UNIVERSITY OF SOUTH ALABAMA
SCHOOL OF COMPUTER & INFORMATION SCIENCES

PARTITION OF A NON-SIMPLE POLYGON INTO SIMPLE POLYGONS

BY

Lavanya Subramaniam

A Thesis

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

Master of Science
in
Computer Science

August 2003

Approved:

Chair of Thesis Committee: Dr. Thomas F. Hain	Date
Member of Committee: Dr. David D. Langan	Date
Member of Committee: Dr. Stephen G. Brick	Date
Dean of School of Computer & Information Sciences: Dr. David L. Feinstein	Date
Director of CIS Graduate Studies: Dr. Roy J. Daigle	Date
Dean of Graduate School: Dr. James L. Wolfe	Date

PARTITION OF A NON-SIMPLE POLYGON INTO SIMPLE POLYGONS

A Thesis

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirement for the
Master of Science
in
Computer Science

by

Lavanya Subramaniam
Bachelor of Engineering, Anna University, India, May 1999

School of Computer & Information Sciences
University of South Alabama

August 2003

This thesis is dedicated to my parent*s* and my teachers

ACKNOWLEDGEMENTS

My sincerest thanks to Dr.Hain, for his guidance, motivation, and patience for the past two and half years that I have worked with him. My thanks to the thesis committee – Dr. Langan , Dr.Fwu-shan shieh, Dr.Brick and Dr.Daigle for his motivation in CIS518 and throughout my thesis. I am immensely grateful to my parents for being so patient and understanding. My thanks to all my teachers, my relatives and friends who have helped me and been with me through all the times good and bad. Last but not the least, Balaji and my friends at Mobile, for being so helpful and caring, without whom this would have been very difficult.

TABLE OF CONTENTS

Page

LIST OF FIGURES ..... **vi**

ABSTRACT ..... **viii**

Chapter 1 Introduction **1**

1.1 Motivation for polygon partitioning 1
1.2 The motivation for the current work 2
1.3 Definitions 3
1.4 Current domain 4
1.5 Insideness of a point 4
1.5.1 Non-zero winding number rule 4
1.5.2 Even-odd parity rule 6
1.6 Methods for filling polygons 7
1.6.1 Image versus object precision 7
1.6.2 Current method 7
1.6.3 Proposed method 7
1.7 Filling polygons using scan conversions 8
1.7.1 Edge coherence 9
1.7.2 Other issues 9

Chapter 2 Related Work **10**

2.1 Introduction 10
2.2 Partitioning algorithms 10
2.3 Line intersections 11
2.4 Other sources 11

Chapter 3 Methodology **13**

3.1 Aspects of work 13
3.2 Correctness 14
3.3 Performance 15
3.4 Hypotheses 16

Chapter 4 Theory **17**

4.1 Outline of NZW partitioning algorithm 17

4.1.1 Terminology 18
4.1.2 Algorithm overview 19
4.1.3 Example 21
4.2 Implementation Details 23
4.2.1 Intersection points 23
4.2.2 Intersection Linkage 24
4.2.3 Data Structures 24
4.2.3.1 Support classes 25
4.2.3.2 Intermediate classes 26
4.2.3.2.1 Setting up the intersection linkage: 27
4.2.3.3 Illustration of the intermediate classes using an example 29
Chapter 5 Results 31
5.1 Correctness 31
5.2 Performance 33
5.2.1 Theoretical complexity 33
5.2.2 Empirical relative performance 34
5.2.2.1 Performance test suite 34
5.2.2.2 Performance results 36
5.2.2.2.1 Performance versus polygon height 36
5.2.2.2.2 Performance versus number of vertices 38
5.2.2.2.3 Performance versus number of intersections 39
5.2.2.3 Relative Performance of NZW 40
5.3 Conclusions 41
5.4 Future work 42
APPENDIX 46
APPENDIX A Code 48
VITA 77

LIST OF FIGURES

Page

Figure 1 Domain of current work 2

Figure 2 Printer graphics [3] 3

Figure 3 Examples of simple and complex polygon 3

Figure 4 Non-zero winding number rule 5

Figure 5 Even-odd parity rule 6

Figure 6 Scan Line Algorithm 9

Figure 7 Interactive testing tool showing random input polygon 15

Figure 8 Pseudo-vertices x_1, and x_2 at intersection point X 18

Figure 9 (a) complex polygon with labeled polygon vertices (b) the result of the filling by NZW algorithm 20

Figure 10 (a) –(e) Component polygons of the example input polygon. (f) Overlapping component polygons 23

Figure 11 (a) `Vertex` object (b) `LineSegment` object, (c) `Polygon` object and (d) `MultiPolygon` object 25

Figure 12 The Polygon Edge Array for the Figure 9(a) 26

Figure 13 (a) `nVertex` object (b) `Intersection` object 27

Figure 14 (a) is the illustration of i_1 in the polygon edge array (b) is the intersection object of i_1 in the intersection master list 28

Figure 15 Complex polygon showing edge segments and pseudo-vertices 29

Figure 16 (a) intersection master list of pseudo-vertices (b) Shows various values in the intersection queue 30

Figure 17 NZW fill using NZW filling rule 32

Figure 18 SLA fill using NZW filling rule 32
Figure 19 The SLA fill using SLA filling rule.....33
Figure 20 NZW time *vs*. Height of the polygons (trend lines) 37
Figure 21 SLA time *vs.* Height of the polygons (trend lines).....37
Figure 22 Performace *vs.* Number of Vertices (50 intersections).....38
Figure 23 Execution Time *vs.* Number of Intersections (for 100-vertex polygons).....39
Figure 24 Performace Ratio of NZW vs. Number of Intersections (20-vertex polygons)40
Figure 25 Performance Ratio *vs.* Number of Intersections (86-vertex polygons) 41

ABSTRACT

Subramaniam, Lavanya, <Bachelors of Engineering, College of Engineering, Anna University, India, May 1999>, Partitioning a Non-simple Polygon into Simple Polygons, Chair of Committee: Dr. Thomas F. Hain.

While there are several algorithms to partition simple polygons into triangles, trapezoids, or monotone polygons, they do not handle self-intersecting (complex) polygons. In this work, we describe the theory, implementation, and testing of a new object-precision algorithm to partition a complex polygon into a set of simple polygons (possibly having holes) using the non-zero winding number rule to determine the "inside"ness of points in the polygon. The outline for the algorithm implemented and tested in this research was developed by Dr. Thomas Hain. This algorithm can be used as a preprocessing step to convert complex polygons into a set of simple polygons, which are then amenable to being processed by the algorithms mentioned above. We show that this algorithm yields better high resolution performance than the image-precision scan line algorithm[6]—the method currently used in practice.

Chapter 1
INTRODUCTION

1.1 Motivation for polygon partitioning

Efficient polygon partitioning has been one of the most outstanding open problems in 2-dimensional computational geometry, and is usually confined to the domain of simple polygons. Simple polygons are ones that have no self-intersecting edges, but that can nevertheless be quite convoluted. To solve a variety of problems, the target polygon is typically partitioned into a disjoint set of covering polygons of restricted form, such as triangles, trapezoids, convex polygons, monotonic polygons, or polygons having other desirable characteristics.

Thus, polygon partitioning is one of the pre-processing steps for many applications in computational geometry and hence in computer graphics. Some of the applications requiring such polygon partitioning are:

- Finding the area of a simple non-convex polygon.
- Discovering whether a point is inside a simple polygon.
- Finding the center of gravity of a simple polygon.
- Filling polygons.
- Discovering whether a polygon is simple (has no self-intersecting edges).

Some real-time applications have triangulation as one of their major components. For example, the Art Gallery problem, which deals with monitoring a convex region, involves triangulation to minimize the number of cameras required to cover the whole guarded area.

Although the partitioning problem seems simple, the solutions tend to be quite complicated. These problems are often complicated and there has been a lot of research [2], [9], [17] that has focused on achieving high efficiency, or low running-times.

1.2 The motivation for the current work

The goal of this research is to offer an efficient algorithm to partition complex polygon into simple polygons so that the domain of such sophisticated algorithms already developed, can be extended to include complex polygons as shown in **Figure 1**. In addition, there are polygon-processing algorithms, which result in complex polygons, which can be further processed after the application of this algorithm.

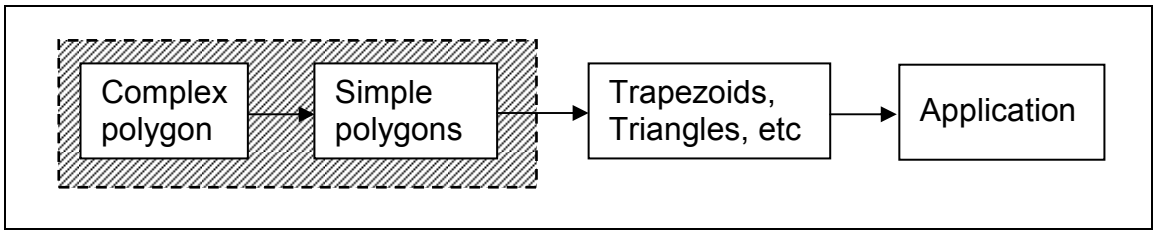

Figure 1 Domain of current work

In a larger context, we have taken as our domain of interest the printer industry. Here, high-level page objects (such as complex polygons) to be rendered are described in a page description language (PDL), such as PostScript©, HP PCL, or HP-GL. These objects are given to a graphics library which converts them into low-level graphical primitives (such as trapezoids) that are then stored in a display list. The objects in the display list can be quickly scan-converted (turned into pixels) into a printed image. The display list represents a semi-digested and compressed version of the page description, and can be efficiently scan-converted multiple times for multi-page, collated printing. The algorithm described here would reside in the graphics library, together with an already existing trapezoidation algorithm to convert simple polygons into lists of trapezoids.

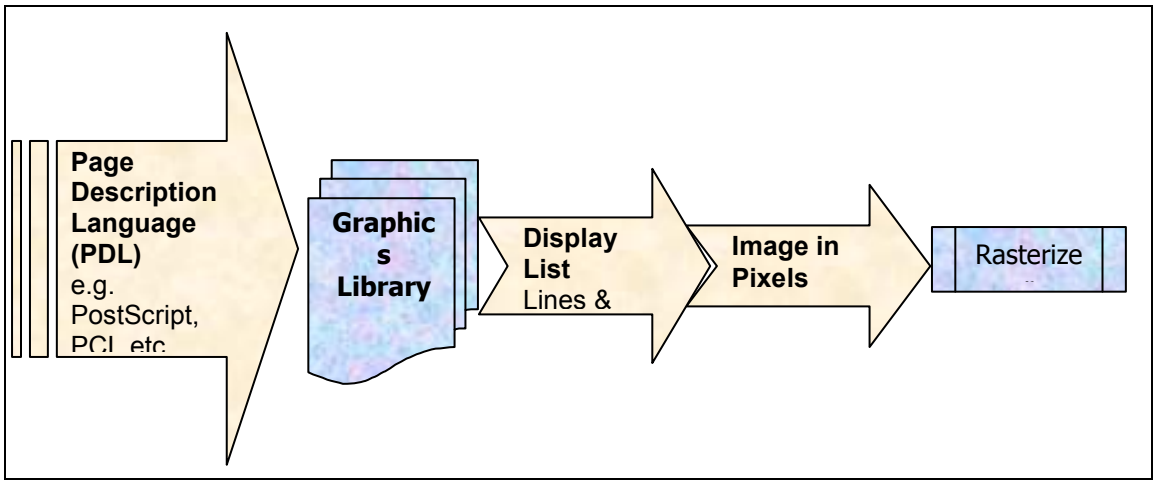

Figure 2 Printer graphics [3]

1.3 Definitions

A *simple polygon* is one in which no pair of edges cross each other. A *complex polygon* (sometimes called a *non-simple polygon*) may have crossing edges. Figure 3(a) gives an example of a simple polygon, and Figure 3(b) gives an example of a complex polygon.

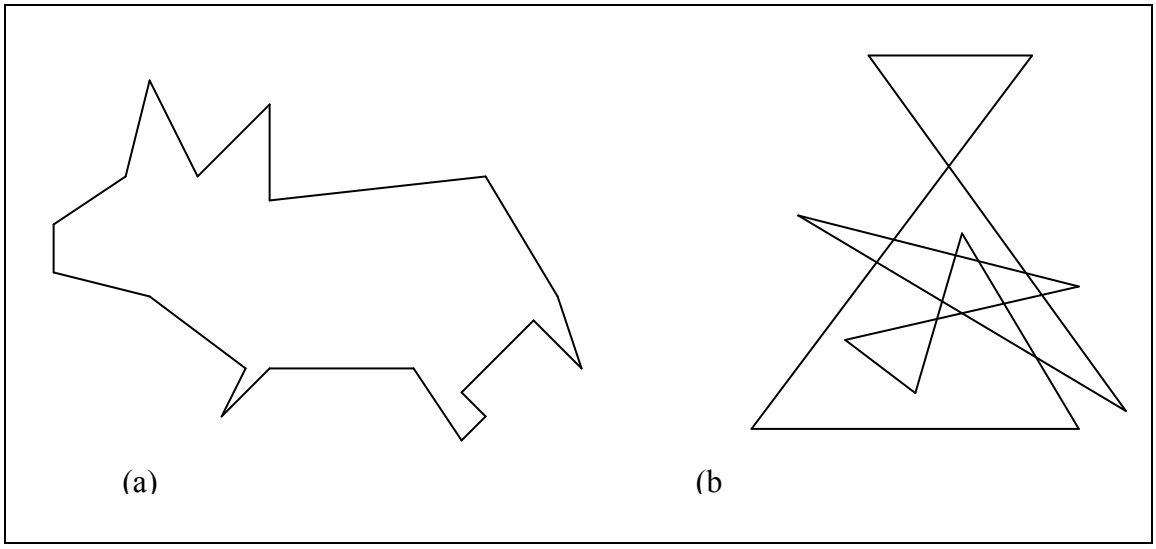

Figure 3 Examples of simple and complex polygon

1.4 Current domain

Filling of complex polygons has been chosen as the application domain, since it has a graphically intuitive output. The focus is particularly on raster devices such as printers or raster displays, in which pixels within polygonal areas are to be colored. It should be noted, however, that the preprocessing of complex polygons into simple polygons developed here, could be applied to any of the other general problems mentioned above.

1.5 Insideness of a point

Filling a polygon involves finding the points that lie inside the polygon. While the question of which points lie inside a polygon is intuitively obvious in the case of simple polygons, the question is less obvious in the case of complex polygons. There are two rules that are defined to find whether a point lies inside a polygon: the even-odd parity rule[13] and the non-zero winding number (NZW) rule[13]. The even-odd parity (EO) rule is simpler to implement, but does not intuitively lend itself to self-overlapping areas. The NZW rule provides a result, which is consistent with the notion that self-overlapping areas should be filled.

1.5.1 Non-zero winding number rule

The non-zero winding number rule determines if a point is inside the polygon by walking along the entire polygonal path, and counting the number of times the point is encircled in a counterclockwise direction (with clockwise encirclements counted as negative). If the resulting *winding number* is non-zero, the point is “inside” the polygon by this rule. Note that the polygon may be regarded as a directed graph for this purpose, although the orientation is immaterial for the insideness test since all that is required is that the winding number be non-zero.

An alternative way of calculating the winding number is by drawing a ray from the point in question to infinity in any direction and then examining all the polygon edges that intersect the ray. All polygon edges that cross the ray in a counterclockwise direction

(relative to the point) contribute +1 to the winding number, and all edges that cross in a clockwise direction contribute −1.

Figure 4(a) shows a complex polygon, star-shaped with crossing edges that was filled using this rule. The pentagonal area in the middle lies inside the polygon according to this rule, being circumnavigated once, and hence it is filled. The two doughnut shaped rings, one with an inner circle with counter-clockwise orientation and the other with a clockwise orientation are filled differently, using this rule. A point P is considered inside the "polygon" in Figure 4(d). A ray is drawn from it, extended in any direction, intersects the polygon at two places, i_1 and i_2. The winding number associated with the point P is initialized to zero. When the ray crosses the point i_1, the winding number is incremented because the direction of the edge it intersected is counterclockwise. Similarly, at i_2 the counter is incremented. The final value of the winding number is non-zero and hence the point P is filled. All the other points, enclosed within the inner-circle are filled likewise, as shown in Figure 4(d).

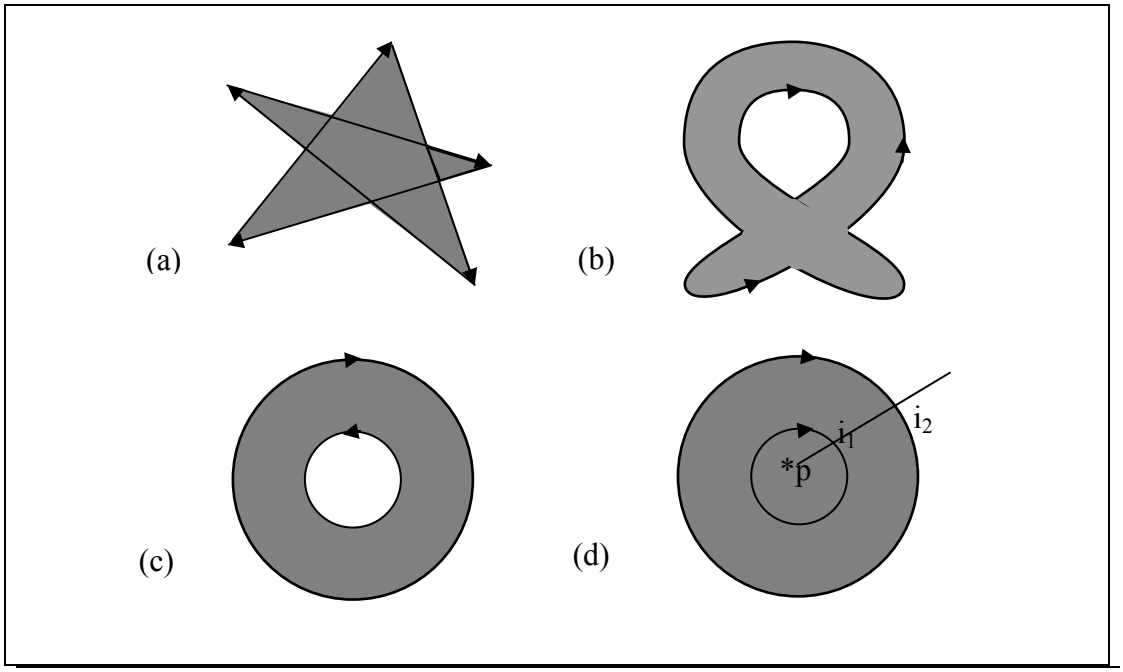

Figure 4 Non-zero winding number rule

1.5.2 Even-odd parity rule

The even-odd parity (EO) rule determines whether a point lies inside or outside a polygon. This rule does not regard the polygon as a directed graph as in the case of the NZW rule, but rather as an undirected graph. A ray is drawn from the point in question in any direction and the number of times the ray is intersected by the polygon edges is counted. If that number is even the point is outside, otherwise it is inside.

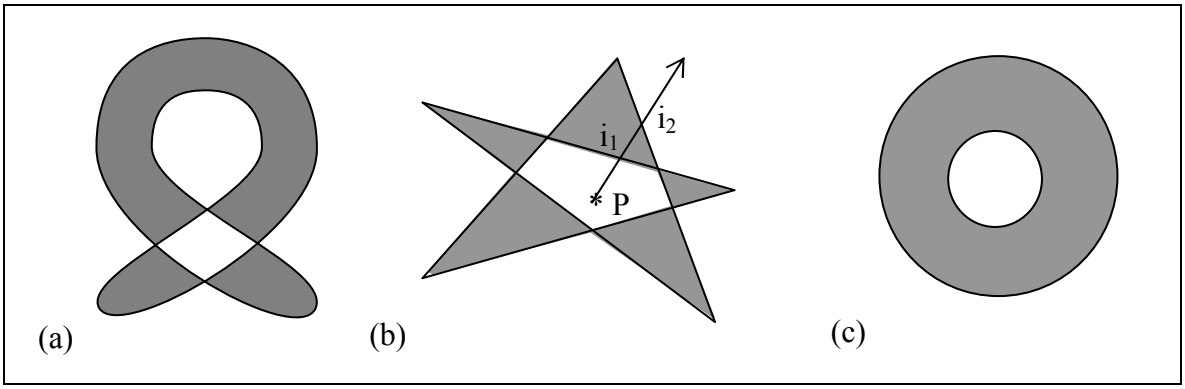

Figure 5 Even-odd parity rule

Figure 5 shows the filling of the polygons using the EO rule. A point P is considered inside pentagonal region of the star-shaped polygon in Figure 5(b). A ray drawn from P intersects the polygon at two points, i_1 and i_2 which is even. Therefore, the region containing P is not filled according to this EO rule. Similarly, the region inside the inner circle in Figure 5(c) is considered to be outside using this method. The drawback in this method is that, it does not render self-intersecting areas in an intuitive way. For instance, the sausage-shaped area in Figure 5(a) created using Microsoft Word does not have its self-overlapping area filled because Microsoft Word uses the EO rule to fill its polygon. Here, the NZW rule gives a more intuitive result as seen in Figure 4(b).

Clearly, these two rules differ in their results, and their choice is application specific. Most drawing applications use the EO rule because it is simple. On the other hand, Postscript, has an operation to select which of the two rules to use, but uses the NZW rule as its default.

1.6 Methods for filling polygons

1.6.1 Image versus object precision

In order to characterize current methods for polygon filling, it is useful to introduce the notions of image-precision versus object-precision algorithms. An underlying construct in *image-precision* algorithms is that of pixels or scan lines (pixels organized in horizontal rows). That is, the algorithm traversal is done on a pixel by pixel, or scan line by scan line, basis. The work required of the algorithm is therefore dependent (typically in an $O(n^2)$ fashion, where *n* is the number of pixels per unit distance) on the resolution of the display raster. On the other hand, *object-precision* algorithms perform their computations in a mathematical manner, independent of any display characteristics. Image-precision algorithms are often simpler to design and implement. They use integer arithmetic and they trade-off accuracy and speed. In high-resolution environments they provide more accuracy, but lower speeds. As device resolutions increase, the speed of these algorithms, which may have been adequate for lower resolutions, becomes more of a bottleneck. Object-precision algorithms are resolution-independent, but require greater use of floating point numbers and hence tend to be slower.

1.6.2 Current method

The current technique used in polygon filling is taken from the printer industry (e.g., Minolta-QMS), which uses page description languages (PDL), such as PostScript©, HP PCL, or HP-GL. A public domain source of sophisticated code for PostScript is ghostscript©. These PDLs use, as their method for rendering filled polygons, an image-precision algorithm, the scan line algorithm (SLA) adapted for EO or NZW filling, and further adapted to generate a list of trapezoids (or triangles, etc.). The SLA algorithm is described in Section 1.7.

1.6.3 Proposed method

The proposed method for generating a covering set of trapezoids uses two object-precision algorithms in tandem; first the current algorithm is used as a preprocessing step

to convert complex polygons into a set of simple polygons (the left-most arrow in Figure 1), and this is followed by an already developed trapezoidation algorithm which takes the simple polygons generated by the current algorithm and generates the trapezoids (the second arrow in Figure 1).

1.7 Filling polygons using scan conversions

The most common method of filling polygons in raster devices, in which pixels are organized in horizontal rows called *scan lines*, is by the well-known scan line algorithm. A scan line scans every pixel in that line, checks for intersections with the edges and fills every pixel, appropriately, by finding whether it is inside or outside the polygon, using one of the above-mentioned rules. This process of discovering the pixels that must be set is called *scan conversion*.

An outline of the SLA follows. Initially a global edge table (GET) is built, which contains all the edges of the polygon sorted by their smaller y- coordinate using bucket sort, every bucket representing a scan line. Within each bucket, edges are arranged in an increasing order of x-coordinate, of the lower endpoint of an edge. Also, every edge contains details about the maximum y-value, x-coordinate of the bottom end point and the x-increment used to step from one scan line to another.

Another main data structure is the active edge table (AET), that contains the details about the current scan line and the edges associated with this scan line, to facilitate further processing using the stored information.

Thus, for every scan line:

1. Find all the intersection points of the scan line with the edges of the polygon.
2. Sort the intersections by increasing x-coordinate.
3. Fill in all the pixels between pairs of intersections (also called *scan run*) that lie interior to the polygon, using the EO or NZW rule.

Figure 6 shows scan line filling for a simple polygon. The horizontal scan line at 6, intersects the polygon at points 4, 8, 12 and 14 and that is the sorted list of the intersection points.

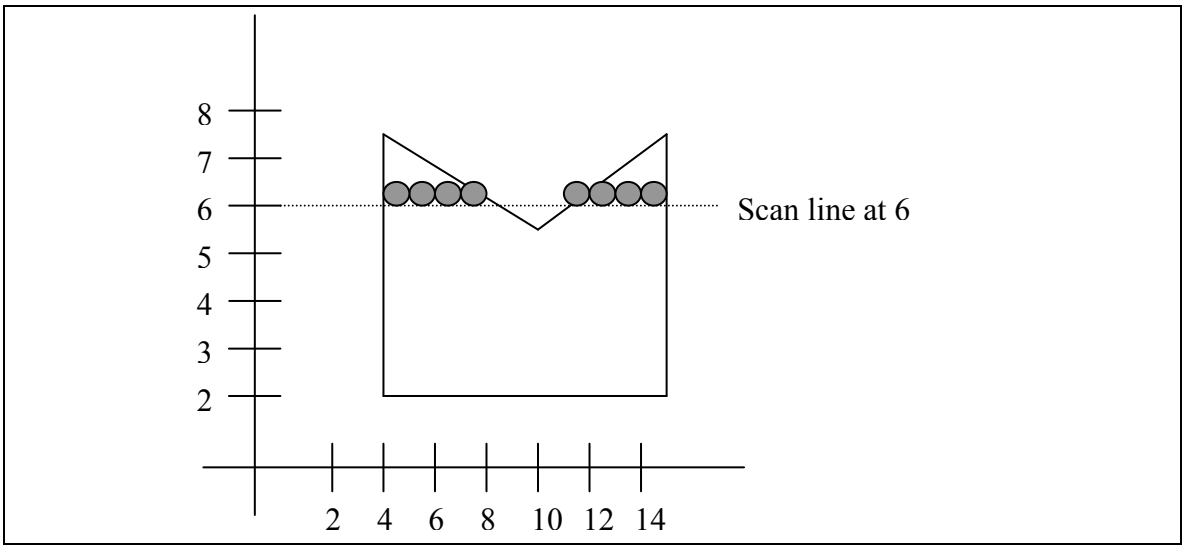

Figure 6 Scan Line Algorithm

1.7.1 Edge coherence

Calculating the intersections needs to be efficient, so checking every edge of the polygon for intersection should be avoided. It is observed that an edge that is intersected by the scan line i, is also intersected by the scan line $i+1$. This is called *edge coherence* or *scan coherence.*

The next x intersection for the new scan line is calculated by using,

$$x_{i+1} = x_i + 1/m$$

where m is the slope of an edge.

1.7.2 Other issues

There are many other issues involved in filling a polygon and making the result appealing to the viewer. For example, an edge that is shared by two polygons, if colored by both the polygons, leaves a bizarre color as a result. Hence, it is conventional that, the leftmost edge and the horizontal edge of a polygon always belong to the current polygon being processed.

Chapter 2
RELATED WORK

2.1 Introduction

The following is a review of papers that relate to the current research. It includes references to research on (simple and complex) polygon partitioning, and line intersection problems for complex polygons, or collections of line segments.

2.2 Partitioning algorithms

The problem of partitioning a polygon is dated very early in the literature around 1911, when the first paper on triangulation of a polyhedron and related problems were addressed by Lennes [24]. Several algorithms for polygon partitioning in the forms of triangulation, trapezoidation have been reported in the literature since 1978. Monotone partitioning by Garey, Johnson, Preparata, and Tarjan (1978) [16] is an algorithm that partitions a simple polygon into monotone polygons, and extends the concept to carry out triangulation. Since then, there have been numerous algorithms devised and implemented that sought solutions for triangulation, with less run-time complexity [14][28][11]. Finally, in 1991, Chazelle [9] came up with an algorithm that triangulates a simple polygon in linear time. But this algorithm was claimed to be 'unimplementable', because of its enormous details. All the above-mentioned algorithms were written to partition a simple polygon into a set of triangles, using various techniques.

In 1995, Bruce Romney [27] used the concept introduced by Chazelle and Incerpi to simplify the required data structures, to vertically decompose the faces of an arrangement into trapezoids, which was extended for triangulation. Also, there is Seidel's

algorithm for triangulation, which is simpler, but not a faster triangulation algorithm that can be widely used in the industry. Hain's algorithm [17] for trapezoidation is a more practical solution for partitioning, which is almost linear in complexity except in certain conditions.

More recently, Nancy M. Amoto, Michael Goodrich and Edgar Ramos [2] published an algorithm for triangulation of a simple polygon in linear time, through trapezoidation of a polygon. They have also stated that it is simpler than Chazelle's optimal deterministic algorithm.

All these algorithms have varied application domains. As mentioned earlier, the domain of interest for this thesis is filling the polygons. This application can be used in the printer industry and various graphics applications, where even some complex characters are considered as polygons to be filled. In this context, problems involved like point location, etc., are discussed by Lee and Preparata in their papers on the applications involved in point location on a planar subdivision [23][22] in 1977, before the break-throughs for triangulation problems that were based on these ideas.

2.3 Line intersections

The other main issue involving polygon partitioning, i.e., finding the intersections between collections of line segments, was addressed by Bentley and Ottman in 1979 [6]. It was further discussed by Chazelle and Guibas [10] in 1989, which is one of the most commonly referenced papers in algorithms that deal with polygon triangulation or trapezoidation. A recent paper by Chazelle in 1992 [7], discusses the solutions putforth to date, for the line intersection problem and the issues involved in that problem that complicate the solutions. It is one of the main referenced papers of this thesis, as it gives an overall idea of all the methods involved in finding the solution, for practically all the test cases, that might occur in the application domain.

2.4 Other sources

The above-mentioned papers do not represent any particular area for application. The Postscript manual documentation [1] describes the process of filling the polygons that is followed as standard in the Postscript specification. Also the standard methods, described

in the books on primitive methods in computer graphics on rasterization, were read and noted [13].

After a careful search in the literature, it was found that no algorithm has been written to partition a non-simple polygon into a set of simple polygons. Most of the algorithms, just avoided the complex polygons, and processed only the simple polygons. This research proposes to fill-in the gap, by providing an implementation for the two main algorithms outlined in Chapter 4.

The algorithms used for filling polygons that have been commonly used by printer companies (like Minolta-QMS), and Page Description Languages, like PostScript [1] were considered for a comparison study in this thesis, but the code for those was not available. However, it is known that the graphics library uses the SLA algorithm for polygon partitioning [18].

In particular, the code for a sophisticated public domain version of a PostScript interpreter and renderer, ghostscript [15], was studied and found to also implement polygon partitioning by the SLA algorithm.

Chapter 3
METHODOLOGY

3.1 Aspects of work

This work concerns a new algorithm to partition a non-simple (complex) polygon into a set of simple polygons, using either the NZW rule or the EO rule, thereby providing a preprocessor for many applications involving polygon partitioning. The work includes the following components

1. Design the algorithm based on an outline provided by Dr. Hain.
2. Implement the algorithm.
3. Create a tool to interactively generate and manipulate polygons (simple and complex). This is useful for the dynamic testing of the algorithm.
4. Test and demonstrate the correctness of the algorithm using this tool.
5. Provide a test bed to demonstrate the relative performance of the algorithm compared to current methods.

The algorithm clearly should be correct, and should be efficient within useful, i.e., practical domains. The algorithm will be empirically compared to the SLA algorithm, the algorithm currently used in industry.

Implementing the algorithm involves determining the data structures that would best solve the problem, minimizing performance overhead. Some ideas found in the literature were adapted for this thesis.

3.2 Correctness

Correctness testing required the generation of input polygons, and availability of their correctly filled outputs. Two generation methods were used in this work, and embodied in a single interactive tool built for this purpose:

1. Random polygon generation.
2. Interactive polygon generation. This method of testing is found to be useful in testing a variety of computational geometry algorithms because of their visual outputs. Because of the dynamic nature of this type of testing, a very large domain of equivalence classes can be covered. This technique was described in [18].

Code was adapted from [4] [5] to generate random polygons for a given number of vertices. This random polygon generator (RPG) is an application developed as a part of Thomas Auer's Master's thesis. It is a set of C programs that use an X-Windows API. This application has few heuristics, and can be utilized to generate different kinds of polygons, like star-shaped, monotone, etc. These polygons are random in their geometric properties, with no restriction on their *sinuosity* (winding property,) or their horizontal or vertical co-linearity.

Interactively produced polygons can be generated and edited using the interactive testing tool. The tool allows the user to move, add, or delete vertices, starting with either a simple triangle, or a randomly generated polygon. The user can generate multiple polygons, within the drawing area.

These polygons can be saved as either a text file, or as a PostScript program. The PostScript output is viewable with a PostScript viewer (ghostscript)—which is presumed to provide the correct output—to enable visual comparison with the results from the NZW algorithm. The correctness was taken as a consistency between the results from the PostScript viewer, the SLA and the NZW algorithms, together with, in relatively simple cases, manual solutions.

The tool showing a sample complex polygon with a randomly generated 20-vertices polygon, is shown in Figure 7.

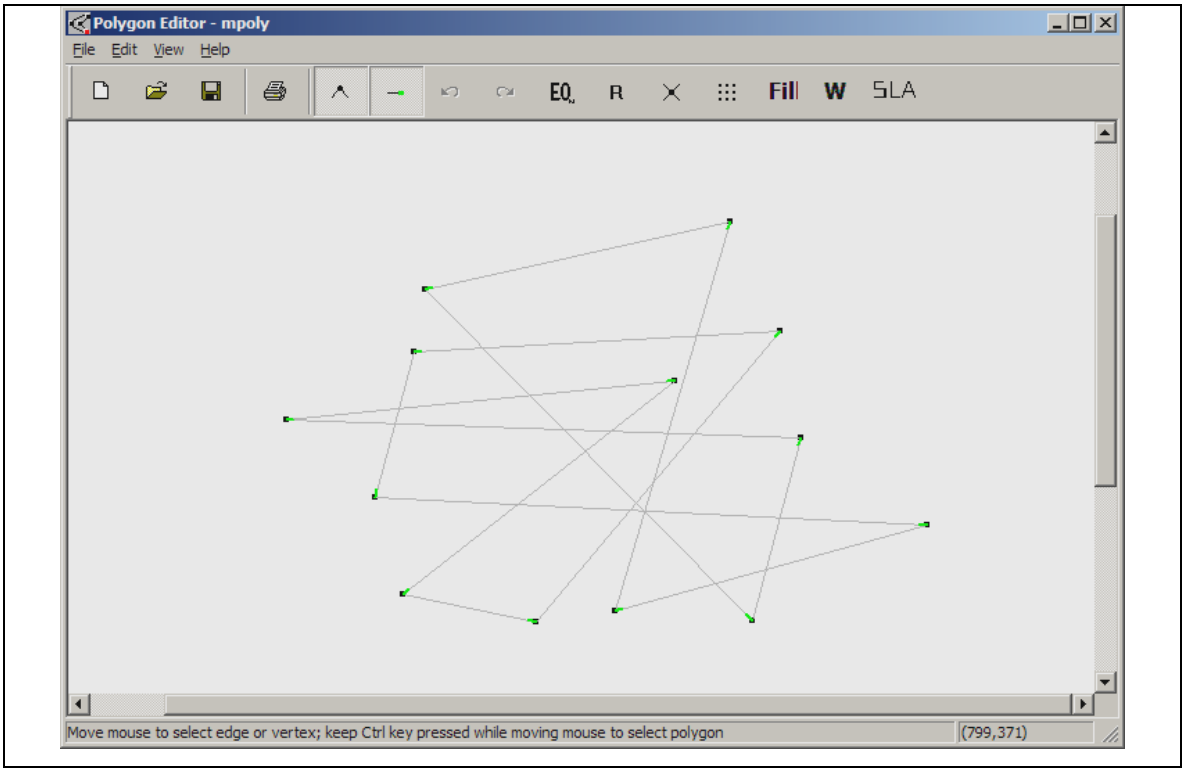

Figure 7 Interactive testing tool showing random input polygon

3.3 Performance

For performance testing both the algorithms (NZW and SLA) were embedded into a Windows console application, and used a high resolution (sub-millisecond) timer provided within the MS Windows© operating system. The timing was performed on a release compile (i.e., optimized, and without debug code.) All possible background processes were removed to make sure that as many of the resources of the computer were devoted to the operation of the algorithms. The timing was restricted to only those operations performed by the algorithms necessary to determine the outline of the fill operation. No scan line spans were actually filled in SLA. No actual rendering was performed. Care was taken to avoid including any IO overhead within the timed brackets.

The timing data suite was a set of randomly generated complex polygons classified by their number of vertices, number of intersections, and height in pixels. The generation of this data suite is further discussed in Section 5.2.2.1.

3.4 Hypotheses

1. An object-precision algorithm can be devised to partition a complex polygon into a set of simple polygons using the NZW rule (NZW Partitioning algorithm.)
2. The devised algorithm will generate a filled polygon no slower than, and, in more complicated cases, faster than the SLA.

Hypothesis 1 can be verified by testing the correctness of the algorithm, as outlined in Section 3.2. Hypothesis 2 can be verified by comparative performance measurements, as outlined in Section 3.3.

Chapter 4
THEORY

In this chapter we describe a new algorithm to partition a complex polygon into a set of simple polygons which outline the filled area. This partitioning is with respect to a particular fill rule—either the non-zero winding number rule described in Section1.5.1, or the even-odd parity rule described in Section1.5.2.

The initial investigations led to the conclusion that the EO partitioning algorithm could be easily designed and implemented using the primitives developed to design and implement the NZW partitioning algorithm. Thus, the focus was to outline an algorithm to partition a complex polygon into a set of simple polygons (possibly with holes) using the NZW rule.

At present, the algorithm presented does not address any singularities such as degenerate polygons (e.g., zero size, coincident vertices, etc.), with the assumption that the algorithm can be extended to include these cases without loss of generality, if the fundamental concept is correct.

4.1 Outline of NZW partitioning algorithm

We will first outline the NZW partitioning algorithm. It should be noted that many implementation details will be omitted at this point, and only a high-level understanding of the algorithm will be provided in this section. We will use the notion that the interior of a simple polygon can be considered as a (an infinite) set of points in a two-dimensional space.

4.1.1 Terminology

This section formally defines the terminology used in the description of the outline of the NZW algorithm.

The input to the algorithm is an ordered list of vertices—that we denote as *polygon vertices*—of a (complex) polygon. The ordering of the polygon vertices induces directed line segments called *polygon edges*.

An *intersection* is defined as the point at which exactly two polygon edges coincide. When more than two polygon edges intersect at the same point, multiple coincident intersections result since each intersection results from a pair of edges.

A directed polygon edge may be divided at intersection points into two or more collinear, directed (in the same sense as the parent edge) *edge segments*. Each polygon edge segment may be thought of as belonging or being the child of a polygon edge. Both the in- degree and out-degree of an intersection is two. The described algorithm considers the intersection as having two coincident *pseudo-vertices* which are *companion* to each other. For the sake of orthogonality, we will consider a polygon vertex as an intersection, and define two associated pseudo-vertices, one of them being null. When we wish to be specific, we will distinguish the pseudo-vertices as *polygon pseudo-vertices*, and *intersection pseudo-vertices*.

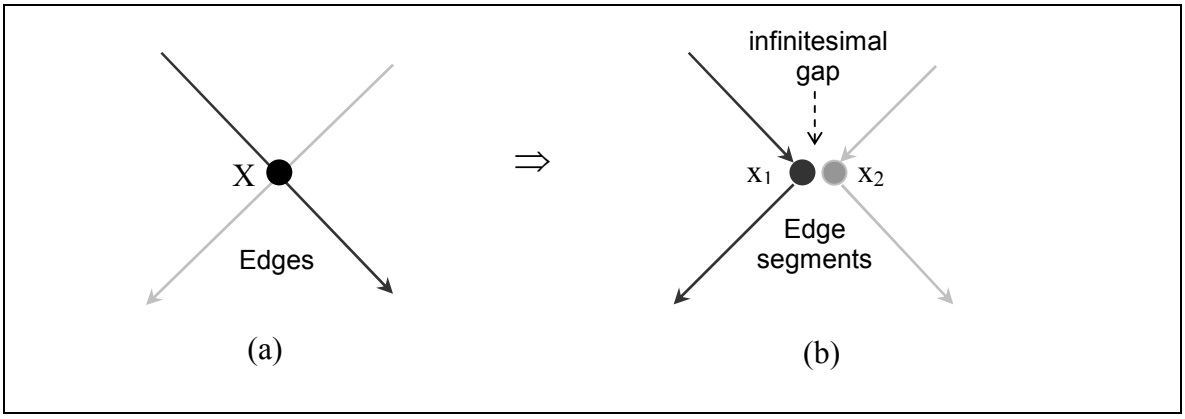

Figure 8 Pseudo-vertices x_1, and x_2 at intersection point X

4.1.2 Algorithm overview

The algorithm joins, at each pseudo-vertex, an incoming edge segment belonging to one polygon edge with the outgoing edge segment belonging to the other polygon edge, as shown in Figure 8. By connecting the pseudo-vertices in this way, a set of polygons are formed which we will term as *component polygons*. Thus, the edges of component polygons are the edge segments of the complex polygon, and the vertices of the component polygon are the pseudo-vertices. Every pseudo-vertex will be associated with a component polygon, and every component polygon will contain a disjoint set of pseudo-vertices. The following theorems characterize component polygons.

Theorem 1: Component polygons are simple.

Proof: No edge segments intersect, otherwise they would be partitioned at that intersection. Each pseudo-vertex is traversed exactly once. If we conceptually consider an infinitesimal gap between the companion pseudo-vertices, as in Figure 8, it can be seen that the sequence of directed edges does not cross any gap between pseudo vertices, and therefore cannot pass the same intersection more than once, and therefore cannot self-intersect. Since there is a finite number of pseudo-vertices, the sequence must close because of the pigeon hole principle, and therefore forms a simple polygon. ■

Theorem 2: Component polygons are disjoint, or fully enclose, or are fully enclosed by another component polygon. That is, in a set of component polygons $P_i,\ 1 \le i \le n$, for each pair P_i and P_j with $i \ne j$ one of the following is true:

1. $P_i \subset P_j$
2. $P_i \supset P_j$
3. $P_i \cap P_j = \varnothing$

Proof:

It is seen that no component polygon can intersect with another component polygon at either a component polygon edge (edge segment) or, again considering a conceptual gap between pseudo-vertices, at a component polygon vertex (pseudo-vertex). Thus the component polygons can only be nested or disjoint. ■

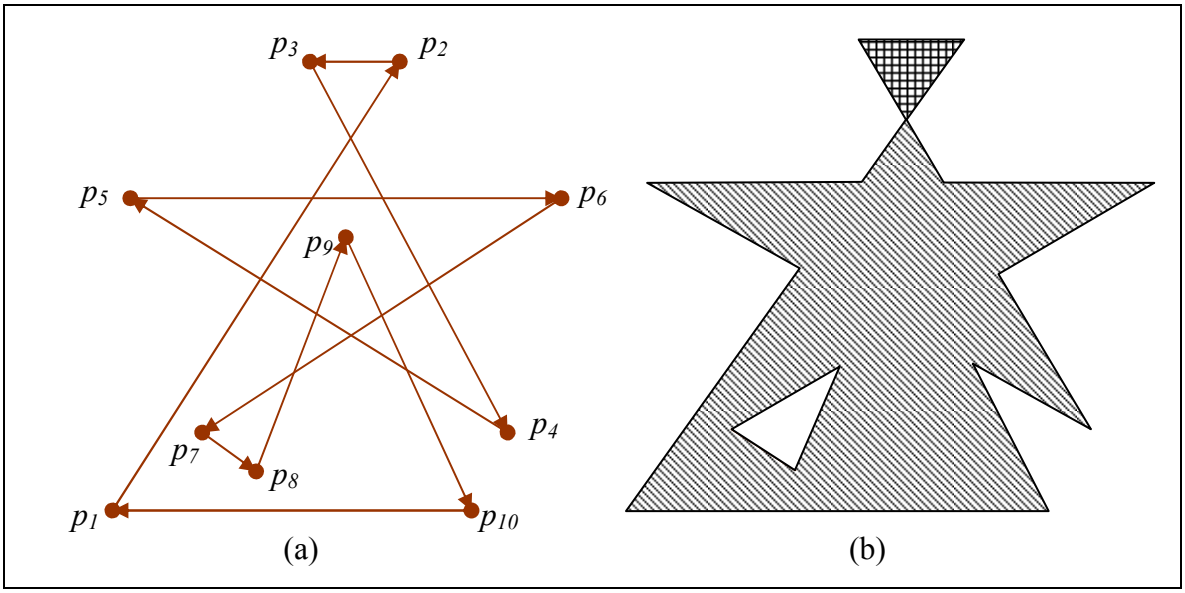

Figure 9 (a) complex polygon with labeled polygon vertices
(b) the result of the filling by NZW algorithm

Each component polygon, being simple, is oriented either counter-clockwise or clockwise, and so contributes a winding number of +1, or –1 to any enclosed point. We therefore associate a winding number of ±1 with each component polygon (or alternatively with each of its pseudo-vertices.) To determine whether a given point is "inside" the original complex polygon, one may simply add the winding numbers of all the component polygons that enclose the point, to see if it is non-zero. We will call the sum of winding numbers of the enclosing polygons the *net winding number*. Since the component polygons are nested, we can associate a net winding number with each polygon (or alternatively with each of its pseudo-vertices.) We can also visualize the component polygons as stacking together with the smaller polygons "on top of" the largest one. The net winding number of any point is the sum of the winding numbers of all the polygons stacked at that point.

We will define a vertex of a simple polygon as being either *convex* (right-turning for a clockwise oriented polygon, or left-turning for a counter-clockwise polygon), or *concave* (left-turning for a clockwise polygon, or right-turning for a counter-clockwise polygon).

The algorithm proceeds by finding the left-most polygon vertex, and corresponding polygon pseudo-vertex. This pseudo-vertex will necessarily be convex, and will define the orientation of the associated component polygon, and consequently the net winding number of this pseudo-vertex can be initialized to either –1 or +1.

As the component polygon is traversed starting at the identified pseudo-vertex, every companion to a convex pseudo-vertex is added to a queue (alternatively, a stack can also be used). This companion will be the starting point for the traversal of a (potentially) new component polygon disjoint to the current one. Actually, this disjoint polygon may "touch" the current polygon at several places. Since we know that the disjoint polygon will have the opposite orientation as the parent, we can update the net winding number of the companion appropriately.

Similarly, as the traversal proceeds, all companions to a concave pseudo-vertex will be added to the queue, and will be the starting point for the traversal of a (potentially) new component polygon that will be an outermost component nested within the current one. Again, this inner component polygon may touch the current component polygon at several places. Since we know that the disjoint polygon will have the same orientation as the parent, we can update the net winding number of the companion appropriately.

Once the current component is completely traversed, a pseudo-vertex is dequeued, and, if it has not already been traversed, starts a new traversal in a like manner to the previous one. This process continues until the queue is empty.

At this point we have a collection of all the component polygons, each with an accumulated net winding number. The outline of the area that is to be filled is the subset of component polygons having net winding number of –1 or +1 (forming the outer boundaries of the filled areas), and component polygons having a net winding number of 0 (forming the inner boundaries, or "holes".)

4.1.3 Example

Let us consider the input complex polygon in Figure 9. This example figure has most of the special cases that need to be handled by the algorithm. Also, it should be noted that

the input polygon edges are directed, which is a factor that decides the insideness of a point in a polygon in the NZW filling rule as seen in Section1.5.1.

The component polygons of the example polygon in Figure 9 are shown in Figure 10. The first component polygon generated in the algorithm is the outer-most polygon P_1 (Figure 10(a)), which has a winding number of *-1* , filled with downward slanting lines, as shown. The second and third polygons P_2 (Figure 10(b)) and P_3 (Figure 10(c)) have a winding number of -1 in accordance with their orientation, and have net winding numbers of -2 (since P_2 overlays P_1) and -3 (since P_3 overlays P_1 and P_2) respectively. The fourth polygon P_4 in Figure 10(d) is not filled because its net winding number is zero, since it has an opposite orientation to its encompassing polygon, P_1 and hence it forms a hole inside P_1. Thus, holes are handled in a natural way by the NZW algorithm. The fifth polygon P_5 in Figure 10(e) is disjoint from P_1 and has a winding number of +1. Figure 10(f) shows the overlapping component polygons, and each area with different shading has a different net winding number.

When the input complex polygon is processed by the NZW algorithm, three simple polygons define the filled area as shown in the Figure 9(b), and one of these is a hole.

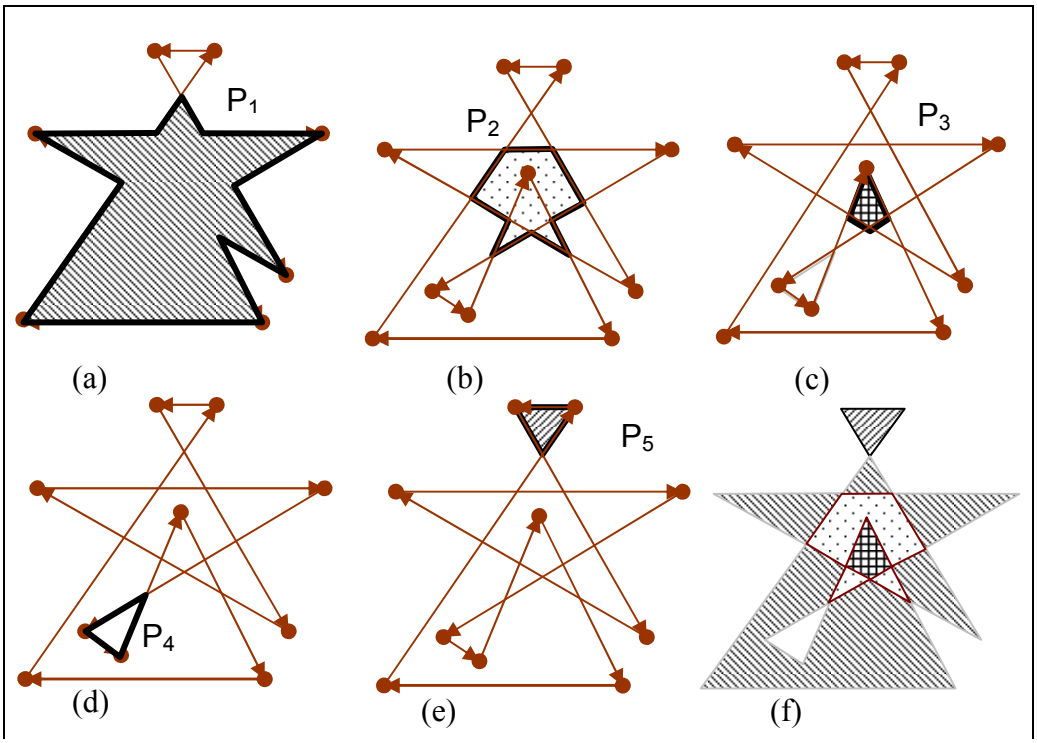

Figure 10 (a) –(e) Component polygons of the example input polygon. (f) Overlapping component polygons

4.2 Implementation Details

4.2.1 Intersection points

Each intersection between a pair of polygon edges is calculated analytically in this algorithm using parametric equations. Naturally, each polygon edge may intersect with up to *n*-1 other edges, and we want to associate this list of intersections with either that edge or, say, its starting vertex.

The evaluation of all intersections is O (n^2) in the worst case, since every pair of polygon edges potentially intersects. The related problem of finding the intersection points of a collection of line segments was addressed by Chazelle [7], and is itself a surprisingly difficult task to do optimally, and we do not propose to implement a theoretically optimal algorithm. Nonetheless, the discovery of intersections probably represents the major computational work of the algorithm proposed here. However, with reasonable heuristics, and an assumption that input polygons are not extremely convoluted in practice, the computation can be considerably reduced. We tried to

minimize the need for intersection calculations by taking advantage of any occurrence of monotonic chains (in either the *x* or the *y* direction), but the overhead was found to be greater than the gains in most typical situations. Trivial rejection of line segment intersections by non-overlapping bounding boxes was used to avoid intersection calculations, and was found to be advantageous.

4.2.2 Intersection Linkage

When the input polygon is traversed to generate the component polygons, each pseudo-vertex that is traversed, has a pointer to the next pseudo-vertex it is linked with. This information that is contained in every pseudo-vertex is called its *intersection linkage*. The following paragraphs briefly introduce the classes that were constructed to establish the necessary linkage. It should be noted that the objects created with these classes, are temporary, i.e. they are needed until the intersection linkage is complete.

A *polygon edge array* of polygon edges with each edge pointing to a list of intersections, sorted by their parametric value relative to the edge (i.e., t = 0 at the start vertex, t = 1 at the end vertex of the edge). This task will give rise to the companion pseudo-vertices, one for each intersecting line segment. The polygon edge array that will be generated for the example in Figure 9 is shown in Figure 12.

An *intersection object* is composed of a pair of pseudo-vertices at an intersection. This includes both the polygon pseudo-vertices and the intersection pseudo-vertices. The polygon edge array created above is used to create an *intersection master list* that contains all the intersection objects. The intersection master list that will be generated for the example in Figure 9 is shown in Figure 16(a). The polygon edge array and the intersection master list are built simultaneously and their data structures are discussed in Section 4.2.3.2.1

4.2.3 Data Structures

The challenge in the implementation of the algorithm lay in the design of the appropriate data structures. In this section, the data structures of all the objects required to implement the algorithm are illustrated in Figure 11 and Figure 13.

4.2.3.1 Support classes

The basic entities that are fundamental are as follows: A polygon vertex is stored as an object of type `Vertex`, a class containing two float values, for *x* and *y* co-ordinates. The input polygon is stored as an object of type `Polygon`, a class containing a set of ordered vertices, implemented as a linked list of vertices, a *vertex list*. A polygon edge is stored as an object of type `LineSegment`, a class containing a start vertex and an end vertex. With all these supporting classes `MultiPolygon` (more than one polygon e.g., a set of component polygons, or a set of outline polygons together with hole polygons) is a class that contains a linked list of polygons (*poly list*), with each polygon in turn containing a pointer to a vertex list. This is illustrated in Figure 11. The linked list, vector and queue were implemented using the container classes from STL.

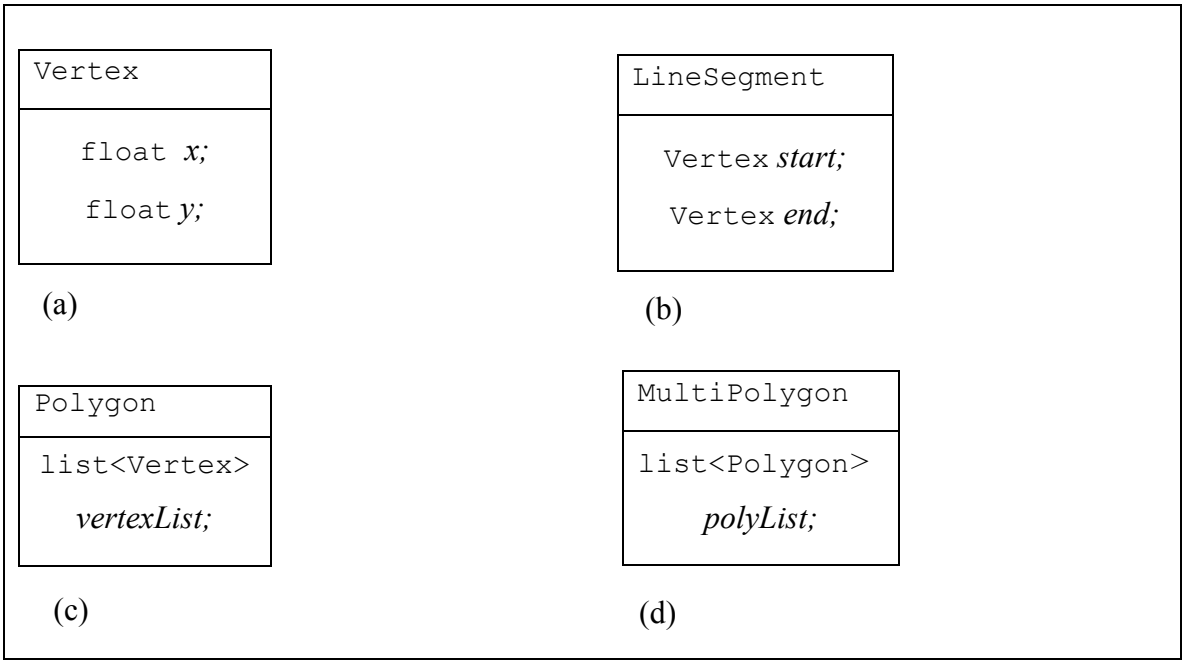

Figure 11 (a) `Vertex` object (b) `LineSegment` object, (c) `Polygon` object and (d) `MultiPolygon` object

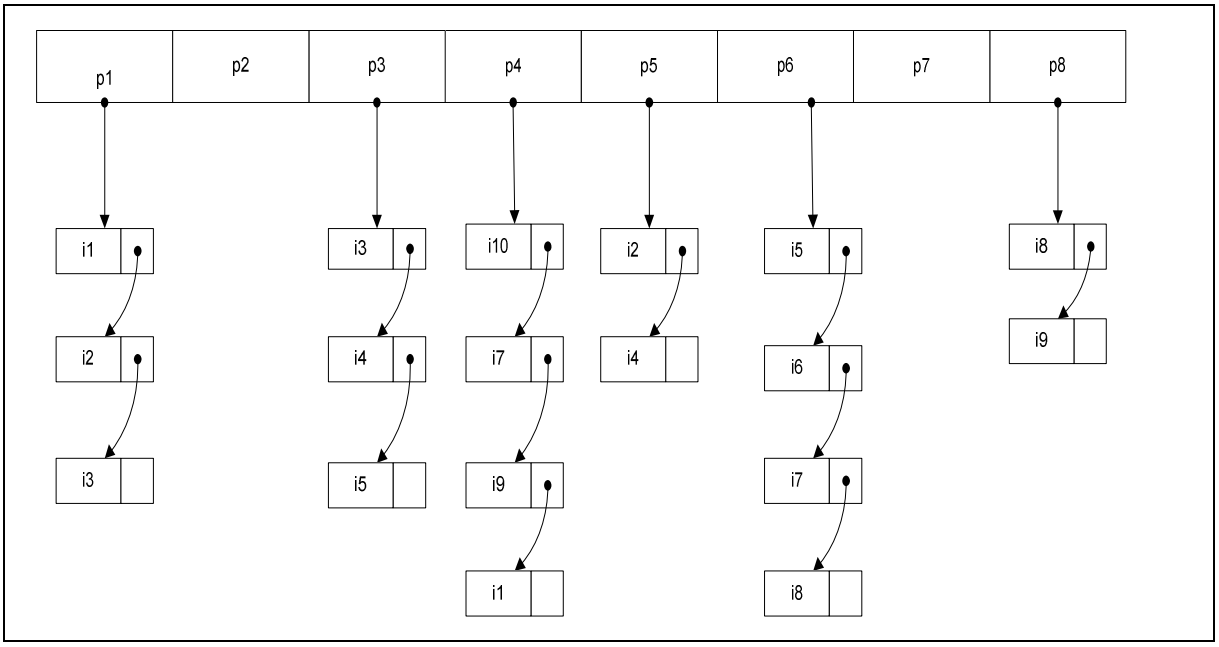

Figure 12 The Polygon Edge Array for the Figure 9(a)

4.2.3.2 Intermediate classes

In order to get the master list of intersection objects that contains the necessary information, various intermediate objects were designed. The polygon edge array was implemented using an object of a class called `nVertex` which contains a vertex (polygon/intersection pseudo-vertex), the intersecting line segment(*l*), the parametric value (*t*), with respect to the line segment and the index of the position of the pseudo-vertex, in the intersection master list, as shown in Figure 13(b). The array of polygon edges is implemented as a vector (*nVertex* objects holding polygon vertices), each having a pointer to a linked list of intersections (*nVertex* objects of intersections), as shown in Figure 12 . It is implemented with the *nVertex* objects that hold necessary information to create the master list.

The most important object is an *intersection object*, which contains information about every pseudo-vertex in the input polygon. Every intersection object contains an intersection point (*v*), the starting vertices of the two intersecting polygon edges (*origin vertex1* and *origin vertex2)*, the pointers to the next intersection objects i.e., the value of the index position in the intersection master list (*index1* and *index2)* and the winding

number of the polygon generated by the corresponding polygon vertex (*winding number*), as shown in Figure 13(a).

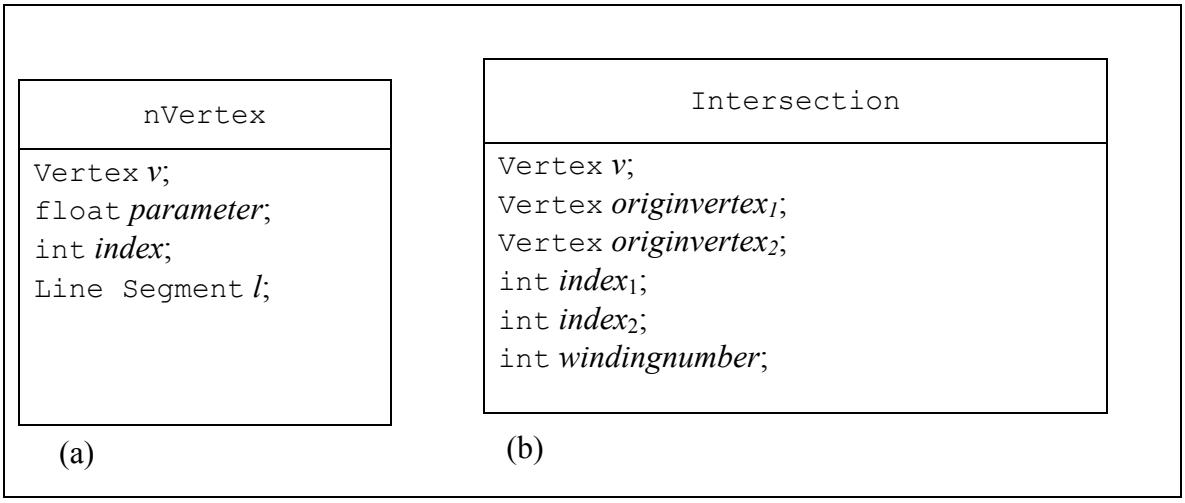

Figure 13 (a*)* `nVertex` object (b) `Intersection` object

4.2.3.2.1 Setting up the intersection linkage:

The intersection master list and the polygon edge array are built simultaneously, since the intersection master list is required to fill the index field for the objects in the polygon edge array. Once all the indices are filled in the polygon edge array, then the values in the intersection master list are filled from the information available from the polygon edge array. For every intersection object in the intersection master list:

1. search for the vertex in the polygon edge array.
2. update the $index_1$ and $index_2$ value of the intersection object, from the corresponding object in the polygon edge array.

If the vertex is a polygon vertex, then it is searched in the polygon edge array. There is only one index to be updated in the intersection object. The other index is marked -1, to imply that the path is invalid. If it is an intersection, then it is searched in the intersection list of every polygon edge, in the polygon edge array. Since an intersection object also has information about intersecting polygon edges, the $index_1$ corresponds to *origin vertex*$_1$ and $index_2$ corresponds to *origin vertex*$_2$.

For illustration , let us consider the example polygon in Figure 9. The intersection object containing the pseudo-vertex i_1 in the master list will have the data as shown in the

Figure 14(b). The pseudo-intersection vertex i_1 is linked as an intersection from both the edges p_1 and p_4, in the polygon edge array, as illustrated in the Figure 14(a). When it is linked under the edge p_1, the index position of i_1 denotes the next intersection object in the intersection master list it is linked with, with respect to the corresponding polygon edge, which is i_2. Similarly, with the polygon edge starting at p_4, it is linked to the intersection which happens to be the polygon vertex p_5.

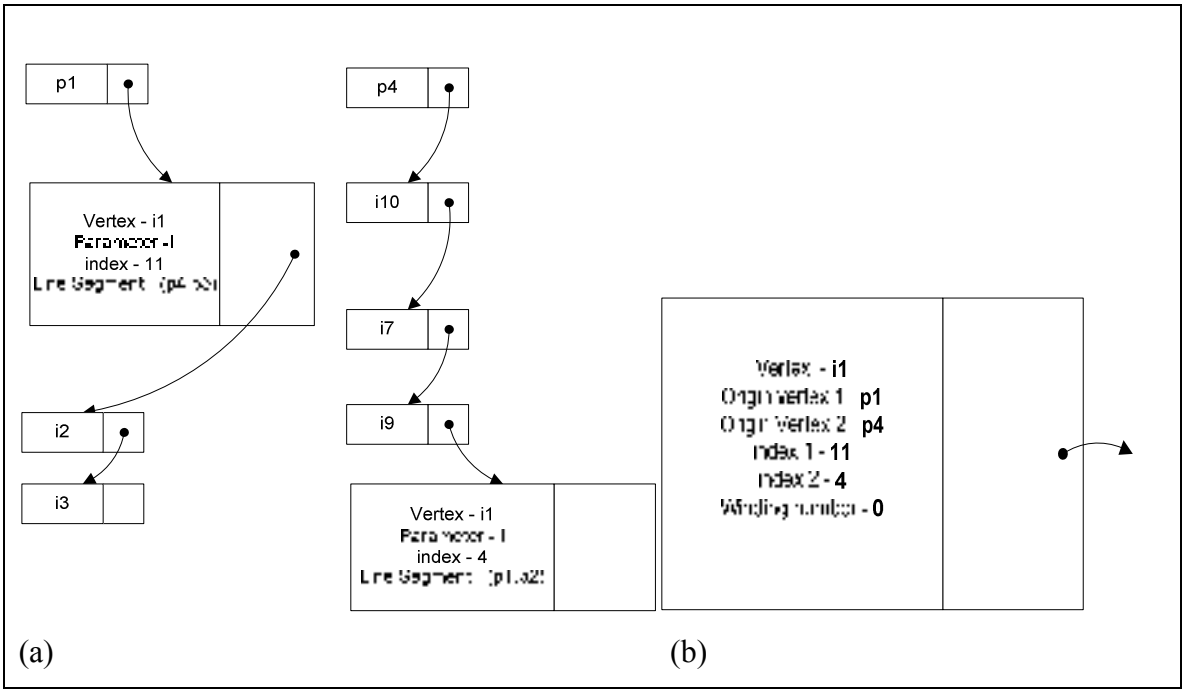

Figure 14 (a) is the illustration of i_1 in the polygon edge array (b) is the intersection object of i_1 in the intersection master list

The data structure for a multi-polygon has also been discussed, because the algorithm can successfully handle more than one polygon, as long as there is at least one intersection point between them. In this case, the intersection points are also calculated, between the polygon edges of every polygon with each other.

4.2.3.3 Illustration of the intermediate classes using an example

This section consists of a more detailed view of the algorithm by illustrating the objects that are created from the intermediate classes. For this purpose the same example in Figure 9 is considered.

After calculating all the intersection points between the polygon edges (of one or more polygons), the edge segments are generated. The intersection points and the edge segments are shown in Figure 15. All of the intersection points and the polygon vertices are then converted into pseudo-vertices.

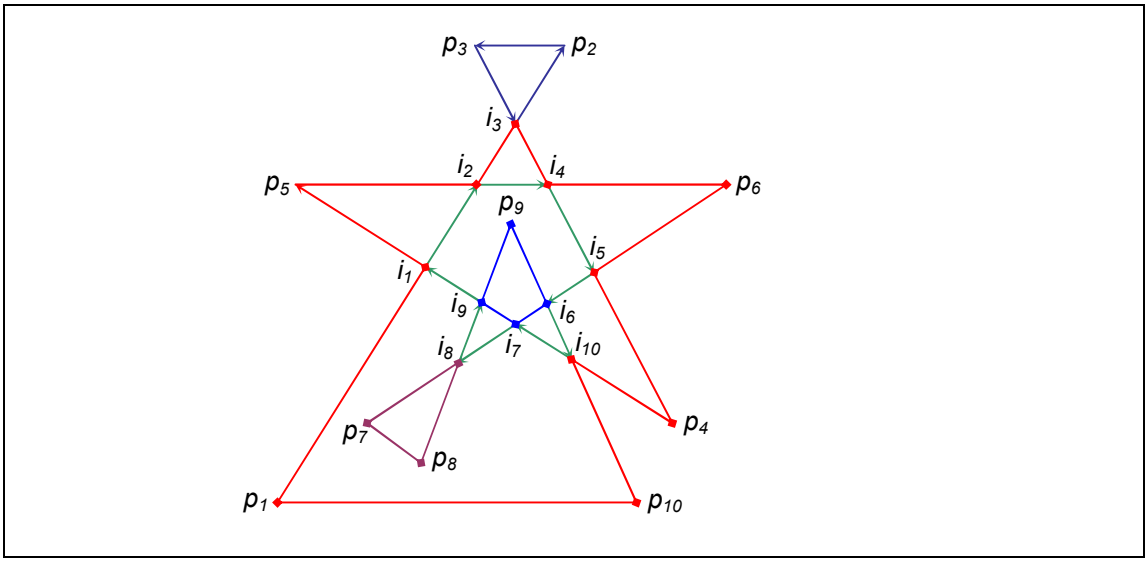

Figure 15 Complex polygon showing edge segments and pseudo-vertices

A polygon edge array, with a list of intersection vertices, sorted according to its parametric value, *t*, is created as shown Figure 12. The intersection master list that has the intersection objects with appropriate pointers, required to traverse the polygon is generated. The list generated for this particular example is shown in Figure 16(a). For this example, the starting pseudo-vertex with the lowest co-ordinate tuple is the polygon vertex p_1. Meanwhile the intersection objects that are visited are pushed into the intersection queue, which is shown in Figure 16(b) for the cycle generated. The indices of the objects that participated in this cycle are updated (exhausted) in the intersection master list.

The next starting pseudo-vertex is identified by searching the intersection queue. For this example it is at i_{10} as shown in Figure 16(b). The complete cycle that generates the next nested component polygon with the vertices. This procedure is repeated until the intersection queue is empty. The values in the intersection queue during the traversal are shown in Figure 16(b), for few component polygons.

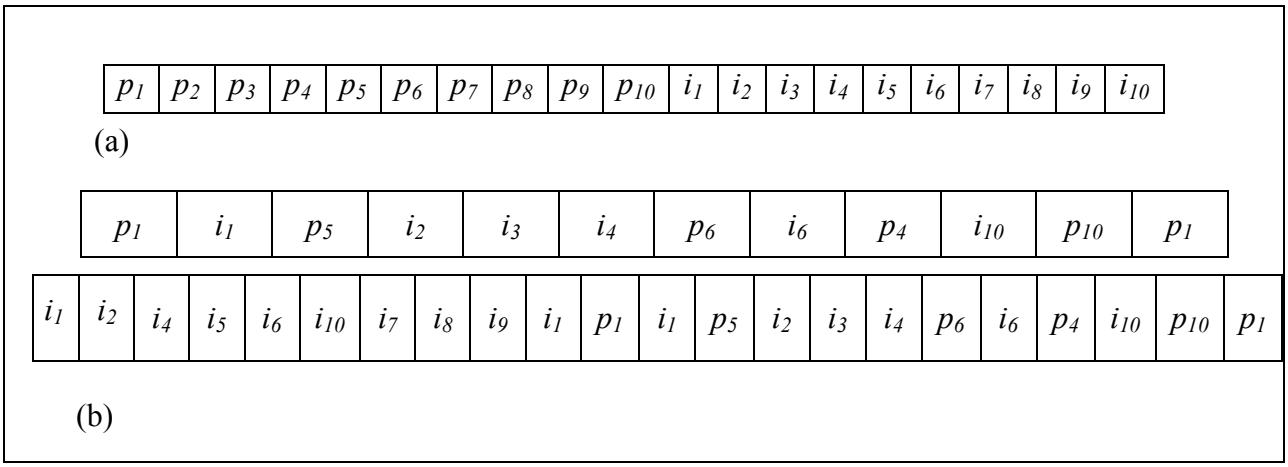

Figure 16 (a) intersection master list of pseudo-vertices (b) Shows various values in the intersection queue

Chapter 5
RESULTS

The NZW algorithm for the partitioning of a non-simple polygon into a set of simple polygons was implemented using the data structures explained and illustrated in section 4.2.3. An implementation for the NZW algorithm was built using the C++ language, using a class collection provided by Dr. Hain. An interactive testing tool to view the output of the algorithm was implemented using C++ and MFC (Microsoft Foundation Classes, and object-oriented set of classes encapsulating the MS Windows API.) Finally, a timing framework was built using a Windows console application, and a high resolution timer.

5.1 Correctness

Correctness testing required the generation of input polygons. Two methods were used in this work, and are described in detail in Section 3.2:

1. Random polygon generation.
2. Interactive polygon generation.

The algorithm showed consistency between the SLA algorithm, the NZW algorithm, and selected polygons that were compared with the ghostscript output. The only precondition for the algorithm is that the polygons are not degenerate (zero area). It should also be noted that the algorithm works for many complex polygons, provided there is at least one intersection point between them. Hence the robustness of the algorithm is demonstrated. Note the difference between the results of filling using the NZW and the EO rule, as shown in Figure 17, Figure 18 and Figure 19.

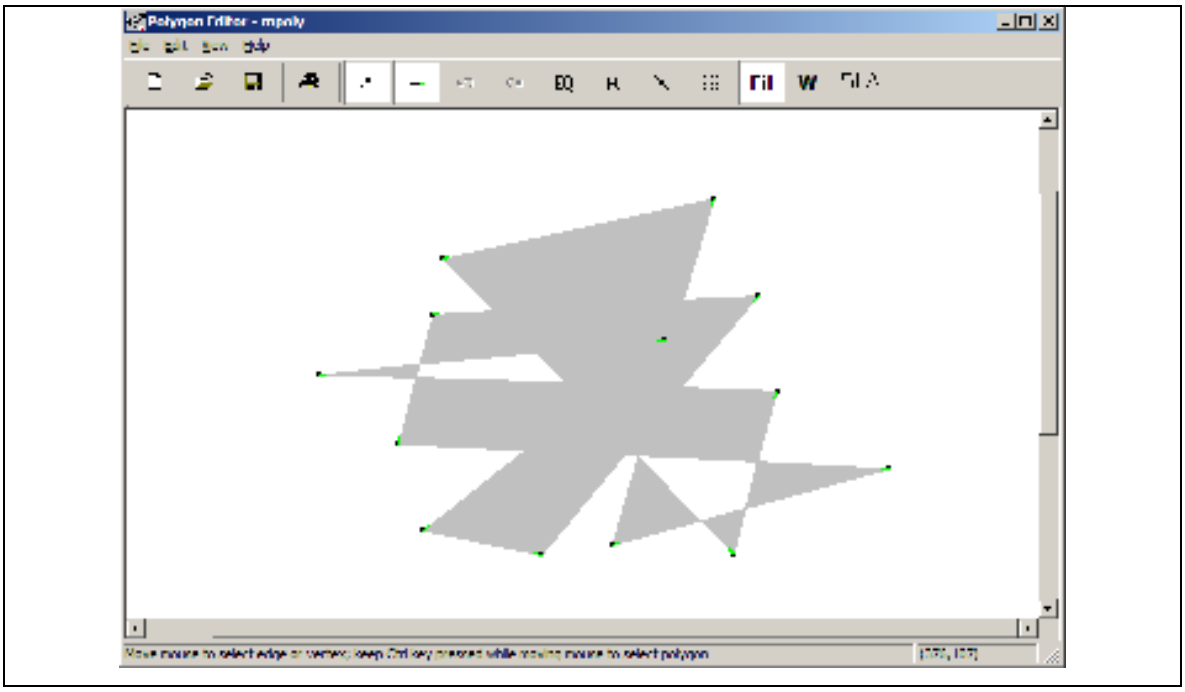

Figure 17 NZW fill using NZW filling rule

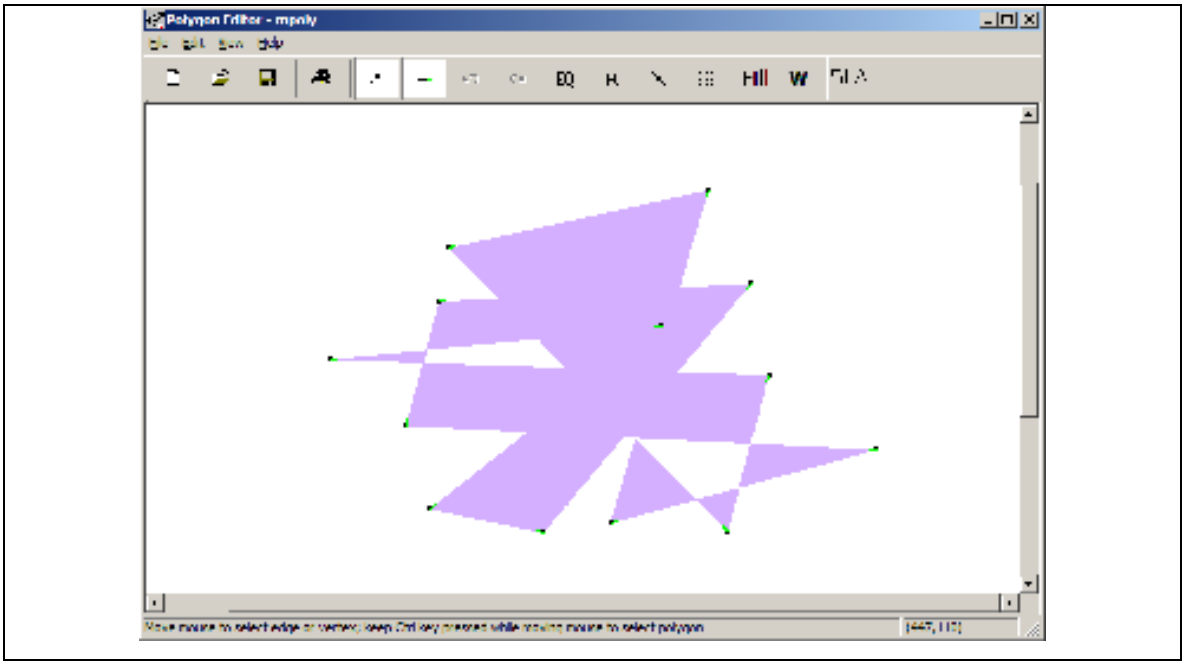

Figure 18 SLA fill using NZW filling rule

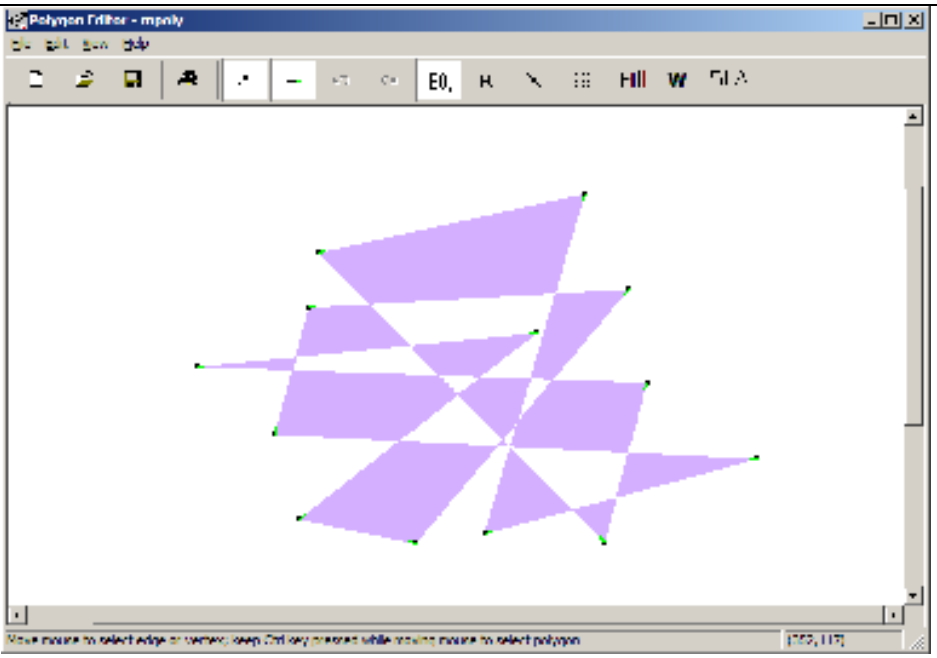

Figure 19 The SLA fill using SLA filling rule

5.2 Performance

The performance of the NZW algorithm was compared with the scan line algorithm, which is the current filling technique typically used in industry. For this purpose, the SLA was implemented[1] based on an outline given in [13]. The following sections dicuss both the empirical and the theoretical correctness of the NZW algorithm.

5.2.1 Theoretical complexity

The factors affecting the performance of the NZW and SLA algorithm are assumed to be:

1. the number of vertices (n),
2. the number of intersections (i), and
3. the screen resolution (*height in pixels*)

The theoretical complexity of the NZW algorithm is $O(s \log s)$ where s is the number of intersections (including the polygon vertices, also regarded as intersections) in the input complex polygon. Since the algorithm basically traverses intersection points (a linear contribution,) and sorts intersection on an edge, with the worst-case being all

[1] Both the filling rules, NZW as well as EO were provided.

intersections occurring on a single edge, yielding $O(s \log s)$. The number of intersection is bounded by $O(n^2)$, where n is the number of vertices (i.e., edges) in the input polygon. Thus, in the worst case, the NZW algorithm has complexity $O(n^2 \log n)$. Since this algorithm is object-precision, the height of the polygon in pixels, which is dependent on the screen resolution, should not affect its performance.

On the other hand, the SLA is an image-precision method, which scans every scan line from top-to-bottom and left-to-right. So the time taken to fill a complex polygon by the SLA, using the non-zero winding number filling rule was assumed to be directly proportional to both the width and the height of the polygon. The SLA calculates the intersection of the polygon edges with the scan line, while scan-converting, using edge-coherence. Thus, the SLA does not partition a polygon, but only decides on which parts need to be filled. SLA does not require the intersections points to be stored, but a test of whether the intersection of two edges with the current scan line changes their relative left-to-right order (i.e., two edges intersect) must be made in any case. Any change in the number of intersections, should have no effect on the SLA, for a given number of vertices of a polygon and for a given height of the polygon in pixels.

In addition, we also wanted to study the performance of the algorithms with respect to the number of vertices of the polygons, for a given number of intersections and height.

5.2.2 Empirical relative performance

5.2.2.1 Performance test suite

A test data set was constructed that consisted of 6 groups of randomly generated polygons whose vertex count were 20, 30, 40, 60, 80 and 100, and each of those having 4 different numbers of intersections (depending on the vertex count). These 24 polygons were again categorized into six separate groups, scaled with different heights of (in the order of) 200, 400, 800, 1600, 3200 and 6400 pixels.

The base polygons were first generated for the given number of vertices and intersections, in a rectangular area of 420 × 700, with a height of 400 pixels. From these

base polygons, the required data sets were then generated by scaling the *y*-values by 0.5, 2, 4, 8 and 16, respectively. The scaling was done only in the *y*-direction, because the SLA depends on the number of scan lines and not in the *x*-direction[2]. There are 10 distinct polygons in each category, and this is believed to provide a sufficiently representative sampling.

The different number of vertices, number of intersections and the heights of the polygons were selected based on the discussion with Dr. Hain, who is quite knowledgable in this field. It was gathered that most commonly used polygons for printing had vertices in the range of 40-100, with less than 50 intersections and their average height was 6400 pixels. The height chosen is a representation of any figure that around 5" long (at 600 or 1200 dots per inch, equivalent to respectively 3200 and 6400 pixels) on a 1200 dots per inch high-resolution printer, which is a reasonable dimension for comparison. Since we do not have an actual data set of practical polygons, we compared the algorithms with polygons of typical size that are used in the printer industry.

For comparison, both the algorithms, NZW and SLA were executed with the same set of data on a machine with the system specifications as shown in Table 1.

Table 1 System Specifications

OS Name	Microsoft Windows XP Professional
Version	5.1.2600 Service Pack 1 Build 2600
OS Manufacturer	Microsoft Corporation
System Manufacturer	Gateway
System Model	E-3600
System Type	X86-based PC
Processor	x86 Family 15 Model 1 Stepping 2 GenuineIntel ~1694 Mhz
BIOS Version/Date	Intel Corp. PT84510A.15A.0003.P01.0110311619,

[2] No actual filling was done, and so the algorithm is independent of the number of pixels in the x-direction.

	10/31/2001
SMBIOS Version	2.3
Hardware Abstraction Layer	Version = "5.1.2600.1106 (xpsp1.020828-1920)"
Total Physical Memory	512.00 MB
Available Physical Memory	251.50 MB
Total Virtual Memory	1.67 GB
Available Virtual Memory	1.10 GB
Page File Space	1.17 GB

5.2.2.2 Performance results

The execution times for the compiler-optimized SLA and NZW algorithms operating on a variety of inputs were collected. Specifically, the effect of the number of vertices, the number of intersections, and the polygon height on the execution time of both the algorithms was measured. Each of these measurements will be discussed separately.

5.2.2.2.1 Performance versus polygon height

The effect of height on the execution time for partitioning a polygon with given a number of vertices and intersections is shown in Figure 20. It can be seen that for NZW the running time is essentially constant with height, as was expected since this is an object-precision algorithm. The results of the exection times for three different sets of polygons are shown to compare the results with the practical sizes. It can be seen that for polygons with 106 vertices and 100 intersections, which are considerably complex, NZW has an impressive execution time as compared to SLA.

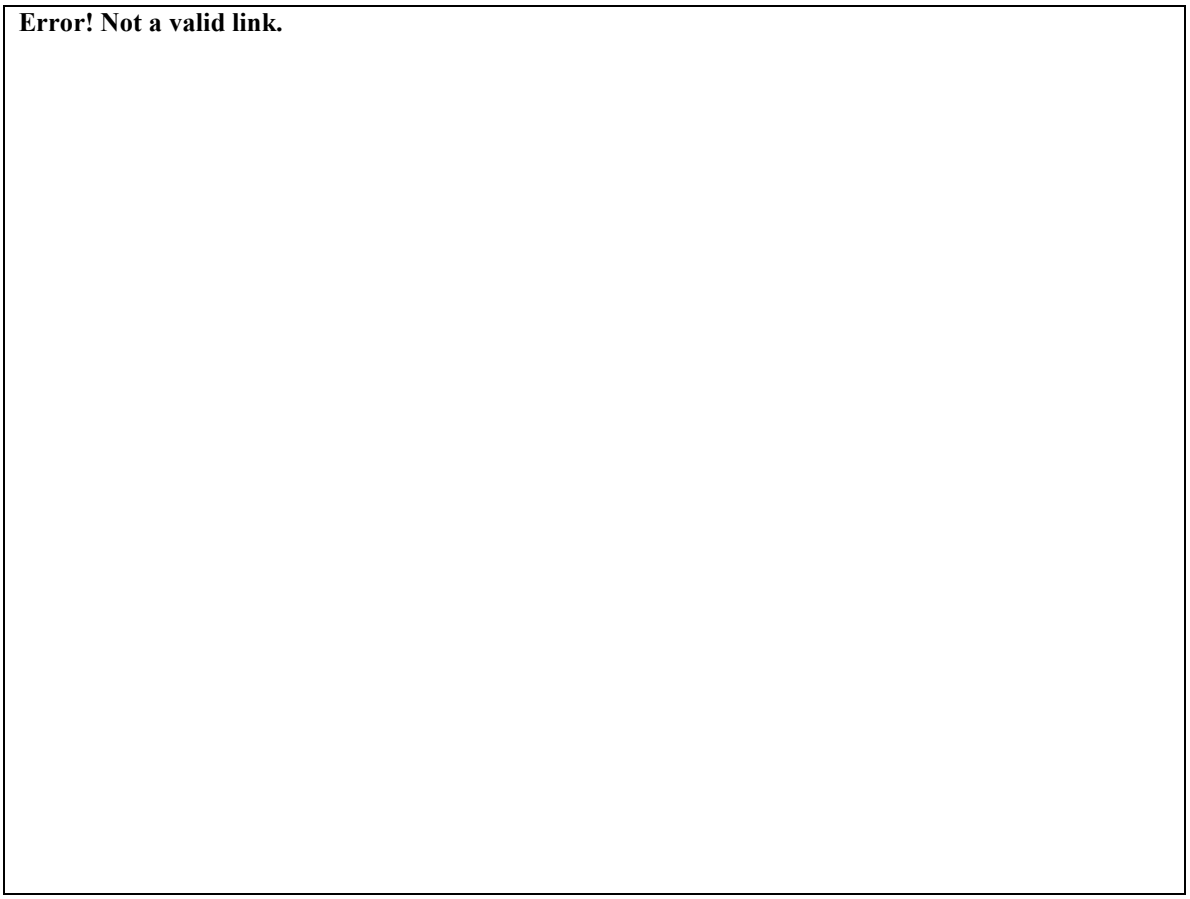

Figure 20 NZW time *vs*. Height of the polygons (trend lines)

Conversely, the SLA timings increase linearly with height, again as was expected, since the number of operations is proportional to the number of scan lines (i.e., height). The graph shown in Figure 21 shows the trend lines of two different sets of polygons, with respect to the height of the polygon. It can be seen that, although for polygons of height 200 pixels and with around 20 vertices SLA seems to perform as well as NZW, for polygons of practical height NZW runs considerably faster than the SLA. For this reason, the performance of the NZW in the following sections are discussed for polygons with height of 6400 pixels.

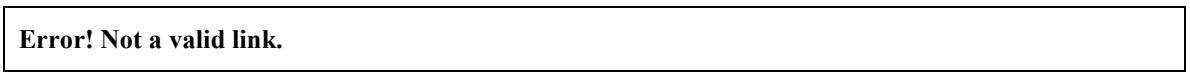

Figure 21 SLA time *vs.* Height of the polygons (trend lines)

5.2.2.2.2 Performance versus number of vertices

The effect of the number of vertices on the performance of the SLA and NZW algorithms was tested from the results obtained. For each of the chosen number of vertices, the random polygon generation algorithm was used to produce a set of 10 polygons. Each of these polygons that were generated had 50 intersections.

Figure 22 shows the effect of the number of vertices on the execution time for polygons, with number of intersections and height kept constant. It can be seen from the results that NZW algorithm runs faster than SLA for a sample data of practical size and height.

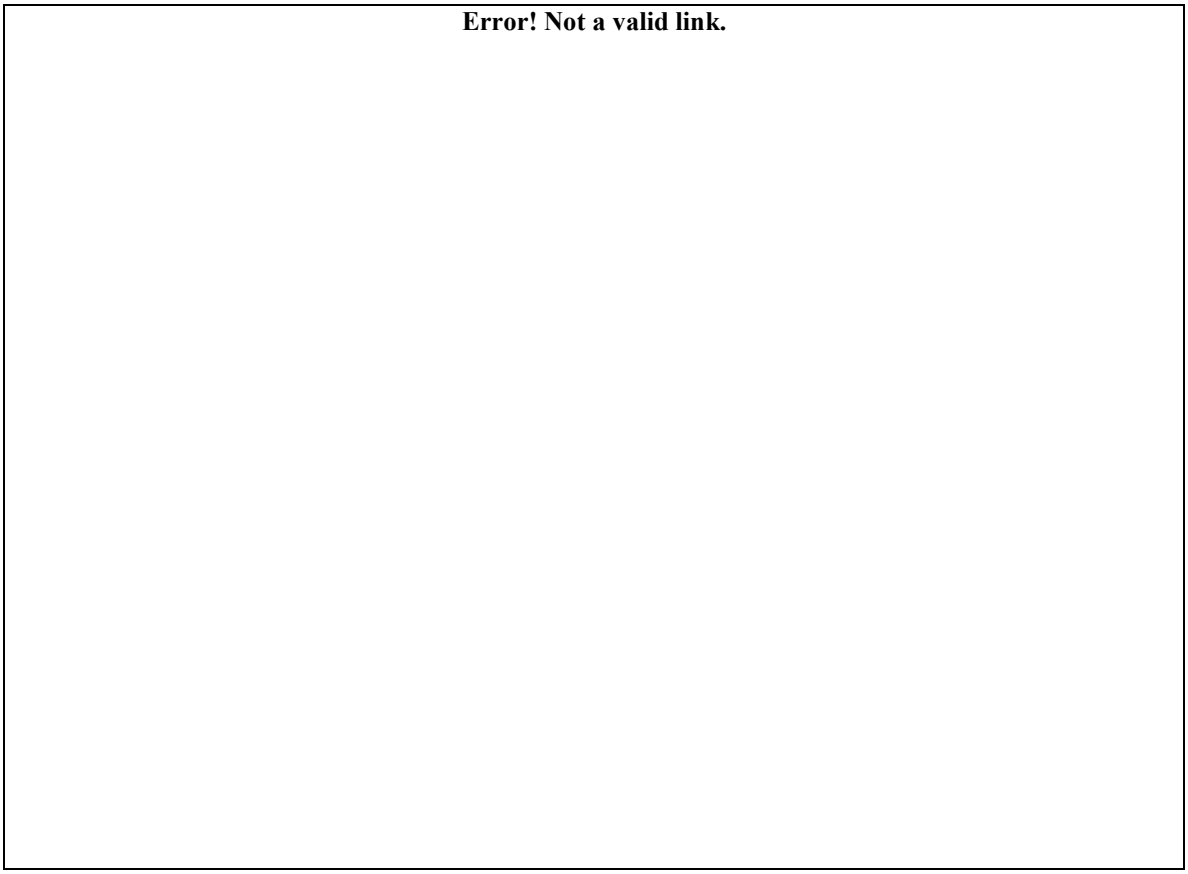

Figure 22 Performace *vs.* Number of Vertices (50 intersections)

5.2.2.2.3 Performance versus number of intersections

Timing tests were conducted to find the effect of the number of intersections on the NZW and SLA algorithms, keeping the number of vertices constant at 100. The results are shown in Figure 23.

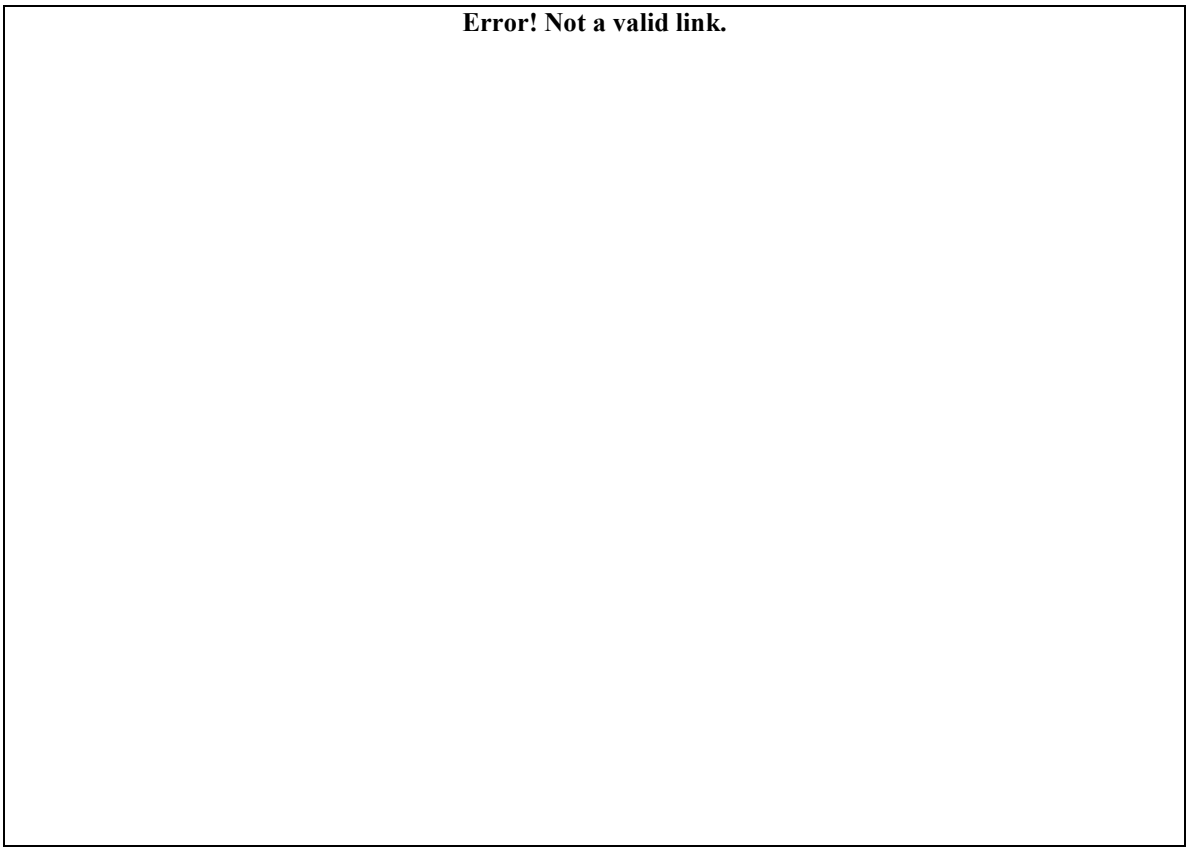

Figure 23 Execution Time *vs.* Number of Intersections (for 100-vertex polygons)

It can be seen that the execution time for SLA is relatively independent of the number of intersections, as was expected. Surprisingly, this was, to a lesser degree, also true for NZW. That is, the NZW algorithm performs well, even with a greater number of intersections. Since NZW runs much faster than SLA, it is more efficient than SLA, especially for polygons of practical sizes.

5.2.2.3 Relative Performance of NZW

The *relative performance* (ratio of the execution time of SLA vesus the execution time of NZW) as a function of the number of intersections has also been calculated. It is shown in Figure 24 for polygons with 20 vertices, and in Figure 25 for polygons with 86 vertices. Clearly, the greater the number of intersection, the more "complicated" is a polygon. It can be seen that SLA runs almost as fast as NZW for quite complicated polygons (86 verticecs and 100 intersections) at very low resolution (200 pixels). However, for similarly complicated polygons in more demanding scenarios required by the higher resolution of modern printers (1200 dot per inch), NZW typically runs more than 10 times faster than SLA.

Error! Not a valid link.

Figure 24 Performace Ratio of NZW vs. Number of Intersections (20-vertex polygons)

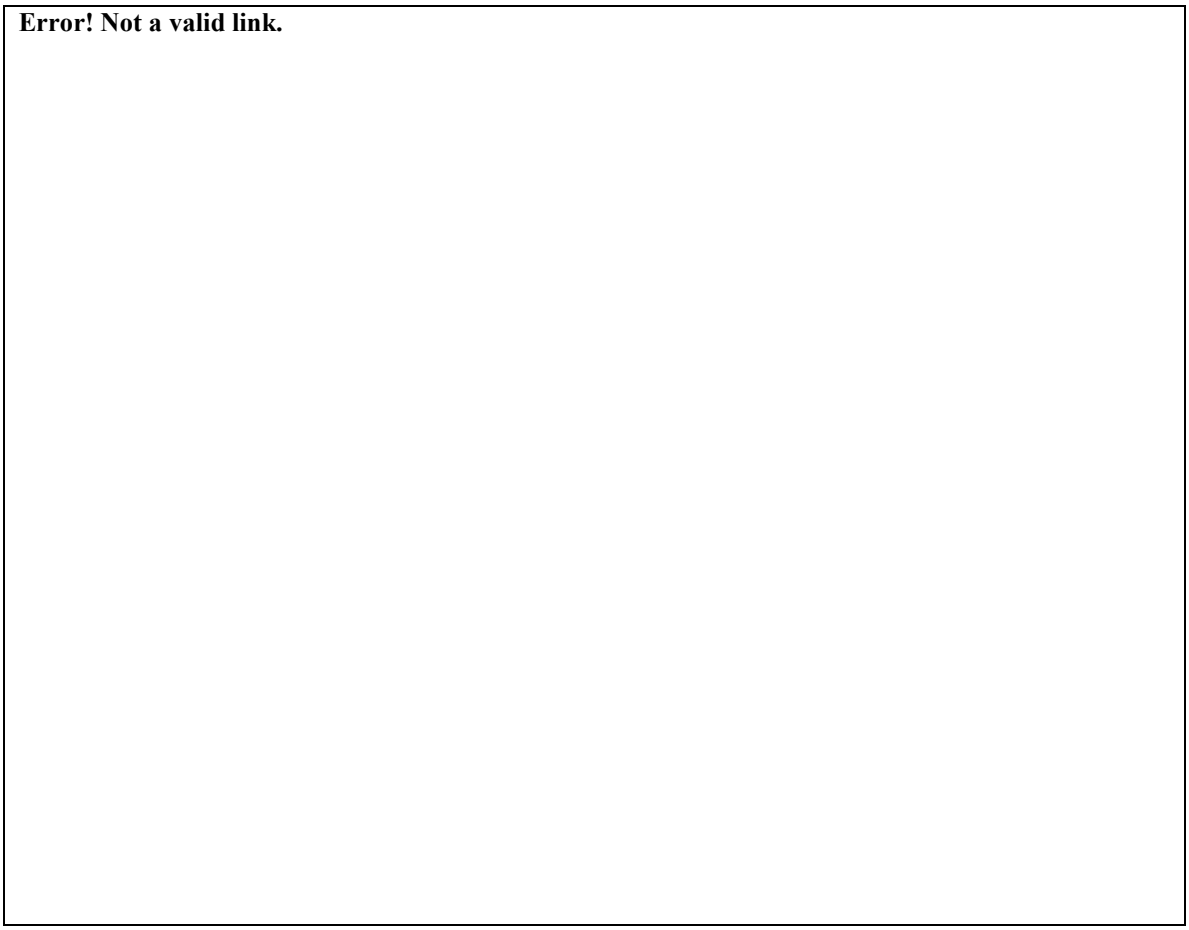

Figure 25 Performance Ratio *vs.* Number of Intersections (86-vertex polygons)

5.3 Conclusions

The design and implementation of the NZW algorithm, and the thorough testing of the algorithm using the interactive testbed described in Section5.1 confirms the first hypothesis which states that an object-precision algorithm can be implemented to partition a complex polygon into a set of simple polygons (using the NZW rule.)

The second hypothesis states that the NZW algorithm fills a complex polygon no slower than the SLA, and in complicated cases faster than the SLA. This hypothesis is confirmed by the results shown above, and for the following cases:

1. With the height and number of intersections of polygons kept constant, NZW executes faster than SLA for polygons with varying number of vertices, as shown in Figure 22.

2. With height and number of vertices of polygons kept constant, NZW executes faster than SLA for polygons with varying number of intersections, as shown in Figure 23

Section 5.2.2.3 on relative performance as a function of the number of intersections, mapped for different heights, further illustrates this hypothesis. The results clearly indicate that NZW runs 10 times faster than SLA for practical sizes. This is an amazing and gratifying result. Also, since NZW is an object-precision algorithm, there is the side-effect of higher precision results. The performance evaluation of the NZW algorithm described here shows it to be superior to the most-commonly used algorithm(SLA), and makes it a valid and valuable pre-processing algorithm useful in graphics libraries of modern laser printers.

5.4 Future work

Future work leading from this research could include the following two extensions.

1. Adapting the NZW algorithm to fill complex polygons using the EO rule. This is a relatively simple modification.
2. Adapting the NZW algorithm to generate the union, intersection, and difference polygons of two or more given overlapping simple concave polygons. While object-precision algorithms for these operation on convex polygons are well-known, the NZW algorithm provides the basis for the same operations on concave polygons, which are currently performed using a variation of SLA. The algorithm is in fact very close to what is needed. For example, the algorithm is already capable, given two overlapping simple concave polygons having the same orientation, of producing a component (in almost the same sense as is used in this thesis) polygon, with possible holes, representing the union of the two polygons. By reversing the orientation of one of the polygons, component polygons representing the differences are generated, and the union of these is the inverse of the intersection

.

REFERENCES

REFERENCES

[1] Adobe Systems Incorporation, *Postscript Language Reference Manual*, Second Edition, Addison-Wesley.

[2] Amato, Nancy M., Michael T. Goodrich, Edgar A. Ramos, "A Randomized Algorithm for Triangulating a Simple Polygon in Linear Time" http://citeseer.nj.nec.com/amato00randomized.html (2000).

[3] Athar Luqman Ahmad "Approximation of a Bezier curve with minimum number of line segments", Master's Thesis, School of Computer and Information Science, University Of South Alabama, USA, May, 2001.

[4] Auer, Thomas, and M. Held, "Heuristic for generation of random polygons", Proc. of 8th Canadian Conf. of Comp. Geom, Ottawa, Canada, August 1996, Carleton University Press.

[5] Auer, Thomas. "Heuristic for generation of random polygons", Masters Thesis, Computer wissenshaften Uni. Of Salzburg, Austria, June 1996.

[6] Bentley J.L. and Ottmann T.A., "Algorithms for reporting and counting geometric intersections," IEEE *Transactions on Computers*, 28, 643-647 (1979).

[7] Chazelle B. and Edelsbrunner H., " An Optimal algorithm for intersecting line segments in the Plane," Journal of ACM, Vol. 39 No. 1, pages 1-54, 1992

[8] Chazelle, B. and J. Incerpi, "Triangulation and shape-complexity", *ACM Trans. Graphics*, Vol. 3, No. 2 (April, 1984), pages 135–152.

[9] Chazelle, Bernard, "Triangulating a simple polygon in linear time", *Discrete Computational Geometry* 6 (1991), pages 485–524

[10] Chazelle, Bernard, and Leonidas J. Guibas. "Visibility and Intersection problems in Planar Geometry", *Discrete Computational Geometry* 4 (1989), pages 551–581

[11] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan, "Polygon triangulation in $O(n \log\log n)$ time with simple data structures", *Proc. 6th Annual. ACM Symposium of Computational. Geom.*,1990, pages 34-43.

[12] De Berg, M, M. van Kreveld, M Overmars, O. Scharzkopf, *"Computational Geometry Algorithms and Applications"* (Springer-Verlag Berlin Heidelberg, 1997).

[13] Foley, D.J., A. Van Dam, S.K. Feiner, anokd J.F. Hughes*, Computer Graphics Principles and Practice*, Addison Wesley, 1995.

[14] Fournier A. and Montuno D.Y., "Triangulating simple polygons and equivalent problems," ACM Transactions on Graphics, 1984, pages 3(2): 153-174.

[15] Ghostscipt home page, http://www.ghostscript.com/.

[16] Grey M.R., Johnson D.S., Preparata, F.P., and Tarjan R.E., "Triangulating a Simple Polygon", *Information Processing Letter*, 7: pages 175-179,1978.

[17] Hain T.F. "A Fast, Practical Algorithm for the Trapezoidation of Simple Polygons," Internal Report USA-CIS 971001, University of South Alabama, 1997 (Submitted to ACM Transactions on Graphics).

[18] Hain, T. F., and Gulve, S. "Interactive, Visual Testing Strategy for Computational Geometry Problems," ACM Southeast Conference, Atlanta, GA, April 1998.

[19] Hain, T. F., private communication, (2002).

[20] http://mathworld.wolfram.com

[21] Laszlo, Michael J., *Computational Geometry and Computer Graphics in C++*, (Prentice Hall Inc., 1996).

[22] Lee D.T. and Preparata F.P. "Location of a point in a planar subdivision and its application," SIAM Journal of Computation, 6: pages 594-606, 1977.

[23] Lee D.T. and Preparata F.P., "Location of a point in a planar subdivision and its application," SIAM J. Computational Geometry, 6:pages 594-606, 1977.

[24] Lennes N.J. "Theorems on Simple Polygon and Polyhedron," American Journal of Mathematics, 33:37-62, 1911.

[25] Lipton, Richard J., and Robert Endre Tarjan, "A Separator theorem for Planar Graphs", *SIAM Journal Of Applied Mathematics*, Vol. 36, No. 2, (April 1979)

[26] Preparata, Franco P., and Michael Ian Shamos, *Computational Geometry: An Introduction*, (January 1991, Springer).

[27] Romney, Bruce, Cyprien Godard,, Michael Goldwasser, and G. Ramkumar. "*Efficient System For Geometric Assembly Sequence Generation And Evaluation* ", Proceedings of the ASME International Computers in Engineering Conference, Boston, Massachusetts, September 1995, pages 699-712.

[28] Toussaint G. " Triangulation and Arrangements", *All Institute Lecture at McGIll University*, Montreal, 1993.

APPENDIX

APPENDIX A

CODE

```
// Multipolyon.h

// Written by T. Hain. Extended by L. Subramaniam, Spring, 2003

#ifndef _MULTIPOLY
#define _MULTIPOLY
#define __MFC

#include <list>
#include <vector>
#include <queue>
#include <ctime>
#include <algorithm>
#include <stack>
#include "stdafx.h"

using namespace std;

typedef float Coord;

////////////////////////////////////////////////////////////////////////////
//  Vertex: vertex
//
//  Constructors
//      Vertex()
//      Vertex(const int x, const int y)
//      Vertex(const CPoint &point)
//
//  Assignment
//      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)
class Vertex: public CObject
{
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; }

    bool operator<= (const Vertex &v) const
    { return x <= v.x; }

    bool operator> (const Vertex &v) const
    { return y < v.y || 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); }
#endif
    Vertex& snap(Coord resolution)    // Snap coords to given resolution
    {
        if (resolution)
        {
            x = resolution * (int)(x/resolution);
            y = resolution * (int)(y/resolution);
        }
        return *this;
    }
};

class LineSegment;
struct Rect
{
    Coord x0, y0, x1, y1;

    Rect(const Coord _x0 = 0, const Coord _y0 = 0,
        const Coord _x1 = 0, const Coord _y1 = 0)
        : x0(_x0), y0(_y0), x1(_x1), y1(_y1) {}

    Rect(const Vertex &v0, const Vertex &v1): x0(v0.x), y0(v0.y), x1(v1.x), y1(v1.y) {}

    Rect(const Rect &r): x0(r.x0), y0(r.y0), x1(r.x1), y1(r.y1) {}

    void normalize()
    {
        if (x0 > x1)
            swap(x0,x1);
        if (y0 > y1)
            swap(y0,y1);
    }
```

```
    // Precondition: rectangle is normalized
    bool isVertexInside(const Vertex &v) const
    { return v.x >= x0 && v.x <= x1 && v.y >= y0 && v.y <= y1; }

    // Precondition: rectangle is normalized
    void inflate(const Coord &dx, const Coord &dy)
    {
        x0 -= dx;
        x1 += dx;
        y0 -= dy;
        y1 += dy;
    }
};

class LineSegment
{
    static const float NOISE;  // used for determining limit of parallel lines

public:
    Vertex vertex0, vertex1;
    bool m_swappedVertices;   // have vertices been swapped during sortVertices?
    Coord ymin, ymax;         // y component of bounding box [don't use before call to
sortVertices()]
    typedef enum
    {
        PARALLEL,      // lines are parallel within tolerance level
        NO_INTERSECT,  // lines segments don't intersect
        INTERSECT      // line segments intersect
    } IntersectionType;

    LineSegment(){}
    // Sets bounding box of line segment to (vertex0.x, ymin, vertex1.x, ymax)
    void sortVertices()
    {
        m_swappedVertices = !(vertex0 < vertex1);
        if (m_swappedVertices)
            swap(vertex0, vertex1);
        if (vertex0.y < vertex1.y)
        {
            ymin = vertex0.y;
            ymax = vertex1.y;
        }
        else
        {
            ymin = vertex1.y;
            ymax = vertex0.y;
        }
    }

    LineSegment(const Vertex &vtx0, const Vertex &vtx1): vertex0(vtx0), vertex1(vtx1) {}

    LineSegment(const LineSegment &l): vertex0(l.vertex0),vertex1(l.vertex1),ymin(l.ymin),
        ymax(l.ymax), m_swappedVertices(l.m_swappedVertices) {}

    IntersectionType calcIntersection(const LineSegment &l, Vertex &intersection,
        float &alpha, float &alpha1);

    bool isVertexNear(const Vertex &p, const Coord &resolution);

    // order by left edge of bounding box
    bool operator< (const LineSegment &ls) const
    {
        return vertex0 < ls.vertex0;
    }
};

typedef list<Vertex>::iterator VtxIt;
```

```
//////////////////////////////////////////////////////////////////
//             Intersection - to create the master list
// Constructors
// Intersection()
// Intersection (const Intersection &i )
// Intersection (const Vertex &vtx,LineSegment &li,LineSegment &lj)
// Intersection (const Vertex &vtx,Vertex &org1,Vertex &org2,float &p1,float
&p2,LineSegment &li,LineSegment &lj)

// Assignment
//   void operator= ( const Intersection &i)

// Relational
//   bool operator< (const Intersection &i) const
//   bool operator<= (const Intersection &i) const
```

```
class Intersection
{
public:
    Vertex v;
    float param1,param2;
    int index1, index2;
    Vertex origin1, origin2;
    int winding;
    int direction;
    int selfIndex;
    LineSegment l1, l2;

    Intersection(){}

    Intersection (const Vertex &vtx, Vertex &org1, Vertex &org2,
        float &p1,float &p2, LineSegment &li, LineSegment &lj)
    {
        v = vtx;
        index1 = -1;
        index2 = -1;
        param1 = p1;
        param2 = p2;
        origin1 = org1;
        origin2 = org2;
        l1 = li;
        l2 = lj;
    }

    Intersection ( const Vertex &vtx,  LineSegment &li, LineSegment &lj)
    {
        v = vtx;
        index1 = -1;
        index2 = -1;
        param1 = 0;
        param2 = 0;
        origin1 = v;
        l1 = li;
        l2 = lj;
        winding = 0;
        direction = 0;
        selfIndex = 0;
    }

    Intersection( const Intersection &i): v(i.v), index1(i.index1), index2(i.index2),
        param1(i.param1),param2(i.param2), origin1(i.origin1), origin2(i.origin2),
        l1(i.l1), l2( i.l2 ), winding  (i.winding), direction (i.direction),
        selfIndex(i.selfIndex) {}

    bool operator< (const Intersection &i) const
    { return (v < i.v ); }
    bool operator<= (const Intersection &i) const
    { return (v <= i.v ); }

    void operator= ( const Intersection &i)
    {
        v = i.v;
        index1 = i.index1;
        index2= i.index2;
        param1 = i.param1;
        param2 = i.param2;
        origin1 = i.origin1;
        origin2 = i.origin2;
        l1 = i.l1;
        l2 = i.l2;
        winding = i.winding;
        direction = i.direction;
        selfIndex = i.selfIndex;
    }
};
```

```
typedef vector<Intersection> ::iterator InterIt;
//////////////////////////////////////////////////////////////////////////
//                nVertex - temporary class to create the objects in the
//                polygon edge array
// Constructors
// nVertex()
// nVertex ( const Vertex &vtx, float &p, int & i, LineSegment &l2 )
// nVertex ( const Vertex &vtx, int & i, LineSegment &l2 )
// nVertex ( const Vertex &vtx,  LineSegment &l2 )
//
// Assignment
//  void operator= ( const nVertex &i)
// Relational
//  bool operator< (const nVertex &n) const
//  bool operator<= (const nVertex &i) const

class nVertex
{
public :
    Vertex v;
    float param;
    int index;
    LineSegment l;

    nVertex() {}

    nVertex ( const Vertex &vtx, float &p, int & i, LineSegment &l2 )
    {
        v = vtx;
        param = p;
        index = i;
        l = l2;
    }

    nVertex ( const Vertex &vtx, int & i, LineSegment &l2 )
    {
        v = vtx;
        param = 0;
        index = i;
        l = l2;
    }

    nVertex ( const Vertex &vtx,  LineSegment &l2 )
    {
        v = vtx;
        l = l2;
    }

    void operator= ( const nVertex &i)
    {
        v = i.v;
        l = i.l;
        param= i.param;
        index = i.index;
    }

    bool operator< (const nVertex &n) const
    { return (param < n.param ); }

    bool operator<= (const nVertex &i) const
    { return (param <= i.param ); }
};
```

```
//////////////////////////////////////////////////////////////////////////////
//                    Pseudovertex - the polygon edge array
// Constructors
//   Pseudovertex()
//   Pseudovertex (const nVertex &n)
//   Pseudovertex (const Pseudovertex &p)
//
// Relational
//   bool operator< (const Pseudovertex &s) const
// Comparison
//   bool operator== (const Pseudovertex &p) const

class Pseudovertex
{
public:
    vector<nVertex> ilist;

    Pseudovertex(){}

    Pseudovertex ( const nVertex &n ) { ilist.push_back(n); }

    Pseudovertex ( const Pseudovertex &p ): ilist(p.ilist) {}

    bool operator< (const Pseudovertex &s) const
    { return ( ilist.begin()->v < s.ilist.begin()->v ); }

    bool operator== (const Pseudovertex &p) const
    { return ( ilist.begin()->v == p.ilist.begin()->v ); }
};

typedef vector<Pseudovertex>::iterator PseudoIt;

//////////////////////////////////////////////////////////////////////////////
//  Poly: polygon; a cyclic collection of vertices
//
//  Constructors
//      Poly()
//      Poly(const CPoint& point): create triangle around point
//      Poly(const Poly& poly)
//
//  Assignment
//      void operator= (const Poly& poly)
//
//  Traversal
//      Precondition: currPos must point to valid vertex.
//      Vertex& GetNextCirc(POSITION &currPos): for circular linkage
//      Vertex& GetPrevCirc(POSITION &currPos): for circular linkage
class Poly
{
public:
    list<Vertex> vtxList;

    Poly(const Poly &p): vtxList(p.vtxList){}

    Poly(){}

    Poly(const Vertex &vtx, Coord resolution)
    {
        vtxList.push_back(Vertex(vtx.x + 10.0f, vtx.y - 10.0f).snap(resolution));
        vtxList.push_back(Vertex(vtx.x - 10.0f, vtx.y - 10.0f).snap(resolution));
        vtxList.push_back(Vertex(vtx.x                , vtx.y + 10.0f).snap(resolution));
    }

    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;
    }

    int size() const
    { return vtxList.size(); }
};

typedef list<Poly>::iterator PolyIt;

////////////////////////////////////////////////////////////////////////////////
// IntersectionList - list of polygon edges
// Constructors
// IntersectionList ()
// IntersectionList( const vector<Pseudovertex> &p)
class IntersectionList
{
public:
    vector<Pseudovertex> p_list;

    IntersectionList () {}

    IntersectionList( const vector<Pseudovertex> &p) : p_list(p)  {}
};

////////////////////////////////////////////////////////////////////////////
//  MultiPolygon: collection of polygons
//
////////////////////////////////////////////////////////////////////////////
//  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 opertator
//          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:
    list<Poly> m_polyList;

    void random(Rect boundRect, int nVtx, int nPoly = 1);

    void dump();

    bool read(const char* file_name);

    bool write(const char* file_name);

    bool writePS(const char* file_name, bool bEOfill);

    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);

    vector<Vertex> findIntersections
        (IntersectionList &List, vector<Intersection> &tempList);
    vector<Vertex>  findMonotone(MultiPoly &resMPoly, vector<int> &windingVector);

    void fillAddress(IntersectionList &ivList, vector <Intersection> &interVector);

    void fillIndices(IntersectionList &ivList, vector <Intersection> &interVector);

    LineSegment::IntersectionType findIntersection( LineSegment &l1,  LineSegment &l2,
        Vertex &intersection, float &alpha1, float &alpha2);

    void gridify(Coord resolution);   // Snap all vertex coordinates to given resolution

    void polygonPartition(MultiPoly &resMPoly, vector<Intersection> &tempList,
        vector<int> &windingVector);
```

```
    // cross-product, pq x qr
    float xProd(const Vertex &p, Vertex &q, Vertex &r)
    {
        return ( (q.x * ( r.y - p.y )) + (p.x*( q.y - r.y )) + (r.x * ( p.y - q.y )) );
    }
};

typedef MultiPoly::MPPos MPPos;
#endif
```

```
// Multipolygon.cpp

#include "MultiPolygon.h"
#include <cmath>
#include <fstream>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;
const float LineSegment::NOISE = 1e-5f;

//////////////////// Line Intersection ////////////////////////
//
//  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 lines
//                 segments were parallel, non-intersecting, or intersecting.
//                 "intersectionPoint" is valid iff the returned type is not PARALLEL.
//
LineSegment::IntersectionType LineSegment::calcIntersection
    (const LineSegment &l, Vertex &intersection, float &alpha, float &alpha1)
{
    float dx21 =   vertex1.x -   vertex0.x, dy21 =   vertex1.y -   vertex0.y,
        dx43 = l.vertex1.x - l.vertex0.x, dy43 = l.vertex1.y - l.vertex0.y;
    float dem  = dx21 * dy43 - dy21 * dx43;

    // parallel lines
    if (fabs(dem) < NOISE)
        return PARALLEL;
    else {
        float dx12 = vertex0.x - l.vertex0.x,
            dy12 = vertex0.y - l.vertex0.y;
        float dx = l.vertex0.x - vertex0.x,
            dy = l.vertex0.y - vertex0.y;


        alpha = (dy12*dx43 - dx12*dy43)/dem;

        alpha1 = ( dy21*dx - dx21*dy)/dem;

        // The intersecting point

        intersection.x = vertex0.x + dx21 * alpha;
        intersection.y = vertex0.y + dy21 * alpha;

        // test for segment intersecting (alpha)

        if ((alpha < 0.0) || (alpha > 1.0))
            return NO_INTERSECT;
        else {
            float num = dy12*dx21 - dx12*dy21;
            if (dem > 0.0) {
                if (num < 0.0 || num > dem)
                    return NO_INTERSECT;
            }
            else {
                if (num > 0.0 || num < dem)
                    return NO_INTERSECT;
            }
        }
    }
    return INTERSECT;
}

bool LineSegment::isVertexNear(const Vertex &vtx, const Coord &resolution)
/*
*  Determine whether point vtx is within distance 'resolution' from line segment.
*
```

```
*  Written by T. Hain, Jan 1998.
*/
{
    //  CRect activeRect(vertex0.m_point, vertex1.m_point);
    //  activeRect.NormalizeRect();
    //  activeRect.InflateRect(resolution, resolution);  // for vertical or horizontal
edges
    //  if (!activeRect.PtInRect(*this))
    //      return false;  // trivial reject (i.e., *this is not within line seg's
bounding box)
    //Rectangle rect(*this);
    //rect.inflate(resolution, resolution);
    //if (!rect.isVertexInside(vtx))
    //  return false;
    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);
}

////////////////////// MultiPoly //////////////////////
//
MultiPoly::MPPos MultiPoly::GetActiveVtxPos(const Vertex &vtx, Coord resolution)
//  Find first vertex such that vtx is within a distance 'resolution' of it.
//  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' of it.
//  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 (int i = 0; i < 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, bool bEOfill)
{
    ofstream out(file_name);
    Rect rect(boundingBox());
    out << "/poly {newpath";
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        out << "\n\t" << pi->vtxList.begin()->x << " " << pi->vtxList.begin()->y
            << " moveto\n";
        for (VtxIt vi = pi->vtxList.begin()++; vi != pi->vtxList.end(); ++vi)
            out << "\t" << vi->x << " " << vi->y << " lineto\n";
        out << "\tclosepath\n";
    }
    out << "} def\n\t16 774 translate\n\t0.8 setgray\n\tpoly "
        << (bEOfill? "eofill\n" : "fill\n")
        << "\t0 setgray\n\tpoly stroke\n\tshowpage";

    return true;
}

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

void MultiPoly::dump()
{
    ofstream out("dump.txt");
    out << m_polyList.size() << "\n";
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        out << pi->vtxList.size() << " ";
        for (VtxIt vi = pi->vtxList.begin(); vi != pi->vtxList.end(); ++vi)
            out << vi->y << " " << vi->x << " ";
    }
    out << "\n";
}
```

```
// generates a random multi polygon within the given rectangular area,
// for a given number of vertices and number of polygons
void MultiPoly::random(Rect boundRect, int nVtx, int nPoly)
{
    srand( (unsigned)time( NULL ) );
    boundRect.normalize();
    m_polyList.clear();
    for (int p = 0; p < nPoly; ++p)
    {
        Poly poly;
        for (int v = 0; v < nVtx; ++v)
        {
            Vertex vtx
                (
                boundRect.x0 + rand()%((int)(boundRect.x1-boundRect.x0)),
                boundRect.y0 + rand()%((int)(boundRect.y1-boundRect.y0))
                );
            poly.vtxList.push_back(vtx);
        }
        m_polyList.push_back(poly);
    }
}

Rect MultiPoly::boundingBox()
{
    Rect rect(*m_polyList.begin()->vtxList.begin(), *m_polyList.begin()->vtxList.begin());
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
        for (VtxIt vi = pi->vtxList.begin(); vi != pi->vtxList.end(); ++vi)
        {
            rect.x0 = min(rect.x0, vi->x);
            rect.y0 = min(rect.y0, vi->y);
            rect.x1 = max(rect.x1, vi->x);
            rect.y1 = max(rect.y1, vi->y);
        }
        return rect;
}

// finds the intersection point between two line segments and thier parametric
// values with respect to each intersecting line segment
LineSegment::IntersectionType MultiPoly::findIntersection( LineSegment &l1,
    LineSegment &l2,  Vertex &intersection, float &alpha1, float &alpha2)
{
    LineSegment l3(l1);
    LineSegment l4(l2);
    l1.sortVertices();
    l2.sortVertices();
    Vertex vtx;
    LineSegment::IntersectionType resultType;
    if ( l1.vertex1 < l2.vertex0 ) // no x-overlap
    {
        // check for y-overlap
        if ( l1 < l2 )
        {
            if (l1.ymax < l2.ymin  )
                return (LineSegment::NO_INTERSECT );
        }
        else
            if (l2.ymin < l1.ymax)
                return LineSegment::NO_INTERSECT;
    }
    else
        // check for adjacency
        if ( (l1.vertex0 == l2.vertex0) ||(l1.vertex1 == l2.vertex1) ||
             (l1.vertex0 == l2.vertex1) || (l1.vertex1 == l2.vertex0) )
            return ( LineSegment::NO_INTERSECT );
    resultType = l3.calcIntersection(l4, vtx, alpha1, alpha2 );
    if ( resultType == LineSegment::INTERSECT )
    {
        intersection = vtx;
```

```
        return resultType;
    }
    return (resultType);
}

// this is the method called to fill the polygon useing NZW algorithm
// finds the intersections using trivial rejections considering the
// bounding box overlap and monotonic chains
vector<Vertex> MultiPoly::findMonotone(MultiPoly &resMPoly, vector<int> &windingVector)
{
    vector<Vertex> intersections;
    vector<Intersection> tempList;
    vector<Pseudovertex> p_list;
    p_list.clear();
    // add the vertices to the ivList in the form of Intersection
    for ( PolyIt pii = m_polyList.begin(); pii != m_polyList.end(); ++pii)
    {
        VtxIt vit0, vit1;
        for (VtxIt vi = pii->vtxList.begin(); vi != pii->vtxList.end();++vi)
        {
            Pseudovertex pseudo;
            vit0 = vi;
            vit1 = vi;
            pii->prevCirc(vit0);
            pii->nextCirc(vit1);
            LineSegment l1(*vit0,*vi);
            LineSegment l2(*vi, *vit1);
            l2.sortVertices();
            nVertex i(*vi, l2);
            tempList.push_back(Intersection ( *vi, l2, l1 ) );
            pseudo.ilist.push_back(i);
            p_list.push_back( pseudo );
        }
    }

    IntersectionList ivList(p_list);

    // add the intersection points to the list, as Intersection objects
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        VtxIt vi0 = pi->vtxList.begin(), vi1;
        unsigned int x_monotone=0;
        bool reached = false;
        vi1 = vi0;
        VtxIt start = vi0;
        ++vi1;
        //find intersection points for every polygon.
        while ( vi1 != pi->vtxList.end() && (reached == false))
        {
            LineSegment l1(*vi0, *vi1);
            vi0 = vi1;
            ++vi1;
            if ( vi1 == pi->vtxList.end() )
            {
                vi1 = start;
                reached  = true ;
            }
            LineSegment l2(*vi0, *vi1);
            LineSegment temp(l2 );
            VtxIt walker, walker1;
            walker = start;
            walker1 = walker;
            ++walker1;
            // walk thru the list from the beginning to find
            // intersection with the non-monotone edge
            while ( walker1 != vi0 )
            {
                LineSegment l1(*walker, *walker1 );
                l2 = temp;
```

```
                Vertex intersection;
                float al1, al2;
                if ( findIntersection(l1,l2,intersection, al1, al2)
                     == LineSegment::INTERSECT )
                {
                    intersections.push_back(intersection);
                    Vertex searchVertex1, searchVertex2;
                    if (l1.m_swappedVertices)
                        searchVertex1 = l1.vertex1;
                    else
                        searchVertex1 = l1.vertex0;
                    if ( l2.m_swappedVertices)
                        searchVertex2 = l2.vertex1;
                    else
                        searchVertex2 = l2.vertex0;
                    tempList.push_back( Intersection ( intersection, searchVertex1,
                        searchVertex2, al1, al2, l1, l2 ) );
                }
                walker= walker1;
                ++walker1;
            }
        }
    }

    vector<Vertex> resultIntersections = findIntersections(ivList, tempList);

    for ( unsigned int i=0; i < resultIntersections.size() ; i++ )
        intersections.push_back(resultIntersections[i]);

    sort ( ivList.p_list.begin(), ivList.p_list.end() );

    fillAddress(ivList, tempList);
    polygonPartition( resMPoly,tempList,windingVector);
    return intersections;
}

// finds the intersection points between every polygon edge
vector<Vertex> MultiPoly::findIntersections(IntersectionList &ivList,
vector<Intersection> &tempList)
{
    vector<LineSegment> edges, edges1;
    vector<Vertex> intersections;
    for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
    {
        PolyIt pi0 = pi;
        VtxIt vi0 = pi0->vtxList.begin(), vi1;
        unsigned nVtx = pi0->vtxList.size();
        for (unsigned i = 0; i < nVtx; ++i)
        {
            vi1 = vi0;
            pi0->nextCirc(vi1);
            Vertex v0(*vi0),v1(*vi1);
            LineSegment edge(*vi0, *vi1);
            edge.sortVertices();
            edges.push_back(edge);
            vi0 = vi1;
        }
        sort(edges.begin(), edges.end());
        ++pi0;
        while ( pi0 != m_polyList.end() )
        {
            VtxIt vi2 = pi0->vtxList.begin(), vi3;
            unsigned nVtx = pi0->vtxList.size();
            for (unsigned i = 0; i < nVtx; ++i)
            {
                vi3 = vi2;
                pi0->nextCirc(vi3);
                Vertex v0(*vi2),v1(*vi3);
                LineSegment edge(*vi2, *vi3);
```

```
            edge.sortVertices();
            edges1.push_back(edge);
            vi2 = vi3;
        }
        sort(edges1.begin(), edges1.end());
        // make vector of intersections
        unsigned nEdges = edges.size();
        unsigned nEdges1 = edges1.size();
        if (nEdges > 2)
        {
            for (unsigned i = 0; i < nEdges; ++i)
                for (unsigned j = 0; j < nEdges1; ++j)
                {
                    LineSegment li, lj;

                    if ( !edges[i].m_swappedVertices )
                        li = edges[i];
                    else
                    {
                        li.vertex0 = edges[i].vertex1;
                        li.vertex1 = edges[i].vertex0;
                    }
                    if ( !edges1[j].m_swappedVertices )
                        lj = edges1[j];
                    else
                    {
                        lj.vertex0 = edges1[j].vertex1;
                        lj.vertex1 = edges1[j].vertex0;
                    }
                    // Because of sort above: edges[i].rect.x0 <= edges[j].rect.x0

                    // if bounding boxes have x-overlap
                    if (edges1[j].vertex0.x < edges[i].vertex1.x)
                    {
                        // check for y-overlap of bounding boxes
                        if (edges[i].ymin < edges1[j].ymin)
                        {
                            if (edges[i].ymax <= edges1[j].ymin) continue;
                        }
                        else
                            if (edges1[j].ymax <= edges[i].ymin) continue;
                        // Check if both left [right] ends are not coincident (i.e.,
                        // valid intersection)
                        if (edges[i].vertex0 != edges1[j].vertex0 && edges[i].vertex1
                            != edges1[j].vertex1)
                        {
                            Vertex intersection;
                            float param1, param2;
                            if (li.calcIntersection(lj, intersection, param1, param2)
                                == LineSegment::INTERSECT)
                            {
                                intersections.push_back(intersection);
                                Vertex searchVertex1, searchVertex2;
                                if (edges[i].m_swappedVertices)
                                    searchVertex1 = edges[i].vertex1;
                                else
                                    searchVertex1 = edges[i].vertex0;
                                if ( edges1[j].m_swappedVertices)
                                    searchVertex2 = edges1[j].vertex1;
                                else
                                    searchVertex2 = edges1[j].vertex0;
                                li.sortVertices();
                                lj.sortVertices();
                                tempList.push_back( Intersection ( intersection,
                                    searchVertex1, searchVertex2, param1, param2,
                                    li , lj) );
                            }
                        }
                    }
```

```
                    }
                }
                ++pi0;
            }// end of while for pi0 counter
        }// end of for
        return intersections;
}

// a function to fill  polygon edge array(ivList) with the indices from the
// intersection master list
void MultiPoly::fillAddress(IntersectionList &ivList, vector<Intersection> &interVector)
{
    sort ( interVector.begin() , interVector.end() );
    for ( int i =0; i < interVector.size() ; i ++ )
    {
        if  ( (interVector[i].param1 != 0 ) && (interVector[i].param2 != 0 ) )
        {
            int i11 = 0, i12 = 0, i21 = 0, i22 = 0;
            Vertex searchVertex1 = interVector[i].origin1;
            Vertex searchVertex2 = interVector[i].origin2;
            while ( ivList.p_list[i11].ilist[0].v != searchVertex1)
                i11++;
            while ( ivList.p_list[i21].ilist[0].v != searchVertex2)
                i21++;
            nVertex Inter1(interVector[i].v,  interVector[i].param1, i,
                interVector[i].l1), Inter2(interVector[i].v, interVector[i].param2, i ,
                interVector[i].l2);
            ivList.p_list[i11].ilist.push_back(Inter1);
            ivList.p_list[i21].ilist.push_back(Inter2);
        }
        else
        {
            int vtxIt = 0;
            Vertex vtx = interVector[i].v;
            while ( ivList.p_list[vtxIt].ilist[0].v != vtx )
                vtxIt++;
            ivList.p_list[vtxIt].ilist[0].index = i;
        }
    }
    for ( PseudoIt ps = ivList.p_list.begin(); ps != ivList.p_list.end(); ++ps)
        sort ( ps->ilist.begin() + 1, ps->ilist.end()  );
    fillIndices( ivList, interVector );
}

// A method to fill in the last node of polygon edge array (ivList), which points to
// the ending vertex of the line
void MultiPoly::fillIndices ( IntersectionList &ivList, vector<Intersection>
                              &interVector)
{
    // setting up the last Vertex that points to the starting vertex of the
    // intersecting and setting up its corresponding index in the intersection
    // master list (interVector)
    for ( PseudoIt ps = ivList.p_list.begin(); ps != ivList.p_list.end(); ++ps)
    {
        int vtxIt = 0;
        Vertex searchVtx;
        LineSegment ls( (ps->ilist.end() - 1)->l );
        if ( ls.m_swappedVertices)
            searchVtx = ls.vertex0;
        else
            searchVtx = ls.vertex1;

        while ( interVector[vtxIt].v != searchVtx )
            vtxIt++;
        ps->ilist.push_back( nVertex( searchVtx, vtxIt,(ps->ilist.end() - 1)->l   ) ) ;
    }
```

```
    // setting up the indices for the intersections in the interVector
    for ( ps = ivList.p_list.begin(); ps != ivList.p_list.end(); ++ps)

        for (int i= 0; i < ps->ilist.size()-1; ++i)
        {

            nVertex tempVtx = ps->ilist[i+1];

            int tempIndex = ps->ilist[i].index;

            if ( interVector[tempIndex].origin2 == ps->ilist[0].v )
                interVector[tempIndex].index2 = tempVtx.index;

            else
                if ( interVector[tempIndex].origin1 == ps->ilist[0].v )
                    interVector[tempIndex].index1 = tempVtx.index;
                else
                    if ( interVector[tempIndex].v == ps->ilist[0].v )
                        interVector[tempIndex].index1 = tempVtx.index;

        }
        // setting up the self-index of each interseciton object
        for ( int i =0 ; i < interVector.size() ; i ++ )
            interVector[i].selfIndex = i;
}


// Snap all vertex coordinates to given resolution
void MultiPoly::gridify(Coord resolution)
{
    if (resolution != 0)
        for (PolyIt pi = m_polyList.begin(); pi != m_polyList.end(); ++pi)
            for (VtxIt vi = pi->vtxList.begin(); vi != pi->vtxList.end(); ++vi)
                vi->snap(resolution);
}


// traverse from the left most vertex of the ivList and form the queue of subsequent
// polygons done until the queue becomes empty i.e all the vertices of the
// multipolygon has been processed
void  MultiPoly::polygonPartition(MultiPoly &resMPoly, vector<Intersection> &tempList,
                                  vector<int> &windingVector)
{
    // resulting polygon
    vector<Intersection> interVector;
    PolyIt pIt;
    VtxIt vit;
    bool finished = false;

    Vertex startVtx, currVtx;
    Intersection prevInter,temp;
    interVector = tempList;
    int i =0, currWinding = 0, startIndex = 0, firstIndex = 0, secondIndex = 0;
    bool foundWinding  = false;
    int currEO =0;
    stack<Intersection> interStack;
    MultiPoly resMP;
    vector<int> wVector;
    int currDirection =0;
    do
    {
        int index =0, currIndex;// index of interVector
        Poly currPoly;

        if ( !interStack.empty() )
        {
            while (  !interStack.empty() )
            {
                index = interStack.top().selfIndex;
```

```
            temp = interStack.top();
            if ( interVector[index].index1 == -1 && interVector[index].index2 == -1)
            {
                interStack.pop();
            }
            else
                break;
        }
    }
    if ( !interStack.empty() &&  interStack.top().selfIndex > 0)
        index= interStack.top().selfIndex;
    else
    {
        index=0;
        while(( interVector[index].index1 == -1)
                && ( interVector[index].index2 == -1 ) &&(index < interVector.size()) )
        {
            temp = interVector[index];
            index++;
        }
    }
    if ( index >= interVector.size()-1 )
        finished = true;
    else
        currIndex = index;

    if ( !finished )
    {
        startVtx = interVector[currIndex].v;
        currVtx = interVector[currIndex].v;
        prevInter = interVector[currIndex];
        startIndex = currIndex;
        firstIndex = startIndex;
        Vertex vtx;

        // fix vertex to find the winding number before changing the index values!!

        if ( interVector[startIndex].index1 == -1 )
        {
            if ( interVector[startIndex].l1.m_swappedVertices)
                vtx = interVector[startIndex].l1.vertex1;
            else
                vtx = interVector[startIndex].l1.vertex0;
        }
        else
        {
            if ( interVector[startIndex].l2.m_swappedVertices)
                vtx = interVector[startIndex].l2.vertex1;
            else
                vtx = interVector[startIndex].l2.vertex0;
        }
        if ( interVector[startIndex].index1 == -1 || interVector[startIndex].index2
             == -1 )

            currWinding = interVector[startIndex].winding;

        currPoly.vtxList.push_back( currVtx);
        interStack.push(interVector[currIndex]);
        do {
            if (( interVector[currIndex].index1 == -1)
                && ( interVector[currIndex].index2 == -1 )  )
                currIndex++;
            else
            {
                if ( ( interVector[currIndex].origin1 == currVtx )
                     && ( interVector[currIndex].index2 == -1 )  )
                {
                    int tempIndex = interVector[currIndex].index1;
                    prevInter = interVector[currIndex];
```

```
                    interVector[currIndex].index1 = -1;
                    currIndex = tempIndex;
                }
                else
                {
                    if ( (( interVector[currIndex].origin1 == prevInter.origin1)
                            ||( interVector[currIndex].origin1 == prevInter.origin2 ))
                            && ( interVector[currIndex].index2 != -1 ) )
                    {
                        int tempIndex = interVector[currIndex].index2;
                        prevInter = interVector[currIndex];
                        interVector[currIndex].index2 = -1;
                        currIndex = tempIndex;
                    }
                    else
                    {
                        int tempIndex = interVector[currIndex].index1;
                        prevInter = interVector[currIndex];
                        interVector[currIndex].index1 = -1;
                        currIndex = tempIndex;
                    }
                }
            }
            if  (currIndex > -1)
            {
                currVtx = interVector[currIndex].v;
                currPoly.vtxList.push_back( currVtx);
                interStack.push(interVector[currIndex]);
                if ( !foundWinding)
                {
                    if (firstIndex == startIndex)
                        firstIndex = currIndex;
                    else
                    {
                        secondIndex = currIndex;
                        foundWinding = true;
                        Intersection temp1,temp2;
                        temp1 = interVector[firstIndex];
                        temp2 = interVector[secondIndex];
                        if (( interVector[startIndex].v.x
                             == interVector[startIndex].origin1.x)
                             && ( interVector[startIndex].v.y
                             == interVector[startIndex].origin1.y )
                             || ( interVector[startIndex].v.x
                             == interVector[startIndex].origin2.x)
                             && ( interVector[startIndex].v.y
                             == interVector[startIndex].origin2.y ))
                        {
                            float win1;
                            win1 = xProd(interVector[startIndex].l2.vertex0,
                                interVector[startIndex].v, interVector[firstIndex].v);
                            if (win1 > 0 )
                            {
                                currDirection = 0;
                                currWinding--;
                            }
                            else
                            {
                                currDirection = 1;
                                currWinding++;
                            }
                            interVector[startIndex].direction = currDirection;
                        }
                        else
                        {
                            float win;
                            win = xProd(vtx, interVector[startIndex].v,
                                interVector[firstIndex].v);
                            if (win > 0 )
```

```
                                {
                                    currDirection = 0;
                                    if ( interVector[startIndex].direction != currDirection)
                                        currWinding= currWinding - 2;
                                    else
                                        currWinding--;
                                }
                                else
                                {
                                    currDirection = 1;
                                    if ( interVector[startIndex].direction != currDirection)
                                        currWinding= currWinding + 2;
                                    else
                                        currWinding++;
                                }
                            }
                            interVector[startIndex].direction  = currDirection;
                            interVector[firstIndex].direction  = currDirection;
                            interVector[secondIndex].direction = currDirection;
                            interVector[startIndex].winding    = currWinding;
                            interVector[firstIndex].winding    = currWinding;
                            interVector[secondIndex].winding   = currWinding;
                        }
                    }
                    else
                    {
                        interVector[currIndex].winding = currWinding;
                        interVector[currIndex].direction = currDirection;
                    }
                }
            } while ( (currVtx != startVtx) && ( currIndex > -1 ) ) ;
            wVector.push_back(currWinding);
            foundWinding = false;
            resMP.m_polyList.push_back(currPoly);
        } ;
    } while ( !finished );
    resMPoly = resMP;
    windingVector = wVector;
}
```

```
// SLA.h
#include "MultiPolygon.h"
#include <fstream>

using namespace std;
////////////////////////////////////////////////////////////////////////////////
// Global Edge Table entry
// Constructors
// GET()
// GET ( const LineSegment &l, Coord y1, Coord y0, Coord x0, Coord m )
// GET( const GET &g): l1(g.l1), ymin(g.ymin), ymax(g.ymax),
//      xOfymin(g.xOfymin),slope(g.slope)
// Assignment
//   void operator= ( const GET &g)
// Relational
//   bool operator< (const GET &g) const
//   bool operator<= (const GET &g) const

class GET
{
public:
    LineSegment l1;

public:
    Coord ymax, ymin, xOfymin, slope;

    GET() {}

    GET ( const LineSegment &l, Coord y1, Coord y0, Coord x0, Coord m )
    {
        l1 = l;
        ymin = y0;
        ymax = y1;
        slope = m;
        xOfymin = x0;
    }

    GET( const GET &g): l1(g.l1), ymin(g.ymin), ymax(g.ymax),
        xOfymin(g.xOfymin),slope(g.slope) {}

    bool operator< (const GET &g) const
    { return (ymin < g.ymin ); }

    bool operator<= (const GET &g) const
    { return (ymin <= g.ymin ); }

    void operator= ( const GET &g)
    {
        l1 = g.l1;
        ymin = g.ymin;
        ymax = g.ymax;
        xOfymin = g.xOfymin;
        slope = g.slope;
    }
};

////////////////////////////////////////////////////////////////////////////////
// Active Edge Table Entry
// Constructors
//   AETEntry ()
//   AETEntry (const LineSegment &l, float y, float x, float m_inv )
//   AETEntry (const AETEntry &a)
// Assignment
//   bool operator= ( AETEntry *aet)
// Relational
//   bool operator<= (const AETEntry &aet) const
//   bool operator< (const AETEntry &aet) const
```

```
class AETEntry
{
public: LineSegment edge;
        float ymax, xval,m_inverse;

        AETEntry () {}

        AETEntry ( const LineSegment &l, float y, float x, float m_inv )
        {
            edge = l;
            ymax = y;
            xval = x;
            m_inverse = m_inv;
        }

        AETEntry ( const AETEntry &a )
            : edge (a.edge ), ymax(a.ymax), xval(a.xval), m_inverse(a.m_inverse){}
        bool operator<= (const AETEntry &aet) const
        {
            return (xval <= aet.xval ) ;
        }

        bool operator= ( AETEntry *aet)
        {
            edge = aet->edge;
            ymax = aet->ymax;
            xval = aet->xval;
            m_inverse = aet->m_inverse;
        }

        bool operator< (const AETEntry &aet) const
        { return (xval < aet.xval ); }
};

typedef vector<LineSegment> edges;
typedef vector<GET> GETable;
typedef vector<GET>::iterator get_it;
typedef vector<AETEntry> AET;
typedef vector<AETEntry>::iterator aet_it;

////////////////////////////////////////////////////////////////////////////////
// SLAM - creates an SLA objetc for a multi polygon
// contains the Global Edge Table and Active Edge Table
// Constructors
// SLAM ()
// SLAM ( MultiPoly &m )
// SLAM ( const GETable &get, AET &aet )
class SLAM
{
public:
    MultiPoly mp;
    GETable polyGET;
    AET polyAET;

    SLAM () {}

    SLAM ( MultiPoly &m )
    { mp = m; }

    SLAM ( const GETable &get, AET &aet )
    { polyGET = get; polyAET = aet; }

    void DrawSpan (float &y, AETEntry *p1, AETEntry *p2, CDC &pDC, bool isEO,
        int &winding, bool even) ;
    void FillPoly (MultiPoly *mpoly, CDC &pDC, bool isEO );
    vector <LineSegment> GetEdges( MultiPoly *m_poly , float &ymax);
    void InitGET(GETable &get, vector<LineSegment> &edgesVector , float y);
    void UpdateAET(AET &aet, float y, GETable &get);
    void UpdateAETEntry(AETEntry &g);
```

```
    void UpdateGET( GETable &get, float &y);

    float xProd(const Vertex &p, Vertex &q, Vertex &r)
    { return ( q.x * ( r.y - p.y ) *( q.y - r.y ) + r.x * ( p.y - q.y ) ); }
};
```

```
// SLA.cpp
#include "SLA.h"
#include <iostream>
using namespace std;
/* draws a span between two Active Edge Table entries of the polygon */
void SLAM::DrawSpan (float &y, AETEntry *p1, AETEntry *p2, CDC &pDC,
                     bool isEO,int &winding, bool even)
{
    float  x1, x2;
    /* don't draw spans with exactly one integer point */
    if (p1 == p2 )
        return;
    if ( p1->xval >  p2->xval )
    {
        x1 = p2->xval;
        x2 = p1->xval;
    }
    else
    {
        x1 = p1->xval;
        x2 = p2->xval;
    }
    /* draw pixels */
    if (isEO  && even)
    {
        if (( x1 > 0 ) && ( x2 > 0 ) && (x1 <= max(p1->xval,p2->xval) )
            && ( x2 <= max(p1->xval,p2->xval ) ))
            for (float x = x1; x <= x2; x++)
                pDC.SetPixelV( (int)x,(int)y, RGB(213, 175,255 ));
    }
    else
        if ( !isEO )
        {
            // find the direction of the segment
            if ( p1->edge.vertex0 > p1->edge.vertex1 )
                winding++;
            else
                winding--;
            if ( winding != 0 )
                if (( x1 > 0 ) && ( x2 > 0 ) && (x1 <= max(p1->xval,p2->xval) )
                    && ( x2 <= max(p1->xval,p2->xval ) ))
                    for (float x = x1; x <= x2; x++)
                        pDC.SetPixelV((int) x,(int)y, RGB(213, 175,255 ));
            if ( p2->edge.vertex0 > p2->edge.vertex1 )
                winding++;
            else
                winding--;
        }
}

// fill an arbitrary polygon with the even-odd interior rule
void SLAM :: FillPoly (MultiPoly *mpoly, CDC &pDC, bool isEO)
{
    float y = 0;
    bool  even = true;
    GETable get;
    AET aet;
    float ymax = 0;
    int winding ;
    edges eVector;
    //  ScanBuffer myScanBuff;
    aet_it p1,p2,p;
    eVector = GetEdges(mpoly, ymax);
    //initialise scanbuffer;
    /* create Global Edge Table, initialize y and AET */
    InitGET (get, eVector,y);
    y =get.begin()->ymin;
    /* sweep the scanline */
    while (y < ymax)
```

```
    {
        /* insert / delete edges in Active Edge Table */
        UpdateAET (aet, y, get);
        // insert or delete edges from get
        if (!get.empty())
            UpdateGET(get,y);
        /* sort Active Edge Table by increasing x */
        sort (aet.begin(), aet.end());
        if ( !aet.empty() )
            if ( aet.begin()->edge.vertex0 > aet.begin()->edge.vertex1 )
                winding =1;
            else
                winding = -1;
        /* draw spans with even-odd rule */
        even = true;
        for ( p1 = aet.begin() ;p1 != aet.end(); p1++) {
            p2 = p1;
            p2++;
            if ( p2 != aet.end() )
            {
                //  DrawSpan (y, p1, p2, pDC,isEO, winding,even );
                DrawSpan (y, (AETEntry *)&(*p1),
                    (AETEntry *)&(*p2), pDC,isEO,winding,even );
            }
            even = !even;
        }
        /* compute intersections for next scanline */
        y++;
        for (p = aet.begin(); p != aet.end(); p++ )
            UpdateAETEntry (*p);
    }
}
// Convert the given multi polygon into a vector of polygon edges
// and also keeps track of the ymax of the given polygon
vector<LineSegment> SLAM::GetEdges( MultiPoly *m_poly, float &ymax )
{
    vector<LineSegment> edges1;
    PolyIt pi = m_poly->m_polyList.begin();
    // for every polygon
    PolyIt pi0 = pi;
    // get the polygon vertex list
    VtxIt vi0 = pi0->vtxList.begin(), vi1;
    unsigned nVtx = pi0->vtxList.size();
    for (unsigned i = 0; i < nVtx; ++i)
    {
        vi1 = vi0;
        pi0->nextCirc(vi1);
        Vertex v0(*vi0),v1(*vi1);
        // set ymax
        if ( ( vi0->y > ymax )&&(vi0->y > vi1->y )   )
            ymax = vi0->y;
        else
            if ( vi1->y > ymax )
                ymax = vi1->y;
        LineSegment edge(*vi0, *vi1);
        edges1.push_back(edge);
        vi0 = vi1;
    }
    // get the next polygon
    ++pi0;
    while ( pi0 != m_poly->m_polyList.end() )
    {
        VtxIt vi2 = pi0->vtxList.begin(), vi3;
        unsigned nVtx = pi0->vtxList.size();
        for (unsigned i = 0; i < nVtx; ++i)
        {
            vi3 = vi2;
            pi0->nextCirc(vi3);
            Vertex v0(*vi2),v1(*vi3);
```

```
            if ( (vi2->y > vi3->y) && ( vi2->y > ymax ) )
                ymax = vi2->y;
            else
                if ( vi3->y > ymax )
                    ymax = vi3->y;
            LineSegment edge(*vi2, *vi3);
            edges1.push_back(edge);
            vi2 = vi3;
        }// for i
        ++pi0;
    }// while
    return ( edges1);
}

// Intialises the Global Edge Table with the edges corresponding to the scan line y
void SLAM::InitGET(GETable &get , vector< LineSegment> &edgesVector, float y){
    GET tempGET;
    LineSegment e;
    float yinf, ymin, x;
    yinf = 1024;
    for ( unsigned int i =0; i < edgesVector.size(); i ++ ){
        e = edgesVector[i];
        ymin = min (e.vertex0.y, e.vertex1.y);
        if ( ymin == e.vertex0.y )
            x = e.vertex0.x;
        else
            x = e.vertex1.x;
        if (ymin < yinf)
            yinf = ymin;
        if (yinf < 0)
            yinf = 0;
        y = yinf;
        tempGET.l1 = e;
        tempGET.slope = (e.vertex0.y - e.vertex1.y ) / ( e.vertex0.x  - e.vertex1.x );
        tempGET.xOfymin = x;
        tempGET.ymax = max ( e.vertex0.y, e.vertex1.y);
        tempGET.ymin = ymin;
        if ( tempGET.slope != 0 )
            get.push_back(tempGET );
    }
    sort ( get.begin(), get.end());
}

// Updates the AEt entry with new edges depending on the scan line y
void SLAM::UpdateAET(AET &aet, float y, GETable &get ){
    AETEntry tempAET;
    aet_it ae = aet.begin();
    int size = aet.size();
    if ( !aet.empty())
        for (int i = 0; i < size;i++ )
        {
            if( ae->ymax  < y+1 )
                aet.erase(ae);
            else
                ae++;
        }
        get_it ge= get.begin();
        for (  unsigned int j=0; j < get.size(); j ++ )
        {
            if ( ge->ymin == y )
            {
                tempAET.edge = ge->l1;
                tempAET.xval = ge->xOfymin;
                tempAET.ymax = ge->ymax;
                tempAET.m_inverse = 1/ge->slope;
                aet.push_back ( tempAET );
            }
            ge++;
        }
```

```
}

// updates the x-val in the AETEntry with scan conversion
void SLAM::UpdateAETEntry( AETEntry &g ){
    if (( g.xval > max( g.edge.vertex0.x,g.edge.vertex1.x)) ||( g.xval < min(
g.edge.vertex0.x,g.edge.vertex1.x) ))
        return;
    g.xval = (g.xval + g.m_inverse) ;
}

// updates with Global Edge Table
// removes the edges with ymin less than the scan line
void SLAM::UpdateGET( GETable &get, float &y){
    get_it ge = get.begin();
    int size = get.size();
    for (int i = 0; i < size;i++ )
    {
        if( ge->ymin  <= y )
            get.erase(ge);
        else
            ge++;
    }
}
```

VITA

VITA

Lavanya Subramaniam was born in Chennai, India, on November 7,1978. She graduated from Anna University, Chennai, India, in first class with a Bachelor's of Engineering in Computer Science and Engineering in 1999. A graduate assistantship was awarded to Lavanya in Spring 2000 at the University of South Alabama. She is a member of the Computer Science Honors Society Upsilon Pi Epsilon.