another, horizontally positioned sensor for estimating object heights. Alternatively, the camera could be repositioned by the robot itself to collect a variety of views. This will be facilitated by the availability of smaller image sensors which would not interfere with the dynamics of the robot.

Improvements can be expected with the evolution of sensor technology and as standard software and computer architectures specifically designed for image processing tasks are established.

APPENDIX A

VISION SYSTEM SPECIFICATIONS

Hardware Features

- TV Camera
  - General Electric TN2500
  - 30 Hz frame rate
  - 4.5045 MHz pixel rate
  - 8-bit grey-scale for each of 244x248 pixels
  - RS 170 output

- Processor
  - Intel iAPX 86/20
  - 8086 microprocessor (16 bit, 5 MHz)
  - 8087 numeric data processor
  - 64K RAM
  - 32K EPROM
  - MULTIBUS interface
  - RS232C serial interface (CRT terminal)

- Input/Output
  - Video-rate binary image acquisition
  - Video-rate histogram generation
  - Robot interface (Serial, 9600 Baud)
  - DMA controller
  - TV monitor for displaying camera images

Software Features

- data base storage for 75 silhouette prototypes
- images may contain any number of objects or holes of any shape
- prototype objects entered in "teach-by-showing" session
- feature tolerance user-settable
- fast vertex point extraction
- feature measurements stored in "real world" units; selection of a different camera lens or position does not require reteaching of object prototypes.
- grey-scale to binary image conversion: automatic or manual
- programmable background color

APRENDIX B

PL/M-86 LISTING OF

VERTEX POINT EXTRACTION ALGORITHM

ISIS-II PL/M-86 V2.1 COMPILATION OF MODULE VERTEX
OBJECT MODULE PLACED IN :F1:VERTEX.OBJ
COMPILER INVOKED BY: PLM86 :F1:VERTEX.SRC LARGE OPTIMIZE(3)

```
 1          VERTEX: DO;

           /*-----------------------------------------------------------------------
           -                                                                      -
           -        V E R T E X   P O I N T S   E X T R A C T I O N               -
           -                                                                      -
           -                      A L G O R I T H M                               -
           -                                                                      -
           ------------------------------------------------------------------------
           -                                                                      -
           -     ACTIVE_LIST DATA STRUCTURE :                                     -
           -                                                                      -
           -                                                                      -
           -   DUMMY          ----   ------------------------------               -
           -    NODE         /   \  /                ------        \              -
           -     V          /     \/                /      \        \             -
           -               /       V               /        \        \            -
           -  [ 0 ]----[ A ]----[ B ]----[ C ]----[ D ]----[ E ]----[ F ]--> NIL  -
           -             |        |        |        |        |        |         -
           -            NIL     [ G ]    [ H ]     NIL     [ K ]     NIL          -
           -                      |        |                 |                  -
           -                     NIL     [ I ]              NIL                   -
           -                               |                                    -
           -                             [ J ]                                  -
           -                               |                                    -
           -                              NIL                                   -
           -                                                                    -
           -                                                                    -
           -                  A              B     C                            -
           -   ----|          |--------------|     |-----------   PREVIOUS LINE -
           -              D              E       F                              -
           -    ----|     |--------------|       |----------        NEW LINE    -
           -                                                                    -
           -                                                                    -
           -    THIS ALGORITHM EXTRACTS A LINKED LIST OF VERTICES OF CONTOURS     -
           -    CONTAINED IN A RASTER-SCANNED BINARY IMAGE.  THE FRAME MAY CONTAIN -
           -    ANY NUMBER OF POLYGONAL REGIONS (OBJECTS) WHICH MAY INTERWEAVE AND -
           -    EACH MAY CONTAIN ANY NUMBER OF HOLES.  OBJECTS WHICH APPEAR AS    -
           -    'ISLANDS' WITHIN HOLES OF LARGER OBJECTS ARE ALSO ALLOWED.  THE   -
           -    OUTPUT OF THIS ALGORITHM IS THE LINKED LIST OF (X,Y) COORDINATES  -
           -    FOR EACH OBJECT AND HOLE.  OBJECTS ARE LINKED COUNTER-CLOCKWISE AND -
           -    HOLES ARE LINKED CLOCK-WISE.  A STARTING POINT INTO THE CHAIN IS  -
           -    GIVEN  ; FOR OBJECTS, IT WILL BE THAT POINT WHICH HAS MAXIMUM 'Y' -
           -    AND MINIMUM 'X' ; FOR HOLES IT WILL BE MAXIMUM 'Y' AND MAXIMUM 'X'. -
           -                                                                      -
           -                                           CONTINUED ...              -
           -                                                                      -
           ------------------------------------------------------------------------ */
```

$EJECT

```
/*--------------------------------------------------------------------------
 -                                                                        -
 -  ACTIVE_LIST IS THE DATA STRUCTURE WHICH CONTAINS THE INFORMATION     -
 -  ABOUT ANY OBJECT WHICH IS CURRENTLY 'ACTIVE' ; I.E. SOME PORTION     -
 -  OF THE OBJECT APPEARS IN THE LAST LINE OF THE FRAME.  THE VARIABLE   -
 -  CURRENT_OBJECT_PTR POINTS AT THE NODE IN ACTIVE_LIST WHICH IS        -
 -  CURRENTLY BEING ANALYZED, AND ( A_PTR, B_PTR AND C_PTR) POINT INTO   -
 -  THE VERTEX_LIST STRUCTURE TO DEFINE THE (A,B,C) VALUES.  THE NEXT    -
 -  LINE'S SEGMENTS ARE STORED IN THE LINE_BUFFER STRUCTURE WHICH WILL   -
 -  EVENTUALLY BE IMPLEMENTED IN HARDWARE.  THE (D,E,F) VALUES ARE       -
 -  OBTAINED FROM THIS STURCTURE AT LOCATIONS DEFINED BY THE POINTERS   -
 -  ( D_PTR, E_PTR, F_PTR ).                                             -
 -                                                                        -
 -  AS OBJECTS ARE COMPLETED DURING THE RASTER SCAN, POINTERS TO THEIR   -
 -  NODES ARE ADDED TO THE STRUCTURE COMPLETE_LIST AND SUBSEQUENTLY      -
 -  DELETED FROM THE ACTIVE_LIST. A LINKED LIST OF HOLES CONTAINED IN    -
 -  AN OBJECT IS ACCESSED VIA THE FIELD HOLE_PTR IN EACH OBJECT NODE.    -
 -                                                                        -
 ---------------------------------------------------------------------------*/
```

```
                $EJECT
                /* ------------------------------------------------------------------------
                              S E R V I C E     D E C L A R A T I O N S
                   ------------------------------------------------------------------------ */
  2   1            CRT$STRING$OUT: PROCEDURE(PTR) EXTERNAL;
  3   2                    DECLARE PTR POINTER;
  4   2            END CRT$STRING$OUT;

  5   1            DECLARE
                    MESSAGE(*) BYTE DATA(0DH,0AH,'VERTEX POINTS found ...',0DH,0AH,03H);

  6   1            DECLARE
                    SEGMENT(1500) STRUCTURE ( XL BYTE, XR BYTE, Y BYTE) AT (6000H);

  7   1            DECLARE
                    N   WORD;
```

```
                 $EJECT

                 /* ----------------------------------------------------------------
                                        D E C L A R A T I O N S
                    ---------------------------------------------------------------- */

  8    1          DECLARE
                     TRUE              LITERALLY 'OFFH',
                     FALSE             LITERALLY '0',
                     NIL               LITERALLY 'OFFFFH',
                     DUMMY             LITERALLY 'ACTIVE_LIST(0)',
                     MAXIMUM_RIGHT     LITERALLY 'OFFH';

  9    1          DECLARE
                     FRAME_IS_NOT_COMPLETE   BYTE;

 10    1          DECLARE
                    VERTEX_LIST (2000) STRUCTURE (X     BYTE,
                                                  Y     BYTE,
                                                  LINK  WORD)        AT (8000H);

 11    1          DECLARE
                    NEW_VERTEX_PTR        WORD;

 12    1          DECLARE
                    ACTIVE_LIST (200) STRUCTURE (DELTA_XL  INTEGER,
                                                 DELTA_XR  INTEGER,
                                                 LEFT      WORD,
                                                 RIGHT     WORD,
                                                 HOLE_PTR  WORD,
                                                 LAST_HOLE WORD,
                                                 FORWARD   WORD,
                                                 BACK      WORD,
                                                 NEXT      WORD)     AT (0A000H);

 13    1          DECLARE
                    NEW_OBJECT_PTR         WORD,
                    PREVIOUS_OBJECT_PTR    WORD,
                    CURRENT_OBJECT_PTR     WORD;

 14    1          DECLARE
                    COMPLETE_LIST(50)      WORD   AT (0B000H),
                    COMPLETE_OBJECT_PTR    WORD,
                    NO_OBJECTS_IN_FRAME    WORD   AT (0B064H);

 15    1          DECLARE
                    ( A,B,C,D,E,F )                                    BYTE,
                    ( A_PTR, B_PTR, C_PTR, D_PTR, E_PTR, F_PTR )       WORD;

 16    1          DECLARE
                    LINE_BUFFER(100)      BYTE    AT (0B066H),
                    LINE_NUMBER           BYTE    AT (0B0C8H),
                    LINE_BUFFER_PTR       WORD;
```

```
                  $EJECT

17   1          READ_NEW_SEGMENTS: PROCEDURE;

                /*   THIS ROUTINE IS USED TO SIMULATE THE HARDWARE ACQUISITION OF
                     RUN-LENGTH CODES FOR ONE LINE IN THE FRAME.  AS THE CODES ARE
                     COLLECTED THEY ARE STORED IN THE LINE_BUFFER STRUCTURE.        */

18   2          DECLARE
                  NUMBER_OF_LINES_IN_FRAME    LITERALLY  '240';

19   2          LINE_BUFFER_PTR = 0;
20   2          LINE_NUMBER     = LINE_NUMBER + 1;

21   2          IF  ( LINE_NUMBER = NUMBER_OF_LINES_IN_FRAME )

                     THEN    FRAME_IS_NOT_COMPLETE = FALSE;

                     ELSE

23   2          DO WHILE ( SEGMENT(N).XL <> OFFH ) AND ( SEGMENT(N).Y = LINE_NUMBER );

24   3             LINE_BUFFER( LINE_BUFFER_PTR ) = SEGMENT(N).XL;
25   3             LINE_BUFFER_PTR = LINE_BUFFER_PTR + 1;

26   3             LINE_BUFFER( LINE_BUFFER_PTR ) = SEGMENT(N).XR;
27   3             LINE_BUFFER_PTR = LINE_BUFFER_PTR + 1;

28   3             N = N + 1;

29   3          END;

30   2          END READ_NEW_SEGMENTS;
```

```
                $EJECT


 31   1         INITIALIZE: PROCEDURE;

                /*   THIS ROUTINE IS CALLED PRIOR TO THE START OF THE ANALYSIS OF A NEW
                     FRAME.  THE POINTERS INTO ACTIVE_LIST, VERTEX_LIST, AND COMPLETE_LIST
                     AS WELL AS THE NUMBER-OF-OBJECTS COUNTER AND THE FRAME-COMPLETE FLAG
                     ARE INITIALIZED HERE                                                    */

 32   2            N           = 0;    /*    COUNTER INTO SEGMENTS ARRAY          */
 33   2            LINE_NUMBER = 0FFH; /*    RESET TO BEGINNING OF FRAME - 1      */

 34   2            PREVIOUS_OBJECT_PTR      = 0;
 35   2            NEW_VERTEX_PTR           = 0;
 36   2            NEW_OBJECT_PTR           = 1;
 37   2            COMPLETE_OBJECT_PTR      = 0;
 38   2            DUMMY(.NEXT              = NIL;
 39   2            CURRENT_OBJECT_PTR       = NIL;
 40   2            NO_OBJECTS_IN_FRAME      = 0;

 41   2            FRAME_IS_NOT_COMPLETE = TRUE;

 42   2            A,C = MAXIMUM_RIGHT;
 43   2            D,F = MAXIMUM_RIGHT;

 44   2         END INITIALIZE;



 45   1         ADVANCE_OBJECT_POINTERS: PROCEDURE;

 46   2            PREVIOUS_OBJECT_PTR = CURRENT_OBJECT_PTR;
 47   2            CURRENT_OBJECT_PTR  = ACTIVE_LIST( CURRENT_OBJECT_PTR ).NEXT;

 48   2         END ADVANCE_OBJECT_POINTERS;
```

```
                $EJECT

 49   1      ADVANCE_ABC: PROCEDURE;

 50   2          A = 0;
 51   2          B = 0;

 52   2          IF  CURRENT_OBJECT_PTR <> NIL

               THEN  DO;
 54   3                  A_PTR = ACTIVE_LIST( CURRENT_OBJECT_PTR ).LEFT;
 55   3                  B_PTR = ACTIVE_LIST( CURRENT_OBJECT_PTR ).RIGHT;

 56   3                  A = VERTEX_LIST( A_PTR ).X;
 57   3                  B = VERTEX_LIST( B_PTR ).X;

 58   3                  IF  ACTIVE_LIST( CURRENT_OBJECT_PTR ).NEXT = NIL

                           THEN   C = MAXIMUM_RIGHT;
 60   3                    ELSE   DO;
 61   4                            C_PTR = ACTIVE_LIST( ACTIVE_LIST( CURRENT_OBJECT_PTR ).NEXT ).LEFT;
 62   4                            C = VERTEX_LIST( C_PTR ).X;
 63   4                            END;


 64   3                  END;

 65   2      END ADVANCE_ABC;


 66   1      RESET_ABC: PROCEDURE;

             /*     THIS ROUTINE POSITIONS A, B AND C TO THE FIRST POSITION IN
                    THE ACTIVE_LIST.  IF ACTIVE_LIST IS EMPTY, THEN A REMAINS AT
                    THE MAXIMUM RIGHT VALUE.  IF THERE IS ONLY ONE ACTIVE-OBJECT,
                    THEN C IS SET TO THE MAXIMUM RIGHT VALUE.                    */

 67   2          PREVIOUS_OBJECT_PTR = 0;
 68   2          CURRENT_OBJECT_PTR  = DUMMY.NEXT;
 69   2          CALL ADVANCE_ABC;

 70   2      END RESET_ABC;
```

```
                $EJECT

 71   1         ADVANCE_DEF: PROCEDURE;

 72   2             D     = F;
 73   2             E     = F;
 74   2             D_PTR = F_PTR;
 75   2             E_PTR = F_PTR + 1;
 76   2             F_PTR = E_PTR + 1;

 77   2             IF D_PTR <> LINE_BUFFER_PTR

                        THEN DO;
 79   3                     D = LINE_BUFFER( D_PTR );
 80   3                     E = LINE_BUFFER( E_PTR );
 81   3                     IF   F_PTR = LINE_BUFFER_PTR

                                THEN    F = MAXIMUM_RIGHT;
 83   3                         ELSE    F = LINE_BUFFER( F_PTR );

 84   3                     END;

 85   2         END ADVANCE_DEF;


 86   1         RESET_DEF: PROCEDURE;

                /*    THIS ROUTINE POSITIONS D, E AND F  AT THE FIRST SEGMENT ON A NEW
                      LINE.  IF THE NEW LINE CONTAINS NO SEGMENTS, THEN D REMAINS AT THE
                      MAXIMUM VALUE.  IF THE NEW LINE CONTAINS ONLY ONE SEGMENT, THEN  F
                      IS SET TO THE MAXIMUM.                                              */

 87   2             F_PTR = 0;
 88   2             CALL ADVANCE_DEF;

 89   2         END RESET_DEF;
```

```
                $EJECT


 90   1        INITIALIZE_NODE: PROCEDURE;

               /*    THIS ROUTINE INITIALIZES THE FIELDS OF A NEW NODE ADDED TO THE
                     ACTIVE_LIST.                                                      */

 91   2            ACTIVE_LIST(NEW_OBJECT_PTR).DELTA_XL  = 256;
 92   2            ACTIVE_LIST(NEW_OBJECT_PTR).DELTA_XR  = 256;
 93   2            ACTIVE_LIST(NEW_OBJECT_PTR).LEFT      = NEW_VERTEX_PTR - 2;
 94   2            ACTIVE_LIST(NEW_OBJECT_PTR).RIGHT     = NEW_VERTEX_PTR - 1;
 95   2            ACTIVE_LIST(NEW_OBJECT_PTR).HOLE_PTR  = NIL;
 96   2            ACTIVE_LIST(NEW_OBJECT_PTR).LAST_HOLE = NEW_OBJECT_PTR;
 97   2            ACTIVE_LIST(NEW_OBJECT_PTR).FORWARD   = NEW_OBJECT_PTR;
 98   2            ACTIVE_LIST(NEW_OBJECT_PTR).BACK      = NEW_OBJECT_PTR;

 99   2        END INITIALIZE_NODE;



100   1        INSERT_NODE_TO_RIGHT: PROCEDURE( PTR );

               /*   THIS ROUTINE INSERTS A NODE IN ACTIVE_LIST TO THE RIGHT OF THE NODE
                    POINTED TO BY 'PTR'                                                */

101   2            DECLARE PTR WORD;

102   2            ACTIVE_LIST(NEW_OBJECT_PTR).NEXT = ACTIVE_LIST(PTR).NEXT;
103   2            ACTIVE_LIST(PTR).NEXT = NEW_OBJECT_PTR;

104   2            CALL INITIALIZE_NODE;

105   2            NEW_OBJECT_PTR = NEW_OBJECT_PTR + 1;

106   2        END INSERT_NODE_TO_RIGHT;
```

```
                $EJECT

107   1         DELETE_NEXT_NODE: PROCEDURE(PTR);

                /*   THIS ROUTINE DELETES A NODE FROM ACTIVE_LIST.  PTR POINTS TO THE NODE
                     WHOSE 'NEXT' POINTER POINTS TO THE NODE TO BE DELETED.              */

108   2            DECLARE PTR WORD;

109   2              ACTIVE_LIST(PTR).NEXT = ACTIVE_LIST(ACTIVE_LIST(PTR).NEXT).NEXT;

110   2         END DELETE_NEXT_NODE;



111   1         JOIN_TO_HOLE_LIST: PROCEDURE( NODE_PTR, FIRST_PTR, LAST_PTR );

                /*     THIS ROUTINE JOINS A HOLE LIST DEFINED BY FIRST_PTR, LAST_PTR TO THE
                       HOLE LIST OF THE OBJECT NODE POINTED TO BY NODE_PTR.              */

112   2            DECLARE ( NODE_PTR, FIRST_PTR, LAST_PTR ) WORD;

113   2            ACTIVE_LIST( ACTIVE_LIST( NODE_PTR ).LAST_HOLE ).HOLE_PTR = FIRST_PTR;
114   2            ACTIVE_LIST( NODE_PTR ).LAST_HOLE = LAST_PTR;

115   2         END JOIN_TO_HOLE_LIST;



116   1         MOVE_XY: PROCEDURE ( POINT, PTR );

                /*     THIS ROUTINE MOVES THE (X,Y) COORDINATES FOR THE POINT SPECIFIED
                       AS POINT TO THE LOCATION IN THE VERTEX_LIST ADDRESSED BY PTR.     */

117   2            DECLARE POINT BYTE, PTR WORD;

118   2            VERTEX_LIST( PTR ).X = POINT;
119   2            VERTEX_LIST( PTR ).Y = LINE_NUMBER;

120   2         END MOVE_XY;
```

```
                 $ EJECT

 121   1      JOIN_E_TO_D: PROCEDURE;

 122   2          CALL MOVE_XY( D, NEW_VERTEX_PTR );
 123   2          NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;
 124   2          CALL MOVE_XY( E, NEW_VERTEX_PTR );
 125   2          VERTEX_LIST( NEW_VERTEX_PTR ).LINK = NEW_VERTEX_PTR - 1;
 126   2          NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;

 127   2      END JOIN_E_TO_D;


 128   1      JOIN_A_TO_B: PROCEDURE;

 129   2          VERTEX_LIST( A_PTR ).LINK = B_PTR;

 130   2      END JOIN_A_TO_B;


 131   1      JOIN_A_TO_D: PROCEDURE;

 132   2          DECLARE
                     CURRENT_DELTA_XL  INTEGER;

 133   2          CURRENT_DELTA_XL = INT( D )  -  INT( A );

 134   2          IF CURRENT_DELTA_XL = ACTIVE_LIST( CURRENT_OBJECT_PTR ).DELTA_XL

                      THEN

 135   2                CALL MOVE_XY ( D, A_PTR );

                      ELSE

 136   2                DO;
 137   3                CALL MOVE_XY( D, NEW_VERTEX_PTR);
 138   3                VERTEX_LIST( A_PTR ).LINK = NEW_VERTEX_PTR;
 139   3                ACTIVE_LIST( CURRENT_OBJECT_PTR ).LEFT = NEW_VERTEX_PTR;
 140   3                NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;
 141   3                ACTIVE_LIST( CURRENT_OBJECT_PTR ).DELTA_XL = CURRENT_DELTA_XL;
 142   3                END;

 143   2      END JOIN_A_TO_D;
```

```
                 $EJECT

144   1       JOIN_E_TO_B: PROCEDURE;

145   2           DECLARE
                    CURRENT_DELTA_XR   INTEGER;

146   2           CURRENT_DELTA_XR = INT( B )  -  INT( E );

147   2           IF CURRENT_DELTA_XR = ACTIVE_LIST( CURRENT_OBJECT_PTR ).DELTA_XR

                      THEN

148   2                 CALL MOVE_XY ( E, D_PTR );

                      ELSE

149   2                 DO;
150   3                 CALL MOVE_XY ( E, NEW_VERTEX_PTR );
151   3                 VERTEX_LIST( NEW_VERTEX_PTR ).LINK = B_PTR;
152   3                 ACTIVE_LIST( CURRENT_OBJECT_PTR ).RIGHT = NEW_VERTEX_PTR;
153   3                 NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;
154   3                 ACTIVE_LIST( CURRENT_OBJECT_PTR ).DELTA_XR = CURRENT_DELTA_XR;
155   3                 END;

156   2       END JOIN_E_TO_B;


157   1       JOIN_E_TO_F: PROCEDURE;

158   2           CALL MOVE_XY( E, NEW_VERTEX_PTR );
159   2           VERTEX_LIST( NEW_VERTEX_PTR ).LINK = NEW_VERTEX_PTR + 1;
160   2           ACTIVE_LIST( CURRENT_OBJECT_PTR ).RIGHT = NEW_VERTEX_PTR;
161   2           NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;
162   2           CALL MOVE_XY( F, NEW_VERTEX_PTR );
163   2           NEW_VERTEX_PTR = NEW_VERTEX_PTR + 1;

164   2       END JOIN_E_TO_F;


165   1       JOIN_C_TO_B: PROCEDURE;

166   2           VERTEX_LIST( C_PTR ).LINK = B_PTR;

167   2       END JOIN_C_TO_B;
```

```
          $EJECT


168  1    UPDATE_FOR_TERMINATE: PROCEDURE;

169  2       ACTIVE_LIST( ACTIVE_LIST( CURRENT_OBJECT_PTR ).FORWARD ).BACK
                     = ACTIVE_LIST( CURRENT_OBJECT_PTR ).BACK;
170  2       ACTIVE_LIST( ACTIVE_LIST( CURRENT_OBJECT_PTR ).BACK ).FORWARD
                     = ACTIVE_LIST( CURRENT_OBJECT_PTR ).FORWARD;

171  2       IF ACTIVE_LIST( CURRENT_OBJECT_PTR ).BACK = CURRENT_OBJECT_PTR

                     THEN

172  2                 DO;
173  3                 COMPLETE_LIST( COMPLETE_OBJECT_PTR ) = CURRENT_OBJECT_PTR;
174  3                 COMPLETE_OBJECT_PTR = COMPLETE_OBJECT_PTR + 1;
175  3                 NO_OBJECTS_IN_FRAME = NO_OBJECTS_IN_FRAME + 1;
176  3                 END;

                     ELSE

177  2       IF   ACTIVE_LIST( CURRENT_OBJECT_PTR ).HOLE_PTR <> NIL

                     THEN

178  2                 CALL JOIN_TO_HOLE_LIST
                             ( ACTIVE_LIST( CURRENT_OBJECT_PTR ).BACK,
                               ACTIVE_LIST( CURRENT_OBJECT_PTR ).HOLE_PTR,
                               ACTIVE_LIST( CURRENT_OBJECT_PTR ).LAST_HOLE );

             CALL DELETE_NEXT_NODE( PREVIOUS_OBJECT_PTR );

180  2       CURRENT_OBJECT_PTR = ACTIVE_LIST( PREVIOUS_OBJECT_PTR ).NEXT;

181  2    END UPDATE_FOR_TERMINATE;



182  1    UPDATE_FOR_CREATE: PROCEDURE;

183  2       CALL INSERT_NODE_TO_RIGHT( PREVIOUS_OBJECT_PTR );

184  2       PREVIOUS_OBJECT_PTR = ACTIVE_LIST( PREVIOUS_OBJECT_PTR ).NEXT;

185  2    END UPDATE_FOR_CREATE;
```

```
          $EJECT

186  1    UPDATE_FOR_SPLIT: PROCEDURE;

187  2       CALL INSERT_NODE_TO_RIGHT( CURRENT_OBJECT_PTR );

188  2       ACTIVE_LIST( ACTIVE_LIST(CURRENT_OBJECT_PTR).NEXT ).FORWARD = ACTIVE_LIST(CURRENT_OBJECT_PTR).FORWARD;
189  2       ACTIVE_LIST( ACTIVE_LIST(CURRENT_OBJECT_PTR).FORWARD).BACK = ACTIVE_LIST(CURRENT_OBJECT_PTR).NEXT;
190  2       ACTIVE_LIST(CURRENT_OBJECT_PTR).FORWARD = ACTIVE_LIST( CURRENT_OBJECT_PTR ).NEXT;
191  2       ACTIVE_LIST( ACTIVE_LIST(CURRENT_OBJECT_PTR).NEXT ).BACK = CURRENT_OBJECT_PTR;

192  2       CALL ADVANCE_OBJECT_POINTERS;

193  2    END UPDATE_FOR_SPLIT;


194  1    UPDATE_FOR_MERGE: PROCEDURE;

195  2       DECLARE NEXT_OBJECT_PTR WORD;

196  2       NEXT_OBJECT_PTR = ACTIVE_LIST( CURRENT_OBJECT_PTR ).NEXT;

197  2       IF ( ACTIVE_LIST(CURRENT_OBJECT_PTR).FORWARD = NEXT_OBJECT_PTR )

               THEN   CALL  JOIN_TO_HOLE_LIST
                            ( CURRENT_OBJECT_PTR,
                              NEXT_OBJECT_PTR,
                              ACTIVE_LIST(NEXT_OBJECT_PTR).LAST_HOLE );

198  2         ELSE   IF ( ACTIVE_LIST (NEXT_OBJECT_PTR).HOLE_PTR  <> NIL )

                        THEN   CALL JOIN_TO_HOLE_LIST
                                    ( CURRENT_OBJECT_PTR,
                                      ACTIVE_LIST(NEXT_OBJECT_PTR).HOLE_PTR,
                                      ACTIVE_LIST(NEXT_OBJECT_PTR).LAST_HOLE );

             ACTIVE_LIST( ACTIVE_LIST(NEXT_OBJECT_PTR).BACK ).FORWARD = ACTIVE_LIST( CURRENT_OBJECT_PTR ).FORWARD;
202  2       ACTIVE_LIST( ACTIVE_LIST( CURRENT_OBJECT_PTR ).FORWARD ).BACK = ACTIVE_LIST( NEXT_OBJECT_PTR ).BACK;
203  2       ACTIVE_LIST( CURRENT_OBJECT_PTR ).FORWARD = ACTIVE_LIST( NEXT_OBJECT_PTR ).FORWARD;
204  2       ACTIVE_LIST( ACTIVE_LIST( NEXT_OBJECT_PTR ).FORWARD ).BACK = CURRENT_OBJECT_PTR;

205  2       ACTIVE_LIST( CURRENT_OBJECT_PTR ).RIGHT = ACTIVE_LIST( NEXT_OBJECT_PTR ).RIGHT;

206  2       CALL DELETE_NEXT_NODE( CURRENT_OBJECT_PTR );

207  2    END UPDATE_FOR_MERGE;
```

```
              $EJECT


208   1       CREATE: PROCEDURE;

209   2           CALL JOIN_E_TO_D;
210   2           CALL UPDATE_FOR_CREATE;
211   2           CALL ADVANCE_DEF;

212   2       END CREATE;



213   1       TERMINATE: PROCEDURE;

214   2           CALL JOIN_A_TO_B;
215   2           CALL UPDATE_FOR_TERMINATE;
216   2           CALL ADVANCE_ABC;

217   2       END TERMINATE;



218   1       MERGE: PROCEDURE;

219   2           CALL JOIN_C_TO_D;
220   2           CALL UPDATE_FOR_MERGE;
221   2           CALL ADVANCE_ABC;

222   2       END MERGE;



223   1       SPLIT: PROCEDURE;

224   2           CALL JOIN_E_TO_F;
225   2           CALL ADVANCE_DEF;
226   2           CALL UPDATE_FOR_SPLIT;

227   2       END SPLIT;
```

```
              $EJECT


 228   1      GET$VERTEX$POINTS: PROCEDURE PUBLIC;

 229   2         CALL INITIALIZE;

 230   2         DO WHILE  FRAME_IS_NOT_COMPLETE;

 231   3            CALL RESET_ABC;
 232   3            CALL READ_NEW_SEGMENTS;
 233   3            CALL RESET_DEF;

 234   3            DO  WHILE ( (A<>MAXIMUM_RIGHT) OR (D<>MAXIMUM_RIGHT) );

 235   4               IF  E < A  THEN      CALL CREATE;

                                ELSE

 237   4               IF  D > B  THEN      CALL TERMINATE;

 238   4                          ELSE      DO;
 240   5                                    CALL JOIN_A_TO_D;
 241   5                                    DO WHILE  (F < B) OR (E > C) ;
 242   6                                       IF (F < B) THEN CALL SPLIT;
 244   6                                                  ELSE CALL MERGE;
 245   6                                    END;
 246   5                                    CALL JOIN_E_TO_B;
 247   5                                    CALL ADVANCE_OBJECT_POINTERS;
 248   5                                    CALL ADVANCE_ABC;
 249   5                                    CALL ADVANCE_DEF;
 250   5                                    END;

 251   4            END;

 252   3         END;



 253   2      CALL CRTSTRINGOUT(@MESSAGE);

 254   2      END GET$VERTEX$POINTS;


 255   1      END VERTEX;
```

```
            $EJECT


MODULE INFORMATION:

     CODE AREA SIZE     = 0806H   2054D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0027H     39D
     MAXIMUM STACK SIZE = 0014H     20D
     690 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

REFERENCES

Chapter 1

[1.1] Suresh, B.R., et al, "A Real-Time Automated Visual Inspection System for Hot Steel Slabs", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 6, 1983, pp. 563-572.

[1.2] Conners, R.W., et al., "Identifying and Locating Surface Defects in Wood: Part of an Automated Lumber Processing System", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 6, 1983, pp. 573-583.

[1.3] Baird, M.L., "GAGESIGHT: A Computer Vision System for Automatic Inspection of Instrument Gages", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 6, 1983, pp. 618-621.

[1.4] Birk, J.R., R.B. Kelley, and L. Wilson, "Acquiring Workpieces: Three Approaches Using Vision", Proceeding of the Eighth International Symposium on Industrial Robots, Stuttgart, West Germany, 1978, pp. 724-733.

[1.5] Kelley, R.B. et al., "A Robot System which Acquires Cylindrical Workpieces from Bins", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-12, No. 2, 1982, pp. 204-213.

[1.6] Perkins, W.A., "A Model-Based Vision System for Industrial Parts", IEEE Transactions on Computers, Vol. C-27, 1978, pp. 126-143.

[1.7] Perkins, W.A., "Area Segmentation of Images Using Edge Points", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-2, No. 1, 1980, pp. 8-15.

[1.8] Agin, G.J., "Computer Vision Systems for Industrial Inspection and Assembly", IEEE Computer, Vol. 13, No. 5, 1980, pp. 11-20.

[1.9] Stoffel, J.C. (ed.), Graphical and Binary Image Processing and Applications, Artech House, Dedhem, MA, 1982.

[1.10] Agin, G.J., "An Experimental Vision System for Industrial Applications", Proceedings of the 5th International Symposium on Industrial Robots, September, 1975, pp. 135-148.

[1.11] Sanderson, R.J., Machine Vision Systems: A Summary and Forecast, Tech Tran Corp., Naperville, Illinois, 1983.

[1.12] Gonzalez, R.C. and P. Wintz, Digital Image Processing, Addison-Wesley Publishing Co., Reading, Massachusetts, 1977, pp. 320-387.

[1.13] Birk, J., R. Kelley, and H. Martins, "An Orienting Robot for Feeding Workpieces Stored in Bins", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-11, 1981, pp. 151-160.

[1.14] Holland, S.W., L. Rossol and M.R. Ward, "CONSIGHT-I: A Vision-Controlled Robot System for Transferring Parts from Belt Conveyors", Computer Vision and Sensor-Based Robots, (ed. Dodd, G.G. and L. Rossol), Plenum Press, New York, 1979, pp. 81-97.

[1.15] Pinkney, H.F.L., "Theory and Development of an On-Line 30 Hz Video Photogrammetry System for Real-Time Three-Dimensional Control", Proceedings of the ISP Symposium on Photogrammetry for Industry, Stockholm, August, 1978.

[1.16] Aleksander, I. (ed.), Artificial Vision for Robots, Chapman and Hall, N.Y., 1983.

[1.17] Pugh, A. (ed.), Robot Vision, Springer-Verlag, N.Y., 1983.

Chapter 2

[2.1] Gonzalez, R.C. and R. Safabakhsh, "Computer Vision Techniques for Industrial Applications and Robot Control", IEEE Computer, Vo. 15, No. 12, 1982, pp. 17-32.

[2.2] Weszka, J.S., "A Survey of Threshold Selection Techniques", Computer Graphics and Image Processing, Vol. 7, 1978, pp. 259-265.

[2.3] Gonzalez, R.C. and P. Wintz, Digital Image Processing, Addison-Wesley Publishing Co., Reading, Massachusetts, 1977, p. 232.

[2.4] Rosenfeld, A., "Connectivity in Digital Pictures", Journal of the Association of Computing Machinery, vol. 17, 1, January 1970, pp. 146-160.

[2.5] Agin, G.J., "Image Processing Algorithms for Industrial Vision", private communication, unpublished.

[2.6] Beer, F.P. and E.R. Johnston, Jr., Mechanics for Engineers: Statics and Dynamics, McGraw-Hill Book Co., New York, N.Y., 1962, p. 143.

[2.7] Agin, G.J., "Computer Vision Systems for Industrial Inspection and Assembly", IEEE Computer, Vol. 13, No. 5, 1980, pp. 11-20.

[2.8] Martini, P. and G. Nehr, "Recognition of Angular Orientation of Objects with the Help of Optical Sensors", The Industrial Robot, Vol. 6, 1979, pp. 62-69.

[2.9] Fahim, A.E.F. and R.M.H. Cheng, "An On-Line System for Identifying the Angular Orientation of a Class of Engineering Components", IEEE Transactions on Industrial Electronics and Control Instrumentation, Vol. IECI-27, No. 3, 1980, pp. 189-196.

Chapter 3

[3.1] Rosenfeld, A., "Connectivity in Digital Pictures", Journal of the Association of Computing Machinery, Vol. 17, 1, Jan. 1970, pp. 146-160.

[3.2] Freeman, H., "On the Encoding of Arbitrary Geometric Configurations", IRE Transactions on Electronic Computers, 10, 2, June 1961, pp. 260-268.

[3.3] Nevatia, R., Machine Perception, Prentice-Hall, New Jersey, 1982, pp. 138.

[3.4] Pavlidis, T., Algorithms for Graphics and Image Processing, Computer Science Press, Rockville, MD, 1982, pp. 142-148.

[3.5] Dudani, S.A., "Region Extraction using Boundary Following", Pattern Recognition and Artificial Intelligence (ed. C.H. Chen), Academic Press, New York, 1976, pp. 216-232.

[3.6] Sobel, I., "Neighborhood Coding of Binary Images for Fast Contour Following and General Binary Array Processing", Computer Graphics and Image Processing, 8, 1978, pp. 127-135.

[3.7] Pavlidis, T., "A Minimum Storage Boundary Tracing Algorithm and Its Application to Automatic Inspection", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-8, No. 1, 1978, pp. 66-69.

[3.8] Dyer, C.R., A. Rosenfeld and H. Samet, "Region Representation: Boundary Codes from Quadtrees", Communications of the Association of Computing Machinery, Vol. 23, No. 3, 1980, pp. 171-179.

[3.9] Grant, G. and A.F. Reid, "An Efficient Algorithm for Boundary Tracing and Feature Extraction", Computer Graphics and Image Processing, 17, 1981, pp. 225-237.

[3.10] Rosenfeld, A. and A.C. Kak, Digital Picture Processing, Academic Press, New York, 1982, Vol. 2, pp. 224-225.

[3.11] Cederberg, R.L.T., "Chain-Link Coding and Segmentation for Raster Scan Devices", Computer Graphics and Image Processing, 10, 1979, pp. 224-234.

[3.12] Agrawala, A.K. and A.V. Kulkarni, "A Sequential Approach to the Extraction of Shape Features", Computer Graphics and Image Processing, 6, 1977, pp. 538-557.

[3.13] Kulkarni, A.V., "Sequential Shape Feature Extraction from Line Drawings", Proceedings of the Pattern Recognition and Image Processing Conference, Chicago, 1978, pp. 230-237.

[3.14] Freeman, H., "Computer Processing of Line-Drawing Images", Computing Surveys, 6, 1, 1974, pp. 57-97.

[3.15] Freeman, H. and R. Shapiro, "Determining the Minimum-Area Encasing rectangle for an Arbitrary Closed Curve", Communications of the Association of Computing Machinery, 18, 7, 1975, pp. 409-413.

Chapter 4

[4.1] User's Guide to VAL, June 1980, Unimation Inc., Danbury, CT.

[4.2] iSBC 86/12A Single Board Computer Hardware Reference Manual, 1978, Intel Corporation.

[4.3] The 8086 Family User's Manual (Numeric Supplement), July, 1980, Intel Corporation.

[4.4] MULTIBUS Handbook, 1983, Intel Corporation.

[4.5] TN2500 Solid State Video/Digital Camera OPERATING MANUAL, April 1980, General Electric, Electronic Systems Division.

[4.6] Pena, M., "Adaptive Lighting for Computer Vision", M.Eng. Thesis, McMaster University, January 1983.

[4.7] PL/M-86 Programming Manual, 1978, Intel Corporation.

[4.8] MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users, 1978, Intel Corporation.

[4.9] 8087 Support Library Reference Manual, 1981, Intel Corporation.

[4.10] Vetter, M. and R.N. Maddison, Data Base Design Methodology, Prentice Hall International, Englewood Cliffs, N.J., 1981.

[4.11] Carlisle, B., S. Roth, J. Gleason and D. McGhie, "The PUMA/VS-100 Robot Vision System", Proceedings of the 1st International Conference on Robot Vision and Sensory Controls, Stratford-upon-Avon, UK, April, 1981, pp. 149-160.

Chapter 5

[5.1] Rosen, C.A. and G.J. Gleason, "Evaluating Vision System Performance", Robotics Today, Fall 1981.

[5.2] Capson, D. and R. Kitai, "Silhouette Area and Centroid Measurement using a CID camera and an 8086 Microcomputer", Proceedings of the IECI Conference on Applications of Mini and Micro computers, San Francisco, CA, 1981.

[5.3] Rosen, C. et al., Machine Intelligence Research Applied to Industrial Automation, Eighth Report, NSF Grant APR75-13074, SRI Project 4391, SRI International, Menlo Park, California, August, 1978, pp. 101-109.

[5.4] Ho, C.-S, "Precision of Digital Vision Systems", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-5, No. 6, 1983, pp. 593-601.