

UNIVERSITY OF SOUTH ALABAMA
SCHOOL OF COMPUTER & INFORMATION SCIENCES

EFFICIENT RENDERING OF THICK POLYLINES

BY

Subramani Swaminadhan

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

December 2003

Approved:

Date:

Chair of Thesis Committee: Dr. Thomas F. Hain

Member of Committee: Dr. David D. Langan

Member of Committee: Dr. Stephen G. Brick

Dean of School of Computer & Information Sciences: Dr. David L. Feinstein

Director of Graduate Studies: Dr. Roy J. Daigle

Dean of Graduate School: Dr. Judy P. Stout

EFFICIENT RENDERING OF THICK POLYLINES

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

by
Subramani Swaminadhan
B.E., Bangalore University, 1998
December 2003

Dedicated to my parents,
Mrs. Lakshmi Swaminadhan and Mr. Swaminadhan Sundaram,
for their patience, belief and full support in my actions.

ACKNOWLEDGMENTS

I wish to acknowledge the support of my thesis chair, Dr. Thomas Hain, for he introduced me to the exciting world of graphics and substantiated this introduction with a lot of support in the form of ideas and code samples, pointing me to the right direction every time I made a wrong turn.

I wish to express my gratitude to Dr. David Langan and Dr. Stephen Brick for being a part of the committee and helping me in preparing the material. Their valuable collective feedback helped me in building my confidence during the thesis defense.

I also thank Dr. Roy Daigle, for he introduced me to the concept of an organized scientific research study. His initial reviews extensively helped me in formulating my ideas in a concise and methodical manner.

I acknowledge the help of many of my peers, most importantly Ms. Lavanya Subramaniam for helping me in understanding the problem and approaching the solution in more than one way, and for helping me in completing the documentation.

I would like to thank my family and friends for their support in my work and motivation in getting the research completed. Whenever I felt defeated by an error in the problem, they tried to understand and suggest ways to overcome it. I am eternally grateful for being in this company of family and friends.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	vi
ABSTRACT	viii
CHAPTER 1 INTRODUCTION	1
1.1 Introduction	1
1.1.1 Current Print Process	1
1.2 The Environment	3
1.2.1 Rendering of Polylines	3
1.2.2 The Proposed Idea	4
1.3 Basic Definitions	5
1.4 The Proposed Approach	7
1.4.1 Advantages of the New Approach	8
1.4.2 The Rendering Process	9
CHAPTER 2 BACKGROUND RESEARCH.....	12
2.1 Introduction	12
2.1.1 Intersecting Line Segments	12
2.1.2 Path Tracing	13
2.1.3 Polygon Union.....	13
2.1.4 Clipping Techniques	14
2.1.5 Trivial Rejection.....	15
2.2 Line Segment Attributes	16
2.2.1 End Caps	16
2.2.2 Join Considerations	17

CHAPTER 3 METHODOLOGY	19
3.1 Abstracting the Problem.....	19
3.2 Techniques Employed for Creating Polygonal Outline	19
3.2.1 Linear Selection.....	19
3.2.2 Divide-and-Conquer.....	20
3.2.3 Page Partition	21
3.3 Hypothesis.....	21
3.4 Visual Testing Tool.....	22
CHAPTER 4 THEORETICAL DEVELOPMENT	24
4.1 Proposed Algorithms.....	24
4.1.1 Outline of the Divide-and-Conquer Algorithm	24
4.1.2 Outline of the Page Partition Algorithm	27
4.2 The Linear Selection Algorithm	30
4.2.1 Near Line Segments Check.....	33
4.2.2 Preliminary Overlap Polygon.....	33
4.2.3 End Extension Check	34
4.2.4 Bevel Joins	35
4.2.5 Short Line Segments	35
4.2.6 Inline Optimizations.....	35
4.3 Hain’s Complex Polygon Decomposition Algorithm.....	36
CHAPTER 5 CONCLUSION	37
5.1 Summary of Current Work.....	37
5.2 Future Research.....	38
REFERENCES	40
APPENDICES	42
Appendix A Code.....	42
Appendix B Invoking the Algorithm	75
BIOGRAPHICAL SKETCH	78

LIST OF FIGURES

	Page
Figure 1 The rendering process [2]	2
Figure 2 Rendering overlapping objects.	4
Figure 3 The research idea.	4
Figure 4 A line segment and a thick line segment.	5
Figure 5 Thick polyline, outline before and after joins.	6
Figure 6 Types of polygons.....	7
Figure 7 Advantage of the new algorithm.....	8
Figure 8 Disadvantage of the new algorithm.	9
Figure 9 Existing method for rendering a thick path.	10
Figure 10 New method for rendering a thick path.	10
Figure 11 Path tracing.	13
Figure 12 Polygon union types.....	14
Figure 13 Clipping	15
Figure 14 Trivial rejection.....	16
Figure 15 End caps.....	17
Figure 16 Thick line joins.	18
Figure 17 The miter limit ([1] Pg. 509).....	18
Figure 18 The Divide-and-Conquer algorithm.....	26

Figure 19 The Page Partition algorithm.	29
Figure 20 The Linear Selection algorithm.	31
Figure 21 Types of end cap.	32
Figure 22 Left and right intersection points.	33
Figure 23 Near line segments check.	33
Figure 24 Preliminary overlap polygon with construction.	34
Figure 25 Inline optimizations.	36
Figure 26 Visual polyline editor tool.	38

ABSTRACT

Swaminadhan, Subramani, M.S., University of South Alabama, December 2003.
Efficient Rendering of Thick Polylines. Chair of Committee: Dr. Thomas F. Hain.

This research creates an object-precision algorithm to render thick polylines. Given a polyline, and a thickness, we wish to generate a set of polygons representing the outline of the thick polyline. This set of polygons could then be processed into a display list by extant trapezoidalization or triangulation algorithms. We believe that the algorithm should have an advantageous performance compared to existing methods which create a set of rectangles from thick polyline segments, and generate separate fill polygons for corners and end caps, and often result in overlapping areas that need to be rendered multiple times. Application domains include rendering of graphical objects on raster displays and on the printed page.

CHAPTER 1

INTRODUCTION

1.1 Introduction

In the following, the current print process and computing environment will be outlined, giving a context and motivation for the research described later.

1.1.1 Current Print Process

Modern printers are capable of creating images from a collection of graphical objects of types such as bitmaps, characters, lines and curves. Input to the printer may be a high-level description of the document expressed in a page description language (PDL) such as Adobe[®] PostScript[®] [1] or PCL. The printer is composed of a controller, and a print engine. The controller has two software components: the Language Level (LL) component interprets the PDL; the Graphics Library (GL) generates a memory (scan buffer) image of the raster (a grid of pixels placed on paper) from the interpretation of the LL output. This raster is then laid out in ink on paper by the print engine. The GL, which is the domain of interest in this research, generates a set of pixels in the scan buffer corresponding to each of the graphical objects described in the PDL. This task is extremely compute-intensive, and must be very efficient.

In fact, rather than the GL producing a bit map directly, it produces an intermediate structure, called a Display List (DL), composed of primitive graphical components (line segments, trapezoids, and—of little relevance here—bitmaps) which are a decomposition of the graphical objects described in the PDL. The DL has the advantages that it is a highly compressed version of the image, and that it can be efficiently converted into the page image by a software (or firmware) process called the Raster Image Processor (RIP) or the rasterizer. Compression is useful since a decomposed page can be saved in a relatively small buffer between repeated printing of a page during multiple printing of a multi-page document. The alternative of regenerating the page image (from its PDL form) each time it is needed would be much too computationally expensive.

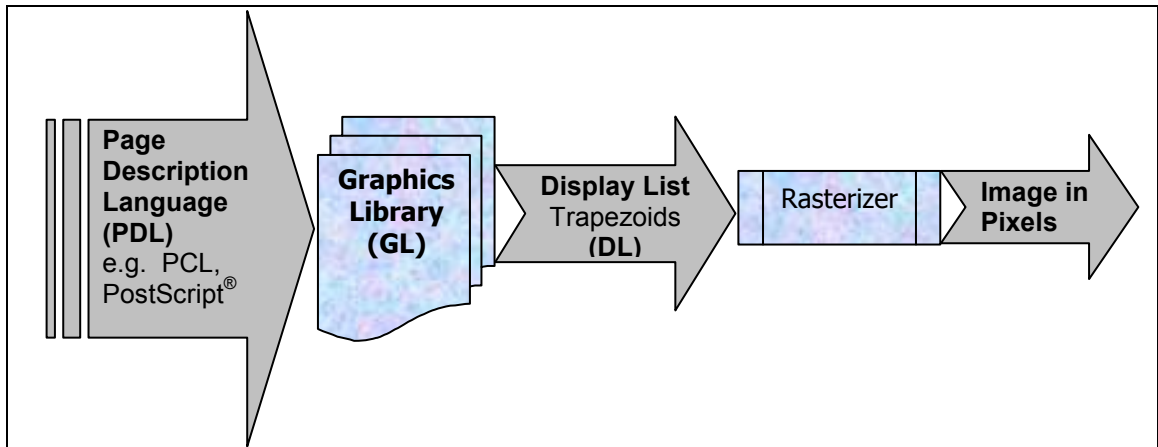


Figure 1 The rendering process [2]

A typical print process pipeline showing the GL and the rasterizer is detailed in Figure 1. PDL data is sent to the printer. The GL converts the data into a list of DL objects (lines and trapezoids), and the rasterizer converts the DL objects into a scan

buffer that will be sent to the print engine. The whole process of converting a page described by PDL into a set of pixels on a page is known as rendering.

1.2 The Environment

This research follows the PostScript model for page description. PostScript is a programming language that describes the appearance of text, graphical shapes and images on printed pages as mathematical shapes and curves. Since the page appearance is a mathematical description, the language is resolution-independent where resolution depends on the target device that renders the page. The PostScript language interpreter behaves like a state machine. The states involved in describing line segments include thickness, stroke settings, end caps and joins. Paths that are described by PostScript might contain line segments and/or curves, and may be discontinuous. Such paths are either “stroked” or, if they are closed, can be “filled”. Different area regions defined by the filling or thick stroking of a path might overlap each other arbitrarily.

Curved elements of a path are typically “flattened” by converting them to a set of connected line segments (polylines), which are then rendered using the current display state. Thus, polyline paths play an important role during the rendering of many shapes.

1.2.1 Rendering of Polylines

A polyline is a collection of line segments described directly in a PDL, or as a result of the flattening of curves described in the PDL. This research focused specifically on the problem of rendering stroked polylines given a (nonzero) thickness state.

The current method used in GLs (in GhostScript [10], or in Minolta-QMS firmware[13]) to render thick polylines is by repetitive application of thick line functions together with application of end cap and join corrections, rather than a single application of a polyline function. A generic example of rendering overlapping thick lines is shown in Figure 2

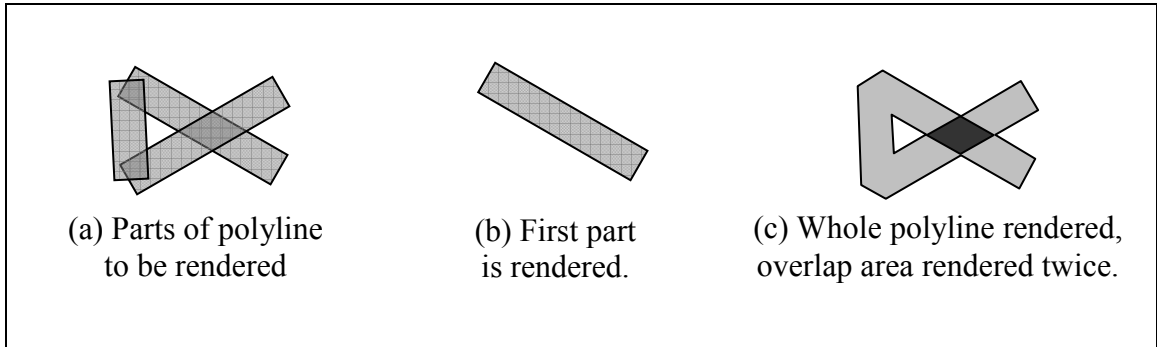


Figure 2 Rendering overlapping objects.

1.2.2 The Proposed Idea

To overcome the problem of rendering overlapping areas multiple times, the objects that overlap can be converted into a polygon and then the GL can process the polygon efficiently. This is the idea behind this research (Figure 3).

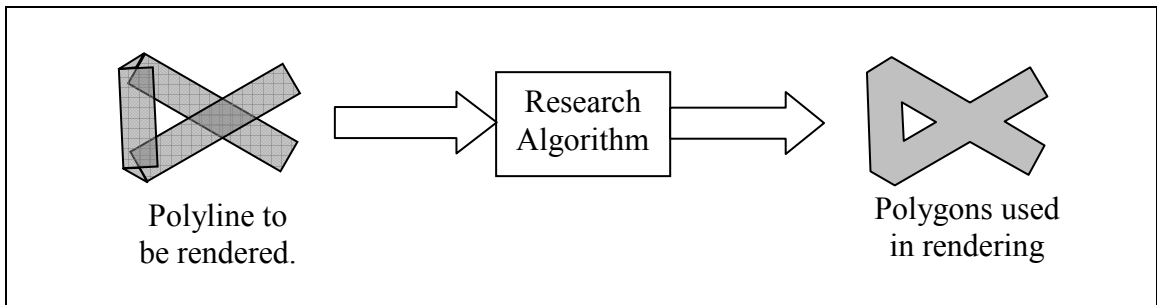


Figure 3 The research idea.

1.3 Basic Definitions

In this section, the reader is introduced to terminology used in this document, with figures being used for further elucidation.

Vertex: A point represented as $[x,y]$ co-ordinates in 2-dimensional space.

Line: A zero-width, infinite length set of points passing through two non-coincident points in space.

Line segment: A finite section of the infinite line that originates at a vertex and terminates at another vertex. (Line segment \overline{pq}).

Path: The PostScript description for a collection of mathematical line segments and curves. A path may be *stroked* (rendered with a predefined thickness and style) or, if closed, *filled* (with a color and/or pattern).

Thick line segment: A path with an associated thickness t . Thickness is manifested by filling an area within $t/2$ on either side of the line.

Thick line outline: The four line segments that border the thick line. (Segments \overline{AB} , \overline{BC} , \overline{CD} , and \overline{DA} in Figure 4)

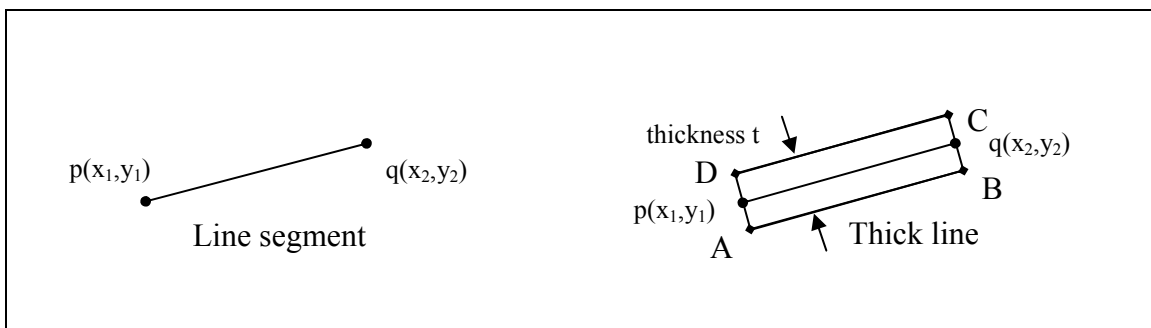


Figure 4 A line segment and a thick line segment.

Thick line outline edge: One specific line segment in the thick line outline.

Polyline: A set of line segments.

Thick polyline outline: A collection of polygons (Figure 5 – b, c)

Polygon: A closed figure made up of three or more line segments. While all polylines are polygons, all polygons are not necessarily polylines.

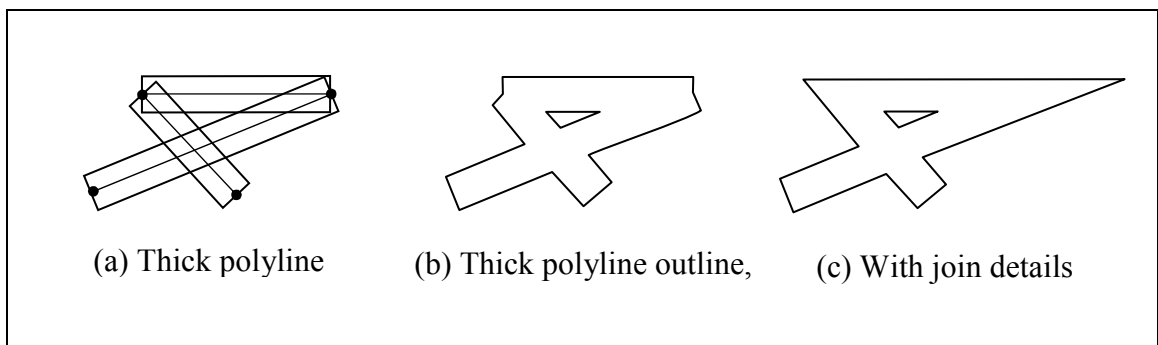


Figure 5 Thick polyline, outline before and after joins.

Polygon edge: One of the line segments that form the boundary of the polygon.

Polygon types: Polygons can be classified into two types. Simple and Complex.

Simple polygon: A polygon in which the edges do not cross each other.

Simple polygon types: Simple polygons can be further classified into Convex and Concave polygons.

Convex polygon: A simple polygon in which all internal angles are less than 180° .

Concave polygon: A simple polygon in which at least one internal angle is greater than 180° . (Figure 6, Concave polygon)

Complex polygon: A polygon in which at least two edges cross each other.

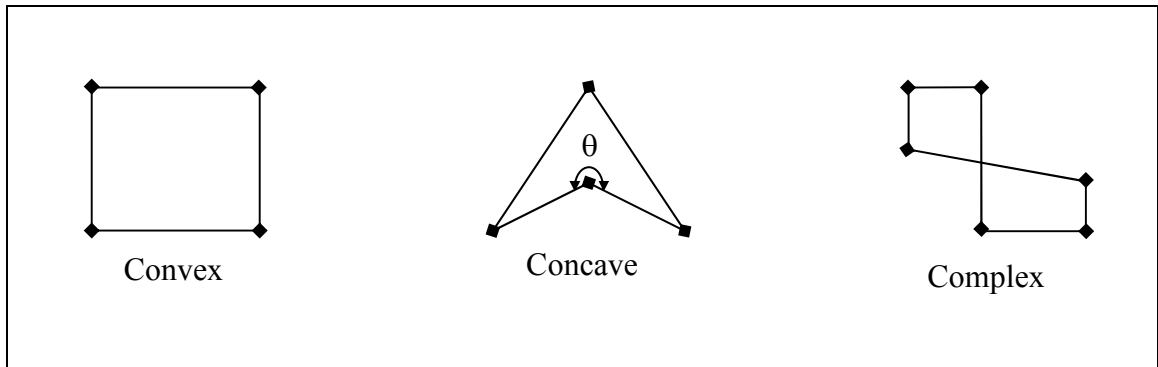


Figure 6 Types of polygons.

Image precision algorithm: An algorithm that treats graphical objects as a set of pixels and manipulates these pixels to alter the object's properties. These algorithms are used by the rasterizer for displaying the image on the selected device.

Object precision algorithm: An algorithm that mathematically treats graphical objects, independent of their display characteristics. As these algorithms do not depend on pixel-level manipulation, the resulting resolution of the graphical object will depend on the capabilities of the target device.

1.4 The Proposed Approach

The Graphics Library is optimized to handle basic page objects like bitmaps and characters, but is not optimized for overlapping objects like polylines (Figure 2). Polylines play an important role in systems such as Graphical Information Systems (GIS) and Printed Circuit Board (PCB) manufacturing. In these situations, any improvement of rendering times is desirable. The current research mathematically generates the outline polygon (Figure 3) of thick polylines, which can then be input to the trapezoidalization function [6] in the GL (which would break up the polygon into trapezoids within the DL).

1.4.1 Advantages of the New Approach

We believe that, for a given thick polyline, the research algorithm can create a display list of non-overlapping trapezoids faster than the corresponding line algorithms in a graphics library such as GhostScript. In many cases, the number of trapezoids generated for the DL will be smaller than by the traditional method. This is true as joins will be incorporated into the outline polygon instead of an adjustment polygon (with resulting trapezoids) having to be created. A typical advantageous situation and a disadvantageous situation is represented in Figure 7 and Figure 8 respectively. On the whole, we believe the research algorithm will produce a smaller DL as join and end cap adjustments are more common in polyline applications than individual segments crossing over each other [8].

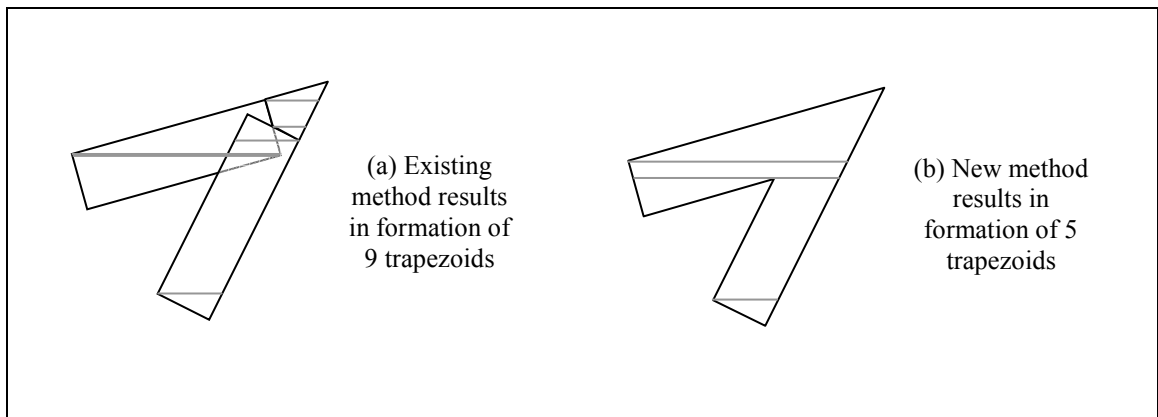


Figure 7 Advantage of the new algorithm.

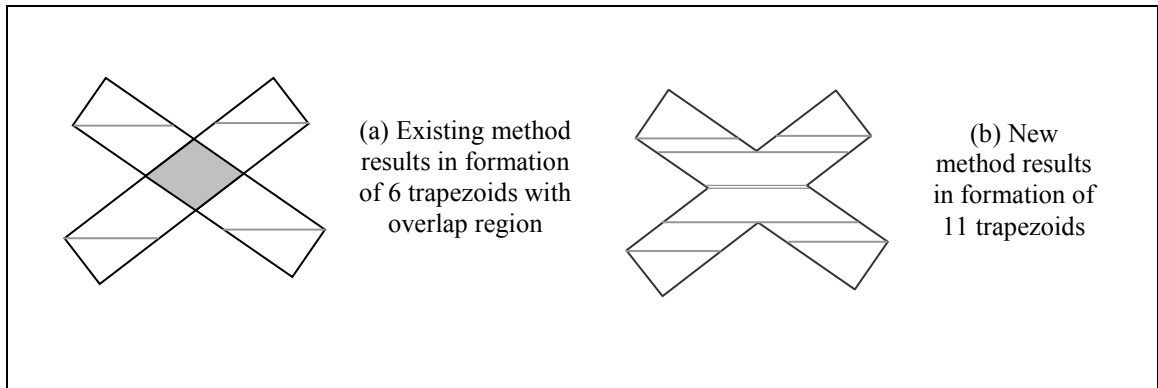


Figure 8 Disadvantage of the new algorithm.

The new algorithm mathematically creates a polygonal outline of a thick polyline, which could then be fed into a trapezoidalization function (typically extant in a GL) to produce the set of disjoint trapezoids. This approach would be more efficient than the current approach because, once the polyline is converted to polygonal outlines (i.e., polygons), the GL will have fewer page objects to work with and will not waste time rendering overlap regions (GL is optimized to render polygons). We attempt to leverage the ability to reuse mathematical results of computations involving parallel pairs of line segments.

1.4.2 The Rendering Process

Figure 9 illustrates the typical process of rendering thick polylines. The line details for each segment are input into a thick line algorithm in GL. The resultant outline polygons are merged, and corner and end cap details are fixed. This is then processed by a trapezoidalization algorithm resulting in a display list of primitives (see Figure 7a and

Figure 8a). This display list (Display List 1) is sent to the Raster Image Processor (RIP) which transfers the display list onto a print media, such as paper.

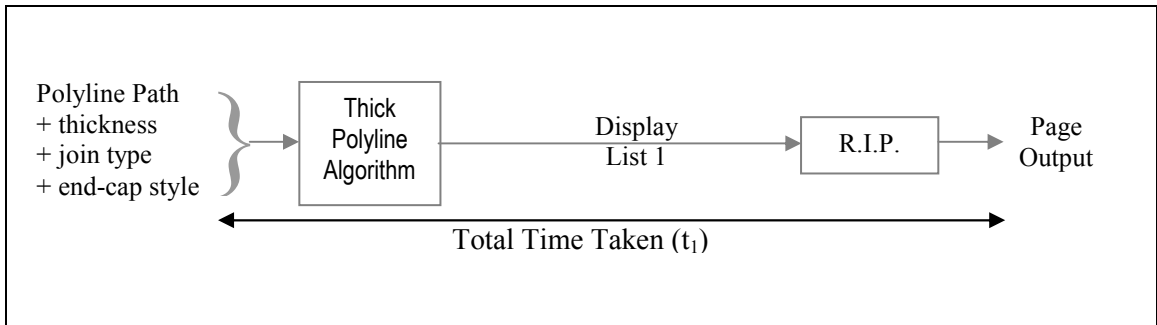


Figure 9 Existing method for rendering a thick path.

Figure 10 illustrates the developed algorithm's differences in rendering thick lines. The line details are input into the new algorithm instead of a typical line processing algorithm in the GL. This would result in outline polygons with joins and end caps being formed. These polygons will be input into the trapezoidalization algorithm which will result in a different display list (Display List 2) of primitives (see Figure 7b and Figure 8b). The RIP uses the new display list data to create an image of the page for printing.

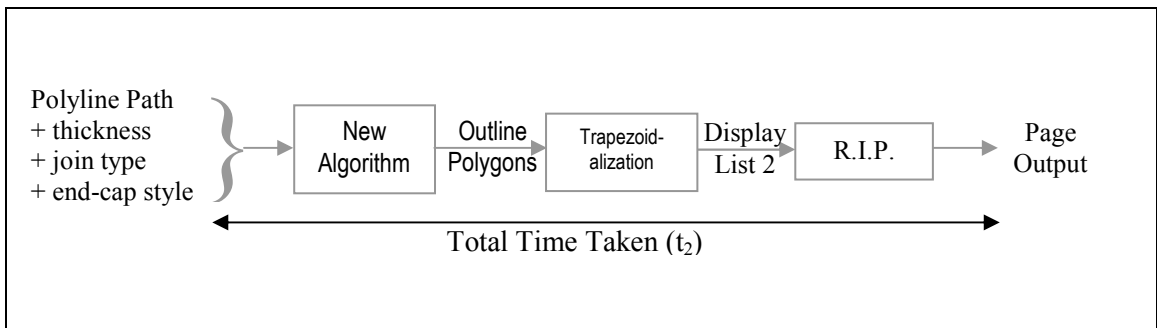


Figure 10 New method for rendering a thick path.

Even though the two display lists resulting from different algorithms are different, the data they represent is the same. The objective of this research is to create an algorithm to process thick polyline data and create outline polygons that will produce a smaller display list when processed by the trapezoidalization algorithm in the GL.

CHAPTER 2

BACKGROUND RESEARCH

2.1 Introduction

Polylines can be treated as objects that form a classification between lines and polygons. In a literature search for thick polylines, most of the references pertained to either lines or polygons. A surprising number of references were oriented towards Geographical Information Systems (GIS) (the primitives in GIS consist predominantly of thick line segments). A polyline that is described as a path will have line segments that might intersect with each other. Intersections will have to be calculated efficiently so that there is as much precision as possible without using up too much of computational resources. With this vision, a search for calculating intersection line segments was performed.

2.1.1 Intersecting Line Segments

Bernard Chazelle [3] and Herbert Edelsbrunner [4] have published papers on the topic of intersecting line segments. These papers deal with generic line segments intersection in two-dimensional and three-dimensional planes. In this research, the techniques developed in this paper to find intersection of multiple line segments

efficiently was considered. However, the complexity of the algorithm was prohibitive, and it was not felt that a substantial advantage could be gained.

2.1.2 Path Tracing

Various techniques useful to the problem of finding outlines of shapes, in general and specifically, of thick polylines were found in the literature. Path tracing [5] was one technique that was considered. Path tracing is the process of tracing the outline of a polygon by running another smaller, simpler polygon (like a square) over it and tracing the path outlined by the simpler polygon. This approach is useful for Robot motion planning but is inefficient to implement here due to the non-simple calculations involved.

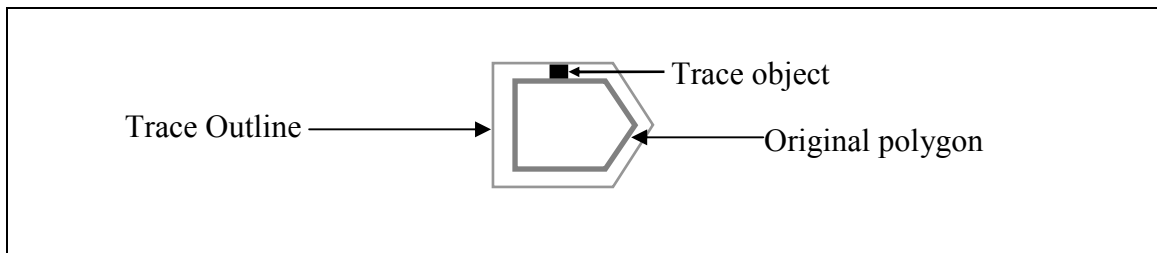


Figure 11 Path tracing.

2.1.3 Polygon Union

Polygon union [7] [15] is another technique to find the outline of overlapping polygons. Here, overlapping polygons are traversed by an algorithm which calculates the intersecting points. These points are used to determine the outer edge of the two polygons to create the union of them. When non-simple polygons are unioned, they

create holes (regions that do not belong to either polygon) —as illustrated in Figure 12—
b and c.

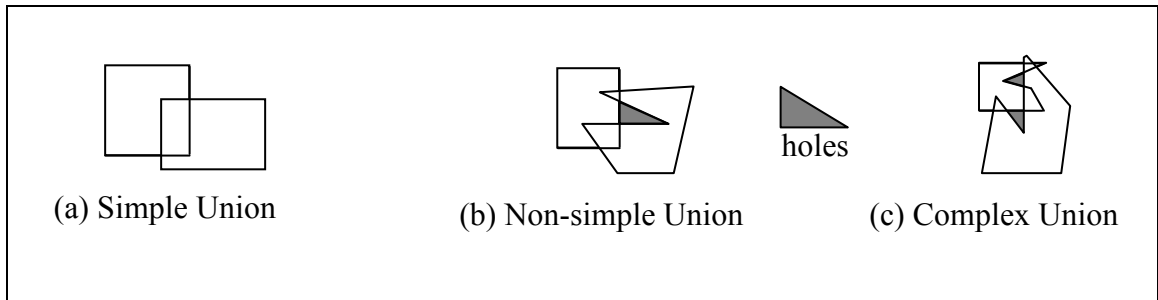


Figure 12 Polygon union types.

2.1.4 Clipping Techniques

Optimizations in checking for line intersections with rectangular clipping areas can be performed by using techniques developed by Cohen-Sutherland [16] [17] and Liang-Barsky [11]. Clipping is used to determine the section of a line that lies inside the area of interest. Clipping is used here to judge whether a line segment falls in the boundary of a polygon (thick line) and if so, the part of the line segment that is clipped (Figure 13). The clipping technique may be modified here to get information on the part of the line segment that falls outside the clipping area.

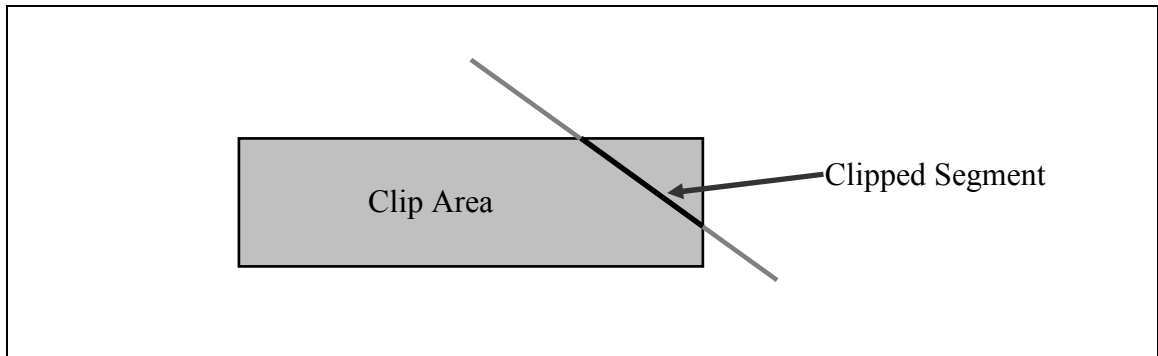


Figure 13 Clipping

2.1.5 Trivial Rejection

Trivial rejection [17] techniques can also be used in this research. Given a set of line segments in two-dimensional space, trivial rejection is a process of quickly eliminating the line segments that do not intersect with the clipping polygon. This is done by determining if the line segment falls inside the area of interest (called the bounding box) by scanning the co-ordinates that determine the ends of the line segments and the co-ordinates of the bounding box.

With reference to Figure 14, if one needs to calculate whether a line, say L_3 , could intersect with a bounding box region, one has to determine if the line falls entirely to one side outside the bounding box.

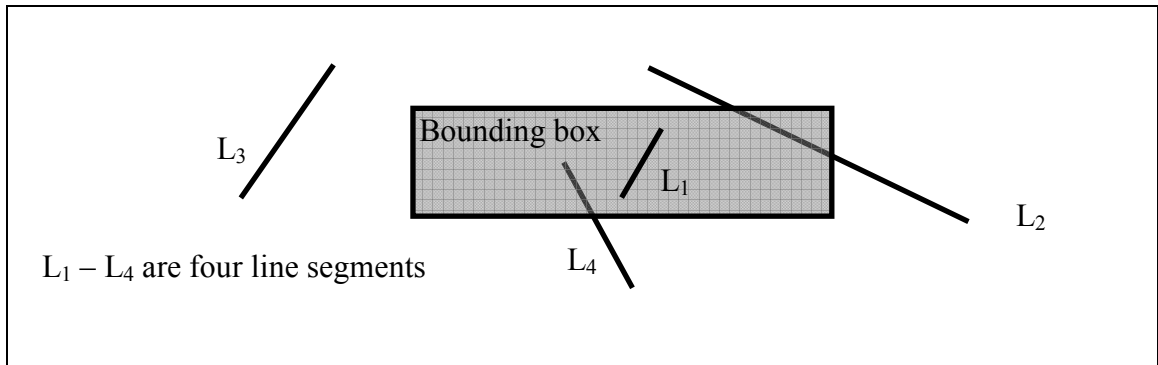


Figure 14 Trivial rejection.

In the case of L_3 , as it falls totally to the left of the bounding box, it can be trivially rejected for intersection calculations. Lines L_2 and L_4 cannot be trivially rejected as they do not fall entirely to one side of the bounding box. Line L_1 can be trivially accepted, as its end points fall within the edges of the bounding box.

The calculations reduced by using a bounding box is an advantage to the research algorithm. Trivial rejection will be employed in this research to eliminate the calculation of intersections between line segments that never intersect.

2.2 Line Segment Attributes

2.2.1 End Caps

Other than the thickness state, thick line segment stroking considers two other states—the end-cap style state, and the join style state. End-caps are ways of terminating thick lines. A thick line terminated at the vertices that form the path line segments, is said to have no caps. Sometimes, it is desirable to extend the end of the line segment to accommodate the path line segment's thickness. The extension is half the thickness of

the path line segment. Such ends are called square caps. In certain cases, preserving smoothness of the line is a priority and so, the ends are rounded off. Such end caps are called rounded caps. An example of each end cap is shown in Figure 15.

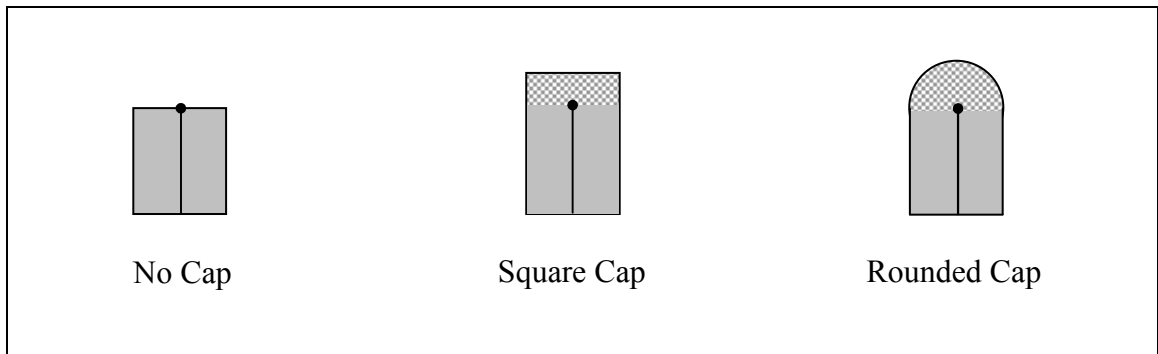


Figure 15 End caps.

2.2.2 Join Considerations

Thick line segments require a join style state that informs the rendering device about the way thick lines are to be joined. When two zero-line thickness (default) line segments are joined, there is no issue of smoothness as each line is only a pixel wide. Figure 16 shows the three join types that are defined in PostScript and a non-standard join, called the triangular join.

The miter join extends the outer edges of the thick line segments to meet at a single point, to preserve continuity. The bevel join is similar to the miter join, but the join is chopped off (bevel cut) for join angles less than an angle related to the miter limit. This is done to prevent the line segments from forming a long extended section, like the elbow shown in Figure 5c. The miter limit is defined as the ratio of the maximum length of the join to the width of the intersecting lines, as shown in Figure 17.

A rounded join is formed by rounding off the join area with a circle whose radius is half the thickness of the polyline. The triangular join is a non-standard implementation that is popular in many fields of interest such as CAD/CAM and GIS. A triangular join is formed by a line from the outer edge of each line to a point half the width of the line from the intersection point, with the maximum extension of the join to be no greater than half the width of thick line segment (measured from base segments' intersection point).

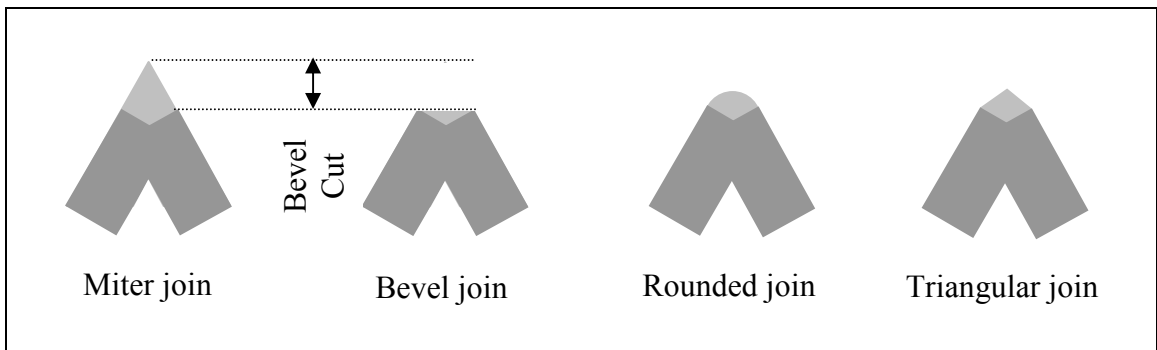


Figure 16 Thick line joins.

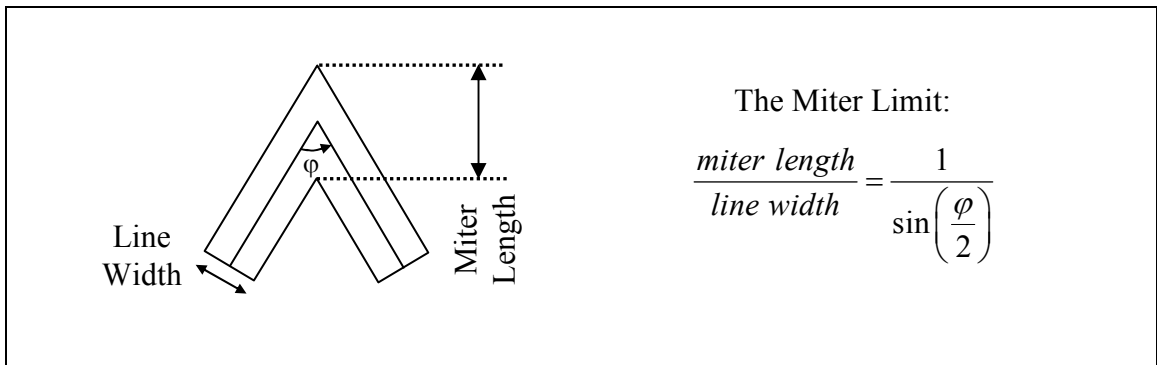


Figure 17 The miter limit ([1] Pg. 509).

CHAPTER 3

METHODOLOGY

3.1 Abstracting the Problem

A thick polyline as described in the PostScript language is composed of a polyline path, a thickness state, a join style state, and an end-cap state. In PostScript, paths are represented as sequences of vertices. The process of mathematically converting a thick polyline into a polygonal outline will be referred to here as “fattening the polyline”.

3.2 Techniques Employed for Creating Polygonal Outline

Determining a polygonal outline of a thick polyline might be done in many ways. Approaches that were investigated in this research were based on linear selection, recursive approaches, and space partitioning approaches [14]. These approaches will be referred to as “Linear Selection”, “Divide-and-Conquer ” and “Page Partition”.

3.2.1 Linear Selection

In this approach, line segments are converted to thick line outlines and are tested for intersection with an existing line segment outline. If a join occurs, then the details of the join are calculated and the multi-polygon is altered to reflect the change. If an intersection occurs, then the parallelism of lines is used to calculate the intersections. If

no overlaps such as joins or intersections occur, the thick line segment is added to the multi-polygon and this process is repeated for other segments in a given polyline. The process is repeated for all polylines in the multi-polyline.

When all of the polylines are converted to outline polygons, the resulting polygons might overlap each other. To remove overlaps, the polygons are unioned to form one non-intersecting multi-polygon, which can be rendered by the rasterizer.

3.2.2 Divide-and-Conquer

The linear selection approach tests every outline segment with a selected polygonal outline for intersection. Many of these line segments might not intersect each other. To prevent unnecessary testing of intersections, a recursive method that first creates polygonal outlines from individual subpaths and then tests one outline against the rest of the outlines for overlap is adopted in this approach.

Checking of outlines for overlap is minimized by providing bounding boxes to each polygonal outline. If the bounding boxes between polygonal outlines do not overlap, then those polygonal outlines do not intersect. If the bounding boxes between polygonal outlines overlap, then the polygons might possibly overlap. In this case, all segments between the two polygonal outlines are tested for intersections. The resulting polygons are unioned to form one non-intersecting multi-polygon. This multi-polygon is sent to the rasterizer for rendering on the target device.

3.2.3 Page Partition

This method is an extension of the Divide-and-Conquer approach. Here, instead of localizing polygons, the page is split into four bounding boxes, one for each corner of a virtual “page”. The bounding boxes can be named as north-west, north-east, south-east and south-west bounding boxes.

All polygons that are present completely inside a bounding box are tested for intersections and are unioned into one multi-polygon. The result is a maximum of four multi-polygons with polygons spanning across bounding boxes. These multi-polygons and polygonal outlines are unioned and result in one non-intersecting multi-polygon that can be rasterized by the rasterizer. This provides for an orderly approach at creating and using bounding boxes.

3.3 Hypothesis

An object-precision algorithm can be devised which will efficiently generate the outline polygons (including holes) of a polyline, using both mitered and bevel joins, and both square and flat end-caps.

We have proposed an alternative way to render thick polyline outlines by mathematically creating a polygonal outline of the thick polyline that has the joins and end caps constructed when the outline is built, instead of creating these attributes as an attachment to the constructed thick polyline as is conventionally done.

3.4 Visual Testing Tool

A visual testing tool has been developed to facilitate visual correctness verification of the developed algorithm. With this tool, interactive creation, altering and thickness control of thick polylines, along with alteration of thick line attributes such as square caps vs. flat caps, bevel joins vs. miter joins, and changing of miter limit settings are made possible.

This tool has been developed to visually compare the results of three different processors of thick polylines—the Microsoft Windows API implementation of thick polylines, the PostScript implementation of thick polylines, and the developed algorithm for thick polyline processing. In the screen output, one can verify correctness of the developed algorithm [7] by running it along with the Windows implementation. In the print output, correctness can be verified by running the developed algorithm along with the PostScript implementation.

The visual polyline editor has a drawing canvas where the base multi-polyline is created. For this multi-polyline, properties such as polyline thickness, end cap type and join type can be specified through a pull-down menu or a shortcut toolbar. The icons associated with multi-polyline actions, such as changing end cap type and changing join type, are red in color. An icon associated with the Windows implementation of polyline outline is also present in this group of icons. When invoked, the Microsoft Windows implementation of thick polyline outline is shown on the drawing canvas as a painted grey overlay on the base multi-polyline.

The research algorithm can be invoked from the visual polyline editor through the shortcut icon on the toolbar or from the pull-down menu. Once invoked, the algorithm

calculates and displays the mathematical outline of the multi-polygon result. This outline might have overlaps and intersections, which are handled by Hain and Subramaniam's complex polygon decomposition [18] algorithm.

On changing of attributes of the polyline—such as miter limit setting, bevel joins against miter joins and square caps at the ends, the developed algorithm should behave the same as the existing Windows and PostScript implementations. This behavior will signify the correctness of the developed algorithm.

CHAPTER 4

THEORETICAL DEVELOPMENT

4.1 Proposed Algorithms

Of the three algorithms proposed in this research, only one algorithm was implemented (see code in Appendix A)—the Linear Selection Algorithm. The Divide-and-Conquer Algorithm and the Page Partition Algorithm were investigated, but not developed due to time constraints. The unimplemented algorithms are outlined in the following subsections.

4.1.1 Outline of the Divide-and-Conquer Algorithm

The Linear Selection Algorithm (discussed in the next section) was developed based on simplicity and calculation reuse which led to many anomalies that cropped up in the resulting outline polygon. To remove the anomalies, various checks were performed to keep track on correctness of the output of the algorithm.

The Divide-and-Conquer Algorithm can be developed based on the principle of locality. Each outline polygon would be created locally and the polygons from different polylines would be merged using bounding boxes for trivial rejection of polylines that do not overlap or intersect. Individual polygons would be created based on thick segment outline that would result in possibly overlapping polygons. These overlaps would be

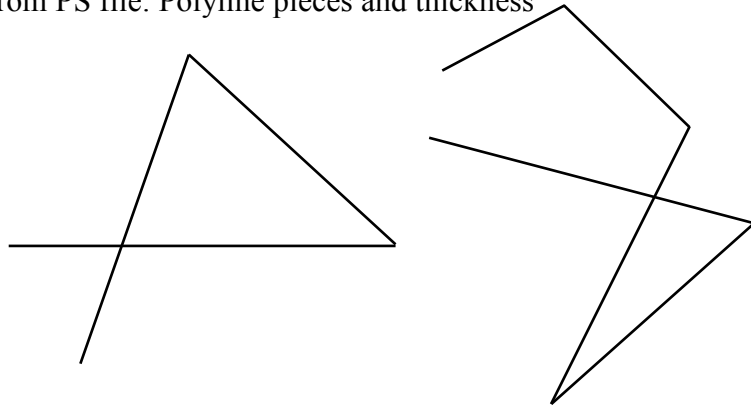
removed when polygons are merged together to calculate a set of non-intersecting and non-overlapping polygons.

A subpath is a set of connected line segments. A path can be made up of subpaths. In the Divide-and-Conquer approach, a single line segment is selected from a subpath and its thick line segment is created. Other line segments in the same subpath are converted to thick line segments, tested for overlap and merged with join details.

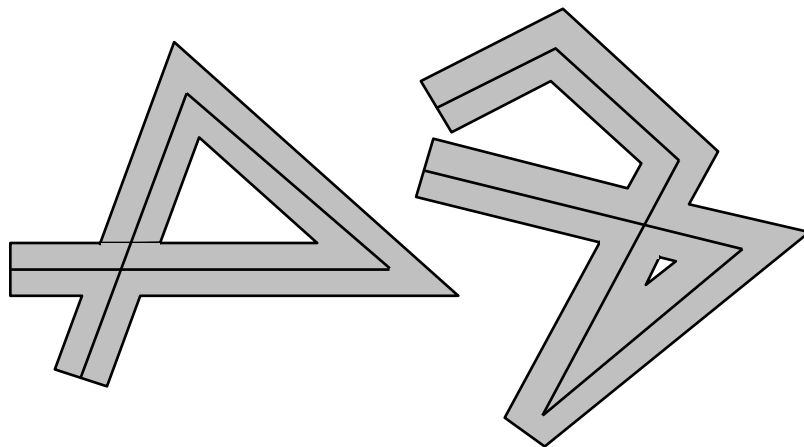
Once all the subpaths are converted to polygonal outlines, a bounding box check will be used to trivially reject polygonal outlines that never overlap. The polygons that do overlap will be merged to form a multi-polygon. This should minimize the time taken to create the final multi-polygon. The algorithm is as follows:

1. Read input
2. Select the first path line segment from the first subpath
3. Create thick line outline for selected line segment
4. Select the next path line segment from same subpath, create its thick outline
5. If more line segments exist in the subpath, perform previous step
6. If more subpaths exist, select next subpath, select a path line segment and perform step 3
7. Select first polygon from list of polygons
8. Select next polygon and check for polygon overlap
9. If polygons overlap, then calculate the new polygon and write it to list of polygons, remove the details of the two individual polygons from list
10. If more polygons exist, go to step 8
11. Decompose the resulting thick line / multi-polygon to display list.

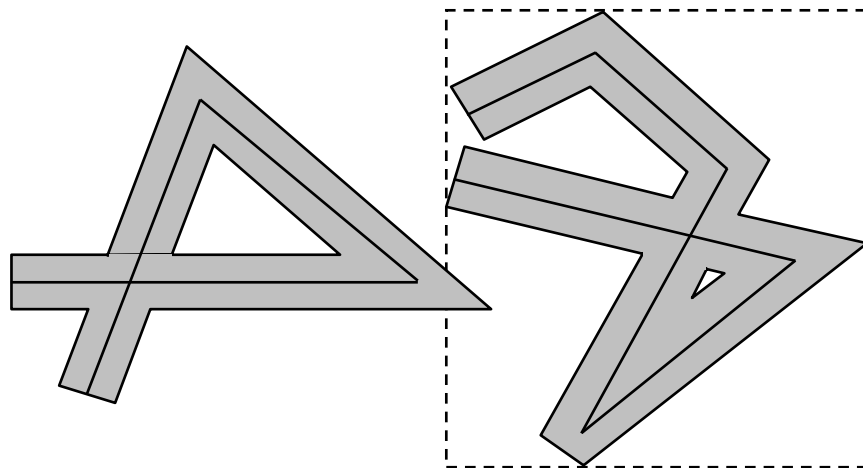
(a) Input from PS file: Polyline pieces and thickness



(b) Create polygonal outline from individual subpaths by linear selection method



(c) Test for intersection between polygonal outlines by using bounding boxes



(d) Merge polygons and decompose to Display List

Figure 18 The Divide-and-Conquer algorithm.

An example based on the Divide-and-Conquer method is shown in Figure 18. In this approach, polyline pieces are built such that each polyline piece is converted to a polygonal outline. Based on the bounding box techniques, polygonal outlines that do not overlap can be trivially rejected.

4.1.2 Outline of the Page Partition Algorithm

The Page Partition Algorithm would be developed as an extension of the Divide-and-Conquer Algorithm. In the Divide-and-Conquer Algorithm, the number of preliminary polygons generated would be equal to the number of disjoint polylines. In the Page Partition Algorithm, there would be only four preliminary polygons—one for each corner of a virtual “page” (North-East, North-West, South-East and South-West).

The reason for this proposal is that edge intersection calculations can be trivially rejected based on the regions in which the edges exist. We believe that trivial rejection of intersection between edges would lead to minimal intersection calculations which would speed up the execution time of the Divide-and-Conquer Algorithm.

The Page Partition Algorithm:

1. Designate four partitions of the page as NW, NE, SE and SW
2. Read the list of path line segments from input file
3. Select the first path line segment
4. Determine partition the selected line segment’s first end co-ordinate falls in
5. Save the line segment info in the list corresponding to that partition
6. If more line segments exist, select the next path line segment, go to step 4

7. Select one partition of the page, select the first path line segment in the partition, create thick line outline
8. Select next path line segment in the same partition, create thick line outline and check for intersection with existing thick line outline / polygonal outline
9. If an overlap exists, then merge the thick line outline and polygonal outline and calculate the join details
10. If an overlap does not exist, write the thick line outline as a new polygonal outline
11. If more path line segments exist, go to step 9
12. If all path line segments in partition have been exhausted, then go to step 16
13. Calculate thick line segment of previously marked line and its intersection with the current thick line, write polygon to the list of polygons
14. Select the next non-intersecting path line segment, go to step 9
15. If more partitions exist, select next partition, go to step 8
16. Select first polygon from the list of polygons
17. Select next polygon and if the polygons are not in the same quadrant, trivially reject the polygon for intersection calculations.
18. If polygons overlap, then calculate the new polygon and write it to the list of polygons, remove the details of the two individual polygons from the list
19. If more polygons exist, go to step 17
20. Write the resulting thick line / polygon data to the display list.

An example based on the Page Partition approach is presented in Figure 19. The advantage of this approach is that polygons confined to a single quadrant can be trivially rejected with polygons confined to another quadrant. Polygons that span quadrants cannot be trivially rejected with other polygons in either quadrants and have to be tested for intersections.

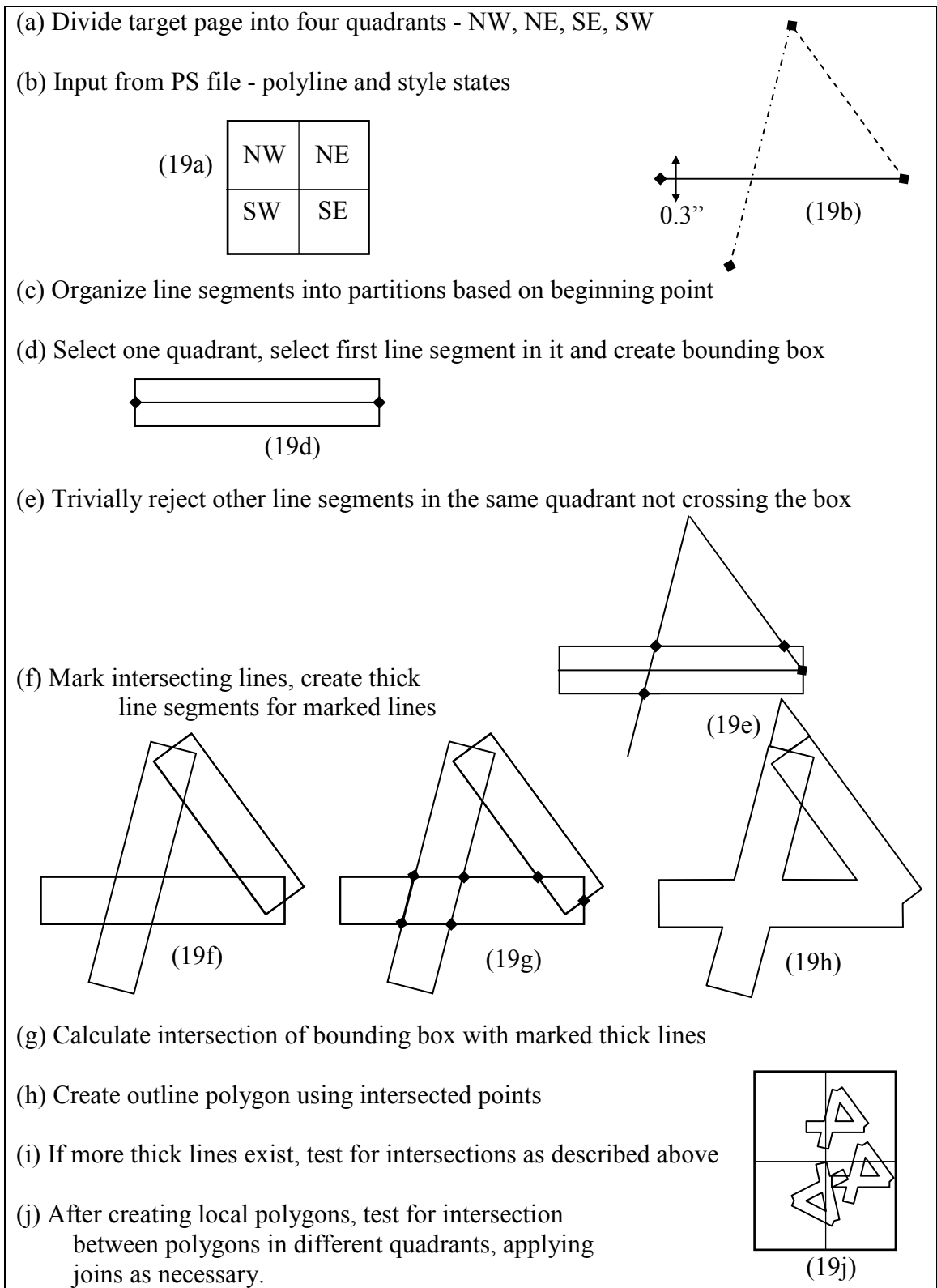


Figure 19 The Page Partition algorithm.

4.2 The Linear Selection Algorithm

The Linear Selection algorithm was developed using the properties of line intersections. A thick polyline is made up of a set of connected line segments enclosing the base polyline at a specified distance.

The Linear Selection Algorithm is outlined below and described in detail in the following sections:

1. Read input
2. Select first path line segment
3. Create a thick line outline from the selected path line segment attributes and end cap details
4. Select next path line segment, create thick line outline and test for intersection with current thick line outline
5. Create polygonal outline from merged thick line outlines and join states
6. If the thick line outline does not intersect with the current polyline outline, create a multi-polygon from the current polyline outline and existing thick line outline
7. If a join or an intersection has occurred due to a merge of thick line outline with polygonal outline and calculate join details, write to multi-polygon.
8. If more thick lines exist, repeat steps 5, 6 and 7
9. If all thick lines and/or polygonal outlines have been exhausted, then decompose the resulting multi-polygon data to display list.

An example based on the linear selection approach is demonstrated in Figure 20.

This approach is linear and certain optimizations can increase the performance of the algorithm. The existing path line segments can be tested for intersections by using the clipping algorithm of Liang-Barsky [11] and Cohen-Sutherland's [16] [17] trivial rejection algorithm.

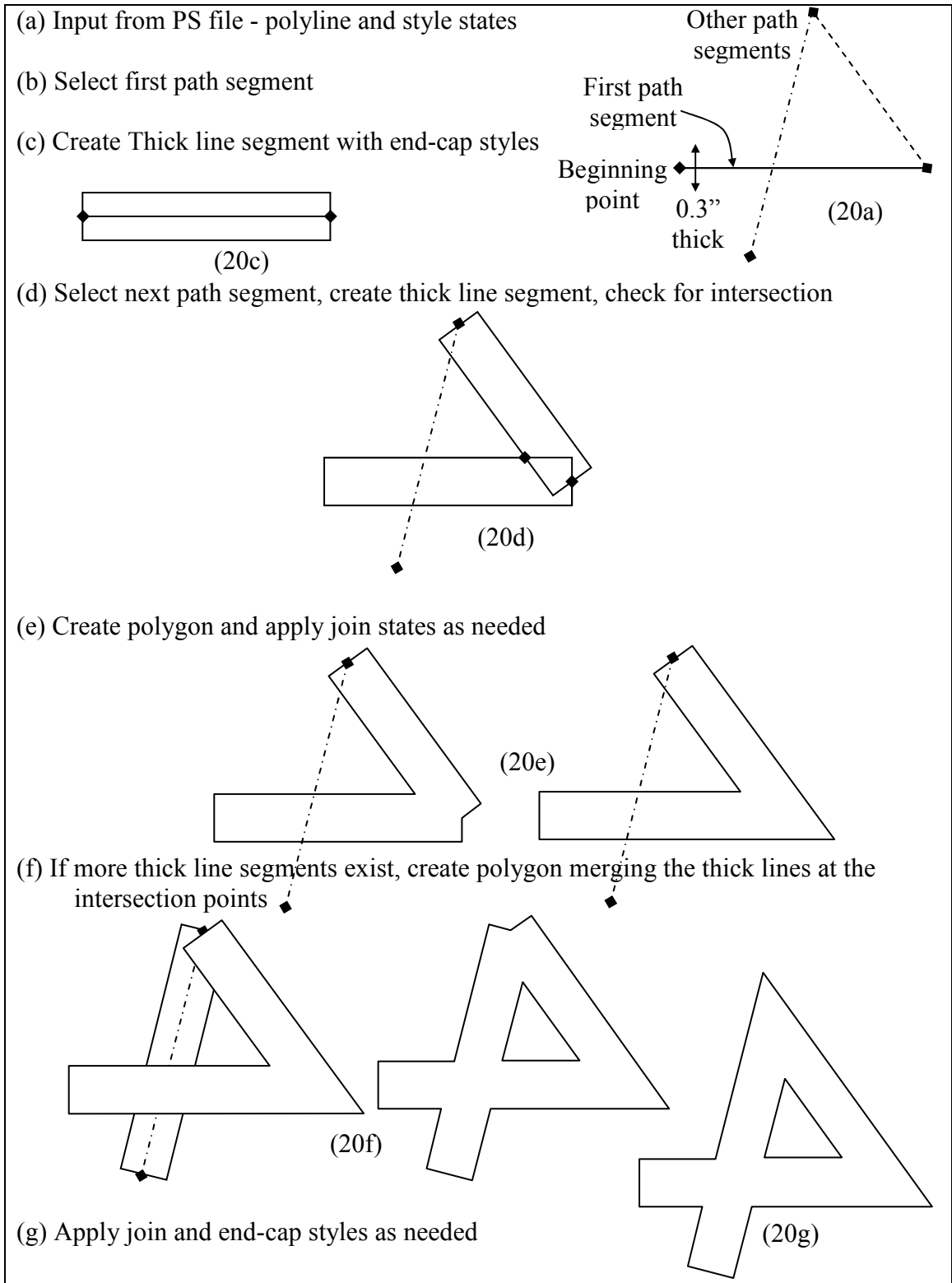


Figure 20 The Linear Selection algorithm.

When the algorithm is invoked to create an outline for a polyline, the end enclosure is created perpendicular to the base line segment. If square end caps are needed, the perpendicular line segment is offset to half the thickness of the polyline.

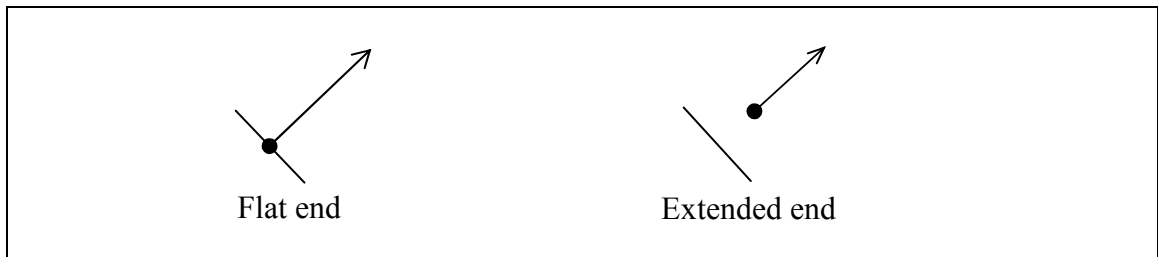


Figure 21 Types of end cap.

Once the end segment is created, the line parallel to the left side of the base line segment is created. This line is checked for intersection with the left parallel line to the next segment of the polyline. Depending on the turn of the polyline, the intersection point would lie inside (direction of turn is left) or outside (direction of turn is right) the perpendicular line segments. Based on the direction of turn, inner and outer intersection points are determined. Their corresponding line segments are labeled inner and outer line segments.

The point where the first line segment extends to the second is termed the mid point. Once the first intersection point (left intersection) is determined, it is projected on the other side of the mid point to get the right intersection point.

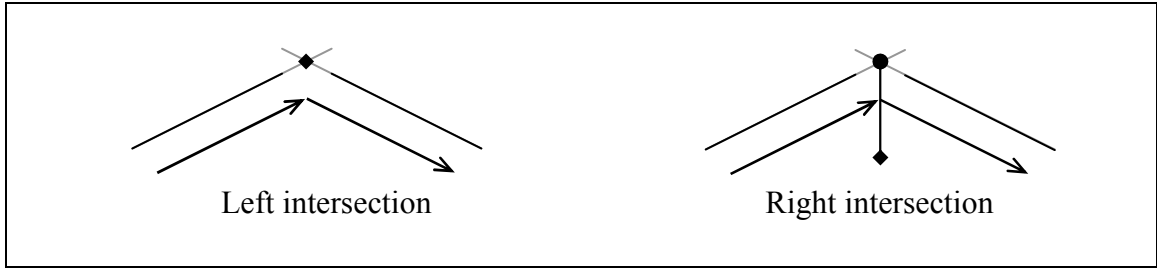


Figure 22 Left and right intersection points

4.2.1 Near Line Segments Check

Once the two intersection points are determined, a check has to be made to determine if the inner intersection point falls outside either of the inner line segments. This condition occurs when the base line segments are quite close to each other.

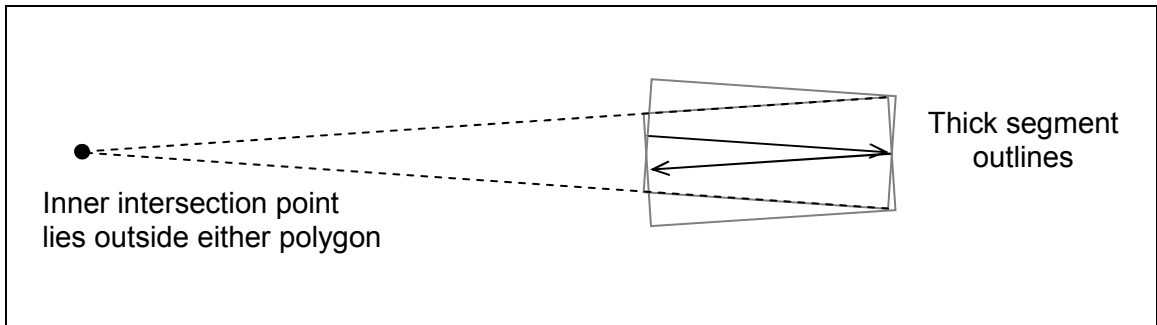


Figure 23 Near line segments check

4.2.2 Preliminary Overlap Polygon

To maintain consistency throughout the algorithm, lines and line segments are only checked for intersection if they are consecutive. When an inner intersection point lies outside a corresponding parallel line segment, the original polygonal outline is closed and a new polygonal outline starting from the end of the previous polygonal outline is

created. This is termed a preliminary overlap polygon. Overlaps are allowed here as the polygonal outline created by this algorithm will be run through a polygon union algorithm, devised by Hain [7] (see also [15]) before being sent to the trapezoidalization processor.

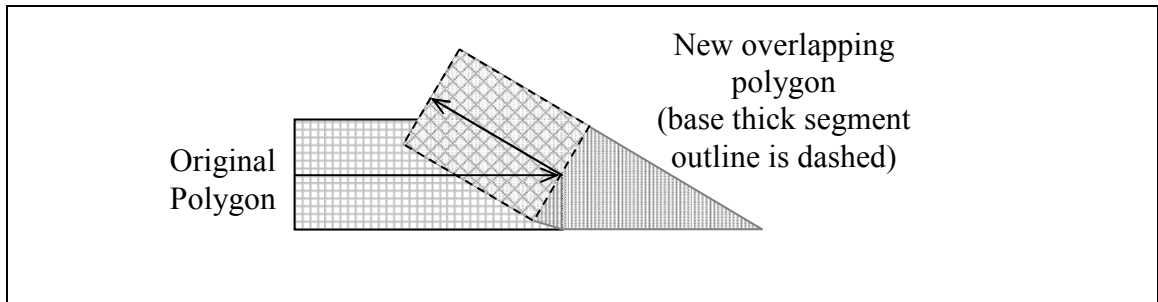


Figure 24 Preliminary overlap polygon with construction.

4.2.3 End Extension Check

The process of building the outline polygon from the base polyline with specified thickness is continued until there are no more line segments to process in the current polyline. Once the last segment of the polyline has been traversed and intersections calculated, then the closing segment is built. Here again, the overshoot caused by intersection point lying outside either of the two consecutive inner segments is checked for. If this condition occurs, then the erroneous point is removed, the polygon closed, a new polygon created from the point of closure of the old polygon, and the outline is completed. This prevents overshoot of segments at the ends.

4.2.4 Bevel Joins

The visual test tool creates miter joins by default, as described in PostScript. If two consecutive segments in a polyline create an outline extension greater than the miter length, or if the Bevel Join option is selected in the visual test tool, then bevel joins are created. Beveling causes the polygon outline to be joined at the ends of the original thick line segments. In this case, as before, it is necessary to check for errors caused by overshoot of inner intersection point, which will need closure of the existing polygon, and creation of a new overlap polygon to continue processing from that segment onward.

4.2.5 Short Line Segments

The developed algorithm works well for line segments that are longer than the thickness of the resulting thick polyline. When the line segment's length is shorter than the thickness, then the algorithm does not behave well and causes offshoots in both directions of the line segment. To prevent this behavior, short line segments are closed and the required extensions caused by a miter or bevel are recreated in the new outline polygon. These form overlapping polygons to create a smoother output without any glitches and overshoots, and conform to behavior of other implementations [10], [12].

4.2.6 Inline Optimizations

Special considerations are provided for optimization if the polyline segments are inline. A check is made if the line segments are in the same direction, and if true, the segments are merged. If the line segments are inline and in opposite direction, then the original segment is closed and a new polygon is created in continuation with the polyline.

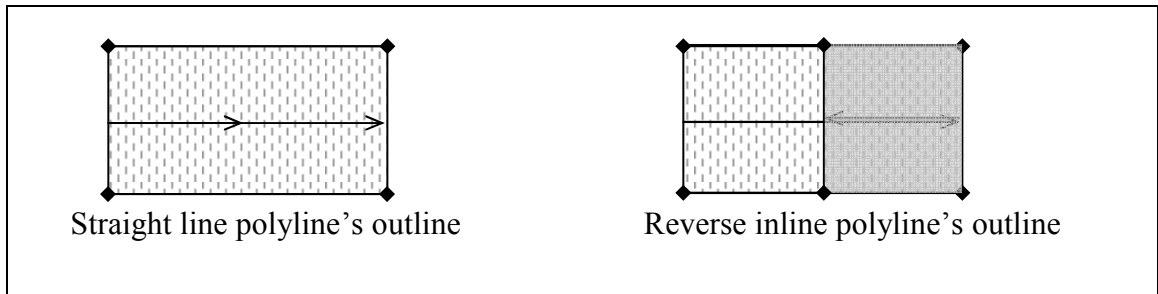


Figure 25 Inline optimizations.

4.3 Hain's Complex Polygon Decomposition Algorithm

The output of the developed algorithm is a set of convex polygons with possibly overlapping segments. When these polygons are input to Hain's polygon union algorithm [7], the result is a set of non-intersecting and non-overlapping (simple) polygons. This output can be used as input for the trapezoidalization algorithm in the Graphics Library. Since any given set of polygons can be merged by Hain's algorithm, the preliminary processing of decomposing complex polygons is not performed by the developed algorithm. This helps to avoid duplication of efforts in merging the polygonal outline.

CHAPTER 5

CONCLUSION

5.1 Summary of Current Work

This research created a new object-precision algorithm—the Linear Selection Algorithm—to generate a thick polyline’s outline. The algorithm was designed to optimize performance by factoring computations in such a way as to avoid duplication of intermediate results, and to produce a smaller display list of disjoint primitives (e.g., trapezoids) than current methods. The correctness of the implementation (details on invoking the algorithm are given in Appendix B) was exhaustively tested by visual comparison with two existing implementations—the Win32 graphics library, and the GhostScript implementation of PostScript—and was found to be correct to a high tolerance under all tested conditions. Thus the hypothesis stated in Section 3.3 was satisfied.

The visual workbench (see code in Appendix A) discussed in Section 3.4, was constructed. It was used extensively for testing of algorithmic behavior, and for output comparison with other implementations (see above). It was found to be invaluable in both the development, and the testing of the algorithm.

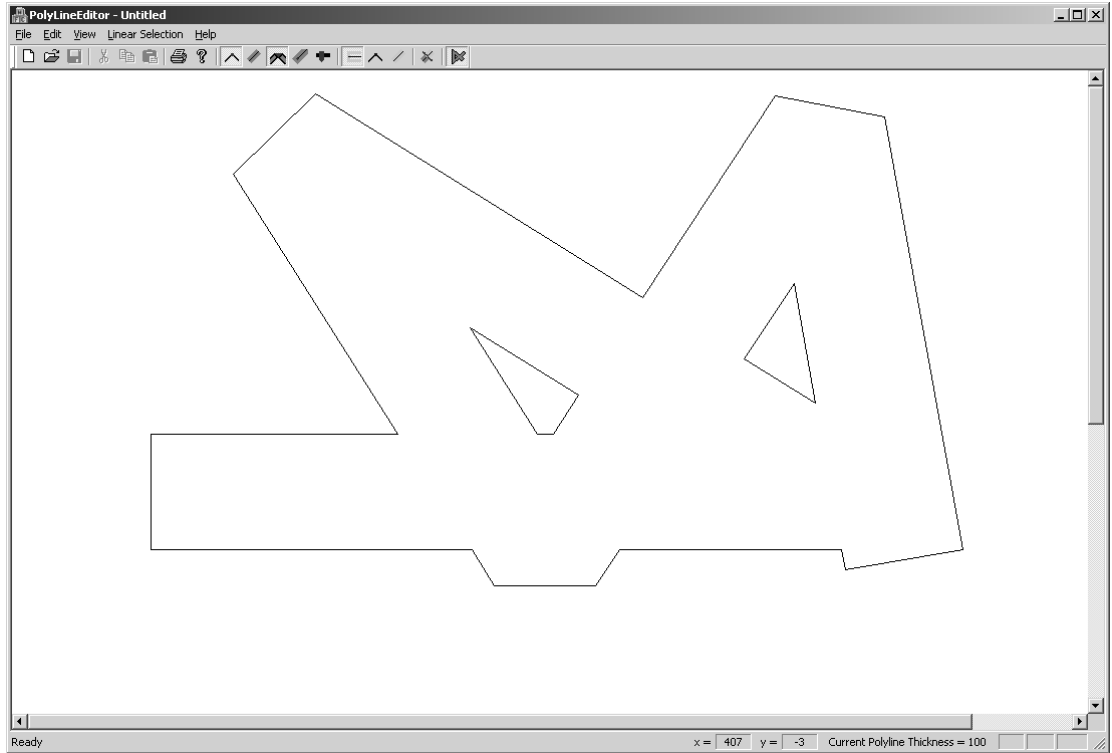


Figure 26 Visual polyline editor tool.

It is believed that the Linear Selection Algorithm will work more efficiently than current algorithms implemented for thick-line rendering. Current algorithms used for thick-line rendering are either proprietary (Minolta-QMS firmware [13]) or too complex to extract from public domain software (GhostScript). Due to non-availability of a comparison algorithm, timing tests were not made on the developed algorithm.

5.2 Future Research

As part of future research in this topic, the Divide-and-Conquer Algorithm, outlined in Section 4.1.1, and the Page Partition Algorithm, outlined in Section 4.1.2, could be developed. With discovery of better techniques, the above-mentioned

algorithms could be optimized based on better understanding of properties of line segments, thick line segments and their internal intersections. Implementation and timing tests between the various algorithms could lead to selecting the right algorithm for faster processing of thick line segments.

In this research, two very popular join types—the miter join and the bevel join have been investigated. The rounded join could be implemented as a series of line segments flattening the curve where as the triangular join could be implemented as an extension to the bevel join.

With regards to end caps, the flat end caps and square end caps have been implemented in this thesis. The rounded end cap that is not implemented in the current research could be implemented as a series of line segments flattening the closing semi-circle, similar to the rounded join implementation. The triangular end cap is more straight forward to implement, but does not have any standard implementation, at this time, to be compared with for correctness.

REFERENCES

REFERENCES

- [1] Adobe Systems Inc., “PostScript[®] Language Reference Manual”, Addison-Wesley Publishing Co., Reading, Mass., 1999.
- [2] Ahmad, Athar L., “Approximation of a Bezier Curve with Minimum Number of Line Segments”, *Master’s Thesis*, School of CIS, University Of South Alabama, May, 2001.
- [3] Chazelle, Bernard and Herbert Edelsbrunner “An Optimum Algorithm for Intersecting Line Segments in the Plane”, *Journal of the ACM*, Vol. 39, No. 1, Jan 1992, Pages 1-54 [<http://doi.acm.org/10.1145/147508.147511>]
- [4] Edelsbrunner, Herbert “Lines in Space—A Collection of Results”, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 6, 1991, Pages 77–93, Editors - Goodman, Pollack and Steiger.
- [5] Flato, Eyal and Dan Halperin “Robust and Efficient Construction of Planar Minkowski Sums”, Abstracts 16th European Workshop on Computational Geometry, Eilat, 2000, Pages 85-88 [<http://citeseer.nj.nec.com/flato00robust.html>]
- [6] Hain, T.F. “A Fast, Practical Algorithm for the Trapezoidalization of Simple Polygons”, *Internal Report USA-CIS 971001*, University of South Alabama, 1997.
- [7] Hain, T.F. Private communication. New code developed for polygon union, intersection, difference, and symmetric difference. Based on [18], Sept 2003.
- [8] Hain, T.F. Private Communication.
- [9] Hain, T.F. and Gulve, S. “Interactive, Visual Testing Strategy for Computational Geometry Problems”, *1998 ACM Southeast Conference*, Atlanta, GA, April 1998.
- [10] Lang, Russel “GhostScript—An Interpreter for the PostScript Language and for PDF” homepage [<http://www.cs.wisc.edu/~ghost/>]
- [11] Liang, Y.D. and Brian Barsky “A New Concept and Method for Line Clipping”, *ACM Transactions on Graphics*, Vol. 3, No. 1, Jan 1984, Pages 1-22 [<http://doi.acm.org/10.1145/357332.357333>]
- [12] Microsoft Corp. “Visual Studio .NET” homepage [<http://msdn.microsoft.com/vstudio/>]

- [13] Minolta-QMS “The Graphics Library”, *Internal Report*, 1999.
- [14] Mulmuley, Kenan “A Fast Planar Partition Algorithm, II,” *Journal of the ACM*, Vol. 38, Issue 1, Jan 1991, Pages 74-103.
- [15] Murta, Alan “General Polygon Clipping (GPC) Algorithm” homepage [<http://www.cs.man.ac.uk/aig/staff/alan/software/>]
- [16] Newman, William M. and R.F. Sproull “Principles of Interactive Computer Graphics”, 2nd Ed., McGraw Hill, New York, 1979.
- [17] Rogers, David F. “Procedural Elements for Computer Graphics”, 2nd Ed., McGraw Hill, Boston, Mass., 1998, Pages 175-286.
- [18] Subramaniam, Lavanya “Partitioning a Complex Polygon Into a Set of Simple Polygons”, *Master’s Thesis*, School of CIS, University of South Alabama, June 2003.

APPENDICES

Appendix A

Code

```
//////////////////////////////////// POINT.H //////////////////////////////////////
// Vertex: vertex Written by T. Hain, Jan 1998,
//          extended by Swaminadhan, 2003
//  Attributes: Coord (float) x, y.
//
// Constructors
//   Vertex()
//   Vertex(const int x, const int y)
//   Vertex(const CPoint &point)
//
// Operators
//   void operator + (const Vertex &v)
//   void operator - (const Vertex &v)
//   void operator += (const Vertex &v)
//   void operator -= (const Vertex &v)
//   void operator == (const Vertex &v)
//   void operator != (const Vertex &v)
//   void operator < (const Vertex &v)
//
// Cast to CPoint
//   operator CPoint()
//
// Tests if vertex is within distance 'resolution' of line (p1,p2)
//   bool isNearLine(const Vertex &p1, const Vertex &p2, int
//       resolution)
//
// Calculate normal point to line (_p1, _p2)
//   Vertex calcNormalPoint(Vertex _p1, Vertex _p2, float _dist, enum
//       SIDE _side)
//
// Cross product between three successive vertices (this, q, r)
//   Coord xProd(const Vertex &q, Vertex &r) const
//
// Project a point wrt. another point with projection ratio
//   Vertex project(Vertex vertex1, float pRatio)
//
// Calculate distance between current vertex and Vertex vtxRight
//   double distance(Vertex vtxRight)
```

```

#define ROUNDOFF 0.1
enum SIDE {LEFT, RIGHT};
typedef float Coord;

class Vertex
#ifdef MFC
    : public CObject
#endif

{
public:
    Coord x, y;

    Vertex(const Coord _x = 0, const Coord _y = 0): x(_x), y(_y)
    {
    }
    Vertex(const Vertex &v): x(v.x), y(v.y) {}
    void operator= (const Vertex &v)
    {
        x = v.x; y = v.y;
    }
    Vertex operator+ (const Vertex &v) const
    {
        return Vertex(x + v.x, y + v.y);
    }
    Vertex operator- (const Vertex &v) const
    {
        return Vertex(x - v.x, y - v.y);
    }
    Vertex& operator+= (const Vertex &v)
    {
        x += v.x;
        y += v.y;
        return *this;
    }
    Vertex& operator-= (const Vertex &v)
    {
        x -= v.x;
        y -= v.y;
        return *this;
    }
    bool operator== (const Vertex &v) const
    {
        return x == v.x && y == v.y;
    }
    bool operator!= (const Vertex &v) const
    {
        return x != v.x || y != v.y;
    }
    bool operator< (const Vertex &v) const
    {
        return x < v.x || x == v.x && y < v.y;
    }
}

```

```

#ifdef __MFC
Vertex(const CPoint &p)
{
    x = (Coord)p.x;
    y = (Coord)p.y;
}

operator CPoint()
{
    return CPoint((int)x, (int)y);
}
operator CPoint() const
{
    return CPoint((int)x, (int)y);
}
#endif

Vertex calcNormalPoint(Vertex _p1, Vertex _p2, float _distance, enum
    SIDE _side)
{
    float dx = -3.0f * _p1.x + 3.0f * _p2.x;
    float dy = -3.0f * _p1.y + 3.0f * _p2.y;
    float h = (float)sqrt(dx * dx + dy * dy);

    float factor = _distance / h;
    if(_side == RIGHT)
        factor = -factor;

    x = _p2.x - dy * factor;
    y = _p2.y + dx * factor;
    return *this;
}

Vertex& snap(Coord resolution) // Snap coords to given resolution
{
    if (resolution)
    {
        x = resolution * (int)(x/resolution);
        y = resolution * (int)(y/resolution);
    }
    return *this;
}

Coord xProd(const Vertex &q, Vertex &r) const
{
    return q.y * (r.x - this->x) + this->y * (q.x - r.x) + r.y *
        (this->x - q.x);
}

void dump(ofstream &out)
{
    out << "(" << x << "," << y << ")";
}

```

```

    BOOL isNearLine(const Vertex &p1, const Vertex &p2, int
        resolution) const
    {
        CRect activeRect((int)p1.x, (int)p1.y, (int)p2.x, (int)p2.y);
        activeRect.NormalizeRect();

        // for vertical or horizontal edges
        activeRect.InflateRect(resolution, resolution);
        if (!activeRect.PtInRect(*this))
            return false; // trivial reject (i.e., *this is not
        float x1 = p1.x, // within line seg's bounding box)
            y1 = p1.y;
        float x2 = p2.x,
            y2 = p2.y;
        float a = (y1 - y2),
            b = - (x1 - x2),
            c = ( x1 * (y2 - y1) - y1 * (x2 - x1) );
        float xba = x2 - x1,
            yba = y2 - y1;
        double lsqr = xba * xba + yba * yba,
            l = sqrt(lsqr);
        float yac = y1 - y,
            xac = x1 - x;
        double r = ( -yac * yba - xac * xba);
        double s = ( yac * xba - xac * yba) / l;
        return (fabs(s) < resolution && r > 0 && r < lsqr);
    }

    Vertex project(Vertex vertex1, float pRatio)
    {
        return Vertex(pRatio*vertex1.x + (1.0f-pRatio)*x,
            pRatio*vertex1.y + (1.0f-pRatio)*y);
    }

    double distance(Vertex vtxRight)
    {
        return sqrt((vtxRight.x - this->x) * (vtxRight.x - this->x) +
            (vtxRight.y - this->y) * (vtxRight.y - this->y));
    }
};

```

```
////////////////////////////////////  
// Baseline: baseline           Written by T. Hain, Jan 1998.  
//  
// Attributes: Vertex p1, Vertex p2  
//  
// Constructors  
//   Baseline(Vertex _p1, Vertex _p2)  
//  
// Transform  
//   Vertex Transform (Vertex vtx)  
//  
// InvTransform  
//   Vertex InvTransform (Vertex vtx)  
//  
// Tests if vertex is within distance 'resolution' of line (p1,p2)  
//   bool isNearLine(const Vertex &p1, const Vertex &p2, int  
//     resolution)
```

```

struct Baseline
{
    Vertex p1, p2;
    Coord x21, y21, length, x21l, y21l;
    Baseline(Vertex p1, Vertex p2)

    {
        p1 = _p1; p2 = _p2;
        x21 = p2.x - p1.x; y21 = p2.y - p1.y;
        length = (Coord)sqrt(x21*x21 + y21*y21);
        x21l = x21/length; y21l = y21/length;
    }
    Vertex Transform(Vertex vtx)
    {
        return Vertex(vtx.x*x21l + vtx.y*y21l + -p1.x*x21l - p1.y*y21l,
            -vtx.x*y21l + vtx.y*x21l + p1.x*y21l - p1.y*x21l);
    }
    Vertex InvTransform(Vertex vtx)
    {
        return Vertex(vtx.x*x21l + vtx.y*(-y21l) + p1.x,
            vtx.x*y21l + vtx.y*x21l + p1.y);
    }
    double Length()
    {
        return length;
    }
};

////////////////////////////////////
// Line: line Written by T. Hain, extended by Swaminadhan, 2003.
//
// Attribute: float r, s, t
//
// Constructors
//   Line ()
//   Line (Vertex p1, Vertex p2)
//   Line (const Lin& oldLine)
//
// Operators
//   void operator = (const Line &line)
//
// Normalize Line
//   void normalize()
//
class Line
{
public:
    float r, s, t;
    void normalize()
    {
        float n = (float)sqrt(r*r+s*s); r = r/n; s = s/n; t = t/n;
    }
};

```

```

Line(Vertex p1, Vertex p2)
{
    r = p2.y - p1.y; s = p1.x - p2.x; t = p2.x * p1.y - p1.x * p2.y;
    this->normalize();
}
Line(const Line& oldLine)
{
    r=oldLine.r; s=oldLine.s; t=oldLine.t;
}
Line()
{
    r = -1; s = 1; t = 0;
}
void operator=(const Line &line)
{
    r=line.r; s=line.s; t=line.t;
}
Vertex intersect(Line l2) const
{
    double dem = l2.s * r - s * l2.r;
    return Vertex((s * l2.t - l2.s * t)/(float)dem, (t * l2.r - l2.t
        * r)/(float)dem);
}
bool isInline(Vertex vtx)
{
    float test;
    test = vtx.x*r + vtx.y*s + t;
    if(test <= ROUNDOFF && test >= ROUNDOFF*-1)
        return true;
    return false;
}
Line Line::parallel(float distance, SIDE side);
Line Line::perpendicularTo(const Vertex &vtx);
};

typedef list<Vertex>::iterator VtxIt;
typedef list<Vertex>::const_iterator VtxConstIt;

```

```

//////////////////////////////////// POINT.CPP //////////////////////////////////////

#define OFFSET 100

// Find a line parallel to current line on SIDE
Line Line::parallel(float distance, SIDE side)
{
    Line newLine(*this);

    if(r > 0)    //determine orientation of parallel line
    {
        if (side == LEFT)
            newLine.t += distance;
        else
            newLine.t -= distance;
    }

    else
    {
        if(side == RIGHT)
            newLine.t -= distance;
        else
            newLine.t += distance;
    }

    newLine.normalize();
    return newLine;
}

Line Line::perpendicularTo(const Vertex &vtx)
{
    return Line(Vertex(vtx.x - r * OFFSET, vtx.y - s * OFFSET),
                Vertex(vtx.x + r * OFFSET, vtx.y + s * OFFSET));
}

```

```

//////////////////////////////////// MISC.H //////////////////////////////////////
// LineSegment: A single segment (point1, point2)
//
// Attributes: Vertex vertex0, Vertex vertex1, Rect bbox.
//
// Constructors
//   LineSegment ()
//   LineSegment (const Vertex &vtx0, const Vertex &vtx1)
//   LineSegment (const LineSegment &l)
//
// enum IntersectionType
//   PARALLEL, NO_INTERSECT, INTERSECT
//
// Calculate Intersection of two line segments
//   IntersectionType calcIntersection(const LineSegment &l,
//   Vertex &intersection, float &alpha, float &alpha l) const
//
// Check if vertex is near line segment within particular resolution
//   bool isVertexNear(const Vertex &p, const Coord &resolution)
//   Written by T. Hain
//
// Check orientation of vertex with respect to line segment
//   bool isLeftOf (Vertex &v) const
//   bool isRightOf (Vertex &v) const
//   bool isPtInLine(Vertex vtx)
//
// Operators
//   bool operator< (const LineSegment &ls) const
//
// Create LineSegment on required side at required distance
//   LineSegment Left (float halfThickness)
//   LineSegment Right (float halfThickness)
//
// Get Length of line segment
//   float Length ()

class LineSegment
{
public:
    Vertex vertex0, vertex1;
    Rect bbox;
    enum IntersectionType
    {
        PARALLEL, // lines are parallel within tolerance level
        NO_INTERSECT, // lines segments don't intersect
        INTERSECT // line segments intersect
    };

    LineSegment()
    {}

    LineSegment(const Vertex &vtx0, const Vertex &vtx1): vertex0(vtx0),
        vertex1(vtx1), bbox(vtx0,vtx1)
    { bbox.normalize(); }

```

```

LineSegment(const LineSegment &l): vertex0(l.vertex0),
    vertex1(l.vertex1), bbox(l.bbox)
{}

IntersectionType calcIntersection(const LineSegment &l, Vertex
    &intersection, float &alpha, float &alpha l) const;
    bool isVertexNear(const Vertex &p, const Coord &resolution);

    bool isLeftOf(Vertex &v) const // vertex v is left of (oriented)
segment
    {
        return (vertex1.y * (vertex0.x - v.x) + vertex0.y * (v.x -
vertex1.x) + v.y * (vertex1.x - vertex0.x)) > 0;
    }

    bool isRightOf (Vertex &vtx )const {return(!isLeftOf(vtx));}

    bool isPtinLine(Vertex vtx);

// order by left (bottom) edge of bounding box
bool operator< (const LineSegment &ls) const
{
    return bbox.x0 < ls.bbox.x0 ? true : bbox.y0 < ls.bbox.y0;
}

LineSegment Left(float halfThickness);

LineSegment Right(float halfThickness);

float Length(void);
};

```

```

//////////////////////////////////// MISC.CPP //////////////////////////////////////
//calcIntersection
//   Calculates the point of intersection of two line segments.
//
// Precondition:  both line segments have non-zero lengths.
// Postcondition: The enumerated type function return value tells
//                whether the line segments were parallel, non-
//                intersecting, or intersecting. "intersectionPoint"
//                is valid iff the returned type is not PARALLEL.
//
//

static const float NOISE = 1e-5f;

LineSegment::IntersectionType LineSegment::calcIntersection(const
LineSegment &l, Vertex &intersection, float &alpha, float &alpha_l)
const
{
    Coord dx21 = vertex1.x - vertex0.x, dy21 = vertex1.y - vertex0.y,
          dx43 = l.vertex1.x - l.vertex0.x, dy43 = l.vertex1.y -
          l.vertex0.y, dem = dx21 * dy43 - dy21 * dx43;

    if (fabs(dem) < NOISE)
        return PARALLEL;

    Coord dx12    = vertex0.x - l.vertex0.x,
          dy12    = vertex0.y - l.vertex0.y;

    alpha = static_cast<float>(dx43*dy12 - dx12*dy43)/dem,
    //parametric value of intersection along *this

    alpha_l = static_cast<float>(dx21*dy12 - dx12*dy21)/dem;
    //parametric value of intersection along l

    if (alpha < 0.0 || 1.0 < alpha || alpha_l < 0.0 || 1.0 < alpha_l)
        return NO_INTERSECT;

    intersection.x = vertex0.x + dx21 * static_cast<Coord>(alpha);
    intersection.y = vertex0.y + dy21 * static_cast<Coord>(alpha);
    return INTERSECT;
}

// Determine whether point vtx is within distance 'resolution' from
line segment. Written by T. Hain, Jan 1998.
bool LineSegment::isVertexNear(const Vertex &vtx, const Coord
&resolution)
{
    Coord x1 = vertex0.x,
          y1 = vertex0.y;
    Coord x2 = vertex1.x,
          y2 = vertex1.y;
    Coord a  = (y1 - y2),
          b  = - (x1 - x2),
          c  = ( x1 * (y2 - y1) - y1 * (x2 - x1) );

    Coord xba = x2 - x1,
          yba = y2 - y1;

```

```

    double lsqr = xba * xba + yba * yba, l = sqrt(lsqr);
    Coord yac = y1 - vtx.y, xac = x1 - vtx.x;
    double r = ( -yac * yba - xac * xba);
    double s = ( yac * xba - xac * yba) / l;
    return (fabs(s) < resolution && r > -resolution && r < lsqr +
            resolution);
}

bool LineSegment::isPtinLine(Vertex vtx)
{
    bool rValue = false;
    float t;
    if(vtx.x-vertex0.x)
        t = (vtx.x - vertex0.x) / (vertex1.x - vertex0.x);
    else
        t = (vtx.y - vertex0.y) / (vertex1.y - vertex0.y);
    if(t>0.0f && t<1.0f)
        rValue = true;
    return rValue;
}

LineSegment LineSegment::Left(float halfThickness)
{
    Vertex left1, left2;
    Line base(vertex0, vertex1); base.normalize();
    left1 = Vertex(vertex0.x - base.r * halfThickness, vertex0.y -
        base.s * halfThickness);
    left2 = Vertex(vertex1.x - base.r * halfThickness, vertex1.y -
        base.s * halfThickness);
    LineSegment newSegment(left1, left2);
    return newSegment;
}

LineSegment LineSegment::Right(float halfThickness)
{
    Vertex right1, right2;
    Line base(vertex0, vertex1); base.normalize();
    right1 = Vertex(vertex0.x + base.r * halfThickness, vertex0.y +
        base.s * halfThickness);
    right2 = Vertex(vertex1.x + base.r * halfThickness, vertex1.y +
        base.s * halfThickness);
    LineSegment newSegment(right1, right2);
    return newSegment;
}

float LineSegment::Length(void)
{
    return (float)vertex0.distance(vertex1);
}

```

```

////////////////////////////////////// POLYGON.H ////////////////////////////////////////
// Poly: a list of vertices representing a single polygon
//
// Constructors
//   Poly()
//   Poly(const Poly &p)
//
// Members
//   nextCirc(list<Vertex>::iterator &it) - next linkage in path
//   prevCirc(list<Vertex>::iterator &it) - previous link in path
//   size() - returns number of vertices in current polygon
//

class Poly
{
public:
    list<Vertex> vtxList;

    Poly(const Poly &p): vtxList(p.vtxList) {}
    Poly()
    {
    }
    void nextCirc(list<Vertex>::iterator &it) // for circular linkage
    {
        if (++it == vtxList.end())
            it = vtxList.begin();
    }
    void prevCirc(list<Vertex>::iterator &it) // for circular linkage
    {
        if (it == vtxList.begin())
            it = vtxList.end();
        --it;
    }
    void nextCirc(list<Vertex>::const_iterator &it) const
    {
        if (++it == vtxList.end())
            it = vtxList.begin();
    }
    void prevCirc(list<Vertex>::const_iterator &it) const
    {
        if (it == vtxList.begin())
            it = vtxList.end();
        --it;
    }
    unsigned size() const
    {
        return static_cast<unsigned>(vtxList.size());
    }
};

typedef list<Poly>::iterator PolyIt;
typedef list<Poly>::const_iterator PolyConstIt;

```

```

//////////////////////////////////// MULTIPOLYGON.H //////////////////////////////////////
// MultiPolygon: collection of polygons   Written by T. Hain
//
////////////////////////////////////
// Embedded class CMPPos: Position of vertex within multipolygon,
//   (POSITION polygon, POSITION vertex)
//
// Constructors
//   CMPPos()
//   CMPPos(const POSITION _polyPos, const POSITION _vtxPos)
//
// Comparators
//   bool operator== (const CMPPos pos) const
//   bool operator!= (const CMPPos pos) const
//
// Assignment operator
//   void operator= (const CMPPos pos)
//
// Predicate function to tell if position is valid
//   bool isValid() const
//
////////////////////////////////////
// Constructors
//   MultiPolygon()
//   MultiPolygon(const MultiPolygon& mPoly)
//
// Assignment
//   void operator= (const MultiPolygon& mPoly)
//
// Get position of multipolygon vertex within distance 'resolution' of
// a given vertex
//   MPPos GetActiveVtxPos (Vertex point, int resolution);
//
// For a given vertex. 'point', get position of multipolygon vertex v1
// of a line (v1,v2) such that it is within a distance 'resolution' of
// the line.
//   MPPos GetActiveEdgePos(Vertex point, int resolution)
//
// Returns polygon containing vertex at given CMPPos position
// Precondition: mpPos must be valid
//   Polygon& polygon(const CMPPos &mpPos)
//
// Returns vertex at given CMPPos position
// Precondition: mpPos must be valid
//   Vertex& vertex(const CMPPos &mpPos)

```

```

class MultiPoly
{
public:

    void dump() const;
    bool read(const char* file_name);

    bool write(const char* file_name);
    bool writePS(const char* file_name);
    list<Poly> m_polyList;
    struct MPPos
    {
        list<Poly>::iterator m_polyIt;
        list<Vertex>::iterator m_vtxIt;

        MPPos(): m_vtxIt(NULL), m_polyIt(NULL)
        {
        }
        MPPos(PolyIt pi, VtxIt vi): m_polyIt(pi), m_vtxIt(vi) {}
        bool operator== (const MPPos pos) const
        {
            return m_vtxIt == pos.m_vtxIt && m_polyIt == pos.m_polyIt;
        }
        bool operator!= (const MPPos pos) const
        {
            return !(*this == pos);
        }
        void operator= (const MPPos pos)
        {
            m_vtxIt = pos.m_vtxIt;
            m_polyIt = pos.m_polyIt;
        }
        bool isValid() const
        {
            return m_polyIt != NULL;
        }
    };
};

MultiPoly(const MultiPoly &mp): m_polyList(mp.m_polyList) {}
MultiPoly() {}
MPPos GetActiveVtxPos (const Vertex &vtx, Coord resolution);
MPPos GetActiveEdgePos(const Vertex &vtx, Coord resolution);
Rect boundingBox(void) const;
void getEdgeList();
void getIntersectionList();
// Snap all vertex coordinates to given resolution
void gridify(Coord resolution);
void removeInlineVtxs(void);

```

```

struct AllIntersectionsVecElem
{
    enum {INVALID};
    Vertex m_intersection;
    int m_ndxEdge0, m_ndxEdge1;    // index of edge into edge list.
    union
    {
        // parametric value of intersection on edge0.
        float m_alpha0;
        // index of next x point in output poly
        int m_ndxNextIntersection0;
    };
    union
    {
        // parametric value of intersection on edge1.
        float m_alpha1;
        // index of next intersection point in output poly
        int m_ndxNextIntersection1;
    };

    vector<AllIntersectionsVecElem> m_allIntersectionsVec;

    AllIntersectionsVecElem(Vertex vtx, int i, int j, float alpha0,
        float alpha1)
        :m_intersection(vtx),m_ndxEdge0(i),m_ndxEdge1(j), m_alpha0
        (alpha0), m_alpha1(alpha1) {}
    bool operator < (const AllIntersectionsVecElem &e) const
        {return m_intersection < e.m_intersection;}
};

struct AllEdgesVecElem: public LineSegment
{
    struct IntersectionVecElem
    {
        float m_alpha;
        unsigned m_ndxAllIntersectionsVec;
        IntersectionVecElem(float alpha, unsigned ndx):
            m_alpha(alpha), m_ndxAllIntersectionsVec(ndx) {}
        bool operator < (const IntersectionVecElem &e)
            {return m_alpha < e.m_alpha;}
    };
    vector<IntersectionVecElem> m_intersectionVec;
    AllEdgesVecElem(const Vertex &v0, const Vertex &v1)
        :LineSegment(v0,v1) {}
};
vector<AllEdgesVecElem> m_allEdgesVec;
void linkIntersectionList(void);
};

typedef MultiPoly::AllEdgesVecElem AllEdgesVecElem;
typedef MultiPoly::AllIntersectionsVecElem AllIntersectionsVecElem;
typedef MultiPoly::AllEdgesVecElem::IntersectionVecElem
IntersectionVecElem;

```

```

//////////////////////////////////////  MULTIPOLYGON.CPP  ////////////////////////////////////////

MultiPoly::MPPos MultiPoly::GetActiveVtxPos(const Vertex &vtx, Coord
resolution)
// Find first vertex such that vtx is within a distance 'resolution'
// Output MPPos for that 'active' vertex, or invalid if none is found
{
    Rect activeArea(vtx, vtx);
    activeArea.inflate(resolution, resolution);
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
        for (VtxIt vi = pi->vtxList.begin(); vi != pi->vtxList.end();
            ++vi)
            if (activeArea.isVertexInside(*vi))
                return MPPos(pi,vi);
    return MPPos(NULL,NULL);
}

MultiPoly::MPPos MultiPoly::GetActiveEdgePos(const Vertex &vtx, Coord
resolution)
// Find first edge such that vtx is within a distance 'resolution'
// Output (activePolyIter,activeVtxIter) is only valid if active edge
// is found, and refers to vertex at one end of edge. Vertex at other
// end of edge is (activePolyPos,GetNextCirc(activeVtxPos)).
// Returns true if an active edge is found.
{
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        VtxIt vi = pi->vtxList.begin();
        Vertex v0 = *vi;
        for (unsigned i = 0; i < static_cast<unsigned>(pi->size()); ++i)
        {
            pi->prevCirc(vi);
            Vertex v1 = *vi;
            LineSegment l(v0,v1);
            if (l.isVertexNear(vtx, resolution))
                return MPPos(pi,vi);
            v0 = v1;
        }
    }
    return MPPos(NULL,NULL);
}

bool MultiPoly::writePS(const char *file_name)
{
    if(!m_polyList.empty())
    {
        ofstream out(file_name);
        out << "/poly {newpath";
        for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end();
            ++pi)
        {
            VtxIt vi = pi->vtxList.begin();
            float beginX = vi->x, beginY = vi->y;

```

```

out << "\n\t" << beginX << " " << beginY << " moveto\n";
    for(++vi; vi != pi->vtxList.end(); ++vi)
    {
        out << "\t" << vi->x << " " << vi->y << " lineto\n";
    }
    out << "\t" << beginX << " " << beginY << " lineto\n";
}
out << "\tclosepath\n";

out << "} def\n\t0 0 translate\n\t90 rotate\n\t0.8 setgray\n\t"
    << " poly fill\n\t0 setgray\n\tpoly stroke\n\tshowpage";
return true;
}
return false;
}

bool MultiPoly::write(const char *file_name)
{
    ofstream out(file_name);
    out << static_cast<unsigned>(m_polyList.size()) << endl;
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        out << static_cast<unsigned>(pi->vtxList.size()) << " ";
        for (VtxIt vi = pi->vtxList.begin(); vi != pi->vtxList.end();
            ++vi)
            out << vi->y << " " << vi->x << " ";
        out << endl;
    }
    return true;
}

bool MultiPoly::read(const char *file_name)
{
    ifstream in(file_name);
    m_polyList.clear();
    list<Poly>::size_type nPoly(*istream_iterator<typedef list<Poly>::
        size_type>(in));
    for (list<Poly>::size_type p = 0; p < nPoly; ++p)
    {
        Poly poly;
        list<Vertex>::size_type nVtx(*istream_iterator<list<Vertex>::
            size_type>(in));
        for (list<Vertex>::size_type v = 0; v < nVtx; ++v)
        {
            Vertex vtx(*istream_iterator<Coord>(in), *istream_iterator
                <Coord>(in));
            poly.vtxList.push_back(vtx);
        }
        m_polyList.push_back(poly);
    }
    return true;
}
}

```

```

////////////////////////////////////// POLYLINE.H ////////////////////////////////////////
// Written by T. Hain, extended by Swaminadhan
//
// PolyLine: polyline; a sequence of line segments defined by a
// sequence of vertices
// Inherits CList<Vertex, Vertex>
//
// Constructors
// PolyLine() // default constructor
// PolyLine(const PolyLine& polyline) // copy constructor
// ~PolyLine() // destructor
//
// Assignment
// void operator= (const PolyLine& poly)
//

typedef CList<Vertex, Vertex> VertexList;

class PolyLine: public VertexList
{
public:
    PolyLine(): VertexList() {}
    PolyLine(const PolyLine& polyline);
    void operator= (const PolyLine& polyline);
    virtual ~PolyLine() {}
};

typedef CList<PolyLine, PolyLine> PolylineList;

//////////////////////////////////////
// CMultiCPolyLine: collection of CPolyLines
// Inherits CList<PolyLine, PolyLine>
//
//////////////////////////////////////
// Embedded class MPPos: Position of vertex within multipolyline,
// (POSITION polyline, POSITION vertex)
//
// Constructors
// MPPos()
// MPPos(const POSITION _polyPos, const POSITION _vtxPos)
//
// Comparators
// BOOL operator== (const MPPos pos) const
// BOOL operator!= (const MPPos pos) const
//
// Assignment opertator
// void operator= (const MPPos pos)
//
// Predicate function to tell if position is valid
// BOOL isValid() const
//

```

```

////////////////////////////////////
// Constructors

// MultiPolyline()
// MultiPolyline(const MultiPolyline& mPoly)
//
// Assignment
// void operator= (const MultiPolyline& mPoly)
//
// MPPos GetActiveMPPos(Vertex point, int resolution)
// Returns MPPos of first vertex in multipolyline within distance
// "resolution" of point "point", or null MPPos if there are none.
//
// Polyline& polyline(const MPPos &mpPos)
// Returns polyline at given MPPos position
//
// Vertex& vertex(const MPPos &mpPos)
// Returns vertex at given MPPos position

class MultiPolyline: public PolylineList
{
public:
    MultiPolyline(): PolylineList() {}
    MultiPolyline(const MultiPolyline &mPoly);
    void operator= (const MultiPolyline& mpolyline);

    struct MPPos
    {
        enum ActiveState {ACTIVE_NOthing, ACTIVE_VERTEX, ACTIVE_EDGE};
        ActiveState m_activeState;
        POSITION m_polyPos;
        POSITION m_vtxPos;
        MPPos(const POSITION _polyPos = NULL, const POSITION _vtxPos =
            NULL, ActiveState _activeState = ACTIVE_NOthing)
            :m_polyPos(_polyPos), m_vtxPos(_vtxPos),
            m_activeState(_activeState) {}
        BOOL operator== (const MPPos pos) const
        { return m_vtxPos == pos.m_vtxPos && m_polyPos == pos.m_polyPos;}

        BOOL operator!= (const MPPos pos) const
        { return !(*this == pos);}
        void operator= (const MPPos pos)
        { m_vtxPos = pos.m_vtxPos; m_polyPos = pos.m_polyPos;
          m_activeState = pos.m_activeState; }
    };

    MPPos GetActivePos(Vertex point, int resolution);
    PolyLine& polyline(const MPPos mpPos)
        { return GetAt(mpPos.m_polyPos); }
    Vertex& vertex(const MPPos mpPos)
        { return polyline(mpPos).GetAt(mpPos.m_vtxPos); }
    Vertex& nextVertex(const MPPos mpPos)
        { polyline(mpPos).GetNext((POSITION)mpPos.m_vtxPos);
          return polyline(mpPos).GetPrev((POSITION)mpPos.m_vtxPos); }
    virtual ~MultiPolyline() {}
}

```

```

    bool read(const char * infile, float *nMiterLimit, int *nHalfThick,
              int *imp_MFC);
    bool write(const char *outfile, float miterlimit, int halfThickness,
              int implement_MFC);
    bool writePS(const char * file_name, float miterlimit, int
              thickness);
};

////////////////////////////////////
// Written by S. Swaminadhan
//
// ThickPolyline: thickPolyline - structure to hold thick polyline
//   attributes
//
// Attributes: halfThickness - half thickness of thick segment outline
//             miterLimit    - numerical miter limit value
//             miterLength   - numerical maximum miter length allowed
//
// ThickPolyline extends MultiPolyline and has access to its methods
//
// Constructors:
//   operator= (const MultiPolyline& mpolyline) - copy constructor
//
// Set methods:
//   setThickness(float lineWidth) - sets the half thickness attribute
//   setMiterLimit(float newLimit) - sets new miter limit value
//   setMiterLength(float newLength) - sets new miter length value
//
// Create outline for thick polyline using research algorithm
//   MultiPoly CreateOutline(bool doBevel, bool m_SquareCap)
//
class ThickPolyline: public MultiPolyline
{
public:
    float halfThickness;
    float miterLimit; // Limit = miterLength/thickness
    float miterLength; // Postscript default - 11° connection angle

    ThickPolyline(): halfThickness(5.0f), miterLimit(10.0f),
        miterLength(miterLimit){}
    //ThickPolyline(MultiPolyline mPoly, int n_Thickness);

    void setThickness(float lineWidth) {halfThickness = lineWidth/2.0f;
        setMiterLength(halfThickness*2*miterLimit);}
    void setMiterLimit(float newLimit) {miterLimit = newLimit;}
    void setMiterLength(float newLength) {miterLength = newLength;}
    void operator= (const MultiPolyline& mpolyline);
    MultiPoly CreateOutline(bool doBevel, bool m_SquareCaps);
};

```

```

////////////////////////////////////// POLYLINE.CPP ////////////////////////////////////////
// PolyLine members

void PolyLine::operator= (const PolyLine& polyline)
{
    if (this != &polyline)
    {
        RemoveAll();
        for (POSITION p = polyline.GetHeadPosition(); p; )
            AddTail((Vertex)polyline.GetNext(p));
    }
}

PolyLine::PolyLine(const PolyLine& polyline)
{
    for (POSITION p = polyline.GetHeadPosition(); p; )
        AddTail((Vertex)polyline.GetNext(p));
}

//////////////////////////////////////
// MultiPolyline members

void MultiPolyline::operator= (const MultiPolyline& mpolyline)
{
    if (this != &mpolyline)
    {
        RemoveAll();
        for (POSITION p = mpolyline.GetHeadPosition(); p; )
            AddTail((PolyLine)mpolyline.GetNext(p));
    }
}

MultiPolyline::MultiPolyline(const MultiPolyline &mpolyline)
{
    for (POSITION p = mpolyline.GetHeadPosition(); p; )
        AddTail((PolyLine)mpolyline.GetNext(p));
}

MultiPolyline::MPPos MultiPolyline::GetActivePos(Vertex point, int
    resolution)
// Find first vertex or edge such that point is within a distance
// 'resolution' of it. The position of the vertex (or first vertex of
// edge) is returned. activeState is set to one of ACTIVE_NOTHING,
// ACTIVE_VERTEX, ACTIVE_EDGE.
{
    CRect activeArea((CPoint)point, (CPoint)point);
    activeArea.InflateRect(resolution, resolution);

    // See if point is close to vertex
    for(MPPos mpPos(GetHeadPosition(), NULL, MPPos::ACTIVE_VERTEX);
        mpPos.m_polyPos; GetNext(mpPos.m_polyPos))
        for(mpPos.m_vtxPos = polyline(mpPos).GetHeadPosition();
            mpPos.m_vtxPos; polyline(mpPos).GetNext(mpPos.m_vtxPos))

            if (activeArea.PtInRect(vertex(mpPos)))
                return mpPos;
}

```

```

// See if point is near edge
for(MPPos mpPos(GetHeadPosition(), NULL, MPPos::ACTIVE_EDGE);
   mpPos.m_polyPos; GetNext(mpPos.m_polyPos))
  if (polyline(mpPos).GetSize() > 1)
    for(mpPos.m_vtxPos = polyline(mpPos).GetHeadPosition();
       mpPos.m_vtxPos != polyline(mpPos).GetTailPosition();
       polyline(mpPos).GetNext(mpPos.m_vtxPos))
      {
        Vertex &vtx1 = polyline(mpPos).GetNext(mpPos.m_vtxPos),
          &vtx2 = polyline(mpPos).GetPrev(mpPos.m_vtxPos);
        if (Vertex(point).isNearLine(vtx1, vtx2, resolution))
          return mpPos;
      }
return MPPos();
}

bool MultiPolyline::read(const char* infile, float *nMiterLimit, int
 *nHalfThick, int *imp_MFC)
{
  ifstream in(infile);
  unsigned nPoly;
  in >> *nMiterLimit;
  in >> *nHalfThick;
  in >> *imp_MFC;
  in >> nPoly;
  for (unsigned p = 0; p < nPoly; ++p)
  {
    PolyLine pLine;
    unsigned nVtx;
    in >> nVtx;
    for (unsigned v = 0; v < nVtx; ++v)
    {
      Vertex vtx;
      in >> vtx.x >> vtx.y;
      pLine.AddTail(vtx);
    }
    AddTail(pLine);
  }
  return true;
}

bool MultiPolyline::write(const char *outfile, float miterlimit, int
 halfThickness, int implement_MFC)
{
  PolyLine pl;
  Vertex vtx;
  ofstream out(outfile);
  out << miterlimit << "\n";
  out << halfThickness << "\n";
  out << implement_MFC << "\n";

  out << (int)GetSize() << "\n";
}

```

```

for (POSITION mPos = GetHeadPosition(); mPos; )
{
    pl = GetNext(mPos);

    out << (int)pl.GetSize() << " ";
    for(POSITION pos = pl.GetHeadPosition(); pos; )
    {
        vtx = pl.GetNext(pos);
        out << vtx.x << " " << vtx.y << " ";
    }
    out << "\n";
}
out << "\n";
return true;
}

bool MultiPolyline::writePS(const char * file_name, float miterlimit,
int thickness)
{
    ofstream out(file_name);
    out << "/path\n{newpath}";
    PolyLine pl;
    Vertex vtx;
    for (POSITION mPos = GetHeadPosition(); mPos; )
    {
        pl = GetNext(mPos);
        POSITION pos = pl.GetHeadPosition();
        vtx = pl.GetNext(pos);
        // 90° CCW rotation
        out << "\n\t" << vtx.y * -1 << " " << vtx.x << " moveto\n";

        for(pos; pos; )
        {
            vtx = pl.GetNext(pos);
            out << "\t" << vtx.y * -1 << " " << vtx.x << " lineto\n";
        }
    }
    out << "}def\n";
    out << "\n0.75 0.75 scale\n\npath\n0 0 translate\n0 rotate\n "
        << "\ n0.8 setgray " << miterlimit << " setmiterlimit"
        << "\n" << thickness << " setlinewidth"
        << "\nstroke";

    out << "\n\npath\n0 setgray"
        << "\n" << 2 << " setlinewidth"
        << "\nstroke\nshowpage";

    return true;
}

```

```

////////////////////////////////////
// ThickPolyline members

void ThickPolyline::operator= (const MultiPolyline& mpolyline)
{
    RemoveAll();
    for (POSITION p = mpolyline.GetHeadPosition(); p; )
        AddTail((PolyLine)mpolyline.GetNext(p));
}

MultiPoly ThickPolyline::CreateOutline(bool doBevel, bool m_SquareCaps)
{
    Poly outlinePoly;
    PolyLine pl;
    MultiPoly mPoly;
    LineSegment begin, end;
    Vertex midPoint, aLeft, aRight, bLeft, bRight;
    Line firstbase, firstLeft, secondbase, secondLeft;
    SIDE DIR=LEFT;
    bool isStraight, shortSegment;
    int nVertices, nLines;

    POSITION mPos = GetHeadPosition(); // Select Polyline
    nLines = (int)GetSize();
    do
    {
        pl = GetNext(mPos);
        --nLines;
        isStraight = false;
        nVertices = (int)pl.GetSize();

        // Select first line segment (AB)
        POSITION pos = pl.GetHeadPosition();
        begin.vertex0 = pl.GetNext(pos);
        begin.vertex1 = pl.GetNext(pos);
        nVertices = nVertices - 2;

        // Create A'A" at distance d on line perpendicular to AB
        firstbase = Line(begin.vertex0, begin.vertex1);
        firstbase.normalize();
        firstLeft = Line(firstbase.parallel(halfThickness, LEFT));
        firstLeft.normalize();

        aLeft = Vertex(begin.vertex0.x - firstbase.r * halfThickness,
            begin.vertex0.y - firstbase.s * halfThickness);
        aRight = Vertex(begin.vertex0.x + firstbase.r * halfThickness,
            begin.vertex0.y + firstbase.s * halfThickness);

        if(m_SquareCaps)
        {
            LineSegment shift = LineSegment(aLeft, aRight).Right(
                halfThickness);

            aLeft = shift.vertex0;
            aRight = shift.vertex1;
        }
    }
}

```

```

// Insert A'A" into outline polygon
outlinePoly.vtxList.push_front(aLeft);
outlinePoly.vtxList.push_back(aRight);

if(nVertices == 0) // Nothing more to process
    end = begin;

// Traverse upto the last point in polyline
while(nVertices > 0)
{
    // Select next line segment (BC)
    end.vertex0 = begin.vertex1;
    end.vertex1 = pl.GetNext(pos);
    --nVertices; shortSegment = false;

    if(end.vertex0 == end.vertex1)
    {
        --nVertices;
        continue;
    }
    isStraight = false;
    midPoint = end.vertex0;

    // Check direction of turn of the line
    if(firstbase.isInline(end.vertex1)) // If in line,
    {
        isStraight = true;

        if(LineSegment(begin.vertex0, end.vertex1).isPtinLine(
            begin.vertex1)) // And same direction
        {
            begin = end;
            continue;
        }
    }
}

```

```

else // Opposite direction
{
    bLeft = Vertex(end.vertex0.x - firstbase.r *
        halfThickness, end.vertex0.y - firstbase.s *
        halfThickness);
    bRight = Vertex(end.vertex0.x + firstbase.r *
        halfThickness, end.vertex0.y + firstbase.s *
        halfThickness);
    outlinePoly.vtxList.push_front(bLeft);
    outlinePoly.vtxList.push_back(bRight);

    // Close existing polygon
    mPoly.m_polyList.push_back(outlinePoly);
    outlinePoly.vtxList.clear();
    // Start fresh polygon
    outlinePoly.vtxList.push_back(bLeft);
    outlinePoly.vtxList.push_front(bRight); // Dir changes
    begin = end;
    secondbase = firstbase;
    firstLeft = firstbase.parallel(halfThickness, RIGHT);
    continue;
}
}
else
{
    if(begin.isLeftOf(end.vertex1))
        DIR = LEFT;
    else
        DIR = RIGHT;
}

// Create left parallel line to latter line segment
secondbase = Line(end.vertex0, end.vertex1);
secondLeft = Line(secondbase.parallel(halfThickness, LEFT));

// Find intersection between two left parallel lines
bLeft = firstLeft.intersect(secondLeft);
bRight = midPoint.project(bLeft, -1.0f);

// Needed to check Anamoly
LineSegment beginLeft = begin.Left(halfThickness);
LineSegment beginRight = begin.Right(halfThickness);
LineSegment endLeft = end.Left(halfThickness);
LineSegment endRight = end.Right(halfThickness);

Line firstRight = firstbase.parallel(halfThickness, RIGHT);
Line secondRight = secondbase.parallel(halfThickness, RIGHT);

// Check bevel requirement
if(bLeft.distance(bRight) <= miterLength && doBevel == false)
{ // No Bevel required

    if(end.Length() > halfThickness*2)
    {

```

```

// Check for anomaly here
if(DIR == LEFT && (!beginLeft.isPtinLine(bLeft)
|| !endLeft.isPtinLine(bLeft)) )
{
    if(end.Length() > halfThickness*1.5)
    {
        Vertex link1 = firstLeft.intersect(secondRight);
        Vertex link2 = firstRight.intersect(secondLeft);

        outlinePoly.vtxList.push_front(link1);
        outlinePoly.vtxList.push_front(bRight);

        // Close existing polygon
        mPoly.m_polyList.push_back(outlinePoly);
        outlinePoly.vtxList.clear();
    // Start fresh polygon
        outlinePoly.vtxList.push_front(link2);
        outlinePoly.vtxList.push_back(link1);
    }

    else
    {
        // close first polygon
        outlinePoly.vtxList.push_front(beginLeft.vertex1);
        outlinePoly.vtxList.push_back(beginRight.vertex1);
        mPoly.m_polyList.push_back(outlinePoly);
        outlinePoly.vtxList.clear();
        // create extension
        // close second polygon
    }
}

else if(DIR == RIGHT && (!beginRight.isPtinLine(
bRight) || !endRight.isPtinLine(bRight)))
{
    Vertex link1 = firstLeft.intersect(secondRight);
    Vertex link2 = firstRight.intersect(secondLeft);

    outlinePoly.vtxList.push_front(bLeft);
    outlinePoly.vtxList.push_back(link2);

    mPoly.m_polyList.push_back(outlinePoly);
    outlinePoly.vtxList.clear();
    outlinePoly.vtxList.push_front(link2);
    outlinePoly.vtxList.push_back(link1);
}

else
{
    outlinePoly.vtxList.push_front(bLeft);
    outlinePoly.vtxList.push_back(bRight);
}
}

```

```

else
{
    shortSegment = true;
    outlinePoly.vtxList.push_front(beginLeft.vertex1);
    outlinePoly.vtxList.push_back(beginRight.vertex1);
    mPoly.m polyList.push_back(outlinePoly);
    outlinePoly.vtxList.clear();

    if(DIR == LEFT)
    {
        outlinePoly.vtxList.push_front(firstRight.intersect
            (secondRight));
        outlinePoly.vtxList.push_front(beginRight.vertex1);
        Vertex test = firstLeft.intersect(secondLeft);

        if(endLeft.isPtinLine(test))
            outlinePoly.vtxList.push_front(test);

        else
            outlinePoly.vtxList.push_front(firstRight.
                intersect(secondLeft));
    }
    else
    {
        Vertex test = firstLeft.intersect(secondLeft);
        outlinePoly.vtxList.push_front(test);

        if(!beginLeft.isPtinLine(test))
            outlinePoly.vtxList.push_back(beginLeft.vertex1);
        test = firstRight.intersect(secondRight);
        if(endRight.isPtinLine(test))
            outlinePoly.vtxList.push_back(test);
        else
            outlinePoly.vtxList.push_back(firstLeft.intersect
                (secondRight));
    }
}
}

```

```

else // BEVEL
{
    double dist = bLeft.distance(bRight);
    Vertex ptBevelLeft, ptBevelRight;
    Line lineBevel;

    if(end.Length() > halfThickness*2)
    {

        if(DIR==LEFT)
        {
            ptBevelLeft = endRight.vertex0;
            ptBevelRight = beginRight.vertex1;

            if(!beginLeft.isPtinLine(bLeft) ||
                !endLeft.isPtinLine(bLeft))
            {
                Vertex link1 = firstLeft.intersect(secondRight);
                Vertex link2 = firstRight.intersect(secondLeft);

                outlinePoly.vtxList.push_front(link1);
                outlinePoly.vtxList.push_front(ptBevelLeft);
                outlinePoly.vtxList.push_back(ptBevelRight);

                mPoly.m_polyList.push_back(outlinePoly);
                outlinePoly.vtxList.clear();
                outlinePoly.vtxList.push_front(link2);
                outlinePoly.vtxList.push_back(link1);
            }

            else
            {
                outlinePoly.vtxList.push_front(bLeft);
                outlinePoly.vtxList.push_back(ptBevelRight);
                outlinePoly.vtxList.push_back(ptBevelLeft);
            }
        }

        else // DIR = RIGHT
        {
            ptBevelLeft = beginLeft.vertex1;
            ptBevelRight = endLeft.vertex0;

            if(!beginRight.isPtinLine(bRight) ||
                !endRight.isPtinLine(bRight))
            {
                Vertex link1 = firstLeft.intersect(secondRight);
                Vertex link2 = firstRight.intersect(secondLeft);

                outlinePoly.vtxList.push_front(ptBevelLeft);
                outlinePoly.vtxList.push_front(ptBevelRight);
                outlinePoly.vtxList.push_back(link2);

                mPoly.m_polyList.push_back(outlinePoly);
                outlinePoly.vtxList.clear();
            }
        }
    }
}

```

```

        outlinePoly.vtxList.push_front(link2);
        outlinePoly.vtxList.push_back(link1);
    }

    else
    {
        outlinePoly.vtxList.push_front(ptBevelLeft);
        outlinePoly.vtxList.push_front(ptBevelRight);
        outlinePoly.vtxList.push_back(bRight);
    }
}

else
{
    shortSegment = true;
    outlinePoly.vtxList.push_front(beginLeft.vertex1);
    outlinePoly.vtxList.push_back(beginRight.vertex1);
    mPoly.m_polyList.push_back(outlinePoly);
    outlinePoly.vtxList.clear();

    if(DIR == LEFT)
    {
        outlinePoly.vtxList.push_back(beginRight.vertex1);
        outlinePoly.vtxList.push_back(endRight.vertex0);
        Vertex test = firstLeft.intersect(secondLeft);
        if(endLeft.isPtinLine(test))

            outlinePoly.vtxList.push_front(test);
        else
            outlinePoly.vtxList.push_front(firstRight
                .intersect(secondLeft));
    }

    else // DIR = RIGHT
    {
        outlinePoly.vtxList.push_back(beginLeft.vertex1);
        outlinePoly.vtxList.push_front(endLeft.vertex0);
        Vertex test = firstRight.intersect(secondRight);
        if(endRight.isPtinLine(test))
            outlinePoly.vtxList.push_back(test);
        else
            outlinePoly.vtxList.push_back(firstLeft.intersect
                (secondRight));
    }
}

}

if(nVertices >= 1)
{
    begin = end;
    firstbase = secondbase;
    firstLeft = secondLeft;
}
}

```

```

// Last vertex in current polyline
secondbase = Line(end.vertex0, end.vertex1);
secondbase.normalize();

Vertex cLeft = Vertex(end.vertex1.x - secondbase.r *
    halfThickness, end.vertex1.y - secondbase.s * halfThickness);
Vertex cRight = Vertex(end.vertex1.x + secondbase.r *
    halfThickness, end.vertex1.y + secondbase.s * halfThickness);

if(m_SquareCaps)
{
    LineSegment shift = LineSegment(cLeft, cRight).Left(
        halfThickness);
    cLeft = shift.vertex0;
    cRight = shift.vertex1;
}

if(!isStraight && (int)pl.GetSize() > 2 && !shortSegment)
{
    LineSegment beginLeft = begin.Left(halfThickness);
    LineSegment beginRight = begin.Right(halfThickness);
    LineSegment endLeft = end.Left(halfThickness);
    LineSegment endRight = end.Right(halfThickness);

    Line firstRight = firstbase.parallel(halfThickness, RIGHT);
    Line secondRight = secondbase.parallel(halfThickness, RIGHT);

    Vertex link1 = firstLeft.intersect(secondRight);
    Vertex link2 = firstRight.intersect(secondLeft);

    if(DIR == LEFT)
    {
        if(!beginLeft.isPtinLine(bLeft) ||
            !endLeft.isPtinLine(bLeft))
        {
            outlinePoly.vtxList.pop_front();
            outlinePoly.vtxList.push_front(link1);

            mPoly.m_polyList.push_back(outlinePoly);
            outlinePoly.vtxList.clear();
            outlinePoly.vtxList.push_front(link2);
            outlinePoly.vtxList.push_back(link1);
        }
    }
    else

```

```

    {
        if(!beginRight.isPtinLine(bRight) ||
            !endRight.isPtinLine(bRight))
        {
            outlinePoly.vtxList.pop_back();
            outlinePoly.vtxList.push_back(link2);

            mPoly.m_polyList.push_back(outlinePoly);
            outlinePoly.vtxList.clear();
            outlinePoly.vtxList.push_front(link2);
            outlinePoly.vtxList.push_back(link1);
        }
    }

    // Insert C'C" into outline polygon
    outlinePoly.vtxList.push_front(cLeft);
    outlinePoly.vtxList.push_back(cRight);

    mPoly.m_polyList.push_front(outlinePoly);
    outlinePoly.vtxList.clear();
} while(nLines > 0);

return mPoly;
}

```

Appendix B

Invoking the Algorithm

The research algorithm can be invoked by calling the CreateOutline routine on a created thick polyline. The research algorithm requires two boolean values to calculate the polygonal outline – the bevel setting and the end cap setting. In this appendix, the pseudocode that was used to create the resulting outline on a given thick polyline, and the way Windows was invoked to draw the same outline is provided.

```
PolygonEditorViewClass::OnDraw (pointer to Drawing Canvas)
{
    if (there is a polyline to process)
    {
        if(WIN32_IMPLEMENTATION is ON)                // Draw all edges
        {
            // Draw thick line using MFC
            if(BEVEL_JOIN is ON)
            {
                if(SQUARE_CAP is ON)
                    Create brush with bevel join and square cap attributes
                else
                    Create brush with bevel join and flat cap attributes
            }
            else
            {
                if(SQUARE_CAP is ON)
                    Create brush with miter join and square cap attributes
                else
                    Create brush with miter join and flat cap attributes
            }
        }

        Set drawing canvas to draw path lines
        For every polyline in created MultiPolyline,
            Move pen to first point of polyline
            For every point in the polyline
                Draw a line with selected pen
        Stroke path with selected brush

        // Draw base polyline
        Create grey pen
        For every polyline in MultiPolyline
            Move pen to first point in the polyline
            For every point in the polyline
                Draw a line with selected pen
    }
}
```

```

    Select old pen
}

else
{
    if(FLATTEN_OUTLINE is OFF)
    {
        For every polyline in created MultiPolyline,
        Move pen to first point of polyline
        For every point in the polyline
        Draw a line with selected pen
    }
}

if (FLATTEN_OUTLINE is OFF and DRAW_MARKERS is ON)
{
    For every polyline in created MultiPolyline,
    Move pen to first point of polyline
    For every point in the polyline
    Draw a small square rectangle at point with selected pen
}

// Draw hovered marker or hovered edge
if (Mouse Cursor is on a polyline edge)
{
    Select Red Pen
    Get edge at which cursor is present
    Redraw that edge with red pen
    Select old pen
}
else if (Mouse Cursor is on a polyline vertex)
    Redraw the selected vertex with a Red Pen

if(RESEARCH_ALGORITHM is ON) // RESEARCH ALGORITHM IS INVOKED
{
    // Create outline for polyline
    ThickPolyline tpl; // Create a thick polyline
    MultiPoly m_Poly; // Create a multi polyline
    tpl = MultiPolyline on Canvas;
    tpl.setMiterLimit(CURRENT_MITER_LIMIT);
    tpl.setThickness (CURRENT_LINE_THICKNESS);
    m_Poly = tpl.CreateOutline(BEVEL_JOIN, SQUARE_CAP);

    if(FLATTEN_OUTLINE is ON)
    {
        m_Poly = (Result of Hain's Polygon Union Algorithm);
    }
}

```

```

For every polygon in created MultiPolygon
  For every vertex in present polygon
    Draw a point on canvas representing current vertex
    Close off polygon by redrawing first vertex in polygon
  if(RESEARCH_ALGORITHM_FILL is ON)
  {
    Set Fill Mode to NON-ZERO WINDING FILL
    Fill created MultiPolygon outline with different color
  }

  if(RESEARCH_ALGORITHM_VERTICES is ON)
  {
    Create and select Blue Pen
    For every polygon in created MultiPolygon
      For every vertex in present polygon
        Draw a small rectangle representing current vertex
      }
    }
  }
}

```

BIOGRAPHICAL SKETCH

BIOGRAPHICAL SKETCH

Name of Author: Subramani Swaminadhan

Place of Birth: Kumbakonam, India

Date of Birth: March 31, 1976

Graduate and Undergraduate Schools Attended:
University of South Alabama, Mobile, Alabama
Bangalore University, Bangalore, India

Degrees Awarded:
Bachelor of Engineering in Electrical and Electronics, 1998, Bangalore, India

Awards and Honors:
Graduate Assistant, 2001-2003
Department-sponsored YΠE (Upsilon Pi Epsilon) member, 2002