UNIVERSITY OF MIAMI

ROBUST TOPOLOGICALLY INVARIANT SET OPERATIONS ON 2D SEMI-ALGEBRAIC SETS

By

Steven Cy Trac

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

December 2005

UNIVERSITY OF MIAMI

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

ROBUST TOPOLOGICALLY INVARIANT SET OPERATIONS ON
2D SEMI-ALGEBRAIC SETS

Steven Cy Trac

Approved:

Dr. Victor Milenkovic Committee Chairman Associate Professor of Computer Science	Dr. Steven G. Ullmann Dean of the Graduate School
Dr. Christian Duncan Committee Member Assistant Professor of Computer Science	Dr. Elisha Sacks Committee Member Professor of Computer Science Purdue University

Vita

Steven Cy Trac was born in Baltimore, Maryland, on July 12, 1981. His parents are Tan Ba Trac and Judy Trac. He has one younger brother, Ivan Cy Trac. He received his elementary education at Sabal Palm Elementary School and his secondary education at Charles W. Flanagan High School. In September 1999 he entered the University of Miami from which he was graduated with a BS degree *cum laude*, double majoring in Computer Science and Mathematics, in May 2003.

In August 2003 he was admitted to the Graduate School of the University of Miami where he was granted a master's degree in Computer Science in December 2006.

Permanent Address: 201 SW 98 Terrace, Pembroke Pines, Florida 33025

TRAC, STEVEN (M.S., Computer Science)

Robust Topologically Invariant Set Operations on 2D Semi-Algebraic Sets (December 2005)

Abstract of a thesis at the University of Miami.

Thesis supervised by Professor Victor Milenkovic.
No. of pages in text. (71)

We present robust floating point algorithms for performing topologically invariant set operations on $2D$ semi-algebraic sets. A $2D$ semi-algebraic set is a set of cells from an arrangement of segments of algebraic curves. These algorithms rotate/translate the arrangement, while maintaining its topology. In addition, when multiple arrangements are overlayed for set operations, the topology is maintained in each of the individual arrangements.

Because the algorithms are implemented in floating point, certain topological changes are unavoidable. For instance, after a floating point Euclidean transformation, two very close but distinct points may transform into the same point. The topological changes that can result from floating point rounding are analyzed. The algorithms are invariant except for these types of changes. We discovered that the unavoidable topological changes in our robust topologically invariant algorithms occur rarely in testing.

We ran tests on our topologically invariant algorithms against the arrangement algorithms by Milenkovic and Sacks, comparing computing time and accuracy. Testing shows that our topologically invariant algorithms were able to compute arrangements faster than the algorithms by Milenkovic and Sacks. Testing also shows our algorithms do not introduce a significant amount of new error, with respect to their algorithms.

Dedication

This thesis is dedicated to my parents, Judy Trac and Tan Trac, and my brother, Ivan Trac. This is also dedicated to my late grandparents who passed away recently in my life, as well as my late Uncle Sing.

Acknowledgements

I would like to thank Dr. Victor Milenkovic, my advisor, for giving me the chance and opportunity to work under him. It has been a privilege to learn so much about computational geometry, about his research, and just general knowledge of everything with science. I want to thank Dr. Milenkovic also for funding me through out my years as a graduate student through his own grants from the National Science Foundation. I want to thank the generous donors to the Department of Computer Science for funding me as well.

I would like to thank Dr. Christian Duncan and Dr. Elisha Sacks for serving in my M.S. Thesis Committee and for their constructive and effective comments about my thesis.

I would also like to thank Dr. Geoff Sutcliffe for allowing me to join his research group here at the University of Miami. He was able to spend a lot of time to proofread my thesis and give constructive comments. I want to also thank my friends here at the University of Miami and my friends abroad who have kept me sane through out these past years.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Background 1

1.2 Prior Work 5

1.2.1 Exact Methods 5

1.2.2 Perturbation Methods 6

1.2.3 Numerical Methods 7

1.3 Motivation and Goals 8

1.4 Robust Topological Invariance 9

1.4.1 Acceptable Euler Operation: Add/Remove a Vertex 9

1.4.2 Floating Point Rounding 11

1.4.3 Unavoidable Euler Operations 14

2 Arrangement Construction 16

2.1 Line Sweep Algorithm 17

2.1.1 Insert Event 19

2.1.2 Remove Event 19

2.1.3 Swap Event 21

2.2 Inconsistencies 22

2.3 Loops 24

2.4 Cells 27

2.5 Running Times 29

3 Regularized Semi-Algebraic Set **30**

3.1 Cell Selection by Depths 30

3.2 Creating an Oriented Boundary Arrangement 31

3.3 Assigning Winding Numbers 33

3.4 Transforming Sets 34

3.4.1 Algorithm 34

3.4.2 Inconsistencies 36

3.5 Overlay and Boolean Set Operations on Sets 38

3.5.1 Overlay Algorithm 38

3.5.2 Inconsistencies 40

3.5.3 Set Operations 41

3.6 Running Times 42

4 Transformation **43**

4.1 Algorithm 43

4.2 Inconsistencies and Degeneracies in Rotation of the Arrangement . . 49

4.3 Correctness 54

5 Overlay **58**

5.1 Algorithm 58

5.2 Inconsistencies 60

6 Testing Results **62**

7 Conclusion **69**

References **69**

List of Tables

6.1 Milenkovic and Sacks' algorithm 66

6.2 Test results for our algorithm 67

6.3 Error 1-3 results for Milenkovic and Sacks' algorithm 67

6.4 Error 1-3 results for our algorithm 67

6.5 Error 4-6 results for Milenkovic and Sacks' algorithm 68

6.6 Error 4-6 results for our algorithm 68

List of Figures

1.1 Parts of an arrangement 2

1.2 Minimal information 3

1.3 Inferring vertical order 4

1.4 Adding and removing vertices example 10

1.5 Floating point rounding when rotating 12

1.6 Floating point rounding when translating 12

1.7 Floating point rounding between non-neighboring points 13

1.8 VFV 14

1.9 VEV 14

1.10 EVE→VFV→VFV 15

2.1 Status tree T during the line sweep 17

2.2 Three event types: insert, swap, remove 18

2.3 Duplicate swap events 21

2.4 Inconsistent swap event 22

2.5 Pinching of segments 23

2.6 Circular ordering of subsegments at an event point 26

2.7 Upper and lower side of a subsegment 26

2.8 Generating loops 27

2.9 Grouping of loops to form cells 28

2.10 Forming of cells 28

2.11 Pairing of loops 29

3.1 Depth values in an annulus 31
3.2 Removing segments bounding cells with same depths 32
3.3 Winding number rules for cells 34
3.4 Flipping the orientation of a segment 35
3.5 Flipping the orientation of a segment with turning points 36
3.6 Winding numbers less than 0 from transformations 37
3.7 Winding numbers greater than 1 from transformation 38
3.8 Overlay of two arrangements 40

4.1 Vertical tangency from rotation 44
4.2 Missing minimal information for new event points 45
4.3 Doubly linked list of segments at an event point 46
4.4 Four possible place holders at an event point 47
4.5 Picking a place holder at an event point 48
4.6 Merging circular lists of two event points 50
4.7 Merging when event point has no incoming segments 51
4.8 Invalid turning point 51
4.9 Pinching after a rotation 54
4.10 Example of place holders at an event point 55
4.11 Two clusters A and B, at an event point 55

5.1 Red segment order is preserved 59

6.1 (a) Randomly generated curve, (b) Set generated from the random curve, (c) Arrangement a_0, (d) Arrangement a_1, (e) Arrangement a_3 . 62
6.2 The y-distance error between event point with incident segments . . . 64
6.3 The y-distance error between event point with the segment above . . 64
6.4 Closest distance error from an event point to a segment 65

Chapter 1

Introduction

1.1 Background

The background of this thesis lies in field of computational geometry. When computational geometers implement their algorithms, one of the main problems they deal with is the issue of *robustness*. Robustness is the degree to which a process can function correctly given the finite precision of the primitive numerical operations available on a computer. In this thesis, we want robustness in our *arrangement* algorithms. Section 1.2 discusses prior work in robust computational geometry.

Before stepping into the discussion of arrangements, we first introduce some terminology used throughout the thesis. A curve segment is a connected sub-manifold of the x, y plane that is the graph of a continuous function $y = f(x) : I(f) \rightarrow \mathbb{R}$; $I(f)$ is a finite closed interval, and let f denote $\{(x, f(x)) \| x \in I(f)\}$. Let $\min_x(f) = \min(I(f))$ and $\max_x(f) = \max(I(f))$. The segment has end points $\texttt{tail}(f) = (\min_x(f), f(\min_x(f)))$ and $\texttt{head}(f) = ((\max_x(f), f(\max_x(f))))$. A semi-algebraic curve is a curve segment that lies on an algebraic curve $F = \{(x, y) \| F(x, y) = 0\}$, where F is a polynomial in two variables with rational coefficients. In exact arithmetic, this means $F(x, f(x)) = 0$ for $x \in I(f)$. In floating point arithmetic, this means every point of f lies within some small distance ϵ of F. Additionally, in floating point arithmetic, $\texttt{tail}(f)$ and $\texttt{head}(f)$ may not lie exactly on F [11].

An arrangement of algebraic curves subdivides the plane into vertices, x-monotonic semi-algebraic curve segment edges, and semi-algebraic region cells (see Figure 1.1). In order to generate a regularized semi-algebraic set, we select some of the cells from the arrangement, and we take the closure of the disjoint union of the interiors of these cells [13]. This is represented by the set of segments of the arrangement which bound a selected cell from an unselected cell. These boundary segments form a special type of boundary arrangement:

- Intersections are only at end points of segments.
- There is an even number of *incident segments* at each vertex. An incident segment of a vertex is a segment that has this vertex as an end point.
- The regions, between each pair of incident segments at each vertex, alternate inside, outside.

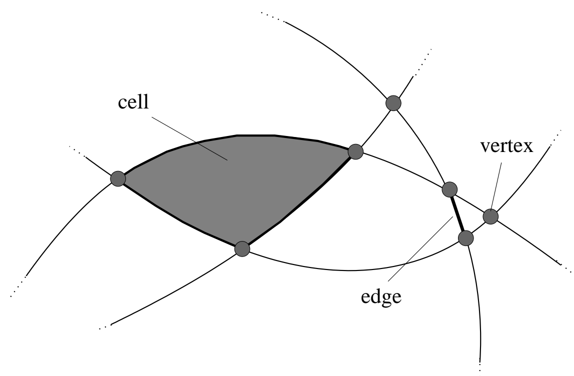

Figure 1.1: Parts of an arrangement

The combinatorial structure of the semi-algebraic set consists of its connected components and the semi-algebraic curves that bound them. The structure can be determined from the following minimal information: each vertex knows the vertical order of incident segments incoming from the left, it knows the vertical order of incident segments outgoing to the right, and it knows the segment above and below

it in y. The minimal information is illustrated in Figure 1.2. Coordinate-frame-independent structure, such as the circular order of incident segments at each vertex, can be derived from it.

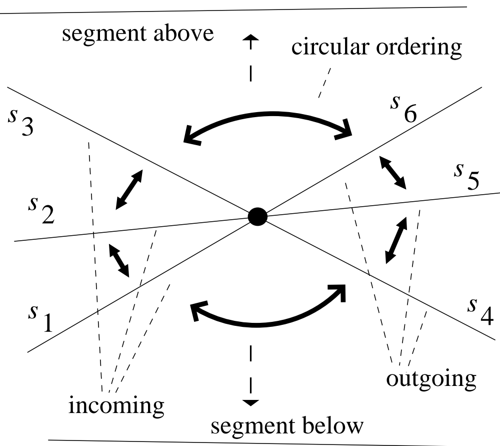

Figure 1.2: Minimal information

Each semi-algebraic set is represented by its boundary arrangement, and set operations on multiple semi-algebraic sets are done by overlaying them. In addition, a semi-algebraic set can be transformed by rotating or translating its boundary arrangement. If we can transform and overlay arrangements topologically invariantly, we can then perform topologically invariant set operations, such as union, intersection, difference, and set symmetric difference on the semi-algebraic sets.

Our problem is that rotation via recomputing the arrangement is inefficient; the *line sweep* recalculates intersections after a rotation. If the intersections are carried out in floating point, then almost always the intersection of rotated curves will not equal the rotation of their original intersection. Hence, rotating by resweeping will add additional vertices and other structures and not be topologically invariant.

In this thesis, we devised a way to generate the minimal information of the rotated set from the minimal information of the original set, and thus we maintain the combinatorial structure of the set: there is a one-to-one mapping of vertices, segments, and cells, each segment bounds two cells, each vertex has a circular order of segments incident at it, etc. This presents two challenges: combinatorial and numerical.

The combinatorial challenge is to infer the rotated minimal information without any numerical operations (e.g. intersection and vertical order tests). After a rotation, if every vertex has incoming segments (except the left-most), we can determine the rotated vertical order of segments without any numerical operations. However, if a vertex has no incoming segments after a rotation, then we are forced to use numerical operations. The solution is to add artificial segments called place holders in the original set such that each vertex of the rotated set either has a true incoming segment or an artificial incoming segment. We illustrate the idea of place holders in Figure 1.3. Segment t is a place holder, and it infers the vertical order of the rest of the outgoing segments at vertex p after a rotation. Without segment t, the algorithm is unable to infer the vertical order of those outgoing segments at vertex p without numerical operations.

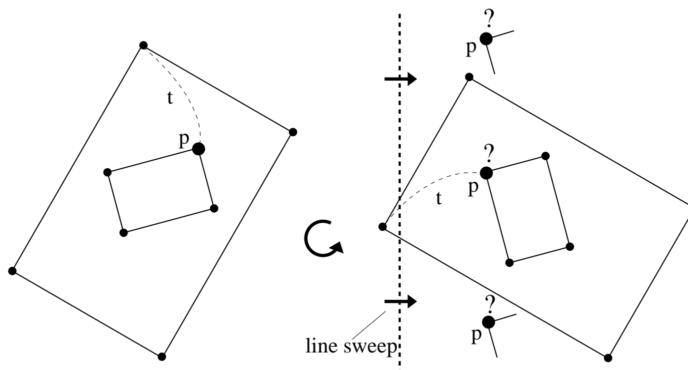

Figure 1.3: Inferring vertical order

The numerical challenge is to handle complications that arise when floating point rotation forces topological changes (e.g. floating point rounding). We handle these rare cases consistently via Euler operations. We discovered these unavoidable topological changes occur rarely in experiments.

We also devised a way to overlay multiple sets topologically invariantly. We prohibit self intersections, and we reuse the minimal information of each set. The same numerical issues of rotation arise in overlays.

The rest of this thesis is organized as follows. First, we provide the algorithm for constructing an initial arrangement of a set of semi-algebraic curve segments in Chapter 2. Afterward, we discuss the construction of a regularized set and Milenkovic and Sacks' method of transforming and overlaying a boundary arrangement in Chapter 3. Next, we discuss the topologically invariant algorithms separately: Chapter 4 presents our topologically invariant transformation algorithm, and Chapter 5 describes our topologically invariant overlay algorithm. Chapter 6 gives the testing results for both the algorithms by Milenkovic and Sacks, and our topologically invariant algorithms. Chapter 7 discusses our results and our conclusion.

1.2 Prior Work

There have been several approaches for robust geometric computations on arrangements. The following approaches are explained in turn: exact methods, perturbation methods, and numerical methods.

1.2.1 Exact Methods

Exact Geometric Computation (EGC) methods are robust, because they compute the combinatorial part exactly. EGC methods employ custom geometric algorithms, constructive root bounds, and floating point filters, to compute correct combinatorial structures. Yap [17] surveys this approach. His results are as follows:

- Keyser et al. [9] compute arrangements of non-degenerate rational curves with an $O(n^2)$ algorithm. Arranging 12 curves of degree at most 4 with 80 bit coefficients takes 1142 seconds on a 400MHz Pentium 2.

- LEDA [10] and CGAL [4] compute arrangements of line segments via generalizations of Bentley's sweep algorithm that employ filtered rational arithmetic.

- Wein [15] extends the CGAL arrangement algorithm to conics. Arranging 20 random conics take 2 seconds on a 450MHz Pentium 2.

- Berberich et al. [1] extend the LEDA arrangement algorithm to conics. Arranging 60 random conics with 50 bit coefficients takes 49 seconds on a 846MHz Pentium 3.

- Eigenwillig et al. [3] extend the LEDA arrangement algorithm to cubics. Arranging 60/90/120/250 random cubics with 100 bit coefficients takes 20/60/110/180 seconds respectively on a 1.2GHz Pentium 3.

- Geismann et al. [6] compute arrangements of special quartics (used to compute arrangements of 3D quadratics) with a sweep algorithm. Arranging 3 quartics with 30 bit coefficients takes 186 seconds on a Pentium 700.

- Wolpert [16] computes arrangements of nonsingular algebraic curves by an unimplemented sweep algorithm.

- Mourrain et al. [12] compute arrangements of 3D quadratics by an unimplemented plane sweep algorithm.

- Geismann et al. [6, 14] compute arrangements of 3D quadratics.

- Keyser et al. [8] compute arrangements of low-degree sculpted solids without degeneracies.

Exact arithmetic gives exact answers, but the computation is slow and not ideal. Exact arithmetic can not give Computer Aided Design (CAD)/Computer Aided Manufacturing (CAM) users quickly computed results. This slow approach is the reason other methods are necessary.

1.2.2 Perturbation Methods

Perturbation methods redefine the problem so that the troublesome inputs (degeneracies) are changed to less troublesome ones. Another perturbation approach is to redefine what the result should be, so that bad outputs seem “correct”.

Halperin and Leiserowitz [7] compute arrangements of circles by a perturbation method that does not need to compute the correct combinatorial structure. They

perturb the input so that each floating point computation is guaranteed to be correct with respect to the perturbed circles. Their method is useful when the perturbation is much less than the manufacturing accuracy, although the output may be incorrect for the input circles. Although they achieve polynomial running time, they perturb ill-conditioned coordinates, not just much less frequent inconsistent coordinates, and this results in much larger error, especially as operations are cascaded.

1.2.3 Numerical Methods

Fortune [5] surveys prior work on numerical methods for robustness in computational geometry that attempts to make direct use of floating point in the spirit of numerical analysis. He notes two major approaches. One approach formalizes geometric rounding. He points out that "the generalization to complex geometric objects is not straightforward." Milenkovic and Sacks [11] follow the other approach that Fortune outlines:

> A second floating-point approach is modeled on the error analysis of numerical methods, particularly linear algebra. The goal is to show that a suitably implemented algorithm provides an answer that is in some precise sense near the mathematically correct answer. Error analysis of geometric algorithms requires consideration of both combinatorial and numeric structure. Often it is easy to argue that an algorithm produces combinatorially valid output...at least with suitably relaxed requirements. It has turned out to be much more difficult to argue that the numerical error associated with combinatorial structure is small. Full error analysis has been carried [out] only for a few simple algorithms. [5].

Research by Milenkovic and Sacks [11] distinguishes itself from prior work by expressing the running time and error in terms of the number of *inconsistencies*. Approximate geometric computations can violate the laws of geometry, just as floating point operations can violate the laws of algebra. In Milenkovic and Sacks' computational model, this problem arises when the model assigns three curves an inconsistent, cyclic vertical order, which is an inconsistency. For example, the vertical order of three

curves is a is above b, b is above c, and c is above a. This vertical ordering is inconsistent, for this is not a valid stable vertical ordering; do not know which curve is above or which curve is below.

Milenkovic and Sacks consider the approximate computation as a separate module, and treat the number k of inconsistencies that it generates as an input property. They express the running time and the error in terms of k. They demonstrate experimentally that k is small, hence that the algorithm achieves "floating point speed with floating point accuracy". Moreover, they handle semi-algebraic curves, whereas prior work is restricted to linear objects.

1.3 Motivation and Goals

Using the algorithm by Milenkovic and Sacks [11], the resulting arrangements can be topologically variant. Let A be a constructed arrangement; that is, the cell structures have been created and the vertices and segments of each cell is identified accordingly. By this construction of the arrangement A, all intersections have been identified and the intersected segments are split by these intersections (this process is discussed in Chapter 2). If a transformation is applied to A, it is not guaranteed that two very close curves will not intersect each other after the transformation because of floating point inaccuracies; however, before the transformation, all intersections were supposedly detected. The consequence of a newly found intersection after a transformation is a change in the topology of arrangement A.

Christoph Hoffmann of Purdue University argued that the topology, once defined, should not change when transformed by a translation or a rotation. The topology changes because new intersections are found after a transformation or an overlay with another arrangement. This thesis builds on the work by Milenkovic and Sacks [11]. The arrangement construction in Chapter 2 is based off their work. The difference in the transformation and overlay algorithms between our work and Milenkovic and Sacks is discussed in Chapter 4 and Chapter 5, respectively.

1.4 Robust Topological Invariance

This section describes how we handle the numerical challenges of using floating point arithmetic. We allow for acceptable and unavoidable Euler operations in our robust topologically invariant algorithms:

- +EVE, splitting an edge at a new vertex;
- −EVE, joining two edges at a common vertex;
- VFV, combining two vertices, not connected by an edge;
- VEV, contracting an edge to a single vertex.

Before we explain the Euler operations, we first touch on the Euler formula. This formula determines if the topology of our arrangement is planar:

$$F - E + V = 2 \tag{1.1}$$

F is the number of faces, E is the number of edges, and V is the number of vertices. For example, a square has $F = 2$, $E = 4$, $V = 4$: $2-4+4 = 2$. By adhering to (1.1), we preserve the planarity of the topology. The situation is more complicated if the faces are not simply connected; however, the argument is essentially the same, because we can add virtual edges to make the faces simply connected.

1.4.1 Acceptable Euler Operation: Add/Remove a Vertex

We call “adding” and “removing” a vertex an EVE (edge, vertex, edge) operation. Look at the following example of the circle in Figure 1.4. We use x-monotonic curve segments to represent the object we are dealing with. In this case the object is a circle. We first break the circle into x-monotonic pieces by its *turning points*. Turning points are vertical tangent points on the curve. The circle is partitioned into the upper half and the lower half by its turning points, p_1 and p_2.

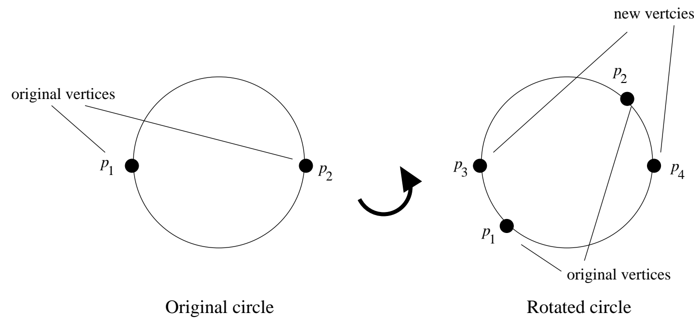

Figure 1.4: Adding and removing vertices example

The initial arrangement of this circle is made of two curve edges with two vertices that connect the two curves. If this circle is rotated by some angle θ, then we have to again break the rotated curve segments by their new turning points, p_3 and p_4. Adding new turning points is a +EVE operation. We take an edge and split it into two edges, while inserting a new vertex into the arrangement. The +EVE operation satisfies (1.1): increases E and V by 1. We consider the +EVE operation as acceptable because adding new turning points is necessary in describing the arrangement structure. If desired the two edges meeting the turning point can be considered part of a single "virtual" edge. Hence these +EVE operations do not change the underlying topology of "virtual" edges.

After the circle is rotated, the two former turning points, p_1 and p_2, no longer represent anything of importance, so they are removed. By removing a vertex, we merge two edges together. We call this a −EVE operation, and it is the reverse of the +EVE operation. This also satisfies (1.1): decreases V and E by 1. This excess information about the topology is not needed. Removing former turning points is necessary, for each new rotation can potentially split every edge and double the number of vertices, which can bog down computations. Again, if desired the two pre-rotated edges meeting the turning point can be considered part of a single "virtual" edge. Thus these −EVE operations allow for the same underlying topology of "virtual" edges.

The topology is robustly invariant under the acceptable EVE Euler operations of adding/removing a vertex. If we remove all the degree two (two incident edges)

vertices between the original circle and the rotated circle, then it is apparent the rotated circle was topologically unchanged from the original circle. The ±EVE operations happen regularly after every rotation of the arrangement. These operations occur during overlay of multiple arrangements as well, because the arrangements' edges intersect each other, and causes new vertices and split edges at each original arrangement.

1.4.2 Floating Point Rounding

Before explaining the unavoidable Euler operations, we touch on floating point rounding. Two numbers that ought to be different can round to the same value in floating point. Consequently, two points that ought to be different can round to the same point, if they belong to the same *rounding cell*. A rounding cell is a range of points that round to the same representable point.

Next, we explain the notion of *neighboring*. In floating point, two numbers are neighboring if there does not exist another number in between those two numbers. Consequently, two points are neighboring if they are neighboring in one of their dimensions and equal in the rest of their dimensions. We illustrate rounding through a few examples.

Example 1: Let us assume all rounding cells are of the same size. Let us use only exact arithmetic and represent numbers using floating point. Because all rounding cells are of the same size, the distance between two neighboring points (length of a rounding cell) is shorter than the length of the diagonal of a rounding cell. If we perform a Euclidean rotation on two neighboring points, it is possible both points end up in the same rounding cell after a rotation, see Figure 1.5 for a $2D$ example. However, if we perform a Euclidean translation, two neighboring points will never end up in the same rounding cell, since we have same size rounding cells.

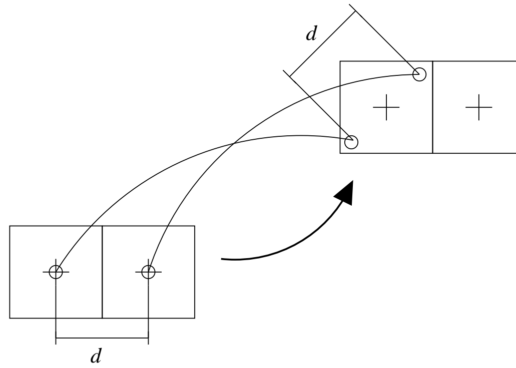

Figure 1.5: Floating point rounding when rotating

Example 2: Because we are using floating point representation, rounding cells can vary in size; the farther away the point is from the origin, the greater the size of the rounding cell. Therefore, in this example, let us use different size rounding cells, depending on their distance from the origin. Let us again use only exact arithmetic and represent numbers using floating point. Because rounding cells get larger as the point moves away from the origin, two neighboring points can round to the same cell after a Euclidean translation, see Figure 1.6 for a $2D$ example.

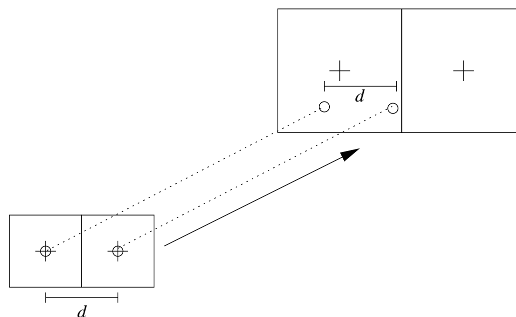

Figure 1.6: Floating point rounding when translating

Example 3: Because we want to compute quickly, we implement Euclidean transformations using floating point arithmetic instead of exact arithmetic. To perform a Euclidean rotation on a $2D$ point around the origin, the following formulas are used:

$$x' = x\cos\theta - y\sin\theta, \tag{1.2}$$

$$y' = x\sin\theta + y\cos\theta. \tag{1.3}$$

The values x, y are the initial coordinates of the point, θ is the angle of rotation, and the values x', y' are the new coordinates. For each formula, there are two multiplications followed by a sum of the two products. After each multiplication, the products are rounded to the nearest representable floating point number, summed together, and then rounded again. Because of the compounded error from more than one depth of calculations and rounding, non-neighboring points may be able to transform to the same rounding cell, see Figure 1.7 for a $2D$ illustration. This problem occurs even if we can guarantee $\cos^2\theta + \sin^2\theta = 1$; which might not be true since we are using floating point arithmetic.

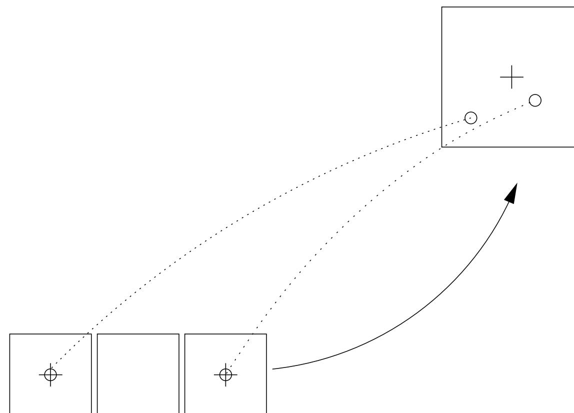

Figure 1.7: Floating point rounding between non-neighboring points

These three examples showed that rounding is unavoidable in Euclidean transformations when using floating point arithmetic or even just floating point representation. Consequently, this leads to the Euler operations that are labeled as unavoidable: VFV and VEV operations.

1.4.3 Unavoidable Euler Operations

Because of the rounding problem in floating point numbers, it was necessary for the next Euler operation: VFV (vertex, face, vertex) operation that combines two vertices and splits a face into two. We explain the necessity of this operation through an example, look at Figure 1.8. Let us have two very close together vertices that belong to two different curve segments, s_1 and s_2. As explained, rounding can cause these two distinct vertices to become the same identical vertex after rotation. We can not prevent these two vertices from round to an identical vertex during an Euclidean transformation, so this type of Euler operation must be allowed. The combining of both vertices splits the face that was between the two vertices into two. This satisfies (1.1): increases F by 1 and decreases V by 1.

The last Euler operation we discovered is similar to VFV, we call it a VEV (vertex, edge, vertex) operation. As explained before, two very close points can become one single point. If there was a curve segment that has these two very close points as end points, then that segment, after rotation, can deform into a single point, look at Figure 1.9 for an illustration. Contracting an edge, s_3, into a single point can not be avoided, so it needs to be accounted for. This also satisfies (1.1): decreases E and V by 1. Edges with same end points can cause our algorithms problems, so they are discarded.

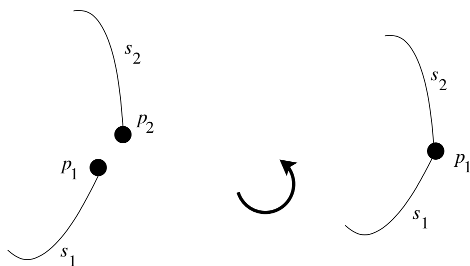

Figure 1.8: VFV

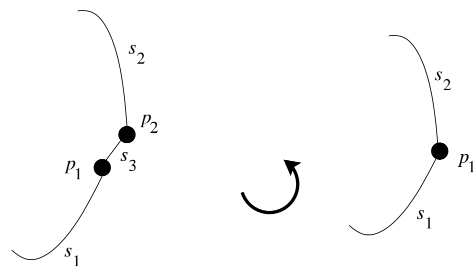

Figure 1.9: VEV

A more extreme case of the VFV operation is when there is another curve segment between the two very close vertices, look at Figure 1.10. After rotation, because the two very close vertices merged into one vertex, this extra curve segment s_3 is *pinched*

by the incident curves s_1 and s_2. Pinching means we intersect an edge to an existing vertex in order to maintain planarity (explained in further detail in Section 2.2). The first step is a +EVE operation to add a vertex to this pinched curved segment s_3 in order to split the edge into two. As we discovered earlier, +EVE operations are not considered changes to the topology and are allowed. After the +EVE operation, VFV operations are performed to combine the new vertex with the merging ones. Pinching can not be avoided when using floating point arithmetic, so this combination of Euler operations must be allowed. There can be more than one curve segment in between s_1 and s_2, so each will need a +EVE operation, and VFV operations afterward applied to all of them.

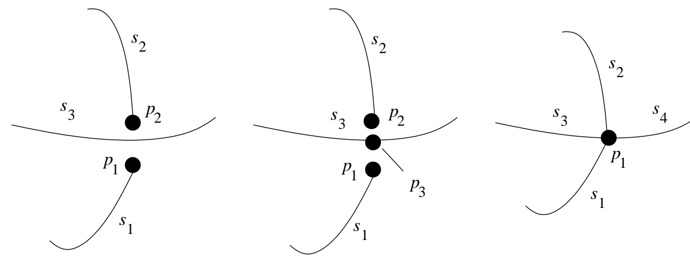

Figure 1.10: EVE→VFV→VFV

VFV and VEV Euler operations do change the topology; however, we still maintain robust topological invariance, for these operations can not be avoided due to rounding in floating point numbers. VFV and VEV Euler operations are one way - no operation for merging two faces by splitting a vertex into two vertices (no reverse VFV), and no operation for taking a vertex and stretching it into two vertices with a new edge in the middle (no reverse VEV). VFV operation occurs rarely. Pinching and rounding of vertices can happen during transformations and overlays of the arrangements. VEV operation is very rare, and occurred only once or twice during the entire testing.

Chapter 2

Arrangement Construction

In this chapter we explain how to robustly construct an arrangement when given a set L of $2D$ semi-algebraic curves. In order to construct a $2D$ arrangement from the input set L, we first have to find the intersections within L. This is done by doing a *line sweep* on L.

The input to a line sweep algorithm is a set L of x-monotonic curve segments defined by two endpoints: `tail`(f) and `head`(f), where the function f defines the curve. If the input set has curve segments that are not x-monotonic, then each non-x-monotonic segment is split into separate segments by its turning points.

The assumption in the exact case is that the input is generically oriented, meaning that if two vertices or segment intersections have the same x, they should also have the same y. This also implies no vertical line segments. For floating point input, the assumption has to be relaxed to equal x implies nearly equal y. Empirically, the floating point assumption can be made true by randomly rotating the input before sweeping it. This is a randomization technique Milenkovic and Sacks use in their sweep algorithm [11].

The goal of the line sweep algorithm is to find all the intersections in the input set L. Each segment $l \in L$ is partitioned into subsegments based on these intersections. The endpoints and intersections from the line sweep become the endpoints for the subsegments in the output set S. Section 2.1 discusses the line sweep algorithm. We explain what inconsistencies can occur during the line sweep in Section 2.2.

After the output set S is created, we create loops by traversing the vertices and edges, as explained in Section 2.3. Cells are formed by grouping of loops, as discussed in Section 2.4. After the arrangement is constructed, we explain how to create a regularized set from the constructed arrangement in Chapter 3.

2.1 Line Sweep Algorithm

The line sweep algorithm uses two data structures: a status tree T and a priority queue Q. The status tree T, called the sweep list, represents the vertical ordering of segments from lowest to highest along the vertical sweep line. T is implemented as a balanced binary search tree (e.g. red-black tree [2]) in which each node has a pointer to its predecessor and successor, and each data item has a pointer back to its node. We can insert a node into T in $O(\log n)$ time and find/remove a node in T in $O(1)$ time through the data item.

In the binary tree T, each node has one parent node and at most two child nodes, except the root which has no parent. For a given node and its segment s, the segments at the left subtree (the left child and its subtrees) are vertically below s in the sweep line. The segments at the right subtree are vertically above s. Thus, the in-order traversal of T gives the vertical list order of segments currently in the sweep, see Figure 2.1. Initially T is empty.

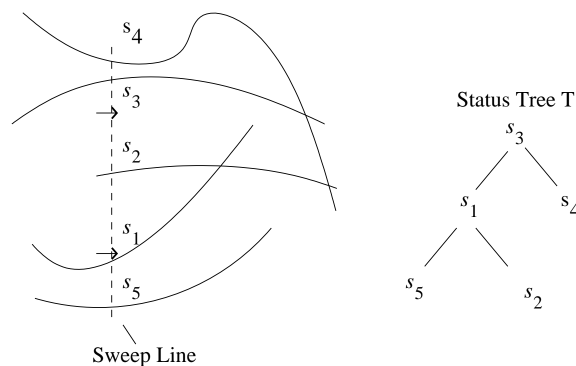

Figure 2.1: Status tree T during the line sweep

The data structure Q is a priority queue of *event points*. Event points are groups of *events* that have equal x-coordinates. Each event point in Q has a distinct x-coordinate. There are three types of events: insert events, remove events, and swap events, see Figure 2.2. Each event type represents a change in the status tree T:

- Insert event represents a new segment being inserted into the data structure T, as discussed in Section 2.1.1;
- Remove event represents an existing segment in T being removed, as explained in Section 2.1.2;
- Swap event represents two existing segments in T swapping vertical ordering, discussed in Section 2.1.3.

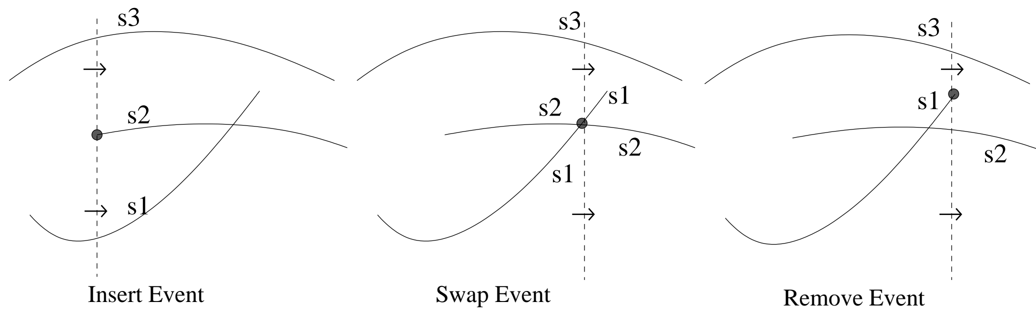

Figure 2.2: Three event types: insert, swap, remove

After events are grouped into event points, Q is sorted by the x-coordinate of these event points. Thus, the line sweep is simulated by processing the event points in Q in the sorted order. The line sweep algorithm terminates when all event points in Q are processed. We then partition each of the segments into subsegments based on the intersections found during the line sweep (swap event points). These subsegments and event points are used to form the loops and cells.

Each event point $q \in Q$ consists of at least one event. The order of events at any event point q is: remove events comes before swap events, and swap events before insert events. If event types are the same at an event point, then ties are broken arbitrarily.

2.1.1 Insert Event

When an insert event is reached in the priority queue Q, this means the line sweep has just encountered a segment $l \in L$ at its $\texttt{tail}(f)$ endpoint. This is shown in the left-most figure in Figure 2.2. Our first task at this insert event is to insert l to into T in a binary fashion.

In order to explain how we insert segment l into the binary tree T, we first explain how to utilize function f of each segment. Each segment has a function f that returns the y-value of the segment based on the x-value we give it; $y = f(x)$. For any given segment, by giving an x-value to its function f, we are able to determine the vertical ordering of that segment with respect to the rest of the segments in T, at that x.

The x-value we need for insertion is the x-value from the insert event (same as $\texttt{tail}(f)$ endpoint of segment l). If y for segment l is smaller than y for the current node's segment in T, we continue checking at the left child of the current node; initially the current node is the root node. If y for segment l is greater than y for the current node's segment, we continue checking at the right child. Ties, caused by same end point segments, are broken by checking the vertical order of both segments slightly to the right of x. We continue checking the left, right child nodes in T until we reach a current node with no left/right child. Segment l then becomes the segment of the new left/right child node, and l has now been inserted successfully into T.

Once the new node with segment l is inserted in T, we check for intersections with the segment above and below in T; segment above and below are found from the successor and predecessor of the new node in T, respectively. If an intersection is found, we add a swap event point to Q. This event point could possibly exist already in Q. If it exists, instead of adding a new event point to Q, we add a swap event to that existing event point.

2.1.2 Remove Event

At a remove event, the line sweep has just reached the end of a segment $l \in T$, and is at the $\texttt{head}(f)$ endpoint. This is the right-most figure in Figure 2.2. Since segment l

has a pointer to its node in T, we can remove the node from T in $O(1)$ time, and the tree and its nodes are updated accordingly.

It is important to note that we do not do a look up of segments currently in T, because using the function f at different x-values might yield unstable results. There is a possibility the segment might not be found due to floating point approximations using function f; the vertical ordering base solely on function f might differ from the current vertical order in T.

Once the node with segment l is removed from T, the predecessor and successor of that removed node are now adjacent. First we check for an *immediate swap* between the two newly adjacent segments. Immediate swaps take place between two segments, a and b, when they are newly adjacent in T, but are in the wrong vertical order. This happens because of error produced by floating point arithmetic. Originally, segment a is above the segment we removed, call it c, and c is above segment b. But once c is removed, b is checked to be above a at this event point. The check of vertical order is done using function f from each segment. This inconsistency is called an inconsistent triangle, for a is above c, who is above b, who is above a. An immediate swap is put in place at this event point, so that b can be above a, keeping a stable vertical ordering in the status tree T.

Swapping immediately removes the inconsistent triangle. Remove events come before swap events at every event point because of these possible immediate swaps, so that all remove events are done before all swap events. If there is no immediate swap between the two newly adjacent nodes in T, then we check for a future intersection between the pair. If an intersection is found, we add a new event point q with this swap event p to Q. The event p and the event point q could possibly exist already. If the event point q exists, we add the event p to that event point q'.

If the swap event p exists already as well, we do not add a duplicate swap event. The swap event exists because it was detected previously in the line sweep. The example in Figure 2.3 illustrates one instance of how a duplicate swap event could take place.

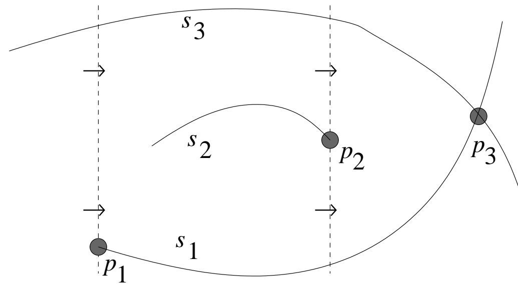

Figure 2.3: Duplicate swap events

Initially the insert of segment s_1 at p_1 caused a future swap event point, p_3, between segment s_1 and segment s_3. After segment s_2 is removed at p_2, segment s_1 and segment s_3 are adjacent again, and the same intersection at p_3 is found.

2.1.3 Swap Event

At a swap event, there are two adjacent segments that are incident to this event. These two adjacent segments intersect at this swap event, and we must switch vertical orders of these two segments in T, see the center figure at Figure 2.2. Each segment knows its node location in the status tree T. First we verify both nodes are adjacent in T by checking the predecessor and successor information accordingly. If they are adjacent in the correct order as indicated by the swap event, then we swap the nodes segments, look at the figure for an illustration. Segment s_1 and segment s_2 swap at point p. To swap them, we place segment s_1 of node a in node b, and then we place segment s_2 of node b in node a. Thus the vertical order in T after this swap event is consistent.

Once the segments are swapped, we check if the higher segment needs an immediate swap with the segment above. If not, then we check for a future intersection between this pair. We do a similar testing for the lower segment with the segment below. Two curves can intersect multiple times, so intersection tests are done with the two swapped segments themselves in case there is another future intersection. If intersections are found, we add swap event points to Q accordingly.

2.2 Inconsistencies

There are some *inconsistencies* during the line sweep algorithm. Inconsistencies are inconsistent topological data that may lead to a non-planar arrangement; inconsistencies are caused by floating point arithmetic. One inconsistency is when the swap event's segments may not belong to adjacent nodes in T. If they are not adjacent, then we drop the swap event; if those segments are not adjacent, we can not swap them. There must exist at least one segment between these two segments, so the switching of vertical orders of segments not adjacent in T are prohibited.

In Figure 2.4, three segments s_1, s_2, s_3 interact with each other between two event points p_1, p_2.

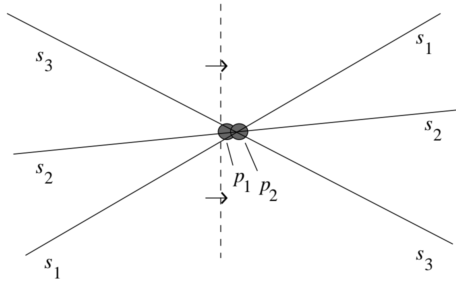

Figure 2.4: Inconsistent swap event

To illustrate an inconsistency, let $\{s_1, s_3\}$ swap at p_1, and let $\{s_1, s_2\}$ and $\{s_2, s_3\}$ swap at p_2. Since $p_1.x < p_2.x$, p_1 is encountered by the line sweep first. But since s_1 and s_3 are not adjacent, we do not swap them in T. If these two are allowed to swap, then the arrangement ends up non-planar. This swap event between $\{s_1, s_3\}$ is discarded, and the event point p_1 is deleted, for there are no more events. At p_2, without loss of generality, let $\{s_1, s_2\}$ swap first. Then, s_1 would be newly adjacent to s_3. An immediate swap between $\{s_1, s_3\}$ is detected, and is added to p_2. The two swaps left at p_2, $\{s_1, s_3\}$ and $\{s_2, s_3\}$, are now carried out as well, and the result is three segments intersecting at one event point, p_2.

Another inconsistency is *pinching*. Pinching associates segments to event points because the segment is "sandwiched" between the event point's incident segments, see Figure 2.5.

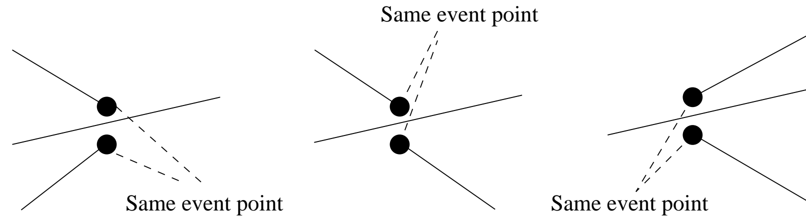

Figure 2.5: Pinching of segments

Pinching happens because at least two segments start/end at the same event point, and there exists non-incident segments in T that lie between these two segments in the vertical ordering. In Figure 2.5, a few examples of how a segment can be pinched are displayed. The left-most figure shows a segment pinched by two remove events, the center figure shows a segment being pinched by a remove event and an insert event, and the right-most figure shows a segment being pinched by two insert events.

Once segments are identified as pinched, then after the line sweep, we split the pinched segments at those pinching event points. If these pinched segments are not split into subsegments at those pinching event points, then the arrangement ends up non-planar. One of our requirements is for the arrangement algorithm to return a planar arrangement.

Here is the following algorithm we use to spot all pinched segments:

```
for each q in Q {
  infimum = null;
  supremum = null;
  for each event p in q {
    above = segment above p’s incident segment(s) in T;
    below = segment below p’s incident segment(s) in T;
    if (supremum == null or above > supremum) {
```

```
      supremum = above;
    }
    if (infimum == null or below < infimum) {
      infimum = below;
    }
  }
  for each segment l in T, from infimum to supremum {
    if (l not assciated with q)
      add a split in l at q; //segment l is pinched at event point q
  }
}
```

The comparison, `above` > `supremum`, is done not numerically, but instead symbolically by checking their locations in T, since T represents the vertical ordering of segments currently in the sweep list. For example, as explained earlier, an in-order traversal of the tree represents the vertical ordering of segments, if `above` comes after `supremum` in the in-order traversal, then `above` > `supremum` is true. This algorithm will find the highest segment not associated with this event point, and similarly the lowest segment. All segments in between will be "pinched" if not associated already.

The segment `infimum` is the segment in T that is below all segments associated with an event point. The segment `supremum` is the segment in T that is above all segments associated with an event point. Once this algorithm finds the `infimum` and `supremum`, all segments in T between these two are associated to this event point if they have not already. Those newly associated are the pinched segments.

2.3 Loops

In order to generate the cells needed to build an arrangement, loops are formed first. In order to generate loops, we first have to gather the minimal information:

- Each event point $q \in Q$ has to know the subsegment above this event point, and the subsegment below this event point. This information can be gathered from

the pinching algorithm. The segment `infimum` has one of its subsegments below this event point. Similarly, the segment `supremum` has one of its subsegments above this event point.

- Each event point $q \in Q$ has to know the order of all outgoing subsegments. Outgoing means these subsegments are coming out to the right. After all events from this event point are processed, the vertical list from the `infimum` to the `supremum` in T gives the order of all outgoing segments. To find the corresponding subsegments to these segments, we find the subsegment that has this event point as its left endpoint.

- Each event point $q \in Q$ has to know the order of all incoming subsegments. Incoming means these subsegments are coming in from the left of the event point. We can gather this information by making a copy of swap and remove events plus the `infimum` and `supremum` segments from T, and un-applying the events in reverse order. Once these events are reversed, the order between the `infimum` and `supremum` in this reversed, copied sublist of T gives the incoming order of the segments. To find the corresponding subsegments to these segments, we find the subsegment that has this event point as its right endpoint.

Since we know the order of all incoming/outgoing segments, we can generate the circular ordering of subsegments at each event point. From the lowest incoming subsegment, the next clockwise subsegment, at this event point, is the next subsegment in the incoming list, and the next clockwise subsegment from that subsegment is the next one in the list after that, etc. The clockwise order from the highest outgoing to the lowest outgoing is calculated similarly. To link both incoming and outgoing subsegments, the highest incoming segment's clockwise subsegment is the highest outgoing subsegment. The lowest outgoing subsegment's clockwise subsegment is the lowest incoming subsegment. Figure 2.6 shows an example of a circular clockwise ordering: $s_1, s_2, s_3, s_6, s_5, s_4$, and back to s_1. In addition to the clockwise ordering, the counter clockwise ordering is needed later as well. The counter clockwise order is just a reversing of the clockwise order.

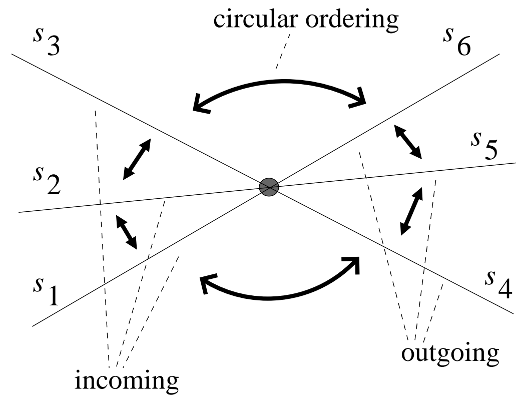

Figure 2.6: Circular ordering of subsegments at an event point

Every subsegment has 2 sides, call it `upper` and `lower`, see Figure 2.7. Both sides of the subsegments are associated to loops. Side `upper` associates itself to a loop from its left event point to its right event point. Side `lower` associates itself to a loop from its right event point to its left event point.

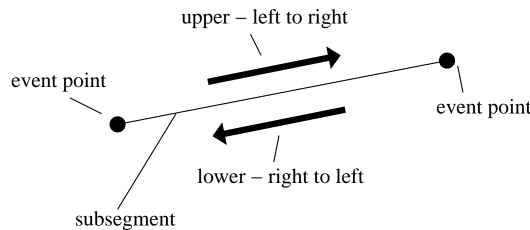

Figure 2.7: Upper and lower side of a subsegment

To form loops, we start with any side of any subsegment. That subsegment has two end points which are event points. At each event point of a subsegment, the subsegment knows the subsegment clockwise at this event point. To form a loop, if the algorithm is currently at the `lower` side of the subsegment, then the next subsegment in the loop is the subsegment clockwise to the left event point. Similarly, if the algorithm is currently on the `upper` side of the subsegment, then the next subsegment in the loop is the subsegment clockwise to the right event point. For example, in Figure 2.8, the loop starts at upper side of s_1.

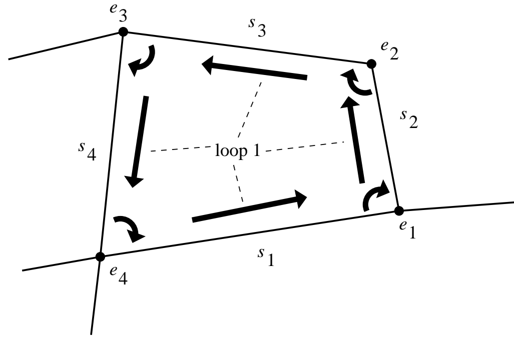

Figure 2.8: Generating loops

Since the algorithm starts at the `upper` side of subsegment s_1, the loop runs through this subsegment from the left event point to the right event point. The next subsegment in the loop is the subsegment clockwise at the right event point e_1, and this subsegment is s_2. Since e_1 is the right event point of s_2, that means the loop is currently at the `lower` side of s_2. The loop is at the `lower` side because the left event point e_2 has a smaller x-value than the right event point e_1. The loop continues to find subsegments in this clockwise fashion until the first subsegment s_1 is reached, meaning this loop is finished.

Once a loop is finished, we check to see if another loop exists by checking if both sides of all subsegments are associated to loops. If both sides of all subsegments in S are associated to loops, then the loop algorithm has successfully finished. Once the loop algorithm is finished, we can then form the cells.

2.4 Cells

Once all the loops are formed, the cells can be formed by the grouping of loops. We accomplish this by using the segment above and segment below information at each event point. At each event point, there is a highest and lowest outgoing and a highest and lowest incoming subsegment.

The highest incoming and highest outgoing subsegments at an event point are associated with the same loop. Based on the way loops are formed, the subsegment

above at this event point is either part of the same loop, or its part of a different loop in the same cell. If it is the same loop, then we continue. However, if the loop above is in a different loop, then these two loops point to the same cell, see Figure 2.9.

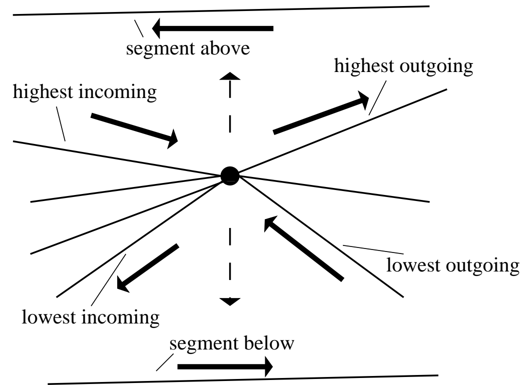

Figure 2.9: Grouping of loops to form cells

If there is no segment above, then this loop, formed from the highest incoming and highest outgoing, is part of the outer cell. Let us call the outer cell o. Similarly, if there is no segment below, then the loop, formed from the lowest incoming and lowest outgoing, is part of the outer cell o.

At an event point, if segment above is part of a different loop, then there are at least two loops in this cell. For each event point, we check for this grouping of loops above the event point and below the event point. Figure 2.10 shows an example of how loop l_2 and loop l_3 are grouped into the same cell c_2.

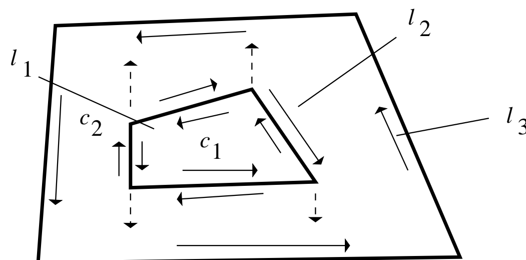

Figure 2.10: Forming of cells

The formation of cells is done by doing the classic union-find algorithm on the loop pairs [2]. After all union-find algorithm on the loops, the final cells for the example in Figure 2.11 are $c_0 = \{l_0\}$, $c_1 = \{l_1\}$, $c_2 = \{l_2\}$, $c_3 = \{l_3, l_4, l_5\}$. Cell c_0 is the outer cell that surrounds the arrangement. As shown in this example, each cell in an arrangement has at most one outer loop and can have multiple inner loops.

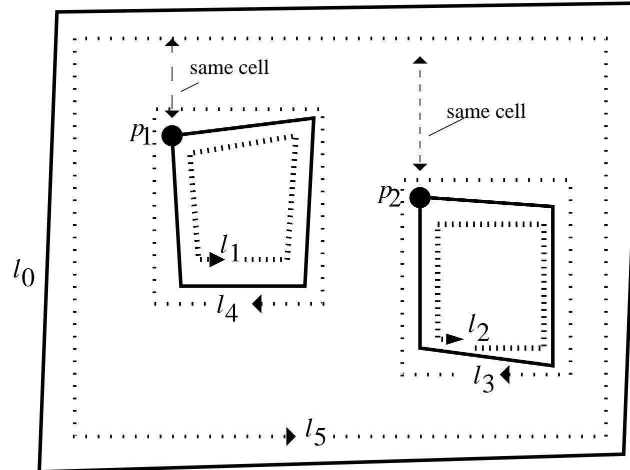

Figure 2.11: Pairing of loops

2.5 Running Times

The running time of the line sweep is $O((n + N + k)\log n)$, where n is the number of curve segments, N is the number of intersections, and $k = O(n^3)$ is the number of inconsistencies. Assume the intersection detection, and function value retrieval is constant.

The finding of loops takes $O(n+N+k)$ time, for there are $n+N$ subsegments, and the loop algorithm looks at each subsegment at most twice. The look up of next clockwise subsegment at each event point is $O(1)$. Pairing of loops is also linear time, for it looks at each event point only once, and just compares at most two loops together. The running time of the overall arrangement algorithm is $0((n + N + k)logn)$, since the line sweep algorithm dominates the running time.

Chapter 3

Regularized Semi-Algebraic Set

As explained in Chapter 1, in order to generated a regularized semi-algebraic set, we select a subset of the cells of the arrangement, and we take the closure of the disjoint union of the interiors of these cells [13]. The selection of some of the cells is arbitrary. In this chapter we show a natural way of selecting cells using depth values in Section 3.1. After selecting some of the cells, a boundary arrangement is formed that gives an oriented boundary of the regularized set in Section 3.2. We show another way of selecting cells using winding numbers in Section 3.3. With winding numbers, we are able to transform a single set or overlay multiple sets as discussed in Section 3.4 and Section 3.5, respectively.

3.1 Cell Selection by Depths

Consider an abstract undirected graph whose nodes are the cells. Two nodes are linked if the corresponding cells have a common edge. This is called the incidence graph of the cells. The depth of each cell is its distance from the outermost cell, where distance is defined by the number of links in the minimum link path in the graph. The depth of each cell can be determined by standard breadth first search (BFS) on the graph starting at the outermost cell [2].

We can select all cells of even depth or we can select all cells of odd depth. We illustrate the cell selection by depths in Figure 3.1. If we select cells of odd depth, then cell c_1 and c_3 are outside and cell c_2 is inside.

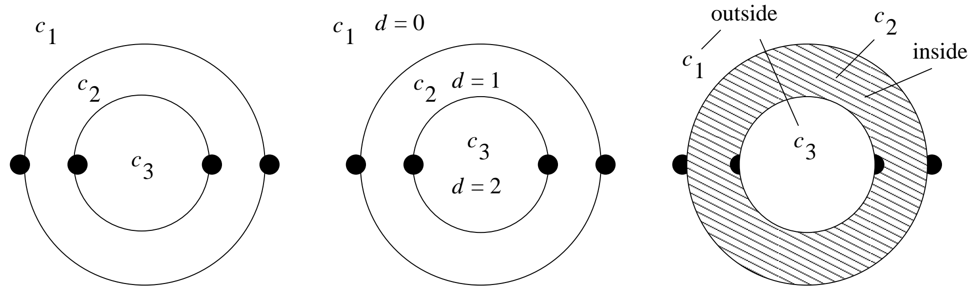

Figure 3.1: Depth values in an annulus

3.2 Creating an Oriented Boundary Arrangement

Given a selection of cells, we can create the oriented boundary of the regularized set. First, we discard any segments of the arrangement that do not bound a selected cell from a not-selected cell. For example, when we select cells using depths, the discarded segments are those which bound cells of the same depth. The removal of these extra segments is allowed since we ultimately want a regularized semi-algebraic set; the closure of the interior of a cell will count these removed segments as part of the cell.

Two cells selected from the original arrangement may merge into a single cell of the boundary arrangement. Figure 3.2 shows and example of removing segments that do not bound a selected cell from a not-selected cell. Segment s_1 bounded two cells of the same depth: c_2 and c_3 have a depth of $d = 1$. We remove segment s_1, since both c_2 and c_3 are selected cells.

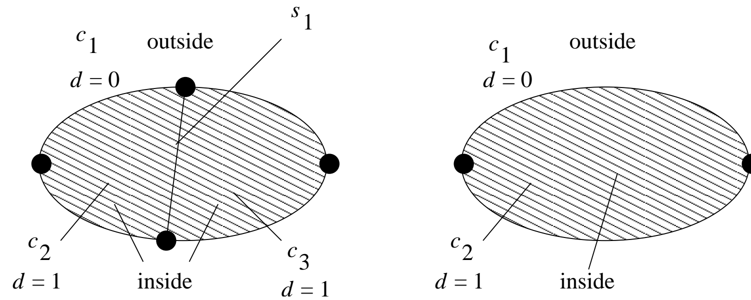

Figure 3.2: Removing segments bounding cells with same depths

Removing superfluous segments invalidates the minimal information, particular if the segment immediately above or below a vertex is removed. We restore the minimal information as follows. Before segment removal, a partial order "is below" on the segments can be defined from the minimal information. For each segment:

- If the segment is not the lowest outgoing at its left event point, then the segment clockwise at its left event point "is below" it.
- Similarly, if the segment is not the lowest incoming to its right event point, then the segment counter clockwise at its right event point "is below" it.
- If the segment is the lowest incoming or outgoing segment at its respective event point, then the segment immediately below the event point "is below" it.
- Similarly, if the segment is the highest incoming or outgoing segment at its respective event point, then it "is below" the segment immediately above the event point.

With the partial order of "is below", we can establish a total order on the segments. The standard depth first search (DFS) algorithm [2] assigns an integer height to each segment such that if a is below b, then height of a is less than height of b. We run the arrangement algorithm on the remaining segments, and run a resweep using the newly assigned height values to represent the vertical order of the segments in the status tree T. With the new sweep, no new intersections are found, so no intersection

tests take place. If intersection tests are allowed, intersections can possibly be found due to floating point error. After the resweep, minimal information is restored.

After discarding the non-boundary segments of the arrangement, we orient the remaining segments so that the selected cell is on its left. The orientation is done by assigning a direction to each segment based on which side the selected cell is on. There are only two directions: `ltor` (left to right) or `rtol` (right to left), and each subsegment can have only one direction. If the direction is `ltor`, the cell on its left (the upper cell) is a selected cell. If the direction is `rtol`, the cell on its left (the lower cell) is a selected cell.

3.3 Assigning Winding Numbers

Once the resweep is finished, loops and cells are formed again. After the resweep, we select cells based on winding numbers, which are in turn based on the directions we assigned earlier.

These directions determine which cell is selected and which cell is not-selected. For a given cell c, let $c_{\texttt{wind}}$ represent the cell's winding number. If cell c is selected, $c_{\texttt{wind}} = 1$; otherwise, $c_{\texttt{wind}} = 0$. Winding numbers are important for transformations and overlays, explained later in Section 3.4 and Section 3.5, respectively.

There are four rules when assigning winding numbers to a cell, as seen in Figure 3.3. Two of the rules are opposites of the other two. Based on the `lower` and `upper` side of the subsegment and the direction assigned to the subsegment, a winding number is given to the cell.

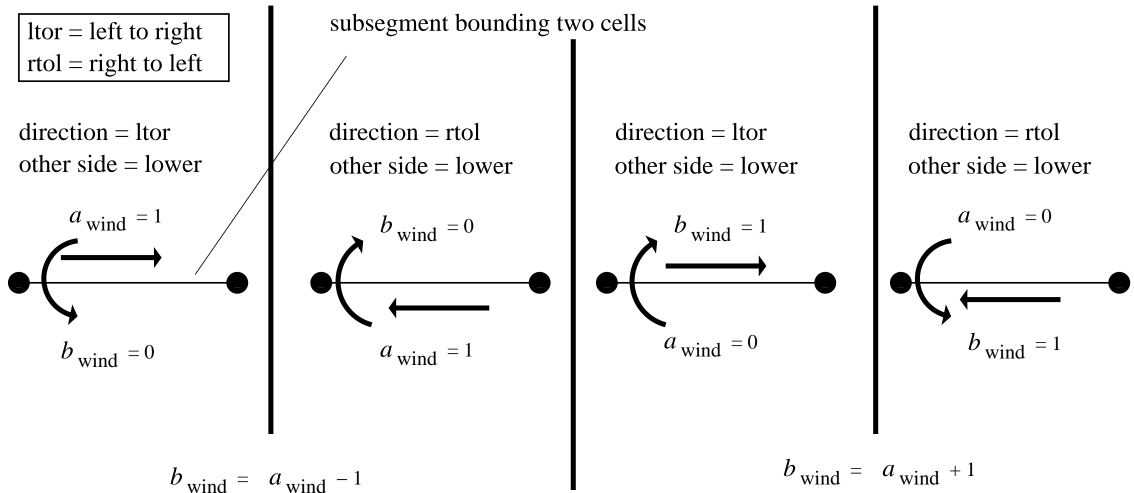

Figure 3.3: Winding number rules for cells

A similar BFS algorithm is used to assign winding numbers as previously done with assigning depths. Starting with the outermost cell o as the first element in our BFS queue, we give the outermost cell $o_{\texttt{wind}} = 0$ if we decide that the outermost cell is outside, or $o_{\texttt{wind}} = 1$ if we decide that the outermost cell is inside. We go to all adjacent cells of o. We apply the rules from Figure 3.3 to these adjacent cells. Since the algorithm is starting from o in-ward, let cell a represent cell o, and let cell b represent an adjacent cell.

So, for example, if the subsegment we are looking at has `direction` = `ltor`, `otherside` = `lower`, then we assign $b_{\texttt{wind}} = a_{\texttt{wind}} + 1$. Winding numbers can be assigned by the incidence graph. Assignment of winding numbers is a key subroutine of the algorithms for transforming or overlaying regularized semi-algebraic sets.

3.4 Transforming Sets

3.4.1 Algorithm

Rotating and translating an arrangement in Milenkovic and Sacks' algorithm requires transforming each subsegment of a set S individually. Once this set S is transformed,

the minimal information gathered from the initial arrangement algorithm becomes outdated; we no longer know what is above and below each event point.

In order to restore this information, we build the arrangement again with a resweep. Since the previously built height information from Section 3.2 has no more meaning after a rotation, the height values, which are based on the vertical order, are cleared. After a transformation, one of the following occurs for each segment:

- It had no turning points from the transformation, and its tail end point is still smaller than its head end point; $\texttt{tail}(f).x < \texttt{head}(f).x$. The segment maintains its original orientation.

- It had no turning points from the transformation, but its tail end point is now larger than its head end point; $\texttt{tail}(f).x > \texttt{head}(f).x$. The segment has "flipped" around due to the transformation, thus we must "flip" the orientation, because the orientation determined which side had a selected cell. In Figure 3.4, segment s_1, for example, has flipped around, so its new orientation is now `rtol`, instead of its original orientation of `ltor`.

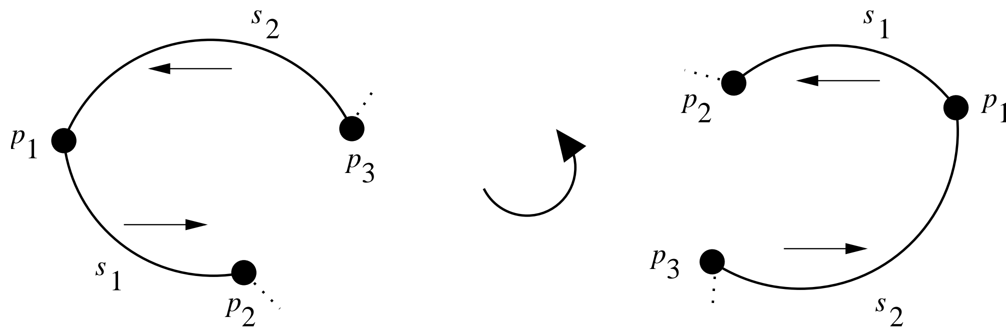

Figure 3.4: Flipping the orientation of a segment

- There were turning points from a transformation at this segment, breaking the segment into separate segments. Not all of these separated segments will have $\texttt{tail}(f).x < \texttt{head}(f).x$, for some of the segments were "flipped" from the transformation, see Figure 3.5 for an illustration. We flip the orientation for the segments that were completely "flipped" around. If they did not flip, then

they retain the original orientation. In Figure 3.5, s_2 breaks into s_3 and s_4 at p_4. Segment s_3 retains the orientation from s_2, while s_4 flips its orientation due to the turning point p_4.

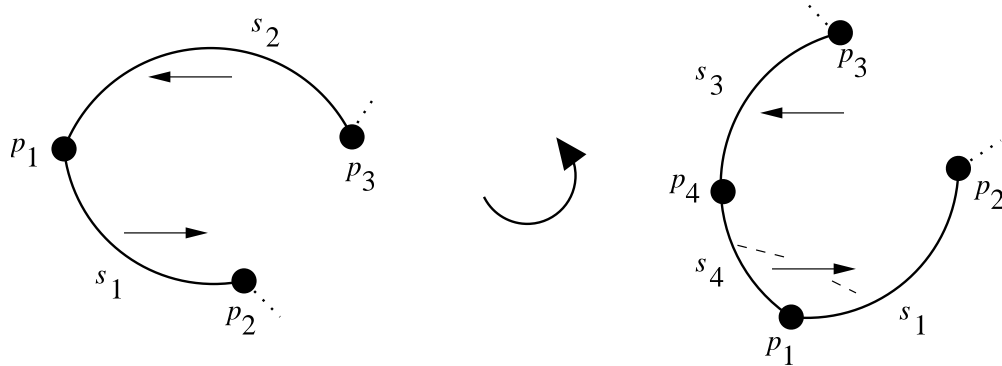

Figure 3.5: Flipping the orientation of a segment with turning points

Because the segments have no more valid height values, the new sweep may encounter new intersections, for we have to numerically re-establish the vertical order of the segments during the resweep. After the sweep is complete, the loops and the cells are created as previously done in Chapter 2. We use the orientation of the segments to re-establish which cells are selected. Because of new intersections, new cells are formed in our arrangement, and these cells end up with invalid winding numbers. The method of dealing with invalid winding numbers is explained in the next subsection.

3.4.2 Inconsistencies

There is an inconsistency in rotation of the arrangement on the winding numbers. Because of new intersections from the line sweep, more cells are formed within our arrangement. Because these cells were from the new intersections, the cells end up with winding number of $\texttt{wind} < 0$ or $\texttt{wind} > 1$. Initially only winding numbers that were assigned were $\texttt{wind} = 0$ or $\texttt{wind} = 1$.

Figure 3.6 is an example of how new intersections generate invalid cell winding numbers. Before rotation in the figure, segments s and t share a common end point p_1. After rotation, s intersects t at p_2'. Segment s breaks into two subsegments: s_1 and

s_2, and segment t breaks into two subsegments as well: t_1 and t_2. Each subsegment retains its original segment's direction. We now apply the first rule, from Figure 3.3, onto subsegment s_1. Because the current cell (the outer cell) has $\texttt{wind} = 0$, then the cell on the other side of s_1 has $\texttt{wind} = 0 - 1 = -1$. The cell gets $\texttt{wind} = -1$ as well if the second rule from the figure is applied to subsegment t_1.

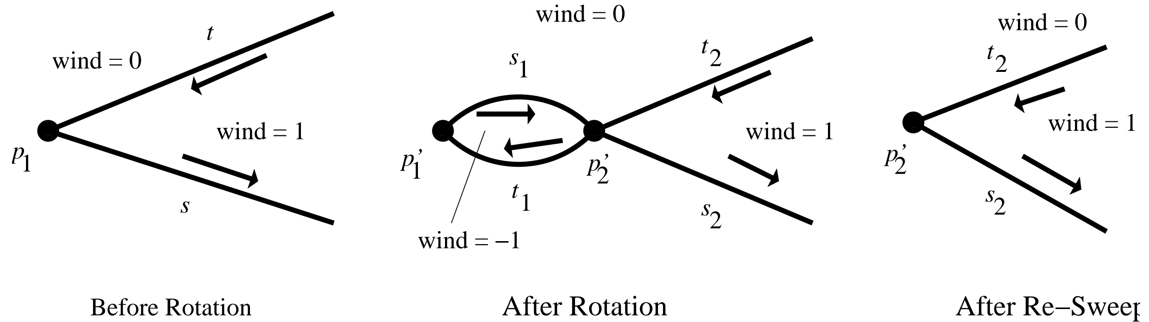

Figure 3.6: Winding numbers less than 0 from transformations

To solve this problem of negative winding numbers, we consider all negative winding numbers to be outside the arrangement. Since $\texttt{wind} = 0$ means outside, if $\texttt{wind} < 0$, we make that cell's winding number $\texttt{wind} = 0$. Since all subsegments that do not bound a selected cell from a not-selected cell are discarded, subsegments s_1 and t_1, from our example in Figure 3.6, are discarded. The figure on the right in the example shows the arrangement at this event point p'_2. If winding numbers were not used, then small cells, caused by new intersections, are kept. Creation of small cells is not ideal, for these two segments should have only intersected once. They were allowed to intersect again, because we lost vital information about the topology of our arrangement when we rotated. This forced the algorithm to redo intersection tests.

Figure 3.7 is another example of how new intersections can cause invalid cell winding numbers. Before rotation, two cells are very close to one another, almost touching. They are so close that after rotation, the two cells have merged into one. This merging causes two new intersections between these two cells.

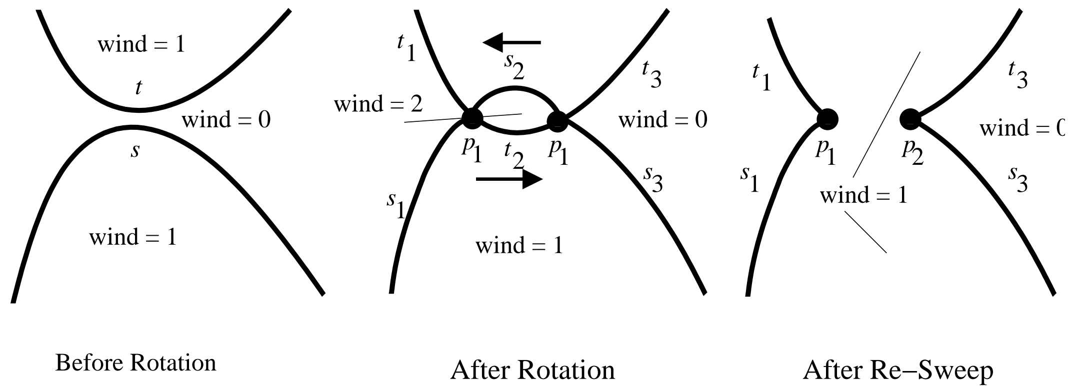

Figure 3.7: Winding numbers greater than 1 from transformation

Based on rules three and four in Figure 3.3 on subsegment s_2 and t_2, respectively, then the cell formed from this merging has `wind` $= 2$. If winding numbers are not used, then these little cells are considered outside if we use depth numbers from Section 3.1. But since we have already defined the topology using the directions, we consider all `wind` $= 2$ cells inside the arrangement. Thus, we convert all `wind` > 1 to `wind` $= 1$. Subsegments s_2 and t_2 are removed because they bounded cells of same winding numbers. This removal of the subsegments cause the two cells to merge into one bigger cell, as illustrated in Figure 3.7.

The problem with merging of cells is that it destroys the original topology of the arrangement. Our topologically invariant transformation algorithm in Chapter 4 shows how we transform the boundary arrangement while keeping the topology unchanged.

3.5 Overlay and Boolean Set Operations on Sets

3.5.1 Overlay Algorithm

Overlay of two or more arrangements in Milenkovic and Sacks' algorithm required the segments from the two or more arrangements to be placed into a single set. This newly combined set of segments becomes the input set S to the arrangement algorithm from Chapter 2.

The segments in the input set S no longer have valid height values that were generated from Section 3.2. These old height values only represent the vertical segment order within each segment's original arrangements; they don't define the vertical order of segments in the overlay. The line sweep has to recompute the vertical order with new numerical operations, and in the process new intersections are detected. One arrangement's segments can intersect with another arrangement's segments in the new line sweep. Additionally, segments can intersect with other segments from its own original arrangement (self-intersections), because of new intersections caused by floating point arithmetic.

Like transformations, from Chapter 4, each segment retains the orientation assigned to it from their original arrangement. Some intersections could occur, so each subsegment retains its original segment's direction.

Loops and cells are created as previously done in Chapter 2. No depths are given since the arrangement, or the topology, is already defined, so winding numbers are assigned to cells. This difference between overlay and transformations is that each cell has n winding numbers, where n is the number of overlaying arrangements. The first winding number represents the first arrangement, the second winding number represents the second arrangement, etc. Each arrangement is given a unique integer value, from 0 to $n-1$, and the value is stored in each of the arrangement's segments.

We use a BFS algorithm to assign winding numbers to the cells, as previously done in Chapter 2. The four rules for winding numbers are used from Figure 3.3. The difference in the overlay algorithm is that we have n winding numbers for each cell; however, we only apply winding number rules to the i-th winding number of a cell, where i is the segment's arrangement integer value. The rest of the winding numbers in the cell have no winding rules applied to them, and are carried over from the adjacent cell that is on the other side of this segment. Figure 3.8 illustrates an example of how winding numbers are assigned.

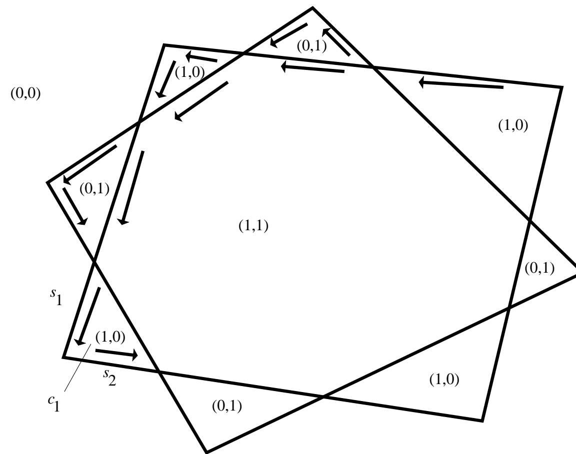

Figure 3.8: Overlay of two arrangements

In the example above, there are two arrangements, arrangement A and arrangement B. Each cell in the overlay of the arrangement has two winding numbers, (X, Y), where X is for A's winding number and Y is for B's winding number. When the winding number algorithm is ran, it starts from the outer cell (0,0) inward. Look at cell c_1. Applying the winding rules on segment s_1 and s_2, the winding number for this inner cell changes to $\texttt{wind} = 0 + 1 = 1$. Since both segments belong to arrangement A, then the winding numbers at cell c_1 is $\texttt{wind} = (1, 0)$, where the second winding number was carried over from the outer cell.

3.5.2 Inconsistencies

There is an inconsistency in overlay of multiple arrangements on the winding numbers. Because of new intersections from the line sweep, more cells are formed from each arrangement within itself. These self intersections result in cells with winding number of $\texttt{wind} < 0$ or $\texttt{wind} > 1$, similarly to inconsistencies found in Section 3.4.2. We solve the inconsistencies in the same fashion: $\texttt{wind} < 0$ turns into $\texttt{wind} = 0$, and $\texttt{wind} > 1$ turns into $\texttt{wind} = 1$. Our topologically invariant overlay algorithm does not allow for self intersections.

3.5.3 Set Operations

Once the overlay algorithm is finished, we can preform set operations on the overlay. For each cell of the overlayed arrangement, there are n winding numbers, each representing one of the n arrangements. A winding number is $\texttt{wind} = 0$ at the i-th position means this cell is an outer cell at the i-th arrangement. If the winding number is $\texttt{wind} = 1$ at the i-th position, that means this cell is a selected cell of the i-th arrangement.

Since we can determine which cell is selected and not-selected in each of our arrangements, we can perform set operations. The following are a few set operation examples:

- **Union** - For a union set operation, the objective is to find the union of all the arrangements. With winding numbers, each cell that has any $\texttt{wind} = 1$ is considered a cell that is a part of the union. So at each cell, add up all the winding numbers and call the total `windtotal`. If $\texttt{windtotal} > 0$, that means this cell is part of the union of the arrangements, else it is not.

- **Intersection** - For an intersection set operation, the objective is to find the intersection of all the arrangements. An intersection means all the winding numbers of a cell has to be $\texttt{wind} = 1$. If any winding number of a cell is $\texttt{wind} = 0$, that means this cell is not a part of the intersection of the arrangements.

- **Set Symmetric Difference** - Set symmetric difference operation only applies to two arrangements. Set symmetric difference means the element belongs to one set and not the other set. For example, for $A = \{1, 2, 3, 4\}$ and $B = \{1, 4, 5\}$, $A \ominus B = \{2, 3, 5\}$, since 2, 3, 5 are each in one, but not both, sets. For our overlay, set symmetric difference keeps cells belonging to one arrangement and not the other. Since there are only two arrangements, all cells with $\texttt{wind} = (0, 1)$ or $\texttt{wind} = (1, 0)$ are considered cells part of the set symmetric difference of the two arrangements. Cells with winding numbers of $\texttt{wind} = (1, 1)$ or $\texttt{wind} = (0, 0)$ are not considered a part of the set symmetric difference.

When the set operation is performed on a cell, if the cell is considered selected based on the set operation, give the cell one new winding number of `wind` $= 1$. After the set operation, each cell has only one winding number instead of n winding numbers, because the set operation combined all the sets. If the cell is not considered selected based on the set operation, then we give the cell one winding number, `wind` $=$ 0. Once the winding numbers are assigned, we consider all segments part of one new arrangement and are given a new orientation based on these new winding numbers for the cells, similarly to how it was done in Section 3.3. If segments do not bound a selected cell from a not-selected cell, then we remove those segments in order to form the boundary arrangement. The arrangement algorithm is ran one more time to rebuild the the final cells and final winding numbers.

3.6 Running Times

The running time of the BFS algorithm takes $O(V + E)$ time, where $V =$ number of cells, and $E =$ number of subsegments $= N+n+k$; n is the number of curve segments, N is the number of intersections, and $k = O(n^3)$ is the number of inconsistencies. There can be no more cells than subsegments, so the running time is $0(V + E) = 0(N + n + k)$.

For the DFS algorithm, it takes $0(V1+E1)$, where $V1 =$ number of subsegments, and $E1 =$ number of directed edges, which is twice the number of subsegments. The running time for the DFS is $0(V1+E1) = O(N+n+k)$. The overall running time is still, however, dominated by the line sweep algorithm, which is $0((n + N + k)logn)$.

Chapter 4

Transformation

This chapter presents our topologically invariant transformation algorithm. Section 4.1 describes the algorithm. Section 4.2 explains the inconsistencies and degeneracies that can happen in the algorithm. Section 4.3 gives the correctness of the algorithm.

4.1 Algorithm

Our current problem with the arrangement algorithm is as follows. Once an arrangement is generated from a list of input x-monotonic curve segments, a certain topology is created. We believe that once this topology is set, operations such as translation or rotation should not alter this topology in any way. However, due to floating point inaccuracies, new intersections are found after the translation/rotation, and thus changes the topology. We feel this alteration should not happen.

An arrangement, as we defined it, has event points and x-monotonic segments. From the minimal information gathered during the line sweep, we know the ordering of incoming segments from the lowest to highest and similarly outgoing segments, plus we know the segment directly above and below this event point.

Before we can transform the set and for simplicity of the arrangement algorithm, we have to preprocess the set as follows:

- We pre-split curves in the original set such that they are x-monotonic in the rotated set.

- We uniquely determine the vertical order of incident segments at the *left-most* vertex, so that we have a starting point in the resweep. Left-most means smallest x-valued vertex.

- We add artifical edges called place holders to ensure each vertex (except the left-most) of the rotated set has either a true incoming segment or an artificial incoming segment.

Pre-splitting curves We pre-split curves because we represent the combinatorial structure of the semi-algebraic set with x-monotonic segments, and rotation may make them non-x-monotonic. We solve this issue by splitting each segment at the inverse rotation of its turning points that are found from rotating the segment, see Figure 4.1. More than one vertical tangency can occur at each curve, so the curves can be split more than once.

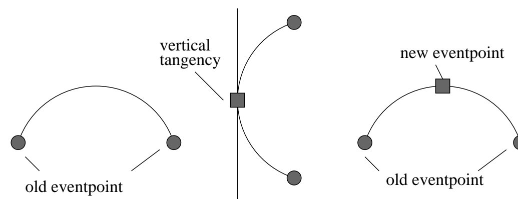

Figure 4.1: Vertical tangency from rotation

The transformation algorithm needs the minimal information of the original set to generate the minimal information of the rotated set. That is a problem since splitting segments into separate segments invalidates the minimal information of the original set. In addition, the original set has new event point vertices, from segment splits, that lack minimal information. We can restore the minimal information at original event point vertices without a resweep as follows. For old event points, the procedure is as follows. If the segment is an incoming segment, this event point points to the last section of that segment. If the segment is an outgoing segment, this event point

points to the first section instead. If the segment above or below is broken up into pieces, then this event point points to the correct subsegment of that segment that spans this event point's x-coordinate.

For new event points, there is only one incoming and outgoing segment, and these are the new segments that meet at a turning point after the rotation. The re-sweep of the line is needed, for we do not know what segment is above or below this new event point.

In Figure 4.2, s_1 and s_3 are the original above segments, but nothing about s_2 is retained in those old event points. If we re-sweep the arrangement, this information is retained in the binary balanced tree T, for s_2 is the segment that is adjacent in T and is above the new event point. However, we do not need a balanced tree for the sweep list, because we already have the vertical order from the original arrangement. All we need now is a doubly-linked list D, for a less costly data structure.

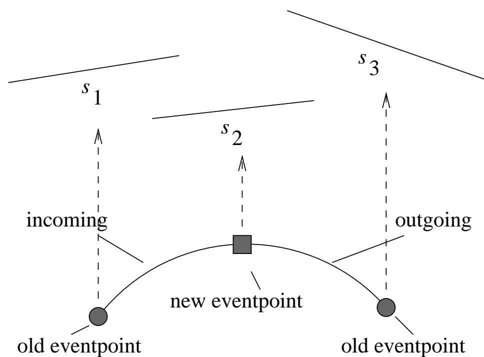

Figure 4.2: Missing minimal information for new event points

At an old event point, we can remove the incoming segments easily in a doubly-linked list, for these incoming segments are the segments in between this event point's above and below segment. Figure 4.3 shows an example of how the doubly-linked list D is used to represent the vertical ordering of segments at the event points.

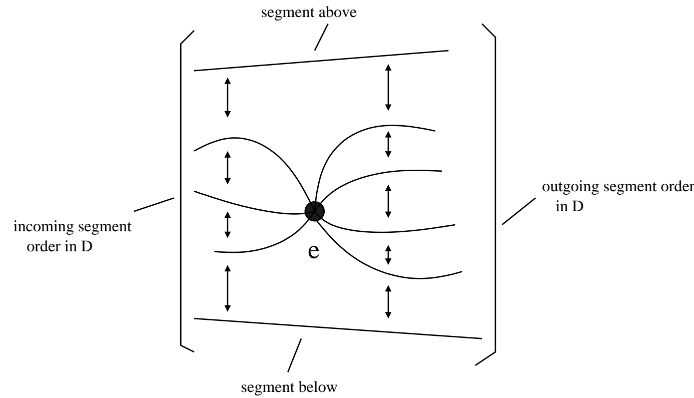

Figure 4.3: Doubly linked list of segments at an event point

To remove all segments incoming at an event point in D, point the segment above to the segment below, and vice-versa. We know the order of all outgoing segments incident to the event point, so we can add the outgoing segments to the same place in D between the segments that were above, below the incoming segments. We can link them into the linked list in their correct order, with the lowest outgoing connected to the segment below, and the highest outgoing connected to the segment above.

For the new event point, finding the segment above and below is trivial since the one incoming segment at this new event point had already been inserted into our doubly linked list. We need to look at the adjacent neighbors of this incoming segment in D to determine the segments above, below. The cost of the sweep is $O(1)$ per segment. However, it is still $O(\log n)$ per event point because we have to sort them by x. Now our data structure has minimal information.

Left-most vertex segments The next preprocess step is to be able to uniquely determine the rotated vertical order of incident segments at the left-most event point, so that the resweep has a starting point. We solve this issue by adding a bounded box: (M, M), $(M, -M)$, $(-M, -M)$, $(-M, M)$, where M is arbitrarily large to incorporate the input set within. After a transformation, the left-most corner of the bounded box is the left-most event point, and it will have only outgoing segments. Also, we can uniquely determine the rotated vertical order of the lowest and highest outgoing segments at this left-most corner since they are segments that formed the bounded

box. Since we know the circular ordering of incident segments at each event point from the minimal information, we can determine the order of the rest of the outgoing segments (between the lowest and highest) at this vertex.

Place holder The last preprocess step is to add place holders to ensure each event point of the rotated set has either a true incoming segment or a place holder as an incoming segment. A place holder is generated as follows: at each event point, there is a segment above and below it (could be the bounded box segments). There can be four possible place holders at each event point: two connected to the end points of the segment above, and two connected to the end points of the segment below. Figure 4.4 shows an example of the four possible place holders p_1, p_2, p_3, and p_4 at an event point e. We choose the end point that satisfies the property that the event point has an incoming segment.

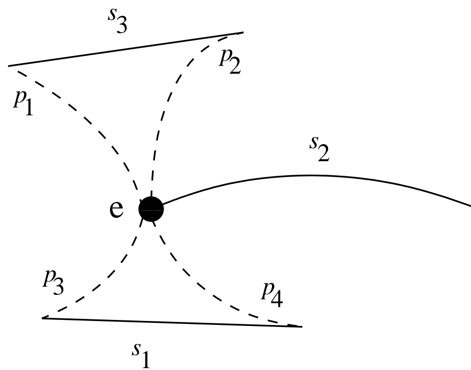

Figure 4.4: Four possible place holders at an event point

To create the place holder, we need to figure out which corner of the bounded box is left-most after rotation. If the left-most is the upper left/right corner, then the place holder chosen is with the smallest x-valued end point of the segment above; there exists at least one point on this segment above that is smaller than this event point after rotation. See Figure 4.5 for an illustration. Segment s_2 is above segment s_1 at event point p_1. At point p_2 on segment s_2, $p_2.x == p_1.x$ and $p_2.y > p_1.y$. So if upper left boundary corner is left-most, then after rotation, $p_2.x' < p_1.x'$, and the line sweep hits p_2 before p_1.

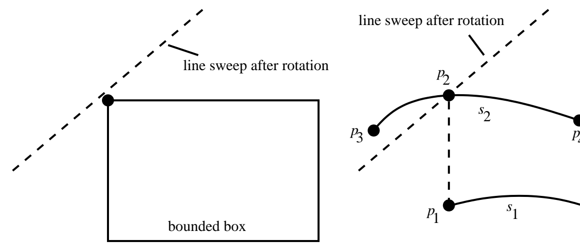

Figure 4.5: Picking a place holder at an event point

If upper right boundary corner is the left-most corner after rotation, still $p_2.x' < p_1.x'$. If the corner chosen is the lower left or lower right corner of the bounded box, then the place holder chosen is connected to one of the end points of the segment below. A similar argument can be given for the segment below.

We add a place holder from this current event point to the end point chosen, and we update the minimal information at both event points. This is done for all the event points that have no incoming segment after rotation. There is a case where this left-most point is actually to the right of the current event point after rotation. This inconsistency is discussed in the following section, Section 4.2. For now, let us ignore this case.

At most there is one new segment (place holder) added per event point, but there may be more than one place holder at each event point. These other place holders come from later event points that chose this event point as left-most. Each place holder added is only a constant factor per event point, so we do not hinder the algorithm's performance.

With the preprocess steps complete, we rotate the set, and resweep to restore the minimal information to the rotated set. After rotation each event point only knows the circular ordering of the incident segments, and every event point (except the left-most) is guaranteed to have at least one incoming segment.

Our sweep starts at the left-most event point that belongs to the bounded box. We can uniquely determine the rotated vertical order of its outgoing segments. We use a doubly linked list D to store the vertical order of segments currently in the

sweep, and the initial list is the list of outgoing segments from this left-most event point.

At each event point after the initial, we know the incoming segments, and this is a sublist of D. By walking forward and back in D, we can find the first (lowest) and last (highest) incoming segment. The segment in the sweep list below the first is the segment below this event point. The segment in the sweep list above the last is the segment above this event point. With our lowest incoming segment, we can use the circular ordering at this event point to figure out the order of lowest to highest outgoing segments. We remove the incoming segments from D, and insert the outgoing segments in the same place, and continue the sweep for all event points to restore the minimal information.

When the resweep is finished, we will have minimal information of the transformed set with place holders. However, we want to have minimal information of the transformed set without the artificial edges. This can be done as follows. After the resweep, we can establish a partial order of the transformed segments with the place holders, similarly to how it was done in Section 3.2. We can then establish a total order of the transformed segments with the place holders that is consistent with the partial order. Removing the place holders leaves a total order of just transformed segments. With the total order of just transformed segments, a final resweep will give us the minimal information with only the transformed set.

4.2 Inconsistencies and Degeneracies in Rotation of the Arrangement

These are the inconsistent and degenerate cases that arise in our rotation algorithm using the floating point method. Here is how the algorithm handles them.

Inconsistent Place Holder An inconsistency can occur where the end point chosen is not incoming before the current event point. We deal with place holder inconsistencies by rounding the two points. This is an artifact of floating point rounding.

If exact methods were used, this inconsistency does not occur. Figure 4.6 illustrates how we round the two points.

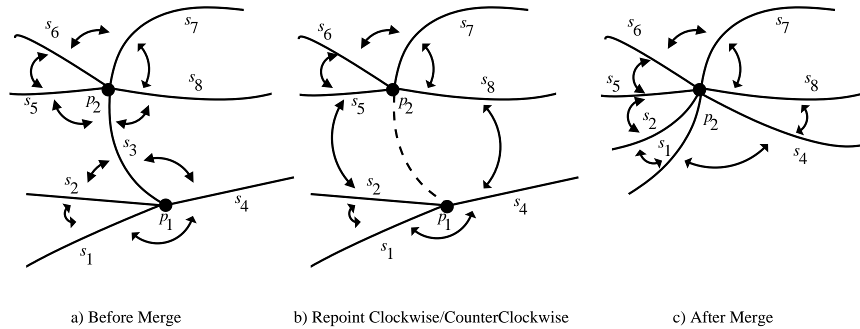

Figure 4.6: Merging circular lists of two event points

Each event point has at least one incident segment, and each event point knows its circular ordering. Without loss of generality, lets say the event point chosen to merge with is the left event point to the segment above, p_2. It is possible there are already common segments, s_3, between these two event points, p_1 and p_2. If there are any common segments, discard them, for they turn into segments of length zero. To combine circular lists, merge the lowest incoming segment of p_2, s_5 to the highest incoming segment of p_1, s_2. Merge the lowest outgoing segment of p_2, s_8, to the highest outgoing segment of p_1, s_4.

If there are no incoming segments at an event point, combine with the next outgoing segment thats clockwise or counterclockwise, depending if its the upper event point or the lower event point. See Figure 4.7 for clarification. Event point p_2 has no incoming segments, so the next clockwise to s_2 of p_1 is s_5 of p_2. Next clockwise to s_4 of p_2 is s_3. If there are no outgoing at an event point, combine with the next incoming segment at that event point accordingly.

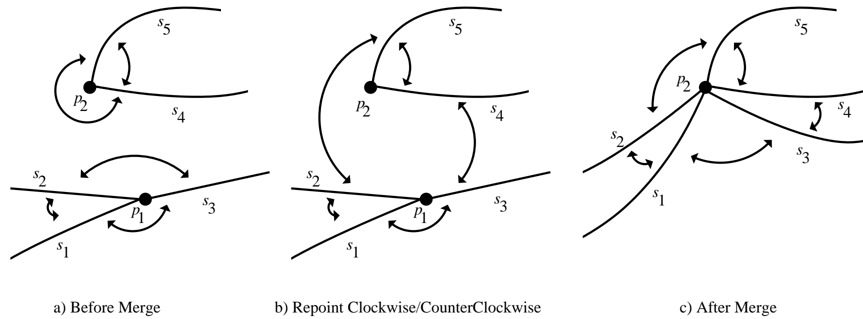

Figure 4.7: Merging when event point has no incoming segments

With exact methods, this inconsistent case would not happen. With floating point methods, we are forced to round the two different points. We categorize this rounding as an unavoidable Euler operation of VFV or VEV depending on if there existed a segment connecting both event points.

Inconsistent Turning Point When rotating, turning points are generated, but because of floating point arithmetic the inverse of these turning points end up being outside the range of the end points of the segment, see Figure 4.8.

If the inverse of the turning point is outside the range of our head(f) and tail(f), that means the turning point was very close in distance to either the rotated head(f) or rotated tail(f). Because of floating point arithmetic, the inverse of the turning point fell outside of the original head(f)/tail(f) range. After rotation, the segment is broken into two separate segments, call them s_1 and s_2. Let s_1 be the small segment.

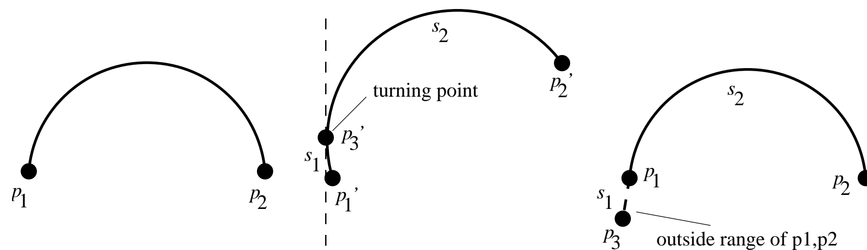

Figure 4.8: Invalid turning point

The algorithm disregards the turning point as a new event point, and reuses the original rotated end points for the rotated segment. Segment s_2 changes its turning

point end point into the other end point of s_1. Segment s_2 maintains its function f, and has end points p'_1 and p'_2.

We disregard these turning points because they were inconsistently created by floating point rounding. We can consider the removal of the turning point after a rotation a −EVE Euler operation. This case would not occur with exact methods.

Event points overlapping in x If event points overlap in x after rotation, we round them together. By keeping a counter and a marker for all event points that have equal x, it is possible to walk up and down the doubly linked list D to recover the order of incoming segments after the transformation. The incoming segments will then round to one event point. The outgoing segments, inferred from the rest of the incident segments at the overlapping x event points, are incident to this new event point as well. The procedure is as follows.

We have a sorted order of all the event points in the priority queue Q, sorted by their x value, and ties are broken by a unique identification given to each event point. When at an event point, we check for all event points following it for event points with equal x values. We group them all together, and run the following rounding algorithm.

Take the first event point, and find an incoming segment in the doubly linked list. This segment has to occur, because the algorithm states that every event point must have at least one incoming segment. With this segment, the location is known in the doubly linked list D. Now walk up and down the doubly linked list D, and record each segment that is incident to the group of event points we are currently looking at, the ones which have equal x-coordinate. As each new segment above and below is seen, mark the event point to see if it was reached before. If it was reached before, the mark at that event point was set to true already, so ignore it. If the mark is false, mark it to true, and increase the counter. When the counter is at the number of event points currently with equal x, then the next segment reached that does not have an incident segment to one of the following event points is the segment above the newly rounded event point. This segment above is not incident at this rounded event point.

Do the same procedure for both directions, up and down the doubly linked list. Evenly go up one segment, and down one segment, in order for efficiency in the algorithm, and correctness of the marking. If the algorithm only moved down till the last incident segment below and then moved up, then we are unsure how far down to move since we do not know if we are finished.

We mark the event points to handle another degenerate case: pinched segments. A pinch segment is found when the algorithm walks up/down in D, and finds a segment who does not belong to any of the equal x event points and not all rounding event points are marked yet.

Once all the segments are found, keep record of the highest segment in the doubly linked list for this event point, and keep a record of the lowest segment in the list for this event point. All other segments within these two segments are considered pinched and incident to this new event point. Originally pinching required checking for the highest incoming or outgoing segment, and lowest such segment, and then find all pinched segments by looking at D. But this algorithm needs not do this, because no other segments can be pinched if not between the highest and lowest segment for this event point.

The pinched segment firstly cannot go through the incident segments of an event point; that would mean the original algorithm failed to catch this pinch. Secondly, all event points must have an incoming segment in order for us to correctly place the outgoing segments in the doubly linked list D. Thus, if the pinched segment was between an incoming segment and an outgoing segment, then the outgoing segment was incident to one of these same x event points that also had at least one incoming segment. It is this incoming segment which also sandwiches the pinched segment with the other incoming segment we identified earlier, see Figure 4.9.

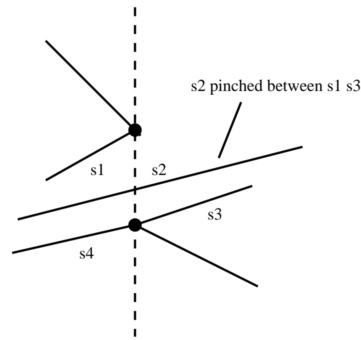

Figure 4.9: Pinching after a rotation

Merging of event points cause inevitable changes in the topology. In addition, the rotation function is not guaranteed to be one-to-one in floating point: two points might rotate to the same point. These cause a few inevitable topology changes as well. These occurrences are very few, and they are reported to the user.

We round the points because of our assumption of generic orientation, and this makes the algorithms robust. We categorize rounding of this degenerate case as unavoidable Euler operation of VFV or VEV depending on if there existed vertical line segments connecting the overlapping x event points. Pinching is an artifact of rounding overlapping x event points. We categorize pinching as a combination of EVE and VFV Euler operations.

4.3 Correctness

In this section, we discuss the correctness of the algorithm:

Property 1: Other than the boundary event points, there is at least one incoming segment at each event point. Place holders guarantee we have one, if we were using exact methods. Since we are using floating point methods, rounding can occur, and give us an inconsistent place holder. By rounding the event points, the point with no incoming segment does not exist anymore.

Property 2: Place holders associated with the same cluster segment do no cross each other. Place holders cluster around segments incident to an event point called

cluster segments; a place holder that came from another event point later on linked to this event point. Each cluster segment can have any number of place holders associated with it. If place holders are above the cluster segment, the clockwise ordering of the place holders is ordered by lowest x-values first. If the place holders are below this cluster segment, the clockwise ordering is ordered by the highest x-value first, see Figure 4.10. Ordering in this fashion allows the place holders associated with a cluster segment to not cross each other.

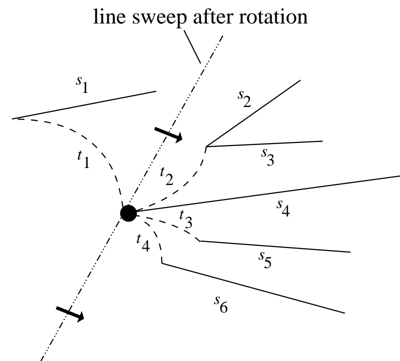

Figure 4.10: Example of place holders at an event point

Property 3: Clusters are disjoint. Clusters are the cluster segment and the place holders associated with that cluster segment. If one cluster segment is above another cluster segment, then the cluster of place holders of the former are above the cluster of place holders of the latter. Since we are placing place holders either next to or after the last place holder of that cluster segment, we avoid mixing clusters together. Figure 4.11 shows an example of two disjoint clusters: A and B.

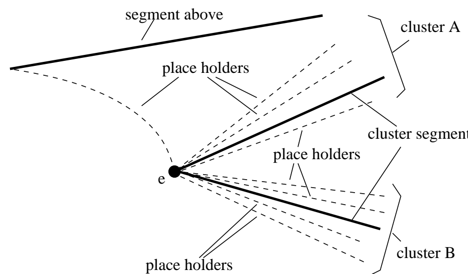

Figure 4.11: Two clusters A and B, at an event point

Property 4: If the arrangement before place holders is planar, then we still maintain planarity if place holders are added. Since clusters are disjoint, we do not introduce new intersections with place holders at each event point. In addition, by creating place holders with the smallest x-valued end point, place holders at different event points do not cross each other.

Property 5: Partial ordering is equivalent to the minimal information. The minimal information implies a vertical ordering of the segments as long as ordering is acyclic in partial ordering, which we have since we have a planar arrangement. Partial ordering also implies the minimal information, for we are able to establish a total order on all the segments to generate the minimal information through a resweep.

We do not consider vertical partial ordering as the minimal information, because to reconstruct the arrangement with just partial ordering as the minimal information will take $O(n \log n)$, while the minimal information that we have now takes $O(n)$ to reconstruct.

Property 6: The combinatorial structure is invariant in a transformation. In our topologically invariant transformation algorithm, we maintain the combinatorial structure by maintaining the circular order of incident segments at each event point. We introduce no new intersections in our algorithm, so there will be no new vertices and segment splits except at turning points for x-monotonic segments. These splits at turning points aren't true changes to the combinatorial structure if we, for example, consider partitioned x-monotonic segments are part of same "virtual" segment, and maintain combinatorial structure of one-to-one mapping of "virtual" segments.

In our transformation algorithm, we add place holders to the input in order to recover the minimal information after a transformation. Each temporary place holder connects two existing vertices, which may split an existing cell, causing a change in the combinatorial structure; however, these temporary place holders are removed after the minimal information is recovered by vertical partial ordering after the transformation, as explained in Section 4.1. The result is the same combinatorial structure as before transformation.

Property 7: We can uniquely determine the minimal information after a transformation. Because of Property 6, we can uniquely determine the minimal information after a transformation with place holders. Because of Property 5, we can then uniquely determine the minimal information after a transformation without place holders.

Property 8: Our transformation algorithm is invariant with exact methods. As shown in Properties 1-7, we maintain planarity and combinatorial structure during a transformation. If we used exact methods to transform segments, we do not get the inconsistencies from Section 4.2 (e.g. we can always guarantee an incoming segment). Also, since we assume input is generically oriented, in the exact case two vertices or segment intersections having the same x should also have the same y, which eliminates the rounding issues. However, if we used floating point methods, we must justify robust topological invariance when we perform rounding, and we did this in Section 4.2.

To recap, in order to maintain same topology, we use helper segments and the circular order of segments at a event point so that we can uniquely determine the minimal information from untransformed to transformed segments with helper segments. First we establish partial order of the transformed segments with helper segments. We can then establish a total order of the transformed segments with helper segments that is consistent with the partial order. We can then remove the helper segments, leaving a total order of just transformed segments. This ordering allows us to uniquely determine the minimal information without helper segments. Thus, the topology is invariant, for the original segments corresponds to the transformed segments uniquely.

Chapter 5

Overlay

This chapter describes our topologically invariant overlay algorithm.

5.1 Algorithm

We devised a way to overlay multiple arrangements topologically invariantly by prohibiting self intersections and reusing the original planar arrangements. The algorithm can be generalized to n arrangements, but we describe the algorithm using only two arrangements for understanding. Our overlay algorithm between two arrangements is as follows:

- Input: Two arrangements: A and B, where A is an arrangement of red segments, and B is an arrangement of blue segments.
- Output: A new arrangement generated from red and blue segments from A and B.

We use the line sweep algorithm from chapter 2 for the input set of segments (red segments from A and blue segments from B). Insert endpoints of A and B segments into an event point priority queue Q, ordered by their x-value. T is a balanced binary tree of segments, and the in order traversal of T gives the vertical order of segments in the line sweep.

When at an insert event, this insert event's segment is inserted into T. We know this segment's original arrangement vertical order at this point in its own line sweep. This segment has to be below and above a certain segment in that original arrangement. We know a below and above segment exists because we have $\pm\infty$ boundary segments. For simplicity, let the boundary segments used in arrangement A be the same boundary segments used in arrangement B.

In Figure 5.1, if a_2 is between a_3 and a_1 in the original arrangement, then in the combined arrangement this ordering shall be preserved, that is $a_1 < a_2 < a_3$. Thus, the insertion of a A/B segment into T has to be within a certain range, and the segment above and below it in their respective original arrangement defines this range. This range, between the above and below segment, shall consist of only segments from the other arrangements, if any. Any insertion outside this range is prohibited. The segment above and below at this event point exists in T because they existed in the original arrangement's T at this event point.

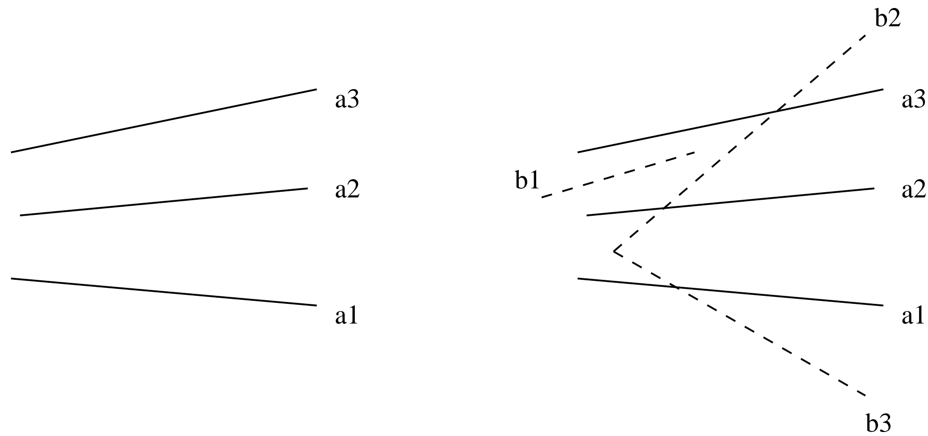

Figure 5.1: Red segment order is preserved

Once inserted, check neighboring segments for intersections. The intersections found are new event points inserted into Q, and are the swap events. Check intersections with neighboring segments if and only if that neighboring segment belongs to the other arrangements. There shall be no new intersection between segments belonging to the same arrangement, for that intersection would have been detected in

that segment's original arrangement. If an intersection test is given, however, there might be intersections that occur, due to floating point inaccuracies.

When at a remove event point, this event's segment is removed from T. When a segment is removed from T, the segments, above and below are now adjacent in the line sweep and are checked for either an immediate intersection or a future intersection. Checking for intersections only if the two segments belong to different arrangements.

At swap events, a swap pair is defined. This swap pair is generated when the intersection was detected from a pair of adjacent segments. This pair is then removed and inserted back into T, with its order reversed. Now, two more intersection tests are performed between the swap pair and the lines above and below. Again, that intersection must be between segments of different arrangements. Continue swapping for all swap pairs at the event point that are adjacent. The line sweep output is then converted to an arrangement with two winding numbers at each cell. An overlay of n arragenements results in n winding numbers at each cell. Generating of winding numbers was explained in Chapter 3. After the overlay algorithm is complete, the set operation on the sets is done in the same fashion as explained in Section 3.5.3.

5.2 Inconsistencies

If there is a swap pair that is not adjacent, discard it, similarly to the method described in Section 2.2. At the end of the line sweep, the segments are broken up at these swap event points.

The line sweep must generated a planar arrangement, so we must take care of pinched segments in overlays. The method is illustrated in Section 2.2: first find $infimum$ and $supremum$, then the segments strictly between these two segments in the tree T of segments is pinched.

This is adapted to the combination of two arrangements as follows. Let the first arrangement be red segments with red event points, and the second arrangement be blue segments with blue event points. When at a red event point (an event point

made entirely from red segments), find the lowest/highest incident red segments, and now find all blue segments within this range inside T. We are guaranteed to only have blue segments in this range because if a red segment existed that is not incident, then it should have been taken care of in the original arrangement. These blue segments are "pinched" and split at this event point. The same method is applied to a blue event point (an event point made entirely from blue segments). An event point that is created from blue and red segments provides no problems for pinching, for splitting the pinched segments between the highest and lowest does not change the vertical order of those segments. Thus, the vertical ordering within each original arrangement has been preserved.

For n arrangements, it is the same method. For each event point made with only incident segments from the same original arrangement, find the highest and lowest incident segment. All segments between these two are "pinched" at this event point. No other segments from this event point's original arrangement are pinched at this event point, because those should have been pinched previously in their original arrangement. If the event point is made from segments from multiple arrangements, treat pinching the same as the two arrangement case; split all non incident segments within the range of highest and lowest incident segment.

Chapter 6

Testing Results

This chapter shows testing results for both Milenkovic and Sacks' algorithm versus our algorithm. To test the algorithms, we used an Intel(R) Pentium(R) 4 3.06GHz Machine with 512 MB of RAM. We randomly chose nine points on the plane and created a semi-algebraic curve that connected these nine points, see Figure 6.1(a). We copied and rotated this curve three times by 90,180,270 degrees in a way that it would connect at its end points to create a box-like shape, see Figure 6.1(b).

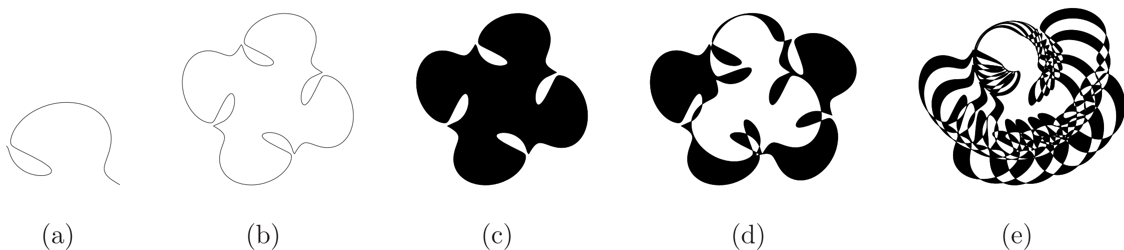

Figure 6.1: (a) Randomly generated curve, (b) Set generated from the random curve, (c) Arrangement a_0, (d) Arrangement a_1, (e) Arrangement a_3

For the input segments, we generated an initial arrangement, a_0, see Figure 6.1(c). Each subsequent set, a_{i+1} is generated by the symmetric difference of a_i with the rotated set of a_i, r_i. The angle of rotation for r_i is $47/2^i$ degrees. To test robustness of the algorithm, we generated degenerate inputs, d_{i+1}, by taking the symmetric differences of the a_{i+1} with the perturbed versions of themselves, b_{i+1}, obtained by

rotating a_{i+1} by 7 degrees then rotating by -7 degrees. The b_{i+1} are identical to the a_{i+1}, except for rounding error, so their joint arrangement is degenerate everywhere.

Arrangement a_1 is displayed Figure 6.1(d). Arrangement a_3 is displayed Figure 6.1(e) to show a more complicated example.

If exact methods were used, then an exact copy of the original arrangement a_{i+1} is generated with b_{i+1}, but since we are using floating point arithmetic, b_{i+1} is a perturbed version of a_{i+1}. We tested the inputs using Milenkovic and Sacks' algorithm and our algorithm. We ran the test for $0 < i < 6$.

During each arrangement generation (a_i, r_i, b_i, d_i), we recorded the following information:

- Segs - Number of segments
- EP - Number of event points
- Cell - Number of faces
- ARR - Running time of the Arrangement Algorithm. This is the total running time of the arrangement algorithm minus the root call time and transform call time. All times are measured in terms of the number of clock ticks used in the CPU.
- ROOT - Running time of the root calls. Given 2 functions, a root is an intersection between the two functions. All times are measured in terms of the number of clock ticks used in the CPU.
- TRNS - Running time of the transform calls. The time it takes to transform the segments' function and their two end points is recorded. All times are measured in terms of the number of clock ticks used in the CPU.
- Error 1 - Max y-distance error at an event point with all incident segments, see Figure 6.2. Check using the segment's function f.

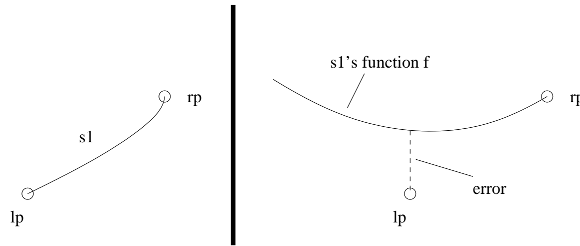

Figure 6.2: The y-distance error between event point with incident segments

- Error 2 - Max y-distance error at an event point with the segment above, see Figure 6.3. This is only error if the segment above is really below this event point. Check using the segment's function f.

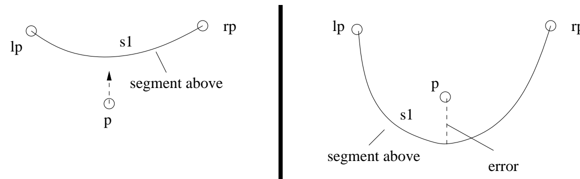

Figure 6.3: The y-distance error between event point with the segment above

- Error 3 - Max y-distance error at an event point with the segment below. This is only error if the segment below is really above this event point. Check using the segment's function f.

- Error 4 - Max intersection y-distance error. When a root was found between two segments, a swap event was created. At each event point, if there was a swap event, calculate the max y-distance error between this event point with both swapping segments. Check using the segments' function f. Similar to Error 1, but only recording error for event points that have swap events.

- Error 5 - Max Closest Distance Error from an event point to an incident segment. Check using the segment's function f, see Figure 6.4.

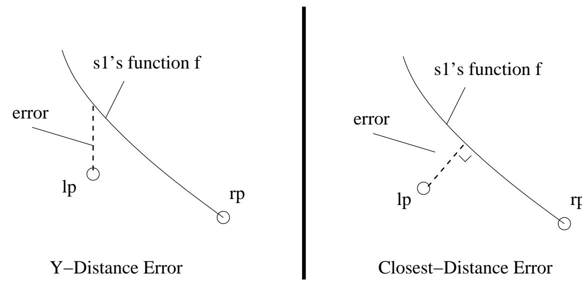

Figure 6.4: Closest distance error from an event point to a segment

- Error 6 - Max Closest Distance Error from an event point to the two incident swap segments. Check using the segments' function f. Similar to Error 6, but only recording error for event points that have swap events.

The testing results is as follows. We looked at the number of segments, event points, and cells created, and running times at each iteration. Table 6.1 are results for the non-topologically invariant algorithm from Milenkovic and Sacks. Table 6.2 are results for our topologically invariant algorithm.

For our topologically invariant algorithm, rotated arrangements took less time to generate, when compared to Milenkovic and Sacks' algorithm. Our algorithm took less time in each rotated iteration because no root calculations are made after the rotation of the segments. Using the place holder segments, we were able to rebuild and maintain the topology of the arrangement after rotation. Milenkovic and Sacks' algorithm allow new intersections to form, forcing the topology to change.

With ever rotation, the amount of segments are doubled in Milenkovic and Sacks' algorithm. These segments are the results of the new intersections. Another result of these new intersections is that the number of cells also doubles after a rotation in their algorithm. Our algorithm keeps the same number of cells before and after a rotation. In some instances, the number of cells does change in our algorithm, which indicates a change in the topology. This is because the rotation function is not guaranteed to be one-to-one, causing inevitable topology changes. These cases are rare and happened only a few instances during our experiments. All cases of topology changes in our algorithm are reported to the user.

For error analysis, the later iterations in testing produced more noticeable results than the earlier iterations. We focus more on those results. Table 6.3 are results for Error 1-3 for the non-topologically invariant algorithm from Milenkovic and Sacks. Table 6.4 are results for Error 1-3 for our topologically invariant algorithm. Table 6.5 are results for Error 4-6 for the non-topologically invariant algorithm from Milenkovic and Sacks. Table 6.6 are results for Error 4-6 for our topologically invariant algorithm.

From these error results, we show that our topologically invariant algorithm does not introduce a significant amount of new error. Our algorithm does not introduce any error for Error 4 and Error 6 during rotations because those error calculations are based on intersections. We do not allow for intersection tests after a rotation in our algorithm, so we get no intersection error.

Arr.	Segs	EP	Cells	ARR	ROOT	TRNS
a_0	21	19	4	0	32	0
r_0	32	26	8	0	31	3
a_1	65	51	16	1	8	0
r_1	91	62	31	0	21	7
b_1	93	65	30	1	29	6
d_1	200	117	85	2	20	0
a_2	239	167	75	3	26	0
r_2	407	251	159	5	52	18
d_2	750	418	338	8	67	0
a_3	979	654	333	15	99	0
r_3	1713	1009	712	28	188	58
d_3	3252	1717	1545	34	276	0
a_4	3876	2568	1332	52	406	0
r_4	6931	4078	2877	96	757	208
d_4	13086	6857	6260	177	1021	0
a_5	15430	10212	5257	240	1614	0
r_5	27718	16312	11443	458	3049	766
d_5	52728	27606	25174	878	3993	0
a_6	61664	40638	21097	1391	5939	0
r_6	114071	65900	48228	2368	8914	2208
d_6	211131	110671	100562	7349	16092	0

Table 6.1: Milenkovic and Sacks' algorithm

Arr.	Segs	EP	Cells	ARR	ROOT	TRNS
a_0	21	19	4	1	37	0
r_0	26	24	4	1	0	3
a_1	71	55	18	0	43	0
r_1	66	51	17	1	0	7
b_1	71	56	17	2	0	8
d_1	219	113	108	6	56	0
a_2	253	164	91	3	37	0
r_2	254	166	90	3	0	21
d_2	877	376	503	13	77	0
a_3	1003	578	427	20	121	0
r_3	1003	579	426	18	0	61
d_3	3743	1435	2310	63	253	0
a_4	3921	2114	1809	68	363	0
r_4	3921	2115	1808	71	0	206
d_4	14807	5356	9453	276	923	0
a_5	15521	8068	7455	322	1270	0
r_5	15528	8075	7455	363	0	727
d_5	59630	21037	38595	1589	3543	0
a_6	61819	31527	30294	1730	4951	0
r_6	61821	31530	30293	3214	0	2032
d_6	239234	83291	155945	11428	14295	0

Table 6.2: Test results for our algorithm

Arr.	Error 1	Error 2	Error 3
a_4	8.09233e-10	1.04761e-11	7.54952e-15
r_4	1.03678e-10	1.4686e-12	4.77396e-15
b_4	9.62654e-11	1.3658e-12	1.24345e-14
d_4	2.66887e-07	1.07854e-10	5.42477e-12
a_5	8.09233e-10	1.04761e-11	6.61693e-14
r_5	1.30991e-10	2.75335e-12	2.0608e-12
b_5	1.0774e-10	1.3658e-12	2.4114e-13
d_5	2.66887e-07	1.07854e-10	4.08982e-11
a_6	8.09233e-10	1.04761e-11	7.42961e-13
r_6	8.59739e-09	3.10596e-12	1.28692e-12
d_6	2.87427e-07	1.07854e-10	8.8711e-11

Table 6.3: Error 1-3 results for Milenkovic and Sacks' algorithm

Arr.	Error 1	Error 2	Error 3
a_4	8.09233e-10	2.53131e-13	3.86358e-14
r_4	1.03678e-10	0	1.28314e-13
b_4	9.62654e-11	0	4.2244e-14
d_4	2.66887e-07	2.60871e-11	3.42006e-11
a_5	8.09233e-10	2.53131e-13	1.28314e-13
r_5	1.12016e-10	6.14397e-13	6.2006e-14
b_5	1.0774e-10	1.24345e-14	4.2244e-14
d_5	2.66887e-07	1.7841e-10	9.1599e-11
a_6	8.09233e-10	6.14397e-13	1.28314e-13
r_6	8.59739e-09	1.07669e-12	4.2244e-14
d_6	2.87427e-07	1.7841e-10	9.1599e-11

Table 6.4: Error 1-3 results for our algorithm

Arr.	Error 4	Error 5	Error 6
a_4	8.09233e-10	7.98458e-10	7.98458e-10
r_4	1.63698e-11	9.72696e-11	1.60443e-11
b_4	1.69618e-11	8.70178e-11	1.62738e-11
d_4	7.74517e-10	7.64205e-10	7.64205e-10
a_5	5.73992e-11	9.72696e-11	3.52157e-11
r_5	2.28741e-11	9.4838e-11	5.64424e-12
b_5	3.75584e-11	9.57827e-11	1.62738e-11
d_5	7.74517e-10	7.64205e-10	7.64205e-10
a_6	4.71029e-10	3.26112e-10	3.26112e-10
r_6	7.79218e-10	8.56691e-09	7.72952e-10
d_6	5.6731e-10	8.56689e-09	2.59451e-11

Table 6.5: Error 4-6 results for Milenkovic and Sacks' algorithm

Arr.	Error 4	Error 5	Error 6
a_4	8.09233e-10	7.98458e-10	7.98458e-10
r_4	0	9.72696e-11	0
b_4	0	8.70178e-11	0
d_4	7.74517e-10	7.64205e-10	7.64205e-10
a_5	5.73992e-11	9.72696e-11	3.52157e-11
r_5	0	9.4838e-11	0
b_5	0	9.57827e-11	0
d_5	7.74517e-10	7.64205e-10	7.64205e-10
a_6	4.71029e-10	3.26112e-10	3.26112e-10
r_6	0	8.56691e-09	0
d_6	7.74517e-10	8.56689e-09	7.64205e-10

Table 6.6: Error 4-6 results for our algorithm

Chapter 7

Conclusion

We validated the algorithms on random, structured, and degenerate inputs. Each input was rotated by a random angle to implement a random sweep direction. We were able to come up with results that kept the topology invariant. There were cases that were discovered that caused inevitable topology changes, since rotation uses floating point arithmetic, and rotation is not a one to one process. These were rare instances that could not be avoided.

Our algorithm has the same running time as Milenkovic and Sacks' algorithm. On rotation, it is faster than Milenkovic and Sacks' algorithm, for it does no new root calculations, so the constant factor is lower. Empirical evidence showed that root calculation is an expensive process and takes more time than the arrangement algorithm process.

References

[1] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In R. H. Möhring and R. Raman, editors, ESA, volume 2461 of Lecture Notes in Computer Science, pages 174–186. Springer, 2002.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. McGraw-Hill Book Company, 2000.

[3] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact, and efficient computations with cubic curves. In J. Snoeyink and J.-D. Boissonnat, editors, Symposium on Computational Geometry, pages 409–418. ACM, 2004.

[4] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in cgal. ACM Journal of Experimental Algorithms, 5:13, 2000.

[5] S. Fortune. Robustness issues in geometric algorithms. Applied computational geometry: towards geometric engineering, pages 9–14, 1996.

[6] N. Geismann, M. Hemmer, and E. Schömer. The convex hull of ellipsoids. In Symposium on Computational Geometry, pages 321–322, 2001.

[7] D. Halperin and E. Leiserowitz. Controlled perturbation for arrangements of circles. In Symposium on Computational Geometry, pages 264–273. ACM, 2003.

[8] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha. Esolid - a system for exact boundary evaluation. Computer-Aided Design, 36(2):175–193, 2004.

[9] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. Efficient and exact manipulation of algebraic points and curves. Computer-Aided Design, 32(11):649–662, 2000.

[10] K. Mehlhorn and S. Näher. LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press, 1999.

[11] V. J. Milenkovic and E. Sacks. An approximate arrangement algorithm for semi-algebraic curves. Technical Report CSD-TR 04-017, Purdue University, June 2004.

[12] B. Mourrain, J.-P. Técourt, and M. Teillaud. Sweeping an arrangement of quadrics in 3d. In Proc. 19th European Workshop on Computational Geometry, pages 31–34, 2003.

[13] A. A. G. Requicha. Geoemtric modeling: A first course 3. http://www-lmr.usc.edu/~requicha/ch3.pdf, 1995-1999.

[14] E. Schömer and N. Wolpert. An exact and efficient approach for computing a cell in an arrangement of quadrics. Accepted for CGTA: Special Issue on Robust Geometric Algorithms and their Implementations.

[15] R. Wein. High-level filtering for arrangements of conic arcs. In Proc. ESA 2002, pages 884–895. Springer-Verlag, 2002.

[16] N. Wolpert. Jacobi curves: Computing the exact topology of arrangements of non-singular algebraic curves. In G. D. Battista and U. Zwick, editors, ESA, volume 2832 of Lecture Notes in Computer Science, pages 532–543. Springer, 2003.

[17] C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, Handbook of discrete and computational geometry. CRC Press, second edition, 2004.