POLYGON REDUCTION ALGORITHM FOR LAYOUT EXTRACTION AND PARASITIC SIGNAL CALCULATION

THESIS

Presented to the Graduate Council of
Southwest Texas State University
in Partial Fulfillment of
the Requirements

For the Degree

Master of COMPUTER SCIENCE

By

Olga V. Zaporojets, M.S.

San Marcos, Texas
May 2002

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Carol Hazlewood for her continued guidance, encouragement, and support during the preparation of this thesis.

Words cannot truly express my deepest gratitude and appreciation to all Southwest Texas University professors for the classes that I had a chance to enjoy.

I want to express my gratitude to my family who always gave me their love and emotional support and Poveda family for watching my little son Nicky while I attended classes at SWT.

This thesis paper was resented to the Graduate Council of Southwest Texas State University on April 23, 2002.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

ABSTRACT

POLYGON REDUCTION ALGORITHM
FOR LAYOUT EXTRACTION AND PARASITIC SIGNAL CALCULATION

The design of a specific polygon reduction algorithm to be applied to the process of layout extraction is proposed. The algorithm is tested with the Star-RCXT parasitic signal extraction tool and the impact of the reduction algorithm is measured and analyzed. There are many polygon reduction algorithms but they all are application specific and not particularly suitable for the extraction tool. The research objective is to generate a reduction algorithm which may be used by any Electric Design Automation (EDA) layout extraction tool and which most effectively achieves a goal of maximum reduction of the number of extracted polygons and preserves connectivity and area of the polygons for further processing. The reduction algorithm described in this paper is created as a local greedy method with some history back tracking for the combined error control. It is used with trapezoidation processing, which is the most common way to perform polygon extraction by EDA tools. The reduction method developed is very general yet effective since it allows not only reduction of "not carrying information" trapezoids but trapezoids with a slight topology modification, fixing trapezoids displacements which may result from the lack of precision of a layout tool used to generate data. The trapezoidation and reduction algorithm was implemented in C++ and tested on real design data. The worst time reduction complexity is $O(n^2)$ and the worst time complexity of the trapezoidation

and reduction performed together is $O(n^2 \log_2 n)$. The reduction of the number of polygons achieved in the most accurate mode is 73% and it may be much higher if the less accurate mode is used. The accuracy tests displayed that extracted data produced using reduction algorithm is identical to the extraction data produced without reduction. Total reduction of the parasitic signal extraction time of the designs in the most accurate reduction mode is approximately 26%, which is very significant for the tape out time constraints in the competitive high technology industry. Implemented as a "stand alone" tool, the reduction algorithm may be used not only with Star_RCXT tool created by Avant!, but with any other extraction tool.

by

OLGA.V. ZAPOROJETS, M.S.

Southwest Texas State University

May 2002

SUPERVISING PROFESSOR: CAROL HAZLEWOOD

CHAPTER I

INTRODUCTION

1.1 Motivation

There are many practical applications where the reduction of polygons is an essential part of successful performance. Automatic polygon reduction is widely used in computer graphics, CAD and other related fields. The idea is to start with a geometric description of an object and to produce a new description, which preserves the properties of the object but has fewer geometric primitives. Many approaches to simplification have been proposed recently [1, 2], at least in part because this technique has the potential to dramatically speed up interactive graphics applications. Applications are confronted with over-sampled surfaces or models too complex for limited hardware capacity. An effective and fast algorithm for dynamically producing accurate approximations of the original model is a valuable tool for managing data complexity. While there is a large number of recent publications on automatic model simplification which are targeting computer graphics applications, especially computer games, the purpose of this work is to create a polygon reduction algorithm which is targeted for Electronic Design Automation (EDA) tools specifically.

1.2 EDA tools overview

EDA tools are used for electrical circuit design, schematic capture, simulation, prototyping, production, verification and parasitic signal extraction. The underlying idea behind EDA tools is to model and create the actual electrical circuit with the properties described in the schematic netlist. The Integrated Circuit (IC) design is advanced to millions of transistors on one chip, which would be impossible to achieve without sophisticated EDA tools. These are the main steps which are performed during a design cycle: design entry, functional verification, physical design, physical verification and parasitic extraction, synthesis, testing. Design entry encapsulates a circuit description. Functional verification confirms that the functionality of a model of a circuit conforms to the intended/specified behavior, by simulation or by formal methods. Physical design creates the mask set of a custom IC, plans, partitions, places and routes a cell-based Application Specific Integrated Circuit (ASIC), maps a netlist into an Field Programmable Gate Array (FPGA), or places and routes parts on a PC board. Synthesis tools translate abstract descriptions of functionality into optimal physical realizations, creating netlist that can be passed to a place-and-route tool. Physical verification ensures that masks sets conform to design rules, and parasitic extraction checks if the intended functionality is not compromised by layout-induced parasitic signals, cross talk, and other effects. Tools for testing generate test patterns to verify that a fabricated IC is free of process-induced faults. Other tools modify the

design to incorporate additional hardware supporting built-in self-testing of the chip, or testing by an external source.

The actual transistors, diodes, resistors, and metal routings are viewed by the tools as a collection of polygons with specific properties. Numerous verifications, optimizations and calculations are performed on this polygon database. Naturally, use of the polygon reduction algorithm will enhance the performance of EDA tools.

The tool that incorporates our algorithm for testing extraction tool Star-RCXT developed at Avant!. This tool analyses several aspects of the design including evaluation of parasitic resistances and capacitance, electromigration, power analysis, and clock analysis. The xTractor engine is the main core of the tool. It is called a swap-scan engine since it reads into the database all routing levels simultaneously. Only polygons necessary for the calculation of the parasitic signal are kept in memory and they are discarded as soon as the area of their influence is exited. This is a very advanced software product and one of the leading extraction tools in the industry. The importance of the accurate parasitic signal extraction grows as the industry advances into submicron technologies, since smaller sizes lead to more significant parasitic signals. Increase of the average number of the transistors in the design leads to huge databases, which consume gigabytes of memory and take days to process. This is why we believe that our polygon reduction algorithm may significantly improve performance of the tool by reducing the number of polygons to be processed.

1.3 Restrictions applied to polygon reduction algorithm

While existing polygon reduction algorithms for graphic applications have the goal to generate a model that looks reasonably similar to the original, my polygon reduction algorithm has to satisfy other specific requirements, i.e. *to preserve the connectivity and the area of the polygons*. The reasons for these restrictions are:

- Connectivity of the polygons represents electrical connectivity of the circuit. By losing original connectivity we may cause an electrical open circuit and by letting newly generated polygons to touch/overlap we may create an electrical short circuit, which will create a design error.
- The area of the polygons is used for the calculation of the resistance (R), capacitance (C), current, and power drop. By modifying the area we will change the circuit parameters, which may lead to serious design errors.

Analogous to all existing polygon reduction algorithms we have to deal with the error introduced by the reduction procedure. The most accurate algorithms are generally very slow, since very complex calculations are involved. The fastest algorithms often produce poor quality results. My goal is to find an effective compromise and to create a fast but accurate algorithm for polygon reduction satisfying restrictions of EDA tools.

1.4 Our approach to solving the problem

I use edge contraction as the method for the simplifying geometry, as have a number of previous researches [12]. This approach selects an edge and replaces it with a single vertex. The edge contraction operation is attractive because it allows the new vertex to be placed in a manner that helps to preserve the location and the area of original structure. Three decisions are central to a simplification method that uses edge contraction:

- Which edge may be collapsed or which vertex can be reduced?
- What is the position of the new vertex created by the edge collapse or how is a vertex reduced?
- How is the error calculated and evaluated? What criteria should be used to terminate polygon reduction?

This is how we approach these questions:

Which edge may be collapsed or which vertex can be reduced?

First of all we examine the candidates for reduction polygons and decide if they may be collapsed into one. The guidelines for this decisions are: if it is possible to reduce the number of polygons without changing the topology of the data, then perform the reduction; if the reduction is possible with a topology change which meets the error threshold and requirements of preserving connectivity and area, then perform the

reduction. The analysis of possible neighboring polygon combinations in the layout design database will be described later in this paper. The algorithm for examining reduction candidates will be defined.

What is the position of the new vertex created by the edge collapse?

The (x, y) coordinates in 2D space characterize every vertex. If the reduction is performed along the y axis, intuitively the new vertex may be placed at one of the original vertices. To determine the vertex, we evaluate the areas of newly generated polygons versus the previous combined area. The new vertex position must be selected so the area is changed as little as possible and satisfies the error threshold restriction. An error threshold is a limit, set by the user, which indicates when reduction should terminate. If accumulated error exceeds the error threshold, then reduction should stop. The guidelines for a new vertex position will be defined for all possible topologies qualified for reduction later in this paper.

When to stop polygon reduction or how to calculate and evaluate the error produced by the reduction?

The edge collapse algorithm used for the implementation of the described in this paper reduction algorithm is a local method. It does not "look ahead" and the accumulation of the error is possible as a result of the series of the consecutive reductions. As a result, the error restriction may be violated if topology of the data

will be modified at several reduction events. To prevent such situation the history of reduction must be kept and the evaluated at

every reduction event. The error threshold shall be controlled by the user, which may set it as an input parameter. The error threshold value shall control the reduction level.

There is a built in default value for the reduction error threshold, which shall be tested on the actual data and guarantee to yield the desired accuracy.

1. 5 Reduction algorithm design and implementation

First of all available reduction algorithms were reviewed and the most appropriate type of reduction methodology was selected. The overview of the previous work is given in the chapter 1 and the analysis of the modern polygon reduction algorithms is given in the chapter 2.

Following methodology decisions were made: decimation was preferred to refinement; local method was preferred to global method; greedy approach with history tracking for the error control was chosen.

Next extraction methodology was defined. Chapter 4 describes the nature of the parasitic extraction; goals and constraints applied to the reduction algorithm developed for the design layout extraction application. The trapezoidation was chosen

as a decomposition method for the polygon extraction.

In order to create a non-proprietary stand alone polygon extraction and reduction tool, a trapezoidation method was implemented. The trapezoidal decomposition and its implementation are described in chapter 5. After review of publicly available trapezoidation algorithms, a trapezoidation algorithm proposed by Borut Zalik was chosen for it efficiency, simplicity, and capability of processing non-monotone polygons with holes. This algorithm was implemented in this research in C++. This implementation displayed exceptional stability.

The "break through" of the created reduction algorithm is the ability not only reduce "not carrying information" trapezoids but trapezoids with a slight topology modification, fixing trapezoids displacements which may result from the lack of precision of a layout tool used to generate data. Three possible types of the trapezoidal displacement and algorithms for their reduction are described in the chapter 6.

The error control mechanism developed and implemented in this research and reduction algorithm complexity analysis is described in the chapter 6. The worst time reduction complexity is $O(n^2)$ and the worst time complexity of the trapezoidation and reduction performed together is $O(n^2 \log_2 n)$.

Tests and reduction algorithm performance is described in the chapter 7. The reduction of the number of polygons achieved in the most accurate mode is 73% and it may be much higher if the less accurate mode is used. The accuracy tests displayed

that extracted data produced using reduction algorithm is identical to the extraction data produced without reduction. Total reduction of the parasitic signal extraction time of the designs in the most accurate reduction mode is approximately 26%. The examples of the test cases used for testing reduction algorithm are given in the appendix. The work is summarized in the chapter 8. The parasitic signal extraction is a significant part of the integrated circuit design cycle. By reducing time necessary for extraction, a design gets to the market weeks faster, which is very significant advantage in a modern integrated circuit design industry.

CHAPTER II

OVERVIEW OF THE PREVIOUS WORK

2.1 Research history. Curves simplification.

Long before computer applications demanded model simplifications, the problems of curves and height fields simplifications were discussed. Since they are special cases of general surfaces they served as a base for solving general surface simplification problems. Approximation of curves satisfying an equation $y = f(x)$ has the longest history. The simplification of the piecewise-linear curves relates closely to the problem of polygonal reduction of the surfaces. Consider a curve implemented with n vertices. The goal is to obtain the approximation of this curve using k vertices, where $k < n$. Imai and Iri [16] proposed an optimal approximation of the functions for achieving the goal: to use the minimum number of vertices necessary for satisfying the given error ε. Garland [18] describes of the approximation error as follows. The error may be described by L_∞ norm, which measures the deviation between original and approximation. Suppose the original function is f(x) and the approximation is g(x). L_∞ norm on interval of interest [a, b] is defined as following:

$$\|f - g\|_\infty = \max |f(x) - g(x)|, \text{ where } a <= x <= b \qquad (1)$$

Another commonly used error norm is L_2, defined as

$$\|f - g\|_2 = \sqrt{\int_a^b (f(x) - g(x))^2 \, dx} \qquad (2)$$

The piece-wise approximation g(x) composed of n segments is called an optimal approximation of f(x) if there is no other n-segment approximation function with smaller error. The choice of which norm should be used depends on a particular function. L_∞ norm is generally regarded as a stronger measure of error because it provides a global absolute bound on the distance between the original and the approximation and it is often easier to verify the quality it guarantees. L_2 is more general and in some cases it may be well defined where L_∞ is not. None of these metrics are ideal: L_∞ provides strong error bounds but it may be overly sensitive to noise and local deviations. L_2 is more tolerant to the noise, but it may discount local deviations. The best precision may be achieved by using combination of both metrics: the approximations where L_2 error is small and where L_∞ error is bounded by some known threshold are most desirable.

While the algorithms for finding optimal approximations of functions have time complexity O(n), the algorithms for plane curves have complexity $O(n^2 \log n)$ and space curves have complexity $O(n^3 \log n)$ which make them impractical for large data.

2.2 Greedy approach

The greedy strategy is powerful and widely applied. I would like to discuss how the popular method of greedy search shall be applied to the problem of polygon reduction.

First of all I will give an overview of the greedy method. Search strategies may be evaluated using the following parameters:

- *Optimality* – does it always find a least-cost solution?
- *Completeness* – does it always find a solution if one exists?
- *Time complexity* – number of nodes generated / expanded
- *Space complexity* – maximum number of nodes in memory

The greedy search idea is to expand the node that appears to be closest to the goal. The evaluation function h(n) = estimate of the cost from n to the goal. The greedy algorithm would look at the adjacent nodes and will choose the one that has the shortest distance to the goal.

The greedy algorithm *is not always optimal.* Greedy search algorithms may be easily mislead to search wrong paths. Can it be optimal? Yes it can, but performance depends on the heuristic, simply the way we are estimating the cost of the path through n to the goal. Another problem of the greedy search is that it often gets stuck in loops. So in general the basic greedy algorithm is not optimum and *not complete.* Optimization is obtainable only in the finite space with repeated-state checking. One

of the proposed solutions was to carry out the process several times, starting from different randomly generated configurations, and save the best result. So another general property of the greedy method is that *it always makes a locally optimal choice, though it doesn't always yield a globally optimal solution.* The *worst run time complexity* of a greedy algorithm is O (b^m), where b is a branching factor of the search tree and m is maximum depth of the search space. The *space complexity* of a greedy algorithm is O (b^m) as well. However, with a good heuristic it may yield dramatically better performance and this is why it is a popular solution for a wide variety of optimization problems. Greedy algorithms are rather simple; easy to implement and yield a very good run time. If the requirement of admissibility and monotonicity is satisfied, greedy algorithms will produce the optimal and compete solution. Greedy algorithms are especially suitable for problems where speed is more important than the absolute accuracy of the solution. This is why greedy algorithm had been used as a solution for the polygon reduction problem by Hoppe [12] and other researches. The persisting problem is: how to select a heuristic formula to evaluate possible solutions and choose the best one. The solution proposed by Hoppe based on greedy algorithm displays great speed and fair quality of the reduced modes. The approach I choose for my solution may be called "greedy", since only local properties of the polygon are evaluated at any given time. There is not any " look ahead" mechanism to confirm that the chosen decision is the best in the long term after several rounds of reduction are done. We believe that this approach is an

appropriate solution for the polygon reduction problem since it gives flexibility to choose how many reduction iterations we want to perform along with the simplicity of implementation and speed.

2.3 Classification of the modern reduction algorithms

2.3.1 Decimation versus refinement

The problems of automatic model simplification have received increasing attention in recent years. The underlying concept of model simplification is a function approximation and has been researched by mathematicians for many years. However the numerous practical polygon reduction algorithms appeared only during the last two decades. My polygon reduction algorithm falls into the category of *decimation* algorithms, which begin with the original surface and remove polygons at each step. Decimation algorithms are opposite to another group of popular simplification algorithm, called *refinement*, where the algorithm starts with a coarse model initially and adds elements at each step. An example of the refinement approach is adaptive subdivision algorithms. Data on the various simplification algorithms may be found in [1, 2]. Rather few algorithms for progressively refining polygonal surfaces have been proposed [14], [15]. While refinement was the traditional solution for curve approximation, decimation is widely used for simplifying general surfaces. I believe

the decimation approach is more suitable for surface simplification because it doesn't require construction of the base approximation while all refinement algorithms do. The base approximation is a difficult task because it must have a topology of the original model in order to use simple subdivision rules. The topology of input surface may be difficult to obtain and it leads to restrictions on the ability to simplify it.

2.3.2 Local versus global

Another way to categorize polygon reduction algorithms is to divide them into *local* or *global* techniques. Local techniques operate on individual primitives such as vertices, edges, triangles, or other polygon characteristics. The examples of local techniques are vertex decimation, edge collapse, polygon curvature evaluation, polygon area evaluation, and adaptive subdivision. Global technique optimizes the polygon mesh based upon high-level yet more general features of the model. The examples of global solutions are polygon re-tiling, mesh optimization, object replacement, and wavelet encoding. The brief overview of these methods is below.

2.3.3 3D versus 2D

Most of the model simplification algorithms are targeting 3D problems. Describe in this paper reduction algorithm is applied to 2D models. The information about 3rd

dimension of the layout for the xTractor is kept separately. The third dimension parameters along with the physical characteristics of the material used for the implementation of the IC are described in a process description database, unique for every foundry, which fabricates the chip in silicon. Therefore my problem may be viewed as an analogous to the problems discussed, only simplified to two dimensions.

2.3.4 Incremental algorithms without history tracking

Since it is impossible to cover all existing work [1, 2], I would like to focus on the algorithms that make small incremental changes to the geometry of a model which is not a case for the algorithms described in [3, 4]. One of the major benefits of iterative contraction is the hierarchical structure it creates. This leads to a useful multiresolution surface representation. The incremental simplification algorithms in turn may be divided into two groups: the algorithms which don't keep a history of the original geometry during simplification and the more recent algorithms which use some history about the original geometry as a way of tracking error during simplification. The example of "non history tracking" algorithms is an algorithm proposed by Schroeder [5]. He is performing successive vertex removal using the distance from a vertex to the plane most nearly passing through adjacent vertices as their priority measure. Renze and Oliver, who concentrate in their work on a triangulation algorithm [6], use the same distance to the plane method. Another

example of "non history tracking" algorithms is Hamman's algorithm, which uses a measure of the curvature to decide which vertices to remove [7]. Lindstrom and Turk [11] proposed a "memoryless" algorithm which makes decisions based on the current approximation alone. No information about original shape is retained. They use linear constraints based on the conservation of volume to decide which edge should be contracted and where remaining vertex should be placed. This algorithm generates good quality results along with low memory consumption.

2.3.5 Incremental algorithms using history for the error evaluation

An example of a "history tracking" algorithm is a more recent variant of the technique proposed by Schroeder [8]. He includes a scalar value at each vertex, which encodes the error created so far in the neighboring area of the vertex. The "negative" and "positive" error bounds are kept through the interactive reduction process by Bajaj and Schikore [9]. The error bounds are computed by projecting old and new triangles to a plane. Another way of calculating the error was proposed by Ciampalini and co-workers [1]. They associate a list of vertices from the original model with every triangle. These vertices allow calculating an error estimate for each triangle during the simplification process.

CHAPTER III

OVERVIEW OF THE MODERN POLYGON REDUCTION ALGORITHMS

3.1 Local methods

This chapter will discuss local reduction algorithms and give an overview of some local reduction methods available.

3.1.1 Vertex decimation

The underlying idea of the all iterative algorithms described above is vertex decimation. Figure 3.1 illustrates this technique, as it is proposed by Schroeder [5].

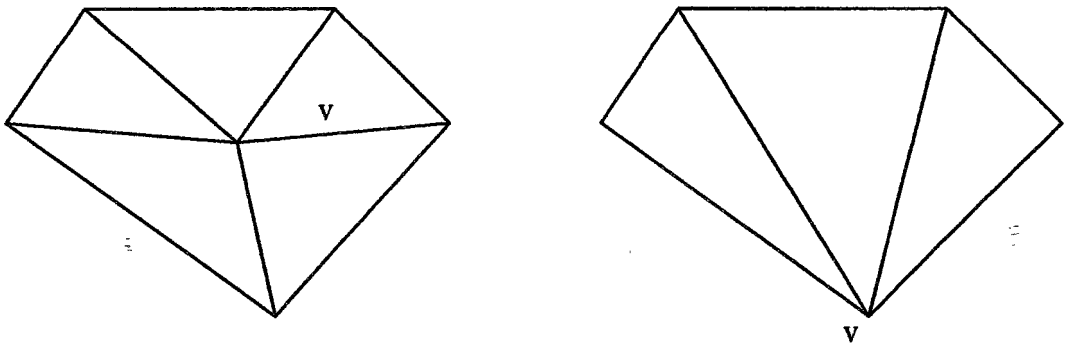


Figure 3.1 Vertex decimation

The following steps implement this operation:

1. All edges and triangles containing selected vertex (v) are removed.

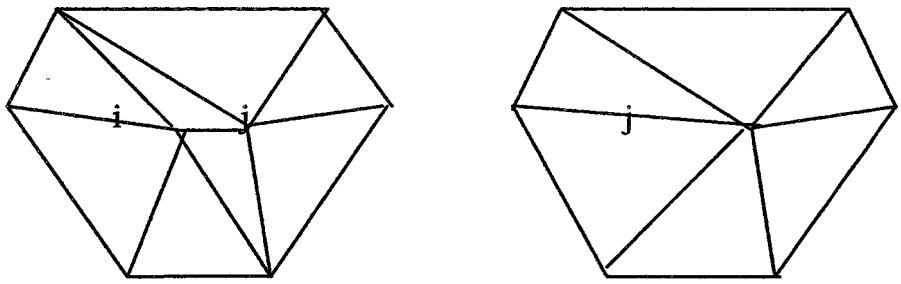
2. Vertex is removed (v)

3. Resulting hole is retriangulated

3.1.2 Edge contraction

While all the algorithms described above are based on vertex removal, my algorithm is based on edge contraction. H. Hoppe [12] appears to be the first to use edge contraction as the underlying mechanism for accomplishing the polygon reduction. Most of the better techniques proposed recently are variations of the progressive meshes algorithm proposed by H. Hoppe [10]. These techniques reduce the complexity of the model by iterative application of the edge contraction operation. This is the basic idea of the edge contraction.

Edge e_{ij} is connecting vertices i and j. One of the vertices is selected (i) and "collapsed" onto another (j)



Before After

Figure 3.2 Edge contraction

The following steps implement this operation:

1. Remove any triangles that have both i and j as vertices, e.g. remove triangles containing edge e_{ij}.

2. Update the remaining triangles that use i as a vertex to use j instead.

3. Remove vertex j.

These steps are repeated until the desired result is reached. At each step one vertex, two triangles and three edges are usually removed. Figure 3.2 describes this procedure.

The vertex decimation methodology is closely related to the edge contraction. As you may see the vertex v removal illustrated on the Figure 3.1 can be easily accomplished by contracting the bottom edge. Edge contraction is usually more robust then

projecting the neighborhood on the plane and retriangulating, since we don't have to worry about finding a plane onto which the neighborhood can be projected without overlap.

3.1.3 Polygon curvature evaluation

This group of algorithms is based on measuring the angles between connected edges to evaluate the curvature of the mesh at any given point. The goal is to reduce details around areas of low curvature and to retain details around high curvature regions. This approach is based on the assumption that the regions with high curvature contribute strongly to the shape of the object. The principal advantage of curvature based reductions is that they remove nearly planar surfaces and provide good shape constancy. Hamann [7] used this method in his reduction algorithm. His algorithm iteratively removes triangles based on the curvature values at their vertices from a mesh and re-triangulates local area.

3.1.4 Polygon area evaluation

The reduction criteria for this group of algorithms are polygon dimensions. The polygons with the small area may be removed or merged. The example of the use of this approach is Holloway's Viper system [19]. Viper improves system performance

by the terminating display of the object when the graphic system becomes overloaded. The polygons of the object are sorted by size and the largest polygons are displayed first, while the smallest may be successfully removed as the system becomes overloaded.

3.2 Global methods

Global reduction methods differ from local methods by keeping in memory the whole data base and making incremental reduction on the data as a whole, modifying multiple vectors simultaneously, unlike local methods. An overview of some global methods follows.

3.2.1 Mesh optimization

Complex triangle meshes arise naturally in many areas of computer graphics. Quadratic error metric allows fast and accurate geometric simplification of the meshes. I think that Hoppe more than anyone else researched this approach and produced a number of algorithms based on mesh optimization. The examples of his work are [10, 12]. An interesting technique was proposed by Hoppe in [23]. This technique introduced the concept of an energy function to model the opposing factors of polygon reduction and their similarity to the original topography. The energy

function was used to provide a measure of the deviance between the original mesh and the simplified version. This was then minimized to find an optimal distribution of vertices for any particular instantiation of the energy function. In another work Hoppe proposed a new metric, based on geometric correspondence in 3D which requires less storage, evaluates more quickly and results in more accurate simplified meshes [24]. In this work Hoppe shows that a wedge-based mesh data structure captures discontinuities efficiently and permits simultaneous optimization of the multiple attribute vectors. He also experiments with memoryless simplification and volume preservation and shows that they are beneficial within the quadratic framework. The mesh optimization technique successfully distributes vertices in relation to the surface curvature and provides a high degree of shape constancy between model approximations.

3.2.2 Polygon Re-tiling

The re-tiling technique is an example ofa global technique. It was proposed by Turk [3]. The idea of re-tiling is to optimize a polygon mesh by introducing new vertices to the mesh and then discarding the old vertices to form new representations. The initial surface is triangulated by the vertices. The new vertices are pseudo-randomly positioned in the planes of the existing polygons and then successfully repelled by their neighbors in order to create a uniform distribution. After that the old vertices are

removed one after another and the surface is locally re-tiled in order to retain the topology of the original surface.

3.2.3 Object replacement

Certain classes of objects may be approximated by object replacement. Sewell developed an algorithm to compute a simple replacement primitive for a complex grouping [20]. This approach can produce substantial complexity reductions because it can decompose clusters of objects, which other techniques would treat as separate entities. The underlying idea of this method is the theory that suggests that the human object recognition is based upon identifying a small number of primitive shapes within the object. I think that this is a very interesting approach but it may produce visual artifacts and is quite complex in implementation.

3.2.4 Wavelet encoding

Wavelet methods provide a clean mathematical framework for the decomposition of a surface into a base shape plus a sequence of successively finer surface details. Approximations can be generated by discarding the least significant details. Wavelets are a means of hierarchically decomposing a function so that it can be described as a coarse general form. Omitting a number of small detail terms called wavelet

coefficients when rebuilding a model may form the coarse approximation of the original object. Wavelet decomposition has been used successfully for producing multiresolution representations of signals and images [21, 22]. The downside of this approach is that the resulting approximations may be relatively far from optimal because they may have a large number of triangles simply to preserve the subdivision connectivity. Another problem is not being able to resolve creases on the surface unless they fall along the edges in the base mesh. Like other subdivision-based schemes, wavelet methods cannot construct approximations with a topology different from the original surface.

3.3 Precision versus speed

Hoppe's algorithm for progressive meshes construction [10] is based on the minimization of the energy function. The algorithm maintains a set of sample points on both the original surface and the approximation. The distances between these points and the closest point on the opposing surface determine the geometric error. This algorithm produces the highest quality results among currently available methods. The downside of this method is a very long running time.

There are a number of faster algorithms then Hoppe's, but quality of reduction is the price paid for increased speed. For example Ronfard and Rossignac [13] developed a

fairly efficient algorithm. Each vertex in the approximation has an associated set of planes. The error of every vertex is defined by the maximum of squared distances to the planes in this set. These sets are merged when the vertices are contracted together. This error metric is much cheaper than Hoppe's since calculation of distances to the planes is faster then measuring distances to triangles. The resulting approximation produces generally good quality and it is more efficient than more precise algorithms. The mesh optimization algorithm of Hoppe [12] and his coworkers was created before the progressive mesh construction algorithm [10]. It performs explicit search rather than simple greedy contraction. Possible approximations are found using edge contraction, edge split, and edge flip operations. This algorithm exhibits a longer running time but it produces the highest quality results.

CHAPTER IV

PARASITIC SIGNAL EXTRACTION

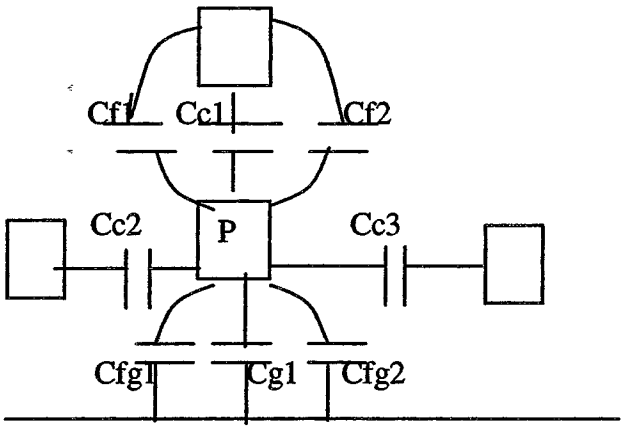
4.1 Nature of parasitic signals and importance of their evaluation

The polygon reduction algorithm discussed in this paper is created for the extraction engine, which reads information from the IC design layout and calculates the parasitic signal values.

Parasitic signal evaluation is necessary for the calculation of correct timing delays on all signal paths. Interconnect wires in a semiconductor chip provide paths for signals. Due to smaller and smaller feature sizes, the importance of interconnects in chip design is growing. Feature size reduction leads to relatively taller metal wires, since metal thickness cannot be scaled down at the same ratio as feature sizes are reduced. Otherwise, the sheet resistance would increase too much. The relatively taller metal wires lead to relatively larger line-to-line parasitic wiring capacitance. As a result, a larger percentage of chip performance is now due to the delay in interconnect wires, called the *interconnect delay*.

The calculation of wiring capacitance among interconnect wires is more complex than the calculation of the resistance. An accurate calculation of wiring capacitance usually requires the use of electromagnetic field simulation software, typically called a *field solver*. To satisfy the needs of circuit designers to perform VLSI circuit

simulations, a fast capacitance extraction formula is needed. The most common configuration for wiring capacitance is a 2D cross-section vertical layout, which consists of row of rectangles between two infinitely long conducting planes. In generations of older technology capacitance extraction formulas were based on the analytic capacitance expressions for a parallel-plate capacitor and for a cylinder above a plane, plus new features to handle the wires on the left and right as well as the second conducting plane on top of it. As semiconductor feature sizes continue to shrink, the shape of the conducting wire changes from the old shape of *fat-short* to the new shape of *thin-tall*, where fringing capacitance is very important and has to be calculated accurately, Figure 4.1.



Cg1 – capacitance of horizontal plane of polygon P to ground

Cc2 and Cc3 – coupling capacitance of P and interconnects of the same layer

Cc1 – coupling capacitance of P and polygon on the routing layer above

Cf1 and Cf2 – fringing capacitance of P and polygon on the routing layer above

Cfg1 and Cfg2 – fringing capacitance of P and ground

Figure 4.1 Parasitic capacitance among layout polygons

Total capacitance of P to ground is:

$$Cgt = \Sigma Cfg_i + \Sigma Cg_j \qquad (3)$$

Total coupling capacitance of P to other interconnects is:

$$Cgt = \Sigma Cf_i + \Sigma Cc_j \qquad (4)$$

The core part of the analytic capacitance expressions is derived based on conformal transformation in complex variable analysis.

4.2 Input data for calculation of parasitic signals and processing steps

Parasitic capacitance and resistance are functions of the *area* of the conducting polygons and the technology values: *permittivity* and *resistivity* of the materials and *height* of the polygons. Height of the polygons is the third dimension of the structures and the value of it is contained in *technology file* along with all other technology characteristics. Technology information, combined with the information about geometry of the polygons, allows performing the parasitic signal calculation. Information about polygon structures is actual design information and it may be represented in different formats. Examples of different formats are the Milkyway format created for Avant! EDA tools, Lef/Def files used by Cadence tools and many others. But the standard format, which is widely used in EDA industry, is GDSII format. Every EDA tool vender provides an ability to translate their internal format into GDSII or works directly with GDSII format. The process of translation of design in/out GDSII is called *streaming in/out*. GDSII format file contains only information

about polygon geometry and doesn't have any information about electrical connectivity. The connectivity information has to be provided from another source in order to create a meaningful netlist of the design.

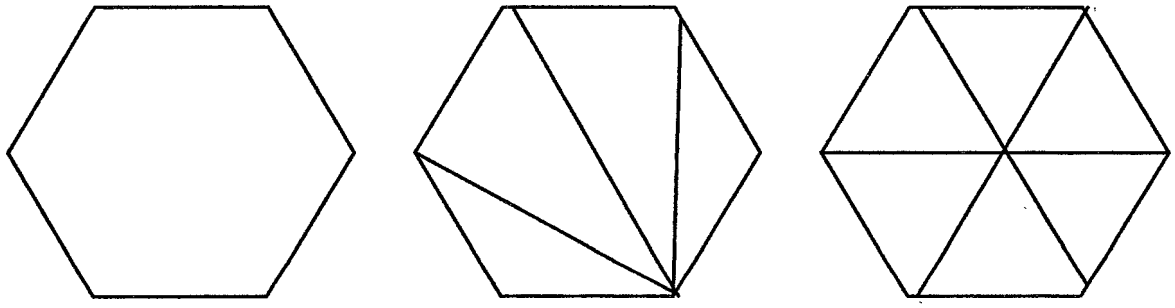
The first processing step is to read in GDSII format data and to process or simplify the database in order to be able to calculate resistance and capacitance. The Extractor, Avant! extraction engine, utilizes a new concept of simultaneous reading of all vertical layers in the design; it is called the *sweep scan band.* Average modern design may have 4 to 8 conducting layers that are read at the same time with the sweep scan band moving. The advantages of such an approach are:

- a decrease of processing time – no need to process one layer at a time;
- the ability to calculate parasitic more accurate accounting influence of all layers in the same time;
- more efficient memory use since we don't need to keep all of the data base in the cache. Because we need only structures within the scan band influencing each other, such that they may be disposed from the memory as soon as the area of their influence is left.

Since calculation of resistance and capacitance directly depends on the area, the most natural approach is to *decompose polygons* for easy and efficient area calculation.

4.3 Why trapezoids are chosen as the decomposition method

The idea of decomposing complex geometric structures into simple components is one of the main focal points of computational geometry and its applications. There is a large number of decomposition techniques that have been proposed. The most popular and widely investigated decomposition technique is triangulation. Triangulation is the breaking of planar or three-dimensional polygons into triangles. Two ways to triangulate a polygon are shown in Figure 4.2.



Original polygon Triangulated polygon Triangulated polygon

Figure 4.2 Examples of triangulation

However, the decomposition method chosen in this work is *trapezoidation.* A trapezoid is a four-sided polygon in which two of the edges are parallel. The triangle may be considered as a special case of a trapezoid, since a triangle has one of the parallel edges equal to zero length. The topology of layout has predominantly vertical and horizontal lines and since processing extracted layout data is an application for this algorithm, trapezoids appear to be a very natural choice for the decomposition shape. The trapezoids are chosen as an optimal component also because of the simplicity of processing trapezoids and the easy calculation of resistance and capacitance of the trapezoid shaped polygon. The area of trapezoid is calculated easily by the formula:

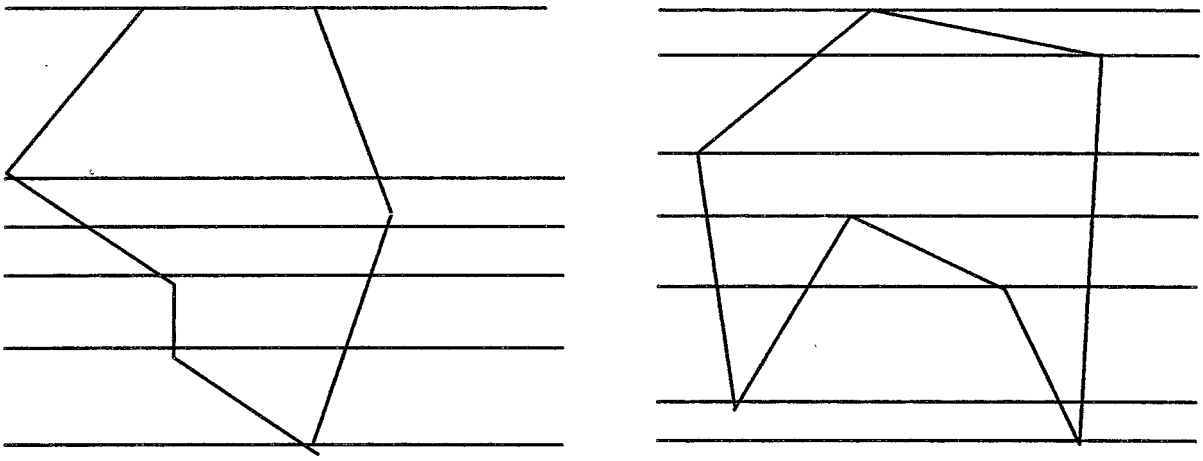
(base one + base two) / 2 * height (5)

CHAPTER V

TRAPEZOIDAL DECOMPOSITION

5.1 Methods and definitions used for trapezoidation

Chazelle and Incerpi proposed the first trapezoidation algorithm [26]. Some triangulation algorithms use trapezoidation as a first step [27]. Polygons may be classified as monotone and non-monotone polygons. A monotone polygon is a polygon whose boundary consists of two y-monotone chains. A set of ordered vertices is a y-monotone chain if, for every vertex in a chain, its y coordinate is greater then or equal to the y coordinate of the previous vertex *for y-monotone increasing chain,* and less then or equal to the y-coordinate of the previous vertex for *the y-monotone decreasing chain.* Examples of horizontal trapezoidation of monotone and non-monotone polygons are shown in Figure 5.1.



a) Monotone polygon b) Non-monotone polygon

Figure 5.1 Horizontal trapezoidation of monotone and non-monotone polygons.

A *horizontal trapezoidation* of a polygon is constructed by adding a horizontal line through every vertex of the polygon. The most popular technique to perform such task is called *plane sweep*. A horizontal line is swept vertically down the plane and the data structure is updated at discrete *events*. For a trapezoidation algorithm on event will be the intersection of the sweep line with one of the vertices.

5.1.1 Vertex classification

This is the list of relationships existing between polygon vertex **p** and its neighboring vertices $\mathbf{p_j}$ and $\mathbf{p_r}$, proposed by O'Rourke [28]:

Intersection (INT):

One of neighboring vertices has greater y-coordinate and the other a lower y-coordinate then **p** (Figure 5.2a).

Local minimum (MIN):

Both neighboring vertices have greater y-coordinates then **p** (Figure 5.2b).

Local maximum (MAX):

Both neighboring vertices have lower y-coordinates then **p** (Figure 5.2c).

Horizontal line segment (H-MAX, H-MIN):

One of the neighbors has the same y-coordinate as **p** within tolerance ε, then **p** is and end point of a horizontal line segment.

If second neighbor has lower y-coordinate then **p**, then **p** is called H-MAX (Figure 5.2d). If second neighbor has a higher y-coordinate then **p**, then **p** is called H-MIN (Figure 5.2e). If second neighbor has the same y-coordinate within ε tolerance, then **p** doesn't form part of any horizontal trapezoid and it can be erased from the list of polygon points at this stage (Figure 5.2f).

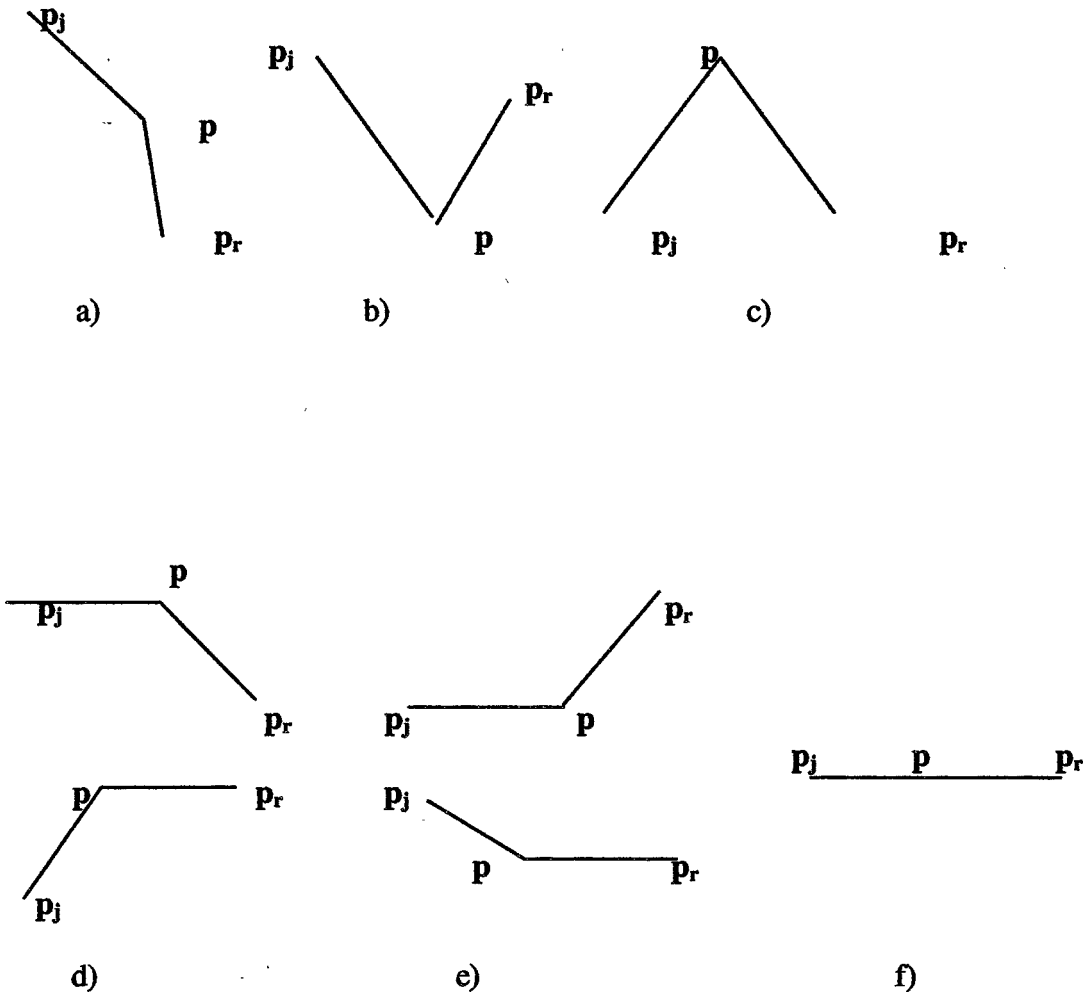


Figure 5.2 Vertex types.

Classification of the vertexes proposed by O'Rourke is used in this research for vertex type identification. The trapezoidation algorithm used in this research was proposed by Borut Zalik and Gordon Clapworthy [25]. We have chosen this algorithm because of its efficiency, simplicity and generality.

5.2 Overview of Borut Zalik's trapezoidation algorithm.

Borut Zalik proposed an algorithm allowing decomposition of non-monotone polygons which may contain holes into trapezoids. A hole is a polygon that has all vertices inside or laying on the edges of an outer polygon. The trapezoidation algorithm consists of two parts: data preparation and creation of the trapezoids. Using the vocabulary of Zalik, the sequence of vertices forming the border of a polygon are called a *loop,* and a sequence of the vertices forming the border of inner closed polygon is called a *ring*.

5.2.1 Processing of a non-monotone polygon without holes

First, the overview of the processing of a polygon without holes will be given. It consists of two steps.

Step 1:

All vertices are sorted by y-coordinate and stored in a dynamic ordered list.
A plane sweep is employed. Since the vertices are already sorted, only the first element in the list is examined. The vertex type is examined and the appropriate action is chosen. The vertices with the same y-coordinate are sorted by x-coordinate in increasing order. Vertices, which are relevant to trapezoidation between the previous scan line and this scan line are retained and others are discarded. If there are

n vertices in the polygon, there are at most n scan lines.

Step 2:

Identify points on current and previous scan lines that are to be joined to create trapezoids.

Implementation of the step 1

- Read in vertex coordinates and analyze type of vertices.
- Store vertex coordinates sequentially in an array named *InputPoints*.
- Sort vertices by y-coordinate and store them in an array *SortedVerticesY*, (Figure 7). Several vertexes with the same y are possible. The data structure is implemented in a way that in addition to vertex coordinates, two pointers are stored which point to neighboring vertices. A Flag *Used,* associated with the vertex, is set initially to FALSE and it is reset to TRUE as soon as the sweep reaches the scan line on which vertex resides.

- Processing points on the first scan line:

If p is MAX, insert it into array *OldPoints* twice;

If p is HMAX, insert it into array *OldPoints* once; Procedure 1

If p is between vertices on the horizontal line, ignore it.

No other type of vertex may happen on the first scan line.

- Move scan line down and process vertices, repeat it for every scan line s_k

(at most n-1 times):

Determine number and types of polygon vertices laying on s_k;

Calculate all intersections between s_k and polygon edges. These points are type INT. To locate those intersections, an array *SortedVerticesY* is searched starting from the first element and finishing just before the first vertex which resides on the current scan line is encountered. *SortedVerticesY* data structure is illustrated in Figure 5.3. If one of the neighboring vertices has the *Used* flag set to FALSE, the edge joining the examined vertex with this neighbor is intersecting the scan line. If both neighbors have flag set to TRUE, then the vertex may be discarded, since this will be the case of three vertices lying on the same line, Figure 5.2f. If the neighbors are ordered by y coordinate, then only one neighbor with the smaller y coordinate has to be tested. Insert the vertices and intersections into an array *Intersections*, ordered by x-coordinate;

- Create an array *CurrentPoints* from *Intersections* using the procedure:

  If MIN, add point to *CurrentPoints* twice;

  If INT, HMIX, add point once to *CurrentPoints*; Procedure 2

  If MAX, HMAX, do not add point.

- Update for the next scan line

  Update *OldPoints* from the points stored in intersections according to their types as follows:

If MAX, add point to *Oldpoints* twice;

If INT, HMAX, add point to *OldPoints* once; Procedure 3

If MIN, HMIN, do not add point.

- Move scan line until the polygon is exhausted.

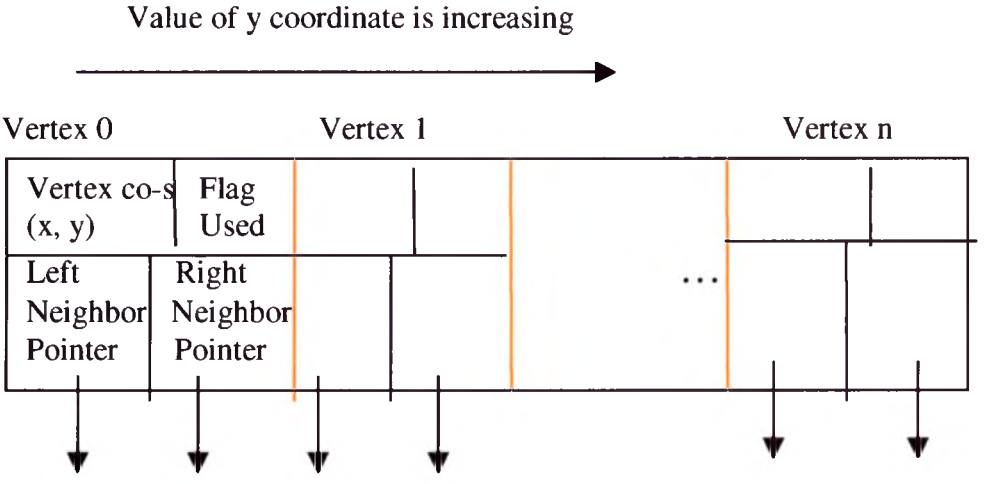


Figure 5.3 SortedVerticesY data structure.

Implementation of the step 2

After processing vertices at every scan line the arrays *OldPoints* and *CurrentPoints* have equal even number of vertices which may be joined by corresponding pairs

between two arrays. Trapezoids between s_{k-1} and s_k are created. This step is repeated inside the loop n-1 times at the most.

The implementation described above is used in most scan-line fill algorithms proposed earlier. But now this algorithm is generalized to accommodate the cases of polygons with several rings (e.g. holes), including the possibility of nested rings.

5.2.2 Processing of a polygon with holes

This algorithm is a generalization for all planar polygons that don't self-intersect. Holes will be processed and decomposed to trapezoids. Let's call a set of vertices composing the most outer polygon a *loop* and a set of vertices composing a hole *ring*. First consider a case when loop and rings are disjoint (this limitation will be lifted later). Loop and ring are *disjoint* when their edges and vertices don't intersect and don't touch.

5.2.2.1 Simple nested rings

Follow the steps of the algorithm for a non-monotone polygon without rings described above. Introduce following changes:

Create new arrays *Ring-j_OldPoints* and *Ring-j_CurrentPoints* for every ring $\mathbf{R_j}$.

- When the scan line encounters the first intersection with a ring, array *CurrentPoints* is generated according to Procedure 2. All ring vertices are ignored at this time because they are MAX and HMAX type.

- When the scan line is moved and Procedure 3 is executed for update of *OldPoints*, the intersections of the scan line with a ring $\mathbf{R_j}$ are now considered in two arrays: The array *Ring-j_OldPoints* belonging to $\mathbf{R_j}$ and The *OldPoints*, for the loop if it is within the loop or the array *Ring-j_CurrentPoints* if $\mathbf{R_j}$ is nested immediately within another ring, $\mathbf{R_k}$.

- Similarly, at subsequent scan lines in addition to the array *CurrentPoints*, for each ring Rj an array *Ring-j_CurrentPoints* is created using Procedure 2, so that ring intersections with the current scan line also occur in two arrays.

This process continues until all scan lines have been processed. If it is determined that any ring is not to be decomposed then the arrays *Ring-j_CurrentPoints* and *Ring-j_OldPoints* can be discarded.

5.2.2.2 Example of processing a non-monotone polygon containing a hole

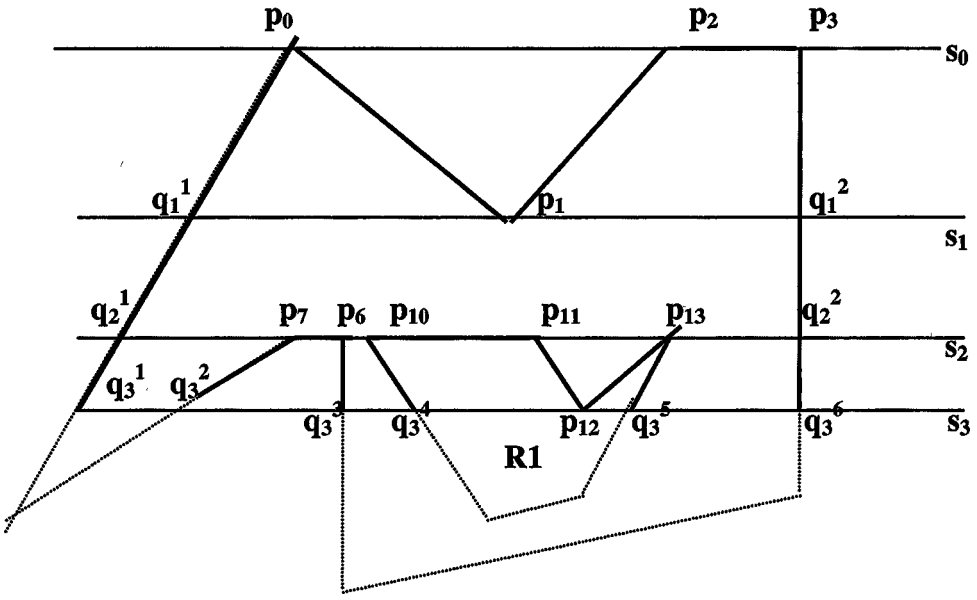


Figure 5.4. Trapezoidation of non-monotone polygon that contains a hole.

Figure 5.4 illustrates an example of trapezoidation algorithm execution, given by Borut Zalik in [25].

As a result of applying Procedure 1 on the s_0 scan line, the array *OldPoints* contains p_0, p_0, p_2, p_3.

1. Move to the next scan line s1. After calculation of the intersection of s_1 and polygon edges, q_1^1 and q_1^2 are obtained. Insert vertices and intersections into the array *Intersections*: q_1^1, p_1, q_1^2. Execute Procedure 2. After sorting, the array *CurrentPoints* contains the points q_1^1, p_1, p_1, q_1^2.

2. Create trapezoids, joining the *OldPoints* and *CurrentPoints* arrays together. Two trapezoids are created:

   p0 q_1^1 p1 p0 and **p2 p1 q_1^2 p3.**

3. Update the scan line, executing Procedure 3. The array *OldPoints* is q_1^1, q_1^2.

4. Move to the next scan line s_2. The *CurrentPoints* array is generated according to Procedure 2 and it contains q_2^1 and q_2^2. All ring points are ignored, because they are of type MAX and HMAX.

5. Create trapezoid $q_1^1\ q_2^1\ q_2^2\ q_1^2$

6. Update the scan line, executing Procedure 3. The Array *OldPoints* is $q_2^1\ p_7\ p_6\ p_{10}\ p_{11}\ p_{13}\ p_{13}\ q_2^2$. The array *Ring-1_OldPoints* is $p_{10}\ p_{11}\ p_{13}\ p_{13}$. The array *CurrentPoints* is $q_3^1\ q_3^2\ q_3^3\ q_3^4\ p_{12}\ p_{12}\ q_3^5\ q_3^6$. The array *Ring-1CurrentPoints* is $q_3^4\ p_{12}\ p_{12}\ q_3^5$.

7. Six new trapezoids are created:

   Trapezoid 1: $q_2^1\ q_3^1\ q_3^2\ p_7$,

   Trapezoid 2: $p_6\ q_3^3\ q_3^4\ p_{10}$,

Trapezoid 3: $p_{11}\ p_{12}\ p_{12}\ p_{13}$,

Trapezoid 4: $p_{13}\ q_3^5\ q_3^6\ q_2^2$,

Trapezoid 5: $p_{10}\ q_3^4\ p_{12}\ p_{11}$,

Trapezoid 6: $p_{13}\ p_{12}\ q_3^5\ p_{13}$.

Trapezoids 1-4 belong to the loop; whole trapezoids 5 and 6 belong to the ring R1.

Trapezoids 3 and 6 degenerated into triangles.

Continue until all scan lines are processed.

5.2.2.3 Touching and coincident rings

Inconsistencies in vertex type interpretation may appear if touching or coincident rings are present. For example, in Figure 5.5 **p** may be interpreted as type INT in ring $\mathbf{R_1}$ and type MIN in ring $\mathbf{R_2}$ (see Figure 6 for vertex classification).

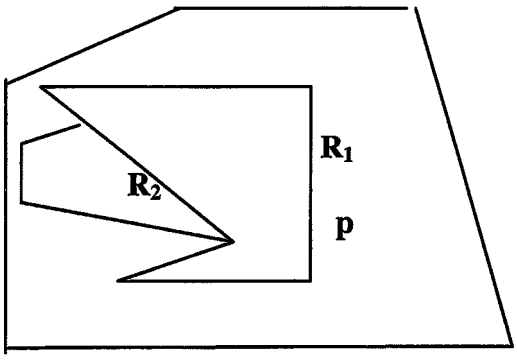


Figure 5.5 Polygon with partially coincident rings.

To avoid this problem Zalik proposed to number a vertex common to more then one ring differently in each ring, regarding it as a set of coincident vertices, rather then as a multiple vertex. The algorithm proceeds as previously and a trapezoid is generated "between" two coincident rings. At the final stage checking the pairs of points on the previous and current scan lines identifies the "null" trapezoids. If the points on both lines are coincident, the trapezoid is discarded. A simple test of the x-coordinates of the trapezoid vertices on the previous and the current scan lines is sufficient to identify such cases.

If a ring is intersected by the first scan line because it is coincides with the loop, the initialization step in Procedure P1 has to be modified to include the creation of an additional array *Ring-j_OldPoints* for each intersected ring.

Summary: The trapezoidation algorithm of Borut Zalik can accommodate any polygon containing any number of holes to any depth of nesting, where polygon edges may touch and overlap, but not intersect.

CHAPTER VI

REDUCTION OF THE NUMBER OF TRAPEZOIDS

The purpose of this work is to perform more efficient trapezoidal reduction, which would make applications using trapezoidation created more efficient. The post processing reduction algorithm we have developed is described in this section. The trapezoidation algorithm itself produces more trapezoids than necessary. The more complex the polygon is, more unnecessary polygons are created during the trapezoidation step, which ultimately leads to need for a more effective polygon reduction. This problem is addressed by developed reduction algorithm.

6.1 Case 1

The most evident and simple case of reduction is when the following conditions are true:

1. Both trapezoids belong to the same ring or loop.
2. There are two adjacent vertices in both trapezoids, e. g. there is a pair of vertices in each trapezoid, where the vertex in the trapezoid one and a vertex in a trapezoid two has equal coordinates x and y.
3. The side edges of two trapezoids are lie on the same line, e. g. parallel.

The example of this case is illustrated on the Figure 6.1 In this case it is easy to detect and reduce unnecessary trapezoids that were produced as a result of not looking ahead during trapezoidation.

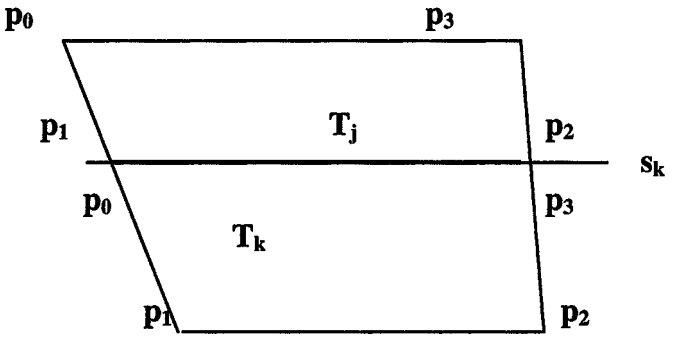


Figure 6.1 Trapezoid optimization, Case 1.

This kind of reduction does not modify the original polygon structure. More interesting situations that are accommodated by the reduction algorithm created in this research are cases that don't satisfy the conditions of Case1. By performing those kinds of reductions, the original polygon structure is modified on the condition that the error threshold is satisfied. Such reduction is desirable in the applications where the processing time of trapezoids is very significant compared to the time invested in reduction and the loss of information after reduction is insignificant. The extraction

engine for parasitic signal calculation is the example of such an application. It is extremely desirable to reduce number of polygons as much as possible if the required parameters of the polygon are preserved. Such parameters are connectivity, e. g. adjacent trapezoids cannot be disconnected and the total area of trapezoids before and after reduction should be within the error threshold.

6.2 Case 2

Assume that conditions 1 and 2 from the Case1 are satisfied. But condition 3 is not satisfied, so non-horizontal edges are not parallel, as on the Figure 6.2.

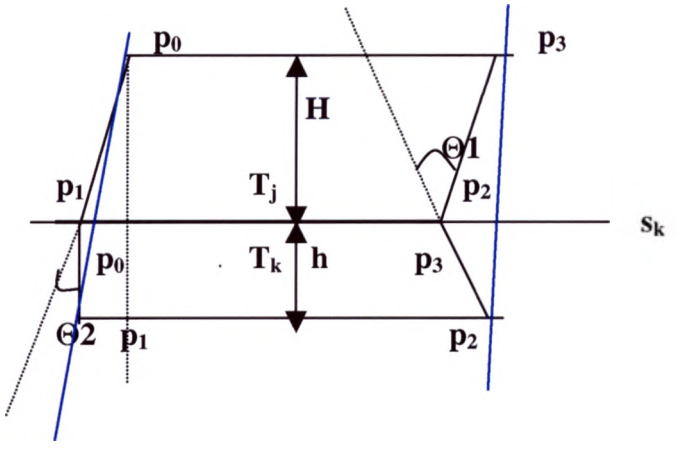


Figure 6.2 Trapezoid optimization, Case 2.

In which situations may such a case be simplified?

First of all a restriction has to be placed that would not allow reduction of the trapezoids with non-parallel edges if those edges do not belong to the loop. This restriction is necessary to avoid loss of information if this case appears in the ring, since reduced vertices $\mathbf{p_0}$ from $\mathbf{T_k}$ and $\mathbf{p_1}$ from $\mathbf{T_j}$ may be a part of some outer trapezoids. Because the reduction algorithm used is a "local" method, e.g. only two neighboring trapezoids are examined at a time, the coordinates of the vertices of the non-adjacent horizontal edges must be preserved. The only possible way to reduce vertex $\mathbf{p_0}$ from $\mathbf{T_k}$ and $\mathbf{p_1}$ from $\mathbf{T_j}$ is to join vertex $\mathbf{p_0}$ from $\mathbf{T_j}$ and $\mathbf{p_1}$ from $\mathbf{T_k}$ with a *reduction line*. But this would lead to decreasing the area of the polygon. This cannot be allowed, since in order to preserve connectivity we cannot decrease the total polygon area. In the example in Figure 10 only reduction of vertices $\mathbf{p_3}$ from $\mathbf{T_k}$ and $\mathbf{p_2}$ from $\mathbf{T_j}$ may be allowed, because after connecting vertices $\mathbf{p_2}$ from $\mathbf{T_k}$ and $\mathbf{p_3}$ from $\mathbf{T_j}$ the polygon area will slightly increase. This situation may occur only of the vertex to be reduced is on the same side as other vertices of the polygon after the cut by the reduction line.

The following constraints may be formulated for the Case2:

1. Vertex to be reduced belongs to loop
2. If after the cut with the reduction line the vertex is on the side where no other vertices from the examined trapezoids are present, do not reduce it.

3. The area increase requirement has to be satisfied, e.g. reduction error is within the threshold.

Let's examine possible configurations when Case 2 applies, illustrated in Figure 11, and formulate conditions which have to be tested in order to insure that the total area of polygon will not be reduced after trapezoidal reduction. Every case in Figure 11 successfully joins two trapezoids ABCD and DCEF into trapezoid ABEF. Reduction line BE eliminates vertex **p** and segment DC. The most important difference among these cases is relation between vertices B, E and segment GG1, which is a vertical segment with y-coordinate equal to y of vertex **p**, which we would like to eliminate. Lets call Θ an angle between upper trapezoid edge BC and vertical line GG1 and Φ an angle between lower trapezoid edge CE and GG1. In configuration Figure 6.3a, B and E is on the same side of GG1. There is no need to examine Θ in order to guarantee that there is no total area reduction. Area S of triangle CBE, as an ultimate measure of error, may be calculated as follows:

$$S_{CBE} = S_{BWV} - S_{BVE} - S_{CEW} \quad (6)$$

Area S = ½ * (height * base), so (7)

$$S_{BWV} = \frac{1}{2} * (WV*BV) =$$

$$\frac{1}{2} * ((WG1 + G1V)*GG1) = \frac{1}{2} * ((WG1 + \mathbf{x1})(\mathbf{y1}+\mathbf{y2})) \quad (8)$$

$$S_{BVE} = \frac{1}{2} * (BV*EV) = \frac{1}{2} * (BV*(G1V - EV)) = \frac{1}{2} * ((\mathbf{y1+y2})(\mathbf{x1-x2})) \quad (9)$$

$$S_{CEW} = \frac{1}{2} * (CG1*EW) = \frac{1}{2} * (\mathbf{y2}*(\mathbf{x2} + WG1)) \quad (10)$$

To calculate WG1 note that $\Phi = \Theta$, and $tg\Theta = \mathbf{x1/y1}$ from triangle GBC. Then

$WG1 = tg\Phi * CG1$

$$WG1 = tg\Theta*\mathbf{y2} = \mathbf{x1*y2/y1} \quad (11)$$

Plug WG1 into the area equation for S_{CBE}:

$$S_{CBE} = \frac{1}{2} * ((WG1 + \mathbf{x1})(\mathbf{y1+y2}) - (\mathbf{y1+y2})(\mathbf{x1-x2}) - (\mathbf{y2}*(\mathbf{x2} + WG1)) =$$

$$\frac{1}{2} * ((\mathbf{x1*y2/y1} + \mathbf{x1})(\mathbf{y1+y2}) - (\mathbf{y1+y2})(\mathbf{x1-x2}) - (\mathbf{y2}*(\mathbf{x2} + \mathbf{x1*y2/y1})) \quad (12)$$

After simplification,

$$S_{CBE} = \frac{1}{2} * (\mathbf{x1y2} + \mathbf{y1x2}) \quad (13)$$

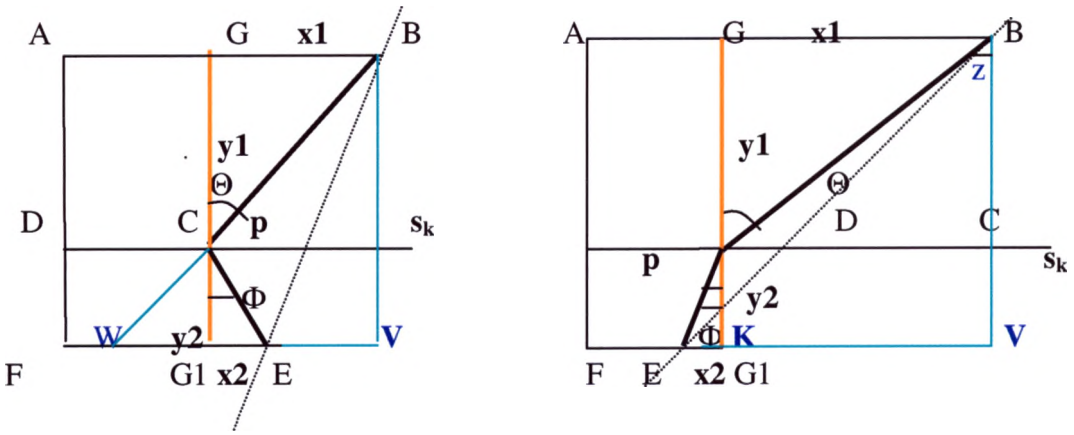


Figure 6.3 Examples of the case 2 topology.

Figure 6.3b illustrates a case when vertices B and E are on the opposite sides of GG1. If we consider GG1 as a 0 x-coordinate, then segment G1E will be negative and GB will be positive, e. g. **x2** <0 and **x1**>0. To avoid total area reduction following restriction must be applied:

$\Phi < \Theta$, it translates to: the absolute value of **x1/y1** has to be bigger than the absolute value of **x2/y2:**

$$\mathbf{ABS\ (x1/y1) < ABS(x2/y2)} \qquad (14)$$

After this condition is satisfied, the area of S_{CBE} may be calculated:

$$S_{CBE} = S_{GG1VB} - S_{GBC+} S_{CEG1-} S_{BVE} =$$

$$\mathbf{x1*(y1+y2) - ½*x1*y1 - ½*(y1+y2)*(x1+x2) + ½*x2*y2 = ½*(x1y2 - y1x2)} \qquad (15)$$

$$S_{CBE} = ½ * (\mathbf{x1y2 - y1x2}) \qquad (16)$$

Figure 6.3c illustrates a case when vertices B and E are on the opposite sides of GG1. If we consider GG1 as a 0 x-coordinate, then segment G1E will be positive and GB will be negative, e. g. **x1** <0 and **x2**>0. To avoid total area reduction, the following restriction must be applied: $\Phi > \Theta$. It translates to: the absolute value of **x1/y1** has to be smaller then absolute value of **x2/y2:**

ABS (x1/y1) > ABS(x2/y2) (17)

After this condition is satisfied, the area of S_{CBE} may be calculated:

$S_{CBE} = S_{BEW} - S_{CG1E} - S_{KCG1W} - S_{BKC} =$

½*(x1+x2)*(y1+y2) – x1y2 - ½*x2*y2- ½* y1*x1 = ½*(x2y1 – y2x1) (18)

S_{CBE} **= ½ * (x2y1 – y2x1)** (19)

The reduction algorithm for case 2 is:

1. Check if there are two vertices on the scan line with the same coordinates for both trapezoids.
2. Create a reduction line, which joins the vertices on the opposite parallel edges. Check if the vertices on the scan line lie on the same side as all other vertices of the trapezoids. If they lie on the reduction line, then case 2 is reduced to case 1. Both vertexes on the scan line must be on the "inner" side of the reduction line.
3. Calculate the error: S_{added} **= ½*(x2y1 + y2x1)** (20)

Where:

y1 is the distance from the scan line to the highest edge;

y2 is the distance to the lowest edge;

x1 is difference between the x coordinate of the highest edge vertex and the x coordinate of the vertex on a scan line;

x2 is difference between x coordinate of the lowest edge vertex and the x coordinate of the vertex on a scan line.

x1 and **x2** may be positive or negative, as it was demonstrated on Figure 6.3(b) and 6.3(c).

4. If S_{added} is smaller or equal to the Error, create a new trapezoid as in Case1 by connecting the highest and the lowest edges of the two polygons. Discard the vertices that reside on the scan line.

6.2 Case 3

Now let's consider case 3, which relaxes the constraints of case 2 even further: requirement 1 from case 1 (and requirement from case 2) is satisfied, but requirements 2 and 3 are not:

1. Both trapezoids belong to the same ring or loop.

2. There are no two adjacent vertices in both trapezoids, e. g. there is a pair of vertices in both trapezoids with the same y coordinate, but x coordinates are different (if there is a pair of vertices from neighboring trapezoids with same x and y coordinates, then this edge reduces to the case 2).

3. The side edges of two trapezoids do not lie on the same line (if they are this case is reduced to case 1).

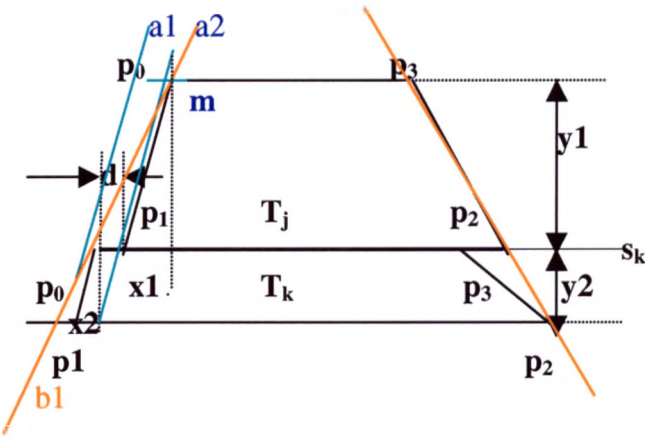


Figure 6.4 Trapezoid optimization, Case 3.

Let's examine two trapezoids on the Figure 6.4. We would like to join edge $\mathbf{p_1p_0}$ from trapezoid $\mathbf{T_k}$ and the edge $\mathbf{p_1p_0}$ from trapezoid $\mathbf{T_j}$. Edges lie on the segments a1 and a2. If we use the same heuristic as in the case 2 to eliminate vertices $\mathbf{p_0}$ from $\mathbf{T_k}$ and $\mathbf{p_1}$ from $\mathbf{T_j}$ by joining vertices $\mathbf{p_1}$ from $\mathbf{T_k}$ and $\mathbf{p_0}$ from $\mathbf{T_j}$ with the segment b1, we would lose some area from $\mathbf{T_k}$. To avoid this situation, another restriction must be applied: no vertices should be cut off by the reduction line from the main loop. The right hand side edges of $\mathbf{T_k}$ and $\mathbf{T_j}$ on the Figure 6.4 satisfy this restriction. So

trapezoids $\mathbf{T_k}$ and $\mathbf{T_j}$ on the Figure 6.4 cannot be reduced because one of the edges doesn't satisfy all reduction requirements.

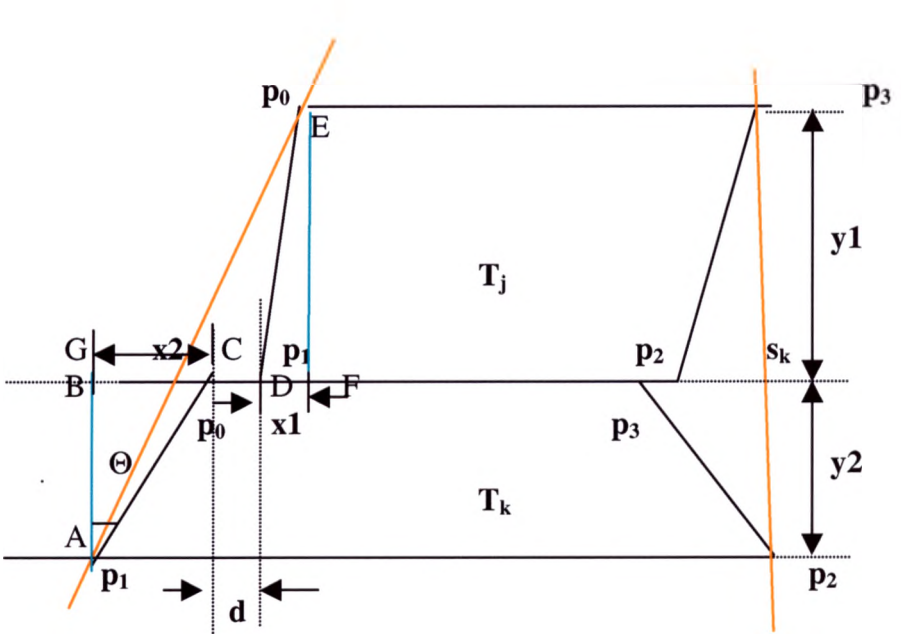


Figure 6.5 Trapezoid optimization, Case 3. Added area calculation.

Let's examine example on the Figure 14.

Note that

$$BD = BC + \mathbf{d} \tag{21}$$

$$BC = \mathbf{x2} - GB = \mathbf{x2} - tg\Theta * \mathbf{y2}; \text{ (from } \Delta AGC) \tag{22}$$

$tg\Theta = BF/y1 = (BC + d + x1)/y1;$ (from ΔBEF) (23)

$BC = x2 - y2*(BC + d + x1)/y1 = x2 - BC*y2/y1 - y2*d/y1 - y2*x1/y1$ (24)

$BC*(1+y2/y1) = x2 - y2*d/y1 - y2*x1/y1$ (25)

multiply both sides of equation (25) by **y1:**

$BC = (x2y1 - y2*d - y2*x1)/ (y1+y2)$ (26)

$S_{added} = S_{BAC} + S_{BED} = ½* BC*y2 + ½* BD*y1 = ½* BC (y2 + y1) + ½*d*y1$ (27)

plug into (27) expression for BC (26):

$S_{added} = ½*(x2y1 - y2*d - y2*x1) + ½*d*y1 =$

$½*(x2*y1 - x1*y2 + d*y1 - d*y2)$ (28)

The reduction algorithm for the case 3 is:

1. Check if there are two vertices on the scan line with the same y coordinates and different x coordinates for both trapezoids. The difference of x coordinates of corresponding vertices lying on the scan line is evaluated:

$$\Delta x = x1 - x2 = d \tag{29}$$

if $\Delta x > \Delta x_{max}$, ignore the case, these trapezoids cannot be eliminated. Δx_{max} is maximum displacement error.

2. Create a reduction line, which would join the vertices on the opposite parallel edges. Check if the vertices on the scan line lie on the same side as all other vertices of the trapezoids. Both vertices on the scan line must be on the "inner" side of the reduction line.

3. Calculate the error: $S_{added} = \frac{1}{2}*(x2*y1 - x1*y2 + d*y1 - d*y2)$

Where:

d is difference of x coordinate of the corresponding vertices to be reduced from different trapezoids ($\mathbf{p_1}$ from $\mathbf{T_j}$ and $\mathbf{p_0}$ from $\mathbf{T_k}$; $\mathbf{p_2}$ from $\mathbf{T_j}$ and $\mathbf{p_3}$ from $\mathbf{T_k}$).

y1 is the distance from scan line to the highest edge;

y2 is the distance to the lowest edge;

x1 is difference between x coordinate of the highest edge vertex and the x coordinate of the vertex on a scan line;

x2 is difference between x coordinate of the lowest edge vertex and the x coordinate of the vertex on a scan line.

x1 and **x2** may be positive or negative, as it was demonstrated on Figure 6.3(b) and 6.3(c).

4. If S_{added} is less then or equal to the Error, create a new trapezoid as in Case1 by connecting the highest and the lowest edges of the two polygons. Discard the vertices that reside on the scan line.

Case 3 is the most general case allowing reduction; it may lead to a significant change of the data topology, if the Error and Δx_{max} is set too large. Selection of the maximum error parameters may be critical in preserving topology of the data.

6.4 Control of the degree of reduction and error evaluation

Since the local (greedy) method is used for this reduction algorithm implementation, accumulation of error may occur as a result of several consecutive reductions. The example of a such situation is demonstrated on the Figure 6.6.

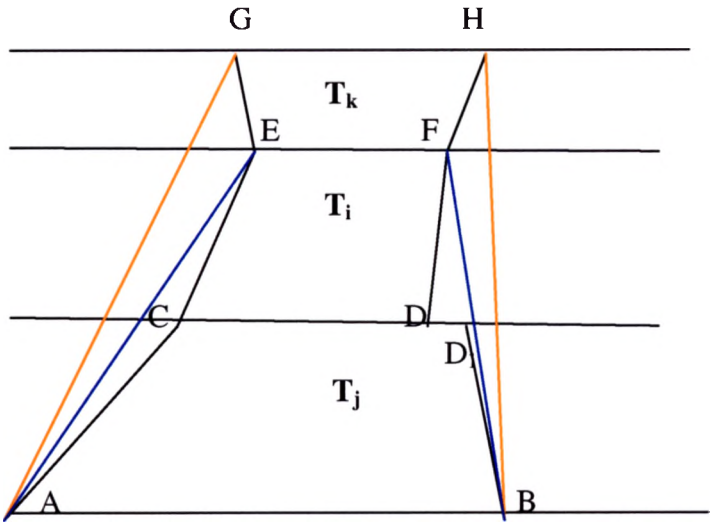


Figure 6.6 Error accumulation as a result of two consecutive reductions.

Figure 6.6 demonstrates the most general case, leading to changing data topology. Assume that at step one trapezoids $\mathbf{T_j}$ and $\mathbf{T_i}$ are joined, since the area increase satisfies the error threshold and connectivity may be preserved. A trapezoid AEFB is generated as a result of a such reduction. The area increase is the area of AEC combined with the area of DFD_1B. At the next step the scan line is lifted and trapezoid $\mathbf{T_k}$ is evaluated with trapezoid AEFB for a possible reduction. The connectivity requirement is satisfied and the aresa of AGE and FHB are examined. It satisfies the error threshold and a trapezoid AGHB is created. But at this stage the actual area change will be the area of AGEC combined with the area of BD_1DFH,

which violates the area threshold. To a avoid situation as this, the variable ERROR_CURRENT will be used. The advantage of this approach is that this is a dynamic variable, which is updated at every scan line move during reduction stage. There is no need for memory allocation to keep historical data as a data member for every trapezoid as is done in most "history keeping" algorithms. One dynamically changing variable is sufficient, since we reduce one trapezoid at the time.

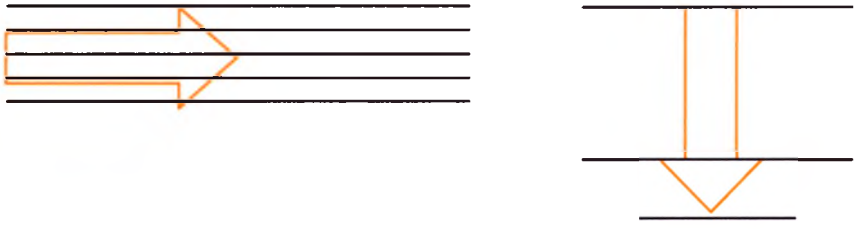
The error evaluation algorithm is:

1. Initialize ERROR_CURRENT to 0.
2. Examine first couple of the trapezoids and if they satisfy the reduction requirements, perform reduction and set the value of ERROR_CURRENT to the value of the total area added during that reduction.
3. Examine next couple of the trapezoids. If they cannot be reduced, set value of the ERROR_CURRENT to 0. If they may be reduced, add the value of the area increase as a result of this possible reduction to the value of the ERROR_CURRENT. Compare the value of the ERROR_CURRENT with the error threshold. If it is smaller then error threshold, perform the reduction. If ERROR_CURRENT is bigger then the error threshold, do not perform reduction and set ERROR_CURRENT to 0.
4. Move the scan line and perform step 3 until all trapezoids are exhausted.

6.5 Complexity analysis of the trapezoidation and reduction algorithm

Time complexity of the trapezoidation and reduction algorithm is determined by the number of scan lines. Number of scan lines depends on the physical form of the polygon, number of vertices, and the orientation of the polygon with respect to the sweep line. The number of scan lines depends on polygon orientation, as it is demonstrated in Figure 6.7. Therefore the analysis of the average case is difficult, so the worst case will be analyzed. In the worst case the number of scan lines m will be equal to the number of the vertices n.

m <= n is always true. n is the number of vertices and m is the number of scan lines passing through them.



5 scan lines for horizontal orientation 3 scan lines for vertical orientation

Figure 6.7 Dependency of number of the scan lines on the polygon orientation.

Analysis of the steps performed:

1. Determining the vertex neighbors requires only one pass and is completed in linear time.

2. Sorting vertices by the y-coordinate to create data structure *SortedVerticesY* is performed using Quick sort. Quick sort complexity is

$$T_{sorty}(n) = O(n \log_2 n) \quad (30)$$

3. Sorting vertices with maximal y-coordinate by x-coordinate to create array OldPoints depends on number of polygon vertices with maximal y-coordinate t. Worst case for t is a triangle with all vertices laying on the upper horizontal edge. Then t=n-1. In general t is not dependent on n and t <<n.

Therefore initialization time is

$$\mathbf{T_{init}(n) = O(n \log_2 n)} \quad (31)$$

4. The main loop of the algorithm contains m-1 steps. The following tasks are performed in the loop:

Determining the number of the vertices t on the scan line (t < n). It is linear process of t steps.

Calculating intersections between edges and scan line to create j intersection points. It is a linear process of j steps.

Sorting all points on the scan line k = j + t by x coordinate. Quick sort algorithm time complexity is

$$T_{sortx}(k) = O(k \log_2 k), \text{ where } k < n. \quad (32)$$

If there are rings, then the parent ring must be identified by searching half of the points on the scan line (negligible time for large n).

Create trapezoids between current and previous scan lines. If there are no rings, the number of trapezoids is k-1 at the most.

If rings are present, the ring intersections should be included into array twice, 2k-1 in the worst case. So process time is linear in k, where k < n.

5. So in the loop execution the most critical part is x-coordinate sorting of the intersections. Therefore

$$T_{process}(n) = (m-1)T_{sortx}(k), \text{ where } m, k < n. \quad (33)$$

$$T_{process}(n) = (m-1)\, O(k \log_2 k) \quad (34)$$

accounting that $(m-1)k <= n^2$ (35)

$$\mathbf{T_{process}(n) = O(n^2 \log_2 n)} \quad (36)$$

6. Trapezoid reduction complexity.

Generated trapezoids are sorted by y-coordinates and x-coordnates. If there are p trapezoids above the scan line and q trapezoids below it then each set has to be tested against the other to check if trapezoids can be joined. If trapezoid cannot be joined it is removed from the set. If there are two trapezoids that can be joined, then two trapezoids are removed from the set. So number of the tests on each scan line is p + q at the most. The test takes place m-2 scan lines. Therefore

$$T_{reduc}(n) = O((p+q)(m-2)) \quad (37)$$

since $p + q < n$, $m < n$:

$$\mathbf{T_{reduc}(n) = O(n^2)} \quad (38)$$

Total worst case time complexity for trapezoidation and reduction is

$$\mathbf{T(n) = T_{init}(n) + T_{process}(n) + T_{reduc}(n)} \quad (39)$$

Substitute (31), (36), and (38) into (39):

$$T(n) = O(n \log_2 n) + O(n^2 \log_2 n) + O(n^2) \quad (40)$$

$$\mathbf{T(n) = O(n^2 \log_2 n)} \quad (41)$$

The formula (41) reflects the worst possible case, however in average case the performance is much better.

CHAPTER VII

PERFORMANCE AND TESTS OF THE REDUCTION ALGORITHM

7.1 Testing reduction algorithm on simple geometry data.

The details of the reduction algorithm performance may be displayed on the simple geometry testcases. The error_threshold and error_delta are set to the values, which are recommended for the layout extraction. The testcases described are displayed in the Appendix. The algorithm was tested on simple and more complex geometries, with the holes and without. The example test cases attached in Appendix demonstrate on which types of structures reduction algorithm is most effective. The geometries with more small details and curves achieve higher rate of reduction then geometries with the large shapes and long edges.

Every test case includes:

- screen shot picture of the geometry after trapezoidation
- screen shot of the geometry after reduction using the reduction algorithm described in this work
- input data file
- list of polygons after reduction

- summary of the number polygons extracted and number of polygons reduced.

The summary of the number of polygons produced and reduced for the attached testcases is demonstrated on the Figure 7.1.

Case1

No of generated trapezoids 417

No of erased trapezoids 283

No of trapezoids after reduction 134

Case2

No of generated trapezoids 28

No of erased trapezoids 15

No of trapezoids after reduction 13

Case3

No of generated trapezoids 191

No of erased trapezoids 99

No of trapezoids after reduction 92

Case4

No of generated trapezoids 7

No of erased trapezoids 1

No of trapezoids after reduction 6

Case5

No of generated trapezoids 87

No of erased trapezoids 51

No of trapezoids after reduction 36

Figure 7.1 Summary of the trapezoid count for testcases.

The reduction rate depends on the geometrical characteristics of the data. The test was performed on 50 geometry testcases and the average reduction time was calculated. The average reduction rate is 3.3 times, e.g. there are 3.3 times fewer trapezoids to process after reduction:

number of trapezoids after reduction/number of trapezoids generated = 0.27 (42)

number of reduced trapezoids/number of trapezoids generated = 0.73 (43)

Formula 43 displays that approximately 73% of trapezoids are reduced. This is a very good result, competitive with all other available polygon reduction algorithms.

7.2 Testing reduction algorithm on the design layout

Figure 1 in the Appendix demonstrates an example of the design viewed in PLE, Cadence layout editor. The extractor tool extracts this layout data, extracted polygons are processed and connectivity is established. Different colors on the layout annotate different layers of the design: green is poly, pink is diffusion, red is metal1 routing, blue is metal2 routing, yellow squares are contacts and read squares are via1. If you look at the layout vertically cross cut, you will see so called *process stack*, as it is demonstrated on the Figure 7.2. This is the third dimension of a design. Every layer in

the stack has it is own physical characteristics: permitivity, resistance, height, and process specific characteristics. All stack information will be taken from technology file, which is provided to extraction tool as an input along with the layout data. After all polygons are extracted from layout and connection among polygons is established, parasitic signals will be calculated, based on the process and layout information. Current design technology allows stacks of 7 to 8 metal layers and millions of polygons on one chip. The processing time grows very fast, as the number of polygons grows. It may take several days to extract parasitic signals on the modern high integrated circuit design. This is why reduction of the number of extracted polygons is so important. Star_RCXT extraction tool has a great advantage of extracting all layers in the design simultaneously, so it doesn't have to do several scans of the layout. But calculation of the parasitic signals may be very time consuming and reduction of the number of polygons decreases processing time dramatically.

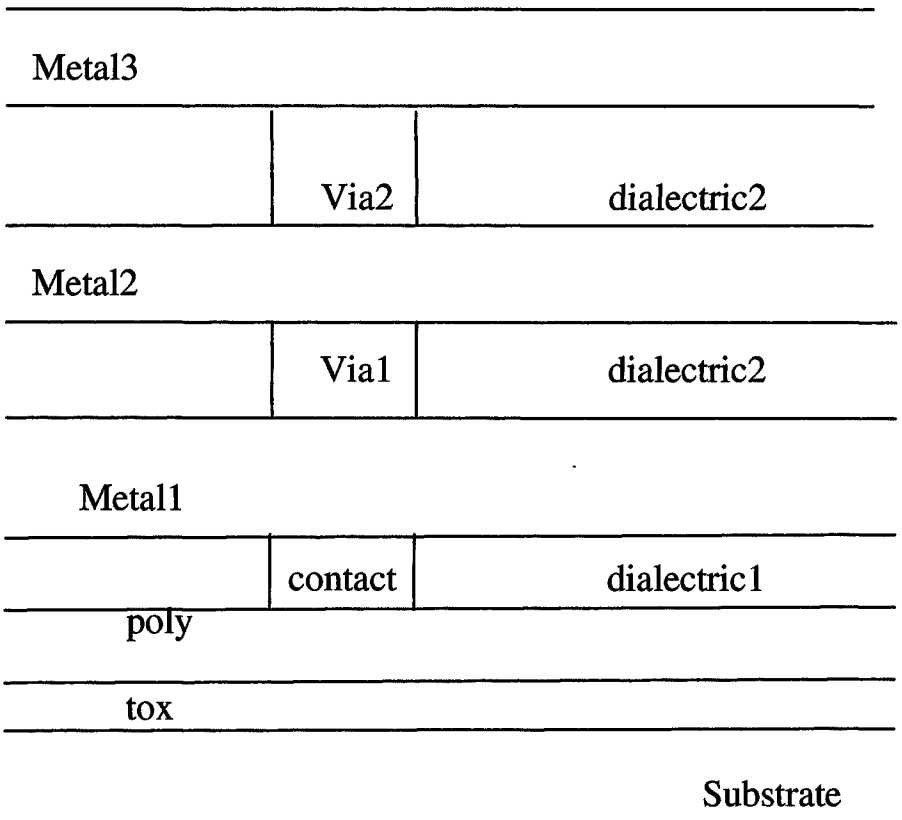


Figure 7.2 Example of a process stack

7.2.1 Tests performed

The tests performed on the code had to verify accuracy of the reduction algorithm and decrease of the total execution time of extraction if reduction is used. Since Star_RCXT executable cannot be accessed by anyone but Avant! (Synopsis)

developers, the code was tested as a separate executable and it will be forwarded to the company R&D for the future incorporation into the xTractor engine of Star_RCXT tool.

1. Accuracy test.

To verify the accuracy of the reduction algorithm the following tests were performed.

Setup:

The extraction run was performed on a design. Layout data was given in gds format. Parasitic signals were extracted and printed in the netlist. The extracted trapezoid file was used as an input file into the reduction function. After reduction was performed, the extraction tool further processed new reduced trapezoid file. The new parasitic netlist was produced. To verify the accuracy of the extracted information after reduction the two netlists were compared.

Example of the spef format netlist, a fragment for one net parasitic information is displayed on the Figure 7.3.

```
*SPEF "IEEE 1481-1998"
*DESIGN "r00mxn52la5"
*DATE "Wed Apr  3 19:20:05 2002"
*VENDOR "Avant!"
*PROGRAM "Star-RCXT"
*D_NET *10 1.7571

*CONN
*I *26:d_x1880y3090 B *C 1.88 3.09 *D P
*I *24:g_x960y3735 I *C 0.96 3.735 *L 0.0696

*N *10:181 *C 1.32 0.18
*N *10:149 *C 0.92 4.45
...

*CAP
1 *10:181 *8:121 0.000876261
2 *10:181 *13:197 3.56865e-05
3 *10:181 *13:192 0.000117115
...

314 *10:341 1.10000e-05
315 *10:330 1.10000e-05
316 *10:338 1.10000e-05
317 *10:323 1.10000e-05

*RES
1 *10:181 *10:311 89.1361 // $l=0.450000 $w=0.0900000 $lvl=23
2 *10:181 *10:314 66.8521 // $l=0.150000 $w=0.0800000 $lvl=23
3 *10:149 *10:148 77.9941 // $l=0.280000 $w=0.0800000 $lvl=23
....

32 *10:236 *10:234 0.00100000 // $l=0.280000 $w=10.0000 $lvl=20
33 *10:236 *10:232 0.00100000 // $l=0.560000 $w=10.0000 $lvl=20
34 *10:230 *10:228 0.00100000 // $l=0.280000 $w=10.0000 $lvl=19
*END
```

Figure 7.3 Example of the parasitic signal information in spef netlist.

Experimenting with the value of the error threshold S_{added} Error (28) and Δx_{max} error delta (29) default values were defined. When error_threshold and error_delta are set to the default value, maximum reduction is achieved, while the netlist produced after reduction is identical to the netlist produced on the unreduced data. The compare tool was used to verify tested netlists and to confirm that the parasitic values netlisted are correct. The necessity of the compare tool arose because netlist size may take more than a gigabyte of memory and it is impossible to verify it without automation.

2. Efficiency and execution time reduction test

To verify execution time reduction, the following tests were performed. Trapezoidation was run the on a significantly large layout database. The CPU time was monitored. The extraction was performed on the original data base and on the reduced data base. There was significant execution time reduction. The time table is displayed on the Figure 19. Tests were performed on a 2G hp machine.

The processing time reduction achieved by using our reduction algorithm is approximately 26%, which translates into big time savings when this algorithm is used in production.

Table 7.1 CPU time of extraction performed on reduced and on not reduced data

Design	extraction time original data	reduction time	extraction time reduced data	reduc. time + extr. time	total time reduction	Size, Mb
Cell1	6,5 h	5.3 min	4.3 h	4.5 h	1 h	185
Cell2	1,5 h	1.1 min	0.8 h	0.9 h	0.6 h	30
Cell3	40,2 h	62.0 min	28.2 h	29.2 h	11 h	1100
Cell4	26,3 h	37.2 min	17.9 h	18.5 h	7.8 h	750

CHAPTER VIII

CONCLUSION

This paper describes a trapezoid reduction algorithm created for the processing of the extracted polygons. This algorithm may be used in a wide variety of graphical applications, but it is specifically targeted for Electrical Design Automation tools (EDA), particularly for design layout extraction. The reduction algorithm was tested in a production extraction tool and significant runtime reduction was displayed along with preserving integrity of the data. This method is seen to deal with a large number of complex polygons effectively and economically. Tests performed on the real layout data demonstrated that the number of trapezoids was reduced by approximately 50%, while connectivity was preserved and the most conservative error threshold satisfied. The complexity of the reduction algorithm is $O(n^2)$, while the complete trapezoidation and reduction processes display worst time complexity of $O(n^2 \log_2 n)$. The decomposition algorithm used in this work is trapezoidation algorithm proposed by Borut Zalik, which successfully processes non-monotone polygons with a number of holes, including nested holes. The reduction algorithm described in this paper is a local method, and it is using greedy heuristics. The originality and novelty of this algorithm is in a feature that allows reduction of trapezoids with the slight modification of the layout topology, controlled by the error threshold, set by the user. This feature allows control of the

level of the reduction and maintains the original connectivity of the polygons, which is necessary for the correct calculation of the parasitic signal and electrical connectivity information. The algorithm was implemented in C++ and displays exceptional stability. The trapezoid reduction rate in a very accurate mode is approximately 73%. If incorporated into the parasitic signal extraction tool and released into production, this reduction algorithm will significantly increase productivity of numerous integrated circuit design teams. Currently it takes several days to extract and calculate parasitic signals on designs containing several million of gates. By using the proposed reduction algorithm the parasitic signal extraction time may be reduced by approximately 26%. This translates into weeks of time saved before design tape out date.

APPENDIX A

Figure 1a. Layout view

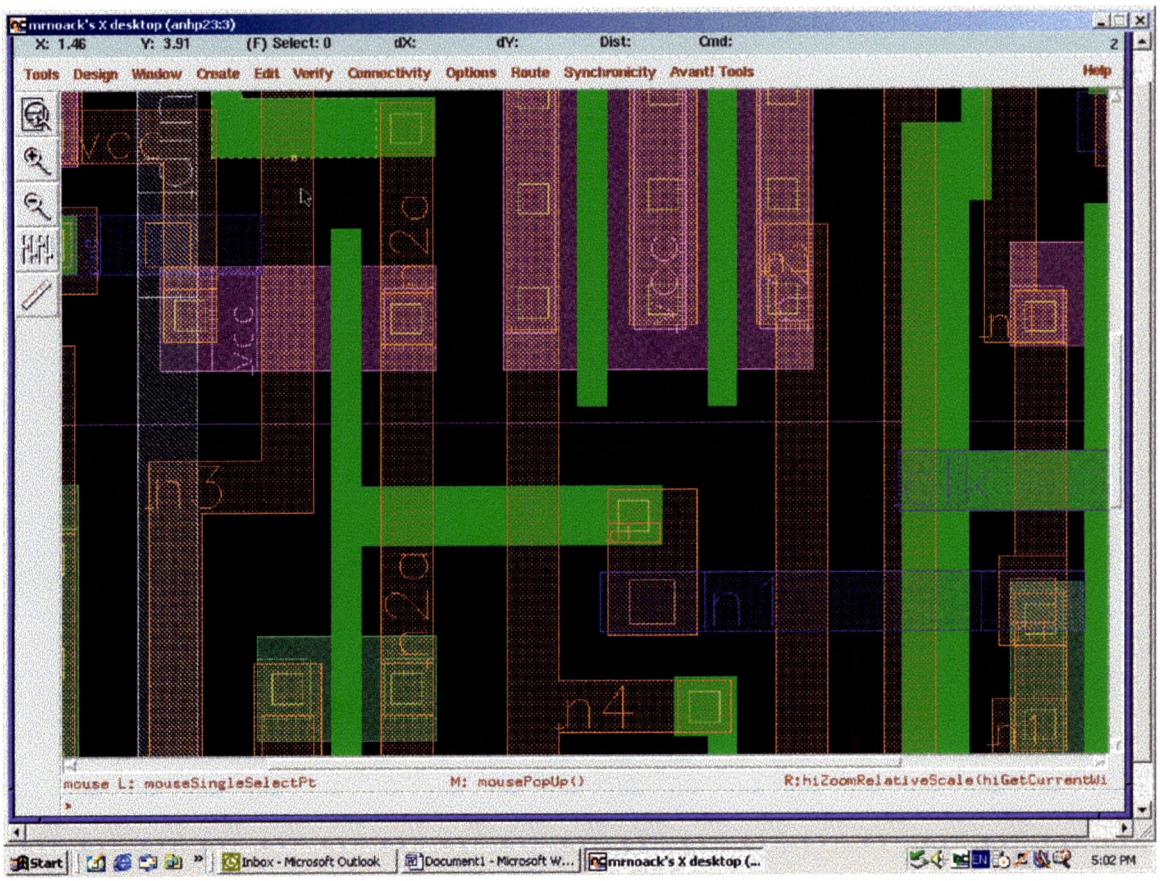
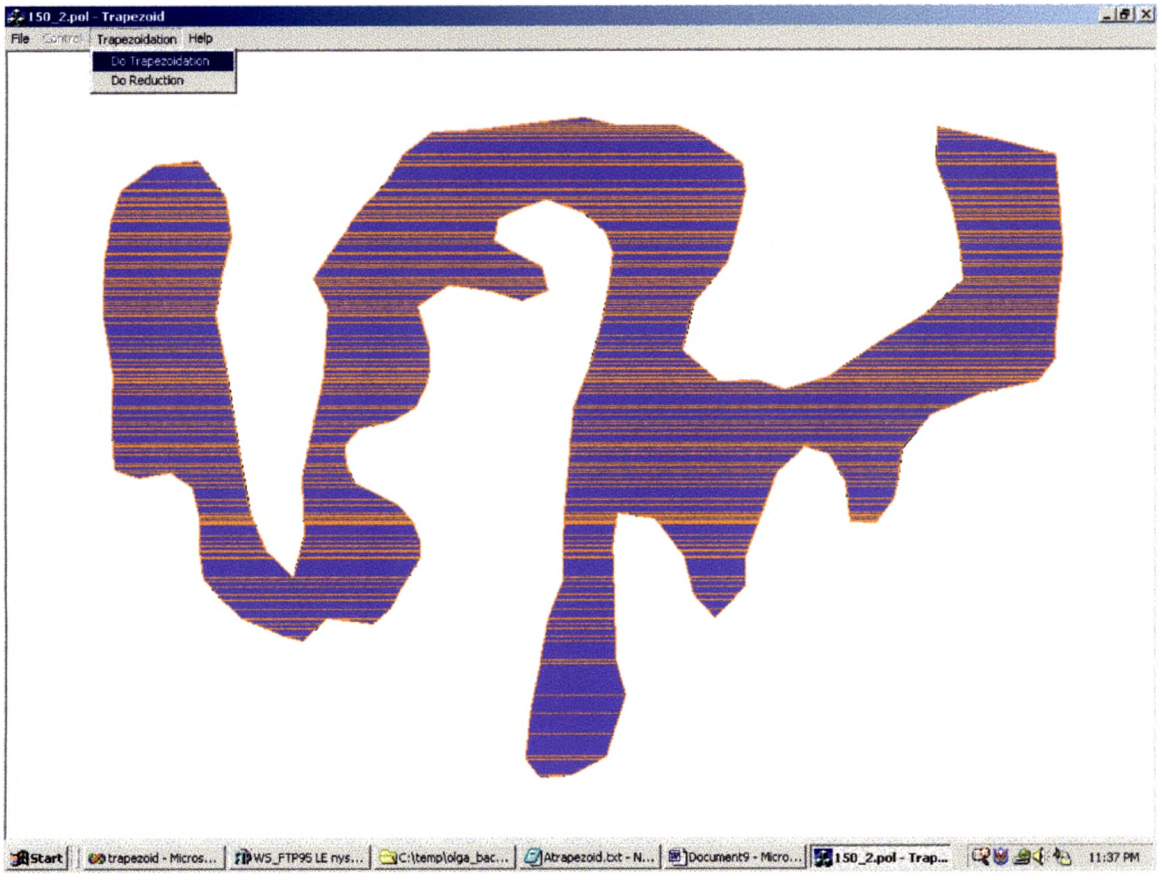


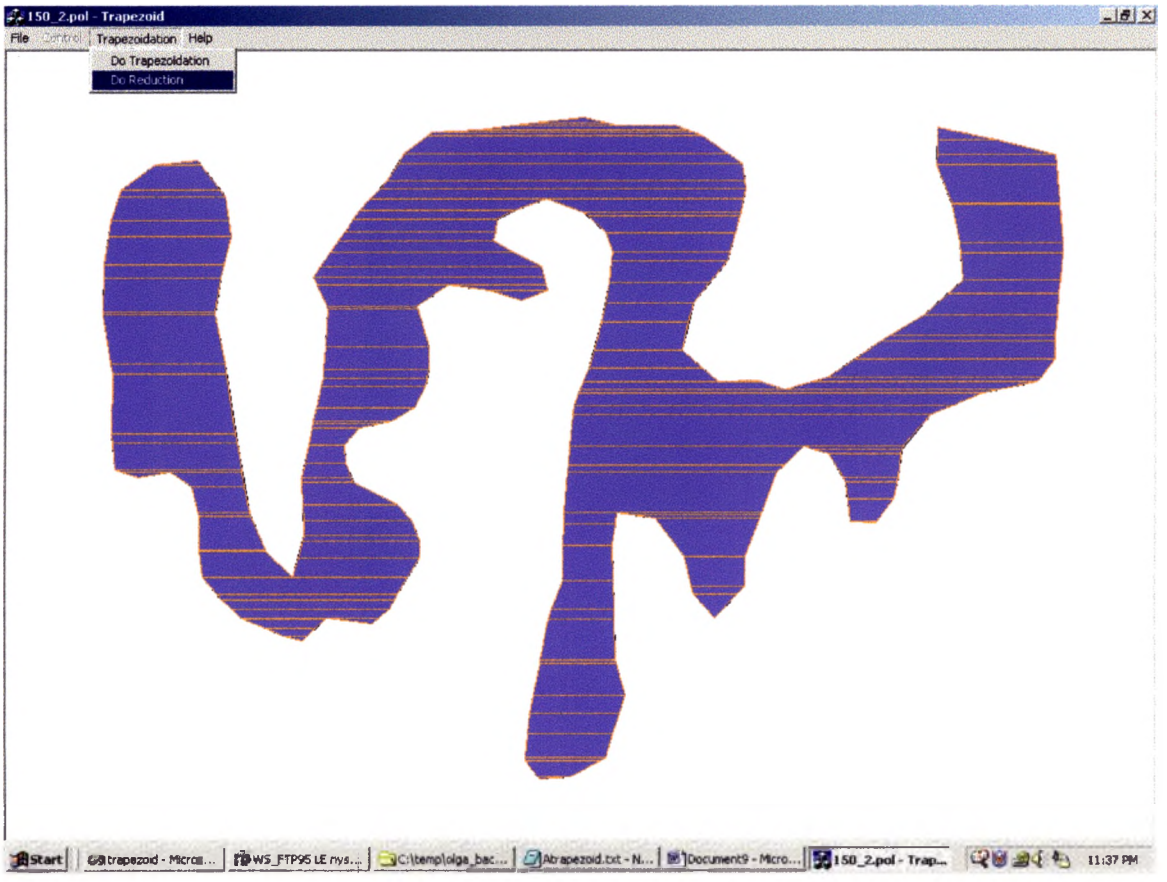
Figure 1b. Layout view

Testcase 1
150_1.pol

Geometry after trapezoidation.



Geometry after reduction



Input data:

150_2.pol:

L 1
R 0
L
140
185.000000 476.000000
168.000000 472.000000
152.000000 465.000000
131.000000 457.000000

112.000000 440.000000
95.000000 420.000000
92.000000 396.000000
92.000000 366.000000
85.000000 340.000000
66.000000 327.000000
37.000000 332.000000
16.000000 325.000000
13.000000 294.000000
14.000000 241.000000
5.000000 186.000000
6.000000 144.000000
12.000000 105.000000
21.000000 78.000000
52.000000 56.000000
90.000000 52.000000
115.000000 84.000000
120.000000 119.000000
111.000000 156.000000
106.000000 188.000000
115.000000 232.000000
126.000000 302.000000
137.000000 363.000000
152.000000 397.000000
176.000000 420.000000
186.000000 383.000000
184.000000 332.000000
194.000000 285.000000
203.000000 245.000000
207.000000 183.000000
194.000000 155.000000
222.000000 115.000000
235.000000 96.000000
260.000000 70.000000
275.000000 46.000000
300.000000 27.000000
343.000000 25.000000
391.000000 19.000000
436.000000 14.000000

465.000000 21.000000
518.000000 21.000000
543.000000 30.000000
579.000000 55.000000
582.000000 77.000000
577.000000 101.000000
566.000000 141.000000
535.000000 178.000000
525.000000 220.000000
558.000000 248.000000
593.000000 247.000000
617.000000 255.000000
656.000000 244.000000
703.000000 212.000000
742.000000 189.000000
777.000000 158.000000
774.000000 125.000000
767.000000 91.000000
753.000000 56.000000
754.000000 23.000000
806.000000 34.000000
860.000000 47.000000
864.000000 90.000000
867.000000 134.000000
863.000000 170.000000
860.000000 207.000000
860.000000 228.000000
844.000000 248.000000
795.000000 258.000000
749.000000 277.000000
722.000000 307.000000
719.000000 332.000000
715.000000 351.000000
699.000000 372.000000
676.000000 371.000000
672.000000 336.000000
657.000000 311.000000
635.000000 305.000000
610.000000 327.000000

596.000000 365.000000
582.000000 402.000000
582.000000 428.000000
555.000000 456.000000
535.000000 435.000000
527.000000 402.000000
501.000000 368.000000
467.000000 363.000000
463.000000 392.000000
465.000000 448.000000
466.000000 493.000000
475.000000 524.000000
464.000000 558.000000
458.000000 579.000000
426.000000 596.000000
398.000000 598.000000
385.000000 582.000000
389.000000 540.000000
398.000000 496.000000
405.000000 457.000000
418.000000 423.000000
419.000000 373.000000
423.000000 321.000000
428.000000 269.000000
442.000000 236.000000
453.000000 195.000000
460.000000 160.000000
463.000000 133.000000
457.000000 116.000000
437.000000 98.000000
403.000000 86.000000
375.000000 96.000000
359.000000 104.000000
356.000000 123.000000
373.000000 133.000000
399.000000 146.000000
404.000000 167.000000
381.000000 176.000000
352.000000 167.000000

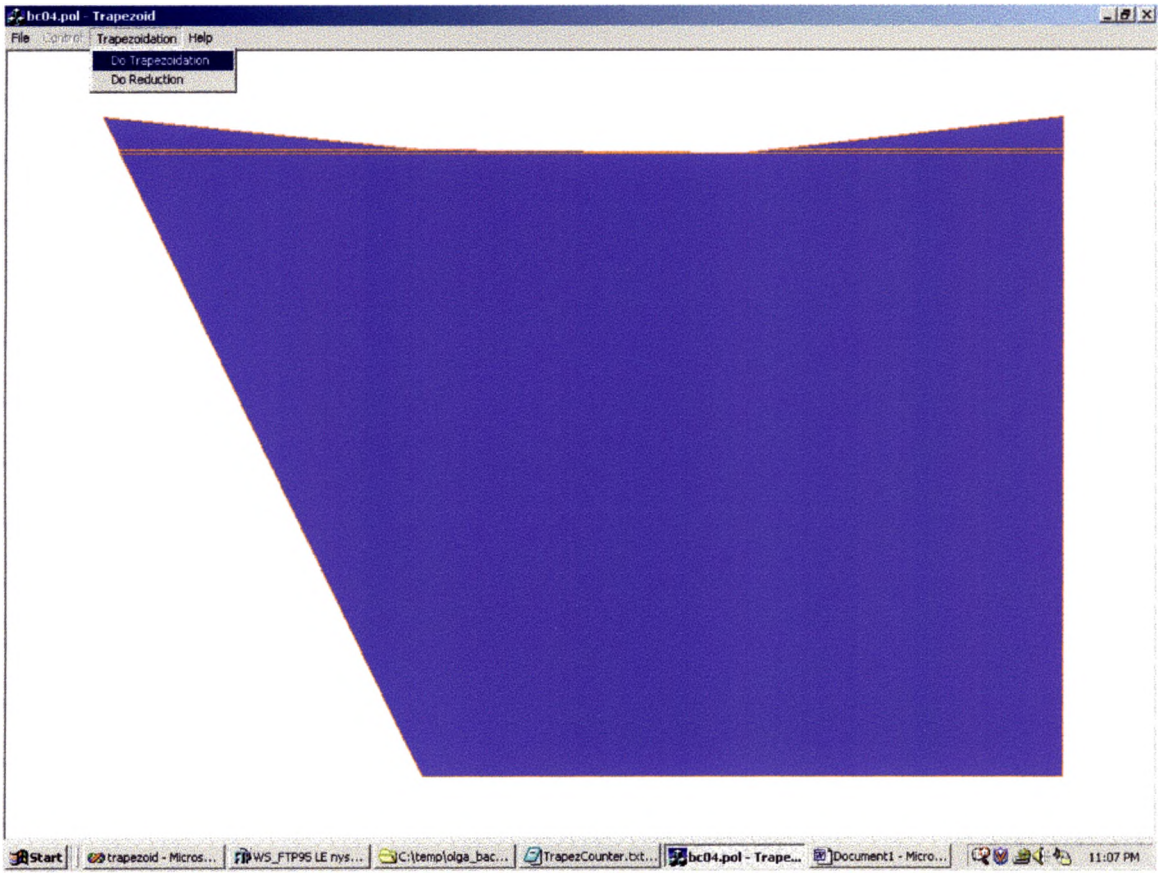
315.000000 162.000000
287.000000 181.000000
298.000000 217.000000
298.000000 237.000000
295.000000 252.000000
285.000000 271.000000
265.000000 283.000000
235.000000 289.000000
222.000000 304.000000
223.000000 319.000000
230.000000 336.000000
270.000000 356.000000
283.000000 370.000000
290.000000 388.000000
290.000000 403.000000
270.000000 435.000000
263.000000 448.000000
248.000000 461.000000
206.0456.000000

OUTPUT
TrapezCount.txt:

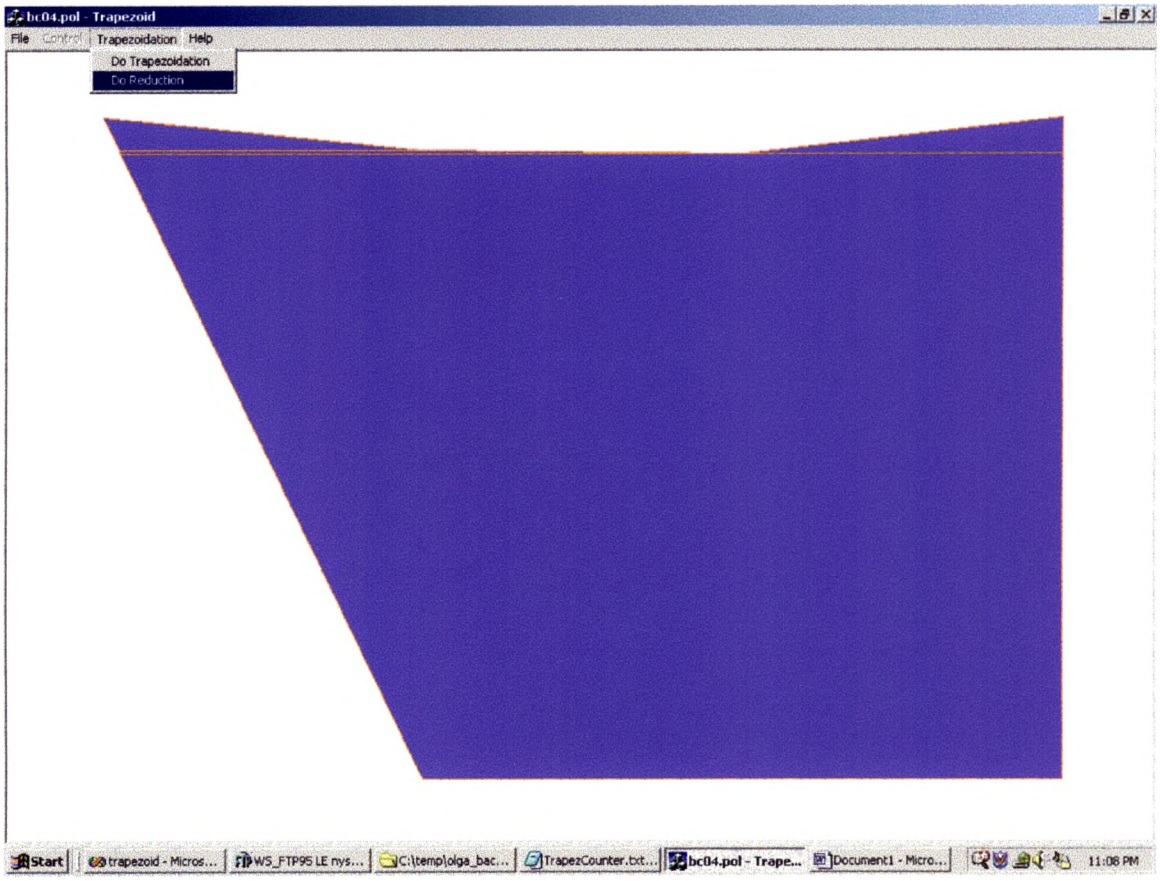
No of generated trapezoids 417
No of erased trapezoids 283
No of trapezoids after reduction 134

Testcase 2
bc04.pol

Geometry after trapezoidation.



Geometry after reduction



Input file:

bc04.pol:

****

L 1
R 0
L
7
3 2
3 0
2 0.11

1 0.1
0.0 0.00000000000000
1 2
2 1.9999999

OUTPUT:
TrapezCounre.txt:
No of generated trapezoids 7
No of erased trapezoids 1
No of trapezoids after reduction 6

Atrapezoid.txt:
(after reduction)

0.000000000000000, 0.000000000000000
0.050000000000000, 0.100000000000000
1.000000000000000, 0.100000000000000
0.000000000000000, 0.000000000000000

ring -1

3.000000000000000, 0.000000000000000
2.000000000000000, 0.110000000000000
3.000000000000000, 0.110000000000000
3.000000000000000, 0.000000000000000

ring -1

0.050000000000000, 0.100000000000000
0.055000000000000, 0.110000000000000
2.000000000000000, 0.110000000000000
1.000000000000000, 0.100000000000000

ring -1

0.055000000000000, 0.110000000000000
0.999999950000000, 1.999999900000000
3.000000000000000, 1.999999900000000
3.000000000000000, 0.110000000000000

ring -1

2.000000000000000, 1.999999900000000
3.000000000000000, 2.000000000000000
3.000000000000000, 2.000000000000000
3.000000000000000, 1.999999900000000

ring -1

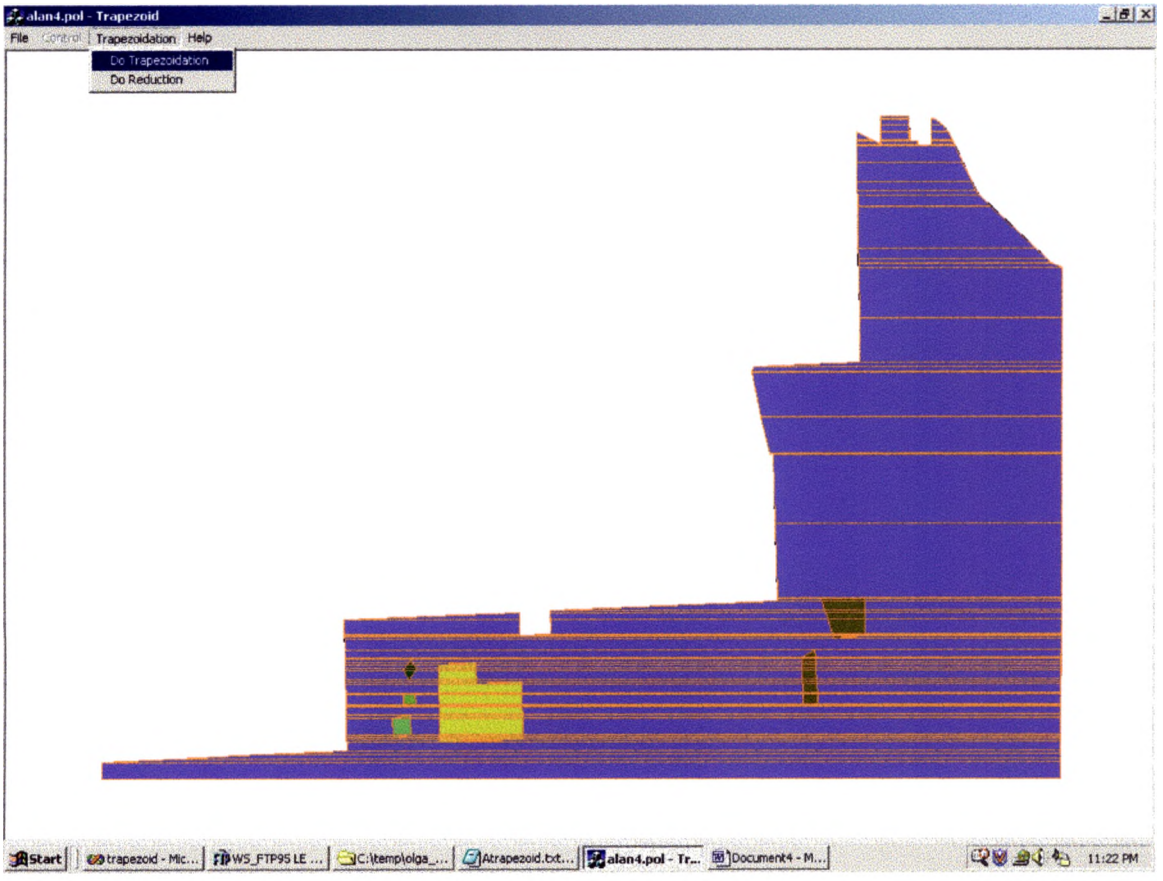
0.999999950000000, 1.999999900000000
1.000000000000000, 2.000000000000000
1.000000000000000, 2.000000000000000
2.000000000000000, 1.999999900000000

ring -1

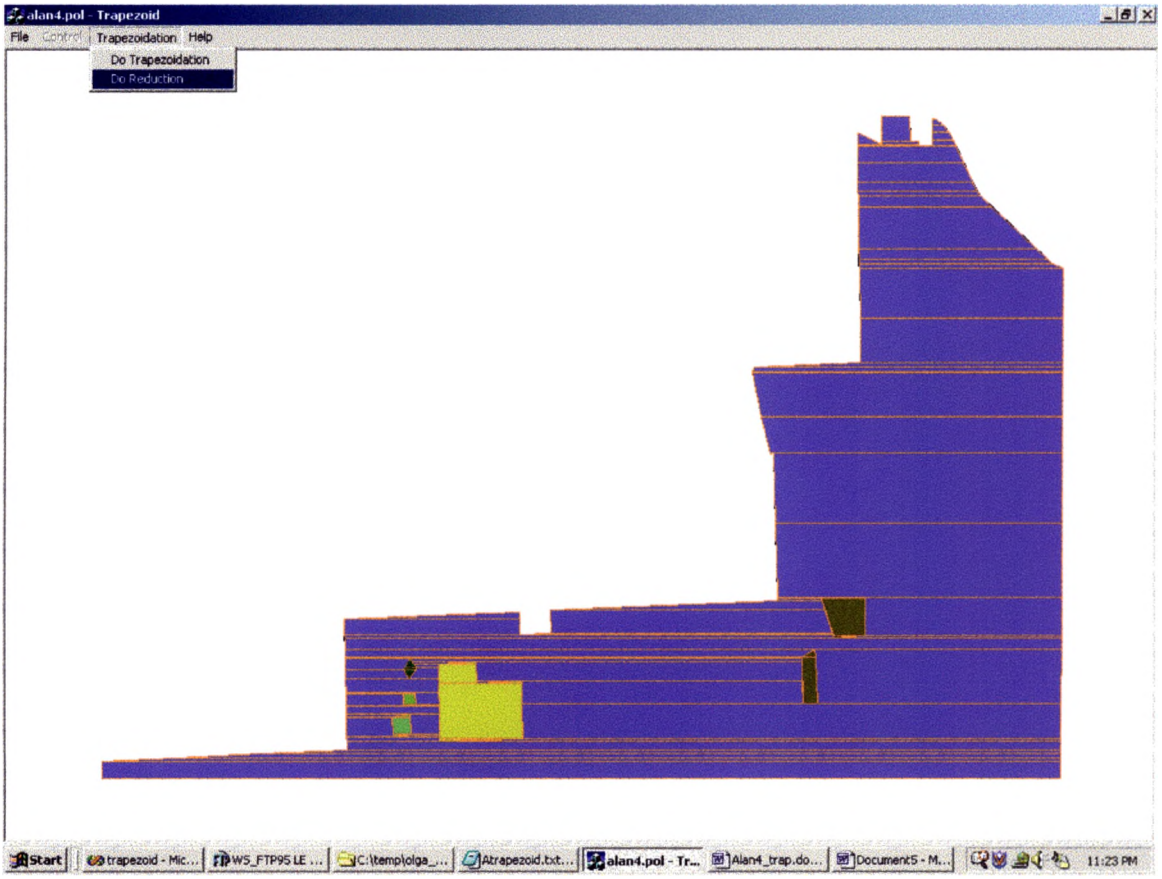
Number of polygons is: 6

Testcase 3
Alan4.pol

Geometry after trapezoidation



Geometry after reduction.



Input data:

Alan4.txt

L 1
R 6
L
43
4303825.000000 304310.000000
4303955.000000 304315.000000
4303950.000000 304356.000000
4304154.000000 304356.000000
4304158.000000 304091.000000

4304228.000000 304102.000000
4304330.000000 304157.000000
4304413.000000 304218.000000
4304487.000000 304301.000000
4304783.000000 304722.000000
4304862.000000 304815.000000
4304913.000000 304861.000000
4305936.000000 305491.000000
4306011.000000 305533.000000
4306189.000000 305581.000000
4306189.000000 308117.000000
4306167.000000 310674.000000
4291427.000000 310674.000000
4291427.000000 310504.000000
4292569.000000 310474.000000
4293866.000000 310425.000000
4295196.000000 310390.000000
4295152.000000 309086.000000
4297841.000000 309008.000000
4297852.000000 309239.000000
4298329.000000 309223.000000
4298320.000000 308988.000000
4301805.000000 308904.000000
4301746.000000 307425.000000
4301695.000000 307429.000000
4301551.000000 307062.000000
4301412.000000 306615.000000
4301410.000000 306612.000000
4301451.000000 306558.000000
4303075.000000 306517.000000
4303064.000000 306077.000000
4303045.000000 305392.000000
4303033.000000 304966.000000
4303021.000000 304525.000000
4303021.000000 304225.000000
4303380.000000 304343.000000
4303376.000000 304069.000000
4303816.000000 304065.000000
R

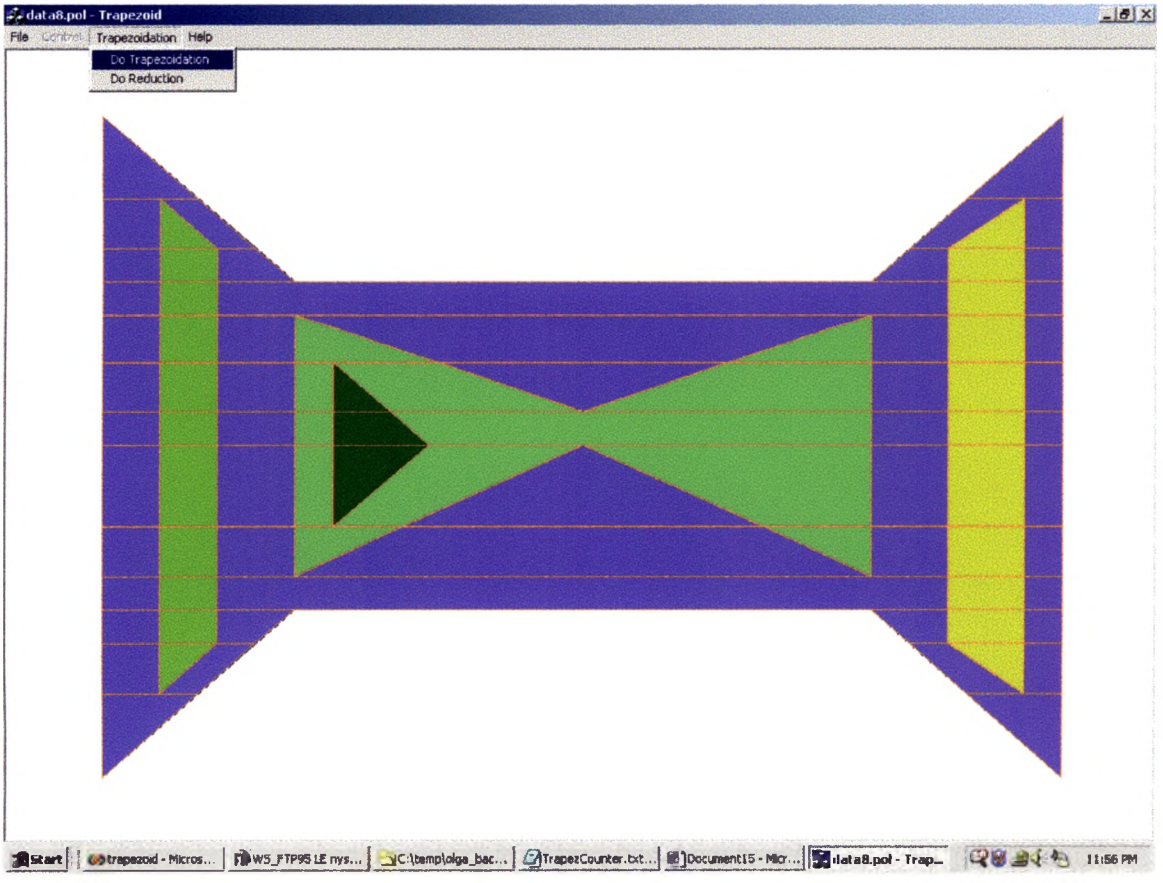
4
4295892.000000 310063.000000
4295920.000000 310260.000000
4296194.000000 310232.000000
4296162.000000 310031.000000
R
4
4296065.000000 309826.000000
4296073.000000 309951.000000
4296250.000000 309943.000000
4296246.000000 309810.000000
R
6
4296602.000000 309528.000000
4296625.000000 310306.000000
4297901.000000 310276.000000
4297882.000000 309703.000000
4297184.000000 309727.000000
4297180.000000 309506.000000
R
4
4296073.000000 309585.000000
4296158.000000 309677.000000
4296266.000000 309569.000000
4296178.000000 309472.000000
R
4
4302190.000000 309462.000000
4302206.000000 309934.000000
4302438.000000 309927.000000
4302382.000000 309382.000000
R
4
4302490.000000 308883.000000
4302677.000000 309272.000000
4303161.000000 309251.000000
4303157.000000 308862.000000

OUTPUT:
TrapezCounter.txt
****
**
No of generated trapezoids 191
No of erased trapezoids 99
No of trapezoids after reduction 92

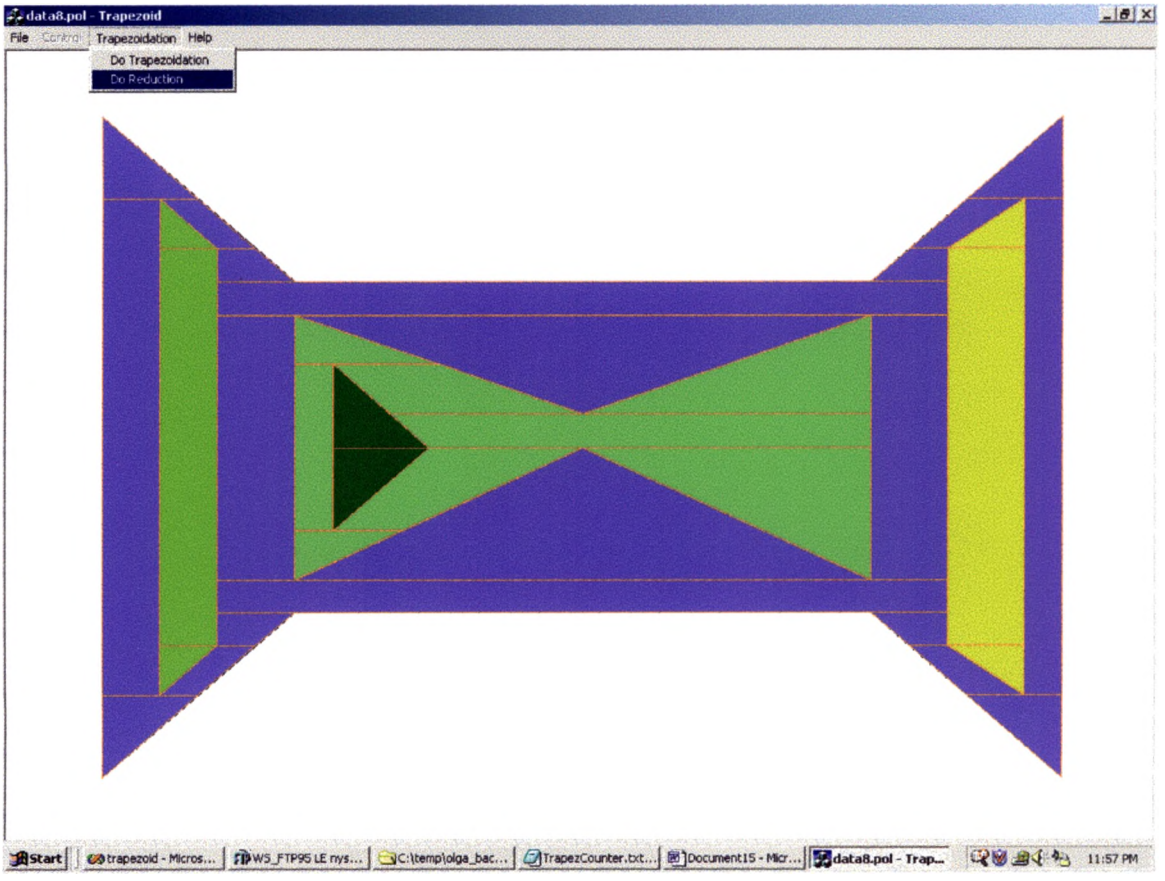
Testcase 4

data8.pol

Geometry after trapezoidation



Geometry after reduction.



Input data:

Data8.pol:

****

L 1
R 4
L
8
100.0 100.0
200.0 200.0

500.0 200.0
600.0 100.0
600.0 500.0
500.0 400.0
200.0 400.0
100.0 500.0
R
6
200.0 220.0
350.0 280.0
500.0 220.0
500.0 380.0
350.0 300.0
200.0 380.0
R
4
130.0 150.0
160.0 180.0
160.0 420.0
130.0 450.0
R
4
580.0 150.0
540.0 180.0
540.0 420.0
580.0 450.0
R
3
220.0 250.0
270.0 300.0
220.0 350.0

OUTPUT
TrapezCount.txt:
****

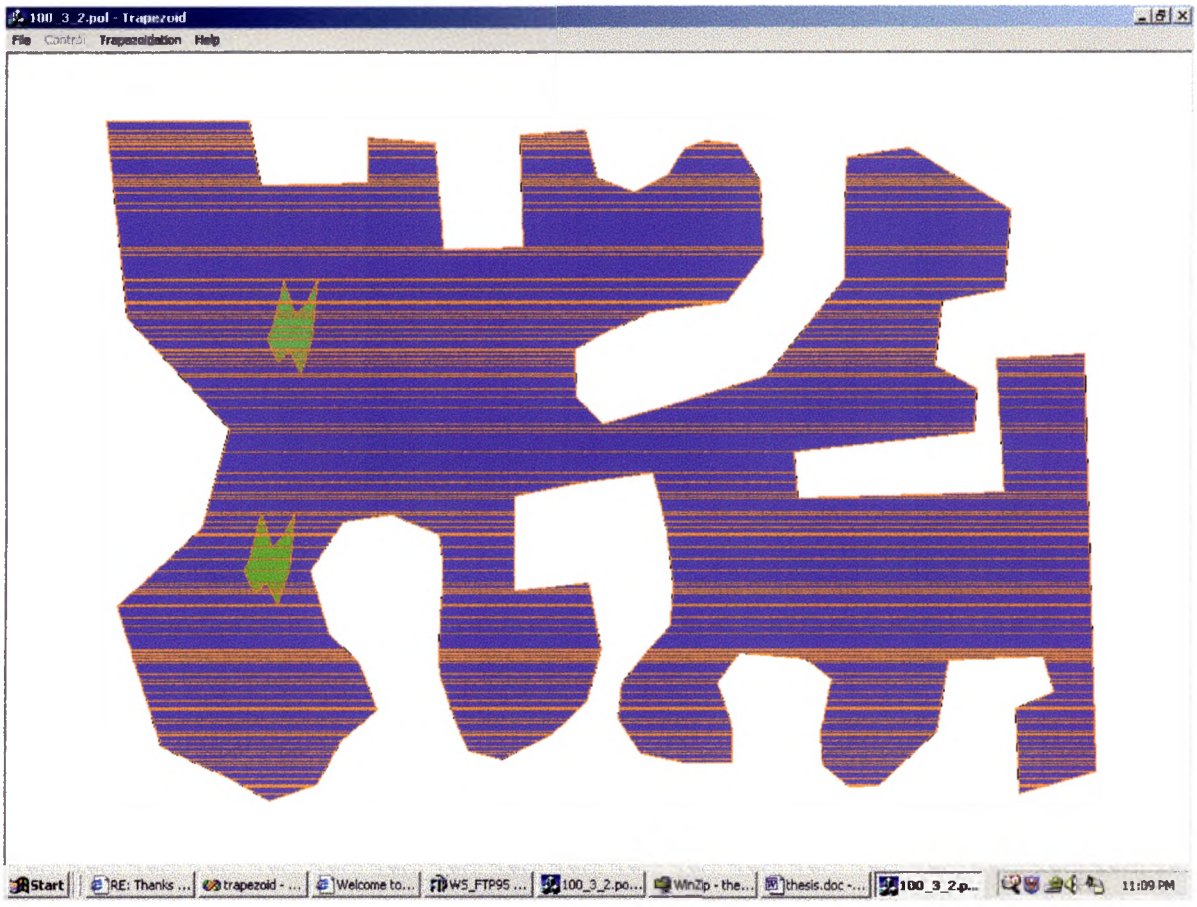
No of generated trapezoids 87

No of erased trapezoids 51
No of trapezoids after reduction 36

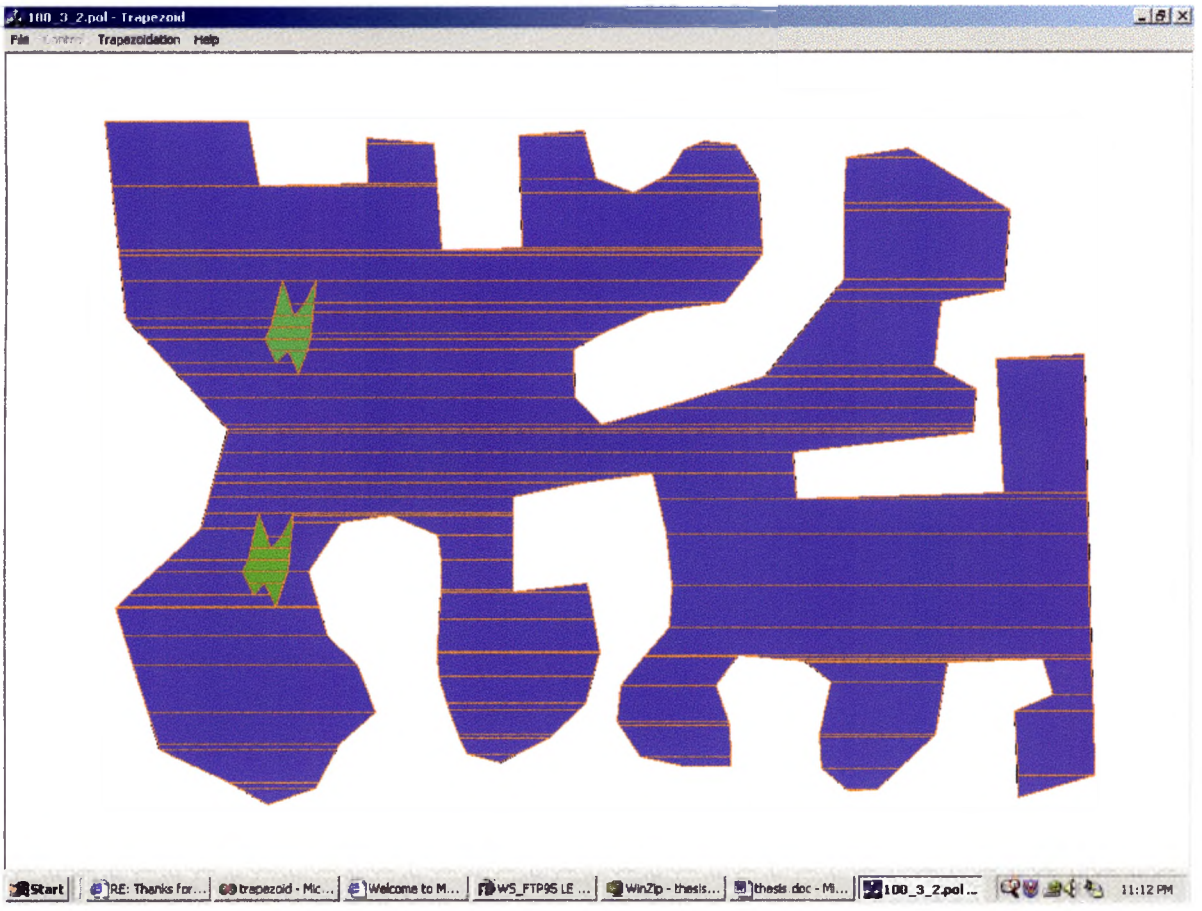
Testcase 5

100_3_2.pol

Geometry after trapezoidation



Geometry after reduction



Input data

100_3_2.pol
****
L 1
R 2
L
99
107.000000 553.000000
67.000000 431.000000
143.000000 363.000000
167.000000 277.000000
76.000000 182.000000
57.000000 14.000000

184.000000 14.000000
194.000000 69.000000
290.000000 67.000000
289.000000 28.000000
349.000000 33.000000
357.000000 124.000000
428.000000 122.000000
425.000000 26.000000
483.000000 22.000000
494.000000 63.000000
527.000000 75.000000
557.000000 60.000000
571.000000 38.000000
590.000000 31.000000
618.000000 34.000000
638.000000 65.000000
642.000000 128.000000
609.000000 169.000000
542.000000 177.000000
500.000000 195.000000
474.000000 209.000000
475.000000 250.000000
499.000000 273.000000
552.000000 259.000000
644.000000 232.000000
714.000000 149.000000
714.000000 84.000000
716.000000 45.000000
771.000000 37.000000
861.000000 90.000000
856.000000 158.000000
800.000000 168.000000
794.000000 223.000000
831.000000 243.000000
829.000000 280.000000
669.000000 297.000000
672.000000 338.000000
856.000000 333.000000
849.000000 217.000000

927.000000 213.000000
938.000000 575.000000
870.000000 594.000000
867.000000 520.000000
901.000000 506.000000
891.000000 475.000000
807.000000 478.000000
796.000000 540.000000
745.000000 587.000000
719.000000 588.000000
695.000000 569.000000
693.000000 542.000000
703.000000 495.000000
680.000000 477.000000
621.000000 472.000000
601.000000 498.000000
614.000000 532.000000
615.000000 567.000000
572.000000 567.000000
534.000000 559.000000
513.000000 529.000000
517.000000 498.000000
537.000000 473.000000
560.000000 448.000000
562.000000 412.000000
557.000000 368.000000
544.000000 316.000000
483.000000 324.000000
421.000000 336.000000
421.000000 418.000000
487.000000 410.000000
498.000000 469.000000
486.000000 513.000000
454.000000 543.000000
409.000000 564.000000
380.000000 557.000000
372.000000 540.000000
364.000000 519.000000
354.000000 490.000000

353.000000 470.000000
355.000000 441.000000
357.000000 416.000000
357.000000 390.000000
353.000000 369.000000
311.000000 352.000000
268.000000 358.000000
239.000000 400.000000
255.000000 455.000000
282.000000 480.000000
299.000000 521.000000
267.000000 548.000000
245.000000 586.000000
203.000000 600.000000
169.000000 581.000000
R
10
200 200
210 220
220 210
230 230
240 200
245 150
235 170
225 180
215 150
205 190
R
10
180 400
190 420
200 410
210 430
220 400
225 350
215 370
205 380
195 350
185 390

OUTPUT:
TrapexCounter.txt:

No of generated trapezoids 462
No of erased trapezoids 332
No of trapezoids after reduction 130

....

ring -1

385.000000000000000, 582.000000000000000
396.375000000000000, 596.000000000000000
426.000000000000000, 596.000000000000000
452.352941176470610, 582.000000000000000

ring -1

396.375000000000000, 596.000000000000000
398.000000000000000, 598.000000000000000
398.000000000000000, 598.000000000000000
426.000000000000000, 596.000000000000000

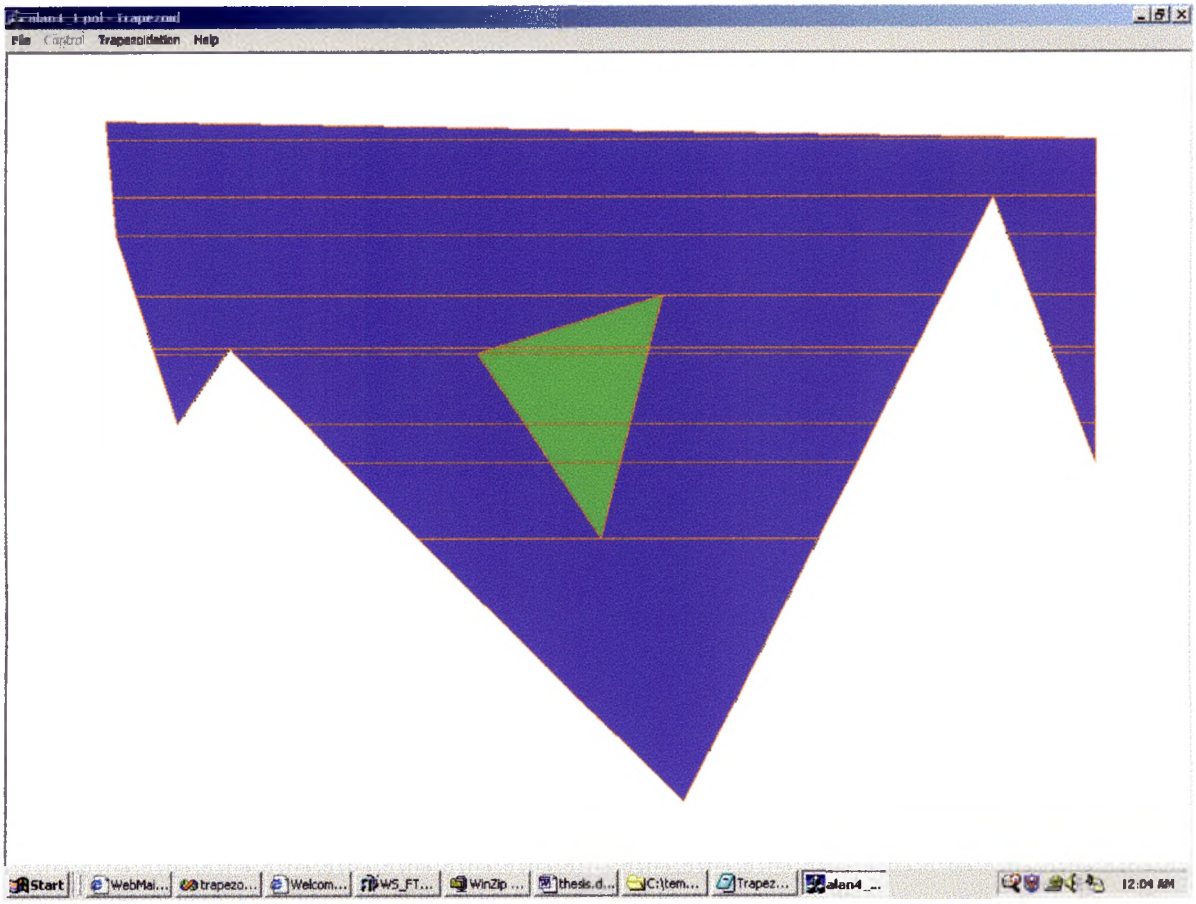
ring -1

Number of polygons is: 130

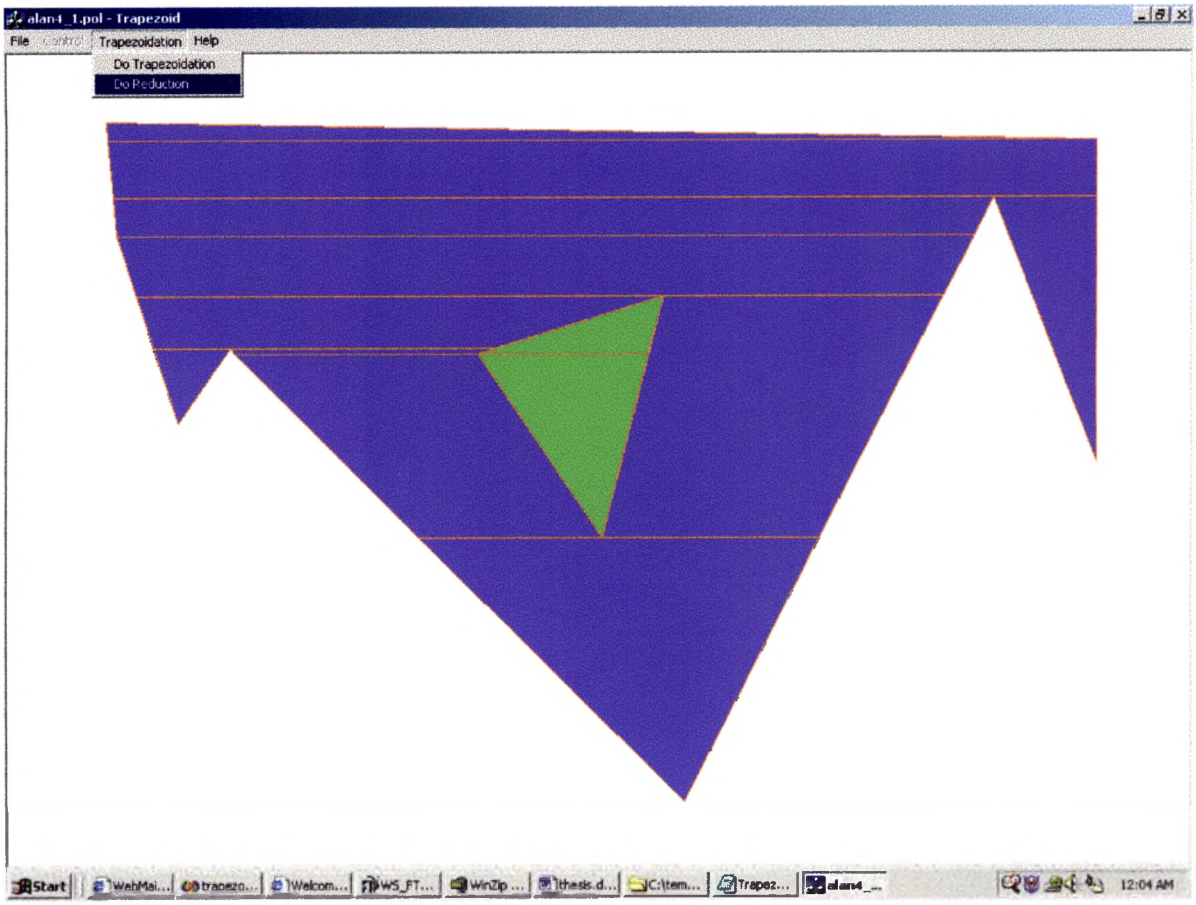
Testcase 6

Alan4_1

Geometry after trapezoidation:



Geometry after reduction:



Input data:

****
****
L 1
R 1
L
8
25.000000 100.000000
55.000000 200.000000
80.000000 160.000000
300.000000 400.000000
450.000000 80.000000

```
500.000000 220.000000
500.000000 50.000000
20.000000  40.000000
R
3
200.000000  163.000000
260.000000  260.000000
290.000000  132.000000
```

OUTPUT:

TrapezCount.txt:

No of generated trapezoids 28
No of erased trapezoids 15
No of trapezoids after reduction 13

Atrapezoid.txt:

```
171.666666666666660,  260.000000000000000
 300.000000000000000,  400.000000000000000
 300.000000000000000,  400.000000000000000
 365.625000000000000,  260.000000000000000

ring -1
-----------------------
  20.000000000000000,   40.000000000000000
  20.833333333333332,   50.000000000000000
 500.000000000000000,   50.000000000000000
  20.000000000000000,   40.000000000000000

ring -1
-----------------------
  20.833333333333332,   50.000000000000000
  23.333333333333332,   80.000000000000000
 500.000000000000000,   80.000000000000000
 500.000000000000000,   50.000000000000000

ring -1
-----------------------
```

```
 23.333333333333332,  80.000000000000000
 25.000000000000000, 100.000000000000000
440.625000000000000, 100.000000000000000
450.000000000000000,  80.000000000000000

ring -1
----------------------
450.000000000000000,  80.000000000000000
500.000000000000000, 220.000000000000000
500.000000000000000, 220.000000000000000
500.000000000000000,  80.000000000000000

ring -1
----------------------
 25.000000000000000, 100.000000000000000
 34.600000000000001, 132.000000000000000
425.625000000000000, 132.000000000000000
440.625000000000000, 100.000000000000000

ring -1
----------------------
 34.600000000000001, 132.000000000000000
 43.000000000000000, 160.000000000000000
208.709677419354850, 160.000000000000000
290.000000000000000, 132.000000000000000

ring -1
----------------------
290.000000000000000, 132.000000000000000
260.000000000000000, 260.000000000000000
365.625000000000000, 260.000000000000000
425.625000000000000, 132.000000000000000

ring -1
----------------------
290.000000000000000, 132.000000000000000
200.000000000000000, 163.000000000000000
282.734375000000000, 163.000000000000000
290.000000000000000, 132.000000000000000
```

ring 0

43.000000000000000, 160.000000000000000
55.000000000000000, 200.000000000000000
55.000000000000000, 200.000000000000000
80.000000000000000, 160.000000000000000

ring -1

80.000000000000000, 160.000000000000000
82.750000000000000, 163.000000000000000
200.000000000000000, 163.000000000000000
208.709677419354850, 160.000000000000000

ring -1

82.750000000000000, 163.000000000000000
171.666666666666660, 260.000000000000000
260.000000000000000, 260.000000000000000
200.000000000000000, 163.000000000000000

ring -1

200.000000000000000, 163.000000000000000
260.000000000000000, 260.000000000000000
260.000000000000000, 260.000000000000000
282.734375000000000, 163.000000000000000

ring 0

Number of polygons is: 13

APPENDIX B

C++ source code

Trapez.h and Trapez.cpp files are displayed. The complete C++ project with all classes and methods is attached on CD.

Trapez.h

```
// Declaration of trapezoidation and reduction algorithms.
// Header file.
#if
!defined(AFX_TRAPEZ_H__A9D0439B_54C2_11D2_9591_000000000000__IN
CLUDED_)
#define
AFX_TRAPEZ_H__A9D0439B_54C2_11D2_9591_000000000000__INCLUDED
_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define USED 1
#define NOT_USED 0
#define LOOP_POINT 1
#define RING_POINT 2
#define INTERSECTION 3
#define LOCAL_MINIMUM 4
#define LOCAL_MAXIMUM 5
#define HORIZONTAL_SEGMENT 6
#define NONE -1

typedef struct {
  double x, y;
} PointInf;

typedef struct {
  double x, y;
```

```
 int Type;
 int PNo;
 int Ring;                          // -1 if intersects main loop, ring no.
otherwise
} IntersectionType;


typedef struct {
 double y;
 int PNo;                           // point number
 int i1, i2;                        // index of edges
 short int Used1, Used2;            // 1 if edge (PNo, i1) or (PNo,i2) is already
used
} YCoord;

struct Ttype{
 double x[4];
 double y[4];
 int Ring;                          // ring number
 struct Ttype* Next;
};                                        // structure Ttype for trapezoids

typedef struct Ttype TrapezType;

class Trapez
{
private:
int NoL;                            // no of points in loop
int NoRings;                        // no of rings
int* NoR;                           // array of no of points in rings
int Total;                          // total number of points
int ProcessedPointsIndex;
int IntersectionNoX;                // number of intersections in X direction
int RingsReached;                   // number of rings in treatment
int AllocateFlag;
double xmin, ymin, xmax, ymax;      // bounding box coordinates - just for
plotting
double MyEpsilon;                   // tolerance when two lines are colinear
```

```
TrapezType *TrapezPointer, *TrapezPointerTail;
TrapezType* tptr;
PointInf* InputPoints;                  // array of input points
YCoord* SortedPointsY;                  // array of sorted points

IntersectionType* OldIntersections;  // array of previous intersections
IntersectionType* Intersections;     // array of actual intersections
IntersectionType** OldRingIntersections;
IntersectionType** NewRingIntersections;
IntersectionType* UsefulIntersections;
IntersectionType* UsefulRingIntersections;


int Mirror_Total;
int Mirror_NoL;
int Mirror_NoRings;
int TrapezCounter, ErasedTrapezoids;

void MirrorInputData();
void ReleaseMirrorData();
void RestoreFromMirror();
void Deallocate();

PointInf* Mirror_InputPoints;
int* Mirror_NoR;

void MainAllocate();
void SortByY();
int LocalMinimum(int);
int LocalMaximum(int);
int LSMaximum(int p);
int HorizontalSegment(int p);
int HorizontalMiddle(int p);

void SortIntersectionsX();
void DetermineNeighbours();
void InitOldBuffer();
void PointsWithTheSameY(int &i1, int &i2);
void CalculateIntersections(int p, int q);
```

```
void MakeTrapezodiation();
int IntersectionWithHLine(double y1, double x3, double y3, double a, double
b,
        double &outx, double &outy);
void MakeTrapezs();
void UpdateOldIntersections();
int WhichRing(int p);
void MarkUsedEdges(int p, int q);
int WhereToPlace(int i);
void FillOldRingIntersections(IntersectionType**, int i, int r1, int* jr);
int FindEncloseingRing(int b1, int b2);
void Add(TrapezType* t);
int TrapezsCanBeJoined(TrapezType* t1, TrapezType* t2, double
&ERROR_ACCUM);
//determines if trapezoids may be joined
void ArrangeTouchingRingpoints();
void CorrectURI(int n);
void SortURIByX(int i1, int i2);
void ClearTrapezoids();
void PointBetween(PointInf p1, PointInf p2, PointInf* p3);

void QSortByY(int iLo, int iHi) ;
void QSortByX(int DnoIn, int VrhIn);
void DetermineEpsilion();

public:
 Trapez(void);
 virtual ~Trapez(void);
 void OptimizeTrapezs();          //function to optimize trapezoids
 int Load(char*);
 int LoadPolygon(char* fname);
 void SetPolygon(int NoOfVertices, int NoOfHoles, int* VerticesInHoles,
            double* xarr, double* yarr);
 void Trapezoidation();
 void Plot(HDC);
 void Plot(HDC h, int xw, int yw);
 void PlotTrapez(HDC);
 void PlotTrapez(HDC h, int xw, int yw);
 TrapezType ReturnTrapez();
```

```
  void ReturnTrapezInit();
  void SplitPoints();
  int ReturnTrapezCounter() {return TrapezCounter;}
  int ReturnErasedTrapezoids() {return ErasedTrapezoids;}
  int ReturnNoOfTrapezoids() {return(TrapezCounter - ErasedTrapezoids);}
  TrapezType* ReturnTrapezPointer() {return TrapezPointer;}

  void EraseWindow(HDC hdc, HWND hWnd);
  void PrintListOfTrapezoids();
};


#endif
```

Trapez.cpp

```
// implementation of trapezoidation algorithm
// it performs horizontal trapezoidation of non/monotone polygons
containing
// nested holes. The holes are trapezaoidated as well. Polygon edges should
not
// intersect. The holes can thouch each other, but should not thouch the loop
// (the polygon border). Trapezoidation algorithm proposed by Borut Zalik.
// Trapezoid reduction algorithm developed by Olga Zaporojets is
implemented.

#include "stdafx.h"
#include "trapez.h"
#include "utility.h"
#include "myCPUTime.h"
#include "Line2D.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef _DEBUG
#define new DEBUG_NEW
```

```
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define shareware

#ifdef shareware
#include <windows.h>
#endif

Trapez::Trapez(void)
//Constructor.
{ Total = NoRings = ProcessedPointsIndex = 0;
  RingsReached = NONE;
  TrapezPointer = NULL;
  TrapezCounter = ErasedTrapezoids = 0;
  AllocateFlag = 0;
}

Trapez::~Trapez(void)
//Destructor.
{
  if (AllocateFlag == 1)
  {
    delete[] InputPoints;
    delete[] SortedPointsY;
    delete[] OldIntersections;
    delete[] Intersections;
    delete[] UsefulIntersections;
    delete[] UsefulRingIntersections;
    delete[] Mirror_InputPoints;
    delete[] Mirror_NoR;
    delete[] NoR;

    if (NoRings > 0)
    { for (int i = 0; i < NoRings; i++)
          { delete[] OldRingIntersections[i];
                delete[] NewRingIntersections[i];
          }
```

```
            delete[] OldRingIntersections;
            delete[] NewRingIntersections;
    }
    ClearTrapezoids();
  }
}

void Trapez::DetermineEpsilion()
//Olga Zaporojets: set the error threshold
{

  xmin = ymin = MAX_DOUBLE;
  xmax = ymax = -MAX_DOUBLE;

  int i;

  for (i = 0; i < NoL; i++)
  { if (xmin > InputPoints[i].x) xmin = InputPoints[i].x;
    if (ymin > InputPoints[i].y) ymin = InputPoints[i].y;
    if (xmax < InputPoints[i].x) xmax = InputPoints[i].x;
    if (ymax < InputPoints[i].y) ymax = InputPoints[i].y;
  }

  double dx = (xmax - xmin) * 0.1;
  double dy = (ymax - ymin) * 0.1;

  xmin -= dx;
  ymin -= dy;
  xmax += dx;
  ymax += dy;

  MyEpsilon = 0.5 * EPSILON * (dx + dy);
  if (MyEpsilon < VERY_SMALL_EPSILON)
    MyEpsilon = VERY_SMALL_EPSILON;
}


void Trapez::ClearTrapezoids()
//delete a list of trapezoids.
```

```
{ TrapezType* a;

  while (TrapezPointer != NULL)
  { a = TrapezPointer;
    TrapezPointer= TrapezPointer->Next;
    delete a;
  }
  TrapezPointerTail = TrapezPointer = NULL;}
void Trapez::Deallocate()
//free memory.
{ delete[] NoR;
  delete[] TrapezPointer;
  delete[] InputPoints;
  delete[] SortedPointsY;
  delete[] OldIntersections;
  delete[] Intersections;
  delete[] UsefulIntersections;
  delete[] UsefulRingIntersections;

  if (NoRings > 0)
  { for (int i = 0; i < NoRings; i++)
        { delete[] OldRingIntersections[i];
              delete[] NewRingIntersections[i];
        }
        delete[] OldRingIntersections;
        delete[] NewRingIntersections;
  }
}


int Trapez::Load(char* Fname)
// File has to have the following structure:
//  Number of points in a loop
//  (x, y) ordered coordinates of loop points
//  number of rings
//  { numer of points in i-th ring
//  ordered coordinates of i-th ring points }
{ FILE* f;
```

```
if ((f = fopen(Fname, "rt")) == NULL) return 0;

double x, y;
int i, kumulacija = 0;

fscanf(f, "%d", &NoL);
NoL--;

PointInf* A = new PointInf[10000];

for (i=0; i < NoL; i++)  // we do not duplicate last point
{ fscanf(f, "%lf %lf", &x, &y);
        A[i].x = x;
        A[i].y = y;
        kumulacija++;
}
fscanf(f, "%lf %lf", &x, &y);  // read last duplicated point

fscanf(f, "%d", &NoRings);   // get number of rings
int r = 0;
       NoR = new int[NoRings+10];
       while (fscanf(f, "%d", &NoR[r]) != EOF)
       { NoR[r]--;
        for (i=0; i < NoR[r]; i++)    // we do not duplicate last point
        { fscanf(f, "%lf %lf", &x, &y);
                A[kumulacija].x = x;
                A[kumulacija].y = y;
                kumulacija++;
        }
        r++;
        fscanf(f, "%lf %lf", &x, &y);  // read last duplicated point
       }

       Total = NoL;
       for (i = 0; i < NoRings; i++)
        Total += NoR[i];

       InputPoints = new PointInf[Total+10];
       for (i = 0; i < Total; i++)
```

```
        InputPoints[i] = A[i];

        MainAllocate();
        fclose(f);
        delete[] A;
        return 1;
}

void Trapez::SetPolygon(int NoOfVertices, int NoOfHoles, int*
VerticesInHoles,
                double* xarr, double* yarr)
{ Total = NoOfVertices;
 NoRings = NoOfHoles;
 NoR = new int[NoRings+10];

 InputPoints = new PointInf[Total+10];
 InputPoints = new PointInf[Total+10];

 int NoOfPointsInLoop = NoOfVertices;
 int i = 0;


 for (i = 0; i < Total; i++)
 { InputPoints[i].x = xarr[i];
  InputPoints[i].y = yarr[i];
 }

 if (NoOfHoles != 0)
 {
  for (i = 0; i < NoOfHoles; i++)
  { NoR[i] = VerticesInHoles[i];
  }
 }

 NoL = Total;
 for (i = 0; i < NoRings; i++)
  NoL -= NoR[i];
  MainAllocate();
}
```

```
int Trapez::LoadPolygon(char* fname)
// File has to have the following structure:
// Number of points in a loop
// (x, y) ordered coordinates of loop points
// number of rings
// { numer of points in i-th ring
// ordered coordinates of i-th ring points }
{ FILE* f;
#define LOOP 1
#define RING 0

 if ((f = fopen(fname, "rt")) == NULL) return 0;

 char c[2];
 c[1] = '\0';
 int No1;
 int No;

 fscanf(f, "%s %d", &c, &No1);      // number of loops
 fscanf(f, "%s %d", &c, &NoRings);  // numer of rings

 NoR = new int[NoRings+10];
 PointInf* A = new PointInf[10000];

 int LoopRingFlag;
 int r = 0;

 int i, j, k, w;
 w = 0;

  for (j = 0; j < (No1 + NoRings); j++)
  { fscanf(f, "%s", &c);
   if (strcmp(c, "L") == 0) LoopRingFlag = LOOP;
   else LoopRingFlag = RING;
   fscanf(f, "%d", &No);

   if (LoopRingFlag == LOOP)
    NoL = No;
```

```
    if (LoopRingFlag == RING)
      NoR[r++] = No;

    for (k = 0; k < No; k++)
    { fscanf(f, "%lf %lf", &A[w].x, &A[w].y);
      w++;
    }
   }

        Total = NoL;
        for (i = 0; i < NoRings; i++)
         Total += NoR[i];
        InputPoints = new PointInf[Total+10];
        for (i = 0; i < Total; i++)
         InputPoints[i] = A[i];

     delete []A;
        MainAllocate();
        fclose(f);
        return 1;
}


void Trapez::SplitPoints()
//Create split points.
{ int Total2 = 2 * Total;
 int NoL2 = 2*NoL;
 int i;
 for (i = 0; i < NoRings; i++)
     NoR[i] = 2 * NoR[i];

 PointInf* InputPoints2 = new PointInf[Total+10];

 int j = 0;
 int k, w;
 for (i = 0 ; i < NoL; i++)
 { InputPoints2[j] = InputPoints[i];
   ++j;
```

```
   PointBetween(InputPoints[i], InputPoints[i+1], &InputPoints2[j]);
   j++;
  }

  w = NoL;
  for (k = 0; k < NoRings; k++)
   for (i = 0; i < NoR[i]; i++)
   { InputPoints2[j] = InputPoints[w];
    j++;
    PointBetween(InputPoints[w], InputPoints[w+1], &InputPoints2[j]);
    j++;
    w++;
   }

  Deallocate();
  InputPoints = new PointInf[Total+10];
  for (i = 0; i < Total; i++)
   InputPoints[i] = InputPoints2[i];
  delete InputPoints2;
  MainAllocate();
}



void Trapez::PointBetween(PointInf p1, PointInf p2, PointInf* p3)
//p3 will be between p1 and p2.
{ Point pout;

  Line2D* l = new Line2D(p1.x, p1.y, p2.x, p2.y);
  l->PointOnLine(0.5, pout);
  p3->x = pout.x; p3->y = pout.y;
}

void Trapez::MirrorInputData()
{ Mirror_Total = Total;
  Mirror_NoL = NoL;
  Mirror_NoRings = NoRings;

  int i;
```

```
 for (i = 0; i < Total; i++)
  Mirror_InputPoints[i] = InputPoints[i];

 Mirror_NoR = new int[NoRings+10];
 for (i = 0; i < NoRings; i++)
  Mirror_NoR[i] = NoR[i];

}

void Trapez::RestoreFromMirror()
//Original data will be assigned to the mirrored data.
{
 Total = Mirror_Total;
 NoL = Mirror_NoL;
 NoRings = Mirror_NoRings;

 int i;
 for (i = 0; i < Total; i++)
  InputPoints[i] = Mirror_InputPoints[i];

  for (i = 0; i < NoRings; i++)
  NoR[i] = Mirror_NoR[i];

}



void Trapez::ReleaseMirrorData()
//Mirror data will be deleted.
{
 delete[] Mirror_InputPoints;
 delete[] Mirror_NoR;
}


void Trapez::MainAllocate()
// Create data structure.
{ int a = 10;
 AllocateFlag = 1;
```

```
        SortedPointsY = new YCoord[Total+ a];
        OldIntersections = new IntersectionType[Total + a];
        Intersections = new IntersectionType[Total];
        UsefulIntersections = new IntersectionType[Total + a];
        UsefulRingIntersections = new IntersectionType[Total + a];

        if (NoRings > 0)
        { OldRingIntersections = new IntersectionType*[NoRings+ a];
         for (int i = 0; i < NoRings; i++)
                OldRingIntersections[i] = new IntersectionType[Total + a];
        }
        if (NoRings > 0)
        { NewRingIntersections = new IntersectionType*[NoRings+ a];
         for (int i = 0; i < NoRings; i++)
                NewRingIntersections[i] = new IntersectionType[Total + a];
        }

 Mirror_InputPoints = new PointInf[Total+a];
 Mirror_NoR = new int[NoRings+a];
}



void Trapez::Plot(HDC h)
// Plot the polygone for the graphical output
{ POINT* plxy = new POINT[NoL+1];

 int i, LastUsed;
 for (i = 0; i < NoL; i++)
 { plxy[i].x = (long) InputPoints[i].x;
        plxy[i].y = (long) InputPoints[i].y;
 }
 plxy[i].x = (long) InputPoints[0].x;
 plxy[i].y = (long) InputPoints[0].y;

 LastUsed = NoL;

 Polyline(h, plxy, NoL+1);
 delete plxy;
```

```
 int k;

 for (k = 0; k < NoRings; k++)
 { plxy = new POINT[NoR[k]+1];
        int j = 0;
        for (i = LastUsed; i < LastUsed + NoR[k]; i++)
        { plxy[j].x = (long) InputPoints[i].x;
              plxy[j].y = (long) InputPoints[i].y;
              j++;
        }
        plxy[j].x = (long) InputPoints[LastUsed].x;
        plxy[j].y = (long) InputPoints[LastUsed].y;

        Polyline(h, plxy, NoR[k]+1);
        delete plxy;
        LastUsed += NoR[k];
 }
}


void Trapez::Plot(HDC h, int xw, int yw)
// Function for graphical output.
{ POINT* plxy = new POINT[NoL+1];

 int i, LastUsed;
 xmin = ymin = MAX_DOUBLE;
 xmax = ymax = -MAX_DOUBLE;

 for (i = 0; i < NoL; i++)
 { if (xmin > InputPoints[i].x) xmin = InputPoints[i].x;
   if (ymin > InputPoints[i].y) ymin = InputPoints[i].y;
   if (xmax < InputPoints[i].x) xmax = InputPoints[i].x;
   if (ymax < InputPoints[i].y) ymax = InputPoints[i].y;
 }

 double dx = (xmax - xmin) * 0.1;
 double dy = (ymax - ymin) * 0.1;
 xmin -= dx;
```

```
 ymin -= dy;
 xmax += dx;
 ymax += dy;


 for (i = 0; i < NoL; i++)
 { plxy[i].x = FromRealToPixel(InputPoints[i].x, xmin, xmax-xmin, xw);
   plxy[i].y = FromRealToPixel(InputPoints[i].y, ymin, ymax-ymin, yw);
 }
 plxy[i].x = plxy[0].x;
 plxy[i].y = plxy[0].y;

 LastUsed = NoL;

 Polyline(h, plxy, NoL+1);
 delete plxy;

 int k;

 for (k = 0; k < NoRings; k++)
 { plxy = new POINT[NoR[k]+1];
        int j = 0;

   for (i = LastUsed; i < LastUsed + NoR[k]; i++)
   { plxy[j].x = FromRealToPixel(InputPoints[i].x, xmin, xmax-xmin, xw);
     plxy[j].y = FromRealToPixel(InputPoints[i].y, ymin, ymax-ymin, yw);
     j++;
   }
   plxy[j].x = plxy[0].x;
   plxy[j].y = plxy[0].y;

   Polyline(h, plxy, NoR[k]+1);
        delete plxy;
        LastUsed += NoR[k];
 }
}


void Trapez::PlotTrapez(HDC h, int xw, int yw)
```

```
//Draws trapezoids.
{ POINT plxy[5];
 TrapezType *t = TrapezPointer;
 HPEN hpen, hpenOld;

 HBRUSH HBrush1 = CreateSolidBrush(RGB(0,0,255));
 HBRUSH HBrush2 = CreateSolidBrush(RGB(0,255,0));
 HBRUSH HBrush3 = CreateSolidBrush(RGB(100,200,0));
 HBRUSH HBrush4 = CreateSolidBrush(RGB(200,200,0));
 HBRUSH HBrush5 = CreateSolidBrush(RGB(30,100,0));
 HBRUSH HBrush6 = CreateSolidBrush(RGB(100,100,0));

 hpen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
 hpenOld = (HPEN)SelectObject(h, hpen);

 while (t != NULL)
 {
  plxy[0].x = FromRealToPixel(t->x[0], xmin, xmax-xmin, xw);
  plxy[0].y = FromRealToPixel(t->y[0], ymin, ymax-ymin, yw);
  plxy[1].x = FromRealToPixel(t->x[1], xmin, xmax-xmin, xw);
  plxy[1].y = FromRealToPixel(t->y[1], ymin, ymax-ymin, yw);
  plxy[2].x = FromRealToPixel(t->x[2], xmin, xmax-xmin, xw);
  plxy[2].y = FromRealToPixel(t->y[2], ymin, ymax-ymin, yw);
  plxy[3].x = FromRealToPixel(t->x[3], xmin, xmax-xmin, xw);
  plxy[3].y = FromRealToPixel(t->y[3], ymin, ymax-ymin, yw);

        switch (t->Ring){
                case NONE: (HBRUSH)SelectObject(h, HBrush1);break;
                case 0: (HBRUSH)SelectObject(h, HBrush2); break;
                case 1: (HBRUSH)SelectObject(h, HBrush3); break;
                case 2: (HBRUSH)SelectObject(h, HBrush4); break;
                case 3:  (HBRUSH)SelectObject(h, HBrush5); break;
                default: (HBRUSH)SelectObject(h, HBrush6);
        }

  Polygon(h, plxy, 4);

        t = t->Next;
}
```

```
 SelectObject(h, hpenOld);
 DeleteObject(hpen);
}



void Trapez::PlotTrapez(HDC h)
//Draws trapezoids.
{ POINT plxy[5];
 TrapezType *t = TrapezPointer;
 HPEN hpen, hpenOld;

 HBRUSH HBrush1 = CreateSolidBrush(RGB(0,0,255));
 HBRUSH HBrush2 = CreateSolidBrush(RGB(0,255,0));
 HBRUSH HBrush3 = CreateSolidBrush(RGB(100,200,0));
 HBRUSH HBrush4 = CreateSolidBrush(RGB(200,200,0));
 HBRUSH HBrush5 = CreateSolidBrush(RGB(30,100,0));
 HBRUSH HBrush6 = CreateSolidBrush(RGB(100,100,0));

 hpen = CreatePen(PS_SOLID, 1, RGB(255,0,0));
 hpenOld = (HPEN)SelectObject(h, hpen);

 while (t != NULL)
 { plxy[0].x = (int)t->x[0];  plxy[0].y = (int)t->y[0];
        plxy[1].x = (int)t->x[1];  plxy[1].y = (int)t->y[1];
        plxy[2].x = (int)t->x[2];  plxy[2].y = (int)t->y[2];
        plxy[3].x = (int)t->x[3];  plxy[3].y = (int)t->y[3];
        plxy[4].x = (int)t->x[0];  plxy[4].y = (int)t->y[0];

        switch (t->Ring){
                case NONE: (HBRUSH)SelectObject(h, HBrush1);break;
                case 0: (HBRUSH)SelectObject(h, HBrush2); break;
                case 1: (HBRUSH)SelectObject(h, HBrush3); break;
                case 2: (HBRUSH)SelectObject(h, HBrush4); break;
                case 3:  (HBRUSH)SelectObject(h, HBrush5); break;
                default: (HBRUSH)SelectObject(h, HBrush6);
        }

  Polygon(h, plxy, 4);
```

```
        t = t->Next;
 }
 SelectObject(h, hpenOld);
 DeleteObject(hpen);
}


void Trapez::QSortByY(int DnoIn, int VrhIn)
//Sorts points by the value of y coordinate.
{
 if (DnoIn < VrhIn)
 { int Dno, Vrh;
  double Mid;

  // application specific
  int k;
  double a;
  Dno = DnoIn;
  Vrh = VrhIn;
  int Sredina = (Dno + Vrh) / 2;
  Mid = SortedPointsY[Sredina].y;

  a = SortedPointsY[Dno].y;
  k = SortedPointsY[Dno].PNo;
  SortedPointsY[Dno] = SortedPointsY[Sredina];
       SortedPointsY[Sredina].y = a;
       SortedPointsY[Sredina].PNo = k;

  Dno++;
  do
  { while ((SortedPointsY[Dno].y <= Mid) && (Dno < VrhIn)) Dno++;
   while ((SortedPointsY[Vrh].y >= Mid) && (Vrh > DnoIn)) Vrh--;
   if (Dno < Vrh)
   { a = SortedPointsY[Dno].y;
         k = SortedPointsY[Dno].PNo;
         SortedPointsY[Dno] = SortedPointsY[Vrh];
         SortedPointsY[Vrh].y = a;
```

```
            SortedPointsY[Vrh].PNo = k;
      }
    } while (Dno < Vrh);

    a = SortedPointsY[DnoIn].y;
    k = SortedPointsY[DnoIn].PNo;
    SortedPointsY[DnoIn] = SortedPointsY[Vrh];
          SortedPointsY[Vrh].y = a;
          SortedPointsY[Vrh].PNo = k;
    QSortByY(DnoIn, Vrh-1);
    QSortByY(Vrh+1, VrhIn);
  }
}

void Trapez::SortByY()
// Methods sorts all points regardless if they belong to loop or rings by their
// Y coordinates and stores them into array SortedPoints
{ int i;

  for (i = 0; i < Total; i++)
  { SortedPointsY[i].y = InputPoints[i].y;
          SortedPointsY[i].PNo = i;
          SortedPointsY[i].Used1 = SortedPointsY[i].Used2 = NOT_USED;
  }

  QSortByY(0, Total-1);

  ProcessedPointsIndex = 0;
  TrapezCounter = ErasedTrapezoids = 0;
}

int Trapez::WhichRing(int p)
// Determine in which ring is point p and return its number [0..n]
// If it is in a loop it returns NONE;
{ if (p >= NoL) // ring has been reached
  { int r1 = 0;
          int r2 = NoL + NoR[r1];
          while (r2 <= p)
          { r1++;
```

```
                r2 += NoR[r1];
        }
        return r1;
 }
 return NONE;
}

int Trapez::LocalMaximum(int p)
// Methods return 1 if observed point is a local maximum. It is the local
maximum
// if its y coordinate is bigger than y coordinates of neighbouring points.
{ int i = 0;
 while (SortedPointsY[i].PNo != p) i++;

 if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i1].y, MyEpsilon)
== 1)
   return 0;  // part of a line segment

 if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i2].y, MyEpsilon)
== 1)
   return 0;  // part of a line segment

 if ((SortedPointsY[i].y > InputPoints[SortedPointsY[i].i1].y) &&
                (SortedPointsY[i].y > InputPoints[SortedPointsY[i].i2].y))
        return 1;
 return 0;
}


int Trapez::LocalMinimum(int p)
// Methods return 1 if observed point is a local mininum. It is the local
minimum
// if its y coordinate is smaller than y coordinates of neighbouring points.
{ int i = 0;
 while (SortedPointsY[i].PNo != p) i++;

 if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i1].y, MyEpsilon)
== 1)
   return 0;  // part of a line segment
```

```
 if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i2].y, MyEpsilon)
== 1)
   return 0;  // part of a line segment

 if ((SortedPointsY[i].y < InputPoints[SortedPointsY[i].i1].y) &&
                (SortedPointsY[i].y < InputPoints[SortedPointsY[i].i2].y) )
        return 1;
 return 0;
}

int Trapez::LSMaximum(int p)
// Methods search the neighbourhoud of a horizontal line segment. If the
line
// which touches the horizontal line segment, either form left or from right
side,
// has lower y coordinate, the the observed end point p of the horizontal line
segment
// is a local maximum and method returns 1;
{ int i = 0;
 while (SortedPointsY[i].PNo != p) i++;

 if (!Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i1].y, MyEpsilon))
   if (SortedPointsY[i].y < InputPoints[SortedPointsY[i].i1].y) return 1;

 if (!Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i2].y, MyEpsilon))
   if (SortedPointsY[i].y < InputPoints[SortedPointsY[i].i2].y) return 1;

 return 0;
}

int Trapez::HorizontalSegment(int p)
// point p defines a horizontal segment if at least one neighbouring point has
// the same y coordinate
{ int i = 0;
 while (SortedPointsY[i].PNo != p) i++;

 if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i1].y, MyEpsilon)
== 1)
```

```
   return 1;
  if (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i2].y, MyEpsilon)
== 1)
   return 1;
  return 0;
}


int Trapez::HorizontalMiddle(int p)
// point p is in the middle of two horisontal trapezes. Such point is not
considered
// for trapezoidation. If this case method returns 1
{ int i = 0;
  while (SortedPointsY[i].PNo != p) i++;

  if ((Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i1].y, MyEpsilon)
== 1) &&
     (Equal(SortedPointsY[i].y, InputPoints[SortedPointsY[i].i2].y, MyEpsilon)
== 1))
   return 1;
   return 0;
}


void Trapez::SortIntersectionsX()
// At first, intersection points are ckecked agains their characteristic type, if
// neccessary new intersection points are inserted or existed are eliminated
// and then resulting intersections are sorted by X
{ int i, j;
  IntersectionType a;

  // this for statement acts only in the case when there are rings with common
points.
  // In this case it checks if sequence of ring points in Intersection array is
  // correct. If it is not, wrong positions are swapped.
  for (i=1; i < IntersectionNoX; i++)
  { if ((Equal(Intersections[i].x, Intersections[i-1].x, MyEpsilon) == 1) &&
                          (Intersections[i].Ring != Intersections[i-1].Ring))
        { int k1, k2;
```

```
                k2 = k1 = i+1;
                while ((Intersections[i-1].Ring != Intersections[k1].Ring) && (k1
< IntersectionNoX)) k1++;
                while ((Intersections[i].Ring != Intersections[k2].Ring) && (k2 <
IntersectionNoX)) k2++;

                if (k2 > k1)
                { IntersectionType it = Intersections[i-1];
                  Intersections[i-1] = Intersections[i];
                  Intersections[i] = it;
                }
        }
 }

 j = 0;
 for (i = 0; i < IntersectionNoX; i++)
 { if (Intersections[i].Type != INTERSECTION)
        { if (LocalMaximum(Intersections[i].PNo) == 1)
                { Intersections[i].Type = LOCAL_MINIMUM;
                  a = Intersections[i];
                  Intersections[IntersectionNoX + j++] = a;
                }
                else
                { if (LocalMinimum(Intersections[i].PNo) == 1)
                        Intersections[i].Type = LOCAL_MAXIMUM;
                  else // it is a part of a horisontal segment
                        if (HorizontalSegment(Intersections[i].PNo) == 1)
                                Intersections[i].Type =
HORIZONTAL_SEGMENT;
                        else
                                Intersections[i].Type = INTERSECTION;
                }
        }
 }
 IntersectionNoX += j;

 QSortByX(0, IntersectionNoX-1);
}
```

```
void Trapez::QSortByX(int DnoIn, int VrhIn)
{
 if (DnoIn < VrhIn)
 { int Dno, Vrh;
  double Mid;

  // application specific
  IntersectionType a;


  Dno = DnoIn;
  Vrh = VrhIn;
  int Sredina = (Dno + Vrh) / 2;
  Mid = Intersections[Sredina].x;

  a = Intersections[Dno];
  Intersections[Dno] = Intersections[Sredina];
       Intersections[Sredina] = a;

  Dno++;
  do
  { while ((Intersections[Dno].x <= Mid) && (Dno < VrhIn)) Dno++;
   while ((Intersections[Vrh].x >= Mid) && (Vrh > DnoIn)) Vrh--;
   if (Dno < Vrh)
   { a = Intersections[Dno];
         Intersections[Dno] = Intersections[Vrh];
         Intersections[Vrh] = a;
   }
  } while (Dno < Vrh);

  a = Intersections[DnoIn];
  Intersections[DnoIn] = Intersections[Vrh];
       Intersections[Vrh] = a;

  QSortByX(DnoIn, Vrh-1);
  QSortByX(Vrh+1, VrhIn);
 }
}
```

```
void Trapez::DetermineNeighbours()
{ // Methods determines which two edges share the same point
int i, v, r;

 for (i=0; i < Total; i++)
 { v = SortedPointsY[i].PNo;
        if ((r = WhichRing(v)) == NONE)  // loop point
        { if (v == 0)
                { SortedPointsY[i].i1 = 1;
                  SortedPointsY[i].i2 = NoL - 1;
                }
                else
                { if (v == NoL-1)
                  { SortedPointsY[i].i1 = 0;
                        SortedPointsY[i].i2 = NoL - 2;
                  }
                  else
                  { SortedPointsY[i].i1 = v - 1;
                        SortedPointsY[i].i2 = v + 1;
                  }
                }
        }
        else
        { int LastInRing, FirstInRing = NoL;

                for (int j = 0; j < r; j++) FirstInRing += NoR[j];

                LastInRing = FirstInRing + NoR[r] - 1;

                if (v == FirstInRing)
                { SortedPointsY[i].i1 = FirstInRing+1;
                  SortedPointsY[i].i2 = LastInRing;
                }
                else
                {
                  if (v == LastInRing)
                  { SortedPointsY[i].i1 = FirstInRing;
```

```
                    SortedPointsY[i].i2 = LastInRing - 1;
                }
                else  // somewhere in the middle of the ring
                { SortedPointsY[i].i1 = v - 1;
                    SortedPointsY[i].i2 = v + 1;
                 }
               }
            }
        }
    }
```

```
int Trapez::IntersectionWithHLine(double y1, double x3, double y3, double a,
double b,
        double &outx, double &outy)
// Method calculates the intersection with horizontal line which y coordinate
is y1.
// Line two passes through point (x3, y3) and has slope k = a/b
// a should be passed as y4-y3 and b as x4-x3.
// Function returns 1 if lines intersects and then the intersection point is
stored
// in (xout, yout)
// if Function returns 0, line l2 is also horisontal and an trivial intersection
// does not exists.
{ if (Equal(a, 0,MyEpsilon) == 1) return 0;   // line l2 is also horisontal
 outy = y1;
 if (Equal(b, 0,MyEpsilon) == 1)
        outx = x3;
 else
        outx = x3 + (y1 - y3) * b / a;
 return 1;
}
```

```
void Trapez::PointsWithTheSameY(int &i1, int &i2)
// Methods finds all points with the same y coordinate in sorted array.
// i1 is the index of the first and i2 of the last such points
{ i2 = i1 = ProcessedPointsIndex;
 while (Equal(SortedPointsY[i1].y, SortedPointsY[i2].y,MyEpsilon) == 1)
```

```
 { i2++;
        if (i2 >= Total)
                break;
 }
 ProcessedPointsIndex = i2;
 i2--;
}

void Trapez::InitOldBuffer()
// initialization of old buffer. There are two possibilities:
// the point with the minimum y is local minimum; in such case two points
are
// inserted
// the point can be also part of a horiyontal line segment and in this case only
// one point is inserted.
{ int yi1, yi2, i, j1;
 int j = 0;

 PointsWithTheSameY(yi1, yi2);

 YCoord a;

 for (i = yi1; i <= yi2-1; i++)
        for (j = i+1; j <= yi2; j++)
        { if (InputPoints[SortedPointsY[j].PNo].x <
InputPoints[SortedPointsY[i].PNo].x)
                { a = SortedPointsY[j];
      SortedPointsY[j] = SortedPointsY[i];
      SortedPointsY[i] = a;
                }
        }
 j = 0;
 for (i = yi1; i <= yi2; i++)
 { if (LocalMinimum(SortedPointsY[i].PNo) == 1)
        { // add two points into OldIntersections
                j1 = j + 1;
                OldIntersections[j].x = OldIntersections[j1].x =
InputPoints[SortedPointsY[i].PNo].x;
```

```
                OldIntersections[j].y = OldIntersections[j1].y =
SortedPointsY[i].y;
                OldIntersections[j].Type = OldIntersections[j1].Type =
LOOP_POINT;
                OldIntersections[j].PNo = OldIntersections[j1].PNo =
SortedPointsY[i].PNo;
                OldIntersections[j].Ring = OldIntersections[j1].Ring = NONE;
                j+=2;
        }
        else // the point is horizontal minimum and therefore, add just one
point
        {
     if (HorizontalMiddle(SortedPointsY[i].PNo) == 0)
     { OldIntersections[j].x = InputPoints[SortedPointsY[i].PNo].x;
                OldIntersections[j].y = SortedPointsY[i].y;
                OldIntersections[j].Type = LOOP_POINT;
                OldIntersections[j].PNo = SortedPointsY[i].PNo;
                OldIntersections[j].Ring = NONE;
                MarkUsedEdges(i, i+1);
      j++;
     }
   }
 }
}


void Trapez::MarkUsedEdges(int p, int q)
// mark edges which touch vertices on skaning line as already use
{ int flag, b, k1, k2;
 if (p > q) b = p;
 else b = q;
 for (k1 = p; k1 <= q; k1++)
        for (k2 = 0; k2 < b; k2++)
        { flag = 0;
                if (SortedPointsY[k1].PNo == SortedPointsY[k2].i1)
                { flag = 1;
                 SortedPointsY[k2].Used1 = USED;
                }
                else
```

```
                if (SortedPointsY[k1].PNo == SortedPointsY[k2].i2)
                { flag = 1;
                        SortedPointsY[k2].Used2 = USED;
                }
               if (flag == 1)
               {
                if (SortedPointsY[k1].i1 == SortedPointsY[k2].PNo)
                                SortedPointsY[k1].Used1 = USED;
                else
                        if (SortedPointsY[k1].i2 == SortedPointsY[k2].PNo)
                                SortedPointsY[k1].Used2 = USED;
               }
        }
}


void Trapez::CalculateIntersections(int p, int q)
// Method finds all intersection points between a horizontal scaning line and
// polygon edges.
// p and q are indexes into array of sorted points of the polygon which has
the
// same y coordinates
// method at first calculate the real intersections and at the end adds q-p+1
// polygon nodes
{ int i, j = 0;

 MarkUsedEdges(p, q);
 for (i = 0; i < p; i++)
 {
   if (HorizontalMiddle(SortedPointsY[i].PNo) == 0)
   {
   if ((SortedPointsY[i].Used1 == NOT_USED) && (SortedPointsY[i].i1 !=
SortedPointsY[i].PNo))
        { IntersectionWithHLine(SortedPointsY[p].y,
                        InputPoints[SortedPointsY[i].i1].x,
                        InputPoints[SortedPointsY[i].i1].y,
                        InputPoints[SortedPointsY[i].i1].y -
InputPoints[SortedPointsY[i].PNo].y,
```

```
                    InputPoints[SortedPointsY[i].i1].x -
InputPoints[SortedPointsY[i].PNo].x,
                    Intersections[j].x, Intersections[j].y);
            Intersections[j].Type = INTERSECTION;
            Intersections[j].PNo = NONE;
            Intersections[j].Ring = WhichRing(SortedPointsY[i].PNo);
            j++;
        }

        if ((SortedPointsY[i].Used2 == NOT_USED) && (SortedPointsY[i].i2 !=
SortedPointsY[i].PNo))
        { IntersectionWithHLine(SortedPointsY[p].y,
                    InputPoints[SortedPointsY[i].i2].x,
                    InputPoints[SortedPointsY[i].i2].y,
                    InputPoints[SortedPointsY[i].i2].y -
InputPoints[SortedPointsY[i].PNo].y,
                    InputPoints[SortedPointsY[i].i2].x -
InputPoints[SortedPointsY[i].PNo].x,
                    Intersections[j].x, Intersections[j].y);
            Intersections[j].Type = INTERSECTION;
            Intersections[j].PNo = NONE;
            Intersections[j].Ring = WhichRing(SortedPointsY[i].PNo);
            j++;
        }
    }
 }
 int r;
 for (i = p; i <= q; i++)
 { if (HorizontalMiddle(SortedPointsY[i].PNo) == 0)
  {
    Intersections[j].x = InputPoints[SortedPointsY[i].PNo].x;
        Intersections[j].y = SortedPointsY[i].y;
        Intersections[j].PNo = SortedPointsY[i].PNo;

        if ( (r = WhichRing(SortedPointsY[i].PNo)) == NONE)
            Intersections[j].Type = LOOP_POINT;
        else
            Intersections[j].Type = RING_POINT;
        Intersections[j].Ring = r;
```

```
        j++;
        }
        IntersectionNoX = j;
    }
}


void Trapez::MakeTrapezs()
// At first obtained intersection points are considered
// and arranged regarding their characteristic type (local minimum,
horizontal segment
// memeber of a ring etc.)
// Then trapez are easily constructed by merging ald and new intersection
points.
{ int i, Where;
 int j=0;
 int ri = 0;

      for (i = 0; i < IntersectionNoX; i++)
      { if ((Where = WhereToPlace(i)) == NONE)
       {
            switch (Intersections[i].Type)
            { case LOCAL_MAXIMUM: break;
                  case HORIZONTAL_SEGMENT:
                  { if (LSMaximum(Intersections[i].PNo) == 0)
                   { if (HorizontalMiddle(Intersections[i].PNo) == 0)
        { UsefulIntersections[j] = Intersections[i];
                        j++;
        }
                   }
                   break;
                  }
                  default:
                  { UsefulIntersections[j] = Intersections[i];
                        j++;
                  }
            }
      }
      else
```

```
{

        switch (Intersections[i].Type)
        { case LOCAL_MAXIMUM: break;
                case HORIZONTAL_SEGMENT:
                { if (LSMaximum(Intersections[i].PNo) == 0)
                  { if (HorizontalMiddle(Intersections[i].PNo) == 0)
 { UsefulRingIntersections[ri] = Intersections[i];
                        UsefulRingIntersections[ri].Ring = Where;
                        ri++;
 }
                  }
                  break;
                }
                default:
                { UsefulRingIntersections[ri] = Intersections[i];
                  UsefulRingIntersections[ri].Ring = Where;
                  ri++;
                }
        }
}

if (Intersections[i].Ring != NONE)
{ switch (Intersections[i].Type)
        { case LOCAL_MAXIMUM: break;
                case HORIZONTAL_SEGMENT:
                { if (LSMaximum(Intersections[i].PNo) == 0)
                  { if (HorizontalMiddle(Intersections[i].PNo) == 0)
 {
   UsefulRingIntersections[ri] = Intersections[i];
                        ri++;
 }
                  }
                  break;
                }
                default:
                { UsefulRingIntersections[ri] = Intersections[i];
                  ri++;
                }
```

```
                }
            }
        }

        CorrectURI(ri);
        for (i = 0; i < j; i+=2)
        { TrapezType* t = new TrapezType;
                t->x[0] = OldIntersections[i].x;     t->y[0] = OldIntersections[i].y;
                t->x[1] = UsefulIntersections[i].x;  t->y[1] =
UsefulIntersections[i].y;
                t->x[2] = UsefulIntersections[i+1].x; t->y[2] =
UsefulIntersections[i+1].y;
                t->x[3] = OldIntersections[i+1].x;   t->y[3] =
OldIntersections[i+1].y;
                t->Ring = NONE;
                t->Next = NULL;
                Add(t);
        }
        int r;
        int *RIndex = new int[NoRings+10];
        int Index;
        for (i = 0; i < NoRings; i++) RIndex[i] = 0;

        for (i = 0; i < ri; i+=2)
        { TrapezType* t = new TrapezType;
          r =  UsefulRingIntersections[i].Ring;
          Index = RIndex[r];
          t->x[0] = OldRingIntersections[r][Index].x;     t->y[0] =
OldRingIntersections[r][Index].y;
          t->x[1] = UsefulRingIntersections[i].x;  t->y[1] =
UsefulRingIntersections[i].y;
          t->x[2] = UsefulRingIntersections[i+1].x; t->y[2] =
UsefulRingIntersections[i+1].y;
          t->x[3] = OldRingIntersections[r][Index+1].x;   t->y[3] =
OldRingIntersections[r][Index+1].y;
          RIndex[r] += 2;
          t->Ring = r;
          t->Next = NULL;
          Add(t);
```

```
 }
 delete[] RIndex;
}


void Trapez::Add(TrapezType* t)
//Add trapesoid to the list.
{ TrapezCounter++;
 if (TrapezPointer == NULL)
 { TrapezPointer = t;
  TrapezPointerTail = TrapezPointer;
  t->Next = NULL;
 }
 else
 { TrapezPointerTail->Next = t;
   TrapezPointerTail = t;
   t->Next = NULL;
 }
}


void Trapez::CorrectURI(int n)
// Method checks array UsefulRingIntersections because fome
inconsistencies can
// appear if rings thouch each others. At first URI is sorted regarding ring
number.
// In this way, the intersections which belongs to the same ring are gruped.
Then
// each grupe is sorted again X
{ int i,j;
 IntersectionType p;

 for (i = 0; i < n-1; i++)
        for (j = i+1; j < n; j++)
        { if (UsefulRingIntersections[i].Ring > UsefulRingIntersections[j].Ring)
                { p = UsefulRingIntersections[i];
                  UsefulRingIntersections[i] = UsefulRingIntersections[j];
                  UsefulRingIntersections[j] = p;
                }
```

```
        }

 int r;
 int i1;
 i = i1 = 0;
 while (i < n)
 { r = UsefulRingIntersections[i].Ring;
        i1 = i;
        i++;
        while ((UsefulRingIntersections[i].Ring == r) && (i < n))
        {       i++; }

        SortURIByX(i1, i);
 }
}


void Trapez::SortURIByX(int i1, int i2)
// sorts array UsefulRingIntersections regarding x between index i1 and i2
{ int i, j;
 IntersectionType p;

 for (i = i1; i < i2-1; i++)
        for (j = i+1; j < i2; j++)
        { if (UsefulRingIntersections[i].x > UsefulRingIntersections[j].x)
                { p = UsefulRingIntersections[i];
                  UsefulRingIntersections[i] = UsefulRingIntersections[j];
                  UsefulRingIntersections[j] = p;
                }
        }
}


int Trapez::WhereToPlace(int i)
// Methods determines if  intersection i has a contact with inside area of a
parcel.
// If it has, it returns NONE, if not, then this points is a common point of two
rings.
// This can happen in two cases:
```

```
// 1. one ring contains another ring
// 2. one ring thouches another ring (exception is if it thouches it in a
horizontal
// direction but this is already handeled.
// Functions examines the sorted sequence of intersections.
{ if (
        (i == 0) | |            // the first and
        (i == IntersectionNoX) | | // the last intersection point has to belongs
to the loop.
        (Intersections[i].Ring == NONE) // Intersection belongs to the main
loop
        )
       return NONE;

 if ( (Intersections[i-1].Ring == NONE) | | (Intersections[i+1].Ring == NONE)
)
        return NONE; // Firts neighbouring (left or right) point belongs to
the loop

 /* Points of the rings can be placed on a common horizontal line segment.
Then
         we find the first intersection which does not belongs to the line
segment
 */

 if (Intersections[i].Type == HORIZONTAL_SEGMENT)
 { int j = i-1;
        while ((Intersections[j].Type == HORIZONTAL_SEGMENT) && (j >
0))
                j--;
        if (j == 0) return NONE;
        else
    if (Equal(Intersections[i].x, Intersections[j].x,MyEpsilon) == 1)
     return Intersections[j].Ring;
 }

 /* Now we are testing, if node belongs to the ring inside ring. We search to
the
```

```
        left and to the ring from the node and cound the number of
        intersections      with  the outside ring (CounterLeft CounterRingt). If
        both counters are odd then  the node is a member of a ring which is
        inside another ring.
*/
// first jump over the intersections with the same ring
int b1 = i-1;
while (Intersections[b1].Ring == Intersections[i].Ring) b1--;

int b2 = i+1;
while (Intersections[b2].Ring == Intersections[i].Ring) b2++;

// then determine if both left and right ring number are the same. If not
// search further
int EnclosingRingNo;

if ((EnclosingRingNo = FindEncloseingRing(b1, b2)) == NONE)
        return NONE;

int j;
int LeftCounter = 0;
int RightCounter = 0;

for (j = 1;  j<= b1; j++)
        if (Intersections[j].Ring == EnclosingRingNo)
        { LeftCounter++;
                if (Intersections[j].PNo != NONE)
                  LeftCounter++;
        }
for (j = IntersectionNoX - 2; j >= b2; j--)
        if (Intersections[j].Ring == EnclosingRingNo)
        { RightCounter++;
                if (Intersections[j].PNo != NONE)
                  RightCounter++;
        }

if ((Even(LeftCounter) == 1) | | (Even(RightCounter) == 1) )
        return NONE;
```

```
 return Intersections[b2].Ring;
}


int Trapez::FindEncloseingRing(int b1, int b2)
// Method finds the enclosing ring of a ring if it exists.
// b1 is the first intersection to the left of treated intersection which does not
// belongs to the same ring as treated intersection
// and b2 is the first such intersection to the right
{ if (Intersections[b1].Ring == Intersections[b2].Ring) return
Intersections[b1].Ring;
 int i;
 while (b2 < IntersectionNoX)
 { i = b1;
        while (i > 0)
        { if (Intersections[i].Ring == Intersections[b2].Ring)
                return Intersections[i].Ring;
              i--;
        }
        b2++;
 }
 return NONE;
}


void Trapez::UpdateOldIntersections()
// Methods stores intersections reagrding their characteristic type.
{ int i, r1, j = 0;
 int *jr = new int[NoRings+10];
 for (i = 0; i < NoRings; i++) jr[i] = 0;
 for (i = 0; i < IntersectionNoX; i++)
 { int Where = WhereToPlace(i);

        if (Where == NONE)  // intersection is part of a main loop
        { switch (Intersections[i].Type)
                { case LOCAL_MINIMUM:
                        break;
                  case HORIZONTAL_SEGMENT:
                  { if (LSMaximum(Intersections[i].PNo) == 1)
```

```
                    { OldIntersections[j] = Intersections[i];
                        j++;
                    }
                    break;
                }
                case LOCAL_MAXIMUM:
                { OldIntersections[j] = OldIntersections[j+1] = Intersections[i];
                    j += 2;
                     break;
                }
                default:
                { OldIntersections[j] = Intersections[i];
                    j++;
                }
               }
        }
        else // intersection is part of a ring (it must thouch the ring or
                    // be placed inside it
        {
                FillOldRingIntersections(OldRingIntersections, i, Where, jr);
                FillOldRingIntersections(OldRingIntersections, i,
Intersections[i].Ring, jr);
                goto ne;
        }

        if ( (r1 = Intersections[i].Ring) != NONE) // ring has been reached
       FillOldRingIntersections(OldRingIntersections, i, r1, jr);
        ne: ;
 }
 delete jr;
}


void Trapez::FillOldRingIntersections(IntersectionType**
OldRingIntersections, int i, int r1, int* jr)
// Methods stores intersections which has been inside rings for next iteration
{ switch (Intersections[i].Type)
 { case LOCAL_MINIMUM: break;
        case HORIZONTAL_SEGMENT:
```

```
        { if (LSMaximum(Intersections[i].PNo) == 1)
                { OldRingIntersections[r1][jr[r1]] = Intersections[i];
                  jr[r1]++;
                }
                break;
        }
        case LOCAL_MAXIMUM:
        { if (LSMaximum(Intersections[i].PNo) == 1)
                { OldRingIntersections[r1][jr[r1]] =
OldRingIntersections[r1][jr[r1]+1] = Intersections[i];
                  jr[r1] += 2;
                }
                break;
        }
        default:
        { OldRingIntersections[r1][jr[r1]] = Intersections[i];
                jr[r1]++;
        }
    }
}


void Trapez::MakeTrapezodiation()
//Create trapezoids.
{ int i1, i2;
        int i = 0;

        while (ProcessedPointsIndex < Total)
        { PointsWithTheSameY(i1, i2);
          CalculateIntersections(i1, i2);
          SortIntersectionsX();
          MakeTrapezs();
          UpdateOldIntersections();

          i++;
        }
}
```

```
void Trapez:: OptimizeTrapezs()
// Developed by Olga Zaporojets
// Method finds and joins those trapezs, which can be joined to reduce the
number
// of trapezs. There is, of course no need to compare each trapezs which
each.
// We have to check only those trapezs which share a common scanning line
{ TrapezType *t1, *t2, *t3;
 int flag, JoinCode, JoinFlag, ExitF;
 double ERROR_ACCUM;


#ifdef shareware
 // SharewareMessage();
#endif

        char* out= "CPUreduction";
        MyCPUTime output;
        ERROR_ACCUM =0;
 output.Start();
 t1 = TrapezPointer;
 while (t1 != NULL)
 { t2 = t1->Next;
        t3 = t1;

   int WorkFlag = 1;
        while ((t2 != NULL) && (WorkFlag == 1))  // jump over the
horisontal trapezs
        { if (t2->y[0] == t1->y[0])
    { t3 = t2;
          t2 = t2->Next;
    }
    else
     WorkFlag = 0;
        }
   // we are in the next row
   flag = FALSE;
   JoinFlag = 0;
        while ((t2 != NULL) && (flag == FALSE))
```

```
    { if ((JoinCode = TrapezsCanBeJoined(t1, t2, ERROR_ACCUM)) != 0)
      {
            ErasedTrapezoids++;
   JoinFlag = 1;
   if (JoinCode == 1)
   { t1->x[1] = t2->x[1];
              t1->y[1] = t2->y[1];
              t1->x[2] = t2->x[2];
              t1->y[2] = t2->y[2];
   }
            t3->Next = t2->Next;
            delete t2;
   t2 = t3->Next;
   /// move in the beginning of the next row
   ExitF = 0;
   while ((t2 != NULL) && (ExitF == 0))
   { if (Equal(t2->y[1], t1->y[1],MyEpsilon) == 1)
    { t3 = t2;
     t2 = t2->Next;
    }
    else ExitF = 1;
   }
      }
      else
      { ERROR_ACCUM = 0;
      //there is no reduction at this point, reset ERROR_ACCUM
            t3 = t2;
            t2 = t2->Next;
   if (t2 != NULL)
    if (Equal(t2->y[1], t3->y[1],MyEpsilon) == 0)
    { if (JoinFlag == 1)
      JoinFlag = 0;
     else
      flag = TRUE;
         }
  }
      }
 t1 = t1->Next;  // else we try to find next touching trapez
}
```

```
        output.Stop();
        output.WriteEllapsedCPUToFile(out);
}


void Trapez::PrintListOfTrapezoids()
//Print trapezoids.
{ TrapezType *t = TrapezPointer;
 int i, k;
 k=0;
 FILE* f = fopen("Atrapezoid.txt", "wt");
 while (t != NULL)
 {
  for (i = 0; i < 4; i++)
   fprintf(f,  "%#20.15lf, %#20.15lf \n", t->x[i], t->y[i] );
  fprintf(f, "\n");
  fprintf(f, "ring %d \n", t->Ring);
  fprintf(f, "---------------------- \n");
                k++;
  t = t->Next;
 }
 k=k/2;
 fprintf(f, "Number of polygons is:  %i ", k);
 fclose(f);
}


int Trapez::TrapezsCanBeJoined(TrapezType* t1, TrapezType* t2, double
&ERROR_ACCUM)
//  Developed by Olga Zaporojets
// Method determines if two trapezs which has a common y coordinate, can
be joined
// It returns 0 if they cannot be joined and a code which determines the way,
how
// trapezs can be joined.
{
// indicate that merge may occure, extra area is added to the
ERROR_ACCUM global variable
// to keep track of combined changes.
```

```
Point p1, p2;
double S1, S2, d_left, d_right;
double error_delta =0.0001;
double error_threshold = 0.001;
// Case 1, no data topology modification. All vertices match and edges lay on
the same lines
if (t1->Ring != t2->Ring) return 0;
//if this is different rings, don't reduce
if (((Equal(t1->x[1], t2->x[0], MyEpsilon)==1) && (Equal(t1->x[2],t2->x[3],
MyEpsilon)==1)))
//displacement d =0; case 1 or 2
        {
        if (((Collinear2D(t1->x[0], t1->y[0], t1->x[1], t1->y[1], t2->x[1],t2->y[1],
MyEpsilon)) &&
        (Collinear2D(t1->x[2], t1->y[2], t1->x[3], t1->y[3], t2->x[2], t2->y[2],
MyEpsilon))))
                {//edges are colinear, so this is case 1
                        return 1;
                }
        //else  edges are not colinear, this is case 2
        if ((t1->Ring != -1) || (t2->Ring != -1)) return 0; //not a loop
        //FIND FUNCTION POINT ON LINE,
        //create reduction lines and check if no area is cut off
        Line2D* l1 = new Line2D (t2->x[1], t2->y[1], t1->x[0], t1->y[0]);
        Line2D* l2 = new Line2D (t2->x[2], t2->y[2], t1->x[3], t1->y[3]);
        l1->PointOnLine(t1->y[1], p1);
        l2->PointOnLine(t1->y[1], p2);
        if ((p1.x < t1->x[1]) || (p2.x < t1->x[2])) return 0; //area would be cut
        // else calculate area error Sadd = 1/2*(x2y1 + y2x1):
        //left side:
        S1 = ((t1->x[1] - t1->x[0])*(t2->y[1] - t2->y[0])+(t2->x[1] - t2->x[0])*(t1-
>y[1] - t1- >y[0]))/2;
        //right side:
        S2 = ((t1->x[3] - t1->x[2])*(t2->y[1] - t2->y[0])+(t1->x[3] - t1->x[2])*(t1-
>y[1] - t1->y[0]))/2;
        // check if added area is violated:
        ERROR_ACCUM = ERROR_ACCUM + S1 + S2;
        if (ERROR_ACCUM > error_threshold) return 0;
        return 1;
```

```
}//end of case 2 and 1
else
        {
        if ((t1->Ring != -1) || (t2->Ring != -1)) return 0; //not a loop
        // there is displacement d, it is case 3
        //check if displacement is small enouth
        if (((t1->x[1]-t2->x[0]) > error_delta) ||((t1->x[2]-t2->x[3])>
error_delta)) return 0;
        //create reduction lines and check if no area is cut off
        Line2D* l1 = new Line2D (t2->x[1], t2->y[1], t1->x[0], t1->y[0]);
        Line2D* l2 = new Line2D (t2->x[2], t2->y[2], t1->x[3], t1->y[3]);
        l1->PointOnLine(t1->y[1], p1);
        l2->PointOnLine(t1->y[1], p2);
        if ((p1.x < t1->x[1]) || (p2.x < t1->x[2])) return 0; // there is a cut off
from t1
        if ((p1.x < t2->x[0]) || (p2.x < t2->x[3])) return 0; // there is a cut off
from t2
        //calculate d on the left and on the right
        d_left = t1->x[1] - t2->x[0];
        d_right = t1->x[2] - t2->x[3];
        // calculate area error Sadd = 1/2*(x2y1 + x1y2 + dy1 + dy2)
        //left side:
        S1 = ((t1->x[1] - t1->x[0])*(t2->y[1] - t2->y[0])+(t2->x[1] - t2->x[0])*(t1-
>y[1] - t1->y[0]) + d_left*(t2->y[1] - t2->y[0]) + d_left*(t1->y[1] - t1->y[0]))/2;
        //right side:
        S2 = ((t1->x[3] - t1->x[2])*(t2->y[1] - t2->y[0])+(t1->x[3] - t1->x[2])*(t1-
>y[1] - t1->y[0]) + d_right*(t2->y[1] - t2->y[0]) + d_right*(t1->y[1] - t1-
>y[0]))/2;
        // check if added area is violated:
        ERROR_ACCUM = ERROR_ACCUM + S1 + S2;
        if (ERROR_ACCUM > error_threshold) return 0;
        return 1;
}//end of case 3
}


void Trapez::Trapezoidation()
{ if (Total > 3)
  {
```

```
        MyCPUTime* BigDataTime = new MyCPUTime();
    BigDataTime->Start();

    DetermineEpsilion();
    SortByY();
        DetermineNeighbours();
        InitOldBuffer();
    MakeTrapezodiation();

        BigDataTime->Stop();
    BigDataTime->WriteEllapsedCPUToFile("BigData.CPU");
 }
        else {}; // nothing to do
}

void Trapez::EraseWindow(HDC hdc, HWND hWnd)
// Used for GUI output
{ RECT r;
 HBRUSH hbrOld, hbr;

 DWORD dwBackColor = GetBkColor(hdc);
 hbr = CreateSolidBrush(dwBackColor);
 hbrOld = (HBRUSH)SelectObject(hdc, hbr);
 GetClientRect( hWnd, (LPRECT) &r );
 FillRect(hdc, &r, hbr);
 SelectObject(hdc, hbrOld);
 DeleteObject(hbrOld);
}
```

Bibliography

[1] A. Ciampalini, P. Cigoni, C. Montani, and R. Scopigno. Multiresolution decimation based on global error. *The Visual Computer*, Springer International, 13(5), pages 228-246, 1997.

[2] P. Cignoni, C. Roccini, and R. Scopigno. A comparison of mesh simplification algorithms. *Computer Graphics*, 22(1), pages 37-54, 1998.

[3] Greg Turk. Re-Tiling Polygonal Surfaces. Proceedings of SIGGRAPH 92. In *Computer Graphics*, 26(2), pages 55-64, July 1992.

[4] T. He, L. Hong, A. Kaufman, A. Varshney, and S. Wang. Voxel based object simplification. In *IEEE Visualization 95 Conference Proceedings*, pages 296-303, October 1995.

[5] W. Schroeder, J. Zarge, and W. Lorensen. Decimation of triangle meshes. Proceedings of SIGGRAPH 92. In *Computer Graphics*, 26(2), pages 65-70, July 1992.

[6] K. Renze and J. Oliver. Generalized surface and volume decimation for unstructured tessellated domains. *In VRAIS Proceedings*, pages 111-121, 1996

[7] B. Hamann. A data reduction scheme for triangulated surfaces. *Computer Aided Geometric Design*, 11(2), pages 197-214, April 1994.

[8] W. Schroeder. A topology modifying progressive decimation algorithm. In *IEEE Visualization 97 Conference Proceedings*, pages 205-212, October 1997.

[9] C. Bajaj and D. Schikore. Error bounded reduction of triangle meshes with multivariable data. In *Visual Data Exploration and Analysis III Proceedings*, SPIE 2656, pages 34-45, 1996

[10] H. Hoppe. Progressive meshes. Proceedings of SIGGRAPH 96. In *Computer Graphics*, pages 99-108, August 1996. http://research.microsoft.com/~hoppe/.

[11] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *IEEE Visualization 98 Conference Proceedings*, pages 279-286, 544, October 1998.

[12] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. Proceedings of SIGGRAPH 93. In *Computer Graphics*, pages 19-26, August 1993. http://research.microsoft.com/~hoppe/.

[13] R. Ronfard and J. Rossignac. Full-range approximation of triangulated polyhedra. Technical Report IBM Resesarch Report RC 20423, IBM T. J. Watson Research, Yorktown Heights, NY, 1996.

[14] O. Faugeras, M. Hebert, P. Mussi, J. Boissonnat. Polyhedral approximation of 3-D objects without holes. *Computer Vision, Graphics, and Image Processing*, 25:169-183, 1984.

[15] M. DeHaemer, Jr. and M. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2):175-184, 1991.

[16] H. Imai and M Iri. Polygonal approximations of a curve – formulations and algorithms. In G. T. Toussaint, editor, *Computational Morphology*, pages 71-86. Elsevier Science, 1988.

[17] P. Prenter. Splines and Variational Methods. John Wiley & Sons, New York, 1975

[18] M. Garland. Quadric-Based Polygonal Surface Simplification. CMU technical report. CMU-CS-99-105, Dept. of Computer science, Carnegie Mellon University, Pittsburgh, PA, May 1999.

[19] R. Holloway. Viper: A Quasi-Real-Time Virtual-Worlds Application. UNC technical report. TR-92-004, Dept. of Computer Science, University of North Carolina, Chapel Hill, NC. December 1991.

[20] J. Sewell. Automatic Generation of simple Replacements for Groups of Objects. The University of Cambrige Computer Laboratory, UK. 1995.

[21] S. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. on Pattern analysis and Machine Intelligence*, 11(7):674-693, July 1989

[22] M. Lounsberry, T. DeRose, J. Warren. Multiresoluton analysis for surfaces of arbitrary topologycal type. *ACM Trans. on Graphics*, 16(1):34-73, 1997. http://www.cs.washington.edu/research/grephics/pub/.

[23] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganised points. *Computer Graphics*, Proceedings of SIGGRAPH 92, 26(2), pages 71-78, 1992. http://research.microsoft.com/~hoppe/.

[24] H. Hoppe. New quadratic metric for simplifying meshes with appearance attributes. *IEEE Visualization 1999*, pages 59-66, October 1999. http://research.microsoft.com/~hoppe/.

[25] B. Zalik and G. Clapwrthy. A universal trapezoidation algorithm for planar polygons. *Computer and Graphics*, 23: 353-363, 1999.

VITA

Olga Vladimirovna Zaporojets was born in Kiev, Ukraine on September 20th, 1972, the daughter of Galina V Zaporojets and Vladimir V. Zaporojets. She completed her work at Krasnodar Mathematics College, Krasnodar, Russia, in 1989, with the profession of a computer operator. She entered Kuban State University in Krasnodar, Russia in August 1989. During the study at the Kuban State University she worked at the State Research Laboratory as a research assistant and at the Krasnodar Medical College as a physics lecturer. She received the degree of Master of Science from Kuban State University in June 1994. She was accepted to the Postgraduate School at the Physics Department of Kuban State University in June 1994. She entered the Graduate School of Southwest Texas State University in San Marcos Texas in 2000. She was employed at Avant! Corporation as an engineer from May 2000 until September 2001. Since December 2001 she works as an engineer at Intel Corporation in Design Methodology group.

Permanent Address: 106 Hill Drive
San Marcos, Texas 78666

This thesis was typed by Olga Vladimirovna Zaporojets.