INTERACTIVE POLYGON FILLING ON A RASTER GRAPHIC DISPLAY

by

STEPHEN LEONARD STOCH

B.A., The University of Cape Town, 1974
B.Sc., The University of Cape Town, 1978

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

March, 1981

(c) Stephen Leonard Stoch, 1981
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of **COMPUTER SCIENCE**

The University of British Columbia  
2075 Wesbrook Place  
Vancouver, Canada  
V6T 1W5

Date **March 18, 1981**
Abstract

This thesis describes Polygon Filling System, an interactive graphics system, which fills user defined areas on a raster scan graphics display device. The areas may be concave or convex and may be nested within each other. Issues related to the underlying grid model, hexagonal or square tessellation, are discussed. A formal approach to polygon filling is compared to a heuristic domain dependent approach. Connectivity problems arise in processing boundaries. Two user rules are defined. Adherence to these rules ensures correct processing of data. These rules are appropriate for cartographic and remote sensing applications. The implementation of the system and its key data structures are described.
Table of Contents

Chapter I: Introduction ......................................................... 1
   A. Motivation .................................................................. 3
   B. An Example .................................................................. 4
   C. Issues ....................................................................... 9

Chapter II: Underlying Issues .............................................. 10
   A. The Underlying Grid Model ....................................... 10
   B. Rationale of an Heuristic Approach ......................... 15

Chapter III: Boundary Definition: A Functional Overview 16
   A. User Considerations ............................................... 16
   B. Boundary Processing ............................................... 21

Chapter IV: Implementation .................................................. 24
   A. Curve Smoothing ...................................................... 25
      1. Expansion and Contraction ................................... 26
   B. Establish Data ......................................................... 28
   C. Process Boundary ................................................... 33
      1. Smudge Processing ............................................... 34
         1.1 Elimination of Intermediate Bits .................... 38
         1.2 Elimination of Smudge Remaining ................. 40
      2. Closest Point ....................................................... 47
      3. Find the First Point of Boundary .................... 49
      4. Protection of Users Input .................................. 50
   D. Polygon Closure ...................................................... 51
   E. Polygon Filling ........................................................ 52
   F. Tail Detection ........................................................ 57
   G. Deletion of Tails ...................................................... 61
H. Improvements: 2-D Versus 3-D Matrices .......... 63
Chapter V: Problems, Difficulties and Features .......... 65
   A. Sample Run ............................................. 65
   B. System Dependence ...................................... 76
Chapter VI: Conclusion ....................................... 77
Bibliography ..................................................... 80
Appendix A: Program listing for Polygon
            Filling System ........................................ 82
Appendix B: Recursive Algorithm ........................... 132
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Valid set of user defined areas</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Boundaries as defined by the user</td>
<td>4</td>
</tr>
<tr>
<td>1.3</td>
<td>Closed polygons as presented to the user</td>
<td>7</td>
</tr>
<tr>
<td>1.4</td>
<td>Areas filled by PFS</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Square grid showing object ambiguities</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Hexagonal grid showing single object</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Square grid with alternate rows shifted a distance of half a pixel</td>
<td>13</td>
</tr>
<tr>
<td>3.1a</td>
<td>Spacial distribution of pixels conforms to temporal rules</td>
<td>16</td>
</tr>
<tr>
<td>3.1b</td>
<td>Spacial distribution of pixels does not conform to temporal rules</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Spacial distribution of pixels does not conform to temporal rules</td>
<td>18</td>
</tr>
<tr>
<td>3.3a</td>
<td>Correct closure of boundary</td>
<td>19</td>
</tr>
<tr>
<td>3.3b</td>
<td>Incorrect closure of boundary</td>
<td>20</td>
</tr>
<tr>
<td>3.4</td>
<td>Deletion of colinear pixels</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>Deletion of smudge points</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>Flow of control in PFS</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>Elements in a square grid have eight neighbours</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Pascal declarations</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Data structure for the matrix index</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Flow of control in processing an area</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Directions in which the next point may lie</td>
<td>35</td>
</tr>
<tr>
<td>4.7</td>
<td>Traversal of an octagon</td>
<td>36</td>
</tr>
</tbody>
</table>
4.8 Smudge which comprises a thick line with straight outer edge in line with a cartesian axis. .................................37
4.9 Smudge which comprises a group of adjacent pixels on the inside of a reflex vertex .................................37
4.10 Complete smudge .............................................38
4.11a Processing of a sequence of pixels .........................39
4.11b A pixel on the outside of a sequence of pixels halts processing of that sequence ..........40
4.12a Illustration of smudge accuracy ..............................41
4.12b Illustration of smudge accuracy ..............................42
4.13 Smudge processing .............................................45
4.14 Upper and lower polygon arcs .................................53
4.15 Upper polygon arc .............................................54
4.16 Continuous edge strings .......................................55
4.17 Four examples of general edge conditions .................55
4.18 Traced boundary and the resulting closed polygon .58
4.19 Confusion resulting from three changes in state of beam .......................................................59
4.20 Arrows highlighting potential tails .........................60
4.21 Tails joining polygons at arcs and an edge ............61
5.1 Boundaries traced by the user .................................66
5.2 Closed polygon generated in processing boundary ..67
5.3 Problem pixels erased from screen .........................68
5.4 Erased boundary lines retraced ..............................69
5.5 Corrected closed polygon ......................................70
5.6 Arrows highlight existence of a tail ...............71
5.7 Closed polygon generated in processing boundary ..72
5.8 Two new boundaries are added .......................73
5.9 The two new boundaries are verified .................74
5.10 Polygons filled by PFS ..............................75
List of Tables

4.1 Relationship between direction of traversal and smudge outer edge to be traversed ..36

4.2 Relationship between direction of traversal and the neighbour for adjacency check in the first level of recursive smudge processing ....................43

4.3 Relationship between direction of traversal and the neighbour for adjacency check in the second level of recursive smudge processing ....................44
Acknowledgements

I would sincerely like to thank my supervisor, Bob Woodham, for the patience, support, and perceptive criticism that he has given throughout the implementation of this thesis.

I would also like to thank Nou Dadoun, Jay Glicksman, Ihor Gowda, Marc Majka, and Carl Pottle who acted as sounding boards while I postulated various algorithms.

Thanks to Marc for proofreading the thesis, Jay for the use of his FORTRAN subroutines, Dave Brent for his support on the COMTAL, and Nedenia Holm for assistance with diagrams.

A final thanks to office mates Norm and Wendy for tolerating both my ups and downs.
Chapter I: Introduction

This thesis describes Polygon Filling System (PFS), an interactive graphics system which allows the user to define general areas on a raster graphic display device. The boundaries of these areas, hereafter referred to simply as boundaries, are input using a track-ball or some equivalent graphical input device. The areas may be convex or concave. The areas may be nested within each other. The outer most area has a nesting depth of one. Each enclosed area has a nesting depth equal to one plus that of its immediately enclosing area. There may be more than one area of any particular nesting depth.

Fig 1.1 An example of a valid set of user defined areas. The number within each area indicates the nesting depth for that area.
When the user is satisfied with the defined areas, the system will fill all areas whose nesting depth is odd. Areas with even nesting depth are left unfilled.
A. Motivation

The system described in this thesis was motivated by the lack of any facility, either in hardware or software, to perform polygon/area filling on the UBC remote sensing image analysis system.

It was desired that a system be developed which would allow the user to outline areas appearing on Landsat images (lakes, mountains, shadows, etc.) By tracing the boundaries of these areas on the GRAPHIC overlays which form part of the COMTAL Vision One architecture. It was further desired that these areas be filled automatically.

The hardware comprises a COMTAL Vision One image processing system, a high speed bi-directional communication link to the UBC Michigan Terminal System (MTS) facility and the MTS system itself.

The data which is sent from the COMTAL to the PFS can be thought of as a two dimensional matrix of bits. Each element of this matrix can take one of two states, one or zero (on or off). The information is thus spatial in nature. No other information regarding the data entered by the user on the COMTAL's raster screen is available to the program. It is not known, for example, the order (in time) in which the pixels on the screen were turned on or off. This lack of knowledge about the ordering of the bits creates many of the problems addressed in this thesis as will now be shown.
B. An example

The boundaries as traced on the display device by the user are shown in figure 1.2. These boundaries are input with a track-ball or some similar graphical input device.

Figure 1.2 Area boundaries as defined by the user.

Figure 1.2 illustrates three potential problems which arise in user defined boundaries:

(i) Box (a) in the above figure encloses a portion of the boundary curve, the thickness of which is greater than one pixel. This is an occurrence of a smudge. Smudges and the processing thereof are discussed in chapters III and IV below. The problem with smudges is to decide which pixels to retain and which to discard as being noise. PFS follows the outer edge
of the smudge (see chapter IV section C.1).

(ii) Because of the inherent inaccuracies in the trackball, gaps and small holes appear in user defined input. These inaccuracies, and the manner in which they are dealt, are described in chapters III and IV. In figure 1.2, boxes (d) and (e) illustrate examples of a gap and a hole respectively.

(iii) To decide which pixels are connected, the system has to resolve apparent ambiguities resulting from the relative distances between pixels. This ambiguity may exist between pixels from the same boundary or between pixels from different boundaries. Examples of these two forms of ambiguity are shown in the boxes labeled (b) and (c) in figure 1.2. Two rules are given in chapter III which serve as a guide to the user in defining the area boundaries.

PFS processes the user defined area boundaries. In processing these boundaries, the system performs two principle tasks. These are:

(i) Smooth the user defined curves

(ii) Discard redundant or unnecessary pixels, retaining only those whose retention is vital to the preservation of the original shape of the boundary.

Because smudges are traversed along their outer edges, their processing is directionally sensitive. The algorithm for processing smudges is given in chapter IV. The traversal of the boundary along its outer edge preserves the exact shape of the
boundary and allows the user to predict, while tracing the boundary, the shape of the system generated boundary. Neither of these two benefits would be derived if a smudge was processed by passing an average curve through it.

The sequence of pixels which are retained form a polygon, which preserves the shape of the entered boundary. At this stage the polygon is closed by passing the preserved points in pairs to a digital differential analyser (DDA) which sets pixels in a straight line (see chapter IV, section D) between the two points passed to it. That is, it closes off the polygon by joining the points along its perimeter.

The closed polygon is presented to the user for inspection and interactive correction/modification. In the example of figure 1.2, no correction/modification of the closed polygons is necessary. The above process is executed for each of the user defined areas. Figures 1.3 (a-d) show the closed polygons in this run presented one at a time to the user. At this stage the user is invited to modify the polygons and has the opportunity to label them. In this run, a label was added for the outermost area only.
Figure 1.3 Figures (a) through (d) show the different closed polygons as presented to the user for correction/modification.

Once the user defined boundaries are processed, the user is invited to enter more area boundaries. In this run the user declined.
At this stage PFS proceeds to fill the boundaries. The areas whose nesting depth is odd are filled. The areas whose nesting depth is even are left unfilled. Figure 1.4 shows the filled areas.

Figure 1.4  The areas filled by PFS.

Other variations of area filling can easily be implemented. Examples of such variations are: fill areas whose nesting depth is even, fill the entire outer area.

The algorithm which fills the closed polygons is described in chapter IV. In filling areas, problems arise if a closed polygon contains a single column of pixels. A column of this sort is referred to as a tail. If a tail is left unchecked, the filling algorithm will turn the polygon inside out. A discussion of tails appears in chapter IV.
C. Issues

This thesis addresses three major technical issues. These are:

(i) the connectivity of boundaries resulting from uneven data entry (see figure 1.2 above). This involves the following three tasks:
   (a) differentiating between boundaries
   (b) the retention of those points which are vital in maintaining the original shape of the boundary
   (c) the searching of a two dimensional matrix for a pixel's nearest neighbour.

(ii) the closure of the polygons which result from the retention of those pixels retained by (i)(b) above

(iii) the filling of the closed polygons resulting from step (ii) above.
A. The Underlying Grid Model

Two grid systems were considered for storage and manipulation of the screen data:

(i) The square grid is a square array of elements. Each element has four adjacent or primary neighbours at a distance of unity from the element, as well as four vertex or secondary neighbours at a distance of $2$. The distances between a pixel and its eight neighbours are thus directionally sensitive. Such a matrix creates logical difficulties for the analysis of connectivity between pixels [DUDA73] (pp 284-285).

![Square grid showing set and unset pixels.](image)

Figure 2.1 Square grid showing set and unset pixels.

Assume, for example, that the dots in figure 2.1 represent set pixels, while those squares which are
left blank, represent unset pixels. If we let connectivity depend on full adjacency (connected primary neighbours), then the figure represents four unconnected objects which enclose a hole. If on the other hand, connectivity is defined by the adjacency of secondary neighbours, then the figure represents one object, but the hole in the object is connected to the background.

(ii) In the hexagonal grid, each element has six neighbours, each of which is the same distance from the element. Distances between a pixel and its six immediate neighbours are thus not directionally sensitive. All neighbours fall into a single class, that of primary adjacency. Because of this condition, no ambiguities arise with respect to the concept of connectivity. Two pixels are connected if and only if they share a common edge. Figure 2.2 shows a single object which encloses a hole. The hole is not connected to the background.
A comparison between the square and hexagonal grid systems shows that with a square grid, connectivity is inherently ambiguous, while connectivity presents no such ambiguity in a hexagonal system.

The hexagonal grid is more suited to connectivity related problems. However, if the available hardware is inherently square in nature, then preprocessing is required to massage the data in order that the hexagonal system might be simulated. If alternate rows of a square grid are offset by a distance of half a pixel, then the grid layout resembles, pictorially, a brick wall. Each element of this grid has exactly six equidistant neighbours, where all neighbours are fully adjacent. This grid representation is thus analogous to the hexagonal grid system. Figure 2.3 shows this altered representation of a square grid.
The grid system shown in figure 2.3 is an enticing approximation to the hexagonal grid. However, if alternate rows of an existing square grid are simply offset by half a pixel, connectivity in the original figure may not be retained. Additional preprocessing is required to maintain this connectivity. PFS is highly interactive. This implies that transitions between grid systems would have to be done relatively often.

Therefore, despite a hexagonal grid system's benefit with respect to connectivity, it was decided that PFS be implemented using the square grid system because the COMTAL Vision One system's architecture is inherently square in nature. This decision avoids the overheads involved in converting the data back and forth between grid systems. Certain hardware has been designed that is based on the hexagonal grid system [GOLA69],
but because the algorithms implemented in PFS are based on the square grid, they could not be easily transported.
B. Rationale of an Heuristic Approach

Two approaches were possible in designing PFS:

(i) the system could have been formally defined in mathematical terms, and

(ii) the system could have been defined heuristically in terms of certain assumptions about the applications for which the system would be used.

Because of imperfections in the input data traced by the user, as introduced in chapter I and explained in chapter III, it is not possible to state the problem (model of computation) in formal terms such that there will always exist a solution which has the desired result.

For this reason, the second alternative was chosen. Certain assumptions have been made about the types of applications which motivated the development of the system. It is assumed that the system will be used for cartographic (contour filling) and image processing (lakes, mountains, shadows) applications. The assumptions made about the data traced by the user are enumerated in chapters I and III. These assumptions guarantee a high degree of accuracy and predictability for the above mentioned applications. This tailoring of the system towards specific applications however, causes a loss of generality. Certain applications, character recognition for example, are not catered to. Character recognition often requires an averaging of smudge lines [STEF71], rather than the traversal of smudges along their outer edges, as a processing step.
Chapter III: Boundary Definition: A Functional Overview

A. User Considerations

The algorithm which processes the boundaries makes use of spatial knowledge in determining the connectivity of the boundary. In order to achieve the desired result, the user must follow two rules discussed below.

In the discussion of user rules, reference to the ordering of pixels refers to the order in which the pixels were set in time. Rule 1: On any given boundary, the \((n+1)st\) pixel must be closer (in space) to the \(n\)th pixel than is the \((n+k)th\) pixel, for \(k \geq 2\). Failure to comply with this rule will result in the incorrect boundary being generated. Figures 3.1 (a) and (b) illustrate an example of what might occur.

![Diagram](Figure 3.1a) The \((n+1)st\) pixel is closer (in space) to the \(n\)th pixel than is the \((n+k)th\) for \(k \geq 2\). The dots
represent the pixels set by the movement of the track ball. The solid line indicates the desired boundary, while the dashed line represents the boundary generated by PFS. In this figure the correct boundary is generated.

Figure 3.1b There exists a k for which the (n+k)th pixel is closer (in space) to the nth pixel than is the (n+1)st and k>=2. The dots represent the pixels set by the movement of the track ball. The solid line indicates the desired boundary, while the dashed line represents the boundary generated by PFS. In this figure incorrect boundaries are generated.

Figure 3.1a, which adheres to rule 1 stipulated above, achieves the desired result, while figure 3.1b, which breaks this rule, results in the generation of two boundaries and hence two areas instead of one. In figure 3.1b spacial connectivity
does not correspond to temporal connectivity.

Rule 2: Any pixel belonging to a boundary must be closer to neighbours on its curve than to any pixel from a neighbouring curve. This rule is consistent with rule 1. It stipulates that rule 1 applies to pixels from different curves as well as to those along the same curve.

A breach of rule 2 has slightly different results from those shown in figures 3.1 (a) and (b). In the case of a single boundary curve, the tendency is to generate more than one area, whereas disobeying rule 2 tends to decrease the number of areas, as indicated in figure 3.2.

![Diagram](image)

Figure 3.2  This figure indicates a breach of rule 2. The dots represent the pixels which have been set. The solid lines show the two desired boundaries, while the dashed line indicates the single boundary which has been generated.
If temporal knowledge was available to the system, the problems highlighted in figures 3.1 (a) and (b) and 3.2 would be easier to resolve. The solution to other problems, such as smudges, would not, however, be facilitated by knowledge of the temporal order of curve generation.

PFS closes all boundaries in a straight line from the last pixel of the particular boundary to the first pixel of that boundary. Figures 3.3 (a) and (b) illustrate an exaggerated case.

![Diagram of boundary closing](image)

Figure 3.3a Illustration of a boundary being closed from its last to its first points.
Figure 3.3b  Illustration of what would occur if a pixel on boundary B were closer to the last pixel from boundary A than was the first.

In figure 3.3a, the boundary for area A was successfully closed because the last pixel for this area was closer to its first pixel than it was to the closest pixel in boundary B. Figure 3.3b shows what would have resulted if this were not the case. If the user obeys rule 2, then boundary distinctness will be preserved.

At a subsequent stage of the processing, any area whose size is zero, a straight line for example, will be deleted from the system. Such an area would result if the user input a series of points which were colinear, the first point of which was closer to the last than that from any other boundary.
B. Boundary Processing

The initial task of the system is to distinguish between the different area boundaries input by the user. Each boundary is processed separately. The order in which this processing is executed is established by the positions of the highest (on the display screen) pixel of each boundary. The boundary whose highest pixel is highest on the screen will be processed first. That whose highest pixel is second highest on the screen will be processed second and so on. In the event of a tie, the leftmost boundary will be processed first.

Boundaries are processed in a clockwise direction. PFS assumes that the highest pixel for a particular boundary is the first pixel for that boundary. The next pixel in the clockwise direction is regarded as the second pixel in time. The area's boundary is traversed in a clockwise direction. There is, of course, no requirement that the user actually trace the boundary in a clockwise direction. The system will operate, however, as if the user has traced the boundary in a clockwise direction, in accordance with rule 1 above, starting at the highest point.

In deriving the individual boundary, it is essential that the basic shape input by the user be retained. In so doing, it is also desirable to retain only essential points along the boundary. This selection of essential points is conducted according to two rules:

(i) For any sequence of \( n > 2 \) colinear points in the direction of either of the cartesian axes, all points
except the 1st and the nth are discarded. The 1st and nth are retained as being essential in preserving the desired shape of the boundary. For an example see figure 3.4.

Figure 3.4 In this example, the pixels marked with a plus sign are retained, while those indicated by a dot are discarded.

Because no error margin is allowed by the system, the points must be strictly colinear. Directions other than those of the cartesian axes have not been considered. Lines in the diagonal directions are difficult to generate with the available input device.

A track ball is sensitive both to the shaking of the users hand and to the slipping and sticking caused by its own internal friction. As a result, it is almost impossible for the user to trace an entire boundary which is thin. A thin line is one whose width is at most one pixel. Any line which is not thin is
regarded as a smudge.

(ii) A smudge is processed by traversing along its outer edge and discarding all the other pixels which make up the smudge. An example of smudge processing is given in figure 3.5.

Figure 3.5  In the above figure, the pixels shown with a plus are retained while those shown with a dot are discarded.
Chapter IV: Implementation

Figure 4.1 indicates the overall flow of control in PFS. This chapter describes the data structures used and the algorithms implemented at each stage in PFS.

Figure 4.1 Diagram showing overall flow of control in PFS.
A. Curve Smoothing

Curve smoothing is performed by a simple expansion and contraction of the pixel matrix representing the display screen. The system does not prevent user defined boundaries from accidently being joined together. It is the user's responsibility to ensure that the pixels from each boundary are at least three pixels distant from the pixels of any other boundary. This distance allows for one expansion of the matrix. Implicit in this single expansion and contraction of the matrix is that holes or gaps, two pixels wide will be closed. In examining the algorithms which perform expansion and contraction, it should be born in mind that the system operates on a square grid system. Each grid position is considered to have exactly eight neighbours, as shown in figure 4.2.

![Figure 4.2](image_url)

Figure 4.2 Each element in a square grid has eight neighbours.

In order that no special cases arise, the matrix has a border of
extra pixels (which are initialized to zero) around it. The algorithms for expansion and contraction are discussed below:

1. **Expansion and Contraction:**

   Expansion is achieved by setting each pixel which has one or more of its eight neighbours set. Any pixel which was originally set remains set. Expansion is equivalent to the logical inclusive OR of each pixel and its eight neighbours [GOLA69].

   Contraction is achieved by setting each pixel, which is already set and which has all eight neighbours set. Contraction is equivalent to the logical AND of each pixel and its eight neighbours [GOLA69].

   Expansion and contraction can be executed either in sequence or in parallel. A sequential implementation, without copying, would cause the results at step n to influence the results at step n+1. A parallel implementation of the algorithm would require every position in the matrix to be examined simultaneously and any modifications to the matrix to be made simultaneously. In a parallel implementation, the problems encountered in sequential processing do not arise, but the simultaneous processing of a matrix of a large size cannot be achieved simply or cheaply on hardware that is essentially sequential in nature.
True parallelism can be implemented in three ways:

(i) The hardware is truly parallel. True parallelism in hardware means that all pixels in the matrix are examined and modified simultaneously.

(ii) True parallelism can be simulated by processing the matrix recursively (see Appendix B). But, a matrix of resolution $N \times N$ pixels would require that the system recurse to a depth of $N^2$. This requires an excessive amount of stack memory. Few operating systems would make this method practically feasible.

(iii) True parallelism can be simulated by creating a second (temporary) matrix, equal in every respect to the matrix currently being processed. This second matrix is initialized to zero prior to the commencement of the processing. This method leaves the original matrix intact and simply sets the relevant pixels in the second matrix. Hence, the problems of sequential processing do not arise.

In PFS, method (iii) above is implemented. The cost of this method is the extra memory required for the new matrix.

The two required two-dimensional matrices are represented as a single three-dimensional matrix which has a front and back plane. Each plane contains a single two-dimension matrix. The number of expansions/contractions is controlled by the constant $C$ at the head of the program (see Appendix A). This parameter also controls the size of the border around the matrices.
B. Establish data

After the raw data has been smoothed, the pixel matrix is represented by a new data structure, which serves as an ordered index to the pixel matrix. The purpose of this index is to facilitate the searching of the matrix for a pixel's closest neighbour (not necessarily immediate). An efficient search requires only set pixels to be examined. The index, thus serves as an ordered pointer to set pixels in the matrix. The positions of the pixels pointed to by the index are sorted both in the x direction (by column) and in the y direction (by row). Because the locations are sorted, they can be efficiently located using binary search. This data structure will henceforth be referred to as the 'index'. The index is an array of records, the number of which is equal to the x resolution of the COMTAL screen. The index thus has one element for each column of the screen. Each element of the index is a record whose contents are listed below:

(i) the number of pixels in that column
(ii) the highest row in that column in which a pixel appears
(iii) the lowest row in that column in which a pixel appears
(iv) an array of pointers each of which points to an array of integers. [the size of both of these arrays is determined by the y resolution of the screen. The expression \(((y \text{ resolution}) \text{ DIV } 16)\) yields the size of
the arrays. This assumes that the y resolution is a multiple of 16. If it is not, then one extra pointer must be added to the pointers array.

The Pascal declarations which yield this data structure are given in figure 4.3.

CONST
    ptrs_limit = y_resolution DIV 16;
y_res_div_16 = y_resolution DIV 16;

TYPE
    y_block = ARRAY(.1..y_res_div_16.) OF integer;
    {Y_block contains the rows which contain pixles}
    ptr = @y_block;

    index_element = RECORD
        no, {no of pixels in col}
y_min, {highest pixel in col}
y_max : integer; {lowest pixel in col}
        ptrs : ARRAY(.1..ptrs_limit.) OF ptr;
    END;

VAR
    index : ARRAY(.1..x_resolution.) OF index_element;

Figure 4.3 The Pascal declarations yielding the index's data structure.
The graphical representation of the index_element record is given in figure 4.4.

Figure 4.4 The physical representation of the index_element record.

The values entered into the y_block, pointed to by the index_element record, are the row positions of the pixels within each column of the matrix. A new array is created dynamically when the previous array created for that column is full, or if the point being processed is the first point encountered in that column. The maximum number of wasted integer storage locations for any single column is thus \((y \text{ resolution}) \div 16 - 1\). If the integer arrays were not created dynamically, the maximum number of wasted integer storage locations for any one column would be equal to the y resolution of the display screen. This situation would occur if there were no pixels in that particular column.
Prior to data being entered into the y_block, each index_element record is initialized as follows:

(i) the number of pixels in the column is set to zero
(ii) the highest pixel in the column is set to the systems maximum integer value (maxint)
(iii) the lowest pixel in the column is set to the systems minimum integer value (-maxint)
(iv) all pointers in the 'ptrs' array are set to nil.

The data is entered into the index in accordance with the following algorithm:

The pixel matrix is scanned in row order. This is equivalent to scanning the rows of the display screen from the top lefthand corner to the bottom righthand corner. When a pixel is encountered, the record, within the index array, pertaining to the column in which the pixel was found, is accessed. The count of pixels for that column is incremented. The row number in which the pixel is located is compared with the y_min (highest row for a pixel) and y_max (lowest row for a pixel), and if necessary, one of these variables is modified. These maximum and minimum values are used in computing a pixel's closest neighbour. The row number is then entered into the next available space in the y_block. If no space is available, or this is the first pixel encountered in this column, then a new y_block is dynamically created, the next available pointer in the 'ptrs' array is set to point to the new y_block and
the row number is entered into this new y_block. Because the matrix is scanned in row order, the row values stored in the y_blocks pertaining to each column are sorted in ascending row order.

When the whole pixel matrix has been scanned, the field yielding the number of pixels per column of the index array is scanned and two variables are set equal to the first and last columns in which a set pixel occurs.
C. Process boundary

The basic algorithm for processing the boundary is given in figure 4.5.

Figure 4.5  This figure shows the flow of control in processing a boundary.
The algorithms for processing a boundary are described below:

1. **Smudge Processing**

A smudge is defined to be a contiguous region of pixels which is greater than one pixel in width. Smudges are processed by traversing their outer edges and discarding all internal pixels. The outer edge is determined by the direction in which the traversal is progressing. Because the boundary as a whole is traversed in a clockwise direction, the outer edge is to the left of the line of traversal.

In processing smudges, the system not only clears the smudges, but also determines and processes straight line segments which lie in a direction of one of the cartesian axes.

The procedure which processes smudges is passed two parameters. The first is the last point which the system retained as being vital to the preservation of the shape of the boundary. This point is referred to as the current point. The second parameter is the point which has been found to be the closest point to the current point. This point is referred to as the next point. The next point may be changed by this procedure if it is decided that its retention would be redundant in preserving the shape of the boundary curve. Because the system uses a square grid system, the next point can lie along any one of eight directions to the current point. The next point need not be a neighbour of the current point. These directions are
If the next point lies in one of the quadrants, rather than in the direction of a cartesian axis, then the direction of the next point, relative to the current point, is that of the diagonal associated with the quadrant.

The relationship between direction of traversal and the traversed outer edge of the smudge is dictated by the overall clockwise direction of traversal of the boundary. Table 4.1 gives this relationship in table form.
Table 4.1  Table giving relationship between direction of
traversal and the smudge outer edge to be
traversed.

Figure 4.7 shows an octagon, all of whose edges are smudges.
The figure illustrates the edge traversal in accordance with the
relationships established in table 4.1.

A smudge may take one of three forms:

(i) The smudge is a thick (width is greater than one
pixel) line, the outer edge of which is straight and
in the direction of one of the cartesian axes.
Figure 4.8 illustrates such a smudge.

Figure 4.8 Smudge which comprises a thick line with straight outer edge in line with a cartesian axis.

(ii) The smudge is a group of adjacent pixels one of which is adjacent to and on the inside of a reflex vertex (see (a) in figure 4.9). A smudge of this type is illustrated in figure 4.9.

Figure 4.9 Smudge which comprises a group of adjacent pixels on the inside of a reflex vertex.
(iii) The smudge comprises a combination of both (i) and (ii) above as shown in figure 4.10.

Figure 4.10 Complete smudge.

A smudge is processed as if it is of type (iii) above. Because such a smudge contains two conceptually different smudges, it is processed in two stages:

1.1 The elimination of intermediate pixels:

This stage eliminates the intermediate pixels in a sequence of adjoining pixels along the direction of either of the cartesian axes. This is equivalent to processing the part of the smudge that corresponds to type (i) above. This task is performed by the procedure 'nsew_check' in the program listed in Appendix A.

This process involves traversing in the current direction (determined as the direction from the current point to the next point), discarding the pixels being examined as well as the
internal pixels adjacent to the examined pixels as long as there exists a next pixel in the sequence and that next pixel does not have an immediate set neighbour on the outside as defined in table 4.1. Figures 4.11 (a) and (b) show the two situations which cause this discarding of pixels in sequence to terminate.

Figure 4.11a This figure shows that the discarding of pixels, sequentially arranged in the direction of one of the cartesian axes, terminates at the next to last pixel in the sequence. Only pixels marked with a plus sign are retained.
Figure 4.11b  This figure shows that a pixel on the outside of a sequential line of pixels (in the direction of either of the cartesian axes), will terminate the discarding of intermediate pixels.

1.2 The elimination of smudge remaining:

The elimination of the remaining smudge (type(ii) above) is accomplished via a recursive process. This process is performed by the 'clear_up' procedure in the program listing in Appendix A.

This process need only be invoked if indeed there is a pixel (on the outside) adjacent to the pixel currently being processed. The depth to which the algorithm actually deletes pixels belonging to the smudge is determined by the value of a variable known as the smudge factor (acc_smudge, in the program). Ideally, the only pixels which should be removed from
the system at this stage are those, which when removed, will guarantee that the next pixel to be processed will be that one which is immediately adjacent and exterior to the one currently being processed. All pixels which belong to the inward side of the smudge which are not deleted are 'remembered' as having belonged to a smudge. This is done by marking their positions on the back plane of the matrix being processed. The retention of some of the pixels which form the smudge allows for a potential higher degree of accuracy in the traversal of the yet untraversed section of the boundary. Figures 4.12 (a) and (b) illustrate this point.

Figure 4.12a All pixels in the enclosing box are deleted. Subsequent processing of the boundary will terminate at B. Closing off the polygon from B to A will delete wanted areas.
Figure 4.12b Only those pixels in the enclosing box are deleted while processing the smudge at position A. All other pixels belonging to this smudge, are not deleted, but will have their positions marked on the back plane of the matrix being processed. Subsequent processing of the boundary will not terminate at B.

The resulting polygon in figure 4.12a would be closed off by a straight line joining B to A. Figure 4.12b, however, would result in the polygon being closed by having the left hand side of the smudge traversed in the usual manner.

When a boundary has been completely traversed, the back plane of the matrix is scanned, within the area of the smallest rectangle that will enclose the traversed boundary, and all positions which are marked by a pixel are unset, and the corresponding pixel on the front plane of the matrix is deleted from the system. This ensures that any pixels on the front
plane, belonging to an enclosed area with a higher nesting depth than that just processed, will be left intact.

When processing position A, say, the pixels to be considered as part of the smudge for immediate processing are defined recursively in the following step order:

(i) The point adjacent to the current smudge point being processed, and lying in the cartesian axis direction dictated by the relations expressed in table 4.2.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Cartesian axis direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>North</td>
<td>North</td>
</tr>
<tr>
<td>Northeast</td>
<td>North</td>
</tr>
<tr>
<td>Northwest</td>
<td>North</td>
</tr>
<tr>
<td>South</td>
<td>South</td>
</tr>
<tr>
<td>Southeast</td>
<td>South</td>
</tr>
<tr>
<td>Southwest</td>
<td>South</td>
</tr>
<tr>
<td>East</td>
<td>East</td>
</tr>
<tr>
<td>West</td>
<td>West</td>
</tr>
</tbody>
</table>

Table 4.2 The relationships between direction of traversal and the neighbour for adjacency check in the first level of recursive smudge processing.

(ii) The point adjacent to the smudge point currently being considered and lying in the cartesian axis direction dictated by the relations expressed in table 4.3.
Direction | Cartesian axis direction
--- | ---
North | East
Northeast | East
Northwest | East
South | West
Southeast | West
Southwest | West
East | South
West | North

Table 4.3 The relationships between direction of traversal and the neighbour for adjacency check in the second level of recursive smudge processing.

An example of smudge processing is given in figure 4.13. The number associated with each pixel, which is examined in accordance with the recursive algorithm above, is the order in which that pixel is examined in the processing of the smudge.
Figure 4.13  This figure illustrates both the pixels and the order in which they are examined while processing a smudge.

In figure 4.13 above, those pixels marked with a plus sign are not processed as part of the smudge when point A is being considered as the next point. These points are deleted when the line of contiguous points from point A to point B (see figure 4.13) is being processed as per 1.1 above.

In the example depicted in figure 4.13, the next point (point A) will be retained as a point vital in preserving the exact shape of the boundary as defined by the user. Only those points
whose distance along a cartesian axis direction from A (and on
the inside of A as defined above) is greater than or equal to
the smudge factor will be marked on the back plane of the
matrix. These points will not immediately be deleted from the
smudge or the index.

Any point deleted from the matrix must also be deleted from
the index data structure described in section B above.
2. **Closest point**

In determining the closest point (i.e., the next point defined in section 1 above) to the current point, the next point is set equal to the closer of the current point's two neighbour points in the y_block of the index data structure. If there is only one neighbour, then the next point is set equal to this point. If there are no neighbours (the point is the only one in the column), then the next point's coordinates are set equal to the system's maximum integer value (maxint). If there is at least one neighbour, then the next point will be the closest point to the current point in the same column as the current point. This is guaranteed because the row numbers in the y_block are sorted in ascending order. The distance, $D$, is computed as the absolute value of the difference between the row values of the current and next points. The columns to the left and right of that in which the current point resides are now searched, alternatively to the left and right and to a column distance, $D$, from the current point's column, for a point which lies closer to the current point than does the next point. If such a point is found, the next point becomes set to this next point, and $D$ is recomputed to be equal to the maximum of the absolute of the column difference (the x difference) and the absolute of the row difference (y difference), between the current point and the new next point.
If a column comprises more than sixteen points, a binary search is performed on the locations stored in the y_blocks. If not, then a linear search is executed. In the case of a linear search, the lowest and highest points of the column to be searched are compared with the current point and the search begins at the end of the list whose end point lies closer to the current point.

The y_block contains references to only those pixels which are set in the pixel matrix. It thus serves as an ordered pointer to set pixels in the matrix. Without the Y_blocks, the nearest neighbour search would require examination of an element's neighbours irrespective of whether they are set or not. The number of unset pixels requiring examination increases as the distance between the element and its closest neighbour increases.
3. **Find the first point of a boundary**

The first point of the next boundary to be processed is found by scanning the index (see section B above) for the least $y_{\text{min}}$ value. This value represents the highest row, in the picture, in which a pixel is set. Because each element of the index array stores the information pertaining to a column of the display, the left to right (or lowest to highest) scanning of the index array will yield the left most pixel in the event of more than one column having a pixel set in the highest row.
4. **Protection of user's input**

The data, as traced by the user, is read by PFS into a pixel matrix. In order that the integrity of this data be maintained, the contents of this matrix are copied to the front plane of another matrix, called the 'save_matrix'.

A boundary is only deleted from this second matrix when the user confirms that the closed polygon, generated from the original of the boundary, is satisfactory. In order that only those pixels belonging to the current boundary are deleted from the save_matrix, the positions of all pixels, deleted during the processing of the boundary, are stored in a linked list associated with the boundary. Only those pixels whose positions are stored in this list, are deleted from the save_matrix. Thus, the boundaries that have not yet been processed are left intact in the save_matrix. As each closed polygon is determined to be satisfactory, it is copied onto the back plane of the save_matrix. When all boundaries have been processed, the closed polygons representing these boundaries will have been copied to the save_matrix's back plane. It is this matrix which is filled by the system for subsequent display on the screen.
D. **Polygon closure:**

The sequence of pixels which are retained form a minimal polygon (i.e., all vertex points are retained) which preserves the shape of the entered boundary. At this stage the polygon is closed by passing the preserved points in pairs \((P_1,P_2),(P_2,P_3),\ldots,(P_n,P_1)\) to a digital differential analyser (DDA) which sets intermediate pixels in the straight line between the two points passed to it. That is, it closes off the polygon by joining the points along its perimeter. Lines on a raster graphics display cannot be perfectly straight. Any pixel on the line will, however, reside within one half of a grid interval of its true position. The DDA implemented by this system is the simple DDA as described in [NEWM79].

The DDA implemented in this system is the procedure 'dda' in the program in Appendix A. This procedure was taken directly from [NEWM79] pp (20 - 27).
E. **Polygon filling**

The matrix which contains the closed polygons to be filled has all background and interior pixels set to zero, while the pixels representing the closed polygons' boundaries are set to one. The basic algorithm which fills the set of closed polygon boundaries is as follows:

Scan the lines of the matrix one at a time. At the beginning of each scan line, set the beam to off. On encountering the first string of pixels, set the beam to on (a 'string' of pixels refers to a contiguous line of set pixels along the current scan line). At the next string of pixels, turn the beam to off. At each string of pixels, change the state of the beam as described above.

If the beam is on, then the pixels being scanned must either be left set (if currently set) or set (if currently unset). If the beam is off, all pixels scanned must be left in their current state. Certain special cases exist under which the state of the beam does not change. The beam does not change state if the string of pixels being processed forms an upper or lower arc of a polygon. Figure 4.14 shows an example of both upper and lower polygon arcs.
Figure 4.14 An illustration of upper and lower polygon arcs.

The formal definition of an upper or lower polygon arc is as follows:

Assume that the pixels k to r of row i are set, where \( k, r \in \{1, \ldots, x_{\text{resolution}}\}, i \in \{1, \ldots, y_{\text{resolution}}\} \) and \( (r-k)\geq 0 \). Then, this string of \( (r-k) \) pixels is an upper arc if there exists on row \( i+1 \) two contiguous strings of pixels m to n and p to q, where \( m,n,p,q \in \{(k-1), \ldots, (r+1)\}, (n-m)\geq 0, (q-p)\geq 0 \) and \( (p-n)>1 \).

Similarly, this string of \( (r-k) \) pixels is a lower arc if there exists on row \( i-1 \) two contiguous strings of pixels m to n and p to q, where \( m,n,p,q \in \{(k-1), \ldots, (r+1)\}, (n-m)\geq 0, (q-p)\geq 0 \) and \( (p-n)>1 \).
Figure 4.15 illustrates an example of an upper arc.

No problem occurs if $r$ has the value 1 or $k$ has the value $x_{\text{resolution}}$ because the matrix is surrounded by a border of zeros.

Upper and lower arcs are directionally dependant. Their tangents are in the direction of the horizontal cartesian axis.

Conversely, any string of contiguous pixels which is not an upper or lower arc form part of the left or right edge of a polygon and will thus cause a change of state in the beam. Figure 4.16 illustrates contiguous edge strings.
Figure 4.16 Pixels represented by a plus sign are elements of contiguous edge strings. Pixels represented by dots are upper or lower arcs.

Edge pixel strings change the state of the beam.

Figure 4.17 shows three examples of general edge conditions.

Figure 4.17 Four general edge conditions are shown. The general shape of the edge condition over three scan lines is an 'S'. Figures (a) and (b)
illustrate this general shape. This 'S' shape may, however, be stretched out into a 'straight' line as is shown in (c) and (d).
F. Tail detection

The polygon filling algorithm described in section E above depends on the beam being turned on and off an even number of times. If this did not happen, polygons would effectively be turned inside out, resulting in parts of the background being shaded while inner areas of polygons would be left unshaded. The count of the number of strings of contiguous pixels which cause the beam to change state must be even.

An even count of beam state changes does not, however, guarantee that no tails exist. It is possible that an even number of tails exist on a single scan line. Such an occurrence of tails will remain undetected.

A problem occurs if 'tails' are generated by the boundary detection and polygon closing algorithms. A tail is, as the name implies, a column of pixels attached to a closed polygon. The column must be at least one pixel wide. Tails join the polygon at an upper or lower arc or on an edge. They are directionally dependent and occur in the direction of the vertical cartesian axis. Figure 4.18 shows an example of a user input boundary that results in a tail, and the resulting closed polygon.
If a tail is detected, the count of horizontal pixel strings falls into one of two categories:

(i) the count of set pixel strings is equal to 1
(ii) the count of set pixel strings is $\geq 3$, and odd.

Case (i) above is handled trivially by deleting the the sequence of pixels. Case (ii) above, however, presents a problem. The simplest instance of this case is where the count of beam-state changes is three and the length of each of the three corresponding pixel strings is one. In this instance it is not possible for the system to locally discern which of the three pixel strings is the member of the tail. The problem that exists in case (ii) is one of connectivity [MINS69]. Figure 4.19 illustrates the confusion that arises.
Figure 4.19  Illustration of the confusion caused by three changes of the beam's state. This figure is a logical fragment of an entire family of polygons.

Obviously, the problem is compounded as the count of the beam's state changes increases.

Case (ii) is handled by allowing the user to interactively aid the system by identifying the pixels which belong to the tail. The closed polygon is stored on the back plane of a pixel matrix. On the front plane of the same matrix, arrows are drawn (pixels are set so as to form arrows), highlighting each of the pixel strings which contribute to the odd count, as potential tails. The start position of each of these strings is stored in a linked list of 'problem points'. When the search for tails is completed, the closed polygon will be displayed on the first graphic plane of the COMTAL, while the arrows (if indeed any exist) will be displayed on the second graphic plane of the COMTAL. It is then up to the user to recognise the offending
tail and identify the appropriate arrow to the system. For each potential tail, arrows need only be drawn once. Subsequent processing which deletes the tail will process the entire tail, knowing the starting position stored in the linked list. Figure 4.20 shows an example of arrows highlighting a set of problem points.

![Figure 4.20 Arrows highlighting a series of problem points. The tail appears in the figure on the left.](image-url)
G. Deletion of tails

The start position (in the pixel matrix) of potential tails is stored in a linked list. When the tail is interactively identified by the user (via the target on the display screen), the linked list is searched for the closest starting position to the 'hit' point. This implies that the hit does not have to land exactly on the top leftmost point of the tail.

Three types of tails exist. Those which join the polygon at one of its upper arcs, those which join the polygon at one of its lower arcs and those which join the polygon at one of its edges. Figure 4.21 illustrates these three cases.

Figure 4.21 (a) illustrates a tail joining a polygon at an upper arc, (b) illustrates a tail joining a polygon at a lower arc and (c) shows a tail which joins the polygon at an edge. Pixels represented by plus signs belong to tails.
Because tails are processed from top to bottom, the terminating conditions for the processing of a tail are:

(i) the detection of an upper arc,
(ii) the detection of an edge, and
(iii) the disappearance of the tail (i.e., the end of the tail).

The tail is deleted by deleting the string of pixels starting at the position extracted from the linked list. The row count is then incremented and if there are any pixels set directly below those deleted above (and to each edge, within a limit) these are deleted. The above process is repeated until no pixels in the region of the tail width exist or an upper arc is detected or an edge is detected, at which stage the process is halted and the next tail, if any more exist, is deleted in the same way.

In this implementation tails must be removed, either by the system directly or by the system with the aid of the user. The system could be trivially modified to make the removal of tails optional.
H. Improvements: 2-D Versus 3-D Matrices

The pixel matrices are currently three dimensional. If the matrix is defined as having dimensions q, r and s, then to find the \( M(i,j,k) \)th element, the following calculation is made by the code generated by the compiler:

\[
M(i,j,k) = \text{Base Address} + r.s.(i-1) + s.(j-1) + (k-1)
\]

\[
= (\text{Base Address} + r.s - s - 1) + r.s.i + s.j
\]

letting \( a = (\text{Base Address} + r.s - s - 1) \)

and \( b = r.s, \)

we have:

\[
M(i,j,k) = a + b.i + s.j.
\]

This represents two additions and two multiplications.

A two dimensional matrix of dimensions, r and s, has an element's position calculated by the following:

\[
M(i,j) = \text{Base Address} + r.s.(i-1) + s.(j-1) + (k-1)
\]

\[
= (\text{Base Address} - s - 1) + s.i + j
\]

letting \( a = (\text{Base Address} - s - 1) \),

we have:

\[
M(i,j) = a + s.i + j.
\]

This represents two additions and one multiplication.

The current three dimensional matrix, because it has two planes, can be converted to a two dimensional matrix with alternate elements representing elements from the two planes in the three dimensional matrix. If this were done, then with an
extra addition, to account for the offset, the new two dimensional matrix would have its elements accessed with one multiplication and three additions. Whether or not this modification is implemented in the future should depend on the hardware available. If a fast hardware multiply is available, then the modification will probably not be worth the effort. However, if multiplication is slow, then the modification should result in a major improvement in efficiency.
Chapter V: Problems, difficulties and features

This chapter shows, by example, some of the problems, difficulties and features a user encounters in executing PFS. Some of these problems have been alluded to in chapter I. Others result from the user disobeying the two user's rules described in chapter III. The example also shows how the user can cope with these problems on an interactive basis.

Finally, this chapter discusses the extent to which PFS is bound to the hardware and software environment in which it has been implemented.

A. Sample run

A run of PFS comprises a number of interactive steps. Depending on the stage of processing, the user may be instructed by the system to trace on the display screen, verify a task performed by the system or aid the system in making some decision (tail processing). The sequence of interactive steps which generated the sample run is given below:

(i) The user traces the desired boundaries on the graphic display screen. The boundaries entered by the user are shown in figure 5.1.
Two boundaries, labeled A and B in figure 5.1, have been traced. Boundary A encloses boundary B and hence their nesting depths are one and two respectively. In the discussion which follows, the boundaries will be referred to simply as A and B.

(ii) Figure 5.2 illustrates the closed polygon generated in processing A.
As can be seen in the figure, both A and B were actually processed although only A was meant to have been processed at this stage. This has resulted because the boundaries were traced, by the user, in such a manner that a pixel from B is the closest neighbour to a pixel from A. This is a breach of user-rule (ii) in chapter III (see figure 3.2).

When figure 5.2 is displayed on the screen, the user is asked by the system whether the polygon is distinct. In this case, the user's answer is negative. In order that the problem of
distinctness be resolved, the user is instructed by the system to pick (identify the position on the screen with the aid of the screen's target) the top left and bottom right hand corners of the box (imaginary) that encloses the problem area. In the figure, this box is drawn with a dashed line.

(iii) All pixels within the confines of the box, identified in the previous step, are erased from the front plane of the save_matrix (see chapter IV section C.4). This modified matrix is then copied to the display screen. Figure 5.3 shows the result of this copy.

![Figure 5.3](image)

**Figure 5.3** Problem pixels erased from the screen.

The user is now instructed by the system to correctly retrace the boundaries where they have
been erased. Figure 5.4 shows what such a retrace might look like.

Figure 5.4 Erased boundary lines have been retraced.

It should be noted that if any boundaries had already been successfully processed, they would have been displayed on another graphic plane for this step in order that the user does not overtrace a previously processed boundary line. All boundaries already processed are not effected by this step.

(iv) Boundary A is now reprocessed. The new closed polygon representing A is shown in figure 5.5.
Figure 5.5 Correct closed polygon representing boundary A.

The unambiguous tail at the top of the boundary has been automatically removed by the system. On this occasion when the user is asked whether the polygon is distinct the answer is in the affirmative.
Arrows are added to highlight the existence of a tail.

Arrows are now added to the second graphic plane to indicate that ambiguous tails exist. Figure 5.6 shows these tails, which are displayed in a colour different to that in which the polygon is displayed. The system knows that a tail exists, but it does not know which of the columns of pixels, pointed to by the arrows, represents the tail. The user is instructed by the system to identify (pick) the appropriate tail with the screen's cursor. The middle column of pixels is identified and the system automatically removes the tail. The corrected closed polygon is now copied.
to the rear plane of the save_matrix.

(v) Figure 5.7 shows the closed polygon representing boundary B as it is displayed on the screen for verification by the user. This polygon presents no problems and is copied to the rear plane of the save_matrix.

Figure 5.7    Closed polygon representing boundary B.

(vi) At this stage all boundaries, traced by the user, have been processed by the system. The user is now invited to trace more boundaries. In this run, the user indicates that more boundaries are desired, and in response, the system displays the already processed boundaries on the second graphic plane, in order that the user does not overtrace these boundaries. These already-processed boundaries are
displayed in a different colour to those which are now traced. Figure 5.8 shows two new boundaries C and D traced on the first graphic plane. The processed boundaries are shown, as they are displayed on the second graphic plane.

![Diagram](image)

**Figure 5.8** Two new boundaries, C and D, traced by the user.

Boundaries C and D have nesting depths two and one respectively.

**(vii)** Figure 5.9 illustrates the two displays in which the two new boundaries are verified.
Figure 5.9  (a) shows the closed polygon representing boundary C, while (b) shows the polygon which represents boundary D.

Figure 5.9 illustrates two steps of user interaction. Both of the closed polygons, generated by the system, are acceptable to the user and are copied in turn to the rear plane of the save_matrix.

(viii) Again, all boundaries have been processed by the system. This time the user declines the option to trace more boundaries. The system now fills the closed polygons which have been copied to the rear plane of the save_matrix. Figure 5.10 shows the filled polygons.
This example is not exhaustive, but the techniques used to resolve the problems which did arise are used to resolve other similar problems. A breach of user-rule 1 in chapter III (see figure 3.1b) can, for example, be resolved by treating the closed polygon generated as being not distinct and then retracing the offending portion of the boundary.
B. System Dependence

PFS is bound to the hardware configuration described in chapter I section A in as much as communication between PFS and COMTAL Vision One is done via a FORTRAN subroutine package which runs under MTS. Any change in raster display system would necessitate modifications to the FORTRAN subroutine package, if that system were not fully compatible with the COMTAL Vision One system.

PFS itself is implemented in UBC Pascal, which is a superset of Wirth's pascal as described in [JENS75]. The two main extensions to standard Pascal which have been utilized are:

(i) the use of McCarthy precedence in the evaluation of boolean expressions, and

(ii) that functions are occasionally of non scalar type.

If PFS were to be run under an operating system which did not support UBC Pascal, the appropriate changes with respect to the above two points, would have to be made to the program.

A number of the algorithms implemented suggest that a raster scan device be used. If a vector type display device was desired, then a number of changes, both in the philosophy of some of the algorithms and in the implementation of others, filling for example, would be required.
Chapter VI: Conclusion

This thesis has described the Polygon Filling System, as implemented on the COMTAL Vision One hardware. Limitations exist in the system with respect to the input data that it will process correctly. The limitations that exist result directly from breaches of user's rules (i) and (ii) described in chapter III. No other class of input has been found, thus far, which the system will not process correctly.

Chapter I served as a general introduction to the system. It contained a sample run of PFS to serve as a general guide to the user as to what the system does.

Chapter II discussed the rationale of the problem definition. It also discussed the underlying grid model, which involved describing hexagonal and square tesselations, and an explanation as to why a square grid system was chosen.

Chapter III described problems that exist in processing the user defined boundaries. Certain of these problems may be interpreted as being limitations of the system. Two user rules were enumerated. Breaches of these rules will result in the incorrect processing of the boundaries. Adherence to these rules will result in correct processing of the boundaries.

Chapter IV described the implementation of the system. Included in this chapter was a description of the key data structures used by the system and descriptions of the main algorithms which have been implemented.

Chapter V discussed problems, difficulties and features of
PFS by way of a detailed example. Problems which can arise while running the system are illustrated in this example.

At the outset of development of PFS, it was thought, by the author, that the entire system would be automatic. However, as described in chapter IV, sections F and G, and illustrated in the example in chapter V, the occurrence of tails may force the system to seek assistance from the user. To a certain extent this loss of automatism can be seen as a failing of the system. The author feels that the problem of tails is theoretically computable and can thus be resolved automatically. There are, however, clear difficulties in resolving it locally.

The system itself, has been tested on relatively few examples and applications. With continued use, further assumptions about input data might have to be made in order for the system to fully meet the requirements of the applications for which it was designed. On the other hand, new applications might be found for which the system can be used.

Two suggestions are offered for future work related directly to this thesis:

(i) effort be applied to resolving the problem of identifying tails, when an ambiguity exists, and

(ii) the system be extended to allow the user to define an area by specifying the vertices of that area. This would require that the user's input be monitored via a pick command \cite{GSPC78}. Monitored input provides
temporal knowledge of the vertices, and hence, a different, more trivial algorithm can be implemented to close off polygons defined in this way.
Bibliography


Appendix A: Program listing for Polygon Filling System
$LIST T.WHOLE.BU *SINK*

1  PROGRAM polygon(input,output);

2  

3  CONST

4  C = 1;

5  null = chr(255);

6  trailers = 2;

7  front = 1;

8  back = 2;

9  x_resolution = 74;

10  y_resolution = 74;

11  x_res_plus_C = x_resolution+C;

12  y_res_plus_C = y_resolution+C;

13  st = 1;

14  y_div = 16;

15  ptrs_limit = 16;

16  buff_size = 8;

17  len_buff = 32;

18  no_graphics = 4;

19  unitno = 0;

20  word_size = 32;

21  word_size_1 = 31;

22  debug = 20;

23  line_length = 80;

24  base = 10;

25  blank = ' ';

26  zero = ORD('0');

27  str_length = 80;

28  max_smudge = 100;

29  rd = 1;

30  wrt = 0;

31  dscript_length = 80;

32  max_polys = 50;

33  

34  Type

35  string = ARRAY(.0..str_length.) OF char;

36  txt_str = ARRAY(.1..str_length.) OF char;

37  point = RECORD

38      x, y : integer

39  END;

40  poly_elt = RECORD

41      poly_x, poly_y : integer;

42      poly_ptr : @poly_elt

43  END;

44  poly_array_elt = RECORD

45      poly_descript : ARRAY(.1..dscript_length.) OF char;

46      descript_posn : point;

47      base_ptr : @poly_elt;

48      size : integer;

49  END;
poly_array = ARRAY(-1..max_polys.) OF poly_array_elt;  (0th elt for keeping track
of deleted points
-1st elt for keeping track
of tails)

mat_core = ARRAY(.1..((2+2*C+x_resolution) div word_size + 1),1..2.) OF integer;
bool_matrix = ARRAY(st-C-1..y_res_plus_C+1.) OF mat_core;

charset = SET OF char;
y_block = ARRAY(.1..y_div_16.) OF integer;
buff_array = ARRAY(.1..buff_size.) OF integer;

ptr = @y_block;
ptr_type = @>poly_elt;

index_element = RECORD
  no,
y_min,
y_max : integer;
  ptrs : ARRAY(.st..ptrs_limit.) OF ptr
END;

location = RECORD
  ind,
  off : integer;
  found : boolean
END;

point_found = RECORD
  x,
y : integer;
  found : boolean
END;

complaint = (bad_data,limit_error,io_error,cancelled,bad_command);

VAR
  index : ARRAY(.st..x_resolution.) OF index_element;  { index to matrix }
  frst_pt : point_found;  { first point for any area }
  frst_loc : location;  { location in 'index' of first point }
  curr_x, curr_y : integer;  { co-ords of current point }
  next_x, next_y : integer;  { co-ords of next point }
  displace : integer;  { min dist twixt curr pt & nxt pt along row/col }
  min_x, max_x : integer;  { min and max x in matrix }
  pt_set_no : integer;  { polygon number }
  debugging : ARRAY(.1..debug.) OF boolean;  { debugging control array }
  graphics : integer;  { code for call to control_comtal }
  graph : integer;  { control on COMTAL for read }
  buffer : buff_array;  { buffer for reading COMTAL }
  matrix : bool_matrix;  { main matrix for manipulating }
  work_matrix : bool_matrix;  { temp matrix }
  save_matrix : bool_matrix;  { keep a record of inputted points
  and 'good' polygons }
  unit_no,
PROCEDURE primary_initialize; forward;
PROCEDURE secondary_initialize; forward;
PROCEDURE tertiary_initialize; forward;
PROCEDURE write_index(x,y : integer); forward;
PROCEDURE delete_index(x,y,subscript,offset : integer); forward;
PROCEDURE estab_index; forward;
PROCEDURE find_closest(x,y,
curr_col : integer;
    VAR
    displace,
    subscript,
    offset : integer;
    VAR
    next_x,
    next_y : integer); forward;
PROCEDURE find_area; forward;
PROCEDURE smudge(curr_x,curr_y : integer; VAR next_x,next_y : integer); forward;
PROCEDURE nsew_check(x_inc,y_inc : integer); forward;
PROCEDURE clear_up(temp_x,temp_y : integer); forward;
PROCEDURE mat_init(VAR mat : bool_matrix; plane : integer); forward;
PROCEDURE copy_mat(VAR mat1,mat2 : bool_matrix; plate : integer); forward;
PROCEDURE contract; forward;
PROCEDURE expand; forward;
PROCEDURE find_linear(x,y : integer; VAR ind,off : integer); forward;
PROCEDURE pre_process; forward;
PROCEDURE read_comtal(VAR unit_no : integer;
    VAR buffer : buff_array;
    VAR length,
    ret_code : integer); fortran 'CMTRD';
PROCEDURE write_comtal(VAR unit_no : integer;
    VAR buffer : buff_array;
    VAR length,
PROCEDURE disp_poly(VAR mat : bool_matrix; plane : integer); forward;
PROCEDURE cmt_initialize(unit_no : integer; VAR ret_code : integer); forward;
PROCEDURE term_comtal(unit_no : integer; VAR ret_code : integer); forward;
PROCEDURE control_comtal(unit_no,code:integer;VAR ret_code:integer); forward;
PROCEDURE read_matrix; forward;
PROCEDURE debug_data; forward;
PROCEDURE skip_char(skipchars : charset); forward;
PROCEDURE read_int(min,max  : integer; query  : string; VAR valu  : integer); forward;
PROCEDURE read_real(min,max  : real; query  : string; VAR valu  : real); forward;
PROCEDURE pack(VAR to_string  : string; from_string  : string; VAR start : integer); forward;
PROCEDURE read_buffer; forward;
PROCEDURE complain(message  : complaint); forward;
PROCEDURE read_text(query  : string; VAR return_text  : txt_str); forward;
PROCEDURE set_bit(VAR mat : bool_matrix; i,j,k : integer); forward;
PROCEDURE unset_bit(VAR mat : bool_matrix; i,j,k  : integer); forward;
PROCEDURE link_to_list(i : integer); forward;
PROCEDURE return_to_pool(i  : integer); forward;
PROCEDURE insert_point(i,x,y  : integer); forward;
PROCEDURE close_polygon(i : integer); forward;
PROCEDURE set_pixel(y,x  : integer); forward;
PROCEDURE dda(x1,y1,x2,y2  : integer); forward;
PROCEDURE remove_trailers(VAR mat : bool_matrix); forward;
PROCEDURE fill_polygon(min_x,max_x,min_y,max_y  : integer; mat : bool_matrix); forward;
PROCEDURE blank_out(VAR mat : bool_matrix; y1,x1,y2,x2,plane : integer); forward;
PROCEDURE draw_arrows(VAR mat:bool_matrix;VAR marked:boolean;row,x_min,x_max,plane:integer);forward;
FUNCTION first_pt : point_found; forward;
FUNCTION sub_convert(x  : integer)  : integer; forward;
FUNCTION off_convert(x  : integer) integer; forward;
FUNCTION find_binary(x,y,new_x  : integer; VAR next_x,next_y,displace : integer)  : location; forward;
FUNCTION same_col(x,y, ind,off integer; VAR subscript, offset, displace integer) : point_found; forward;
FUNCTION other_cols(x,y, ind,off : Integer; VAR subscript, offset, displace, CM next_x, next_y integer)  : point_found; forward;
FUNCTION check_contract(i,j : integer) boolean; forward;
FUNCTION check_expand(i,j : integer) : boolean; forward;

FUNCTION band(i,j : integer) : integer; external 'BAND$$';

FUNCTION shift(i,j : integer) : integer; external 'SHIFT$$';

FUNCTION bor(i,j : integer) : integer; external 'BOR$$';

FUNCTION rotate(i,j : integer) : integer; external 'ROTATE$$';

FUNCTION bxor(i,j : Integer) : integer; external 'BXOR$$';

FUNCTION shiftr(i,j integer) : integer; forward;

FUNCTION init_comta1(i_o,graphic_p1n : integer) : boolean; forward;

FUNCTION graphic_plane(i_o,graphic_pln : integer) : integer; forward;

FUNCTION read_num : integer; forward;

FUNCTION read_decimal : real; forward;

FUNCTION chrr(valu : integer) : string; forward;

FUNCTION c(str1,str2,str3,str4,str5 : string) : string; forward;

FUNCTION question(query : string) : boolean; forward;

FUNCTION test(VAR mat : bool_matrix; i,j,k : integer): boolean; forward;

FUNCTION mask_bit(bit : integer) : integer; forward;

FUNCTION mask_comp(bit : integer) : integer; forward;

FUNCTION end_check(VAR mat : bool_matrix; row,k,temp,j,plane : Integer) : boolean; forward;

FUNCTION top_bot_check(VAR mat : bool_matrix; temp,i,row,j,plane : integer) : boolean; forward;

FUNCTION evens_check(VAR work:bool_matrix;y_min,y_max,x_min,x_max,plane:integer):boolean;forward;

PROCEDURE debug_data;

VAR
  number,i : integer;

BEGIN
  read_int(1.debug,'$Debug level = ?',number);
  FOR i := 1 TO debug DO
    IF i<=number THEN debugging(i.) := true
    ELSE debugging(i.) := false;
    writeln('** Debugging turned on **')
END;

PROCEDURE skip_char(skipchars : charset);

{ To skip over characters in input_buffer. }

BEGIN
  WHILE input_buffer(.input_index.) IN skipchars AND
  input_index <= buffer_length DO
    incr(input_index)
END;

FUNCTION read_num : integer;

{ Reads an integer out of input_buffer starting at the }
{ location input_index. If there is no valid number, it }
{ will return 0. }

CONST
  neg_sign = '-';

VAR
  val : integer;

BEGIN
negative_num := false;
val := 0;

IF input_buffer(.input_index.) = neg_sign AND
  input_index <= buffer_length THEN
  BEGIN
    negative_num := true;
    incr(input_index)
  END;

WHILE input_buffer(.input_index.) IN digits AND
  input_index <= buffer_length DO
  BEGIN
    val := val * base + ord(input_buffer(.input_index.)) - zero;
    incr(input_index)
  END;

IF negative_num THEN
  val := -val;

IF debugging(.3.) THEN writeln('  read_num returning : ', val);
read_num := val

FUNCTION read_decimal : real;

{ Reads a fraction out of input_buffer starting at the }
{ location input_index. If there is no decimal point or }
{ valid number it will return 0. }

VAR
  val : real;
  divisor : integer;

BEGIN
  val := 0;
  divisor := base;
  IF input_buffer(.input_index.) = '.' THEN
    BEGIN
      incr(input_index);
      WHILE input_buffer(.input_index.) IN digits AND
        input_index <= buffer_length DO
        BEGIN
          val := val +
            (ord(input_buffer(.input_index.)) - zero) /
            divisor;
          divisor := divisor * base;
          incr(input_index)
        END;
    END;

  IF negative_num THEN
    val := -val;

  IF debugging(.3.) THEN writeln('  read_dec returning : ', val);
read_decimal := val
PROCEDURE read_int(min, max : integer; query : string; VAR valu : integer);
{This procedure extracts an integer number from the input buffer}
VAR o_k : boolean;
BEGIN
  signoff := false;
o_k := false;
WHILE NOT o_k DO
  BEGIN
    WRITELN(query);
    read_buffer;
    IF buffer_length<>0 THEN
      BEGIN
        valu := read_num;
        IF input_buffer(.input_index.) = ',' THEN
          IF (min<=valu) AND (valu<=max) THEN o_k := TRUE
          ELSE complain(limit_error) {outside valid range}
        ELSE complain(bad_data) {invalid input data}
      END
    ELSE
      BEGIN
        signoff := true;
o_k := TRUE
      END
    END
  END;
END;

PROCEDURE read_rl(min, max : real; query : string; VAR valu : real);
{This procedure extracts a real number from the input buffer}
VAR o_k : boolean;
BEGIN
  o_k := false;
signoff := false;
WHILE NOT o_k DO
  BEGIN
    WRITELN(query);
    read_buffer;
    IF buffer_length<>0 THEN
      BEGIN
        valu := read_num + read_decimal;
        IF input_buffer(.input_index.) = ',' THEN
          IF (min<=valu) AND (valu<=max) THEN o_k := TRUE
          ELSE complain(limit_error) {outside valid range}
        ELSE complain(bad_data) {invalid input data}
      END
    ELSE
      BEGIN
        signoff := true;
o_k := TRUE
      END
    END
  END;
ELSE complain(bad_data)  (invalid input data)

BEGIN
  signoff := true;
  o_k:=TRUE
END

FUNCTION chrr(valu : integer) : string;
VAR temp,i,j : integer;
BEGIN
  chrr := ' ';  
  i:=0;
  temp:=valu;
  REPEAT
    temp:=temp DIV base;
    i:=i+1
  UNTIL temp=0;
  FOR j:=1 DOWNTO 1 DO
    BEGIN
    temp:=valu MOD base;
    valu:=valu DIV base;
    chrr(.j.) := CHR(temp+zero);
  END
END;

PROCEDURE pack(VAR to_string : string; from_string : string; VAR start : integer);
VAR len_temp_str,pos : integer;
BEGIN
  len_temp_str:=line_length-1;
  WHILE (from_string(.len_temp_str.)=' ') AND (len_temp_str>=1) DO len_temp_str:=len_temp_str-1;
  pos:=0;
  WHILE ((start+pos)<line_length) AND (pos<=len_temp_str) DO
    BEGIN
    to_string(.start+pos.) := from_string(.pos.);
    pos:=pos+1
  END;
  IF (start+pos)<line_length THEN to_string(.start+pos.):=' '; 
  start:=start+len_temp_str+2
END;

FUNCTION c(str1,str2,str3,str4,str5 : string) : string;
CONST
  U = '??X?X?X?X?X';
VAR workstring : string;
  start : integer;

BEGIN
  start:=0;
  workstring:='  ';
  IF str1<>U THEN pack(workstring,str1,start);
  IF str2<>U THEN pack(workstring,str2,start);
  IF str3<>U THEN pack(workstring,str3,start);
  IF str4<>U THEN pack(workstring,str4,start);
  IF str5<>U THEN pack(workstring,str5,start);
  c:=workstring
END;

PROCEDURE read_buffer;
{ This procedure reads the next line of input from the terminal }
VAR
  ch : char;
  o_k : boolean;
BEGIN
  o_k := false;
  input_buffer := blank;
  WHILE NOT o_k DO
    BEGIN
      IF eoln THEN read(ch);
      input_buffer := ' ';
      input_index := 1;
      error_index := 1;
      buffer_length := 0;
      WHILE buffer_length < line_length AND NOT eoln AND NOT eof DO
        BEGIN
          incr(buffer_length);
          read(input_buffer(.buffer_length.))
        END;
      IF input_buffer = blank THEN
        buffer_length := 0;
      IF buffer_length = line_length AND NOT eoln THEN
        BEGIN
          o_k := false;
          readln;
          writeln('&Input line exceeds allowable length : ', line_length);
          writeln('0', input_buffer)
        END;
      ELSE
        o_k := true
      END;
      IF debugging*.10.) THEN
        BEGIN
          writeln('*** from read buffer ***');
          writeln('  input_index  ', input_index);
        END;
    END;
END;
writeln('  error_index : ', error_index);
writeln('  buffer_length : ', buffer_length);
writeln('  ok ? ', o_k)
END

FUNCTION question(query : string) : boolean;
{ Ask a question and get a yes or no answer. }
VAR
  o_k : boolean;
BEGIN
  o_k := false;
  WHILE NOT o_k DO
    BEGIN
      writeln;
writeln(query);
      read_buffer;
      IF buffer_length = 0 OR
        input_buffer = 'N' OR
        input_buffer = 'n' OR
        input_buffer = 'NO' OR
        input_buffer = 'no' THEN
        BEGIN
          o_k := true;
          question := false
        END
      ELSE
        IF input_buffer = 'Y' OR
          input_buffer = 'y' OR
          input_buffer = 'YE' OR
          input_buffer = 'ye' OR
          input_buffer = 'YES' OR
          input_buffer = 'yes' THEN
          BEGIN
            o_k := true;
            question := true
          END
        ELSE
          complain(bad_data)
        END;
      writeln
    END;
PROCEDURE complain(message : complaint);
{ This procedure issues error messages }
BEGIN
  writeln;
  write('  ****  ');
  WRITE('  INVALID  INPUT  DATA');
599  limit_error  : WRITE('INPUT LIMITS EXCEEDED');
600  io_error    : WRITE('I/O error in processing file');
601  cancelled   : WRITE('Command cancelled');
602  bad_comnd   : WRITE('Invalid COMMAND');
603  END;
604
605  writeln(****);
606  writeln
607  END;
608
609  PROCEDURE read_text(query : string; VAR return_text : txt_str);
610
611  VAR
612    j, i : integer;
613
614  BEGIN
615    writeln(query);
616    read_buffer;
617    IF input_buffer <> '' THEN
618      BEGIN
619        j := str_length;
620        IF input_buffer(.j.) = '' THEN REPEAT
621          j := j - 1
622        UNTIL input_buffer(.j.) <> '';
623        FOR i := 1 TO j DO return_text(.i.) := input_buffer(.i.);
624      END
625  END;
626
627  PROCEDURE link_to_list(i : integer);
628  { This procedure links a new polygon element to the list of polygon vertices. }
629
630  VAR
631    x : @poly elt;
632
633  BEGIN
634    WITH polygon(.i.) DO
635      BEGIN
636        IF pool_ptr <> nil
637          THEN
638            BEGIN
639              x := pool_ptr;
640              pool_ptr := x@.poly_ptr
641            END
642        ELSE new(x);
643        IF polygon(.i.).base_ptr = nil
644          THEN
645            BEGIN
646              x@.poly_ptr := nil;
647              base_ptr := x;
648              gen pts(.i.) := x;
649            END
650        ELSE
651          BEGIN
652            x@.poly_ptr := nil;
653            base_ptr := x;
654            gen pts(.i.) := x;
655          END
656        END
PROCEDURE return_to_pool(i : integer);
{ This procedure returns a linked list to the general pool of storage. }

VAR
  x : @poly_elt;
  j : integer;
BEGIN
  WITH polygon(.i.) DO
    BEGIN
      WHILE base_ptr <> nil DO
        BEGIN
          x := base_ptr;
          base_ptr := base_ptr@.poly_ptr;
          x@.poly_ptr := pool_ptr;
          pool_ptr := x
        END;
      polygon(.i.).descript_posn.y := 0;
      polygon(.i.).descript_posn.x := 0;
      polygon(.i.).size := 0;
      FOR j := 1 TO str_length DO polygon(.i.).poly_descript(.j.) := nil;
    END;
  END;
END;

PROCEDURE insert_point(i,x,y : integer);
BEGIN
  link_to_list(i);
  gen_pts(.i.)@.poly_x := x;
  gen_pts(.i.)@.poly_y := y;
END;

PROCEDURE close_polygon(i : integer);
{ This procedure closes a minimal polygon by passing its vertices in pairs to a DDA. }

VAR
  x1,y1,
  x2,y2 : integer;
BEGIN
  WITH polygon(.i.) DO
    BEGIN
      gen_ptr := base_ptr;
      IF gen_ptr@.poly_ptr <> nil THEN
        REPEAT
          x1 := gen_ptr@.poly_x;
          y1 := gen_ptr@.poly_y;
          gen_ptr := gen_ptr@.poly_ptr;
        UNTIL gen_ptr@.poly_ptr = nil;
      END;
    END;
END;
procedure dda (x1, y1, x2, y2 : integer);

{ This procedure implements the simple 'digital differential analyser' as described by Newman and Sproull in their book 'Principles of Interactive Graphics'. This algorithm achieves the same accuracy as the 'symmetrical' algorithm described in the same book, but is more suited to software implementation, each iteration requiring only two additions. }

var
length, l : integer;
x, y, x_incr, y_incr : real;

begin
if (x1 <= x2) or (y1 <= y2) then
begin
length := max(abs(x2 - x1), abs(y2 - y1));
x_incr := (x2 - x1) / length;
y_incr := (y2 - y1) / length;
x := x1 + 0.5;
y := y1 + 0.5;
for l := 1 to length do
begin
set_bit (work_matrix, trunc(y), trunc(x), back);
x := x + x_incr;
y := y + y_incr;
end;
end;
end;
BEGIN
  FOR k := 1 TO trailers DO
    BEGIN
      mat_init(matrix, back);
      FOR i := st TO y_resolution DO
        BEGIN
          FOR j := st TO x_resolution DO
            BEGIN
              IF test(mat, i-1, j, back) THEN
                BEGIN
                  count := 0;
                  IF test(mat, i-1, j-1, back) THEN count := count+1;
                  IF test(mat, i+1, j-1, back) THEN count := count+1;
                  IF test(mat, i+1, j+1, back) THEN count := count+1;
                  IF test(mat, i-1, j+1, back) AND (count<2) THEN count := count+1;
                  IF test(mat, i+1, j+1, back) AND (count<2) THEN count := count+1;
                  IF test(mat, i+1, j-1, back) AND (count<2) THEN count := count+1;
                  IF test(mat, i-1, j-1, back) AND (count<2) THEN count := count+1;
                  IF count>1 THEN set_bit(matrix, i, j, back)
                END;
            END;
        END;
      copy_mat(mat, matrix, back)
    END;
END;

FUNCTION end_check(VAR mat : bool_matrix; row, k, temp, j, plane : integer) : boolean;
{ This function is set to true if the current pixel-string is part of a polygon edge, i.e., it continues in the direction being checked. }
BEGIN
  end_check := false;
  IF temp = j { only one point in a row } THEN
    IF (test(mat, row+k, j-1, plane) OR test(mat, row+k, j, plane) OR
      test(mat, row+k, j+1, plane))
      THEN end_check := true
  ELSE
    IF (test(mat, row+k, temp-1, plane) OR test(mat, row+k, temp, plane) OR
      test(mat, row+k, temp+1, plane) OR
      test(mat, row+k, j-1, plane) OR test(mat, row+k, j, plane) OR
      test(mat, row+k, j+1, plane))
      THEN end_check := true
  ELSE
      end_check := false;
END;
Then end_check := true

FUNCTION top_bot_check(VAR mat : bool_matrix; temp,i,row,j,plane : integer) : boolean;

{ This function is set to true if the pixels being checked do not belong
to an upper or lower arc - depending on which direction is being checked.
If the current string of pixels does belong to an arc, then the function is
set to false. }

VAR
  found,
  break : boolean;
  k,
  count : integer;
BEGIN
  found := false;
  break := false;
  count := 0;
  k := temp - 2;
  REPEAT
    k := k+1;
    IF test(mat,i+row,k,plane) AND NOT found
    THEN
      BEGIN
        found := true;
        count := count + 1
      END
    ELSE
      IF NOT test(mat,i+row,k,plane) AND found
      THEN
        BEGIN
          break := true;
          found := false
        END
      UNTIL (break AND (count = 2)) OR (k = j+1);
  END;
END;

PROCEDURE fill_polygon(min_x,max_x,min_y,max_y : integer; mat : bool_matrix);

CONST
  above = -1;
  below = +1;
VAR
  beam : boolean;
  i,j : integer;
  count : integer;
  s temp : integer;
  change : boolean;
BEGIN
  remove_trailer(mat);
mat_init(mat,front);

FOR i := min_y TO max_y DO
BEGIN
  beam := false;
  count := 0;
  IF debugging(.5.) THEN writeln;
  j := min_x-1;
  WHILE j < max_x DO
BEGIN
    j := j + 1;
    IF test(mat,i,j,back) THEN
      BEGIN
        temp := j;
        WHILE test(mat,i,j+1,back) DO
BEGIN
          set_pixel(i,j);
          j := j + 1
        END;
        IF end_check(mat,1,above,temp,j,back) AND end_check(mat,i,beiow,temp,j,back)
THEN.
BEGIN
  change := top_bot_check(mat,temp,1,above,j,back);
  IF change THEN change := top_bot_check(mat,temp,1,below,j,back);
  IF change THEN beam := NOT beam
END
END;
IF beam THEN  set_pixel(1,j)
ELSE  IF  test(mat,i,j,back)  THEN  set_pixel(i,j)
END
END;
FUNCTION test(VAR mat : bool_matrix; i,j,k : integer) : boolean;
{ This function tests whether the i,j,k th bit of the bit-matrix is on }
BEGIN
VAR
  mask,
  word,
  bit : integer;
  temp : boolean;
BEGIN
  j := j+C+1;
  word := (j+word_size_1) div word_size;
  bit := (j+word_size_1) mod word_size;
  mask := mask_bit(bit);
  IF band(mat(.i,word,k.),mask)<>0
THEN temp := true
ELSE temp := false;

test := temp;

IF debugging(.7.) THEN
  BEGIN
    writeln;
    writeln('**** Debugging test ****');
    writeln('i,j,k = ',i:4,j:4,k:4);
    writeln('word = ',word);
    writeln('bit = ',bit);
    writeln('mask = ',mask);
    writeln
  END
END;

PROCEDURE set_bit(VAR mat : bool_matrix; i,j,k : integer);
{ This procedure sets a bit in the bit-matrix }
VAR
  mask,
  word,
  bit : integer;
BEGIN
  j := j+C+1;
  word := (j+word_size_1) div word_size;
  bit := (j+word_size_1) mod word_size;
  mask := mask_bit(bit);
  mat(i,word,k) := bor(mat(i,word,k),mask);
  IF debugging(.7.) THEN
    BEGIN
      writeln;
      writeln('**** Debugging set bit ****');
      writeln('mat(1,j,k) = ',mat(1,word,k));
    END
  END;
END;

PROCEDURE unset_bit(VAR mat : bool_matrix; i,j,k : integer);
{ This procedure sets a bit in the bit-matrix }
VAR
  mask,
  word,
  bit : integer;
BEGIN
  j := j+C+1;
  word := (j+word_size_1) div word_size;
  bit := (j+word_size_1) mod word_size;
  mask := mask_comp(bit);
1010 \text{mat}(.i,\text{word},k.) := \text{band(mat}(.i,\text{word},k.),\text{mask});
1020$
$1021 \text{IF debugging}(.7.) \text{ THEN}$
1022 BEGIN
1023 writeln;
1024 writeln('**** Debugging set bit ****');
1025 writeln('mat(i,j,k) = ',mat(.1,\text{word},k.));
1026 writeln
1027 END$
1028 END;
1029$
$1030 \text{FUNCTION mask_bit(bit : integer) : integer;}
1031 \{ \text{This function sets the appropriate mask for any bit setting} (0 - 31) \}$
1032
case
1033 BEGIN
1034 CASE bit OF
1035 0 : mask_bit := 2147483648; \{ 2 31 \}
1036 1 : mask_bit := 1073741824; \{ 2 30 \}
1037 2 : mask_bit := 536870912; \{ 2 29 \}
1038 3 : mask_bit := 268435456; \{ 2 28 \}
1039 4 : mask_bit := 134217728; \{ 2 27 \}
1040 5 : mask_bit := 67108864; \{ 2 26 \}
1041 6 : mask_bit := 33554432; \{ 2 25 \}
1042 7 : mask_bit := 16777216; \{ 2 24 \}
1043 8 : mask_bit := 8388608; \{ 2 23 \}
1044 9 : mask_bit := 4194304; \{ 2 22 \}
1045 10 : mask_bit := 2097152; \{ 2 21 \}
1046 11 : mask_bit := 1048576; \{ 2 20 \}
1047 12 : mask_bit := 524288; \{ 2 19 \}
1048 13 : mask_bit := 262144; \{ 2 18 \}
1049 14 : mask_bit := 131072; \{ 2 17 \}
1050 15 : mask_bit := 65536; \{ 2 16 \}
1051 16 : mask_bit := 32768; \{ 2 15 \}
1052 17 : mask_bit := 16384; \{ 2 14 \}
1053 18 : mask_bit := 8192; \{ 2 13 \}
1054 19 : mask_bit := 4096; \{ 2 12 \}
1055 20 : mask_bit := 2048; \{ 2 11 \}
1056 21 : mask_bit := 1024; \{ 2 10 \}
1057 22 : mask_bit := 512; \{ 2 9 \}
1058 23 : mask_bit := 256; \{ 2 8 \}
1059 24 : mask_bit := 128; \{ 2 7 \}
1060 25 : mask_bit := 64; \{ 2 6 \}
1061 26 : mask_bit := 32; \{ 2 5 \}
1062 27 : mask_bit := 16; \{ 2 4 \}
1063 28 : mask_bit := 8; \{ 2 3 \}
1064 29 : mask_bit := 4; \{ 2 2 \}
1065 30 : mask_bit := 2; \{ 2 1 \}
1066 31 : mask_bit := 1; \{ 2 0 \}
1067\end{cases}$
1068 END \{case\}$
1069 END;
1070$
$1071 \text{FUNCTION mask_comp(bit : integer) : integer;}
1072 \{ \text{This function sets the appropriate mask for any bit setting} (0 - 31) \}$
1073 BEGIN
1074 CASE bit OF
1075 0 : mask_comp := 2147483647; \{ 2 31 \}
1076 END
1077$
$1078 FUNCTION mask_bit(bit : integer) : integer;
1079 \{ \text{This function sets the appropriate mask for any bit setting} (0 - 31) \}$
1080 BEGIN
1081 CASE bit OF
1082 0 : mask_bit := 2147483648; \{ 2 31 \}
1083 1 : mask_bit := 1073741824; \{ 2 30 \}
1084 2 : mask_bit := 536870912; \{ 2 29 \}
1085 3 : mask_bit := 268435456; \{ 2 28 \}
1086 4 : mask_bit := 134217728; \{ 2 27 \}
1087 5 : mask_bit := 67108864; \{ 2 26 \}
1088 6 : mask_bit := 33554432; \{ 2 25 \}
1089 7 : mask_bit := 16777216; \{ 2 24 \}
1090 8 : mask_bit := 8388608; \{ 2 23 \}
1091 9 : mask_bit := 4194304; \{ 2 22 \}
1092 10 : mask_bit := 2097152; \{ 2 21 \}
1093 11 : mask_bit := 1048576; \{ 2 20 \}
1094 12 : mask_bit := 524288; \{ 2 19 \}
1095 13 : mask_bit := 262144; \{ 2 18 \}
1096 14 : mask_bit := 131072; \{ 2 17 \}
1097 15 : mask_bit := 65536; \{ 2 16 \}
1098 16 : mask_bit := 32768; \{ 2 15 \}
1099 17 : mask_bit := 16384; \{ 2 14 \}
1100 18 : mask_bit := 8192; \{ 2 13 \}
1101 19 : mask_bit := 4096; \{ 2 12 \}
1102 20 : mask_bit := 2048; \{ 2 11 \}
1103 21 : mask_bit := 1024; \{ 2 10 \}
1104 22 : mask_bit := 512; \{ 2 9 \}
1105 23 : mask_bit := 256; \{ 2 8 \}
1106 24 : mask_bit := 128; \{ 2 7 \}
1107 25 : mask_bit := 64; \{ 2 6 \}
1108 26 : mask_bit := 32; \{ 2 5 \}
1109 27 : mask_bit := 16; \{ 2 4 \}
1110 28 : mask_bit := 8; \{ 2 3 \}
1111 29 : mask_bit := 4; \{ 2 2 \}
1112 30 : mask_bit := 2; \{ 2 1 \}
1113 31 : mask_bit := 1; \{ 2 0 \}
1114 END \{case\}$
1115 END;
1116$
PROCEDURE primary_initialize;

VAR
  i,j : integer;

BEGIN
  already_active := false;
  more_to_come := true;
  pt_set_no := 0;
  digits := ('0'..'9');
  pool_ptr := nil;
  graph := 1;
  unit_no := unitno;
  TRANS('EBBCDIC TO ASCII NONE:',' ',0);
  cmt_initialize(unit_no,ret_code);
  writeln;
  CASE ret_code OF
    0 : writeln('Connection with COMTAL O.K.');
    2 : writeln('Error in connection with COMTAL');
    3 : writeln('COMTAL not operational');
    <0 : writeln('An unspecified error has occurred in linking MTS to THE COMTAL');
  END; {case}
  writeln;
END;
secondary_initialize;
tertiary_initialize;
mat_init(save_matrix,back);
mat_init(save_matrix,front);
FOR i := -1 TO max_polys DO
BEGIN
  polygon(.i.).base_ptr := nil;
  polygon(.i.).descript_posn.y := 0;
  polygon(.i.).descript_posn.x := 0;
  FOR j := 1 TO str_length DO polygon(.i.).poly_descript(.j.) := null;
  polygon(.i.).size := 0;
END;

IF question(’&Debug data ? (Y/N)’) THEN debug_data,
  read_int(1,10,’&Max size for smudges = ?’,acc_smudge);

PROCEDURE secondary_initialize;
VAR
  ch : char;
  i,j : integer;
BEGIN
  { initialize border of ’matrix’ }
  FOR j := 1 TO C+1 DO
  BEGIN
    FOR i := st-C-1 TO x_resolution+C+1 DO unset_bit(matrix,st-j,i,front);
    FOR i := st-C-1 TO x_resolution+C+1 DO unset_bit(matrix,y_resolution+j,i,front);
    FOR i := st-C-1 TO y_resolution+C+1 DO unset_bit(matrix,i,st-j,front);
    FOR i := st-C-1 TO y_resolution+C+1 DO unset_bit(matrix,i,x_resolution+j,front);
  END;

  { initialize ’index’ array }
  FOR i := st TO x_resolution DO
  BEGIN
    index(.i.).no := 0;
    index(.i.).y_min := maxint;
    index(.i.).y_max := -maxint;
    FOR j := st TO ptrs_limit DO index(.i.).ptrs(.j.) := nil
  END;

PROCEDURE tertiary_initialize;
BEGIN

mat_init(work_matrix,back);

no_arrows := 0;

FUNCTION first_pt : point_found;

( This function returns the top left-most co-ordinates in the 'matrix' which
represent a 'true' value, i.e. the first point. )

VAR
  y, x : integer;
  found : boolean;
  i : integer;

BEGIN
  y := maxint;

  FOR i := 1 TO x_resolution DO
    IF index(.i.).no > 0
    THEN IF index(.i.).y_min < y
    THEN BEGIN
      y := index(.i.).y_min;
      x := i
    END;

  IF y = maxint THEN found := false
  ELSE found := true;

  first_pt.found := found;
  first_pt.x := x;
  first_pt.y := y;

  IF debugging(.4.)
  THEN BEGIN
    writeln;
    writeln('**** Debugging find first point ****');
    writeln('first_pt.found = ', found);
    writeln('first_pt.x = ', x);
    writeln('first_pt.y = ', y);
  writeln
  END;

END;

FUNCTION sub_convert(x : integer) : integer;

( This function returns the subscript in the 'ptrs' array for the n'th point )

BEGIN
  sub_convert := (x + y_div_16 - 1) div y_div_16;
END;

FUNCTION off_convert(x : integer) : integer;

( This function returns the offset in the 'y_block' for the n'th point. )

BEGIN
  off_convert := (x + y_div_16 - 1) mod y_div_16 + 1;
FUNCTION find_binary(x,y,new_x : integer; VAR next_x,next_y,displace : integer) : location;

{ This function is called by the procedure 'find_closest' and returns the closest point to the current point as determined by a binary search. }

VAR
  new_y,
  mid_pt,
  i,
  j : integer;
  found : boolean;
  subscript,
  offset : integer;
  temp_dst : integer;

BEGIN
  WITH index(.new_x.) DO
    BEGIN
      i := 1;
      j := no;
      found := false;
      REPEAT
        mid_pt := (i+j) div 2;
        subscript := sub_convert(mid_pt);
        offset := off_convert(mid_pt);
        new_y := ptrs(.subscript.)@(.offset.);
        temp_dst := ((x-next_x)*(x-next_x)+(y-next_y)*(y-next_y));
        IF (next_x=frst_pt.x) AND (next_y=frst_pt.y) THEN temp_dst := temp_dst+1;
        IF ((x-new_x)*(x-new_x)+(y-new_y)*(y-new_y)) < temp_dst THEN
          BEGIN
            found := true;
            next_x := new_x;
            next_y := new_y;
            find_binary.ind := subscript;
            find_binary.off := offset;
            find_binary.found := found;
            IF max(abs(x-new_x),abs(y-new_y)) < displace THEN displace := max(abs(x-new_x),abs(y-new_y));
          END;
        END;
        IF y > ptrs(.subscript.)@(.offset.) THEN i := mid_pt+1
        ELSE j := mid_pt-1
      UNTIL ((ptrs(.subscript.)@(.offset.) = y) OR (i > j));
      IF debugging*.4.) O
    BEGIN
      writeln;
      writeln('**** Debugging find binary ****');
writeln('find_binary.found = ',found);
writeln('find_binary.ind = ',subscript);
writeln('find_binary.off = ',offset);
writeln('next_x  = ',next_x);
writeln('next_y  = ',next_y);
END;

PROCEDURE write_index(x,y : integer);

{ This procedure writes the y ordinate of the point to the y_block pointed to
by the ptrs array. It also increments the count of the number of points
in the relevant column of the matrix & checks against min/max for that col }

VAR
subscript,
offset : integer;
been_new : boolean;
BEGIN
WITH index(.x.) DO
BEGIN
been_new := false;
no := no+1;
subscript := sub_convert(no);
offset := off_convert(no);
IF y < y_min THEN y_min := y;
IF y > y_max THEN y_max := y;
IF ptrs(.subscript.) = nil
THEN
BEGIN
new(ptrs(.subscript.));
been_new := true
END;
ptrs(.subscript.)@(.offset.) := y;
END;
IF debugging*.10.)
THEN
BEGIN
writeln;
writeln('**** Debugging write index ****');
writeln('subscript = ',subscript);
writeln('offset = ',offset);
writeln('x = ',x);
writeln('y = ',y);
writeln('been_new = ',been_new);
writeln
END;
IF debugging(.10.)
THEN
BEGIN
writeln;
writeln('**** Debugging write index ****');
writeln('subscript = ',subscript);
writeln('offset = ',offset);
writeln('x = ',x);
writeln('y = ',y);
writeln('been_new = ',been_new);
writeln
END;

PROCEDURE delete_index(x,y,subscript,offset : integer);

{ This procedure deletes a point's y ordinate from the y_block pointed to by
the ptrs array. It decrements the col's point count and may modify that
cols max/min. It also removes the point from the 'matrix

VAR

i,j,
sub1,sub2,
off1,off2 : integer;

BEGIN

WITH index(.x.) DO

BEGIN

no := no-1;
j := ((subscript-1)*y_div_16)+offset;
unset_bit(matrix,y,x,front);

insert_point(0,x,y); { establish a record of points to be deleted in save_matrix }

FOR i := j TO no DO

BEGIN

sub1 := sub_convert(i);
sub2 := sub_convert(i+1);
off1 := off_convert(i);
off2 := off_convert(i+1);

ptrs(.sub1.)(.off1.) := ptrs(.sub2.)(.off2.)

END;

sub1 := sub_convert(no+1);
off1 := off_convert(no+1);
ptrs(.sub1.)(.off1.) := 0;
IF (offset-1 = 0) AND (j = (no+1)) THEN ptrs(.subscript.) := nil;

IF no > 0 THEN

BEGIN

IF (subscript = st) AND (offset = st)
THEN

y_min := ptrs(.1.)(.1.)
ELSE

IF j = (no+1)
THEN

BEGIN

sub1 := sub_convert(no);
off1 := off_convert(no);
y_max := ptrs(.sub1.)(.off1.)
END

END

END;

IF debugging(.4.) THEN

BEGIN

writeln;
writeln('**** Debugging delete index ****');
writeln('subscript = ',.subscript);
writeln('offset = ',.offset);
writeln('x = ',.x);
writeln('y = ',.y);
writeln

END
PROCEDURE estab_index;

{ This procedure controls the scans the matrix and causes the invocation of
the write_index procedure in order to place the points in the 'matrix' into
the 'index'
}

VAR
  i, j : integer;
  point_count : integer;
  t_min_x, t_max_x : integer; { temp vals for min & max if 'already_active' }
BEGIN
  point_count := 0;
FOR i := st TO y_resolution DO
  FOR j := st TO x_resolution DO
    IF test(matrix,i,j,front)
      THEN
        BEGIN
          write_index(j,i);
          point_count := point_count+1
        END;
    IF pt_set_no <= 1 AND NOT already_active THEN
      BEGIN
        min_x := st;
        IF index(.min_x.).no=0
          THEN WHILE index(.min_x+1.).no=0 DO min_x := min_x+1;
        max_x := x_resolution;
        IF index(.min_x.).no=0
          THEN WHILE index(.max_x-1.).no=0 DO max_x := max_x-1;
      END;
    IF already_active THEN
      BEGIN
        t_min_x := st;
        IF index(.t_min_x.).no=0
          THEN WHILE index(.t_min_x+1.).no=0 DO t_min_x := t_min_x+1;
        t_max_x := x_resolution;
        IF index(.t_min_x.).no=0
          THEN WHILE index(.t_max_x-1.).no=0 DO t_max_x := t_max_x-1;
        min_x := min(min_x,t_min_x);
        max_x := max(max_x,t_max_x);
      END;
    IF debugglng(.4.)
      THEN
        BEGIN
          writeln; {**** Debugging establish index ****}
          writeln('point_count = ',point_count);
          writeln('min_x = ',min_x);
          writeln('max_x = ',max_x);
        END
END;
FUNCTION same_col(x,y,
    ind.off : integer;
VAR
    subscript,
    offset,
    displace : integer) : point_found;
{ This function returns the closest point in the same column to the current
  point. A closest point may or may not be found. }
VAR
    temp_posn,
    posn,
    y1,y2,
    col_x,col_y,
    sub1,sub2,
    off1,off2 : integer;
    found : boolean;
BEGIN
    y1 := maxint;
    y2 := maxint;
    posn := (y_div_16 * (ind-1)) + off;
    IF (posn-1) > 0
    THEN
        BEGIN
            temp_posn := posn-1;
            sub1 := sub_convert(temp_posn);
            off1 := off_convert(temp_posn);
            y1 := index(.x.).ptrs(.sub1.)@(.off1.)
        END;
    IF  (posn+1)  <= index(.x.).no
    THEN
        BEGIN
            temp_posn := posn+1;
            sub2 := sub_convert(temp_posn);
            off2 := off_convert(temp_posn);
            y2 := index(.x.).ptrs(.sub2.)@(.off2.)
        END;
    IF (y1 = maxint)  AND (y2 = maxint)
    THEN
        BEGIN
            found := false;
            col_x := max(x_res_plus_C,y_res_plus_C);
            col_y := max(x_res_plus_C,y_res_plus_C);
            subscript := max(x_res_plus_C,y_res_plus_C);
            offset := max(x_res_plus_C,y_res_plus_C);
        END
    ELSE
        IF abs(y1-y) <= abs(y2-y)
        THEN
            ...
found := true;
col_x := x;
col_y := y;
subscript := sub1;
offset := off1
END
ELSE
BEGIN
found := true;
col_x := x;
col_y := y2;
subscript := sub2;
offset := off2
END

same_col.found := found;
same_col.x := col_x;
same_col.y := col_y;

IF found
THEN displace := abs(col_y - y)
ELSE displace := x_resolution;
IF debugging(.4.)
THEN
BEGIN
writeln;
writeln('**** Debugging same column ****');
writeln('found ',found);
writeln('col_x  = ', col_x);
writeln('col_y  = ', col_y);
writeln('subscript = ', subscript);
writeln('offset = ', offset);
writeln('dlspl = ', displace);
writeln END

PROCEDURE find_closest(x,y,
curr_col  :  integer;
VAR
  displace,
  subscript,
  offset  :  integer;
VAR
  next_x,
  next_y  :  integer;

{ This procedure finds the closest point (in 'other' cols to the current
point. A previous 'closest' point may or may not exist. }

CONST
  linear_limit = 16;

VAR
  bin
  posn,
  new_sub,new_off,
WITH index(.curr_col.) DO
BEGIN
IF no > 0 THEN
IF no > linear_limit THEN BEGIN
bin := find_binary(x,y,curr_col,next_x,next_y,displace);
END
ELSE BEGIN
IF abs(y_max - y) < abs(y_min - y) THEN BEGIN
start := no;
fin := 1;
incr := -1
END ELSE BEGIN
start := 1;
fin := no;
incr := 1
END;
rest_too_far := false;
posn := start-incr;
new_x := curr_col;
new_y2 := maxint;
REPEAT
posn := posn+incr;
new_sub:= sub_convert(posn);
new_off:= off_convert(posn);
new_y1 := ptrs(.new_sub.)@(.new_off.);
temp_dst := (((x-next_x)*(x-next_x)+(y-next_y)*(y-next_y))
IF (next_x=frst_pt.x) AND (next_y=frst_pt.y) THEN temp_dst := temp_dst+1;
IF ((x-new_x)*(x-new_x)+(y-new_y1)*(y-new_y1)) < temp_dst THEN BEGIN
next_x := new_x;
next_y := new_y1;
subscript := new_sub;
END
IF max(abs(x-new_x),abs(y-new_y)) < displace
THEN displace := max(abs(x-new_x),abs(y-new_y));
END;

IF abs(y-new_y1) > abs(y-new_y2)
THEN rest_too_far := true
ELSE new_y2 := new_y1;
UNTIL rest_too_far OR (posn = fin);
END

FUNCTION other_cols(x,y,
ind,off : integer;
VAR
subscript,
offset,
displace,
next_x,next_y : integer) : point_found;
BEGIN
x_disp := 1;
WHILE x_disp <= displace DO
BEGIN
FOR i := 1 TO 2 DO
BEGIN
x_disp := -x_disp;
curr_col := curr_col+x Disp;
END
END
x Disp := 1;
IF (curr_col>=min_x) AND (curr_col<=max_x)
    THEN find_closest(x,y,curr_col,displace,
                     subscript,offset,next_x,next_y);
    IF (i=2) THEN x_disp := x_disp+1
END
END;

PROCEDURE find_area;

{ This procedure controls the search for points belonging to one polygon }

VAR
  curr_pt : location;
  next_pt : location;
  subscript,
  offset : integer;
  i,j : integer;
  this_pt : point_found;
BEGIN
  curr_pt.ind := 1;
  curr_pt.off := 1;
  next_x := max(x_res_plus_C,y_res_plus_C);
  next_y := max(x_res_plus_C,y_res_plus_C);
  max_smudge_x := st;
  max_smudge_y := st;
  min_smudge_x := x_resolution;
  min_smudge_y := y_resolution;
  IF debugging(.2.) THEN mat_init(work_matrix,front);
  insert_point(pt_set_no,curr_x,curr_y);
  REPEAT
    IF (curr_x = frst_pt.x) AND (curr_y = frst_pt.y)
        THEN IF test(matrix,curr_y,curr_x+1,front)
            THEN smudge(curr_x,curr_y-1,curr_x,curr_y):
    this_pt := same_col(curr_x,curr_y,curr_pt.ind,curr_pt.off,
                      next_pt.ind,next_pt.off,displace);
    IF this_pt.found
        THEN
            BEGIN
                next_x := this_pt.x;
                next_y := this_pt.y;
            END
        ELSE
            BEGIN
                next_x := max(x_res_plus_C,y_res_plus_C);
                next_y := max(x_res_plus_C,y_res_plus_C);
            END;
            this_pt := other_cols(curr_x,curr_y,curr_pt.ind,curr_pt.off,
                      next_pt.ind,next_pt.off,displace,next_x,next_y):
    smudge(curr_x,curr_y,next_x,next_y);
1799 IF debugging(.2.) THEN set_bit(work_matrix, next_y, next_x, front);
1800 IF debugging(.2.) THEN writeln(next_x, next_y, next_pt.ind, next_pt.off);
1801 insert_point(pt_set_no, next_x, next_y);
1802 IF next_x > max_smudge_x THEN max_smudge_x := next_x
1803 ELSE IF next_x < min_smudge_x THEN min_smudge_x := next_x;
1804 IF next_y < min_smudge_y THEN min_smudge_y := next_y
1805 ELSE IF next_y > max_smudge_y THEN max_smudge_y := next_y;
1806 IF NOT(curr_x = frst_pt.x) AND (curr_y = frst_pt.y))
1807 THEN
1808 BEGIN
1809 find_linear(curr_x, curr_y, curr_pt.ind, curr_pt.off);
1810 delete_index(curr_x, curr_y, curr_pt.ind, curr_pt.off);
1811 IF test(matrix, curr_y, curr_x, back)
1812 THEN unset_bit(matrix, curr_y, curr_x, back)
1813 END;
1814 find_linear(next_x, next_y, next_pt.ind, next_pt.off);
1815 curr_x := next_x;
1816 curr_y := next_y;
1817 curr_pt.ind := next_pt.ind;
1818 curr_pt.off := next_pt.off
1819 FOR i := min_smudge_y TO max_smudge_y DO
1820 FOR j := min_smudge_x TO max_smudge_x DO
1821 IF test(matrix, i, j, back) THEN
1822 BEGIN
1823 find_linear(j, i, next_pt.ind, next_pt.off);
1824 IF NOT((j = frst_pt.x) AND (i = frst_pt.y)) THEN
1825 delete_index(j, i, next_pt.ind, next_pt.off);
1826 unset_bit(matrix, i, j, back)
1827 END;
1828 close_polygon(pt_set_no);
1829 IF debugging(.2.) THEN
1830 BEGIN
1831 FOR i := st TO y_resolution DO
1832 BEGIN
1833 write('l');
1834 IF (j mod 80) = 0
1835 THEN
1836 writeln;
1837 IF j <> x_resolution THEN write(' ')
1838 END;
1839 writeln;
1840 END;
1841 writeln;
1842 END;
1843 END;
1844 END;
1845 END;
1846 END;
1847 END;
1848 END;
1849 END;
1850 END;
1851 END;
1852 . writeln
1853 END;
1854 END;
1855 . writeln
1856 END;
PROCEDURE smudge(curr_x, curr_y : integer; VAR next_x, next_y : integer);

{ This procedure controls the elimination of smudges to a depth specified
  by the user (acc_smudge). This variable controls the broader accuracy of
  the polygon determination. }

TYPE
dir = (north, south, east, west, southeast, southwest, northeast, northwest);

VAR
direction : dir;

PROCEDURE nsew_check(x_inc, y_inc : integer);

{ This procedure checks the direction between curr and next points and
  establishes the control variables for the procedure 'clear_up'
}

VAR
temp_x, temp_y, temp_ind, temp_off, x_incr, y_incr : integer;

PROCEDURE clear_up(temp_x, temp_y : integer);

{ This procedure actually removes the smudges. The algorithm is recursive
  and deletes all 'internal fully bounded points', where internal means on
  the inside of the polygon, and fully bounded implies complete adjacency
  i.e. no diagonal touches. }

BEGIN { clear_up }
  IF debugging(.4.) THEN BEGIN
    writeln;
    writeln('**** Debugging clear_up ****');
    writeln('temp_x = ',temp_x:0);
    writeln('temp_y = ',temp_y:0);
    writeln
  END;
  IF test(matrix,temp_y+x_incr,temp_x+y_incr,front) THEN clear_up(temp_x+y_incr,temp_y+x_incr);
  IF test(matrix,temp_y-y_incr,temp_x-x_incr,front) THEN clear_up(temp_x-x_incr,temp_y-y_incr);
  IF (temp_x<>next_x) OR (temp_y<>next_y) THEN BEGIN
    IF debugging(.4.) THEN writeln('**** Debugging - from clear_up ****');
    IF (abs(temp_x-next_x) < acc_smudge) AND (abs(temp_y-next_y) < acc_smudge) THEN
      BEGIN
        find_linear(temp_x,temp_y,temp_ind,temp_off);
        IF NOT((temp_x = frst_pt.x) AND (temp_y = frst_pt.y)) THEN
          BEGIN
            delete_index(temp_x,temp_y,temp_ind,temp_off);
          END
      END
  END;
IF test(matrix, temp_y, temp_x, back) THEN unset_bit(matrix, temp_y, temp_x, back)
END
ELSE set_bit(matrix, temp_y, temp_x, back)
END;
BEGIN { nsew check }
CASE direction OF
south, north : BEGIN
  x_incr := x_inc;
  y_incr := y_inc
END;
east, west : BEGIN
  x_incr := -x_inc;
  y_incr := -y_inc
END;
southeast,
southwest,
northeast,
northwest : BEGIN
  x_incr := x_inc;
  y_incr := y_inc
END
END;

IF (direction=north) OR (direction=south) OR (direction=east) OR (direction=west) AND
NOT ((next_x = frst_pt.x) AND (next_y = frst_pt.y))
THEN
BEGIN
  WHILE test(matrix, next_y+x_incr, next_x+y_incr, front) AND
  NOT test(matrix, next_y+y_incr, next_x+x_incr, front) DO
    BEGIN
      temp_x := next_x;
      temp_y := next_y;
      WHILE test(matrix, temp_y, temp_x, front) DO
        BEGIN
          IF debugging(.4.) THEN writeln('**** Debugging - from nsew ****');
          find_linear(temp_x,temp_y,temp_ind,temp_off);
          IF NOT(((temp_x = frst_pt.x) AND (temp_y = frst_pt.y)) THEN
            BEGIN
              delete_index(temp_x,temp_y,temp_ind,temp_off);
              IF test(matrix, temp_y, temp_x, back) THEN unset_bit(matrix, temp_y, temp_x, back)
            END;
            temp_x := temp_x - x_incr;
            temp_y := temp_y - y_incr
            END;
            next_x := next_x + y_incr;
            next_y := next_y + x_incr
          END
        END
      END
    END
  ELSE
    BEGIN
      IF test(matrix, next_y+x_incr, next_x+y_incr, front) THEN
        BEGIN
          temp_x := next_x-x_incr;
          temp_y := next_y-y_incr;
        END
      ELSE
        BEGIN
          temp_x := next_x-x_incr;
          temp_y := next_y-y_incr;
        END
      END
    END
  ELSE
    BEGIN
      IF test(matrix, next_y+x_incr, next_x+y_incr, front) THEN
        BEGIN
          temp_x := next_x-x_incr;
          temp_y := next_y-y_incr;
        END
      ELSE
        BEGIN
          temp_x := next_x-x_incr;
          temp_y := next_y-y_incr;
        END
      END
    END
WHILE test(matrix, temp_y, temp_x, front) DO
BEGIN
IF debugging(.4.) THEN writeln('**** Debugging - from nsew ****');
find_line(temp_x, temp_y, temp_ind, temp_off);
IF NOT((temp_x = frst_pt.x) AND (temp_y = frst_pt.y)) THEN
BEGIN
   delete_index(temp_x, temp_y, temp_ind, temp_off);
   IF test(matrix, temp_y, temp_x, back) THEN unset_bit(matrix, temp_y, temp_x, back)
END;
IF test(matrix, temp_y + y_inc, temp_x + x_inc, front) THEN
   clear_up(temp_x, temp_y);
ELSE IF debugging(.4.) THEN writeln;
BEGIN
   writeln('**** Debugging nsew check ****');
   writeln('x_incr = ', x_incr:0);
   writeln('y_incr = ', y_incr:0);
   writeln
END;
BEGIN { smudge }
IF (curr_x = next_x) AND (curr_y > next_y) THEN direction := north
ELSE IF (curr_x = next_x) AND (curr_y < next_y) THEN direction := south
ELSE IF (curr_x < next_x) AND (curr_y = next_y) THEN direction := east
ELSE IF (curr_x > next_x) AND (curr_y = next_y) THEN direction := west
ELSE IF (curr_x < next_x) AND (curr_y < next_y) THEN direction := southeast
ELSE IF (curr_x > next_x) AND (curr_y < next_y) THEN direction := southwest
ELSE IF (curr_x > next_x) AND (curr_y > next_y) THEN direction := northwest
CASE direction OF
   north : nsew_check(-1,0);
   south : nsew_check(+1,0);
   east : nsew_check(0,-1);
   west : nsew_check(0,+1);
   southeast : nsew_check(+1,0);
   southwest : nsew_check(-1,0);
   northeast : nsew_check(-1,0);
   northwest : nsew_check(+1,0);
END; (* end case *)
END;
FUNCTION check_contract(i,j : integer) : boolean;
{ This function determines whether a point should be left 'lit' or should be
  turned 'off'. For a point to remain lit all its 8 neighbours must be lit. }
VAR
temp : boolean;

BEGIN

IF test(matrix,1,j,front) AND test(matrix,1,j+1,front) AND
test(matrix,i,j+1,front) AND
THEN temp := true
ELSE temp := false;

check_contract := temp;

IF debugging(.8.) AND test(matrix,1,j,front)
THEN
BEGIN
writeln;
writeln('**** Debugging check contract ****');
writeln('i = ',i);
writeln('j = ',j);
writeln
END;

PROCEDURE copy_mat(VAR mat1,mat2 : bool_matrix; plate : integer);
{ This procedure copies matrix 2 to matrix 1 }

VAR i,j,word : integer;

BEGIN
FOR i := (st-C) TO y_res_plus_C DO
FOR j := 1 TO ((x_res_plus_C+C+2) div word_size) + 1 DO
BEGIN
mat1(.i,j,plate.) := mat2(.i,j,plate.)
END;

IF debugging(.3.) THEN
BEGIN
FOR i:= st TO y_resolution DO
BEGIN
write(' ',); FOR j:= st TO x_resolution DO
BEGIN
IF test(mat1,1,j,plate) THEN write('1')
ELSE write('0');
IF (j mod 80) = 0
BEGIN
writeln;
IF j=x_resolution THEN write(' ') END
END;
writeln
END;

PROCEDURE mat_init(VAR mat : bool_matrix; plane : integer);
VAR  
i, j : integer;
BEGIN  
  FOR i := (st-C) TO y_res_plus_C DO 
    FOR j := 1 TO \((x_res_plus_C+C+2) \text{ div word_size} + 1\) DO 
      BEGIN 
        mat(i, j, plane) := 0;
      END;
  IF debugging(.10.) THEN 
    BEGIN 
      writeln;
      writeln('**** Debugging mat init - done ****');
      writeln;
    END 
END;
PROCEDURE contract;
BEGIN  
  VAR 
    i, j, k : integer;

  BEGIN 
    mat_init(work_matrix, front); 
    FOR i := (st-C) TO y_res_plus_C DO 
      FOR j := (st-C) TO x_res_plus_C DO 
        IF test(matrix, i, j, front) THEN 
          IF NOT check_contract(i, j) THEN 
            IF debugging(.10.) THEN writeln('matrix(', i, ',', j, ') = 0') 
              ELSE writeln('matrix(', i, ',', j, ') = 1') 
          END 
        ELSE unset_bit(work_matrix, i, j, front); 
        copy_mat(matrix, work_matrix, front) 
  END;

  FUNCTION check_expand(i, j : integer) : boolean;  
  BEGIN 
    temp := false; 
    IF (i-1 >= 0) THEN temp := temp OR check(num(0)); 
    IF (i+1 < row) THEN temp := temp OR check(num(1)); 
    IF (j-1 >= 0) THEN temp := temp OR check(num(2)); 
    IF (j+1 < col) THEN temp := temp OR check(num(3)); 
    ELSE temp := true; 
    END;
BEGIN
  IF test(matrix, i-1, j-1, front) OR test(matrix, i-1, j, front) OR test(matrix, i-1, j+1, front)
  THEN temp := true
  ELSE
    IF test(matrix, i, j-1, front) OR test(matrix, i, j+1, front) THEN temp := true
    ELSE
      IF test(matrix, i+1, j-1, front) OR test(matrix, i+1, j, front) OR test(matrix, i+1, j+1, front)
      THEN temp := true
      ELSE temp := false;
  END;
  check_expand := temp;
  IF debugging(.10.) AND temp
  THEN
    BEGIN
      writeln;
      writeln(**** Debugging check matrix ****);
      writeln('i = ', i);
      writeln('j = ', j);
      writeln('mat chk = ', temp);
      writeln
    END;
  END;
END;

PROCEDURE expand;
{ This procedure expands the matrix in order to eliminate 'pockets'. }
VAR
  i, j, k : integer;
BEGIN
  mat_init(work_matrix, front);
  FOR i := (st-C) TO y_res_plus_C DO
    FOR j := (st-C) TO x_res_plus_C DO
      IF NOT test(matrix, i, j, front)
      THEN
        BEGIN
          IF check_expand(i, j)
          THEN
            BEGIN
              set_bit(work_matrix, i, j, front);
              BEGIN
                IF debugging(.10.) THEN writeln('matrix(', i:0, ',', j:0, ') = 1')
              END;
            END
          ELSE set_bit(work_matrix, 1, j, front);
        END
      END;
  END;
  copy_mat(matrix, work_matrix, front);
END;

PROCEDURE find_linear(x, y : integer; VAR ind, off : integer);
{ This procedure performs a linear search of the 'index' array in order to locate the location in that array of the specified point. }
VAR
  i, subscript,
  offset : integer;
  found : boolean;
BEGIN
valu := -maxint;
count := 0;
found := false;

WITH index(.x.) DO
  IF no <> 0
  THEN
    REPEAT
      count := count+1;
      subscript := sub_convert(count);
      offset := off_convert(count);
      valu := ptrs(.subscript.@@.offset.);

      IF valu = y THEN found := true
      UNTIL (valu>y) OR found OR (count=no);

    IF found
    THEN
      BEGIN
        ind := subscript;
        off := offset
      END
    ELSE
      BEGIN
        ind := maxint;
        off := maxint
      END;

    IF debugging*.4.)
    THEN
      BEGIN
        writeln;
        writeln('**** Debugging find linear ****');
        writeln('ind = ',subscript:0);
        writeln('off = ',offset:0);
        writeln('x = ',x:0);
        writeln('y = ',y:0);
        writeln;
      END
    END;

FUNCTION shiftr(i,j : integer) : integer;
{ This function shifts the bits to the right. }
VAR
  k : integer;
BEGIN
  FOR k := 1 TO j DO i := i div 2;
  shiftr := i
END;

PROCEDURE pre_process;
VAR
i : integer;
BEGIN
FOR i := 1 TO C DO expand;
FOR i := 1 TO C DO contract;
END;

PROCEDURE read_matrix;

VAR
i,j,k : integer;
word,
bit,
work1,
work2 : integer;
BEGIN
mat_init(matrix,front);
mat_init(matrix,back);
FOR i := st TO y_resolution DO
BEGIN
unit_no := unitno;
read_comtal(unit_no,buffer,length,ret_code);
IF debugging(.10.) THEN
BEGIN
writeln('**** Debugging read matrix ****');
writeln('length = ',length);
writeln('ret_code = ',ret_code);
FOR j := 1 TO 8 DO writeln(' ',j:0,' ',buffer(.j));
j := 80
END;
END;
FOR k := st TO x_resolution DO
BEGIN
word := (k + word_size_1) div word_size;
bit := (k + word_size_1) mod word_size + 1;
work1 := 1;
work1 := shift(work1,word_size - bit);
work2 := band(work1,buffer(.word));
work2 := shiftr(work2,word_size - bit);
IF abs(work2) = 1
THEN
BEGIN
set_bit(matrix,i,k,front);
IF debugging(.6.) THEN write('1');
END
ELSE
BEGIN
END
END;
IF debugging(6.) THEN write('0')
END;

IF debugging(6.) THEN
  IF (k mod j) = 0 THEN
    BEGIN
      writeln;
      write('  ')
    END
  END;
ENDIF debugging(6.) THEN writeln
END;
writeln
END;

FUNCTION graphic_plane(i_o,graphic_pln : integer) : integer;

{ This function calculates the control variable for the call to the read_comtal routine. }

VAR
  work1,work2 : integer;
  temp : integer;
BEGIN
  work1 := 0; {set location to transfer graphic to top of display}
  work2 := 1;
  temp := 9;
  work2 := shift(work2,temp); {set bit 9 to 1}
  work1 := work1 + work2;
  work2 := graphic_pln;
  work2 := work2 - 1;
  temp := 11;
  work2 := shift(work2,temp); {set bits 11&12 to graphic plane}
  work1 := work1 + work2;
  work2 := i_o;
  temp := 15;
  work2 := shift(work2,temp);
  work1 := bor(work1,work2);
  IF debugging(10.) THEN
    BEGIN
      writeln;writeln('**** Debugging graphic plane ****');
      writeln('graphic plane = ',work1:0);
      writeln('graph = ',graph:0);
      writeln;
    END;
  graphic_plane := work1
END;
FUNCTION init_comtal(i_o,graph 1c_pln : integer) : boolean;

{ This function initializes the COMTAL for use by the program. }
BEGIN
graphics := graphic_plane(i_o, graphic_pln);
control_comtal(unit_no, graph, graphics, ret_code);
writeln;
CASE ret_code OF
  0 : writeln('COMTAL connection OK');
  2 : writeln('COMTAL connection error');
  3 : writeln('Invalid command value');
  4 : writeln('Invalid control code value');
  5 : writeln('COMTAL not operational');
  <>: writeln('Unspecified connection error')
END;
writeln;
IF ret_code = 0 THEN init_comtal := true ELSE init_comtal := false END;

PROCEDURE disp_poly(VAR mat : bool_matrix; plane : integer);
VAR
  i, j, k,
  word,
  bit,
  mask : integer;
BEGIN
  unit_no := unitno;
  length := leng_buff;
  FOR i := st to y_resolution DO
    FOR j := 1 TO buff_size DO buffer(.j.) := 0;
    FOR j := st TO x_resolution DO
      IF test(mat, i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;

FUNCTION evens_check(VAR work : bool_matrix; y_min, y_max, x_min, x_max, plane : integer) : boolean;
{ This function is set to true if the number of beam changes associated with
  any single scan line is even. It is set to false otherwise. It also controls
  the deletion of a single unambiguous tail. }
CONST
  above = -1;   below = +1;
BEGIN
  FOR i := y_min TO y_max DO
    FOR j := x_min TO x_max DO
      IF bit(i, j, plane) THEN
        BEGIN
          k := j+C+1;
          word := (j + word_size_1) div word_size;
          bit := (k + word_size_1) mod word_size;
          mask := mask_bit(bit);
          mask := rotate(mask, C+1);
          buffer(.word.) := bor(buffer(.word.), mask)
        END;
      END;
      write_comtal(unit_no, buffer, length, ret_code)
    END;
  END;
END;
VAR
  i, j : integer;  { counters }
  count, posn : integer;  { no of on/off changes }
  change : boolean;  { start posn of last change }
  marked : boolean;  { change on/off of beam }
  temp : integer;
BEGIN
  mat_init(work,(3 - plane));  { initialize for arrows }
  marked := false;
  evens_check := true;
  FOR i := y_min TO y_max DO
    BEGIN
      count := 0;
      j := x_min - 1;
      WHILE j < x_max DO
        BEGIN
          j := j + 1;
          IF test(work,1,j,plane) THEN
            BEGIN
              temp := j;
              WHILE test(work,1,j + 1,plane) DO j := j + 1;
              change := top_bot_check(work,temp,i,above,j,plane);
              IF change THEN change := top_bot_check(work,temp,i,below,j,plane);
              IF change THEN
                BEGIN
                  count := count + 1;
                  posn := temp
                END
            END
        END;
    END;
  IF count = 1 THEN
    BEGIN
      count := count - 1;
      del_bits(work,i,posn,plane)
    END;
  IF odd(count) AND NOT marked THEN drw_arrows(work,marked,i,x_min,x_max,(3 - plane));
  IF marked THEN evens_check := false;
  IF NOT odd(count) THEN marked := false;  { reset 'marked' once arrow is drawn }
  IF debugging(.2.) THEN
    BEGIN
      IF odd(count) THEN
        BEGIN
          writeln; writeln('**** Debugging evens check ****');
          writeln('row = ',i:0);
          writeln('cnt = ',count:0);
        writeln
      END;
    END;
END;
PROCEDURE blank_out(VAR mat : bool_matrix; y1,x1,y2,x2,plane : integer);
{ This procedure blanks out a portion of the user's input in the event
of two boundaries not being distinct }

VAR
i,j : integer;
BEGIN
FOR i := y1 TO y2 DO
  FOR j := x1 TO x2 DO unset_bit(mat,i,j,plane)
END;

PROCEDURE drw_arrows(VAR mat : bool_matrix; VAR marked : boolean; row,x_min,x_max,plane : integer);
{ This procedure draws arrows to highlight potential tails }

CONST
len_arrow = 6;
len_tip = 3;
tip_width = 3;
above = -1;
below = +1;

VAR
l,k,j,m,
t,temp,
posn, { position of last change in beam }
side : Integer; { left or right of line depending on available space }
change : boolean;
BEGIN
j := x_min - 1;
posn := st;
side := -1;
marked := true;
no_arrows := no_arrows + 1;

WHILE j < x_max DO
  BEGIN
    j := j + 1;
    IF test(mat,row,j,3-plane) THEN
      BEGIN
        t := j;
        temp := t;
        WHILE test(mat,row,t+1,3-plane) DO t := t + 1;
        change := top_bot_check(mat,temp,row,above,t,3-plane);
        IF change THEN change := top_bot_check(mat,temp,row,below,t,3-plane);
        IF change THEN
          BEGIN
            insert_point(-1,j,row);
            IF j-posn < len_arrow THEN
              BEGIN
                side := -side;
              END;
          END;
      END;
  END;
END;
WHILE test(mat,row,j+1,3-plane) DO j := j + 1 
END;

{ draw the arrow }
FOR l := 1 TO tip_width DO 
BEGIN 
IF test(mat,row,l,3-plane) THEN
BEGIN 
WHILE test(mat,row,l+1,3-plane) DO l := l + 1;
posn := l 
END; 
END;
END;

IF test(mat,row,j,3-plane) THEN 
BEGIN 
WHILE test(mat,row,j+1,3-plane) DO j := j + 1;
posn := j 
END; 
END;

IF debugging(.2.) THEN 
BEGIN 
writeln('**** Debugging draw arrows ****');
writeln('row = ',row:0);
writeln 
END;

PROCEDURE del_tail(VAR work : bool_matrix; i,posn,last,plane : integer);

{ This procedure deletes tails identified by the user }

VAR 
below = 1; 
above = -1;

j, l, row : integer; 
bits_deleted : boolean; 

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END;

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;

BEGIN 
del_bits(work,l,posn,plane); 
row := i;
REPEAT 
row := row + 1; 
l := posn - 1; 
bits_deleted := false;
IF test(work,row,l,plane) THEN WHILE test(work,row,l-1,plane) DO l := l - 1;
IF NOT test(work,row,l,plane) THEN l := l + 1; 
IF test(work,row,l,plane) THEN IF test(work,row,l-1,plane) THEN 
BEGIN 
WHILE test(work,row,l,plane) DO l := l - 1;
posn := l 
END; 
END; 

END;
REPEAT
  j := 1 + 1
UNTIL test(work,row,1,plane) OR (1 > last);

j := 1;
posn := 1;
WHILE test(work,row,j+1,plane) DO j := j + 1;

IF test(work,row,1,plane)
  THEN
    IF top_bot_check(work,1,row,below,j,plane) AND
       (NOT end_check(work,row,above,1,j,plane) OR row=1) THEN
      BEGIN
        del_bits(work,row,1,plane);
        bits_deleted := true
      END
  UNTIL NOT bits_deleted;

IF debugging(.2.) THEN
  BEGIN
    writeln;
    writeln('**** Debugging delete tail ****');
    writeln('posn = ',posn:0);
    writeln('row = ',1:0);
    writeln
  END
END;

PROCEDURE del_bits(VAR work : bool_matrix; row,posn,plane : integer);

VAR
  col : integer;
BEGIN
  col := posn;
  WHILE test(work,row,col,plane) DO
    BEGIN
      unset_bit(work,row,col,plane);
      col := col + 1
    END;
END;

PROCEDURE best_arrow(VAR row,col : integer);

VAR
  dist : real;  { dist from hit point to tail }
  rw,cl,  { temp values for tail ordinates}
  x,y : integer;  { ordinate values of tails }
BEGIN
  dist := maxint;
  gen_ptr := polygon(.1.).base_ptr;
  BEGIN
    gen_ptr := polygon(.1.).base_ptr;
  END;
  IF debuggin...
WHILE gen_ptr <> nil DO
BEGIN
  x := gen_ptr@.poly_x;
y := gen_ptr@.poly_y;
  IF sqrt(sqr(row-y)+sqr(col-x)) < dist
  THEN
    BEGIN
      dist := sqrt(sqr(row-y)+sqr(col-x));
      rw := y;
      cl := x
    END;
    gen_ptr := gen_ptr@.poly_ptr
  END;
row := rw;  \{ set row & col to those nearest the 'hit' point \}
col := cl;
  IF debugging(.2.) THEN
  BEGIN
    writeln;
    writeln('****  Debugging  best  arrow  ****');
    writeln('row  =  ',row:0);
    writeln('col  =  ',col:0);
    writeln
  END;
END;

PROCEDURE kill_poly_in_save;
BEGIN
  gen_ptr := polygon(.0.).base_ptr;
  IF gen_ptr <> nil THEN
    REPEAT
      unset_bit(save_matrix,gen_ptr@.poly_y,gen_ptr@.poly_x,front);
    gen_ptr := gen_ptr@.poly_ptr
    UNTIL gen_ptr = nil
END;

PROCEDURE copy_poly_to_save;
VAR
  i,j : integer;
BEGIN
  FOR i := min_smudge_y TO max_smudge_y DO
    FOR j := min_smudge_x TO max_smudge_x DO
      IF test(work_matrix,i,j,back) THEN set_bit(save_matrix,i,j,back);
  IF debugging(.1.) THEN copy_mat(save_matrix,save_matrix,back)
END;

PROCEDURE get_label;
VAR
dummy : integer;
BEGIN


read_text('&Enter label for area ( <0 - 20> characters )',polygon(.pt_set_no.).poly_descript);
IF polygon(.pt_set_no.).poly_descript(.1.) <> nul THEN
BEGIN
read_int(1,10,'&Size for label = ? ',polygon(.pt_set_no.).size);
cmt_initialize(unit_no,ret_code);
read_int(0,1,'&Place cursor at position for label & RETURN',dummy);
target_in(polygon(.pt_set_no.).descript_posn.y,
polygon(.pt_set_no.).descript_posn.x);
END
END;

PROCEDURE background(VAR mat : bool_matrix; plane,disp1 : integer);
BEGIN
cmtl_command(' CLEAR GRAPHIC 1 ;',0);
cmtl_command(' CLEAR GRAPHIC 2 ;',0);
cmtl_command(' ADD GRAPHIC 2 ;',0);
IF init_comtal(wrt,disp1) THEN
BEGIN
disp_poly(mat,plane)
END
END;

PROCEDURE pair_and_distinctness_check;
BEGIN
IF debugging(.8.) THEN
BEGIN
COPY Mat(matr_i_x,matr_i_x,front);
write('l');
copy_mat(matrix,matrix,back);
write('l');
copy_mat(work_matrix,work_matrix,front);
write('l');
copy_mat(work_matrix,work_matrix,back);
END;
evens := evens_check(work_matrix,min_smudge_y,max_smudge_y,min_smudge_x,max_smudge_x,back);
cmtl_command(' CLEAR GRAPHIC 1 ;',0);
cmtl_command(' DISPLAY GRAPHIC 1 ;',0);
IF init_comtal(wrt,1) THEN
BEGIN
disp_poly(work_matrix,back);
END;
IF debugging(.8.) THEN
BEGIN
COPY Mat(matr_i_x,matr_i_x,front);
write('l');
copy_mat(matrix,matrix,back);
write('l');
copy_mat(work_matrix,work_matrix,front);
write('l');
copy_mat(work_matrix,work_matrix,back);
snap;
END;
IF question('&CHECK : Is the polygon distinct ? (Y/N)"
THEN
BEGIN
    cmtl_command(' ADD GRAPHIC 2 ;',0);
    IF NOT evens AND (no_arrows > 0) THEN
        BEGIN
            IF init_comtal(wrt,2) THEN
                BEGIN
                    disp_poly(work_matrix,front);
                    END
                END
            FOR arrow := 1 TO no_arrows DO
                BEGIN
                    cmt_initialize(unit_no,ret_code);
                    read_int(0,1,'SPlace cursor at correct arrow & RETURN',dummy);
                    target_in(row,col);
                    best_arrow(row,col);
                    j := col;
                    WHILE test(work_matrix,row,j,back) DO j := j + 1;
                    del_tail(work_matrix,row,col,j,back)
                END
            END
        ELSE
            k1 = poly_in_save;
            copy_poly_to_save;
            get_label;
            END
    ELSE
        BEGIN
            read_int(0,1,'SPlace cursor at top left hand corner for blank out & RETURN',dummy);
            target_in(top_row,top_col);
            read_int(0,1,'SPlace cursor at bot right hand corner for blank out & RETURN',dummy);
            target_in(bot_row,bot_col);
            blank_out(save_matrix,top_row,top_col,bot_row,bot_col,front);
            background(save_matrix,back,2);
            IF init_comtal(wrt,1) THEN
                BEGIN
                    disp_poly(save_matrix,front);
                    cmtl_command(' TRAC E GRAPHIC 1 ;',0);
                    read_int(0,1,'SRetrace blanked area & RETURN',dummy);
                    IF init_comtal(rd,1) THEN
                        BEGIN
                            secondary_initialize;
                            read_matrix;
                            copy_mat(save_matrix,matrix,front);
                            pre_process;
                            estab_index;
                            return_to_pool(pt_set_no);
                            pt_set_no := pt_set_no - 1;
                        END
                    END
                END
            END
        END
    END
END

IF debugging then
    IF init_comtal(wrt,1) THEN
        BEGIN
            disp_poly(work_matrix,back);
        END;
    END;
ENDIF
PROCEDURE read_and_process;
VAR
    count : integer;
BEGIN
    IF init_comtal(rd,1) THEN
        WHILE more_to_come DO
            BEGIN
                read_matrix:
                copy_mat(save_matrix,matrix,front);
                pre_process;
                estab_index;
                frst_pt := first_pt;
                WHILE frst_pt.found DO
                    BEGIN
                        curr_x := frst_pt.x;
                        curr_y := frst_pt.y;
                        pt_set_no := pt_set_no + 1;
                        cmtl_command('CLEAR GRAPHIC 1 ;',0);
                        cmtl_command('CLEAR GRAPHIC 2 ;',0);
                        tertiary_initialize;
                        find_area;
                        find_linear(frst_pt.x,frst_pt.y,frst.ind,frst.off);
                        delete_index(frst_pt.x,frst_pt.y,frst.ind,frst.off);
                        tall_and_distinctness_check:
                        frst_pt := first_pt
                    END;
                END;
            END;
        END;
    END;
ENDIF
BEGIN
    IF question('More polygons ? (Y/N)') THEN
        BEGIN
            more_to_come := true;
            background(save_matrix,back,2);
            IF init_comtal(wrt,1) THEN
                BEGIN

mat_init(save_matrix,front);
mat_init(matrix,front);
cmtl_command('TRACE GRAPHIC 1 ',0);
read_int(0,1,'$Trace desired area(s) & RETURN',dummy);

IF init_comtal(rd,1) THEN
BEGIN
  secondary_initialize;
  already_active := true
END

ELSE more_to_come := false;
END;

IF init_comtal(wrt,1) THEN
BEGIN
  fill_polygon(min_x,max_x,st,y_resolution,save_matrix);
disp_poly(save_matrix,front);
  FOR i := 1 TO pt_set_no DO
    IF polygon(.1.).poly_descript(.1.) <> nul THEN
      BEGIN
        count := 0;
        FOR j := 1 TO str_length DO
          IF polygon(.1.).poly_descript(.j.) <> nul THEN count := count + 1;
        print_text(polygon(.1.).poly_descript,count,graph,polygon(.1.).size, 
                    polygon(.1.).descript_posn.y,polygon(.1.).descript_posn.x, 
                    top_row,top_col)
      END;
    term_comtal(unit_no,ret_code);
    writeln;
    CASE ret_code OF
      0 : writeln('Termination of COMTAL O.K.');
      2 : writeln('Connection error');
      5 : writeln('COMTAL not operational')
    END
END

BEGIN { Main program }
  primary_initialize;
  read_and_process
END.

End of File
$2.88, $2.98T
Appendix B: Recursive Algorithm for Expansion

PROCEDURE expand(n : integer; VAR mat : bool_matrix);

VAR
row,          { row for pixel }
col : integer { col for pixel }
BEGIN  IF n >= 1 THEN
BEGIN
row := get_row(n);
col := get_col(n);
IF NOT pixel_set(row,col)
THEN
  IF check_expand(row,col)
  THEN
    BEGIN
      expand(n-1,mat);
      set_pixel(row,col,mat)
    END
  ELSE expand(n-1,mat)
END
ELSE expand(n-1,mat)
END;
END;

Where :
n : is the number of elements of the matrix which remain to
be processed. The number n serves both as a counter of
the unprocessed matrix elements and as a pointer to the
element to be processed. Both get_row and get_col derive
their values from n.
get_row : returns the row for the nth pixel,
get_col : returns the column for the nth pixel,
pixel_set : is a boolean function which determines whether
pixel (row,col) is set or not,
set_pixel : sets the (row,col)th pixel and
check_expand : is a boolean function which checks if a pixel
should be expanded or not.

The first call to expand is as follows:

expand((x resolution * y resolution),matrix)