from Computer Graphics, Vol. 13. No. 2 (August 1979)
SIGGRAPH '79 Conference Proceedings

GENERATING SMOOTH 2-D MONOCOLOR LINE DRAWINGS ON VIDEO DISPLAYS

Jose Barros
Mathematical Sciences
University of Texas at Dallas
Richardson, TX 75080

Henry Fuchs
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27514

ABSTRACT

One of the major drawbacks of video display systems for line drawing applications has been the poor image quality they usually produce -- "jaggy", "staircased" line edges, moire patterns in regions of closely spaced lines, even, with some systems, lines disappearing ("falling in") between pixels. Correcting these effects, with appropriate area-sampling techniques, has generally been too computationally expensive to adopt.

A new algorithm is presented which generates precise, smooth images of line drawings and solid polygonal-shaped objects on multi-grey-level pixel-mapped video systems. The method is based on an analysis of boundary conditions at each pixel affected by one or more lines. With this method a number of previously needed steps can be quickly eliminated. The commonality of boundary conditions between adjacent pixels and the coherence of such conditions in a raster-scan ordering of such pixels allows efficient generation of these boundary conditions. A recursive subdivision approach allows handling of arbitrarily complex cases by a simple boundary-analyzing technique. Compared with current line-drawing systems, a video system with this algorithm would also display an improved image with respect to certain common visual effects -- e.g., distance modulation of line intensity (which may be desirable), artificial small bright clusters of detail (which is undesirable).

Since the software interface to the algorithm may be handled through already-standard graphical subroutines, adoption of the algorithm may be accomplished without burdening graphic system users or graphic system-utilizing software.

This research was partially supported by the National Science Foundation under grant MCS77-03905.

INTRODUCTION

The use of video graphic terminals has increased significantly in recent years due to their inherent flexibility and virtually limitless display capacity. They are unsurpassed in generation of realistic, continuous-tone and color images, for example. They have not been widely utiized, however, for interactive line drawing applications, for which refresh line-drawing systems and storage tube displays have traditionaly been used. Refresh systems have much more convenient mechanisms for moving already drawn graphical entities and both these systems and storage tubes generally have higher spatial resolution than current video displays. This paper presents a method of using the video display's variable grey-level capability to alleviate some of its deficiencies due to its limited spatial resolution. (Although we focus in this paper on the application of this algorithm to video terminals, the basic ideas also apply to other output devices (printers, CRT's, even weaving looms) which can control the grey level of each picture element in the field of interest.)

Current methods of digital vector generation (see, e.g., Newman and Sproull(1973)) generally assume only a binary value for each picture element (pixel); the resulting line drawing invariably exhibits jaggy, staircase effects (fig. 1). If a small region of the screen is examined closely, the reason for these jaggy lines becomes apparent (fig. 2). The "ideal" edge of a line (now having considerable width in this close-up) may often go through the area of a pixel but not cover it completely. The line-drawing algorithm, however, makes a binary decision for each pixel, implying that it is either all covered by the line or not covered by it at all. This is sometimes done by sampling the point at the center of the pixel; if that point is covered, the pixel is considered covered; if that point is not covered, the pixel is considered not covered. In the above example, even a simpler, more direct algorithm was utilized. Since such an algorithm (a symmetric DDA in this case) simply generates a sequence of pixel positions which should be set to a "1", its decision to put a 1 at a particular pixel is somewhat more difficult to analyze. The obvious solution is to allow a greater range of choices for the line-

drawing algorithm than 0 or 1 and have the final value reflect the percentage of the pixel's area covered by line (fig. 3).

At first glance the solution would appear to be one of simply adding more precision to the line drawing algorithm which calculates the sequence of pixels affected by a given line segment; not simply setting them to 1 or 0, "covered" or "not covered" but calculating a value between 0 and 1 for each such pixel. The futility of any such sequential algorithm, however, is easily demonstrable. In these algorithms the line-segments are handled sequentially, i.e. all the affected pixels of one line segment are determined and appropriately set (even if to variable grey levels) before the next line segment is considered. Let us consider two cases composed of the lines in fig. 4: a) the picture consists solely of line segments 1 and 2; or b) the picture consists solely of line segments 1 and 3.
In either case, after the first line segments had been processed the value of pixel (47, 72) would be 0.5 (on a 0 to 1 scale). On processing the second line segment the algorithm would find 0.5 at (47, 72) but could not possibly decide (without knowing about the previous lines which caused the 0.5 value) how to alter the pixel. In case a) the pixel should, of course, be set to 1.0; in case b) it should be left at 0.5. Thus, in order to determine the proper value for a pixel, all the line segments affecting it have to be considered together. This suggests an algorithm which saves all the line segment information, then calculates the approximate pixel intensity values for every pixel in the image in some convenient (e.g., raster-scan) order, and indeed, our solution adopts this general approach.

PREVIOUS WORK ON THE PROBLEM

In the past few years, with the increased availabiity of multi-grey-level video displays and inexpensive LSI micro-processors, a number of researchers have focused on this problem, with encouraging results.

Some early work at Xerox Palo Alto Research Center on this topic is described in Shoup (1973).

Crow(1977) explains the undesirable visual effects in terms of aliasing phenomena inherent in sampled signals, and suggests a pre-sampling filter as a way to eliminate higher frequencies than the sampling one. More recent work is reported in Crow (1978).

Catmull(1978) of NY Institute of Technology described a visible surface algorithm for 3-D scenes which precisely calculates the value at each pixel by considering all the visible polygons there. This algorithm comes closest to our method. It's generality, however, prevents it from taking advantage of the simplifying observations central to our approach.

PROPOSED SOLUTION

As mentioned above, the overall structure of our solution is a raster-scan type algorithm, similar to Watkins(1970). Within a single pixel, a recursive subdivision approach is utilized (similar to Warnock(1969) and Sutherland(1973)) to calculate the grey-level value for this pixel's intensity, which is, of course, related to the fraction of its area not covered by polygons. (Of course, we could just as well have chosen to determine the fraction of its area covered by polygons, since the sum of these two values must reach unity; it turns out that the former may be just slightly more efficient to calculate than the latter.)

Before presenting the detailed algorithmic structure, we'll first discuss some of the basic elements of our approach.

Edges:

A line-drawing image consists of a set of line segments on the screen. In our algorithm each such line segment is converted into a (long, thin) polygon whose boundary is defined by edges (fig. 5). (In general, our algorithm can accept any convex polygon.) An edge is oriented, in that it has two sides, an open side (toward the outside of the polygon) and a closed side (toward the inside of the polygon).

Boundary Lists:

The basic processed entity in our system is a boundary list of edge intersections for a given area (e.g., pixel) of interest (fig. 6). In general an open area is one free of any polygon covering; a closed area is one covered by one or more polygons. As will be seen below, significant geometric information concerning the enclosed area can be deduced from such a boundary list, thereby avoiding more costly geometric calculations. (Note the difference between the boundary of an area and the boundary of a polygon.)

Considering now the boundary of an area of interest, a (polygon) edge crossing a boundary of such an area is considered to be either a leading intersection or a trailing intersection (fig. 6). A leading intersection is one through which a counter-clockwise traversal at the boundary enters a polygon. A trailing intersection is one through which such a traversal leaves a polygon.

Vertices:

A vertex of a polygon is marked on its two defining edges, each of which is extended to the nearest area boundary (fig. 7). An edge extended in this way forms a virtual intersection.

Extent of Polygon Cover:

A solid edge covers a boundary-defined area between its leading and trailing intersections -- this coverage extending around the boundary until the appearance of another edge of the same polygon (fig. 8).

The only enhancement needed to always be able to calculate the extent of polygon coverage from the boundary list is in the situation when an area is completely surrounded (and thus completely covered) by a polygon. This situation is always detectable in our raster-scan processing order in that before reaching a completely covered pixel the processing encounters at least one of the left edges of the surrounding polygon (fig. 9).
The solution adopted is to detect if the right edge of a pixel is completely covered and save the identification number(s) of the polygon(s) completely covering this edge. A fictitious edge of this polygon is then inserted at this right edge of the pixel so that the coverage of the next pixel can be deduced from its own boundary list (fig. 10).
No special additional processing is required as a result of this enhancement since the desired effect on this new pixel is exactly the same as the effect achieved by a "normal" edge being in this same position covering the entire pixel's area. We note also that this fictitious edge will automatically propagate until a right edge of the polygon is encountered.

Splitting Areas:

We recall that the overall goal of processing a pixel is to calculate the fraction of its area left uncovered by polygons. We adopt a divide-and-conquer approach to processing a single pixel, in that, if the situation inside the pixel (as reflected in its boundary list) is sufficiently simple (in our case this means 0 or 1 polygons present) then we calculate the remaining area directly. Otherwise we recursively subdivide the area until we reach sufficiently simple cases.

We always attempt splitting an area in two along a solid edge in hopes of being able to discard the closed side subarea. We can discard the closed side subarea if it is entirely covered by the splitting edge (fig. 11). (We can simply throw away any completely covered area since the final computation we seek is simply the percentage of the pixel area left open (i.e., uncovered by any polygon).)
A newly split area such as this is completely covered if an only if there is no edge in it from the same polygon as the splitting edge (see again fig. 8 a and b). Note, this implies that in the case where the splitting edge is "virtual" at one or both ends, the "closed" subarea is never immediately discarded (fig. 12 a and b).

The boundary lists increase the efficiency of the splitting process in two ways:
a. identifying a most advantageous edge along which to split the area,
b. identifying precisely those edges which need to be split in two as a result of this area splitting.

We identify a most advantageous edge along which to split the area by calculating a "depth count" for each edge along the boundary; this process, similar to parenthesis depth calculation, is easily achieved in a single scan of the boundary (fig. 13). A best edge is one with minimum depth count and one which completely covers its closed side subarea.

We note that the two ends of a splitting edge partition the boundary list into two contiguous pieces. The edges which are split are simply those with one end in one group and the other end in the other group. Again a simple parentheses-match type process identifies these edges. The boundary list for the splitting edge -- which will now become part of the boundary in each of the two subareas -- is constructed out of these edges just identified as crossing this line.

The complete boundary list for each of the new subareas is constructed out of one of the two parts of the old boundary list appended, in the proper order, to the newly computed boundary list of the splitting edge (fig. 14).
Splitting a Virtual Edge:

Since a virtual intersection is but a marker for an enclosed vertex, its edge need not be extended across any splitting boundaries (fig. 15). Thus these edges' intersections are eliminated from one of the two new subareas (the one which does not contain a vertex of that edge).

This property also implies that a virtual edge need not be considered until processing reaches the pixel in which its vertex (vertices) lies (fig. 16a), and also implies that virtual edges do not need to be carried along from one scan line to the next (fig. 16b).

SYSTEM OGANIZATION

Our method was designed to fit easily into existing video graphic systems. These systems almost always use a full-image memory buffer (a "frame buffer") from which the screen must be constantly refreshed in order to maintain the image on the video screen (fig. 17).

Drawing lines on the screen usually

consists of a number of MOVE(X,Y) and DRAW(X,Y) calls to low level procedures which, for a single line segment, determine and set the affected pixels in the image buffer.

The overall organization of our proposed solution involves storing away (say, on disk) each [(X1,Y1), (X2,Y2)] line-segment pair as it is encountered for processing by the current low level procedure. Then, after all the line segments for an image have been processed, this file of saved information is input to our algorithm, which calculates the more accurate value for each of the pixels on the screen which is affected by one or more line segments. It is interesting to note that adoption of our solution will be completely transparent to the user. A subtle visual effect may be noticed, however, a few seconds after normal ("old fashioned") processing has finished; this should be something of an "ephemeral scan" moving down the screen, adjusting the intensity of certain pixels in its path.

ALGORITHM DESCRIPTION

We now proceed with a description of the overall algorithm.

1. Creating Y Buckets on Disk

As described previously each original line segment (or other elementary symbol) is converted into one or more convex polygons, each polygon being defined by the sequence of edge intersections forming its boundary (see again, fig. 5). We established (usually on disk) a buffer for each scan line and as each polygon is processed, each one of its edges is put into the 'Y' bucket according to its top vertex. When the entire picture has been generated in the standard ("old fashioned) way all the edges will have been put into the proper 'Y' buckets on disk.

2. Scan Line Processing

The overall processing is done in the standard raster scan basis, proceeding from top to bottom on the screen, and in each scan line from left to right (as in Watkins(1970)). As Catmull(1978) properly notes, a scan line is really a finite strip of area the width of a pixel. (Although we continue to use the accepted term "scan line", we want to keep in mind that the term refers to an area.) For each scan line we keep two ordered lists of polygon edge intersections, sorted on X; one for the intersections with the top edge -- 'XTOP' -- and another for the intersections with the bottom edge -- 'XBOTTOM' -- (fig. 18). (As will be described below, the XBOTTOM list of one scan line becomes, with minor updates, the XTOP of the next scan line.)

We begin processing of a new scan line by taking the old XBOTTOM list and removing all virtual edge intersections from it -- these are precisely those edges whose bottom vertices were located in the previous scan line.

The new XBOTTOM is composed of the extensions of the (new) XTOP list and the new entries at this scan line. (This kind of updating is utilized in several raster-scan visible surface algorithms -- see Sutherland, Sproull, and Schumacker(1974) for a good exposition.) Some extensions of the XTOP edge intersections may intersect the XBOTTOM line outside the image area -- off the left or the right sides of the screen. For these edges, the intersections with the edges of the screen (instead of intersections with the XBOTTOM line) are used. The LEFT_LIST, which forms the left boundary of the first pixel on this scan line, is composed of all those entries of the XBOTTOM list with X values of 0.0.

These operations are summarized in the following statements:

SCAN_LINE_PROCESSING:

```
BUCKET ← edges in bucket for this Y value
        (probably from disk);
XTOP ← REMOVE_VIRTUALS( XBOTTOM );
XBOTTOM ← MERGE(XSORT(EXTEND(XTOP)),XSORT(BUCKET));
LEFT_ADJUST(LEFT_LIST,XBOTTOM);
XMIN ← MIN(LEFTMOST(XTOP),RIGHTMOST(XBOTTOM));
XMAX ← MAX(RIGHTMOST(XTOP),RIGHTMOST(XBOTTOM));
for X=0 to 511 do
   if (X<XMIN or X>XMAX)
      then output(BACKGROUND_COLOR)
      else PIXEL_PROCESSING;
end;
```

We now describe the processing at each pixel position.

PIXEL PROCESSING

Creating the 'Right' List:

As described previously we need to form the pixel boundary list, which is composed of a counterclockwise traversal of the pixel. This boundary is actually the concatenation of four lists: 'TOP', 'LEFT', 'BOTTOM', and 'RIGHT'.

We note that:

1. 'TOP' and 'BOTTOM' are already defined between the two pointers carried along the 'XTOP' and 'XBOTTOM' lists;
2. The 'RIGHT' list of one pixel becomes the 'LEFT' list of the next (except for "virtual" edge intersections which are dropped, and "fictitious" covering edges

which are added -- as described above).

In this fashion only one quarter of the overall pixel boundary list needs to be computed, the 'RIGHT' list (fig. 19). The computation of this list can be easily accomplished by first scanning the already existing 3 lists: 'TOP', 'LEFT', and 'BOTTOM'. Then, since each edge has two intersections in a closed boundary list (the area is a convex polygon), the entries for the 'RIGHT' list are exactly those intersections of 'TOP', 'LEFT' and 'BOTTOM' which do not have matches. All that remains to form the 'RIGHT' list is the calculation of these edge intersections with the right edge of the current pixel.

At this point there is a complete boundary list for the current pixel which is made of four separate boundary lists, all of them linked in a counterclockwise order and ready for the actual polygon cover processing. The overall pixel processing, then, can be summarized as follows:

```
PIXEL_PROCESSING:

  if X>0 then
      LEFT_LIST ← MODIFY_RIGHT(RIGHT_LIST);
  PIXEL_LIST ← FINISH_BOUNDARY(
      XTOP, XTOP_POINTERS,
      XBOTTOM, XBOTTOM_POINTERS,
      LEFT_LIST                    );
  output( AREA(PIXEL_LIST));
```

AREA calculates the extent of polygon cover for the area described by the PIXEL_LIST. As mentioned previously, it does this either by direct calculation (if the list contains either 0 or 1 polygons) or by recursively splitting the area until sufficiently simple subareas are encountered.

```
Recursive Function AREA(boundary_list)

  if SIMPLE(BOUNDARY_LIST) then
     return CALCULATE_AREA(BOUNDARY_LIST)
  else
     SELECT(BOUNDARY_LIST,SPLITTING_EDGE);
     CUT(BOUNDARY_LIST,SPLITTING_EDGE,
         OPEN_SIDE_LIST,CLOSED_SIDE_LIST);
  if UNIQUE_EDGE(SPLITTING_EDGE,CLOSED_SIDE_LIST)
     then return AREA(OPEN_SIDE_LIST)
     else return AREA(CLOSED_SIDE_LIST) +
                  AREA(OPEN_SIDE_LIST);

END AREA;
```

SIMPLE is a predicate (Boolean function) which determines whether or not the area represented by "boundary-list" is sufficiently simple to calculate the extent of polygon cover directly. An affirmative answer is reached if a) there are no polygons at all in the area or b) there is only a simple polygon or fraction thereof in the area.

CALCULATE_AREA directly calculates the area of polygon cover (in absolute pixel units) of a simple boundary-defined area.

SELECT determines a best edge to be used for splitting the area into two subareas.

CUT divides an area into two new areas creating a new boundary list for each subarea. It splits the initial list into two non-closed lists and then concatenates each non-closed list with the boundary list found for the splitting edge.

UNIQUE_EDGE determines whether or not the "splitting_edge"'s polygon has another edge which is in the "closed_side_list". (If it doesn't, then the entire closed_side area is covered by the "splitting_edge"'s polygon.)

We present in fig. 20 examples of the results of the just-described algorithm, as compared with the same data displayed on the same system (Genisco GCT3) using the standard line-drawing software.

CONCLUSION

We have presented a method for creating precise, smooth line drawings on multi-grey level video displays. The method determines the proper pixel values even in extreme cases in which many lines appear within a single pixel. Geometric calculations such as line intersections are minimized by reliance on simple analyses of ordered boundary lists.

With commonly available video monitors being limited at present to about 1024 x 1024 pixels, methods such as the one just presented may provide the least expensive ways to achieve higher quality images on these displays.

REFERENCES

Catmull, E. (1978), "A Hidden-Surface Algorithm with Anti-Aliasing", Proceedings of 1978 ACM-SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques, (Vol. 12, (3):6-11, Computer Graphics)

Crow, F.C. (1977), "The Aliasing Problem in Computer-Generated shaded Images", Communications of the ACM, 20 (11):799-805

Crow, F.C. (1978), "The Use of Grayscale for Improved Raster Display of Vectors and Characters", Proceedings of 1978 ACM-SIGGRAPH Annual Conference on Computer Graphics and Interactive Techniques, (Vol. 12, (3):1-5,

Computer Graphics)

Genisco Computers(1977) GCT3/GCT3a Programming Reference Manual, Irvine, California

Newman, W.M. & Sproull, R.F.(1973), Principles of Interactive Computer Graphics, McGraw-Hill

Shoup, Richard G. (1973), "Some Quantization Effects in Digitally Generated Pictures", Digest of Technical Papers of 1973 International Symposium of the Society for Information Display, 58-59.

Sutherland, I.E.(1973), "Polygon Sorting by Subdivision: A Solution to the Hidden-Surface Problem", unpublished

Sutherland, I.E., Sproull, R.F. & Schumacker, R.G.(1974), "A Characterization of Ten Hidden-Surface Algorithms", Computing Surveys, 6, 1

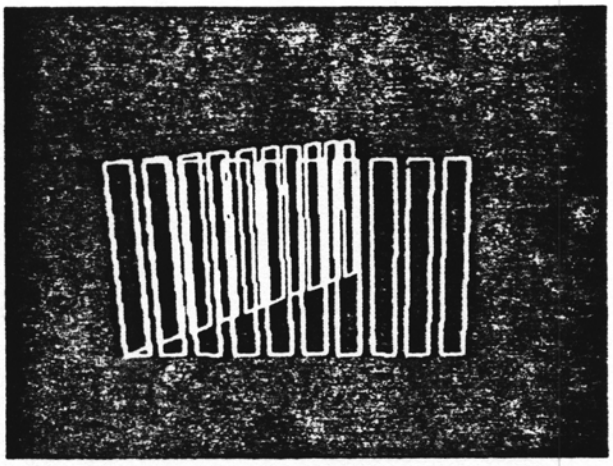


Fig. 1

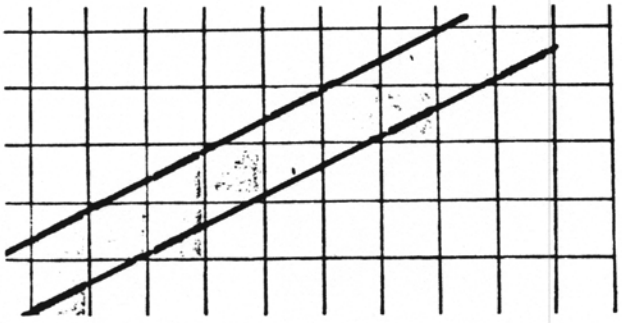


Fig. 2

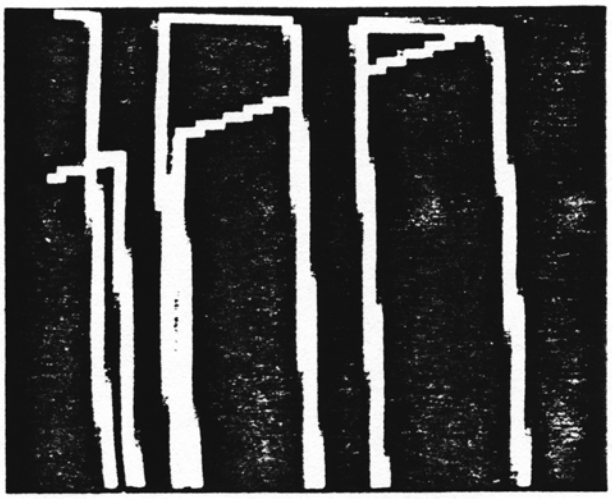
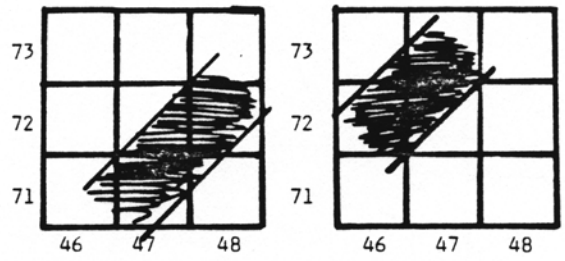


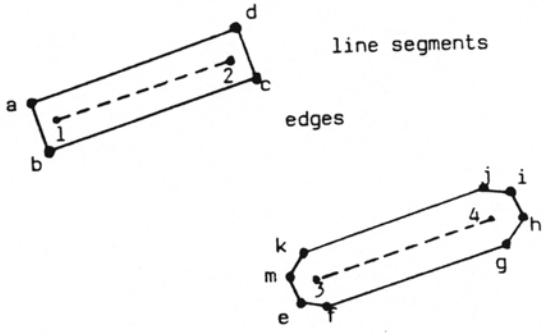
Fig. 3



line segment 1
(line segment 3 is similar)

line segment 2

Fig. 4



1-4: from application program
a-m: internally generated by system

Fig. 5

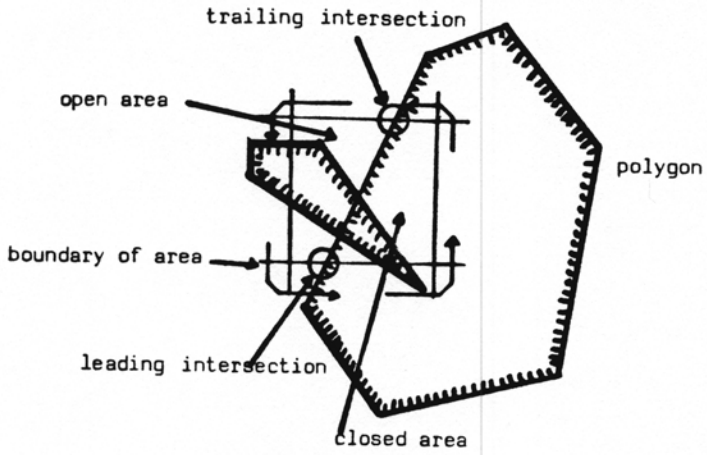


Fig. 6

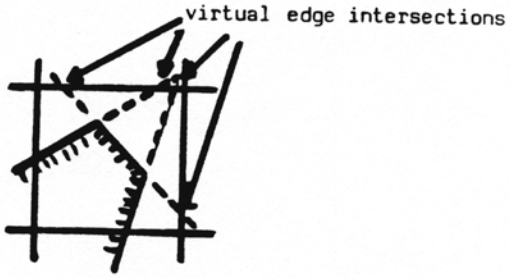
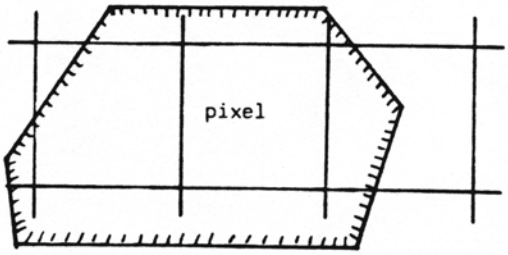


Fig. 7



no edge of the polygon in
this (completely covered) pixel

Fig. 9

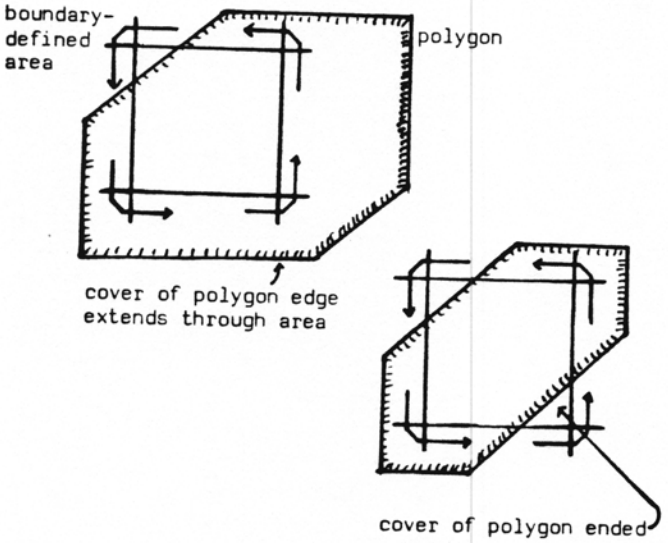


Fig. 8

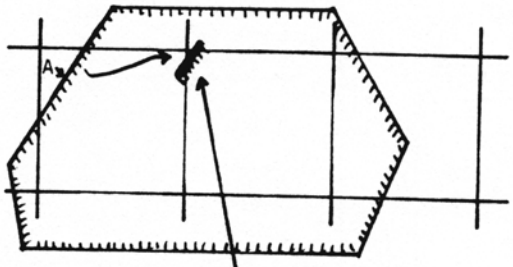


Fig. 10

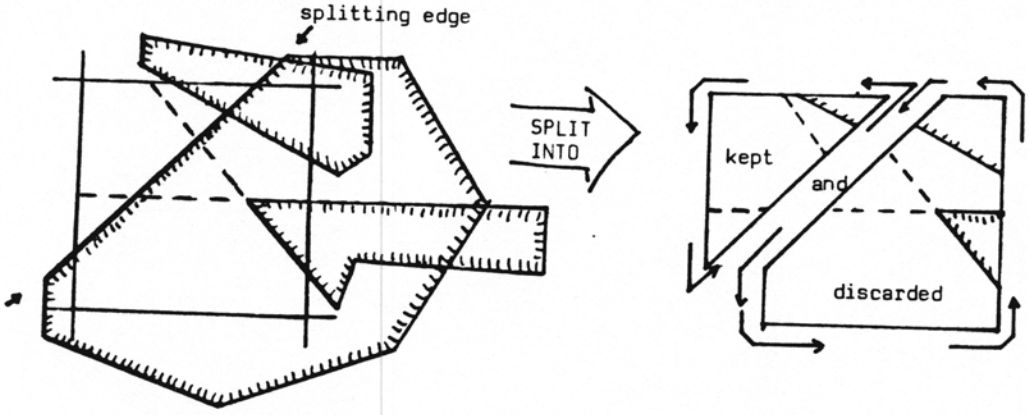


Fig. 11

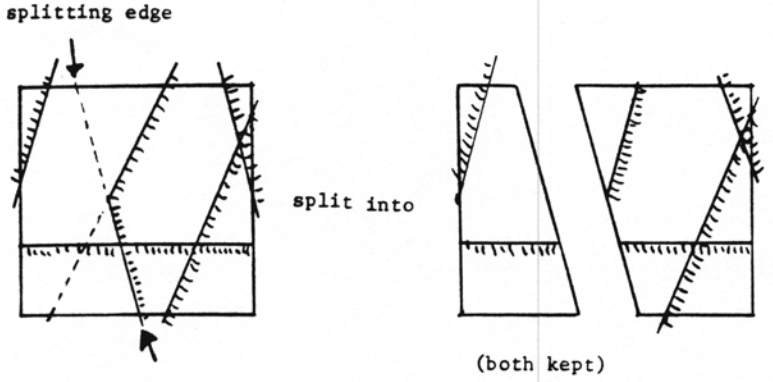


Fig. 12

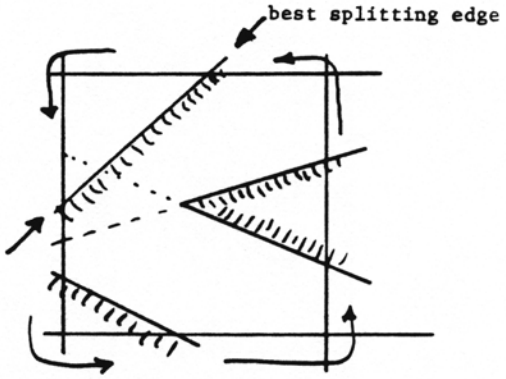
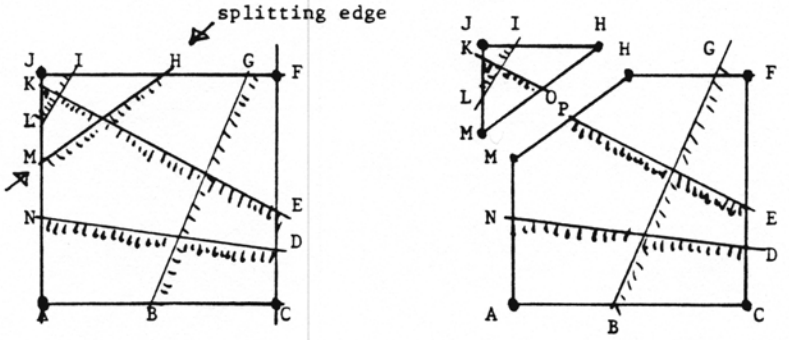


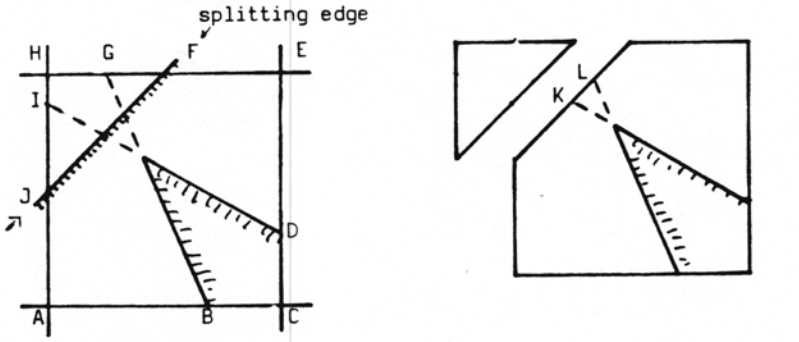
Fig. 13



Original boundary list: MNABCDEFGHIJKLM

Boundary lists after splitting: MNABCDEFGHP OHIJKLM

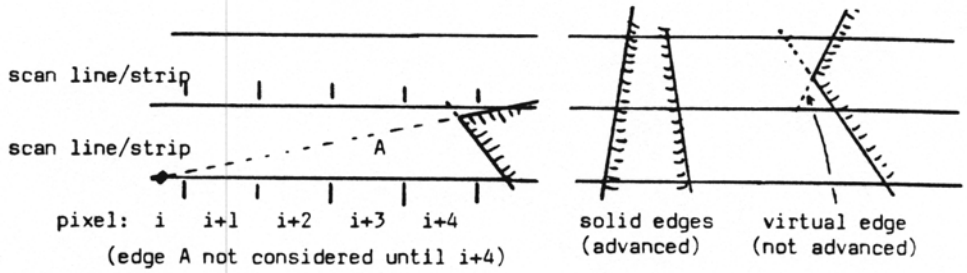
Fig. 14



original boundary list: J A B C D E F G H I J

new boundary list: J A B C D E F L K and F H J

Fig. 15



(a) **Fig. 16** (b)

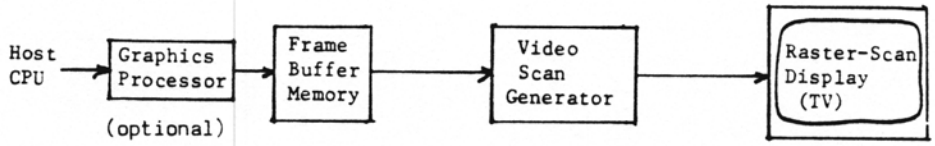


Fig. 17

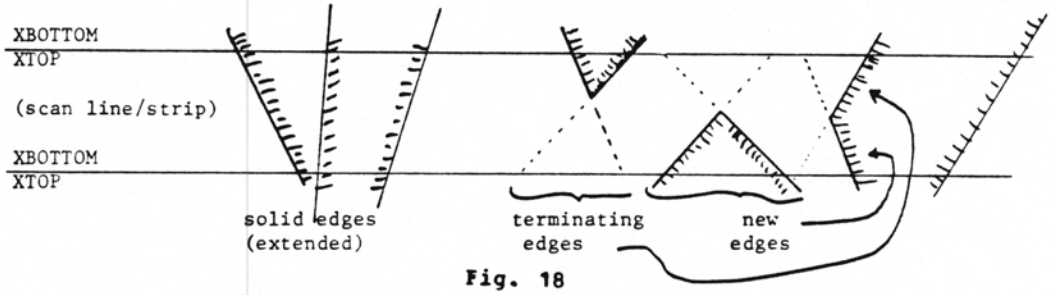


Fig. 18

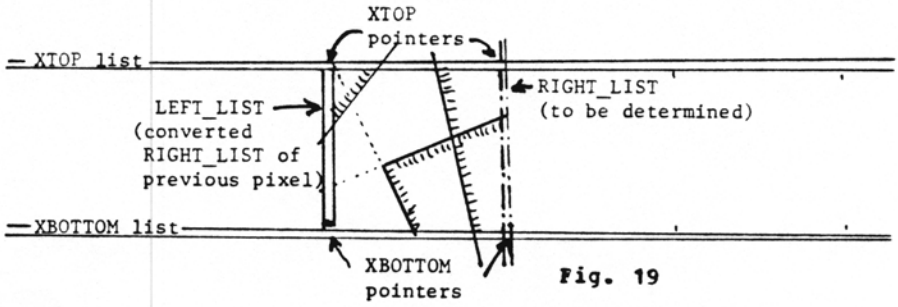


Fig. 19

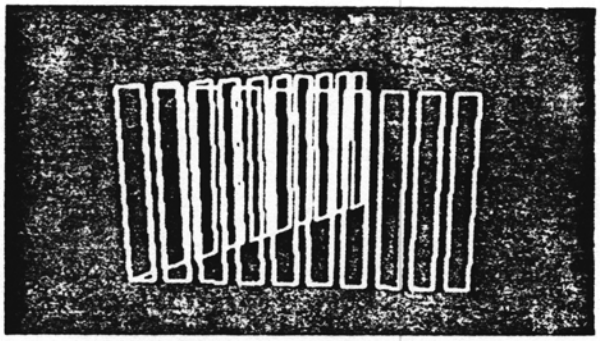


Fig. 20a: Standard processing
(same as Fig. 1; placed here for comparison)

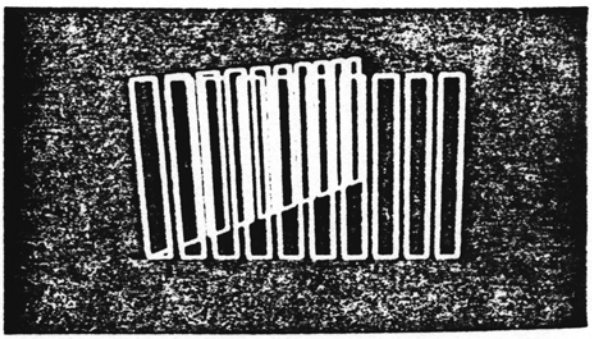


Fig. 20b: New processing

[All parts of Fig. 20 are 256 x 256-pixel images]

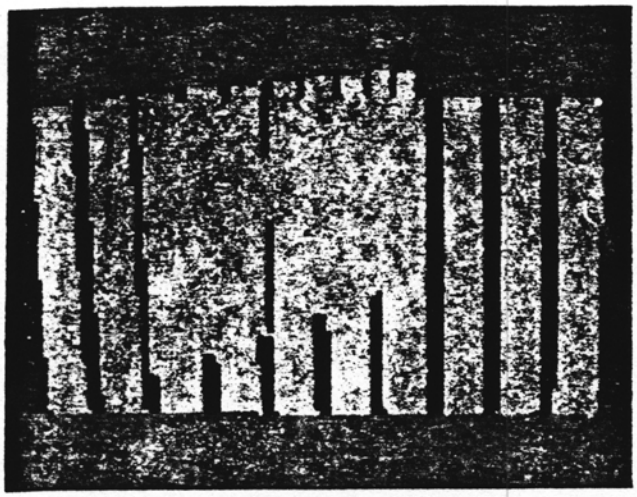


Fig. 20c: Standard processing

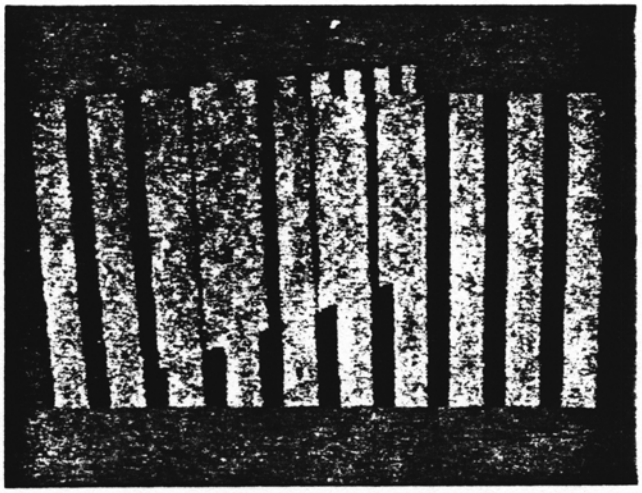


Fig. 20d: New processing

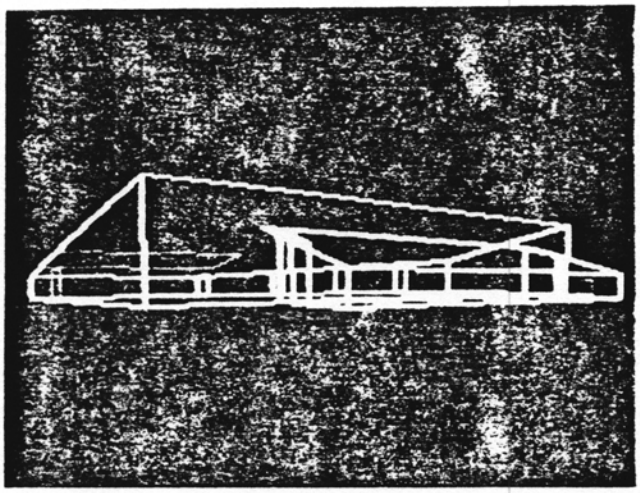


Fig. 20e: Standard processing (different line intensities indicate different colors on monitor)

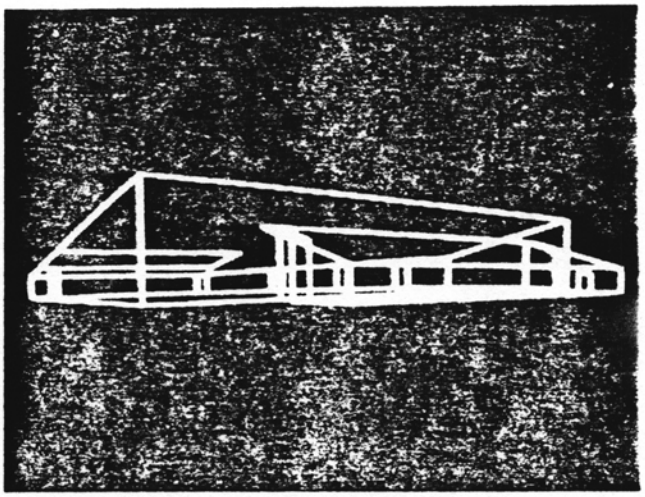


Fig. 20f: New processing