POLYGON PARTITIONING FOR ELECTRON BEAM LITHOGRAPHY OF INTEGRATED CIRCUITS

by

Frederick John Slawson

B.Sc. (Hons.), Simon Fraser University, 1969

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

SIMON FRASER UNIVERSITY

May 1985

APPROVAL

Name: Frederick John Slawson

Degree: Master of Science (Computing Science)

Title of Thesis: Polygon Partitioning for Electron Beam Lithography of Integrated Circuits

Examining Committee:

Chairperson: Binay K. Bhattacharya

Richard F. Hobson
Senior Supervisor
Associate Professor

Louis J. Hafer
Assistant Professor

Wo-Shun Luk
Associate Professor

John C. Dill
External Examiner
VLSI Design Tools and Systems Manager
Microtel Pacific Research Ltd.
Burnaby, British Columbia

Date Approved: 23 May 1985

PARTIAL COPYRIGHT LICENSE

Title of Thesis/Project/Extended Essay

POLYGON PARTITIONING FOR ELECTRON BEAM

LITHOGRAPHY OF INTEGRATED CIRCUITS

Author: ______________________
(signature)

FREDERICK JOHN SLAWSON
(name)

11 SEPTEMBER 1985
(date)

ABSTRACT

Electron beam lithography is a technique used to prepare high resolution masks for integrated circuit fabrication. While most integrated circuit layouts are described by rectangles and polygons, electron beam lithography systems accept only a few simple geometric shapes (triangles and/or quadrilaterals) as input data. Thus software which prepares data for electron beam systems must partition polygons into suitable geometric primitives. Most algorithms which have been devised to perform this partitioning suffer from a lack of generality. This thesis describes the selection and refinement of a polygon partitioning algorithm, resulting in an algorithm of high generality. The refined algorithm has been successfully implemented in the context of a complete electron beam lithography data preparation program.

Little effort appears to have been made to optimize mask data so that electron beam machine mask making time is reduced. Several possible optimization techniques are described. One, which partitions polygons along non-horizontal (as well as horizontal) lines, is explored by developing a heuristic. An algorithm based on this heuristic has been introduced into the data preparation program described above. The observed impact on data optimality is small; however, it does represent a beginning in the area of electron beam lithography data optimization.

ACKNOWLEDGEMENTS

Financial support was provided for the major part of this research by the Natural Sciences and Engineering Research Council of Canada, in the form of a Postgraduate Scholarship. This support is gratefully acknowledged.

Microtel Pacific Research Ltd. (Burnaby, British Columbia) kindly allowed the use of software which formed part of the computing environment in which this research was conducted. Their cooperation is much appreciated.

And last, but not least, I am grateful to my wife Linda, for enduring the long evenings and weekends which I spent preparing this thesis. Thank you, Linda, for your support and your patience.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

INTRODUCTION

1.1. PREVIOUS WORK

Electron beam lithography is rapidly taking over from conventional optical methods for producing large scale integrated circuit (LSI) and very large scale integrated circuit (VLSI) masks. This new method offers both higher speed and higher accuracy, so its growing popularity is no surprise.

Electron beam machines require data in a special format. Previously developed software to prepare such data has not always been reliable. One example of this unreliability was discovered while testing a program written to convert electron beam data back to the original design database format (section 2.4.2). A simple integrated circuit (IC) design was converted into electron beam (e-beam) format, using software on one of the original (and widely used) commercial IC design workstations. The e-beam data was transported (via magnetic tape) to our computer, and used as test data for the conversion program. When the output was plotted, a small but distinct error was observed. Hand-decoding a hexadecimal dump of the e-beam data demonstrated that the error was already present in that data, before it was processed by the conversion program. The data used as input to the workstation was verified to be correct. Thus the error had to be introduced by the workstation software. The error was small enough to have gone unnoticed up to this time, but it was present nonetheless.

Another example of software unreliability was reported to the author during personal correspondence with a major U.S. aerospace corporation. That company had purchased software to prepare data for electron beam lithography, only to find that they had to correct program errors.

Some previously developed software has imposed unreasonable restrictions on the IC designer. For example, self-intersecting polygons are often prohibited; this forces the designer to manually partition such polygons. Also, there appears to have been little effort made to optimize the data so that electron beam machine time is reduced. (This refers to the time taken to process the data and expose the mask.)

1.2. OBJECTIVES

The objectives of this research were

(1) to develop software to prepare electron beam lithography mask data which is accurate and reliable, while imposing no unreasonable restrictions on the designer, and

(2) to explore methods to optimize electron beam data so that e-beam machine time (for data processing and mask exposure) is reduced.

1.3. BACKGROUND

1.3.1. Processing of IC Chips

The processing of IC chips involves the creation of layers of different materials on and under the surface of a silicon wafer. The circuit which is produced is determined by the patterns in which these layers are formed. The fundamental steps in the creation of one layer are as follows (Figure 1-1).

(1) The partially processed silicon wafer is coated with the material needed for this layer. Typical materials are aluminum, polycrystalline silicon, and silicon oxide (made by oxidizing the surface of the wafer).

(2) A coating of photoresist is applied over top of this material.

(3) The photoresist is exposed to ultraviolet light in the desired pattern.

(4) The photoresist is developed. (This involves dissolving the more soluble areas of the

photoresist in a suitable solvent. In positive resists, the more soluble areas are those which have been depolymerized by exposure to the light. In negative resists, the unexposed areas are more soluble, while the exposed areas become polymerized and less soluble.)

(5) The areas (of material applied in step 1) no longer protected by photoresist are etched away (*e.g*. by acid or plasma).

(6) Remaining photoresist is stripped off with an appropriate solvent.

(7) Any required processing (*e.g*. ion implantation) is performed.

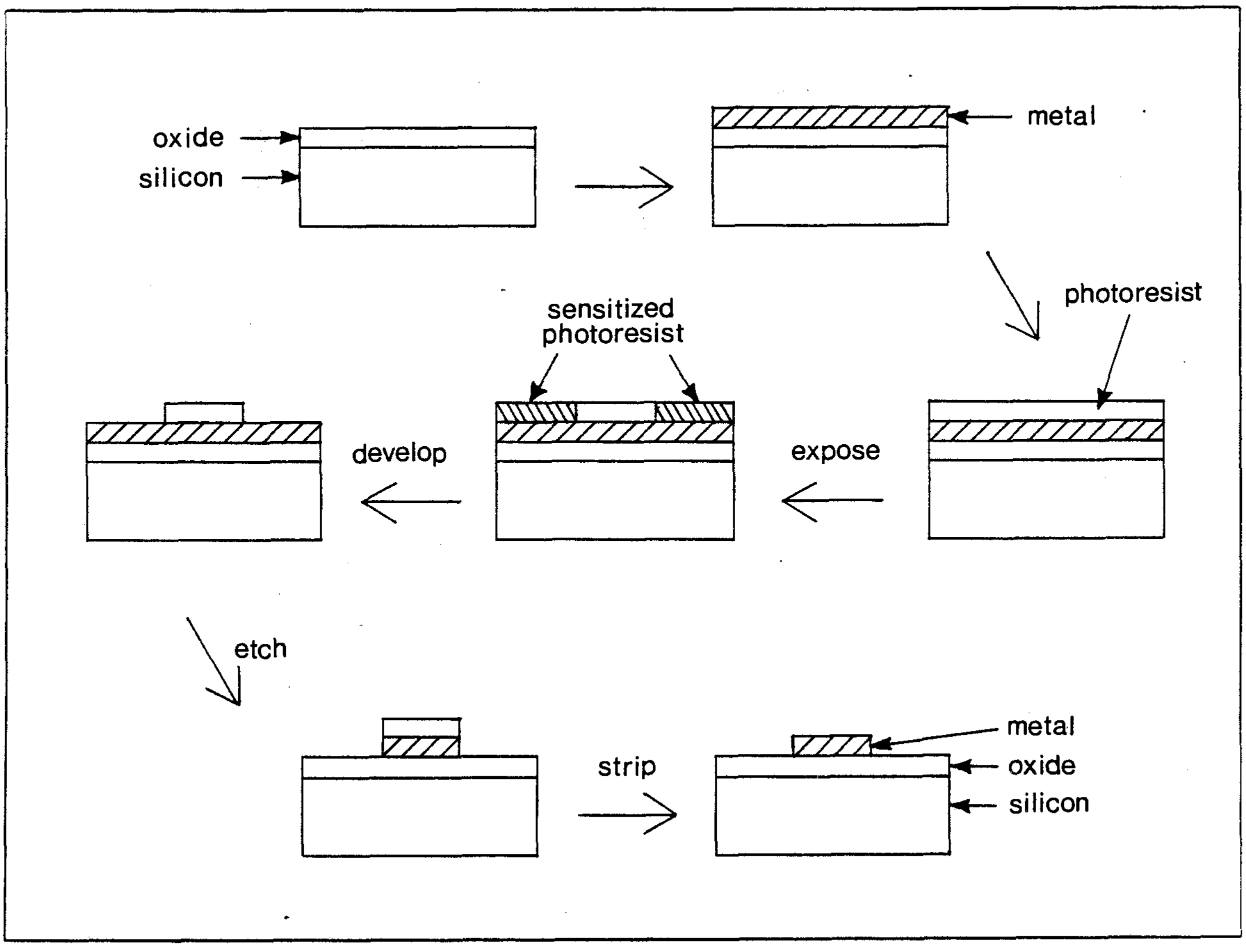

Figure 1-1 Creation of Metal Layer

1.3.2. Exposure of Photoresist

The IC manufacturing step of particular interest in this study is the polymerization or depolymerization of photoresist in the desired pattern. There are two methods for doing this: "optical" and "direct write on wafer".

The conventional method uses the optical approach. A "mask" containing clear and dark areas is placed between an ultraviolet light source and the wafer. When the light is turned on, those areas of the photoresist not blocked by the dark regions of the mask are exposed. (The mask is somewhat analogous to a photographic "negative", except that most negatives have continuously variable density instead of the binary "dark" or "clear" nature of masks. Also, a mask may be a positive image instead of negative.) Thus the pattern on the mask determines the pattern in which the photoresist is exposed. One mask is required for each layer in the IC.

In the "direct write on wafer" method, the photoresist is replaced by an electron resist, and the ultraviolet light and mask are replaced by a computer-controlled electron beam. The beam is selectively blanked as it is scanned over the surface of the wafer. The pattern in which the electron resist is exposed is determined by the blanking/non-blanking behavior of the beam.

While the direct write on wafer method eliminates the need for making masks and offers higher resolution, it is much slower than the optical method. (At present, it takes considerably longer to expose a wafer with an electron beam than it does optically.) For this reason, the optical method is preferred in all but low volume research applications, and for ICs requiring greater resolution than possible optically.

1.3.3. Multi-Chip Wafers

Chips are not fabricated one at a time, for to do so would be economically and temporally undesirable. Instead, a number of chips are laid out in a grid-like pattern on a single silicon wafer (Figure 1-2).

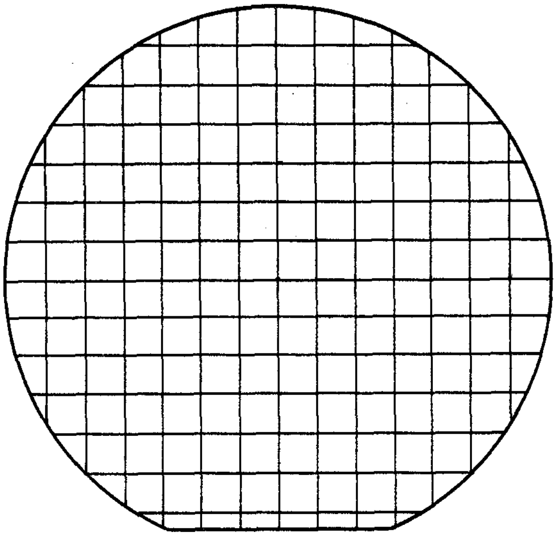

Figure 1-2 Multi-Chip Wafer

The wafer is processed intact, and then sawed up into individual chips after all processing has been completed. A typical 4 inch wafer will produce 100 — 500 chips, depending on chip size. In a production wafer, all chips are identical, whereas a research (multiproject) wafer contains a variety of chips.

For any given layer, the entire wafer is usually sensitized by a single exposure. Thus the multi-chip layout is an inherent part of each mask. (Note: as VLSI minimum feature size shrinks, there will be an increasing tendency to expose each chip on the wafer individually, so that layer to layer registration requirements may be met.)

1.3.4. Mask Making

Making a mask is very similar to the process used to create one layer of an IC. (See Figure 1-3.)

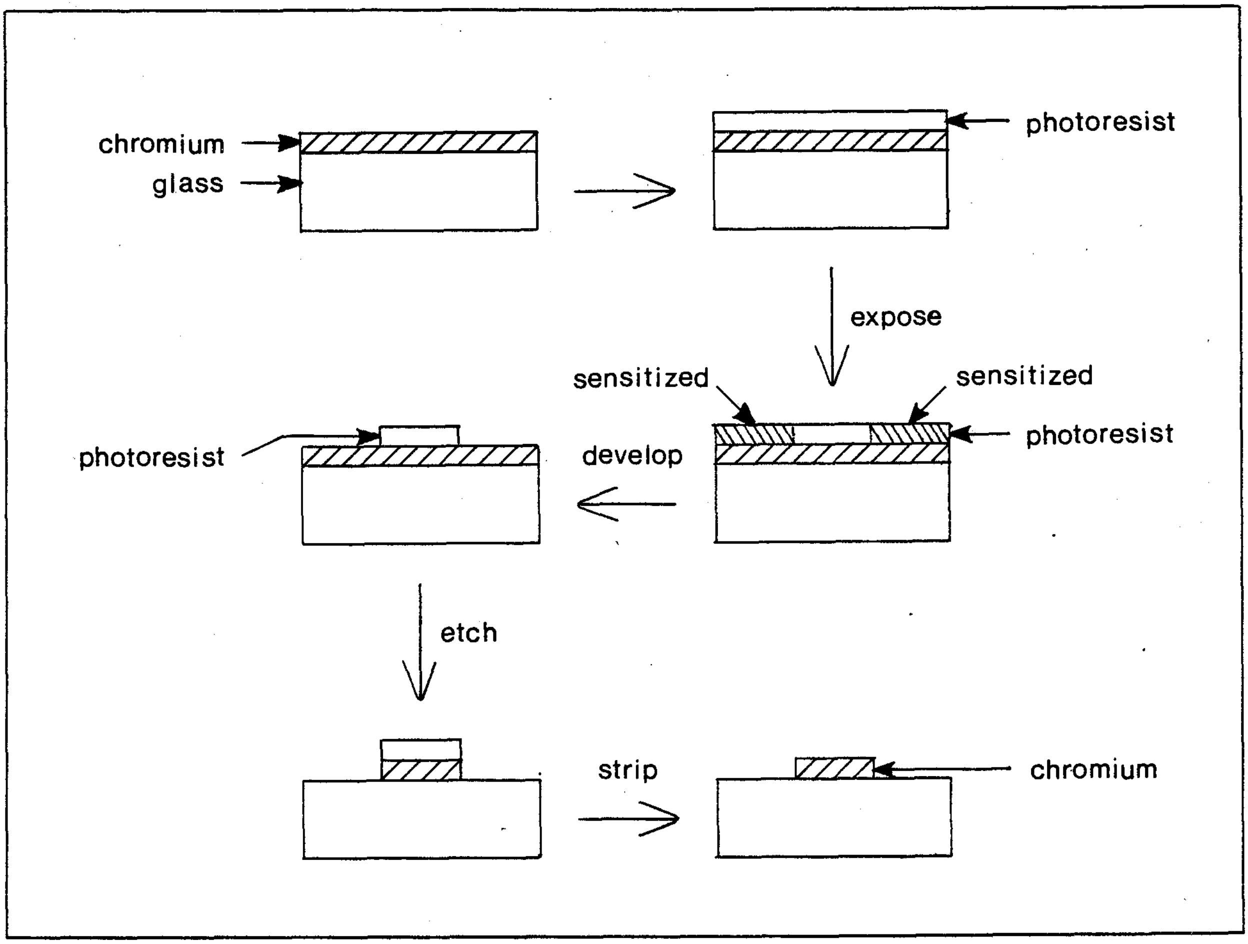

Figure 1-3 Mask Making Process

(1) A glass plate is coated with metal (usually chromium).

(2) The metal is coated with photoresist or electron resist.

(3) The resist material is exposed in the desired pattern, causing polymerization or depolymerization.

(4) The resist is developed (dissolving the more soluble areas).

(5) The metal no longer protected by resist is etched away.

(6) The remaining resist is stripped off.

This leaves a pattern of dark (metal) and clear (glass) areas.

1.3.4.1. Exposing The Mask

The key step in placing the correct pattern on a mask is exposure of the mask resist material. Like exposure of the resist on a wafer, there are two methods: optical and electron beam lithography.

1.3.4.1.1. Optical Method The optical method consists of the following steps.

(1) An intermediate mask is made at ten times the final size. (This is called a *10× reticle.*) Exposure of the reticle's photoresist is done using an *optical pattern generator* (also called a *block flasher.*) This is a device which projects an image of a rectangular aperture onto the reticle, which is mounted on a movable stage. A xenon flashtube provides the light source. By changing the aperture shape, size, and orientation, and by moving the stage between flashes, the photoresist can be exposed in the desired pattern. The optical pattern generator is controlled by a computer, which reads data representing the pattern. Once exposed, the reticle is developed, etched, and stripped of remaining photoresist, as described above.

(2) An image of the reticle, reduced to actual chip size, is projected onto the mask's photoresist.

(3) After the first exposure has been made, the mask is moved and a second exposure is made, right beside the first one.

(4) The mask is stepped to the next chip position, and a third exposure is made. This *step and repeat* process is continued until all desired copies of the reticle image have been made.

1.3.4.1.2. Electron Beam Lithography Method Instead of photoresist, an electron resist is used to coat the mask. The mask is mounted on a movable stage in an evacuated chamber, and is exposed using a focused beam of electrons. Beam position, beam blanking, and stage position are controlled by a computer using data which represents the desired pattern. The electron beam traces out the pattern at actual chip size, for all instances of the chip on the mask.

1.3.4.2. Advantages of Electron Beam Lithography

The fact that there were over fifty commercial e-beam lithography systems developed by 1982 [2] attests to the rapidly growing popularity of this method. The reason for this popularity is a combination of speed and accuracy.

(1) *Speed*. The elimination of the intermediate reticle and the absence of an optical step and repeat process result in significantly faster turnaround for masks exposed by e-beam lithography.

(2) *Accuracy*. The inherently shorter wavelength of electrons allows smaller features to be written, and the absence of a separate step and repeat process results in better layer to layer registration. For these reasons, e-beam lithography is capable of higher accuracy than optical methods. (Note that X-ray lithography also provides a suitably short wavelength, but still requires a separate step and repeat process [2].)

1.3.5. Electron Beam Lithography Systems

E-beam lithography systems can be classified into two categories, depending on their beam scanning strategy: raster scan and vector scan.

1.3.5.1. Raster Scan

In the raster scan technique, the electron beam is scanned back and forth in a regular pattern over the surface of the mask (or wafer). The beam is thus addressed to every possible position within the writing field, whether exposure is needed there or not (Figure 1-4).

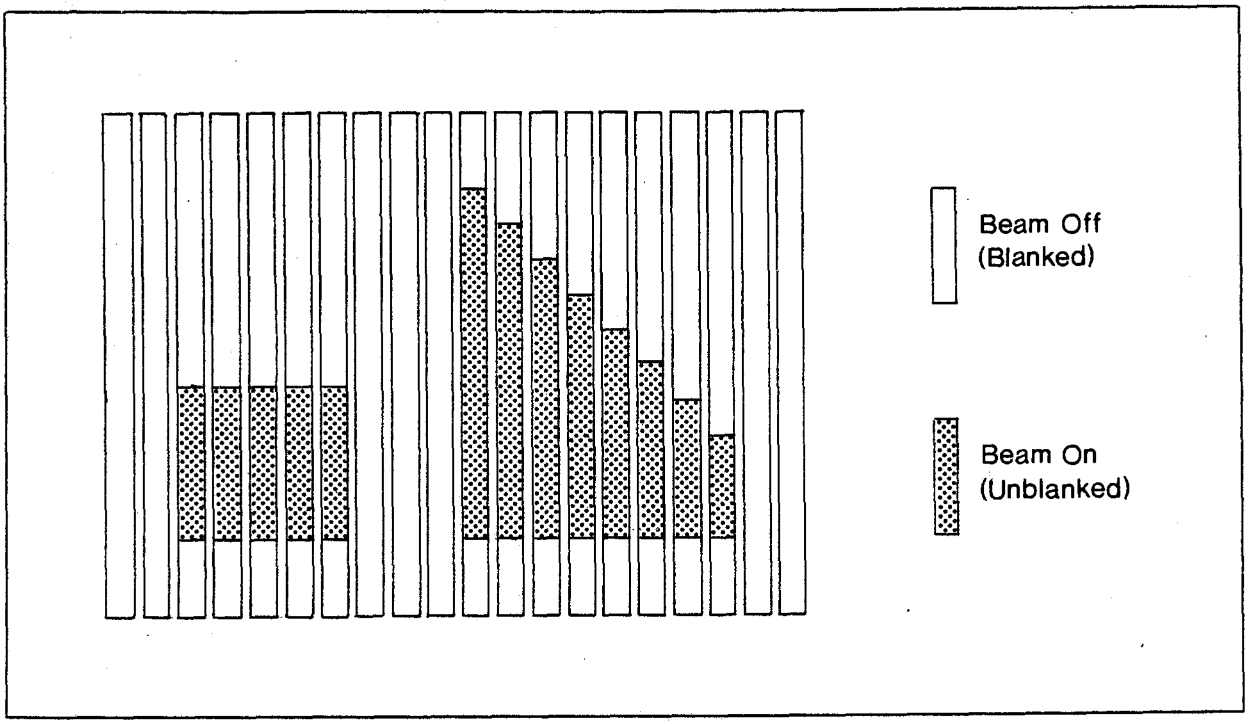

Figure 1-4 Raster Scan Technique

Whenever the beam is aimed at a spot which should not be exposed, it is blanked (deflected away from the mask). The beam is actually scanned in one dimension only; motion in the other dimension is provided by moving the stage on which the mask is mounted.

1.3.5.2. Vector Scan

In the vector scan technique, the electr beam is blanked and addressed to the position of a figure. It is then unblanked and scanned back and forth to fill in the area enclosed by the figure. Next, it is blanked and addressed to the position of the next figure. This sequence continues until all figures within the writing field have been exposed (Figure 1-5).

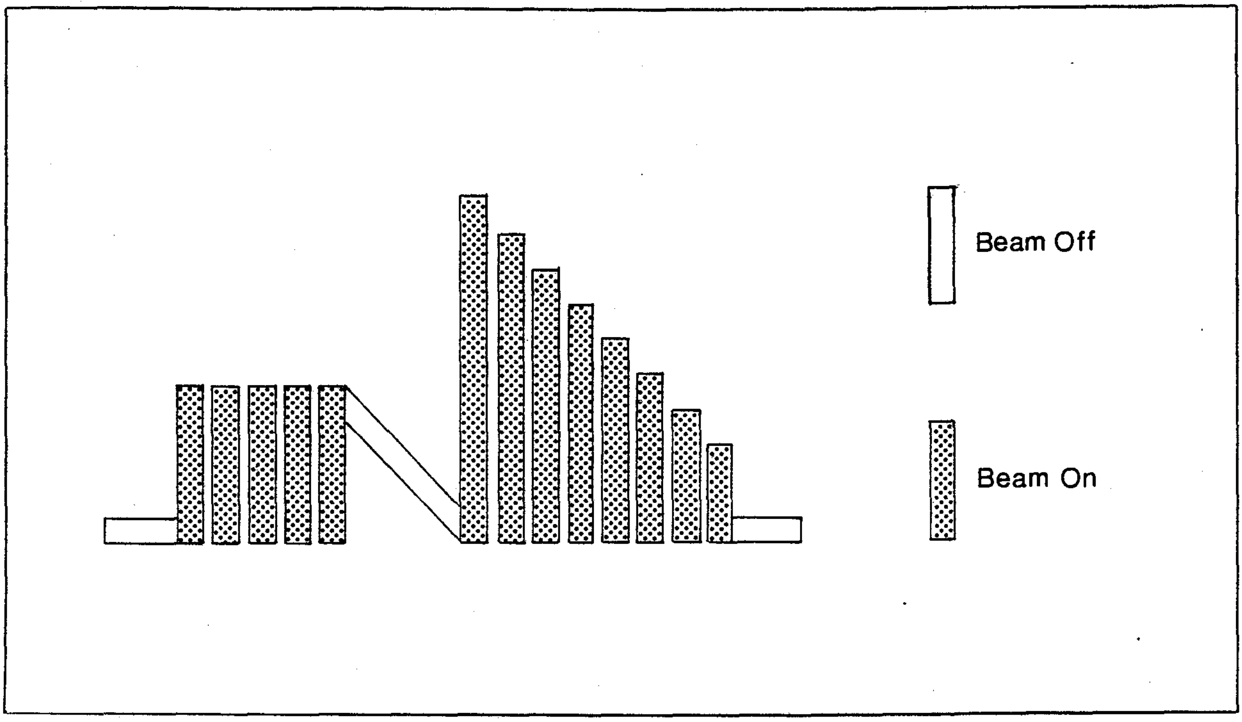

Figure 1-5 Vector Scan Technique

The stage is then moved to the next writing field, and exposure continues.

Unlike the raster scan technique, the beam is addressed to only those positions needing exposure (and those lying on the paths between figures). Also, scanning of the beam takes place in two dimensions (not one). In most vector scan machines, the stage is stationary during exposure (instead of moving in one dimension, as in raster scan machines).

1.3.5.3. Speed Considerations

At first sight, one would expect raster scan to be the slower of the two techniques, since much time is wasted scanning the electron beam over positions which need no exposure. In practice, however, this is not the case.

In vector scan, each movement of the beam from one figure to the next requires an adjustment of the magnetic and electrostatic fields used to aim the beam. These fields are controlled by voltages and currents — analog quantities. Since the description of each figure is in the digital domain, a digital-to-analog conversion is required each time a new figure is to be started. The time required for these digital-to-analog conversions, together with the settling time of the beam following each jump between figures, are responsible for making vector scan machines slower than raster scan.

1.3.5.4. Recent Developments In Vector Scan

Several recent developments in vector scan systems are worthy of note.

- *Beam Shaping*. In this technique, the electron beam shape can be changed from the conventional fixed size round spot to a variety of shapes and sizes. By dynamically shaping the beam, considerable time can be saved while filling in areas to be exposed [5, 9, 10].
- *Continuously Moving Stage*. Fields are still written one at a time, but the stage moves continuously during writing. To make this possible the beam must track the stage as it moves. In a conventional vector scan machine, the stage is moved only between writing of adjacent fields. Following each stage movement, writing must be delayed until stage oscillations have settled. The continuously moving stage approach eliminates this delay [9].

With refinements such as these to vector scan systems, it is likely that they will eventually become faster than raster scan. At the present, however, commercially available raster scan machines are faster.

1.3.6. The Perkin-Elmer MEBES Machine

The software developed in this project produces data for the Perkin-Elmer MEBES machine. This is a raster scan e-beam lithography system capable of direct write on wafer as well as writing masks. (The acronym "MEBES" stands for *Manufacturing Electron Beam Exposure System* . The word "manufacturing" refers to the direct write on wafer capability.) The original version of the e-beam data preparation software was developed for Microtel Pacific

Research Limited, Burnaby, British Columbia, during two work semesters of a Co-operative Education Program. The fact that Microtel employs mask vendors which use the Perkin-Elmer machine explains the choice of this particular e-beam system.

1.3.6.1. Components

The Perkin-Elmer MEBES machine consists of the following components:

- a Data General Eclipse minicomputer, which handles job control and preprocesses geometric figure data from the input files
- a microprocessor, which rasterizes geometric figures and controls the electron beam unit
- a bit map memory, which stores the rasterized figures
- and an electron beam unit. This includes the electron beam column, and the stage to hold the mask or wafer. The stage is movable in both horizontal dimensions under control of the microprocessor, and is positioned accurately by using a laser interferometer servomechanism.

1.3.6.2. The Stripe Concept

The simplest way to save rasterized figures in the bit map would be to use the bit map like the screen memory of a raster scan display device; each bit would correspond to one pixel. Unfortunately, this approach would require an enormous amount of memory. For example, a typical 4 inch wafer could produce 300 square chips, each 5 mm on a side. If the mask for such a wafer were exposed with a 0.5 micrometer diameter beam, it would consist of 30 billion pixels, and the bit map would require approximately 4 gigabytes of memory. Just one 5 mm square chip, exposed with a 0.5 micrometer spot, would require 100 million pixels, and a 12.5 megabyte bit map. Even this is more memory than available on the MEBES system, and so a chip must be rasterized in sections.

To avoid distortion, the distance actually scanned by the electron beam is quite small, typically no more than 250 micrometers. On the other hand, the width of an entire chip can be covered by moving the stage. Since the beam is scanned in one dimension while the stage is moved in the other, the mask is exposed in a series of long, narrow "stripes". The obvious way to rasterize a chip in sections is to make each section correspond to one of these stripes. The bit map can then store the rasterized image of one stripe. When the data is organized into stripes, the x axis is, by convention, parallel to the direction of stage travel, so each stripe is oriented horizontally. (By making the stripe length longer than most chips, the need to section the data vertically is avoided.)

MEBES data preparation software must thus organize the data into stripes, and must partition figures which span boundaries between stripes (Figure 1-6).

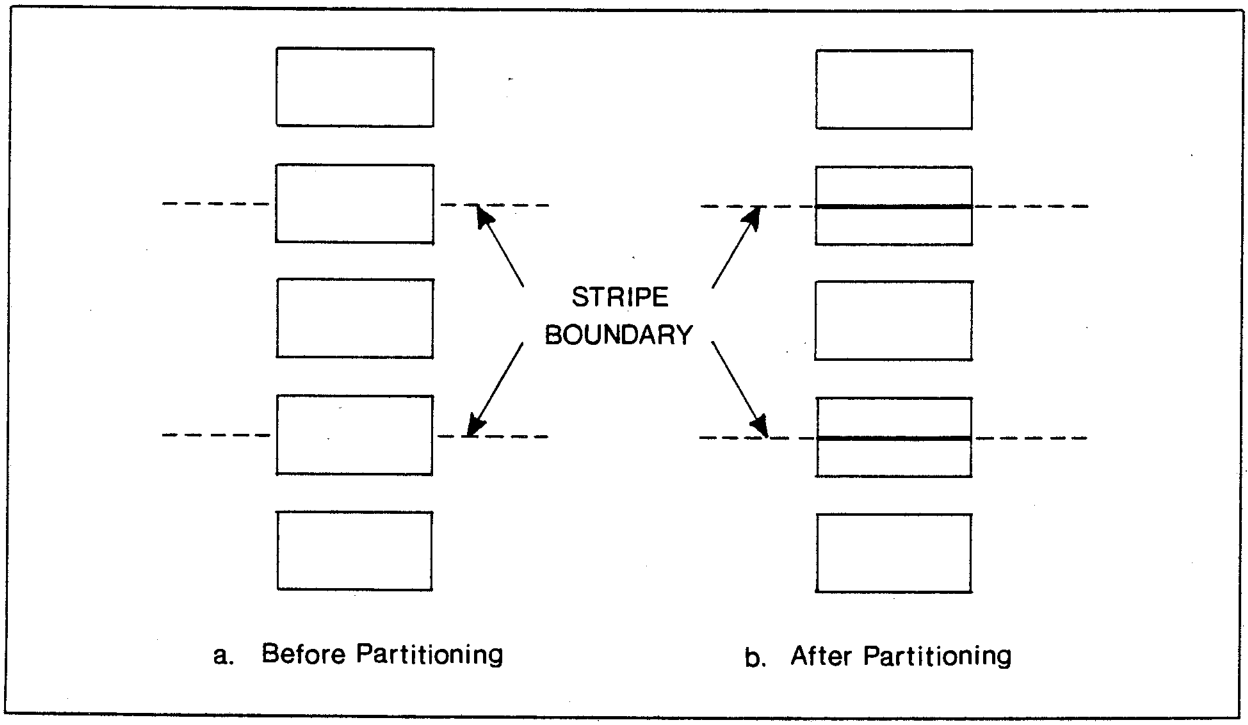

Figure 1-6 Partitioning At Stripe Boundaries

1.3.6.3. Figures Accepted

The MEBES machine software accepts only five types of geometric figures: rectangles, parallelograms, and three types of trapezoids (Figure 1-7).

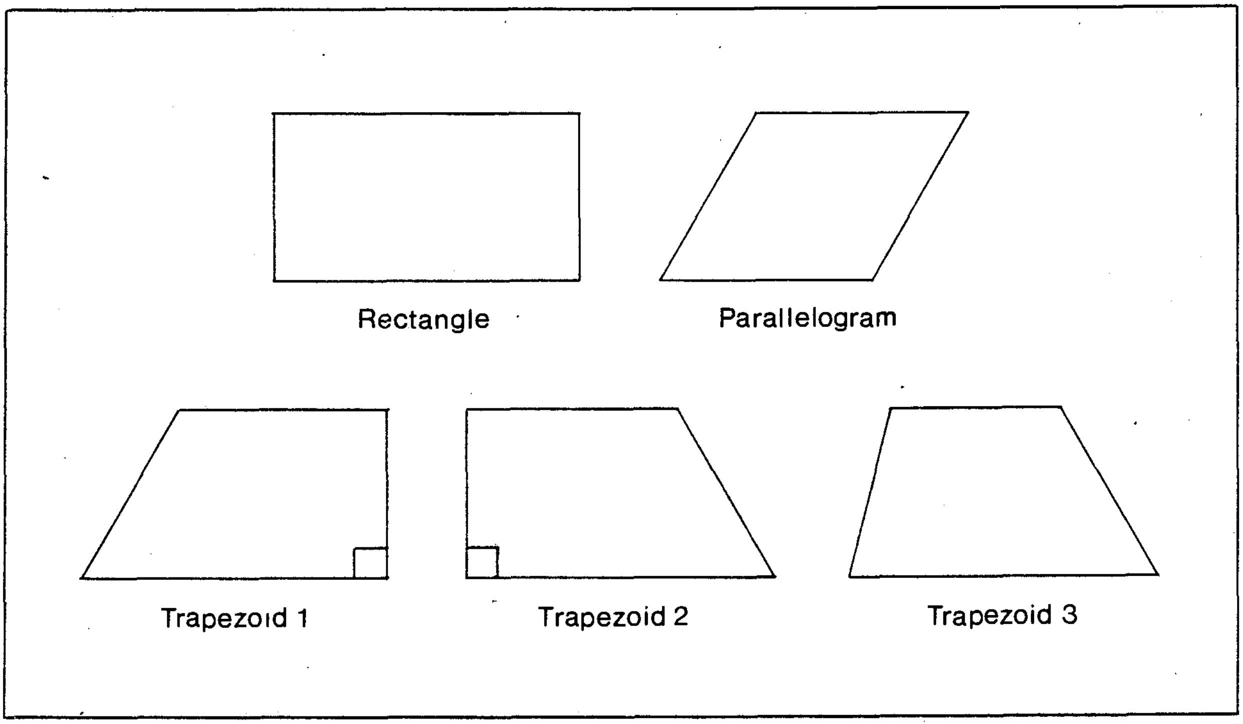

Figure 1-7 Figures Accepted By MEBES Software

(Triangles are represented by a trapezoid with zero width top or base.) The figures used in laying out an integrated circuit are not, however, restricted to these five primitive shapes; polygons abound. This means that *e-beam data preparation software must partition the IC layout polygons into the primitives accepted by the MEBES machine.*

1.3.6.4. Sequencing

On the MEBES machine, the following sequence is followed.

(1) All of the data for one stripe is rasterized and saved in the bit map.

(2) The stage is positioned to the first instance of the current stripe, and the stripe is written.

(3) The stage is moved to the next instance of the current stripe, and the stripe is written.

(4) The above step is repeated until all instances of the current stripe have been written.

(5) Data for the next stripe is rasterized.

(6) All instances of that stripe are written.

(7) *Etc.*

1.4. OVERVIEW OF THESIS

Chapter two discusses algorithms for partitioning of polygons into trapezoids, and describes the implementation of one such algorithm. In chapter three some possible techniques are discussed for optimizing MEBES data. Chapter four describes the development, implementation, and evaluation of an algorithm based on one of the optimizing techniques. Chapter five contains conclusions and recommendations for further work.

Chapter 2

POLYGON PARTITIONING

2.1. INTRODUCTION

As discussed in Chapter One, electron beam lithography systems do not accept general polygons as input data; they accept only a small set of simple geometric figures. The Perkin-Elmer MEBES machine, in particular, accepts five types of figures, all of which can be classified as trapezoids with horizontal top and base. Thus it is necessary to partition all polygons of a mask layout into trapezoids of this type. This polygon partitioning is the process of primary importance, since the rest of the data preparation is really just an exercise in two dimensional graphics.

2.1.1. Objectives

The objectives for this part of the study were

(1) to explore polygon partitioning algorithms, and

(2) to implement the most desirable algorithm in the context of a complete e-beam data preparation program.

2.1.2. Definitions

Before proceeding, it will be useful to define several terms which will be used throughout the remainder of this thesis.

A *self-touching polygon* has non-adjacent edges which touch, but do not cross each other. (See Figure 2-1.)

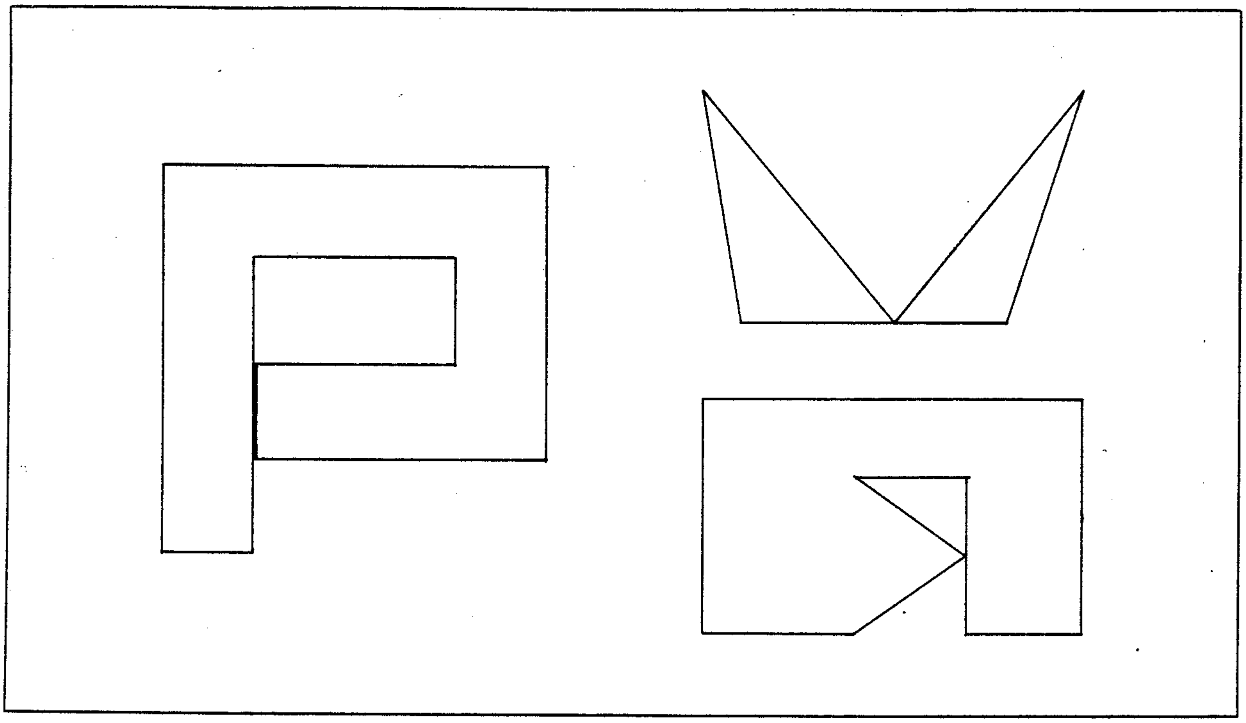

Figure 2-1 Self-Touching Polygons

A *self-crossing polygon* has non-adjacent edges which cross each other. (See Figure 2-2).

A *self-intersecting polygon* is a self-touching or self-crossing polygon.

A *wire* is a geometric primitive of IC layouts; it is most commonly used to electrically connect different points in a circuit (hence the name). A wire is described by its width and center-line, where the center-line is a set of vertices defining the path which the wire follows. An ideal wire is the set of all points within one half-width of the center-line [4]. See Figure 2-3 for an example.

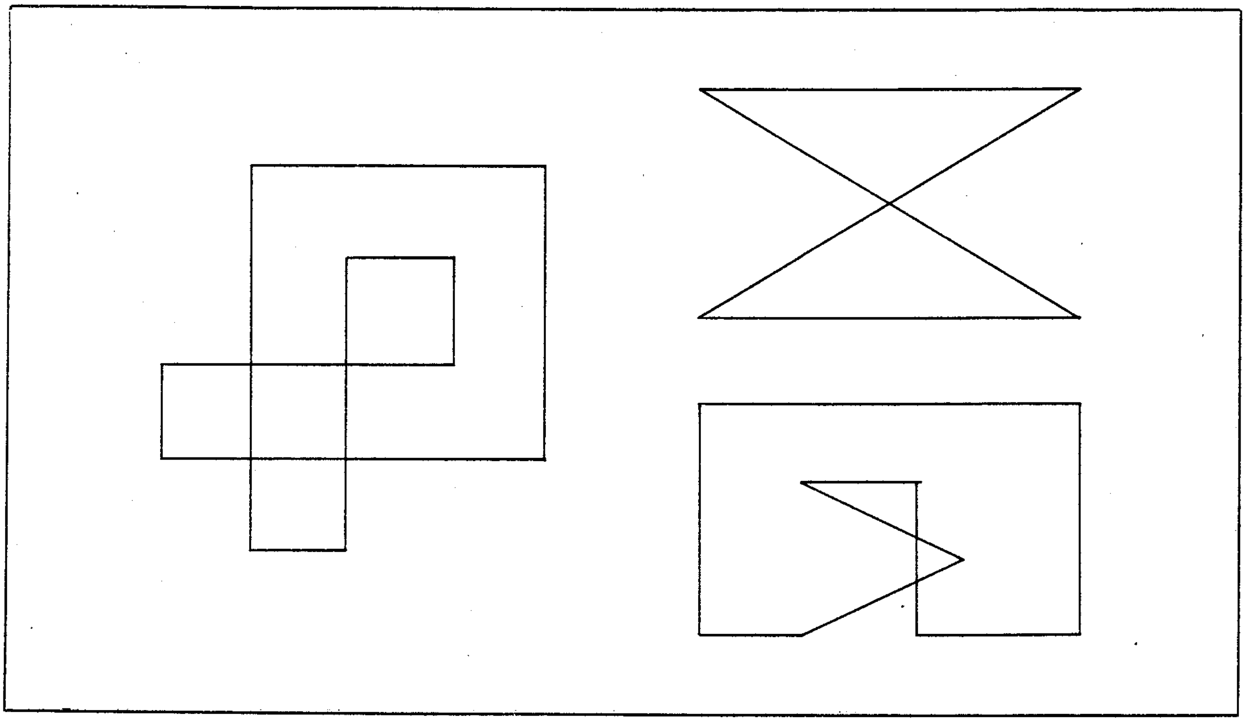

Figure 2-2 Self-Crossing Polygons

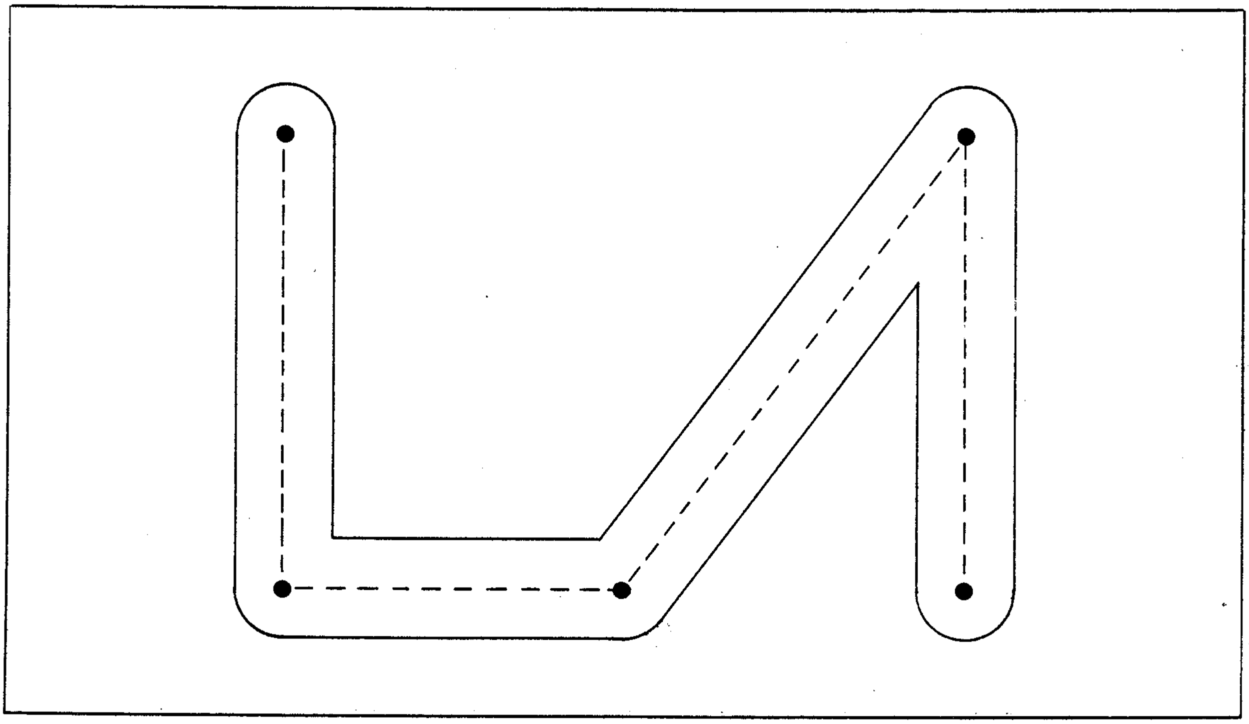

Figure 2-3 Ideal Wire

Note that an ideal wire has rounded corners and a rounded cap on each end. These rounded portions are intended to facilitate connecting wires at arbitrary angles. Because many output devices have difficulty in rendering circular shapes, wires are often simplified to have square ends and no rounded corners. (See Figure 2-4.)

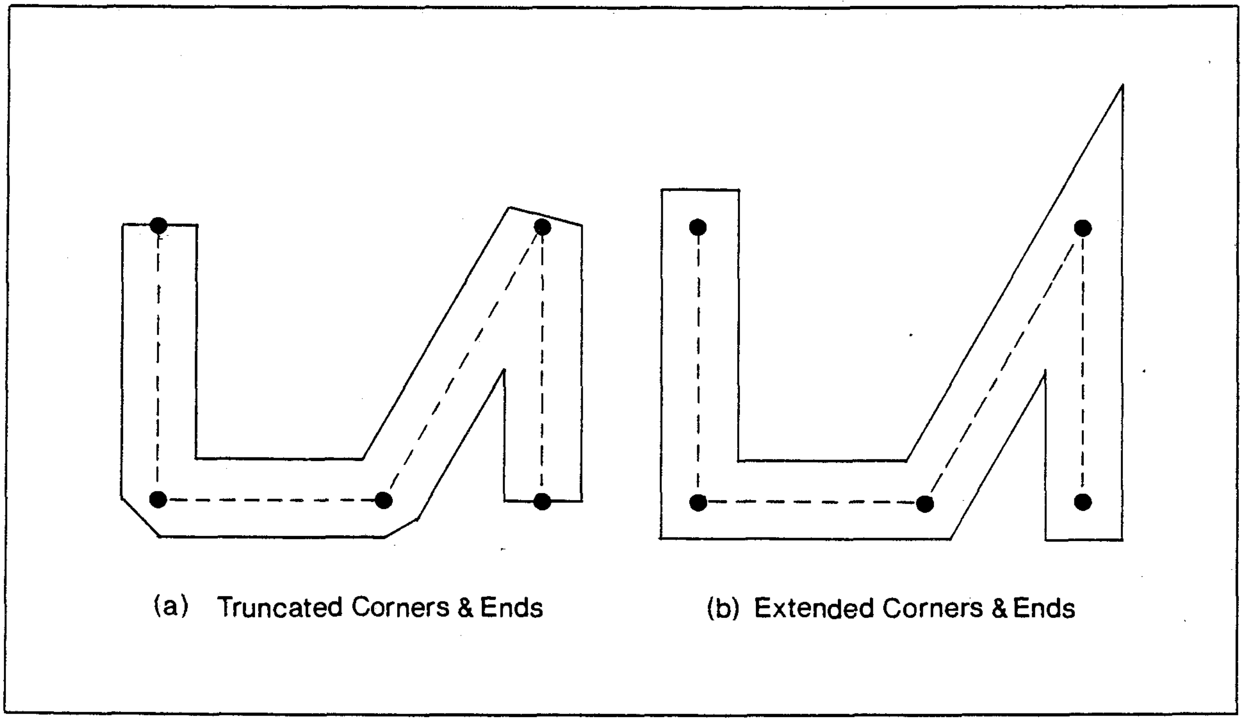

Figure 2-4 Implemented Wires

2.2. PREVIOUS WORK

The constraint that trapezoids have horizontal top and bottom eliminates many polygon partitioning algorithms from consideration. (Many partition polygons into triangles or quadrilaterals, without considering the orientation of the top or bottom.)

One of the earliest algorithms developed for this type of polygon partitioning is described in an M.Sc. thesis by Harris [1]. Although that thesis was not available for examination, an algorithm based on the Harris algorithm is described in a paper by Patel [8].

2.2.1. Patel Algorithm

The Patel algorithm is based on the concept of "uplines" and "downlines". Polygons are classified as "type A" or "type B". A type A polygon encloses an area to be exposed, while a type B polygon encloses an area not to be exposed (*i.e.* a "hole" in a type A polygon). Type B polygons exist only inside of type A polygons. The data for type A polygons must be ordered such that the polygon is traversed in a clockwise direction, while type B polygons must be traversed counterclockwise. All polygons then consist of directed edges. An *upline* is a directed edge which points upward (increasing y coordinate), while a *downline* points downward (decreasing y coordinate).

As described in the paper, the algorithm includes further decomposition of trapezoids into triangles, rectangles and parallelograms for a more primitive e-beam machine. This extra decomposition is not necessary for the MEBES machine, and will not be covered here. The (paraphrased) algorithm is as follows. (Refer to Figure 2-5).

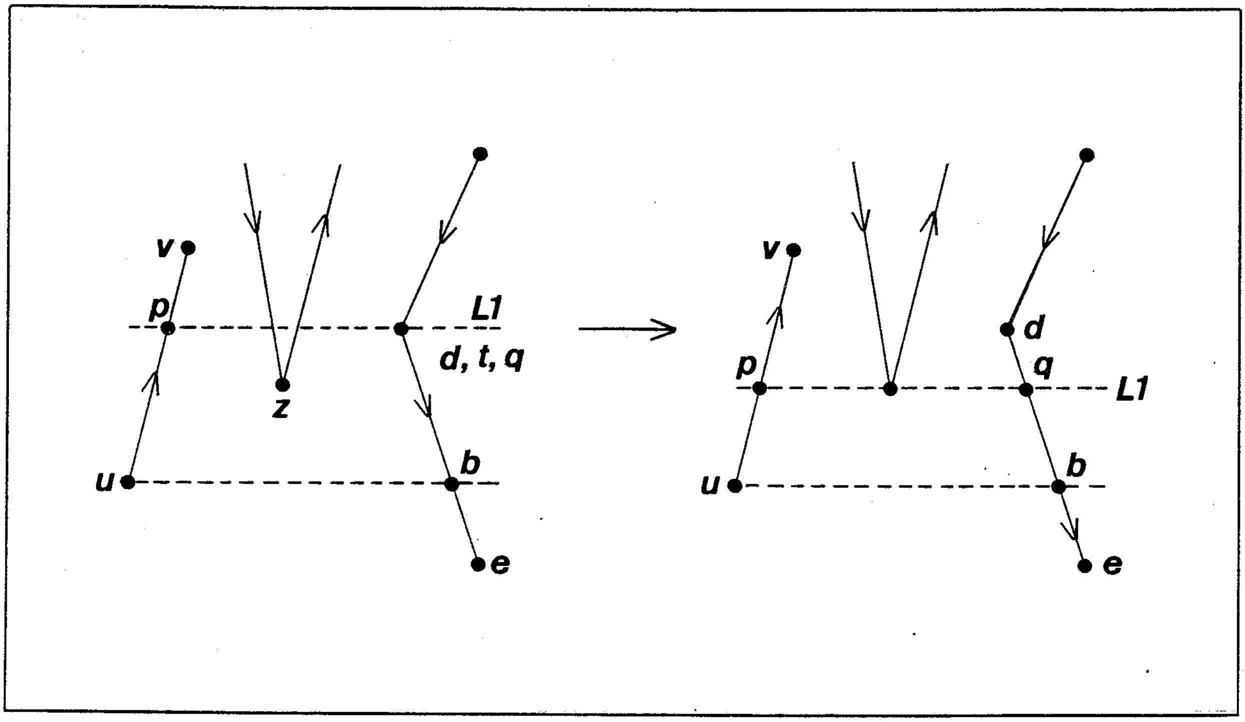

Figure 2-5 Patel Algorithm

```
find uplines and downlines;
for each upline do
begin
    $u \leftarrow$ vertex at upline's lower end;
    $v \leftarrow$ vertex at upline's upper end;
    while $u \neq v$ do
    begin
        find nearest downline;
        /*
          Presumably the nearest downline is the
          nearest one which intersects a horizontal
          ray extending to the right from $u$.
        */
        $d \leftarrow$vertex at downline's upper end;
        $e \leftarrow$vertex at downline's lower end;
        $t \leftarrow v$ or $d$ (whichever is lower);
        $b \leftarrow$ intersection of $de$ with horizontal line
        through $u$;
        L1 $\leftarrow$ horizontal line through $t$ ;
        $p \leftarrow$ intersection of $uv$ with L1;
        $q \leftarrow$ intersection of $de$ with L1;
        $z \leftarrow$ a vertex enclosed by trapezoid $upqb$;
        while $z$ exists do
        begin
            L1 $\leftarrow$ horizontal line through $z$ ;
            $p \leftarrow$ intersection of $uv$ with L1;
            $q \leftarrow$ intersection of $de$ with L1;
            $z \leftarrow$ a vertex enclosed by trapezoid $upqb$;
        end;
        output trapezoid $upqb$;
        $u \leftarrow p$;
    end;
end;
```

It should be pointed out that the statement of the Patel algorithm in [8] contains an error: the check for a vertex enclosed by the tentative trapezoid is performed only once, whereas it must be repeated until no such vertex is found. (This has been corrected in the statement shown here.)

For an example of the operation of the Patel algorithm, consider polygon *ADEFGHIL* shown in Figure 2-6. (The reader is invited to copy the diagram and work through the example by hand, relabeling lines and vertices as they are changed by the algorithm, and shading in trapezoids as they are generated. This suggestion also applies to subsequent examples.)

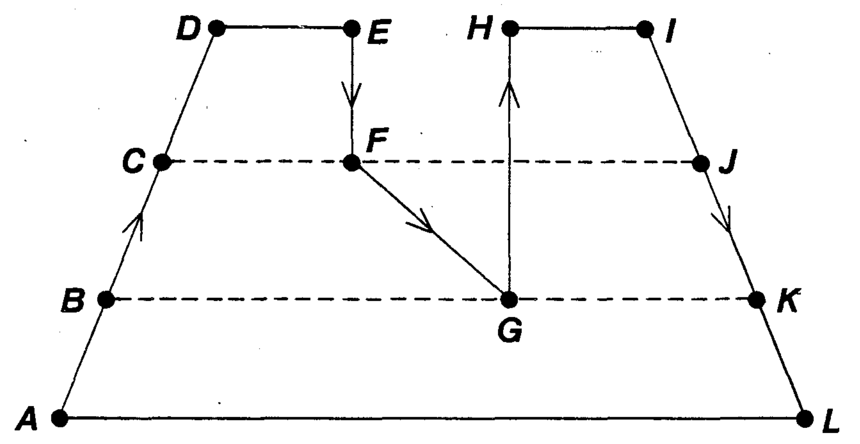

Figure 2-6 Example For Patel Algorithm

The uplines are identified as *AD* and *GH*; the downlines are *EF*, *FG* and *IL*. Upline *AD* is selected. (The order in which uplines are processed is arbitrary; we could start with *GH* and get the same results.) The nearest downline is *IL*. Tentative trapezoid *ADIL* encloses vertex *F*. Tentative trapezoid *ACJL* encloses vertex *G*. Trapezoid *ABKL* encloses no vertex, and is output. The upline now becomes *BD*. The nearest downline is *FG*. Trapezoid *BCFG* contains no vertex, and is output. The upline becomes *CD*. The nearest downline is *EF*. Trapezoid *CDEF* encloses no vertex, and is output. This completes the processing of upline *AD*. Upline *GH* is selected. The nearest downline is *IL*. Trapezoid *GHIK* encloses no vertex, and is output. There are no more uplines to be processed, so this completes the partitioning.

2.2.2. Little & Heuft Algorithm

Little and Heuft [3] describe an algorithm to be used for decomposing polygons into component trapezoids. Although the intended purpose of the trapezoids was to drive a raster graphics display controller, they could just as easily be used for preparing data for the MEBES machine. The algorithm makes use of vertex adjacency information (*i.e.* which vertices are adjacent to each other). This contrasts with other algorithms, which discard this information.

A polygon is represented as a doubly linked list structure. Each node in the list stores the data for one vertex; adjacent nodes store adjacent vertices. The Little & Heuft algorithm is as follows.

```
while polygon vertex list is not empty do
begin
    /* Find top of trapezoid. */
    $a \leftarrow$ vertex with greatest y coordinate;
    if there is a vertex adjacent to $a$ with the same
    y coordinate
    then $b \leftarrow$ that vertex;
    else $b \leftarrow a$;
    /* Find sides. */
    $c \leftarrow$ vertex adjacent to $b$ such that $c \neq a$;
    $d \leftarrow$ vertex adjacent to $a$ such that $d \neq b$;
    /* Find bottom. */
    if the region enclosed by $abcd$ contains any other vertices
    then $e \leftarrow$ the contained vertex with largest y coordinate;
    else $e \leftarrow (-\infty,-\infty)$;
    ybtm $\leftarrow$ max(y($c$),y($d$),y($e$));
    /* y($v$) is the y coordinate of vertex $v$. */
    $f \leftarrow$ intersection of $bc$ with line y = ybtm;
    $g \leftarrow$ intersection of $ad$ with line y = ybtm;
    remove component trapezoid $abfg$ from polygon vertex list;
end;
```

For example, consider application of the algorithm to the polygon in Figure 2-7.

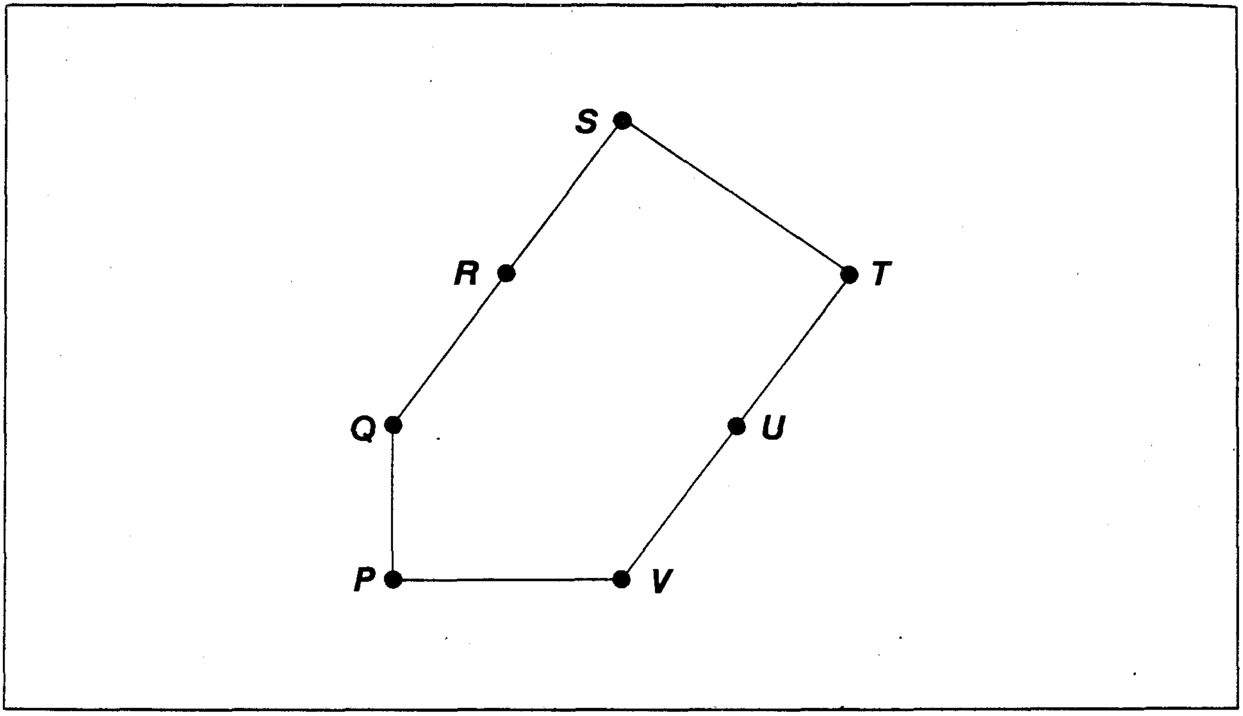

Figure 2-7 Example For Little & Heuft Algorithm

S is identified as the top of the first trapezoid. Region *STQ* contains no other vertex. *R*, the intersection of edge *QS* with line y = y(*T*), is computed. Trapezoid *STR* is removed from the polygon vertex list and output. *RT* becomes the top of the next trapezoid. *RTVQ* contains no other vertex. *U*, the intersection of edge *TV* with line y = y(*Q*), is computed. Trapezoid *RTUQ* is removed. *QU* becomes the top of the next trapezoid. *QUVP* contains no other vertex. No new intersection point is introduced this time. Trapezoid *QUVP* is removed. The polygon vertex list is now empty, and the process is complete.

2.2.3. Newell & Sequin Algorithm

A polygon partitioning algorithm based on scan conversion is described by Newell and Sequin [6]. Their algorithm utilizes a method to determine if a point is inside a polygon, using the "nonzero winding number" convention. Suppose we want to know if point *p* is inside or outside of a polygon. The ***winding number*** of the polygon boundary with respect to *p* is defined as the net number of times that a point (*b*) on the boundary wraps around *p* while *b* makes one complete traversal of the boundary. The *nonzero winding number* convention states that a point is inside a polygon if the winding number of the polygon boundary with respect to that point is nonzero. For example, consider point *P* in Figure 2-8.

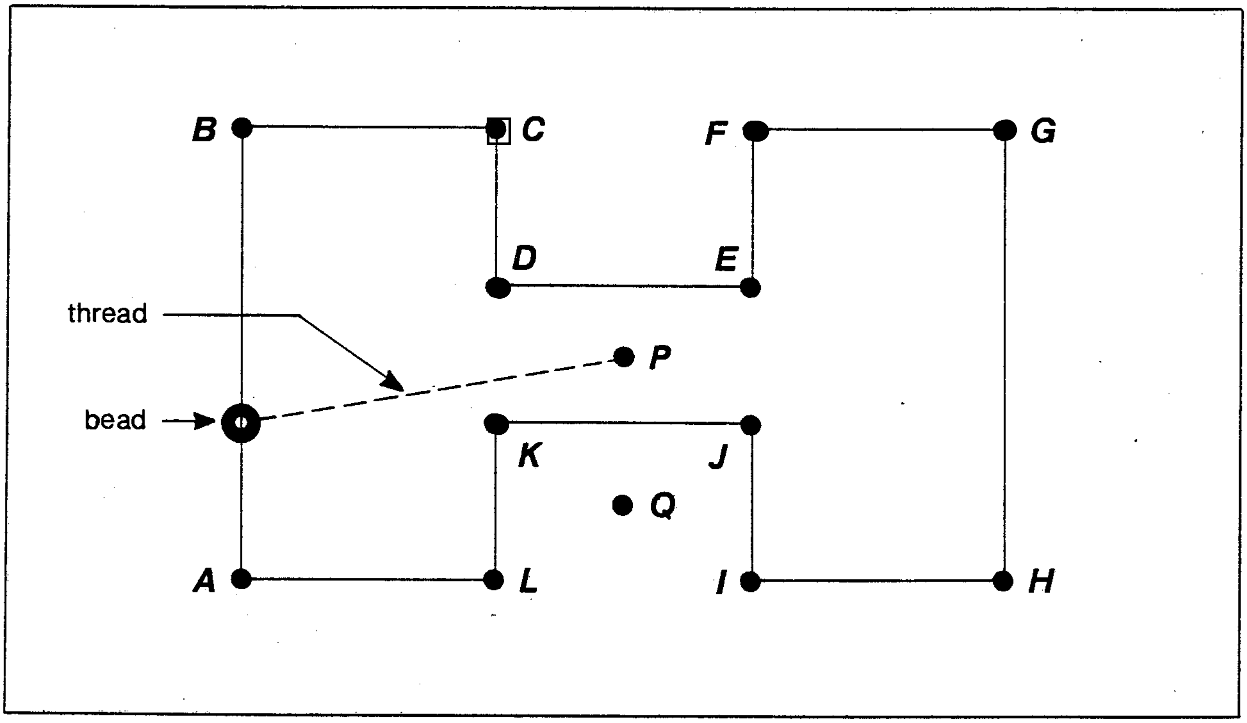

Figure 2-8 Winding Number

Imagine that a loop of stiff wire is bent into the shape of polygon *ABCDEFGHIJKL*. The wire is placed on a board with a peg mounted at point *P*. A piece of thread is anchored to the side of the peg; its other end is attached to a bead which has been threaded on the wire. Moving the bead around the wire loop causes the thread to wind around the peg. Suppose we start with the bead at vertex *A*. By the time the bead has reached vertex *G*, there is half a turn of thread around the peg; by the time the bead has returned to *A*, the peg has one complete turn of thread. The number of turns of thread around the peg represents the winding number of the boundary with respect to point *P*. Thus the winding number for *P* is nonzero, and *P* is deduced to be inside the polygon. Now consider the same exercise for point *Q*. Again, start with the bead at *A*. By the time the bead has reached vertex *I*, there is approximately three quarters of a turn of thread around the peg. But, by the time the bead has returned to *A*, the thread has been unwound, leaving zero turns on the peg. The winding number with respect to Q is zero, and Q is deduced to be outside of the polygon.

The nonzero winding number convention is quite general. The same cannot be said for all interior/exterior testing methods. For example, consider the commonly used "parity convention". A ray is constructed from the test point, extending to infinity in any direction. Intersections of the ray with polygon edges are counted, and the parity of the count determines the state of the point: if the count is even, then the point is outside, else it is inside. This method works in many cases, but fails with some self-crossing polygons. For example, any point in the shaded area of Figure 2-9 will be classified as outside of the polygon. This is the wrong answer for polygons being used to create masks!

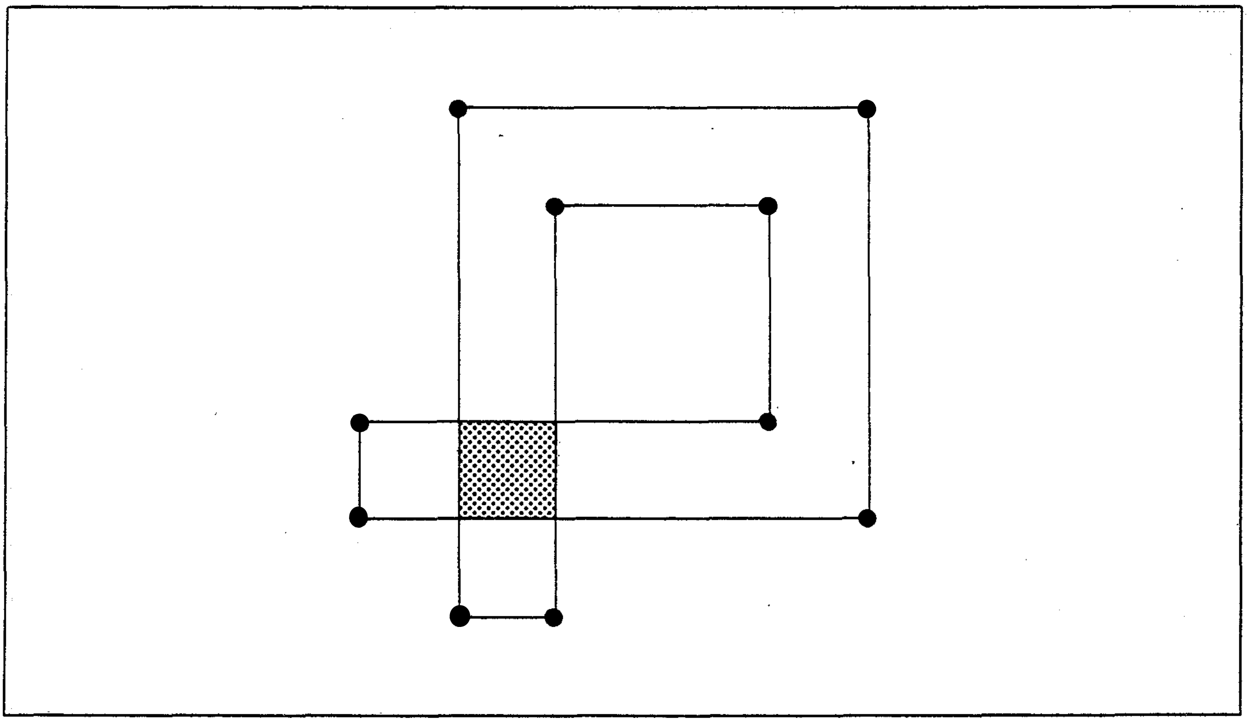

Figure 2-9 Polygon Which Foils Parity Convention

The nonzero winding number convention handles self-crossing polygons without any such difficulties. (The reader is encouraged to visualize the "peg and thread" exercise for this example.)

Newell and Sequin describe a scan conversion algorithm using the nonzero winding number convention to determine which pixels on the current scan line should be turned on. Each polygon edge is assigned a direction tag: +1 for edges which are traversed in an upward (increasing y coordinate) direction; —1 for downward directed edges; and 0 for horizontal edges. If the direction tags are summed for all edges which intersect the scan line to the left of some point *p* on the scan line, then the total is equal to the winding number of the boundary with respect to *p*. For example, consider the polygon in Figure 2-10.

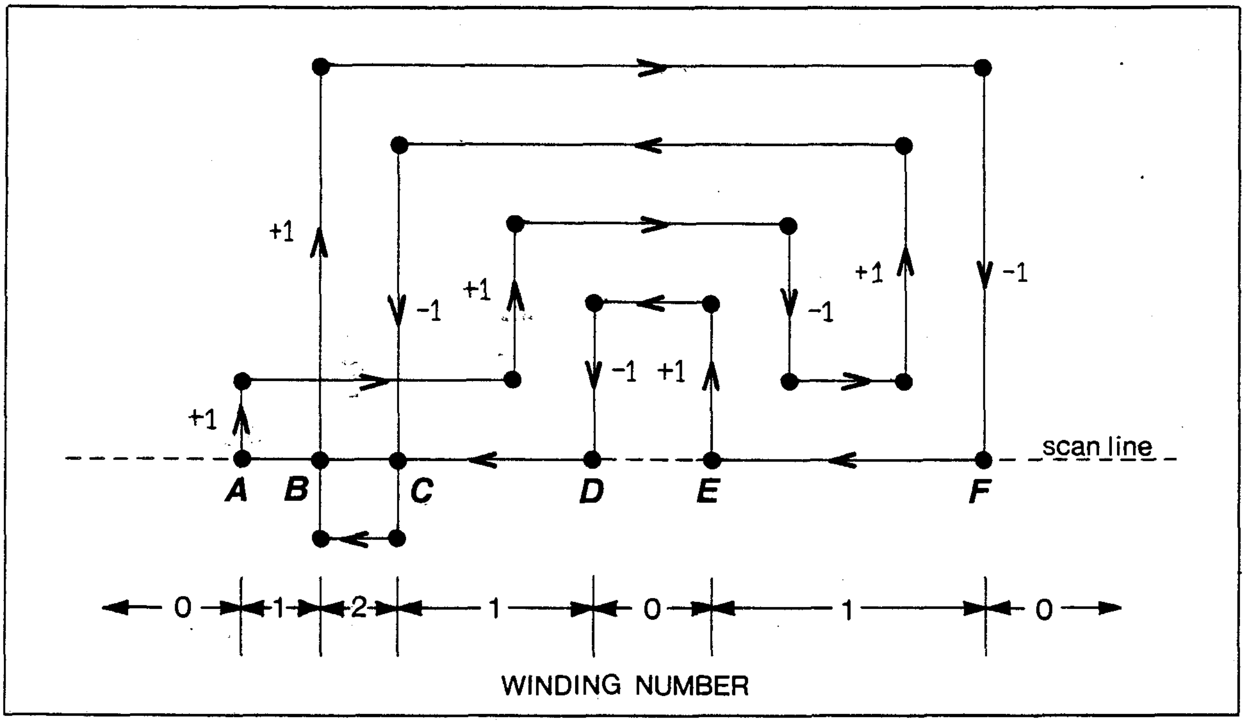

Figure 2-10 Calculation of Winding Number from Direction Tags

The winding number for all points on the scan line to the left of *A* is 0. All points between *A* and *B* have a winding number of 1. Between *B* and *C*, all points have winding number equal to 2. And so forth; winding numbers for points in each region of the scan line are labeled on the diagram. If one chooses any point on the scan line and adds the direction tags for all edges which intersect the scan line to the left of the point, the sum will be equal to the winding number for that point. This is the technique used in the scan conversion algorithm.

Newell and Sequin use a modification of their scan conversion algorithm to perform polygon partitioning. The current y value is stepped from one point of interest to the next, rather than from one raster scan line to the next. The winding number is used as follows. A sum (*wind*) is initialized to zero. As a horizontal line through the current y value (line y = ycurr) is scanned from left to right, the direction tag for each edge encountered is added to *wind*. In this way, as each edge is encountered, *wind* becomes equal to the winding number for points which are on line y = ycurr, to the right of the edge, and to the left of the next edge (Figure 2-10). Thus, as the scan is performed, a change in *wind* from zero to nonzero signals the left side of a trapezoid, and a change from nonzero to zero indicates the right side. The bottom of a trapezoid is a horizontal line with y = y coordinate at which the side edges first become paired. The top of a trapezoid is a horizontal line with y = y coordinate at which the side edges become unpaired. To use this algorithm to generate data for the MEBES machine, it must be modified to partition trapezoids at stripe boundaries. The algorithm, so modified, is as follows.

NEWELL & SEQUIN ALGORITHM

```
generate edge list, omitting horizontal edges and those
outside stripe, and assigning direction tag to each edge;
sort edge list on ymin(edge);
/* ymin(z) = y coordinate of lowermost point in z */
while active list or edge list is not empty do
begin
    if active list is empty then do
    begin
        ycurr ← ymin(first edge in edge list);
        if ycurr < ymin(stripe)
        then ycurr ← ymin(stripe);
    end;
    edge ← first edge in edge list;
    while ymin(edge) ⩽ ycurr do
    begin
        transfer edge from edge list to active list;
        xcurr(edge) ← x(edge,ycurr);
        /*
            xcurr(edge) = x coordinate of
            intersection of edge with line y = ycurr
            x(edge,ycurr) is the x coordinate of point
            on edge with y coordinate = ycurr
        */
        edge ← next edge in edge list;
    end;
    sort active list on xcurr(edge);
    find ynext; /* See below. */
    generate trapezoids; /* See below. */
    for each edge in active list do
    begin
        /*
          ymax(z) = y coordinate of uppermost point in z
        */
        if ymax(edge) = ynext or ynext = ymax(stripe) then do
        begin
            if edge has a mate then output trapezoid
            and unpair edge's mate;
            remove edge from active list;
        end;
        else xcurr(edge) ← xnext(edge);
        /*
            xnext(edge) = x coordinate of intersection
            of edge with line y = ynext
        */
    end;
    ycurr ← ynext;
end;
```

```
"FIND YNEXT" SEQUENCE
ynext ← min(active list ymax values);
ynext ← min(ynext,ymin(first edge in edge list));
ynext ← min(ynext,ymax(stripe));
for each edge in active list do
    xnext(edge) ← x(edge,ynext);
sort active list on xnext(edge);
if active list order at ycurr ≠ order at ynext
then do
begin
    find intersection point of edges out of sort;
    break up 2 intersecting edges into 4 new edges,
    leaving lower 2 in active list and inserting
    upper 2 into edge list;
    ynext ← y(intersection point);
    go back to loop which assigns xnext values;
end;
```

```
"GENERATE TRAPEZOIDS" SEQUENCE
wind ← 0;
left ← first edge in active list;
for each edge in active list do
begin
    wind ← wind + direction(edge);
    /*
      direction(edge) = direction tag assigned to edge
      wind = winding number for points to immediate right of edge
    */
    if wind = 0 then do
    begin
        if left and edge are not mates then do
        begin
            if left has a mate then output trapezoid
            and unpair left's mate;
            if edge has a mate then output trapezoid
            and unpair edge's mate;
            pair left and edge and initialize a
            trapezoid;
        end;
        left ← next edge in active list;
    end;
end;
```

For an example of the functioning of the Newell & Sequin algorithm, refer to Figure 2-11.

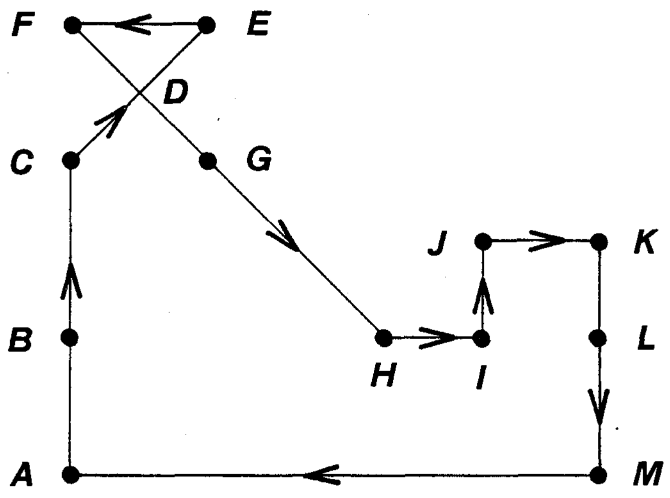

Figure 2-11 Example For Newell & Sequin Algorithm

We are given polygon *ACEFHIJKM*, defined in a clockwise direction. (It could be defined counterclockwise with no detrimental effect on the algorithm.) Note that vertex *D* is not part of the initial polygon; edges *CE* and *FH* simply cross at that point. Edges *AC, CE* and *IJ* receive direction tags of +1; *FH* and *KM* receive direction tags of —1; *EF, HI, JK* and *MA* receive tags of 0. Assume that the polygon lies entirely within a stripe, so no partitioning at stripe boundaries will be performed. Ycurr becomes y(*A*). Edges *AC* and *KM* are transferred from the edge list to the active list. Ynext becomes y(*H*). Edges *AC* and *KM* are paired. Ycurr becomes y(*H*). Edges *FH* and *IJ* are transferred from the edge list to the active list. Ynext becomes y(*J*). Edges *AC* and *FH* should now be paired, but *AC* already has a mate, so trapezoid *ABLM* is output. Edges *AC* and *KM* are unpaired. *AC* is now paired with *FH*, and *IJ* is paired with *KM*. Because ymax(*IJ*) = ynext, this edge will no longer be active once ycurr is stepped, so it must be removed from the active list. First, though, since *IJ* has a mate, trapezoid *IJKL* is output, and *KM* is unpaired. *IJ* is then removed from the active list. *KM* is also removed from the active list because ymax(*KM*) = ynext. Ycurr becomes y(*J*). No edges are transferred from the edge list to the active list at this point. Ynext becomes y(*C*). *AC* and *FH* should be paired, but are found to be already paired, so no action is necessary. Because ymax(*AC*) = ynext, trapezoid *BCGH* is output and edges *AC* and *FH* are unpaired; *AC* is then removed from the active list. Ycurr becomes y(*C*). Edge *CE* is transferred from the edge list to the active list, leaving the edge list empty. Ynext is set to y(*F*), but the active list order at this value of ynext does not match the order at ycurr. Thus the intersection point *D* is found; edge *CE* in the active list is replaced by edge *CD*, and *DE* is inserted into the edge list. Similarly, *FH* in the active list is replaced by *DH*, and *FD* is inserted into the edge list. Ynext becomes y(*D*). Now the active list order is the same at ycurr and ynext. Edges *CD* and *DH* are paired. Then, because ymax(*CD*) = ynext, trapezoid *CDG* is output, and *CD* is removed from the active list. *DH* is also removed. Ycurr becomes y(*D*) Edges *DE* and *FD* are transferred from the edge list to the active list, leaving the edge list empty Ynext becomes y(*F*). Edges *FD* and *DE* are paired. Ymax(*FD*) = ynext, so trapezoid *DEF* is output, and *FD* is removed from the active list. *DE* is also removed, leaving the active list empty. Now both edge list and active list are empty, and the process terminates.

2.2.4. Otto Algorithm

A paper by Otto [7] mentions the reduction of polygons to e-beam primitives using a recursive divide and conquer strategy. The polygon is sliced horizontally or vertically from a reference vertex, producing two new polygons. The procedure is recursively applied to each of the new polygons, until all polygons have been reduced to e-beam primitives.

An approach such as this could potentially lead to an **O** (n log n) algorithm. (Finding a slicing line might require **O** (n) time. If each partition generates two polygons with equal numbers of vertices, then recursion would proceed to a depth of $\log_2 n$ levels.) Unfortunately, the description of the algorithm was too sketchy to permit any kind of evaluation.

2.3. ANALYSIS

2.3.1. Patel Algorithm

The Patel algorithm works satisfactorily for most polygons, but breaks down when applied to self-crossing polygons. Consider the polygon in Figure 2-12.

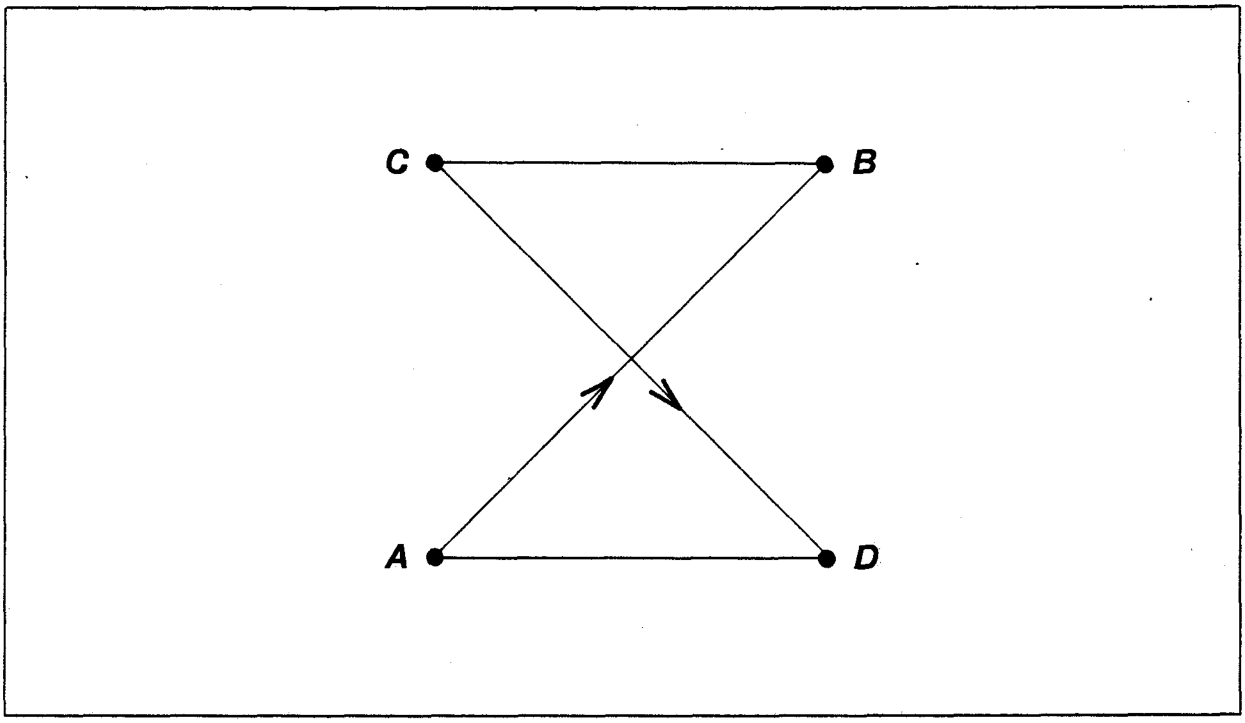

Figure 2-12 Self-Crossing Polygon

When applied to this polygon, the Patel algorithm fails to detect the crossing point, and outputs an invalid trapezoid. This is not a serious drawback for this type of polygon: the internal area is reduced to zero at the crossing point, and the polygon would be invalid in an IC layout. Unfortunately, the Patel algorithm also fails with self-crossing polygons which originate as wires with self-touching center-lines, such as that shown in Figure 2-13.

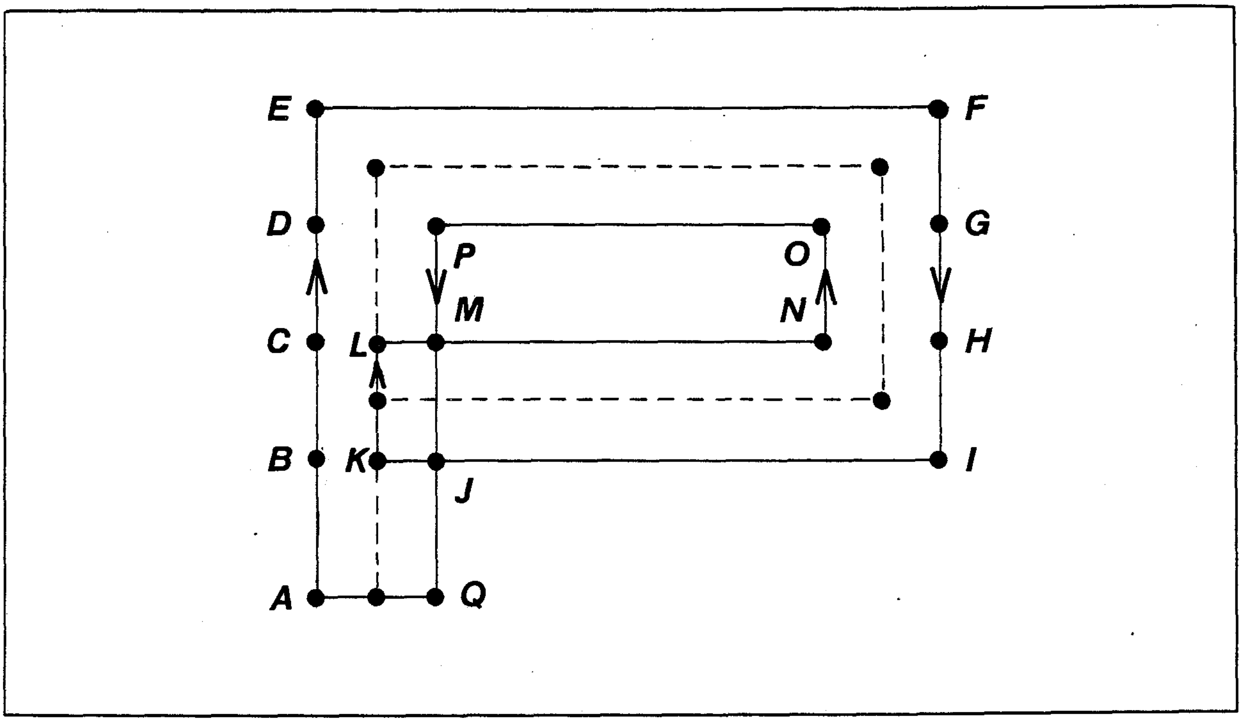

Figure 2-13 Wire With Self-Touching Centerline

In this example, points *B, C, D, G, H, J* and *M* are not part of the original polygon. The Patel algorithm outputs the following trapezoids: *ABJQ, BCMJ, CDPM, DEFG, KLMJ,* and *NOGH*. No data is output to cover region *JMHI*. Because this type of polygon is valid in IC design, the Patel algorithm is unacceptable without substantial refinement.

2.3.2. Little & Heuft Algorithm

The Little & Heuft algorithm gets into serious trouble when applied to a polygon with a concave portion at the bottom, such as shown in Figure 2-14.

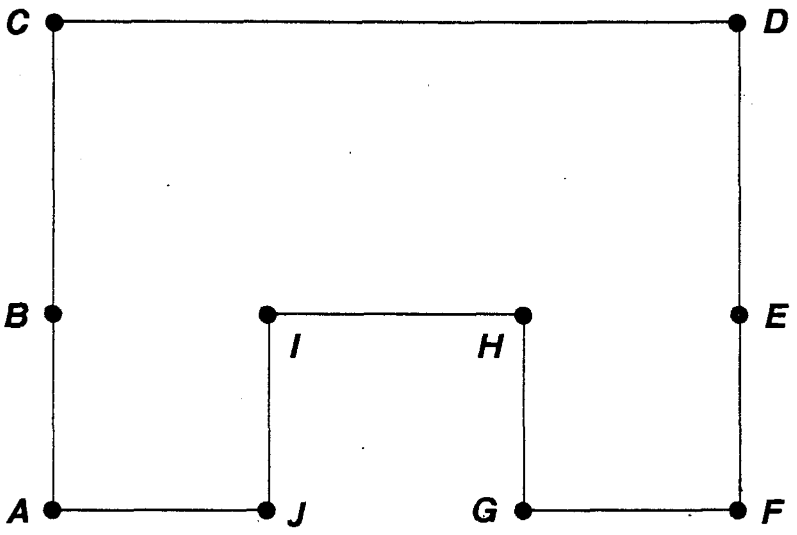

Figure 2-14 Polygon With Concave Bottom

Tracing the operation on this example yields the following. *CD* is identified as the top of the first trapezoid. Region *CDFA* contains vertices *I* and *H*, and ybtm is set to y(*I*). Trapezoid *CDEB* is removed and output. If *IH* is selected as the top of the next trapezoid, then region *IHGJ* is found to contain no other vertices, and a trapezoid is generated for that region. If *BE* is selected as the top of the next trapezoid, then region *BEFA* contains no other vertices, and trapezoid *BEFA* is generated. In either case, the region bounded by *I*, *H*, *G* and *J* becomes filled in. The algorithm is clearly inadequate as stated, and needs to be equipped with the ability to recognize when the polygon has been split horizontally by a concavity.

2.3.3. Newell & Sequin Algorithm

The Newell & Sequin algorithm showed the greatest promise; it was designed to work with arbitrary polygons, including all types of self-crossing polygons. It is not, however, without problems.

One problem arises when attempting to sort the active list on xcurr. If two edges have the same xcurr value, then the active list at ycurr cannot be sorted. A similar problem exists for sorting the active list on xnext. It is essential that the active list be sorted correctly because the determination of winding number relies on this sorting. Also, an incorrectly sorted active list will prevent the test for edges which cross each other ("find ynext" sequence) from working. For example, see Figure 2-15.

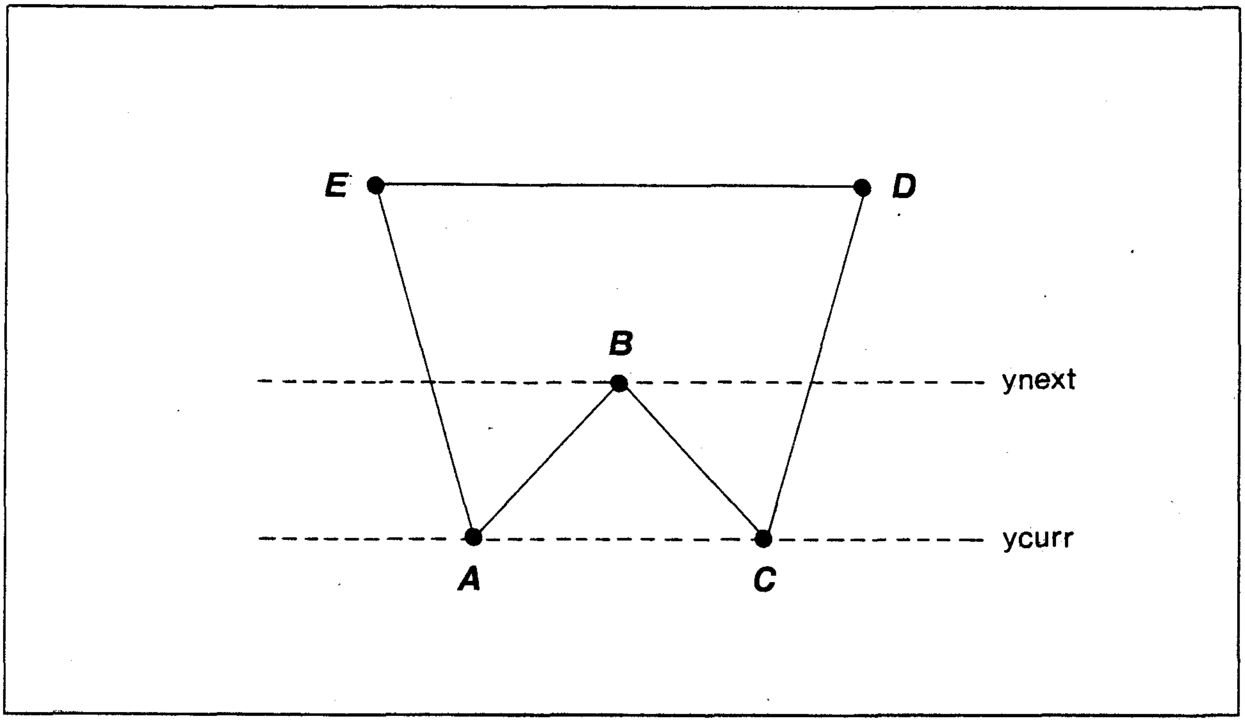

Figure 2-15 Unsortable Edges

When ycurr = y(A), edges EA and AB have the same xcurr value, and edges AB and BC have the same xnext value.

A second problem arises in the test for crossing edges. If two edges cross at y = y(p), where p is some other vertex of the polygon, then the crossing will not be detected. For example, consider Figure 2-16.

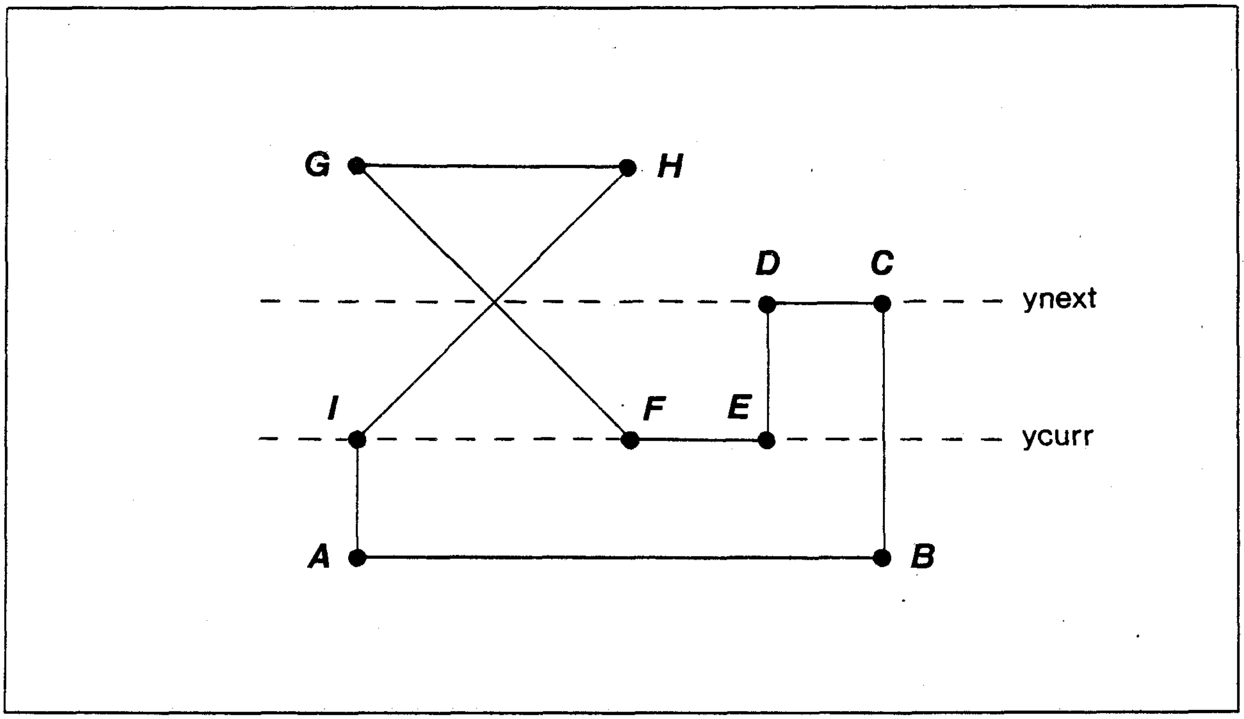

Figure 2-16 Crossing Point Not Detected

When ycurr = y(*I*), the active list contains edges *HI*, *FG*, *DE* and *BC*. Ynext is initially set to y(*D*). Edges *HI* and *FG* will have the same xnext value. Assuming that the active list ordering problem is corrected, the order at ycurr will match the order at ynext, and ynext = y(*D*) is accepted. Edge *HI* will be paired with edge *FG*. When ycurr becomes y(*D*), the active list contains edges *HI* and *FG*. Ynext is tentatively set to y(*G*). The active list order at ycurr and ynext agrees, and ynext = y(*G*) is accepted. Edges *HI* and *FG* remain paired. Before removing these edges from the active list, trapezoid *IFHG* is output. Thus the trapezoids generated from this polygon are as shown in Figure 2-17. Clearly, this is not what was intended!

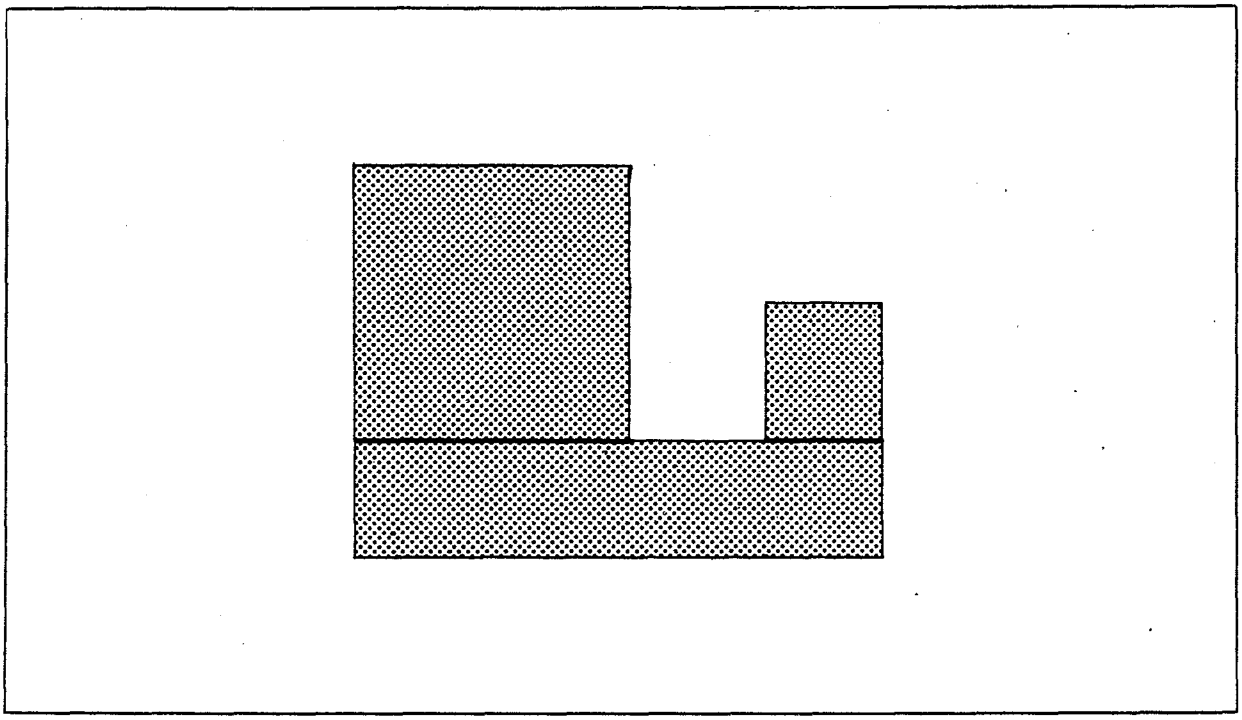

Figure 2-17 Trapezoids Generated

2.3.4. Refinement of Newell & Sequin Algorithm

2.3.4.1. Sorting the Active List

If two edges have the same value of xcurr, then they must intersect at y = ycurr, and they cannot possibly cross each other between ycurr and ynext. Then their order at ynext can be used to resolve the order at ycurr. Similarly, edges which have the same xnext value can be sorted at ynext by using their order at ycurr. This approach would require delaying sorting of the active list at ycurr until the order at ynext is known.

A cleaner solution is to use the cotangent of the angle formed by the line y = ycurr and the edges to be sorted. A justification for this follows. Consider a number of active list edges which intersect at the point (xcurr,ycurr). All such edges must extend above ycurr, otherwise they would have been removed from the active list. We are not concerned with any portions of these edges below ycurr, because such portions have already been processed. So, consider a set of edges (or partial edges) which intersect at the point (xcurr,ycurr), and which extend upwards from that point (Figure 2-18).

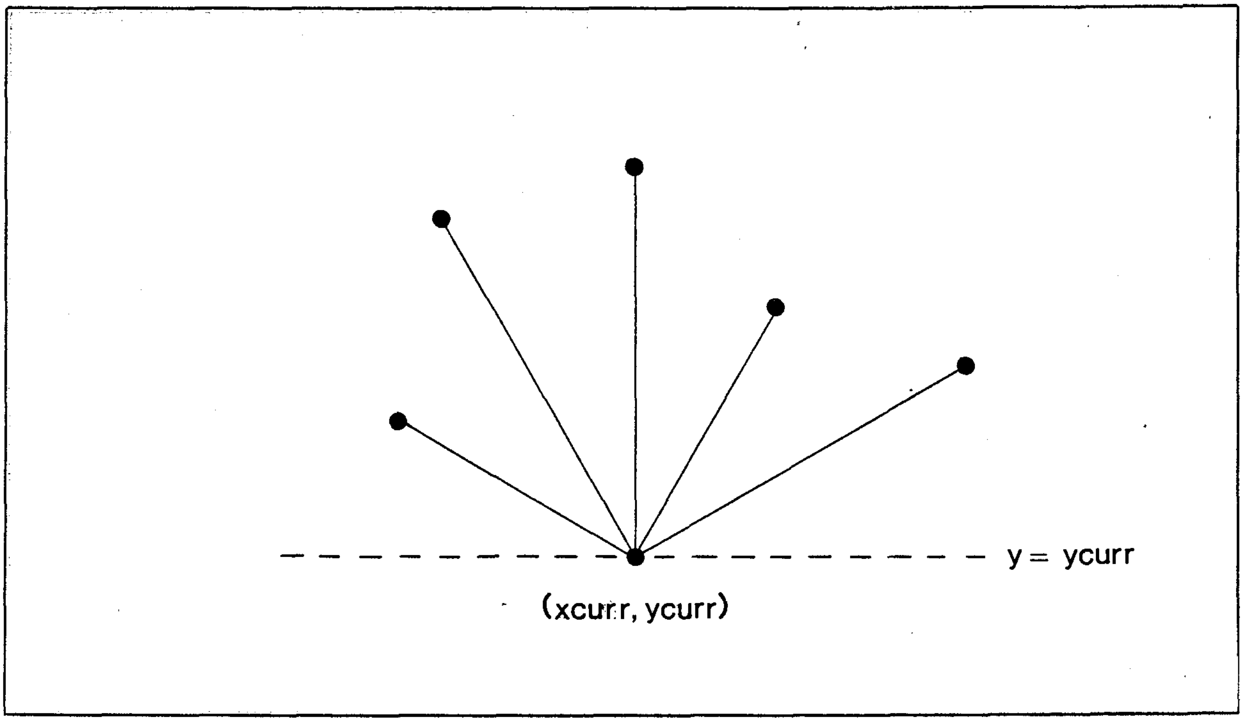

Figure 2-18 Edges With Common xcurr Value

Define ray "ref" as the ray with endpoint (xcurr,ycurr) and equation y = ycurr, and which is directed to the right of (xcurr,ycurr). For each edge, define θ to be the angle formed by ref and the edge, measured in the conventional sense (counterclockwise) (Figure 2-19). θ can be used to sort these edges; the edge with largest θ should appear first in the active list, and the one with smallest θ should appear last. *I.e.* edges with equal xcurr value can be sorted on decreasing θ. This solution would work, but would require considerable machine time to compute the angles from the edge data.

Fortunately, there is a trigonometric function which is easy to compute, and which works just as well as θ to sort the edges: cotangent. (The cotangent of the angle between horizontal line $y = y_1$ and a line segment with end points (x_1,y_1) and (x_2,y_2) is just $(x_2 - x_1) / (y_2 - y_1)$;

see Figure 2-20.)

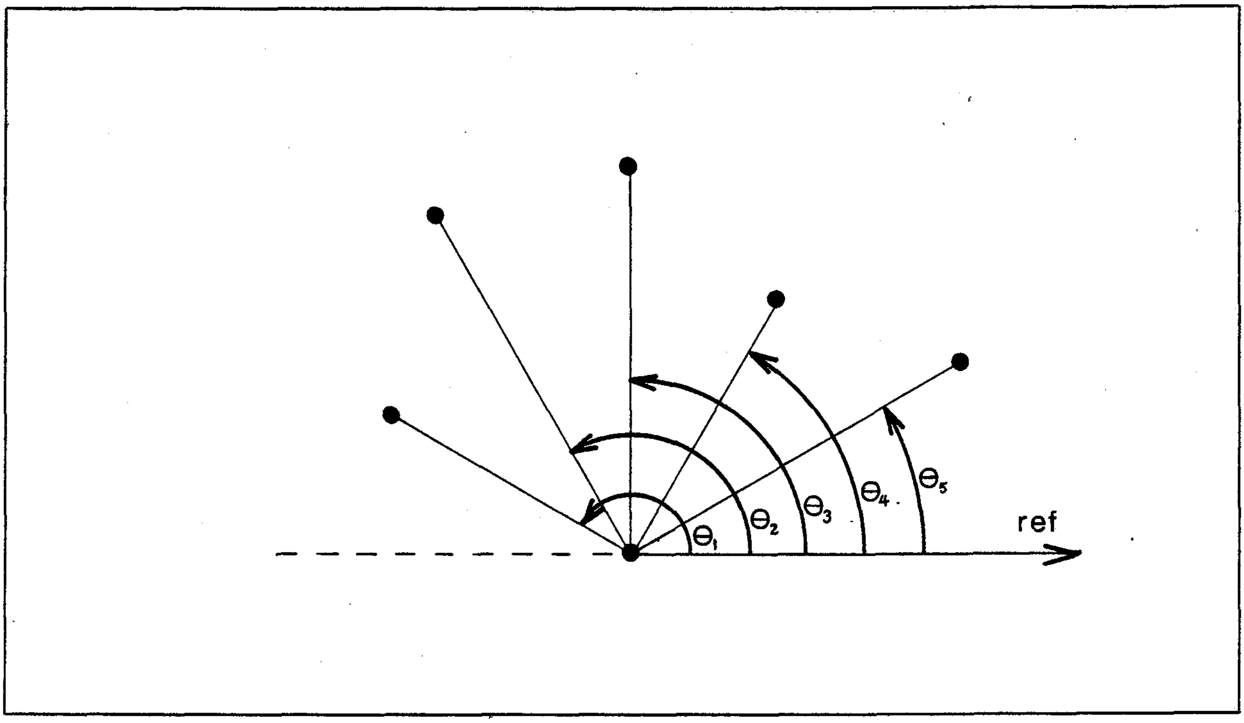

Figure 2-19 Ordering Of Edges By Angle

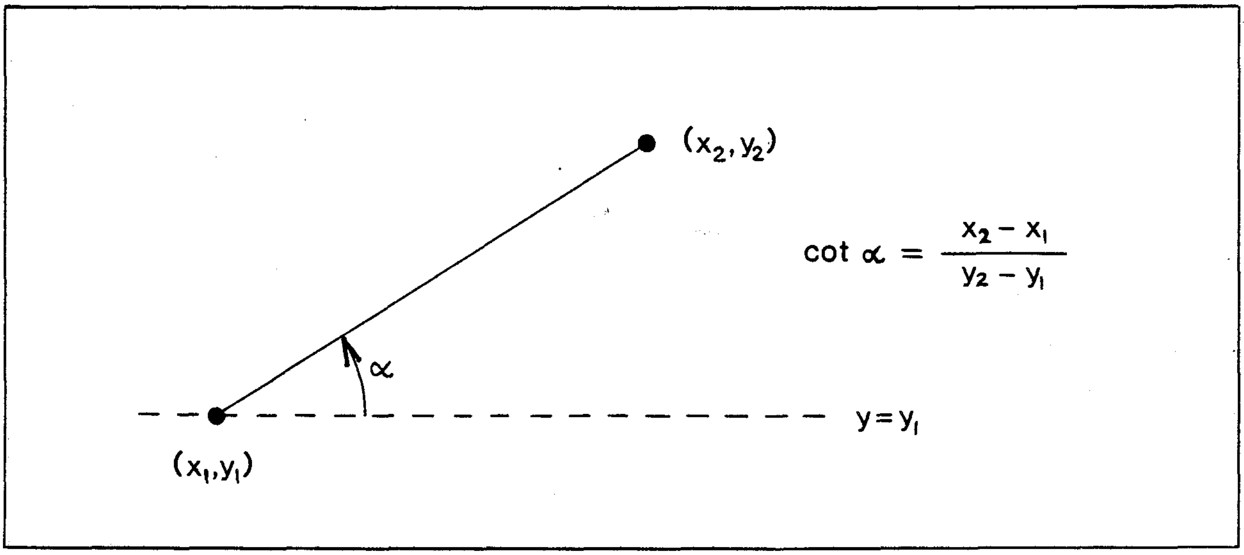

Figure 2-20 Definition of Cotangent

Because there are no horizontal edges in the active list, and we are not concerned with portions of edges below ycurr, $0 < \theta < \pi$. In this range, there is a one-to-one mapping between θ and

$\cot(\theta)$ (Figure 2-21); θ can be treated as a function of $\cot(\theta)$.

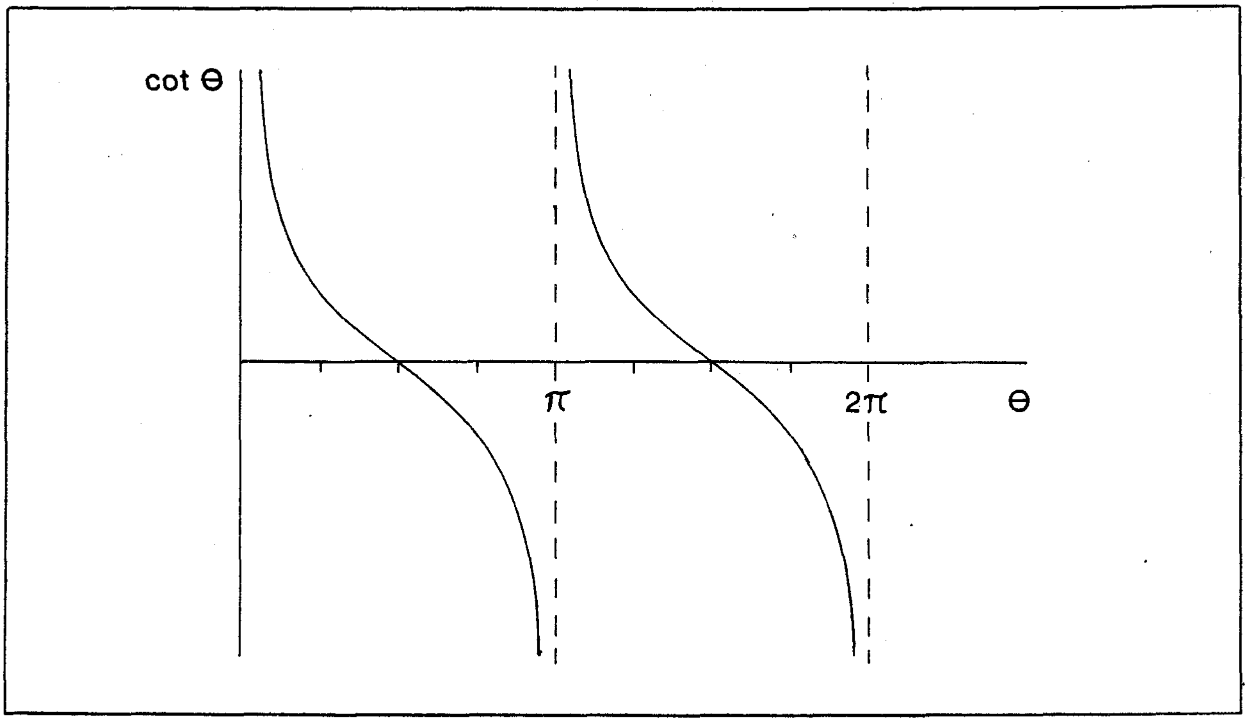

Figure 2-21 Cotangent Function

Thus, because $\cot(\theta)$ increases as θ decreases, active list edges with equal xcurr can be sorted on increasing $\cot(\theta)$.

A similar argument holds for edges with equal xnext value. In this case, we would like to sort on increasing θ, but can equivalently sort on decreasing $\cot(\theta)$.

2.3.4.2. Failure To Detect Crossing Point

The failure of the Newell & Sequin algorithm to detect the crossing of edges when some vertex has the same y coordinate as the crossing point prevents it from being completely general. A solution to the problem is as follows: when sorting the active list on xnext, check for pairs of edges in which both edges have the same xnext value and both extend above ynext. Any such pair crosses at (xnext,ynext), and must be broken up into four non-crossing edges. The only exception is collinear edges.

Why does this test work? The edges intersect at (xnext,ynext) because they have the same xnext value. Since both edges are in the active list, both are active at ycurr; thus both extend below ynext. If both edges also extend above ynext, then they must cross at (xnext,ynext). Collinear edges are an exception because they can have the same value of xnext, can extend below and above ynext, but not cross.

2.3.5. The Refined Newell & Sequin Algorithm

The Newell & Sequin algorithm, refined as described above, is shown below.

NEWELL & SEQUIN ALGORITHM (REFINED)

```
generate edge list, omitting horizontal edges and those
outside stripe, and assigning direction tag to each edge;
sort edge list on ymin(edge);
/* ymin(z) = y coordinate of lowermost point in z */
while active list or edge list is not empty do
begin
   if active list is empty then do
   begin
      ycurr ← ymin(first edge in edge list);
      if ycurr < ymin(stripe)
      then ycurr ← ymin(stripe);
   end;
   edge ← first edge in edge list;
   while ymin(edge) ≤ ycurr do
   begin
      transfer edge from edge list to active list;
      xcurr(edge) ← x(edge,ycurr);
      /*
        xcurr(edge) = x coordinate of intersection
        of edge with line y = ycurr
        x(edge,ycurr) is the x coordinate of point
        on edge with y coordinate = ycurr
      */
      edge ← next edge in edge list;
   end;
   sort active list on xcurr(edge), subsort on
   cotangent of angle between edge and line y = ycurr;
   find ynext; /* See below. */
   generate trapezoids; /* See below. */
   for each edge in active list do
   begin
      /*
        ymax(z) = y coordinate of uppermost point in z
      */
      if ymax(edge) = ynext or ynext = ymax(stripe) then do
      begin
         if edge has a mate then output trapezoid
         and unpair edge's mate;
         remove edge from active list;
      end;
      else xcurr(edge) ← xnext(edge);
      /*
        xnext(edge) = x coordinate of intersection of
        edge with line y = ynext
      */
   end;
   ycurr ← ynext;
end;
```

```
"FIND YNEXT" SEQUENCE (REFINED)
ynext ← min(active list ymax values);
ynext ← min(ynext,ymin(first edge in edge list));
ynext ← min(ynext,ymax(stripe));
for each edge in active list do
    xnext(edge) ← x(edge,ynext);
sort active list on xnext(edge), subsort on cotangent
of angle between edge and line y = ynext;
for each pair of active list edges (edge1 and edge2)
such that xnext(edge1) = xnext(edge2) do
begin
    if ymax(edge1) > ynext and ymax(edge2) > ynext and
    edge1 and edge2 are not collinear then do
    begin
        create edge3 with endpoints
        (xnext(edge1),ynext) and upper_end(edge1);
        create edge4 with endpoints
        (xnext(edge2),ynext) and upper_end(edge2);
        insert edge3 and edge4 into edge list;
        upper_end(edge1) ← (xnext(edge1),ynext);
        upper_end(edge2) ← (xnext(edge2),ynext);
    end;
    if active list order at ycurr ≠ order at ynext
    then do
    begin
        find intersection point of edges out of sort;
        break up 2 intersecting edges into 4 new
        edges, leaving lower 2 in active list and
        inserting upper 2 into edge list;
        ynext ← y(intersection point);
        go back to loop which assigns xnext values;
    end;
```

```
"GENERATE TRAPEZOIDS" SEQUENCE
wind ← 0;
left ← first edge in active list;
for each edge in active list do
begin
   wind ← wind + direction(edge);
   /*
     direction(edge) = direction tag assigned to edge
     wind = winding number for points to immediate right of edge
   */
   if wind = 0 then do
   begin
      if left and edge are not mates then do
      begin
         if left has a mate then output trapezoid
         and unpair left's mate;
         if edge has a mate then output trapezoid
         and unpair edge's mate;
         pair left and edge and initialize a
         trapezoid;
      end;
      left ← next edge in active list;
   end;
end;
```

2.4. EXPERIMENTAL

2.4.1. Context

The partitioning of polygons into trapezoids has been implemented within the context of a program to generate IC mask data for the Perkin-Elmer MEBES machine. The program was initially developed while working at Microtel Pacific Research under the Co-operative Education Program, and is still in routine use there. That version uses a polygon partitioning algorithm similar to that by Little and Heuft [3], but refined to handle horizontal splitting due to concavities. All program segments were initially written in the C programming language and designed to run under the UNIX operating system. Several of the high profile modules were later rewritten in assembler (by Microtel Pacific Research staff) to improve speed.

During subsequent work at Simon Fraser University the polygon partitioning part of the program was rewritten, using the refined Newell & Sequin algorithm. This change not only enabled the program to correctly handle arbitrary polygons (including all types of self-intersections) but also resulted in a reduction of CPU time, typically in the order of 50%.

2.4.1.1. Input

The program (called "bifmebes") receives as data an IC layout in Microtel's "Binary Intermediate Form" (BIF). BIF is a low level graphics language, somewhat similar to Caltech Intermediate Form (CIF) [4]; the differences are of no consequence here.

2.4.1.2. Output

Bifmebes generates the following output.

- one MEBES data file for each layer of the IC
- one MEBES tape header file (which is used as a tape directory)
- one conversion log file, in which are reported the names of BIF modules used, warnings and error messages, timing statistics, and statistics on e-beam primitives generated.

2.4.2. Testing

The program was tested on two types of data. It was first tested on data contrived to exercise various features, including various aspects of the partitioning algorithm. It was then tested on actual IC design data. The output was verified using several different techniques:

- examination of the conversion log file
- hand decoding of a hexadecimal dump of the MEBES binary data
- conversion of the MEBES data back to BIF, followed by overlaying plots of the original and converted polygons in different colors on the same monitor
- comparison of generated data to MEBES data produced by the earlier version of the

program. The earlier version has been tested by the above techniques and also by photographic enlargement of a mask produced on the MEBES machine using bifmebes output.

2.4.3. Results

In all cases, polygon partitioning has been shown to be correct. The data generated is consistently accurate to within the resolution of the MEBES machine.

Chapter 3

OPTIMIZATION TECHNIQUES

3.1. INTRODUCTION

Mask vendors usually include the total amount of e-beam machine time consumed in the cost of mask fabrication. Typically there is a fixed charge up to a certain time limit, and then the customer pays for each minute of machine time past that limit. This provides economic motivation to minimize e-beam machine time, and makes optimization of the MEBES data desirable. There are several possible techniques to do this:

- reducing the trapezoid count
- recognizing repeated patterns, and
- sequencing of rasterization and stage movement.

3.2. REDUCTION OF TRAPEZOID COUNT

Each trapezoid in the e-beam data must be decoded and rasterized — the more trapezoids, the more machine overhead. (With raster scan machines, the actual time spent exposing the mask will not be affected, as writing does not start until all trapezoids of the current stripe have been rasterized and saved in the bit map. With vector scan systems, though, the trapezoid count would affect exposure time.) Two possible techniques for reducing the trapezoid count are non-horizontal partitioning and trapezoid merging.

3.2.1. Non-Horizontal Partitioning

Although Newell and Sequin [6] claim that their algorithm generates a minimum set of trapezoids, this is not true. Neither is it true for any of the other polygon partitioning algorithms examined. The reason? All such algorithms are limited to partitioning along horizontal lines. If polygons could also be partitioned non-horizontally, then the total trapezoid count could be reduced.

For example, refer to Figure 3-1.

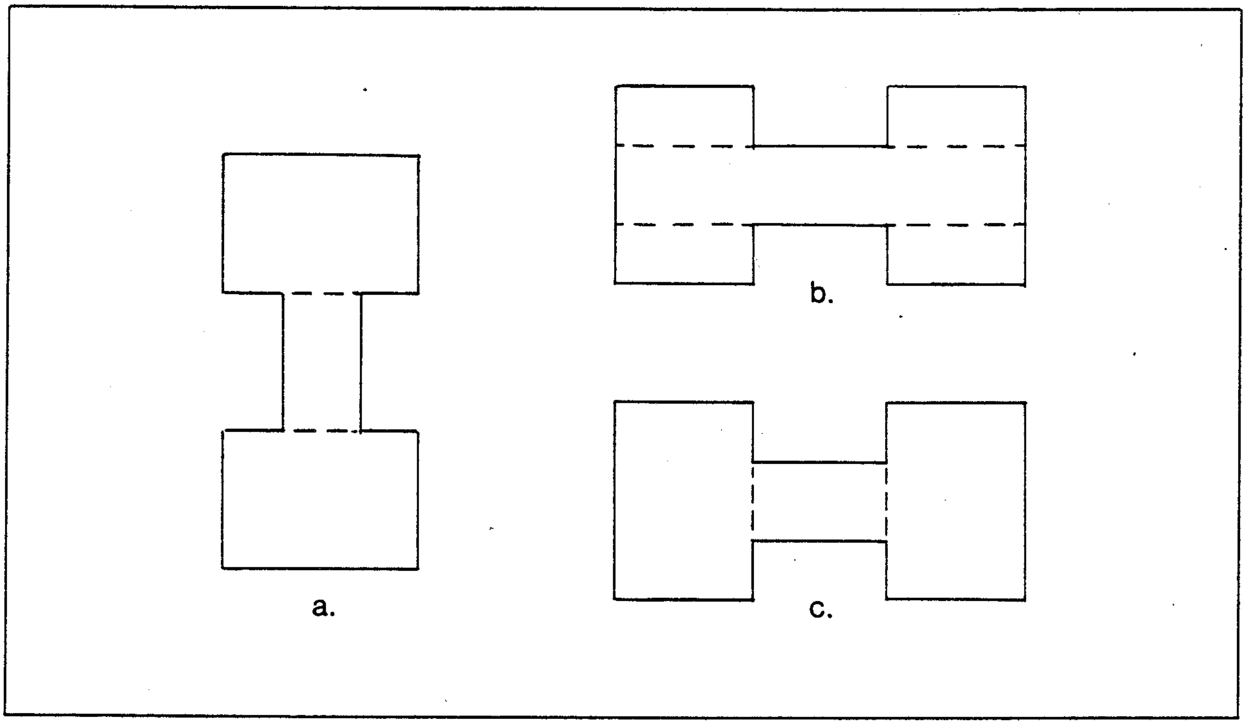

Figure 3-1 Horizontal & Non-Horizontal Partitioning

The figure on the left (a) cannot be partitioned into any less than three trapezoids. In this orientation, horizontal partitioning yields the minimum set. But if the figure is rotated through 90 degrees, this is no longer true. Horizontal partitioning now produces five trapezoids (figure b), while non-horizontal partitioning yields three (figure c).

3.2.2. Trapezoid Merging

In this technique adjoining trapezoids which satisfy certain conditions are merged into a single trapezoid. Consider two adjoining trapezoids, *ABCD* and *EFGH* (Figure 3-2).

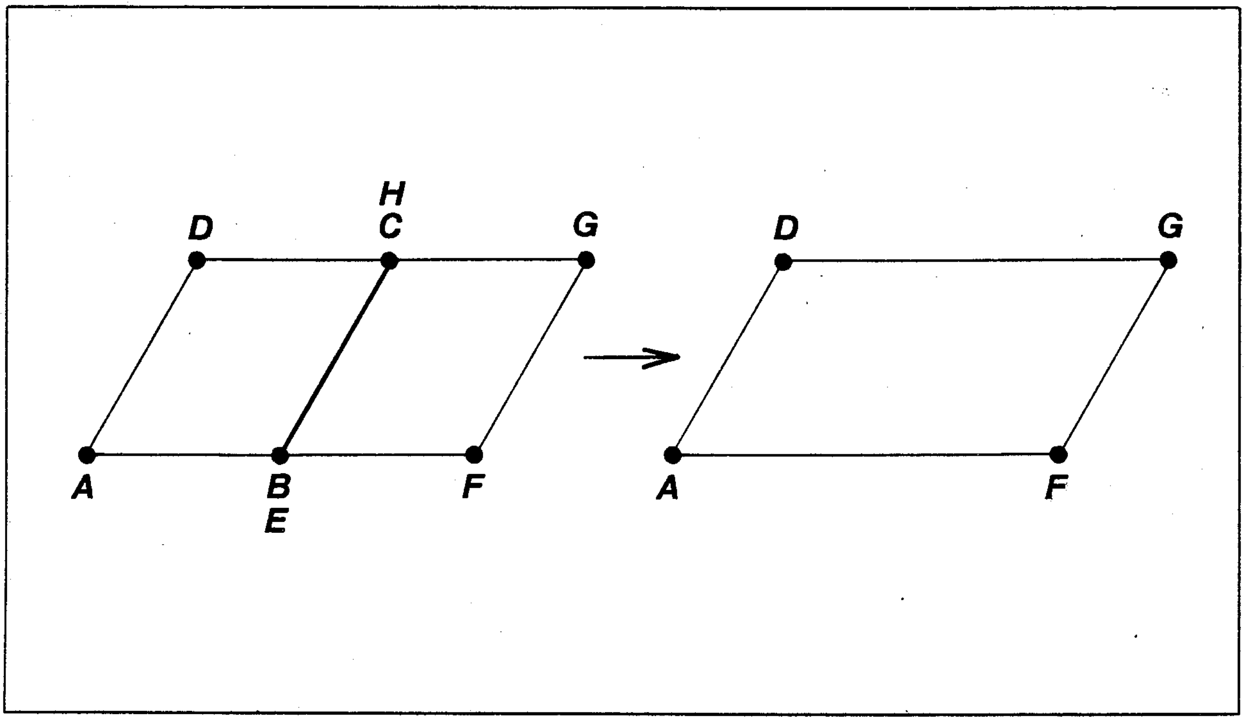

Figure 3-2 Trapezoid Merging

Let vertex B be coincident with vertex E, and let vertex C be coincident with vertex H. If edges AB and EF are collinear, and if edges CD and GH are collinear, then the trapezoids may be merged, yielding one new trapezoid, $AFGD$.

It is possible that trapezoid merging could significantly reduce the number of trapezoids. The application of hierarchical design techniques to integrated circuits results in modularization of the IC layout. Each module is placed into the overall design wherever its pattern is required. Often, modules are designed to abut, greatly reducing the amount of explicit interconnection needed. *I.e.* interconnection is achieved by butting polygons, and butting polygons will give rise to mergable trapezoids.

3.3. RECOGNITION OF REPEATED PATTERNS

The MEBES system software accepts a "data compaction option", which is of benefit in designs having a high degree of regularity (*e.g.* memory). This option is equivalent to an array of trapezoids. A trapezoid is described once, together with a row count, a column count, the inter-row spacing, and the inter-column spacing, and the MEBES software generates and rasterizes all instances of the trapezoid. Making use of the data compaction option would require the recognition of array-like patterns of trapezoids. (Although this type of information is often available at the design level, it is lost on translation to BIF or CIF.)

3.4. SEQUENCING OF RASTERIZATION AND STAGE MOVEMENT

As described in Chapter One, the MEBES machine rasterizes one stripe, and then writes all instances of that stripe on the mask before proceding to the next stripe. This is an efficient approach for production type masks, in which all dies on the mask are identical. (A *die* is the pattern corresponding to one chip.) It may not be so efficient, though, for research type masks, in which a number of different dies are present. With a production mask, stage movement between instances of a stripe is small. But in a multi-project mask, such stage movement can be fairly large, and much time can be wasted.

An alternative sequence would be as follows.

```
until all instances of all stripes have been written do
begin
    rasterize the stripe closest to the current stage position;
    write that stripe onto the mask;
end;
```

This sequence could also be inefficient at times. For example, if the next instance of the current stripe is close, it may be faster to move the stage than to rasterize the next stripe. The optimum sequence is somewhere in between the two extremes.

A branch and bound approach could be used to find the optimum sequence, but would be far more time consuming than just using the normal sequence. There is a very large number of stripes on a mask, so the branching factor in a branch and bound tree would be high. Thus a heuristic would be needed to find a near optimal sequence.

3.5. SELECTION OF OPTIMIZING TECHNIQUE

Trapezoid merging was not attempted due to insufficient time.

The recognition of array-like patterns for the data compaction option was not tried because it was unlikely to have much impact on e-beam machine time. The major benefit of data compaction is in reducing the quantity of data; data compaction does produce a slight reduction in overhead, but its effect on overall machine time is minimal (personal communication with Lee Bernier, Software Manager at Perkin Elmer, 4 May 1983).

The sequencing of rasterization and stage movement was not attempted for two reasons.

(1) Current MEBES machines do not provide any control over such sequencing.

(2) Information needed to simulate the effect of different sequences on machine time was difficult to obtain.

The only optimization technique actually tried was non-horizontal partitioning. This is discussed in detail in Chapter Four.

Chapter 4

NON-HORIZONTAL PARTITIONING

4.1. OBJECTIVES

As discussed in Chapter 3, non-horizontal partitioning of polygons is one way to reduce the total trapezoid count and hence partially optimize the e-beam data. The objectives of this part of the study were

(1) to develop an algorithm for non-horizontal partitioning of polygons,

(2) to implement the algorithm, and

(3) to evaluate its effectiveness.

4.2. ANALYSIS

4.2.1. The Need For A Heuristic

An obvious method to introduce non-horizontal partitioning would be to use a branch and bound algorithm instead of one like Newell and Sequin's. In this approach, the tree of all possible partitions would be traversed, backtracking wherever the partition does not yield acceptable e-beam trapezoids or wherever the trapezoid count exceeds the best solution found so far. This technique would find an optimum partitioning, but would likely be prohibitive in CPU time. This suggests the need for a heuristic which could be used to partition a polygon non-horizontally. Once no further partitioning by the heuristic is possible, each new polygon generated can be partitioned horizontally using the Newell and Sequin algorithm.

4.2.2. The Collinear Edge Heuristic

The observation which prompted the "collinear edge heuristic" is as follows: *in many cases, non-optimal partitioning is caused by failing to recognize that two collinear edges should (under the right conditions) be rearranged to form two new edges, partitioning the polygon in the process*. An example of this is seen in Figure 3-1. In this example, horizontal partitioning of the figure in (b) yields five trapezoids. In (c), the trapezoid count has been reduced to three by rearranging pairs of collinear edges. A rough algorithm for edge rearrangement in the collinear edge heuristic follows. (Refer to Figure 4-1.)

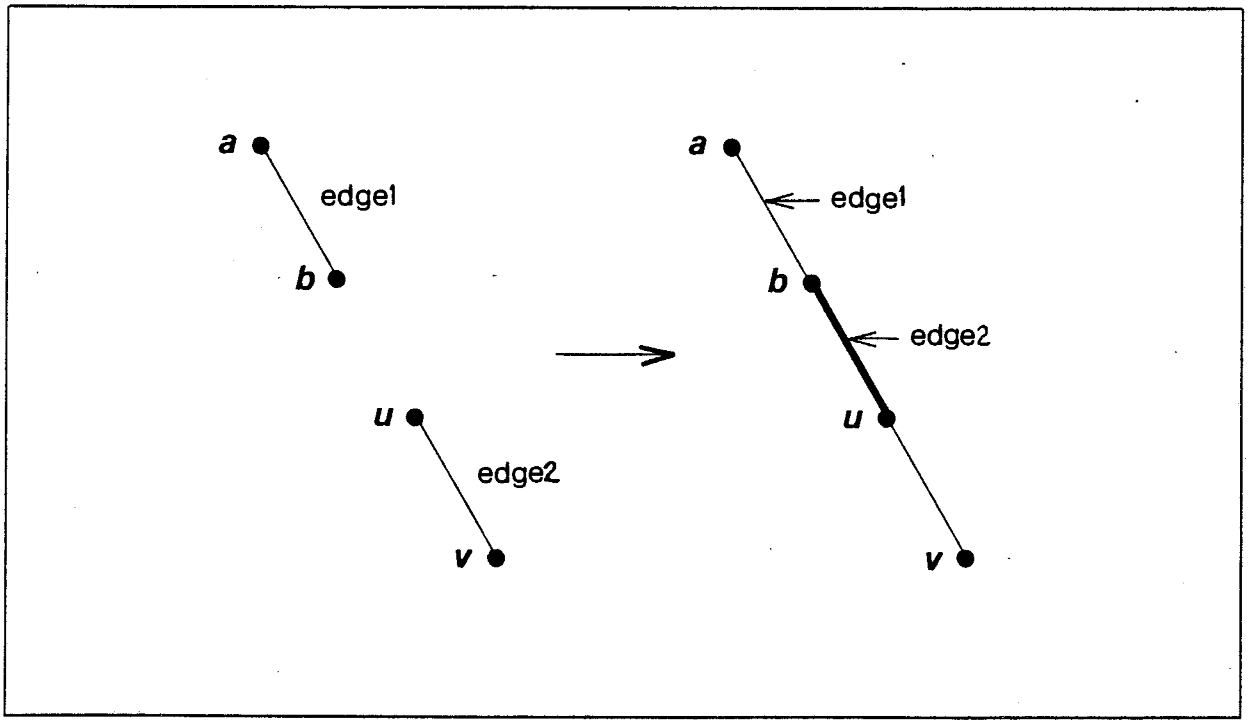

Figure 4-1 Edge Rearrangement in Collinear Edge Heuristic

```
find a pair of collinear edges (edge1 and edge2) which
satisfy the conditions for the collinear edge heuristic;
a ← endpoint of edge1 farthest from edge2;
b ← endpoint of edge1 closest to edge2;
u ← endpoint of edge2 closest to edge1;
v ← endpoint of edge2 farthest from edge1;
edge1 ← edge av ;
edge2 ← edge ub;
polygon1 ← polygon containing av ;
polygon2 ← polygon containing ub;
```

For example, application of the algorithm to the polygon in Figure 4-2 proceeds as follows.

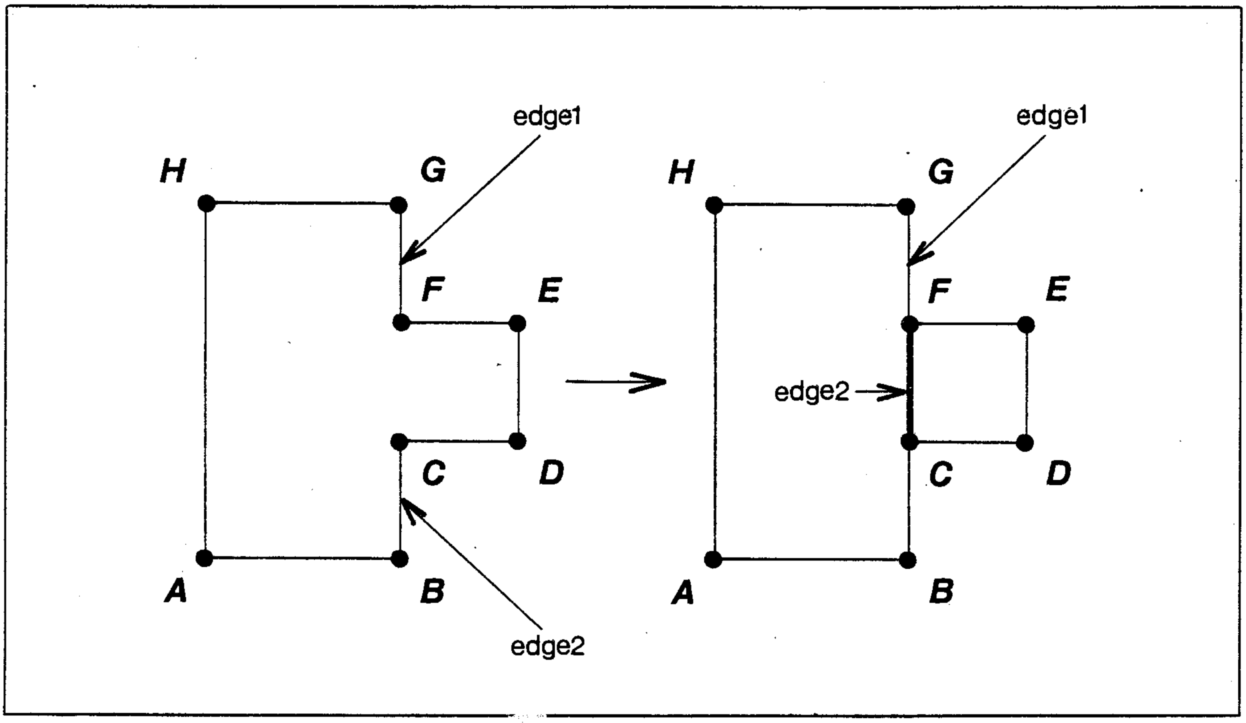

Figure 4-2 Example of Edge Rearrangement Algorithm

$a \leftarrow G$, $b \leftarrow F$, $u \leftarrow C$, and $v \leftarrow B$. Edge1 ← GB and edge2 ← CF. Polygon1 ← $ABGH$ and polygon2 ← $CDEF$.

4.2.3. Partitioning Conditions For Collinear Edge Heuristic

All pairs of collinear edges can't be rearranged to partition a polygon and produce valid e-beam data. For example, consider the polygon shown in Figure 4-3.

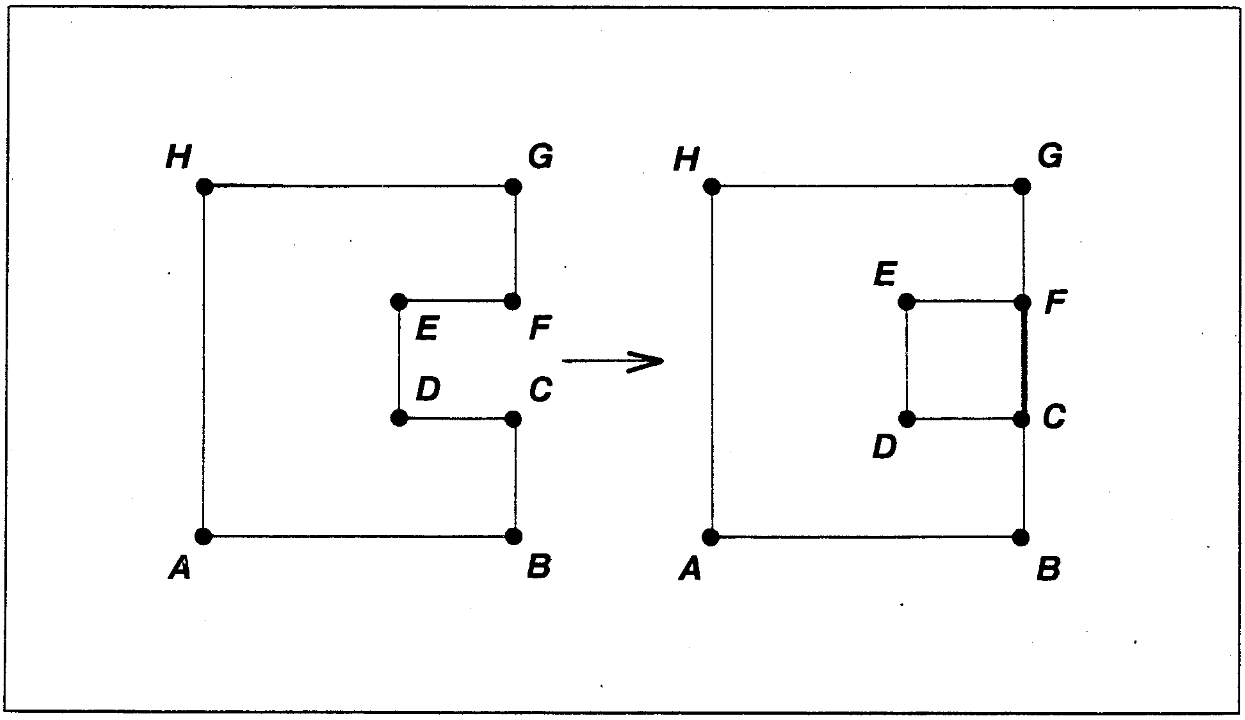

Figure 4-3 Invalid Edge Rearrangement

Edges *BC* and *FG* are collinear, but rearrangement as described above produces two new polygons which, together, do not resemble the original. Specifically, the region bounded by vertices *C*, *D*, *E* and *F* has been filled in.

The conditions which must be met before applying edge rearrangement in the collinear edge heuristic are as follows.

(1) Edges to be modified must be collinear.

(2) Edges to be modified must have the same traversal direction.

(3) After edge rearrangement, polygon1 must not enclose any vertex of polygon2, and *vice versa* (*non-enclosure condition*).

(4) After edge rearrangement, if polygon1 or polygon2 has any regions of zero width, then each such region must be no larger than a point (*non-zero-width condition*).

4.2.4. Justification of Partitioning Conditions

These four conditions are necessary and sufficient to permit non-horizontal partitioning using the collinear edge heuristic with no loss of data integrity. In the following justification for this claim, let the edges involved be as shown in Figure 4-1.

4.2.4.1. Necessity of Conditions

4.2.4.1.1. Collinearity Suppose collinearity of edges *ab* and *uv* isn't necessary. Then rearrangement of two edges which satisfy all of the other conditions for partitioning must produce a valid partition. Consider the polygon in Figure 4-4.

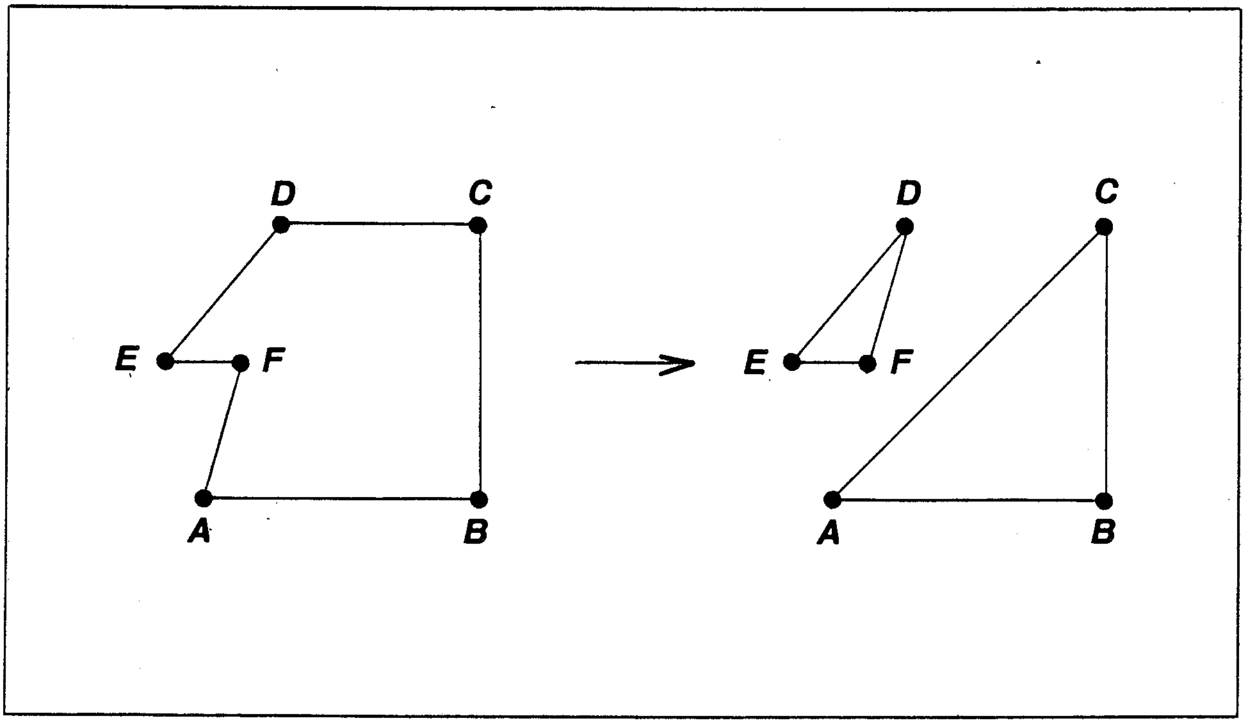

Figure 4-4 Rearrangement of Non-Collinear Edges

Edges *CD* and *FA* are not collinear, but satisfy the traversal direction, non-enclosure and non-zero-width conditions. Edge rearrangement must involve formation of an edge between *C* and *A*. (The only other possibility would be an edge between *C* and *F*, but this would leave edge

FA intact while breaking edge *EF*, *i.e.* edge *EF* would be involved instead of *FA*.) Thus edges *CD* and *FA* are broken, and edges *CA* and *FD* are formed. This yields two polygons whose union bears little resemblance to the original. Thus a counterexample has been demonstrated which contradicts the above supposition, indicating that edge collinearity is a necessary condition.

4.2.4.1.2. Traversal Direction Consider a point which traverses the polygon boundary in such a way that each vertex is passed only once. The *traversal direction* of each edge is the direction in which the point travels as it traverses that edge. The traversal direction condition states that edges to be rearranged in the collinear edge heuristic must have the same traversal direction. Suppose this condition is not necessary. Let edges *ab* and *uv* satisfy all of the other partitioning conditions, but have opposite traversal directions (Figure 4-5).

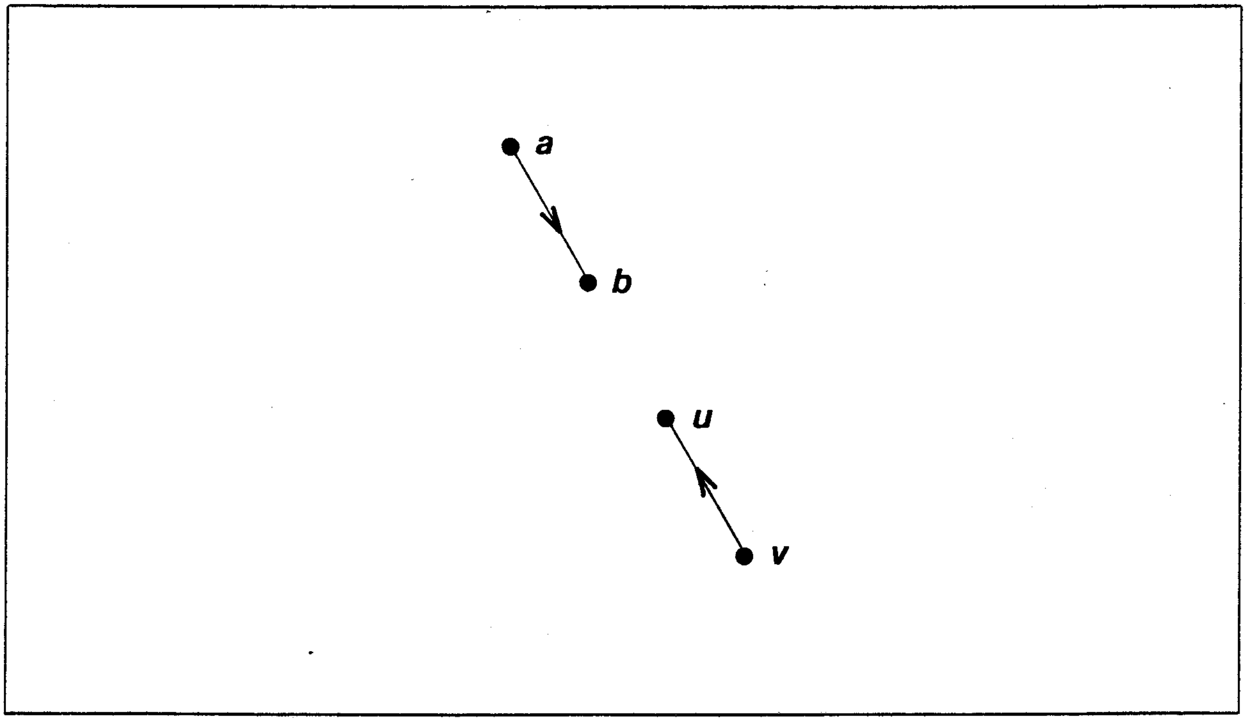

Figure 4-5 Edges With Opposite Traversal Direction

Aside from edge *ab*, there must be a path from *b* to *a* to close the polygon. This path must include *u* and *v*, otherwise those vertices would not belong to the same polygon. The portion of this path between *b* and *u* must include *v*, otherwise the traversal directions of *ab* and *uv* would be violated. Thus the vertex adjacency list for the polygon is *a,b,...,v,u,...,a*. Rearrangement of *ab* and *uv* as per the collinear edge heuristic generates the following vertex adjacency list: *a,v,...,b,u,...,a*. But this is still one polygon; partitioning has not occurred! This example contradicts the assumption regarding traversal direction. Edges to be rearranged must have the same traversal direction.

4.2.4.1.3. Non-Enclosure Edge rearrangement partitions a polygon into two new polygons, polygon1 and polygon2. Suppose polygon1 may enclose some of polygon2's vertices (or *vice versa*) without harming data integrity. Consider the polygon in Figure 4-3. Edges *BC* and *FG* satisfy all conditions of the collinear edge heuristic except for non-enclosure. Rearrangement of these two edges yields polygons *ABGH* and *CDEF*. This will have the effect of filling in the region enclosed by vertices *C*, *D*, *E*, and *F*, which is clearly an error. The counterexample contradicts the assumption that non-enclosure is not necessary.

4.2.4.1.4. Non-Zero-Width If partitioning a polygon yields a region of zero width which is larger than a point, then subsequent partitioning will generate a zero-area trapezoid for that region. This increases, rather than decreases the amount of e-beam machine overhead, and on some machines, may not even be tolerated at all. Thus the non-zero-width condition is necessary.

For example, refer to Figure 4-6. Edges *BC* and *FG* satisfy all collinear edge heuristic conditions except for non-zero width. Edge rearrangement yields polygons *CDEF* and *ABGHIJKL*. Subsequent horizontal partitioning of the latter polygon generates trapezoids *ABKL*, *GHIJ* and *JKKJ*.

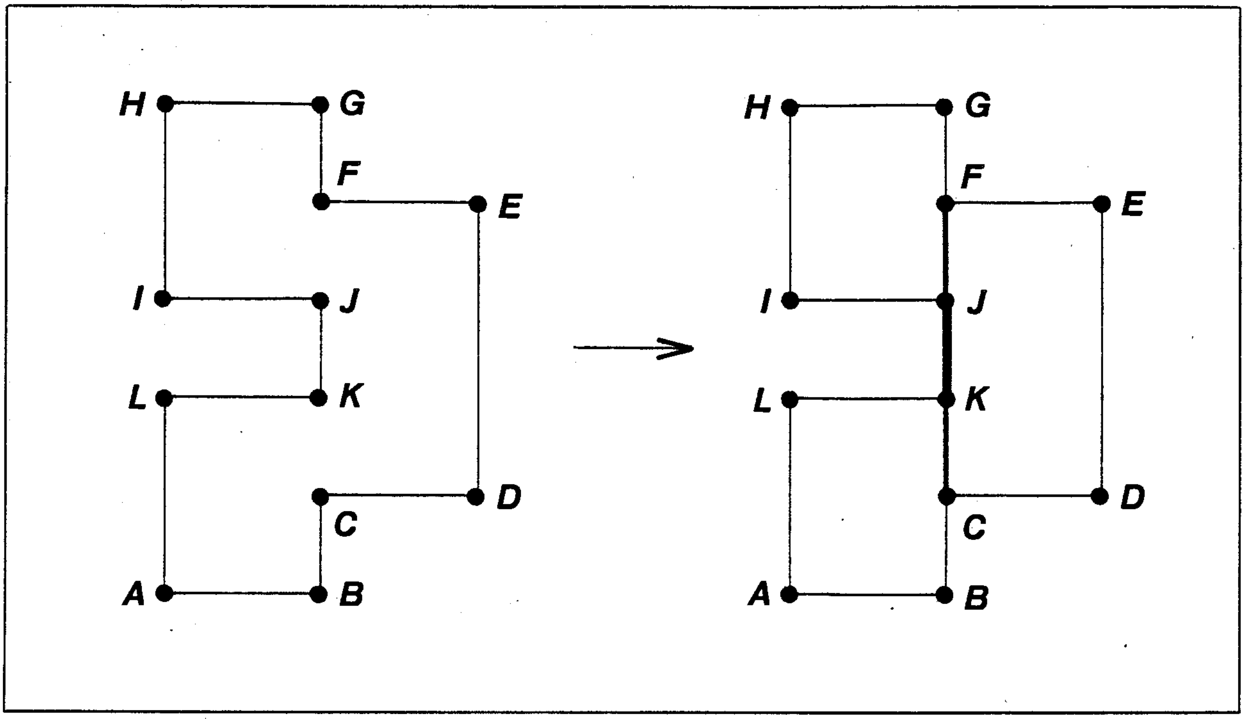

Figure 4-6 Generation of Zero-Area Trapezoid

4.2.4.2. Sufficiency of Conditions

The conditions for partitioning using the collinear edge heuristic are sufficient to ensure data integrity if the shape of the union of the areas enclosed by the new polygons is the same as the shape of the area enclosed by the original polygon. Collinear edge heuristic partitioning introduces no new vertices, deletes no vertices, and moves no vertices. Furthermore, because of the non-enclosure condition, no vertex will be obscured by another polygon. Thus any change in polygon shape must be due to edge modification, not vertex modification.

Two, and only two edges are modified (Figure 4-7).

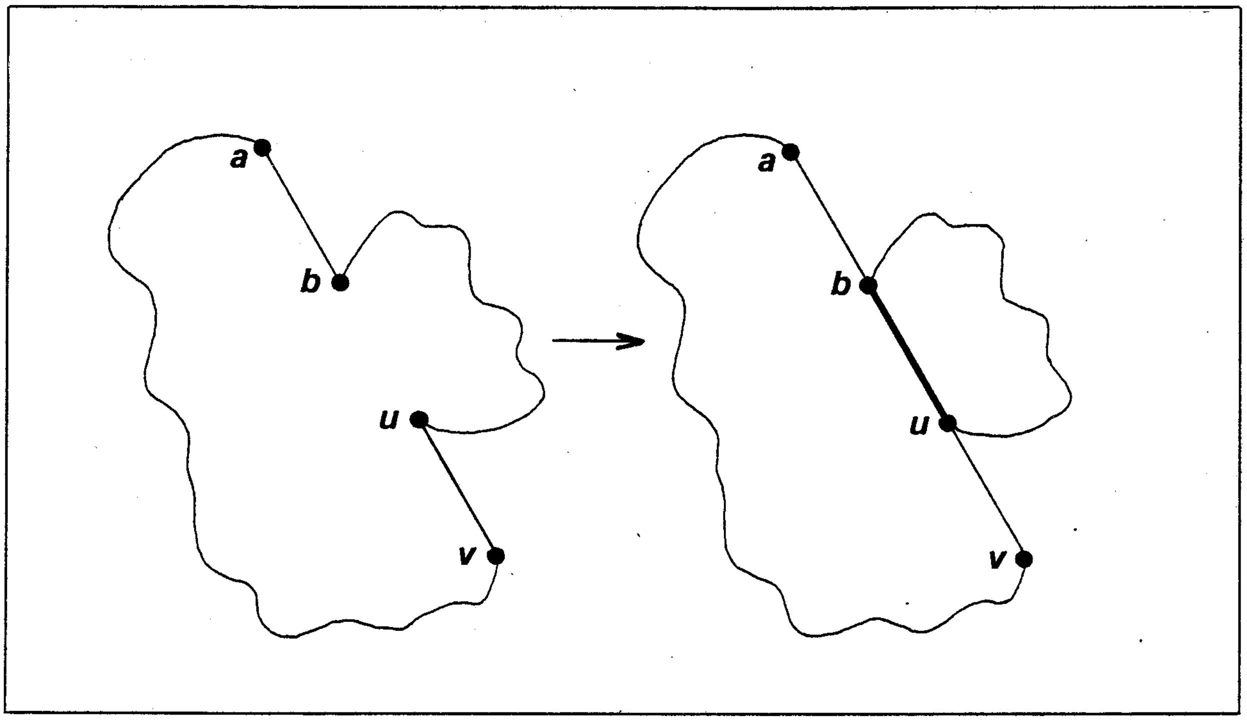

Figure 4-7

Consider the polygon before edge rearrangement. Because of the collinearity condition, *a*, *b*, *u*, and *v* all lie on the same line. As a result of the non-enclosure condition, no part of the boundary can pass between *b* and *u*, thus this region must be inside the polygon. Since polygons are area-filled on a mask, a line segment between *b* and *u* would not affect the shape on the mask. Now consider the two new polygons after edge rearrangement. Vertices *a*, *b*, *u* and *v* have not been moved, thus they must still lie on the same line. The portion of *av* between *a* and *b* (after partitioning) is equivalent to *ab* (before partitioning). The portion of *av* between *u* and *v* is equivalent to *uv*. The portion of *av* between *b* and *u*, and edge *ub* are both equivalent to a line segment between *b* and *u*. Nothing has been introduced which can alter the shape of the enclosed area. Thus the collinear edge heuristic partitioning conditions are sufficient to ensure data integrity.

4.2.5. Tests Used In Partitioning Conditions

For a discussion of the collinearity test, see Appendix 1.

The test for traversal direction is quite simple. Because the edges under test will only be considered if they are collinear and non-horizontal, the test reduces to determining whether each edge is upwards or downwards directed. This is just the edge direction tag used in the Newell and Sequin algorithm.

An obvious algorithm for the non-enclosure test is to tentatively partition the polygon, and then test each vertex of each new polygon for enclosure by the other polygon. This method would have complexity of **O** (n^2). An alternative algorithm uses the convexity or concavity of vertices; this algorithm (which is discussed next) also tests for the non-zero-width condition at the same time.

4.2.5.1. Vertex Convexity Algorithm

This approach requires that the polygon not be self-crossing, and that the collinearity and traversal direction conditions be met. The algorithm is as follows. (Refer to Figure 4-1.)

```
if vertex $b$ is convex or vertex $u$ is convex
then reject the partition;
else
    if any edge of the polygon (except those adjacent to
    $b$ and $u$) intersects the gap between $b$ and $u$
    then reject the partition;
```

The *gap* between *b* and *u* is defined to be the non-existent straight line segment with end points *b* and *u*. Also, recall that (at least within this thesis) *intersect* means to touch or cross. For a description of how to determine the convexity or concavity of a vertex, refer to Appendix 2.

The requirement that the polygon be non-self-crossing makes this algorithm less general than the obvious one. On the other hand, what is gained by this restriction is a reduction in complexity from **O** (n^2) to **O** (n).

4.2.5.2. Justification of Vertex Convexity Algorithm

As mentioned above, the vertex convexity algorithm provides a test for both the non-enclosure and non-zero-width conditions. In the following proof of this claim, let *ab* and *uv* be the edges under test, let them belong to a non-self-crossing polygon, and let them satisfy the collinearity and traversal direction conditions. Edges *ab* and *uv* are as shown in Figure 4-1.

4.2.5.2.1. Non-Enclosure Condition We must show that

(a) the convexity of *b* or *u* implies that non-enclosure cannot be guaranteed, and

(b) the concavity of *b* and *u* together with the absence of any edge which intersects gap *bu* implies that non-enclosure is met.

The line containing *ab* and *uv* partitions the plane into two half-planes. Let H1 be the half-plane on the interior side of *ab*, and H2 be the half-plane on the exterior side (Figure 4-8).

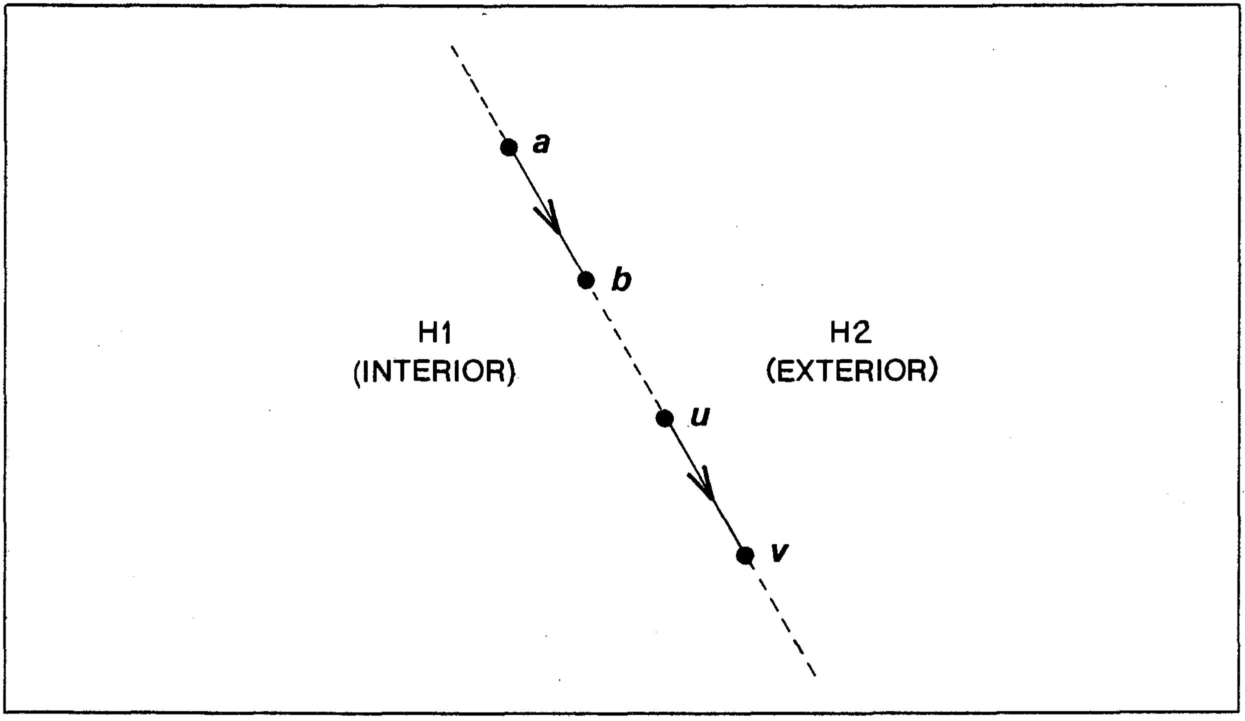

Figure 4-8 Half-Planes H1 and H2

There must be a path from *b* to *a* (other than edge *ab*) to close the polygon. This path must include *u* and *v*, otherwise these vertices would not be part of the same polygon. The portion of this path between *b* and *v* must include *u*, otherwise traversal direction would be violated. Thus the vertex adjacency list is *a,b,...,u,v,...,a*. Imagine a point moving along the polygon boundary from *a* to *v*. Using that point as a viewpoint, and the traversal direction as the viewing direction, the polygon interior is always on the same side of the boundary. (The only way this could be violated is if edges were allowed to cross each other.) Thus, regardless of what course the path from *b* to *u* takes, H1 is always on the interior side of both *ab* and *uv*.

First let us address point (a). Given that vertex *b* is convex, prove that non-enclosure cannot be guaranteed. Let *c* be the vertex ≠ *a* adjacent to *b* (Figure 4-9).

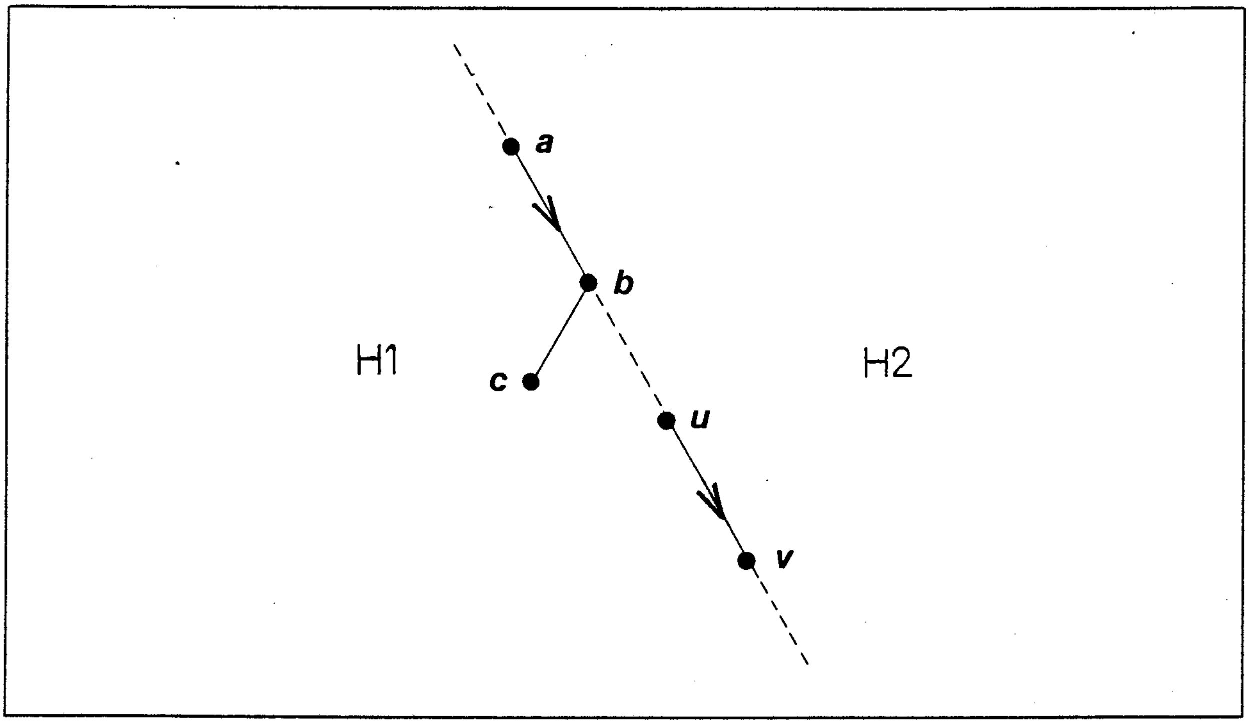

Figure 4-9

By the definition of a convex vertex, interior angle(*abc*) < 180°, and *c* lies in H1.

Suppose edge rearrangement is performed on *ab* and *uv* ; these edges are replaced by *av* and *ub* (Figure 4-10). Let polygon1 be the new polygon containing *a* and *v*. Let polygon2 be the new polygon containing *u*, *b* and *c*. The section of *av* between *a* and *b* (after partitioning) is equivalent to *ab* (before partitioning). The section of *av* between *u* and *v* is equivalent to *uv*. Because H1 was on the interior side of *ab* and *uv*, it is also on the interior side of *av*. Vertex *c* still lies in H1. Since no edge crosses *bc*, no part of the boundary separates *c* from the interior of polygon1. Thus a vertex of polygon2 (*c*) is enclosed by polygon1, and non-enclosure has not been met. A similar argument holds for the case where *u* is convex. It may then be concluded that neither *b* nor *u* may be convex if non-enclosure is to be guaranteed.

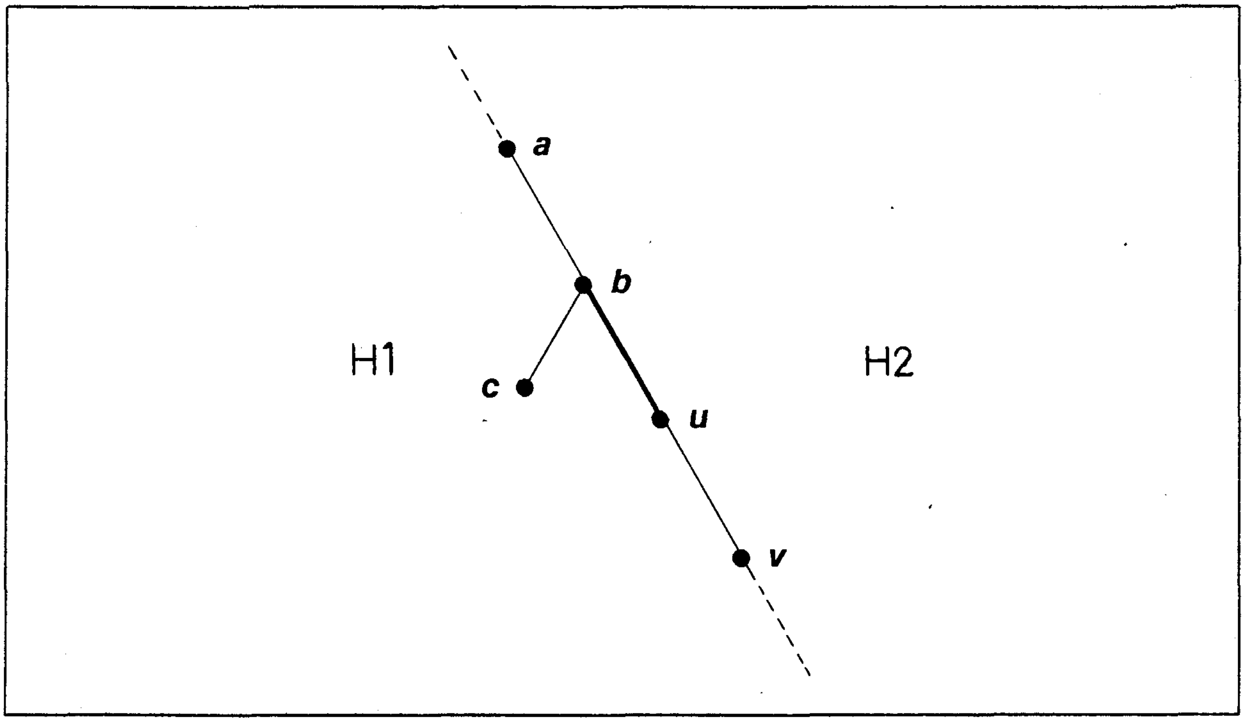

Figure 4-10

Now we will address point (b): the concavity of *b* and *u* together with the absence of any edge intersecting gap *bu* ensures that non-enclosure will be met. Let *ab*, *uv*, H1 and H2 be as before. Let *c* be the vertex ≠ *a* adjacent to *b*, and let *t* be the vertex ≠ *v* adjacent to *u*. Given that *b* and *u* are both concave, and that no edge intersects gap *bu*. Refer to Figure 4-11.

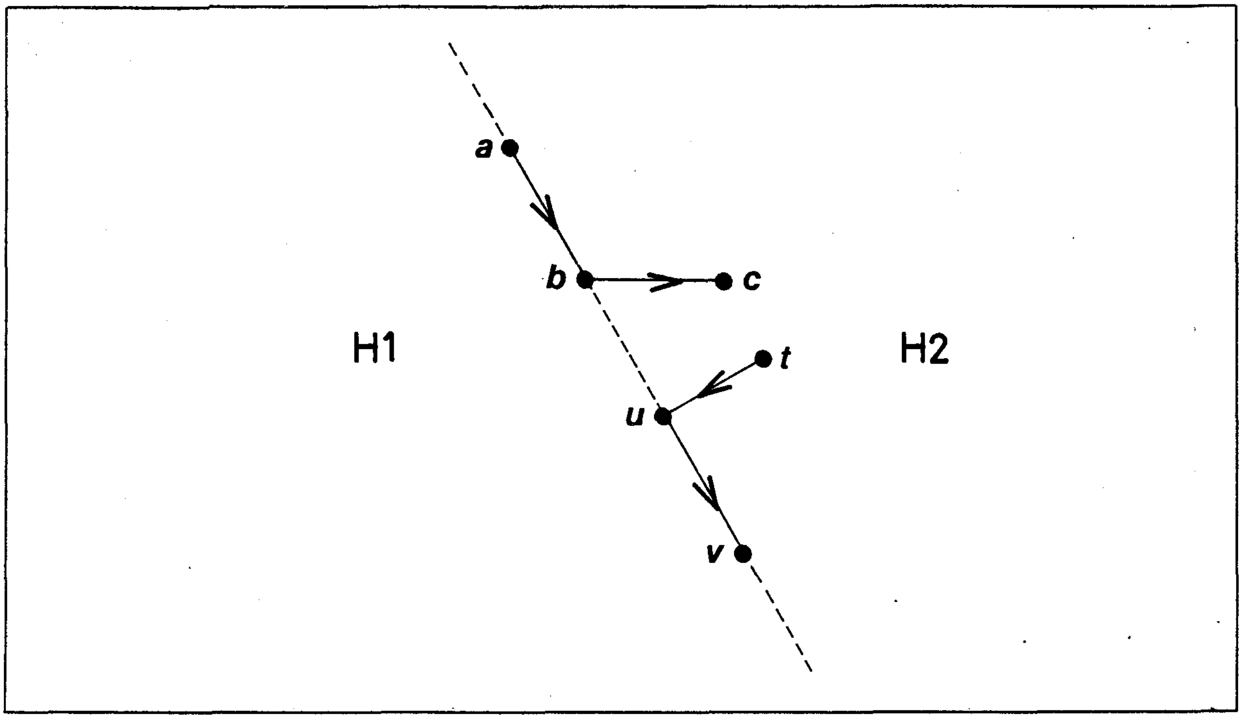

Figure 4-11

There must be a path from *c* to *a* (other than *a,b,c*) to close the polygon. This path must include *t*, *u* and *v*, since these vertices are part of the polygon. Also, the part of the path between *c* and *v* must include *t* and *u*, otherwise the traversal direction would be violated. Thus the vertex adjacency list is *a,b,c,....,t,u,v,....,a*. When edge rearrangement is applied to edges *ab* and *uv*, they are replaced by new edges *av* and *ub* (Figure 4-12).

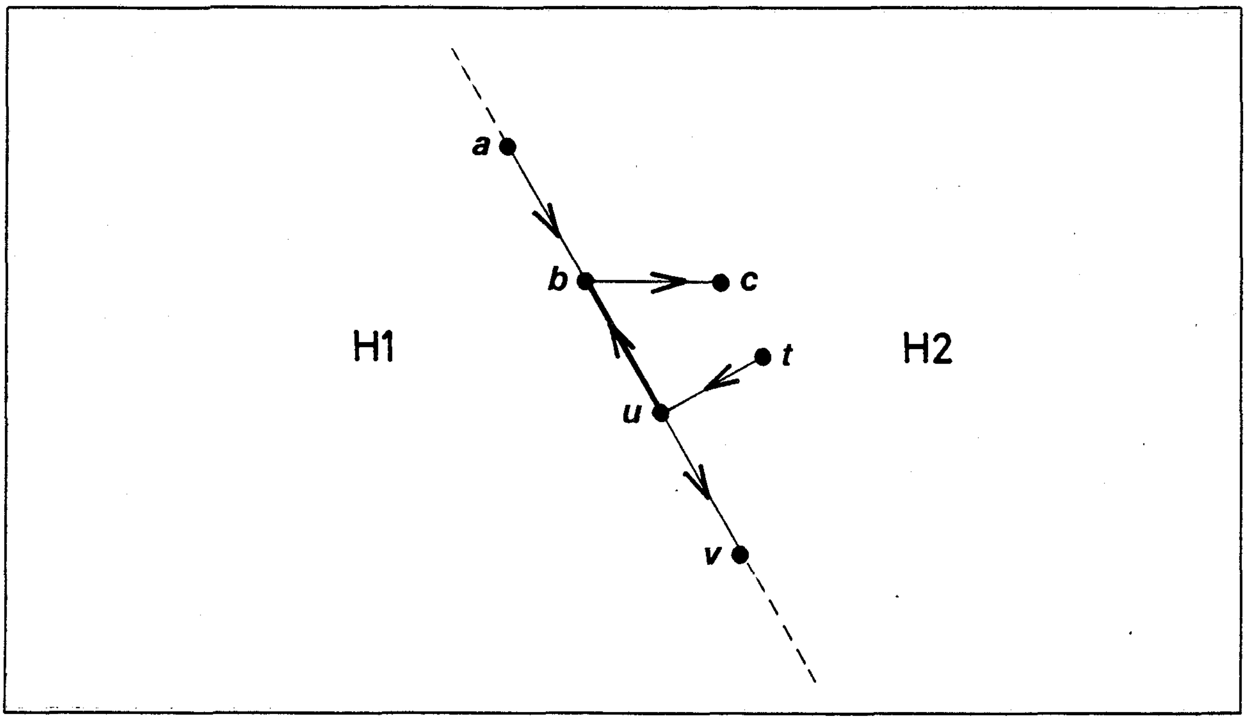

Figure 4-12

One new polygon (polygon1) will have vertex adjacency list *a,v,...,a*; the other (polygon2), will have vertex adjacency list *b,c,...,t,u,b*.

The only edges affected by the partitioning are *ab* and *uv*. Because no other edge of the original polygon crosses any other edge, the only edges which might introduce edge crossing after partitioning are *av* and *ub*. The portion of *av* between *a* and *b* is equivalent to *ab*, the portion between *u* and *v* is equivalent to *uv*, and the portion between *b* and *u* is equivalent to gap *bu*. No edge crosses *ab* or *uv*. Since no edge intersects gap *bu*, no edge crosses this gap. Thus no edge will cross *av*. Edge *ub* is equivalent to gap *bu*; since no edge crosses the gap, no edge will cross *ub*. Thus polygon1 and polygon2 are non-self-crossing. Furthermore, no edge of polygon1 crosses any edge of polygon2.

Refer back to the pre-partitioned situation (Figure 4-11). Because *b* is concave, interior angle(*abc*) > 180°, and *c* lies in H2. Similarly, *t* also lies in H2. After partitioning (Figure 4-12), H1 will be on the interior side of *av*, and H2 on the exterior side. Because no vertices are moved, *c* and *t* still lie in H2. No edge crosses *bc* or *tu*, so no part of a boundary can separate *c* or *t* from the exterior of polygon1. Thus *c* and *t* are outside of polygon1. Vertices *b* and *u* lie on edge *av*, but are not inside polygon1. The only way in which any other vertex of polygon2 can be inside polygon1 is for a polygon2 edge to cross a polygon1 edge. But this has been shown to not happen. Thus polygon1 cannot enclose any polygon2 vertex.

But what about polygon2 enclosing a polygon1 vertex? Consider the polygon before partitioning (Figure 4-13).

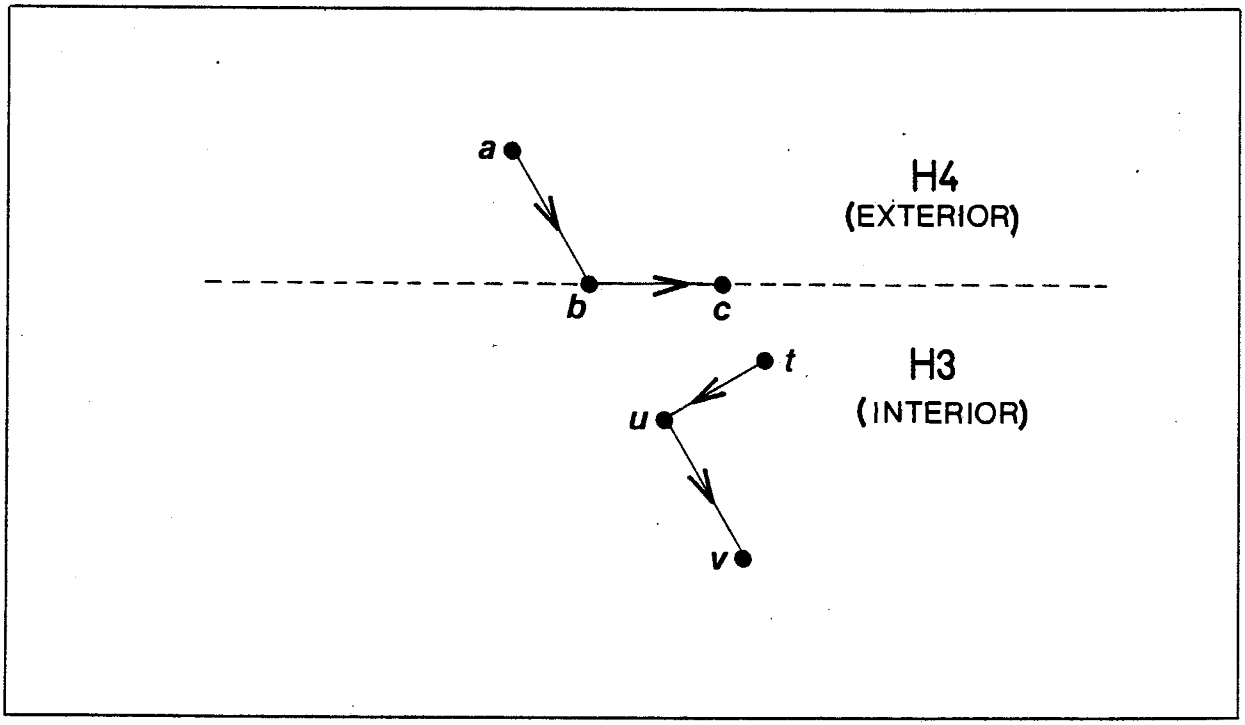

Figure 4-13

The line containing edge *bc* partitions the plane into two half-planes. Let H3 be the half-plane on the interior side of *bc*, and H4, the half-plane on the exterior side. Because *b* is concave, interior angle(*abc*) > 180°, and *a* lies in H4. Now consider the situation after partitioning (Figure 4-14).

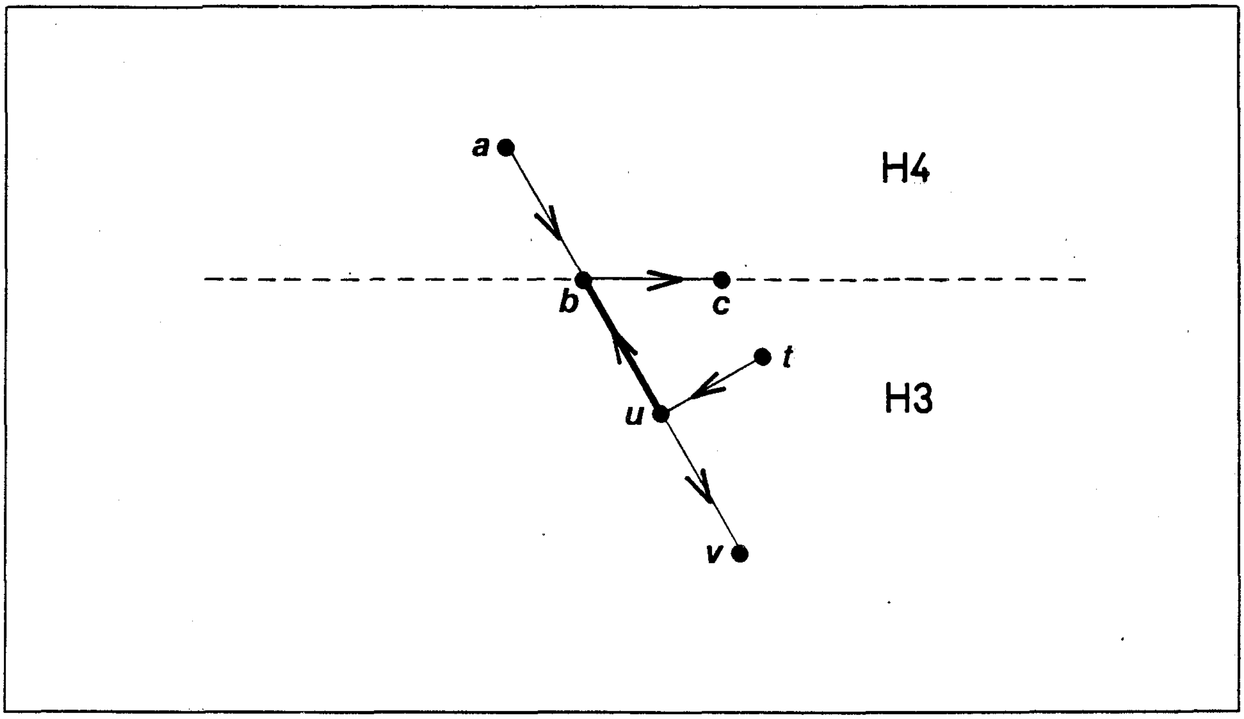

Figure 4-14

Edge *bc* is still in the same place. Polygon2's vertex adjacency list means that the region bounded by edges *tu*, *ub* and *bc* is inside polygon2. Thus, following edge rearrangement, H3 is still on the interior side of *bc*, and H4 is still on the exterior side. Because no vertex is moved, *a* still lies in H4. No edge crosses *av* (established above), thus no boundary separates *a* from the exterior of polygon2, *i.e.* *a* is outside of polygon2.

A similar argument holds for *v* ; this vertex is also outside polygon2. The only way for any other polygon1 vertex to be inside polygon2 is for a polygon1 edge to cross a polygon2 edge. Since this cannot happen, no vertex of polygon1 can be enclosed by polygon2.

It has been shown that no vertex of polygon1 is enclosed by polygon2, and that no vertex of polygon2 is enclosed by polygon1, and so the vertex convexity algorithm provides a reliable test for non-enclosure.

Note: throughout this discussion, it has been assumed that there is a space between vertices *b* and *u*. If this is not true, then edges *ab* and *uv* must overlap, or at least touch. In this case, since no edge crosses *ab* or *uv*, no edge will cross gap *bu*, and it would be unnecessary to test for such a crossing. In most cases, however, there will be a space between *b* and *u*, and it will be necessary to perform the test.

4.2.5.2.2. Non-Zero-Width Condition We will now justify the claim that the vertex convexity algorithm establishes the non-zero-width condition. (Review Figure 4-6 for an example of the production of a polygon containing a zero-width region.) Collinear edge heuristic edge rearrangement adds, deletes and moves no vertices. Only two edges are affected: *ab* and *uv*, which become *av* and *ub* after partitioning (Figure 4-1). As before, let polygon1 be the new polygon containing *a* and *v*, and polygon2 be the new polygon containing *u* and *b*.

As discussed before, the portion of *av* between *a* and *b* is equivalent to *ab*, and the portion between *u* and *v* is equivalent to *uv*. All that's new in polygon1 is the portion of *av* between *b* and *u*; this is the only place in polygon1 where edge rearrangement might introduce a zero-width region. To produce such a zero-width region, some part of the polygon boundary must touch *av* between *b* and *u*. But this part of *av* is equivalent to the pre-partitioning gap *bu*, and the vertex convexity algorithm prevents partitioning if any edge of the polygon intersects this gap. If partitioning has taken place, then no part of the polygon boundary intersects *av* between *b* and *u*, and no zero-width region is introduced into polygon1. (This is not to say that there could not have been any zero-width regions in the polygon before partitioning, only that partitioning does not *introduce* any zero-width regions.)

In polygon2, all that's new is edge *ub*. To produce a zero-width region, some part of the polygon boundary must touch this edge. But *ub* is equivalent to gap *bu*, and the same argument applies to polygon2 as to polygon1. Thus the vertex convexity algorithm establishes the non-zero-width condition as well as the non-enclosure condition.

4.2.6. An Algorithm For Non-Horizontal Partitioning

An algorithm has been developed for non-horizontal partitioning, based on the collinear edge heuristic. The test for collinearity is performed as described in Appendix 1. Edge traversal direction is tested as discussed above (using direction tags). The vertex convexity algorithm is employed to establish non-enclosure and non-zero-width. Each new polygon produced is recursively partitioned until no more non-horizontal partitions are possible, at which point it is partitioned horizontally (using the Newell and Sequin algorithm). (The switch to horizontal partitioning also provides the recursion escape mechanism.) The algorithm is as follows.

```
procedure partition(polygon)
begin
    for edge1 ← first edge to last edge of polygon do
    begin
        if edge1 is horizontal then iterate with next edge1;
        for edge2 ← edge after edge1 to last edge do
        begin
            if direction(edge1) ≠ direction(edge2)
            then iterate with next edge2;
            if edge1 and edge2 are not collinear
            then iterate with next edge2;
            a ← endpoint of edge1 farthest from edge2;
            b ← endpoint of edge1 closest to edge2;
            u ← endpoint of edge2 closest to edge1;
            v ← endpoint of edge2 farthest from edge1;
            if b is convex or u is convex
            then iterate with next edge2;
            for edge3 ← each edge of polygon (excluding
            edge1, edge2, and the edges adjacent to these) do
                if edge3 intersects gap between b and u
                then iterate with next edge2;
            edge1 ← edge av ;
            edge2 ← edge ub;
            polygon1 ← polygon containing edge1;
            polygon2 ← polygon containing edge2;
            partition(polygon1);
            partition(polygon2);
            return;
        end;
    end;
    horizontally_partition(polygon);
    return;
end;
```

4.3. EXPERIMENTAL

4.3.1. Context

The above algorithm has been implemented within the context of the "bifmebes" program described in Chapter 2. Procedure "partition" and associated procedures for determining collinearity, vertex convexity, *etc.* were written in C. Bifmebes was modified to pass each polygon to partition, instead of to the Newell and Sequin procedure; partition takes care of calls to the latter, as discussed above.

4.3.2. Testing

The revised program was tested using data contrived to exercise various aspects of the algorithm, and then with actual IC layout data. MEBES data generated was verifed by examination of the conversion log file, and by overlaying plots of pre- and post-conversion data, as before.

4.3.3. Results

In all cases, polygon partitioning was shown to be correct, and the output accurate to within the resolution of the MEBES machine.

Two versions of bifmebes, one with and one without non-horizontal partitioning (NHP), were run on data for several designs. The following table summarizes the differences observed.

Design	Trapezoids Generated			CPU Time (min:sec)		
	No NHP	With NHP	Change	No NHP	With NHP	Change
1	372	351	-5.6%	0:07	0:08	+14.3%
2	392	366	-6.6%	0:08	0:10	+25.0%
3	5851	5694	-2.7%	0:56	1:08	+21.4%
4	6022	5904	-2.0%	1:00	1:09	+15.0%
5	174790	165070	-5.6%	25:02	37:02	+47.9%
6	176229	169488	-3.8%	26:06	32:52	+25.9%
7	221403	218868	-1.1%	33:32	39:25	+17.5%
8	214713	211821	-1.4%	33:26	37:11	+11.2%

(Design 1 is a flipflop cell, 3 is an arithmetic logic unit, 5 is a multi-project chip, and 7 is a microprocessor. Designs 2, 4, 6 and 8 are the same as 1, 3, 5 and 7 respectively, except that each has been rotated through 90°.) Non-horizontal partitioning resulted in a trapezoid count reduction of 1% to 6%, accompanied by a CPU time increase of 11% to 48%.

4.4. DISCUSSION

The impact of the collinear edge heuristic on the total trapezoid count was disappointing, although not insignificant. It is clear that the magnitude of the trapezoid count reduction is dependent on the nature of the IC design.

The use of the vertex convexity algorithm is effective in establishing the non-enclosure and non-zero-width conditions. In fact, in some cases it represents some overkill. Consider the polygon in Figure 4-15.

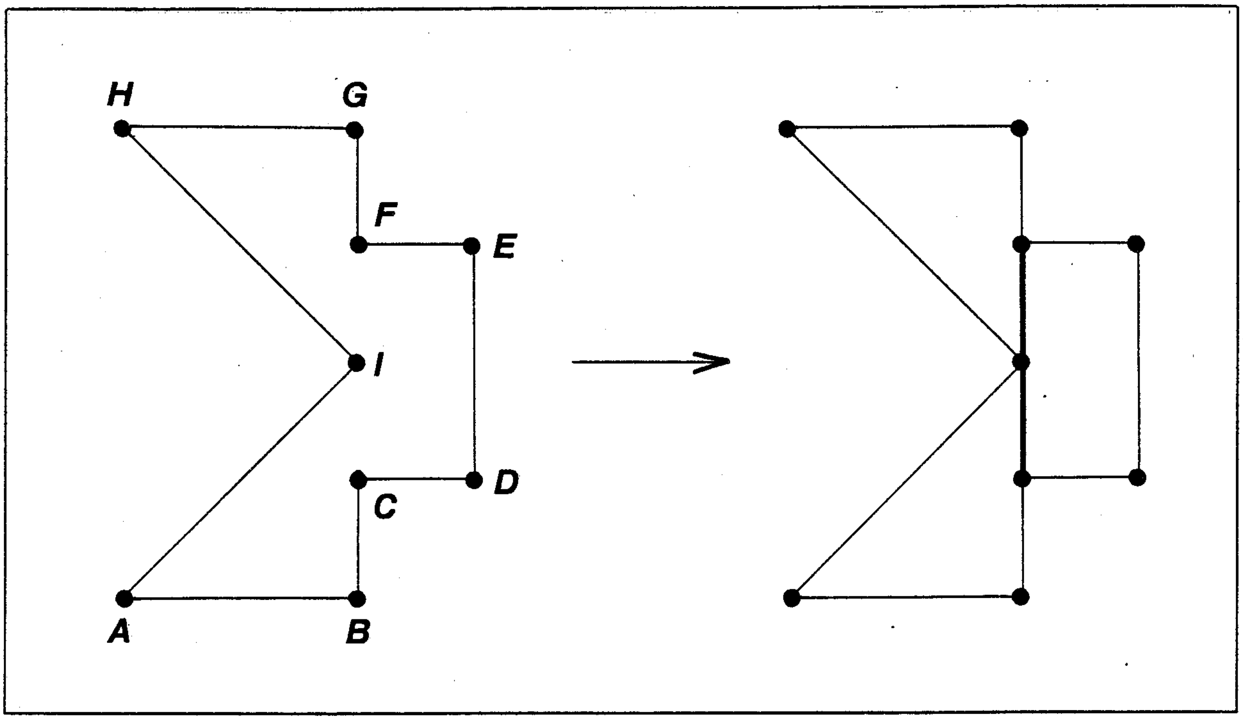

Figure 4-15 Legitimate Partition Rejected

Edges *BC* and *FG* satisfy all conditions for the collinear edge heuristic. However, because edges *HI* and *IA* intersect gap *CF*, the partition is rejected by the vertex convexity algorithm. Thus some cases for non-horizontal partitioning are missed, although probably not many. (Shapes such as that in Figure 4-15 are not common in IC layouts.)

4.4.1. Exclusion of Self-Crossing Polygons

The failure to handle self-crossing polygons is a definite disadvantage. This drawback can, however, be partially offset. One source of self-crossing polygons is the expansion of wires with self-touching center-line. Let *ab* and *uv* be two edges of a wire envelope polygon, with *b* closest to *uv*, and *u* closest to *ab* (Figure 4-16).

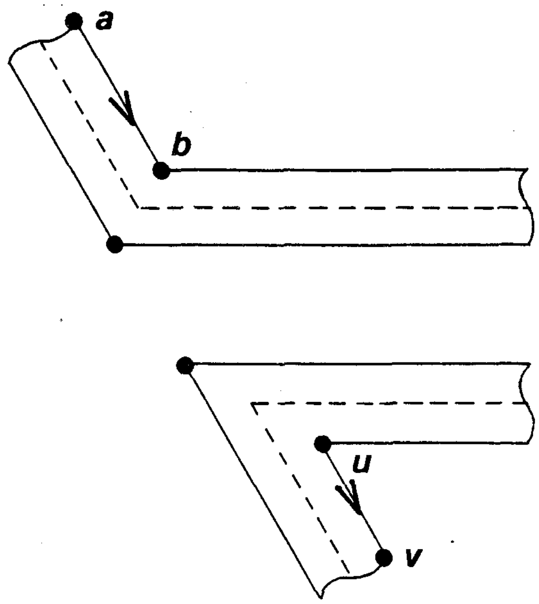

Figure 4-16 Wire Envelope

Let these edges be collinear and have the same traversal direction. If edge rearrangement is to be applied, then vertices *b* and *u* must both be concave. As a result, the other side of the wire envelope will cross gap *bu*, and the partition will be rejected. Because application of the collinear edge heuristic to wires would seldom (if ever) produce any partitions, little is lost by not even trying to apply it to wires. Thus, wires can be processed directly by the horizontal partitioning routine; non-horizontal partitioning is bypassed, and wires with self-touching centerlines will be handled correctly.

This, of course, does not solve the problem for self-crossing polygons in general. The approach is satisfactory for experimental purposes, but not for commercial applications. The program should check the data for self-crossing polygons as a first step, and issue a warning for each one found. This would make the implementation safe, but would still leave the task of dealing with self-crossing polygons to the designer.

The collinear edge heuristic is an experiment in e-beam data optimization. In view of its inability to handle self-crossing polygons, the small reduction in trapezoid count achieved, and the increased CPU time requirements, it is probably not suitable for commercial applications. Nonetheless, it does represent a step towards optimization of e-beam data.

Chapter 5

CONCLUSIONS

The objectives of this research (as set out in Chapter 1) were

(1) to develop software to prepare electron beam lithography mask data which is accurate and reliable, and

(2) to explore methods to optimize e-beam data so that e-beam machine time is reduced.

5.1. POLYGON PARTITIONING

The first objective necessitated the selection and implementation of a suitable algorithm to partition polygons into e-beam trapezoids. A number of previously developed algorithms have been reviewed, the most promising of which was one by Newell and Sequin, based on the winding number convention [6]. In spite of its intended generality, the Newell and Sequin algorithm suffers from several limitations. The refinements discussed in this thesis have overcome these limitations, producing an algorithm which is capable of partitioning arbitrary polygons, including all types of self-intersections.

5.2. BIFMEBES

"Bifmebes" represents an implementation of the refined Newell and Sequin algorithm within the context of a complete e-beam data preparation program. This program meets the objectives stated above: it is accurate and reliable, and imposes no unreasonable restrictions on the designer.

5.3. E-BEAM DATA OPTIMIZATION

Several possible techniques for optimizing e-beam data have been discussed. One method, which attempts to reduce the trapezoid count by the use of non-horizontal partitioning, has been implemented using the collinear edge heuristic. Application of this heuristic to test data produced a small but desirable reduction in the total trapezoid count, at the expense of an increase in computer time. (This increase takes place on the local computer used to prepare the data, not on the computer at the e-beam installation.) Although the collinear edge heuristic has only a small impact on the trapezoid count and is unable to handle self-crossing polygons, it does represent a beginning in the area of optimizing data for e-beam lithography systems.

5.4. RECOMMENDATIONS FOR FURTHER WORK

Two possible optimization techniques discussed in Chapter 3 bear further investigation: trapezoid merging and e-beam stage control. Merging of butting trapezoids would definitely reduce the total trapezoid count; by how much remains to be seen. But perhaps the greatest reduction in e-beam machine time (at least for multi-project masks) will be achieved by exercising more intelligent control over the sequence of rasterization and stage movement. Although this will not be possible until manufacturers of e-beam lithography systems provide for such control, it would be worthwhile to simulate its impact on mask fabrication costs.

In view of the flaws uncovered in all of the earlier polygon partitioning algorithms examined, it would be highly desirable to develop a proof for the refined Newell and Sequin algorithm. This project is being considered for further work, but will not be included in this thesis.

Appendix 1

COLLINEARITY

In this appendix we will address the problem of determining whether two polygon edges (or more generally, two line segments) are collinear.

First, consider the familiar form of the line equation:

$$y = mx + b$$

where m is the slope, and b is the y intercept. This form is not sufficiently general, since calculation of m runs into trouble if Δx is 0. A preferable form of the equation is:

$$Ax + By + C = 0$$

where the coefficents A, B and C may be found as follows:

$$A = y_2 - y_1$$

$$B = x_1 - x_2$$

$$C = x_2 y_1 - x_1 y_2$$

Here (x_1,y_1) and (x_2,y_2) are two unique points on the line.

THEOREM A1.1

Two line segments are collinear if and only if their line equation coefficients are proportional (*i.e.* if segment1 has coefficients A_1, B_1 and C_1, and segment2 has coefficients A_2, B_2 and C_2, then $A_1/A_2 = B_1/B_2 = C_1/C_2$).

Proof

We will first prove that collinearity is implied by proportional line equation coefficients. Let segment1, segment2, A_1, B_1, C_1, A_2, B_2 and C_2 be as above. Let $A_1/A_2 = B_1/B_2 = C_1/C_2 \equiv r$. Line segment2 has the equation

$$A_2 x + B_2 y + C_2 = 0$$

Multiply both sides by r :

$$rA_2x + rB_2y + rC_2 = 0$$

Substitute the value of r and simplify:

$$(A_1/A_2)A_2x + (B_1/B_2)B_2y + (C_1/C_2)C_2 = 0$$

$$A_1x + B_1y + C_1 = 0$$

which is just the equation for segment1. Segment1 and segment2 have the same line equation, so they must lie on the same line. Thus segment1 and segment2 are collinear.

We will now prove the converse. Given that segment1 and segment2 are collinear, we must prove that their line equation coefficients are proportional. Let A, B and C be the coefficients of the line which contains segment1 and segment2. Let segment1 have end points (x_{11},y_{11}) and (x_{12},y_{12}). The line equation for segment1 is

$$Ax + By + C = 0$$

Substitute the coordinates at (x_{11},y_{11}):

$$Ax_{11} + By_{11} + C = 0$$

Solve for y_{11}:

$$y_{11} = -(Ax_{11} + C)/B$$

Now repeat for end point (x_{12},y_{12}):

$$y_{12} = -(Ax_{12} + C)/B$$

The line equation coefficients for segment1 may be calculated from its end points, as follows.

$$A_1 = y_{12} - y_{11}$$

$$B_1 = x_{11} - x_{12}$$

$$C_1 = x_{12}y_{11} - x_{11}y_{12}$$

Substituting the values of y_{11} and y_{12} from above and simplifying yields

$$A_1 = (A/B)(x_{11} - x_{12})$$

$$B_1 = x_{11} - x_{12}$$

$C_1 = (C/B)(x_{11} - x_{12})$

Now repeat this exercise for segment2. Let segment2's end points be (x_{21},y_{21}) and (x_{22},y_{22}). The line equation for segment2 is

$Ax + By + C = 0$

Substitute the coordinates at (x_{21},y_{21}) and solve for y_{21}:

$y_{21} = -(Ax_{21} + C)/B$

Substitute the coordinates at (x_{22},y_{22}) and solve for y_{22}:

$y_{22} = -(Ax_{22} + C)/B$

Calculate the line equation coefficients for segment2 from its end points.

$A_2 = y_{22} - y_{21}$

$B_2 = x_{21} - x_{22}$

$C_2 = x_{22}y_{21} - x_{21}y_{22}$

Substitute the values of y_{21} and y_{22} from above and simplify.

$A_2 = (A/B)(x_{21} - x_{22})$

$B_2 = x_{21} - x_{22}$

$C_2 = (C/B)(x_{21} - x_{22})$

We are now in a position to calculate coefficient ratios for segment1 and segment2.

$A_1/A_2 = (A/B)(x_{11} - x_{12}) / ((A/B)(x_{21} - x_{22})) = (x_{11} - x_{12})/(x_{21} - x_{22})$

$B_1/B_2 = (x_{11} - x_{12})/(x_{21} - x_{22})$

$C_1/C_2 = C/B\,(x_{11} - x_{12}) / ((C/B)(x_{21} - x_{22})) = (x_{11} - x_{12})/(x_{21} - x_{22})$

Thus $A_1/A_2 = B_1/B_2 = C_1/C_2$, and the collinearity of segment1 and segment2 implies the proportionality of their line equation coefficients.

QED

Definition: a *normalized line equation* is one in which all coefficients have been divided by the coefficient with the smallest non-zero absolute value.

THEOREM A1.2

Two line segments are collinear if and only if their normalized line equations have identical coefficients (*i.e.* if segment1 has coefficients A_1, B_1 and C_1, and segment2 has coefficients A_2, B_2 and C_2, then $A_1 = A_2$, $B_1 = B_2$ and $C_1 = C_2$).

Proof

First we will prove that collinearity implies identical coefficients. Given collinear line segments segment1 and segment2; segment1 has line equation coefficients A_1, B_1 and C_1; segment2 has coefficients A_2, B_2 and C_2. Because segment1 and segment2 are collinear, they are contained by the same line; let that line's equation have coefficients A, B, and C, and let C be the one with smallest non-zero absolute value. (A similar argument holds for other cases.) Because segment1 and the line containing it are collinear, their coefficients are proportional (by Theorem A1.1). Thus

$$A_1/A = B_1/B = C_1/C \equiv r$$

$$A_1 = Ar$$

$$B_1 = Br$$

$$C_1 = Cr$$

From the above condition on C

$$|C| \leqslant |A|$$

Multiply both sides of this inequality by $|r|$ and simplify.

$$|C||r| \leqslant |A||r|$$

$$|Cr| \leqslant |Ar|$$

$$|C_1| \leqslant |A_1|$$

Similarly,

$$|C| \leqslant |B|$$

$$|C||r| \leqslant |B||r|$$

$$|Cr| \leqslant |Br|$$

$$|C_1| \leqslant |B_1|$$

Thus C_1 has the smallest absolute value of segment1's coefficients. Because $C_1 = Cr$, and because $C \neq 0$, C_1 can only be zero if r is zero. If $r = 0$, then A_1, B_1 and C_1 must all be zero, and segment1 is nothing more than a single point. Since there is little value in testing the collinearity of a point and a line, it is safe to assume that A_1, B_1 and C_1 cannot *all* be zero. Thus $r \neq 0$, and $C_1 \neq 0$. So C_1 is the coefficient which would be chosen to normalize segment1's coefficients. Let segment1's normalized coefficients be A_1^*, B_1^* and C_1^*.

$$A_1^* = A_1/C_1 = Ar/Cr = A/C$$

$$B_1^* = B_1/C_1 = Br/Cr = B/C$$

$$C_1^* = C_1/C_1 = 1$$

Now consider segment2; it and the line containing it are collinear, so by Theorem A1.1,

$$A_2/A = B_2/B = C_2/C \equiv s$$

$$A_2 = As$$

$$B_2 = Bs$$

$$C_2 = Cs$$

From the condition imposed on C

$$|C| \leqslant |A|$$

$$|C| \leqslant |B|$$

Multiply both sides of each inequality by $|s|$ and simplify.

$$|C||s| \leqslant |A||s|$$

$$|Cs| \leqslant |As|$$

$$|C_2| \leqslant |A_2|$$

$$|C||s| \leqslant |B||s|$$

$$|Cs| \leqslant |Bs|$$

$$|C_2| \leqslant |B_2|$$

By a similar argument to that used above, $s \neq 0$. And since $C \neq 0$, $C_2 \neq 0$. Thus C_2 is the coefficient of segment2 with minimum absolute value, and would be used to normalize segment2.

$$A_2^* = A_2/C_2 = As/Cs = A/C$$

$$B_2^* = B_2/C_2 = Bs/Cs = B/C$$

$$C_2^* = C_2/C_2 = 1$$

Thus

$$A_1^* = A_2^*$$

$$B_1^* = B_2^*$$

$$C_1^* = C_2^*$$

We have demonstrated that the collinearity of segment1 and segment2 implies that their normalized line equation coefficients are identical. Now we will prove the converse.

Given segment1 with normalized line equation coefficients A_1^*, B_1^* and C_1^*, and segment2 with normalized line equation coefficients A_2^*, B_2^* and C_2^*, such that

$$A_1^* = A_2^*$$

$$B_1^* = B_2^*$$

$$C_1^* = C_2^*$$

Let segment1's unnormalized coefficients be A_1, B_1 and C_1. Let segment2's unnormalized coefficients be A_2, B_2 and C_2. Let

$$C_1^* = C_2^* = 1$$

(A similar argument holds for the other cases.) Then C_1 is the coefficient used to normalize segment1, and C_2 is used to normalize segment2.

$$A_1^* = A_1/C_1$$

$$A_2^* = A_2/C_2$$

$$A_1/C_1 = A_2/C_2$$

$$A_1/A_2 = C_1/C_2$$

$$B_1^* = B_1/C_1$$

$$B_2^* = B_2/C_2$$

$$B_1/C_1 = B_2/C_2$$

$$B_1/B_2 = C_1/C_2$$

$$A_1/A_2 = B_1/B_2 = C_1/C_2$$

Segment1 and segment2 have proportional line equation coefficients, and by Theorem A1.1, they are collinear.

QED

Theorem A1.2 is very useful for testing the collinearity of polygon edges. The normalized line equation coefficients can be calculated once for each edge and stored. Subsequent tests for collinearity are accomplished simply and quickly by comparing normalized coefficients.

Appendix 2

VERTEX CONVEXITY

In this appendix we will discuss methods for determining whether a polygon vertex is convex or concave.

Definition: consider a point which is traversing the boundary of a polygon in such a way that each vertex is passed only once. The *turning direction* of a vertex is the direction (left or right) in which that point must turn as it passes the vertex.

Definition: a *minimum y vertex* is a polygon vertex whose y coordinate is less than or equal to that of any other vertex of the same polygon.

Definition: a *minimum yx vertex* is a minimum y vertex whose x coordinate is less than or equal to that of any other minimum y vertex of the same polygon.

Definition: consider polygon vertex *b*, with adjacent vertices *a* and *c*. Let m_1 be the absolute value of the slope of edge *ab*. Let m_2 be the absolute value of the slope of edge *bc*. The *least slope magnitude* of *b* is the lesser of m_1 and m_2.

Definition: if a polygon has only one minimum yx vertex, then it is the *reference vertex* of the polygon. If there is more than one minimum yx vertex, then the one with the minimum least slope magnitude is the reference vertex.

In the research discussed in Chapter 4, the method used to determine if a polygon vertex is convex or concave is based on the following concept: within a given polygon,

(1) all convex vertices have the same turning direction,

(2) all concave vertices have the same turning direction, and

(3) the turning direction of convex vertices is opposite to that of concave vertices.

The polygon is first "cleaned up" by eliminating any zero-length edges, and merging any adjacent collinear edges. The reference vertex is then found, and its turning direction determined. (Turning direction is found by comparing the unit vectors of the edges adjacent to the vertex.) The reference vertex is always convex, thus any vertex can be tested by comparing its turning direction to that of the reference vertex. If the directions are the same, then the vertex is convex, otherwise it is concave.

A proof of this method has been developed, but it is not presented here in view of the fact that a superior test for convexity was discovered after the Chapter 4 research had already been completed. The latter test, which is based on a polygon convexity algorithm by Shamos [11], is superior because of its simplicity, and because the computations performed are expected to require less machine time. In any further work requiring a test for vertex convexity, this would certainly be the preferred method, and so it is presented in some detail here.

Definition: a simple polygon is in *standard form* if its vertices are traversed in a counterclockwise direction, zero-length edges have been eliminated, adjacent collinear edges have been merged, and traversal begins with the minimum yx vertex.

The traversal direction of a polygon of n vertices is found by calculating the *signed area* using the following algorithm.

```
signed_area ← 0;
prev ← n;
for i = 1 to n do
begin
   next ← modulo(i,n) + 1; /* modulo(i,n) = i modulo n */
   signed_area ← signed_area + (x[i] * (y[next] — y[prev]));
   /*
     x[i] = x coordinate of vertex i
     y[i] = y coordinate of vertex i
   */
   prev ← i;
end;
signed_area ← signed_area / 2;
```

The traversal direction is counterclockwise if the signed area is positive, and clockwise if it is negative.

Each edge of a polygon has a length and a direction (its traversal direction) and can thus be represented by a vector.

Definition: an *edge angle* is the angle between the x axis and the vector representing the edge, measured in the conventional sense (counterclockwise). For example, in Figure A2-1, α is the edge angle for AB, and β is the edge angle for DA.

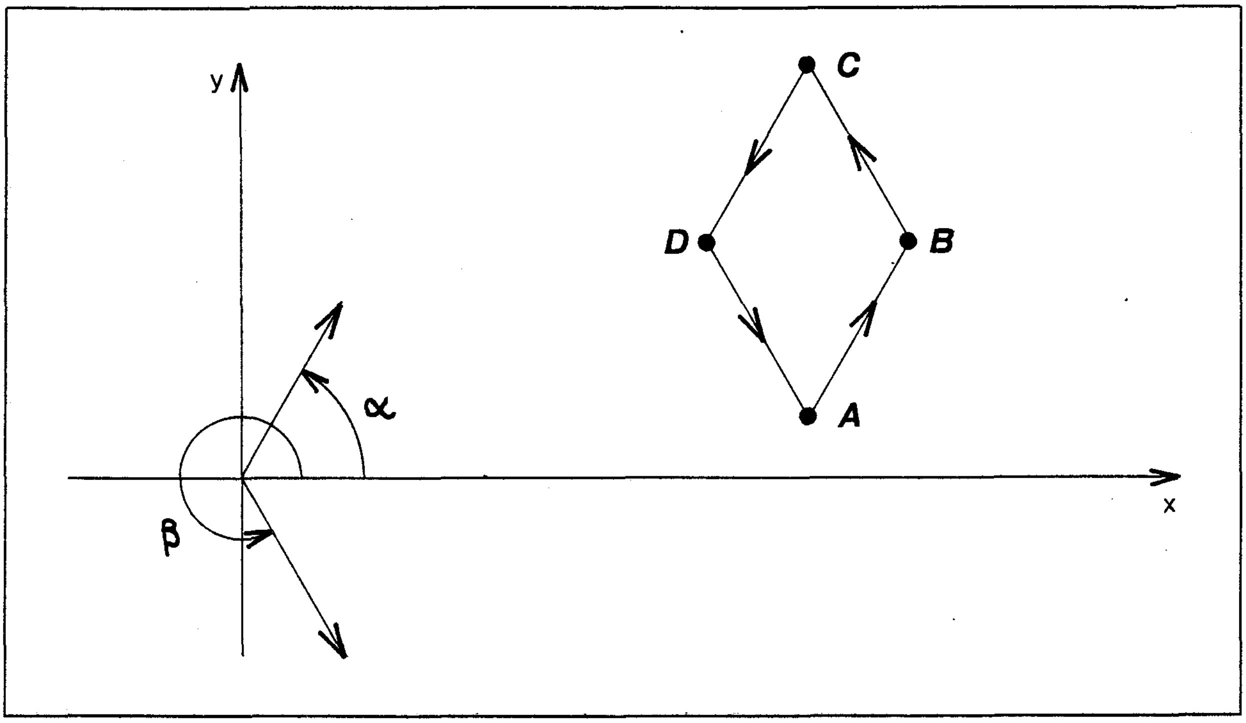

Figure A2-1 Edge Angle.

In Theorem 2.4 of [11], Shamos proves the following: a polygon is convex iff in standard form its edge angles are non-decreasing. For example, the polygon in Figure A2-2(a) has non-decreasing edge angles ($\theta_1 < \theta_2 < \theta_3 < \theta_4 < \theta_5$) and the polygon is convex.

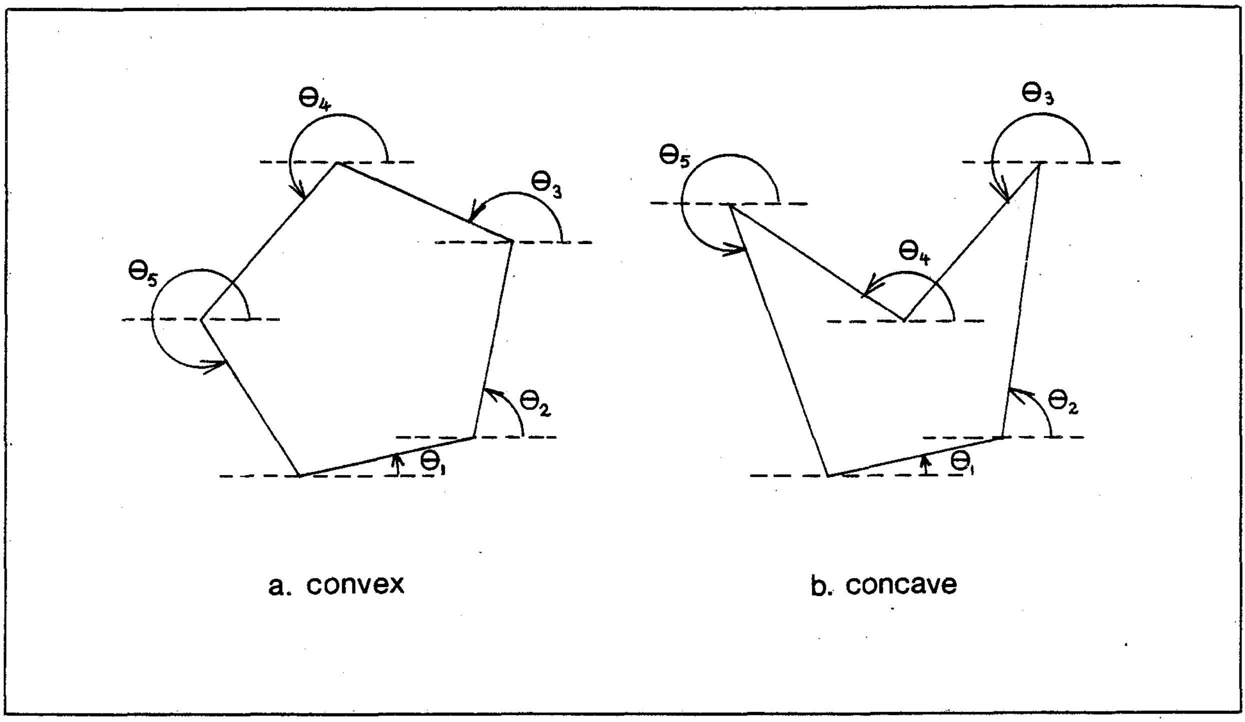

Figure A2-2 Convex and Concave Polygons

In contrast, the polygon in Figure A2-2(b) has one decreasing edge angle ($\theta_3 > \theta_4$) and the polygon is concave.

In a convex polygon, all vertices must be convex, and so Shamos' theorem readily leads to a test for a single vertex. Another way of viewing the theorem is: a polygon is convex iff in standard form the edge angles are non-decreasing for each pair of adjacent edges. Each pair of adjacent edges intersect at a polygon vertex, and so we have the following: a polygon vertex v is convex iff, with the polygon in standard form, the edges adjacent to v have non-decreasing edge angles. For example, refer to Figure A2-3.

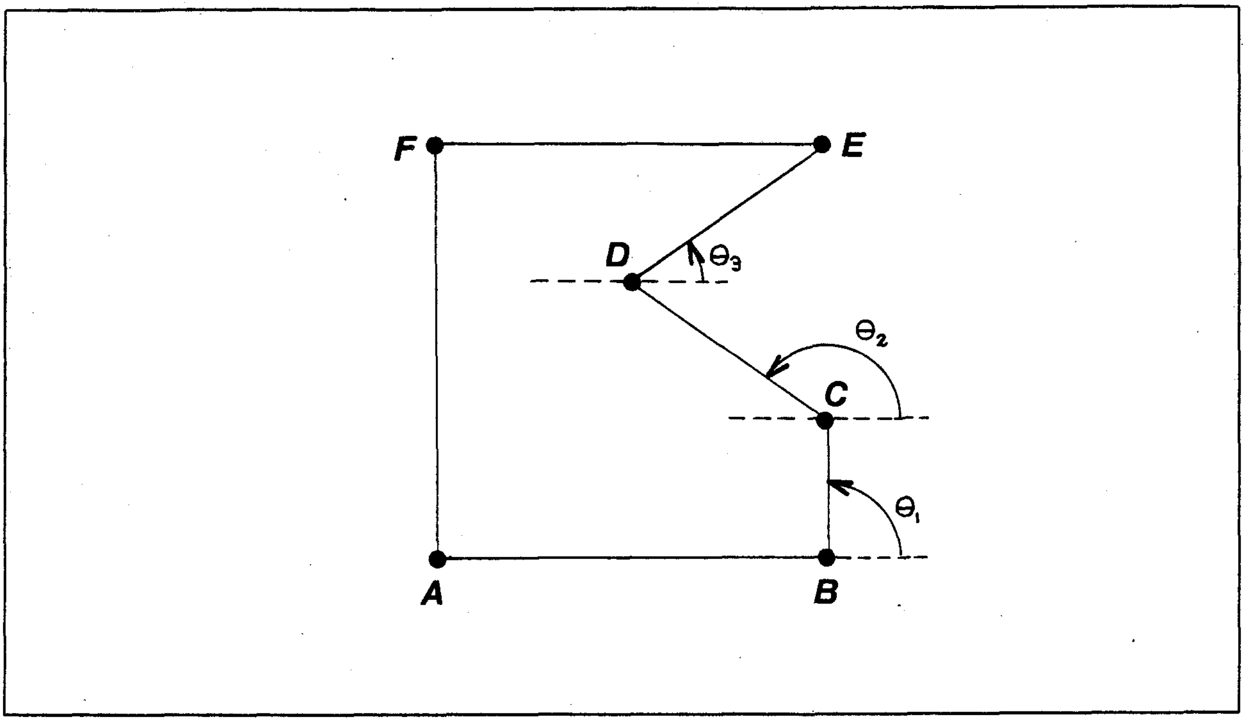

Figure A2-3 Testing a Single Vertex

Vertex *C* has adjacent edges *BC* and *CD*, with edge angles θ_1 and θ_2, respectively. Since $\theta_1 < \theta_2$, *C* is convex. Vertex *D* has adjacent edges *CD* and *DE*, with edge angles θ_2 and θ_3, respectively. Since $\theta_2 > \theta_3$, *D* is concave.

As Shamos points out in Chapter 3 of [11], it isn't necessary to actually compute edge angles to compare them. Consider polygon edges *AB* and *BC* (Figure A2-4); we wish to compare their edge angles.

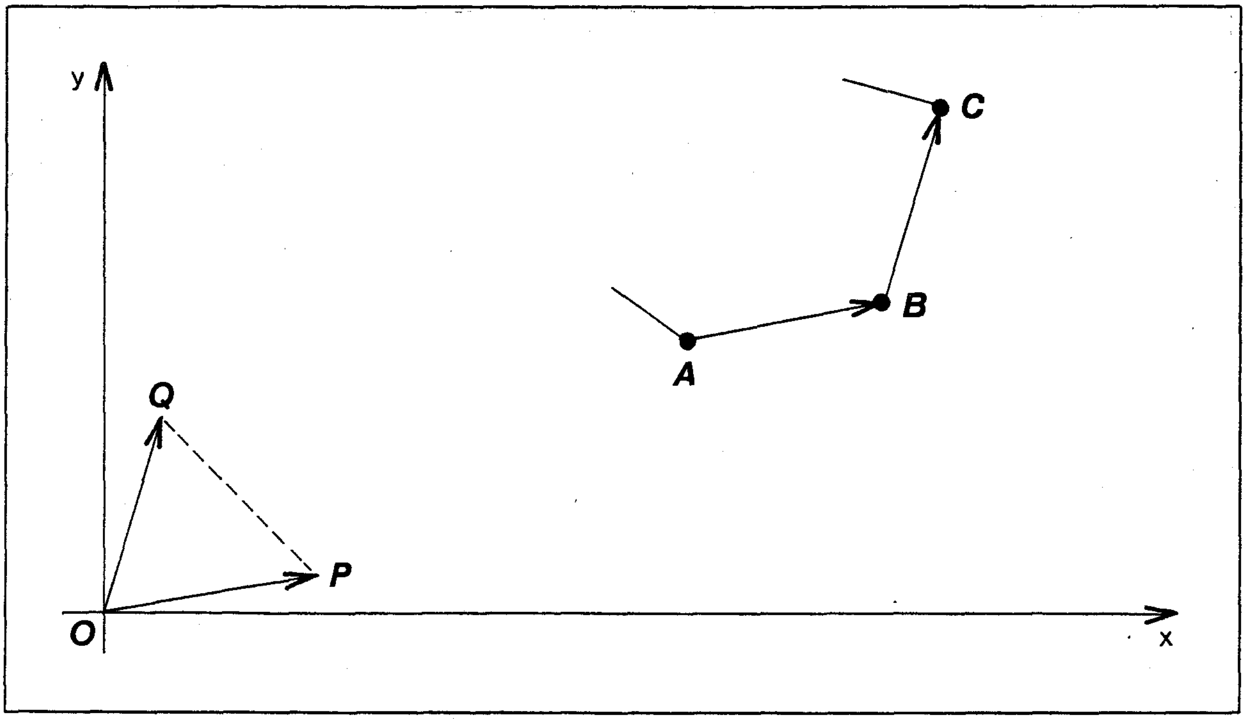

Figure A2-4 Edge Angle Comparison

Translate the vectors representing these edges such that each has its tail at the origin (*O*). Let *P* be the point at the head of *AB*'s vector, and *Q*, the point at the head of *BC*'s vector (Figure A2-4). Evaluate the signed area of triangle *OPQ*. If it is positive, then the triangle is traversed counterclockwise, and *OQ*'s edge angle is greater than that of *OP*; thus *BC* has a greater edge angle than *AB*. If the signed area is negative, then *AB*'s edge angle exceeds that of *BC*. Thus a few simple calculations suffice to compare edge angles; trigonometric functions are not necessary.

(Note that the order of traversal of the triangle was arbitrarily chosen to be *OPQ*. Any order is fine, as long as the interpretation of the signed area is consistent with the order chosen.)

REFERENCES

[1] R.A. Harris, "Data Processing For The Electron Beam Machine", *M.Sc. thesis*, Electrical Engineering Department, University of Manchester, UK (1970)

[2] D.R. Herriott, "Electron Beam Lithography", *Journal of Vacuum Science and Technology*, vol. 20, pp. 781-785 (1982)

[3] W.D. Little and R. Heuft, "An Area Shading Graphics Display System", *IEEE Transactions On Computers*, vol. C-28 no. 7, pp. 528-531 (1979)

[4] C. Mead and L. Conway, *Introduction To VLSI Systems*, Addison-Wesley, Philippines (1980) pp. 115-127

[5] R.D. Moore, "EL Systems: High Throughput Electron Beam Lithography Tools", *Solid State Technology*, vol. 26 no. 9, pp. 127-132 (1983)

[6] M.E. Newell and C.H. Sequin, "The Inside Story On Self-Intersecting Polygons", *Lambda*, pp. 20-24 (Second Quarter, 1980)

[7] O.W. Otto, "EBCAD - Fully Integrated Pattern Data Processing For Direct Write Electron-Beam Lithography Systems", *Journal of Vacuum Science and Technology*, vol. 19, pp. 993-997 (1981)

[8] K. Patel, "Computer-Aided Decomposition Of Geometric Contours Into Standardized Areas", *Computer-Aided Design*, vol. 9 no. 3, pp. 199-203 (1977)

[9] P. Petric and O. Woodard, "Direct-Write Electron Beam System", *Solid State Technology*, vol. 29 no. 9, pp. 154-160 (1983)

[10] H.C. Pfeiffer, "Recent Advances In Electron-Beam Lithography For The High-Volume Production Of VLSI Devices", *IEEE Transactions On Electron Devices*, vol. ED-26 no. 4, pp. 663-674 (1979)

[11] M.I. Shamos, "Computational Geometry", *Ph.D. thesis*, Yale University (1978)

BIBLIOGRAPHY

[1] E. Munro, "Electron Beam Lithography", *Advances in Electronics and Electron Physics*, Supplement 13B, pp. 73-131 (1980)

[2] J.A. Reynolds, "An Overview of E-Beam Mask Making", *Solid State Technology*, pp. 87-94 (August 1979)