**Intersecting Red and Blue Line Segments in Optimal
Time and Precision**

by

Andrea Mantler

B.C.Sc.(Hons.), University of Manitoba, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

November 2001

© Andrea Mantler, 2001

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date August 22, 2001

Abstract

In this thesis we describe a new algorithm, SPAGHETTISWEEP, for solving the red-blue line segment intersection problem. We use a symmetric lazy sweep and at most degree two predicates. Our method is numerically robust, and handles all degeneracies in the input data. Our algorithm finds all intersecting pairs of segments in $O(n \log n + k)$ time and $O(n)$ space, where n is the total number of segments, and k is the number of intersection points. With a small modification, we are able to count intersections in $O(n \log n)$ time and $O(n)$ space. Using only degree two predicates, we are also able to compute the arrangement in $O(n \log n + k)$ time.

Contents

Abstract	ii
Contents	iii
List of Tables	vi
List of Figures	vii
Acknowledgements	viii
1 Introduction	1
1.1 Problem Descriptions	1
1.2 Degeneracies	3
1.2.1 Non-distinct x -coordinates	4
1.2.2 Zero-length Segments	4
1.2.3 Shared Endpoint	5
1.2.4 Point on a Line Segment	5
1.2.5 Overlapping Segments	6
1.3 Thesis Overview	6
2 Previous Work	7
2.1 Summary of Previous Work	7
2.1.1 General Segment Intersection	7
2.1.2 Red-Blue Segment Intersection	8
2.1.3 Robustness Issues in Segment Intersection	9
2.2 Bounded Degree Calculations	11
2.2.1 Reasons For Using Low Degree Predicates	12
2.2.2 Bounded Degree Predicates	12
2.2.3 Computing the Arrangement Using Low Degree Predicates	14

3	Pseudo-Segments, Events, and Witnesses	15
3.1	Pseudo-Segments	15
3.2	Events	16
3.3	Witnesses	17
3.4	Delaying Intersections	18
4	Algorithm Description	20
4.1	Input and Output	20
4.2	Invariants	20
4.2.1	Regular Sweep	20
4.2.2	Red-Blue Pseudo-Segment Sweep	21
4.3	Data Structures	22
4.3.1	Supporting Lists	22
4.3.2	Sweep Line Data Structures	22
4.3.3	Asymmetry	23
4.3.4	Underlying Tree Structure	23
4.4	Processing Events	25
4.4.1	Main Event Loop	25
4.4.2	Swapping Bundles	28
4.4.3	Recording Intersections	30
4.4.4	Ending and Starting Bundles	32
4.5	Proof of Correctness	33
4.5.1	While Loops	33
4.5.2	Algorithm Correctness	34
4.5.3	Partial Ordering of Intersections	36
4.6	Algorithm Analysis	37
4.6.1	Time	37
4.6.2	Space	39
5	Comparisons	40
5.1	Types of Comparisons	40
5.1.1	Initial Sort	40
5.1.2	Ends After	41
5.1.3	Above, On or Below	41
6	Extensions	45
6.1	Counting Intersections	45
6.1.1	Number of Segments in a Bundle	45
6.1.2	Analysis	47

6.1.3	Recording Without Reporting	48
6.2	Finding the Arrangement	50
6.3	Intersections with More Than Two Colours	52
6.4	Future Work	53
6.4.1	Finding the Arrangement	54
6.4.2	Underlying Tree Structure	54
6.4.3	Library Implementation	54
6.4.4	Compare Running Times	55

Bibliography		56
---------------------	--	-----------

List of Tables

2.1	Degree of predicates for line segments [6].	13
3.1	When an event witnesses an intersection. The symbols +, = and – indicate if the segment is above, equal to, and below the event respectively.	18
4.1	Allowed adjacencies when the <i>above</i> , <i>equal</i> , and <i>below</i> sections are disjoint. The symbols +, = and – stand for <i>above</i> , <i>equal</i> , and <i>below</i> bundles (or segments).	33

List of Figures

3.1	Pseudo-segments and witnesses. The two curves are pseudo-segments, and the point w is the witness for their intersection.	18
4.1	The processing of intersections is delayed.	22
4.2	Segments along the sweep line are bundled, and the bundles of one colour are stored in a bundle tree.	24
4.3	Pseudo-code for algorithm SPAGHETTISWEEP.	26
4.4	The four while loops executed by SPAGHETTISWEEP. The dashed line, where a red bundle may have been split, indicates where the blue bundles are swapped to.	29
4.5	The witness w causes the bundles A and B to swap.	30
4.6	The lexicographically earlier event will witness the lexicographically earlier intersection along the segment. In neither case is w_2 actually a witness.	36
6.1	Keeping track of bundle size during a splay operation. Each case has a symmetric variant (not shown). The operations, from top to bottom are: (a) Zig, (b) Zig-zig, and (c) Zig-zag [26].	46
6.2	Making splay trees persistent. Copy root to x path before splaying x	48
6.3	Unconnected groups of segments may nest. For example, the shaded region contains three connected components of segments.	50
6.4	Regions may split or join as the sweep line moves. We know a connected component C is contained in the region R when R splits and then later rejoins.	51

Acknowledgements

I would like to acknowledge the financial support I received from both the Natural Sciences and Engineering Research Council of Canada and the National Science Foundation (Grant No. 9988742).

I would like to thank Dr. Jack Snoeyink and Dr. David Kirkpatrick for their ideas, helpful comments, suggestions, and criticism. Their feedback has greatly improved this thesis.

I would also like to thank the computer science departments at the University of British Columbia and the University of North Carolina at Chapel Hill for being accommodating and supportive.

Finally, I would like to thank my friends, family, and the Department of Computer Science at the University of Manitoba for all the support and encouragement they have given me.

ANDREA MANTLER

*The University of British Columbia
November 2001*

Chapter 1

Introduction

1.1 Problem Descriptions

Consider the following hypothetical problems:

1. A game developer wishes to simulate the game Pick Up Sticks. After she chooses the locations of the line segment “sticks”, she needs to find the intersections so that she can set the vertical ordering of the sticks at each intersection.
2. An engineer is brought into town to inspect all of the local bridges. He is given two data sets consisting of line segments: one representing the road network, and the second representing the rivers and streams. Unfortunately, none of the bridges are marked. To determine where the bridges are likely located, he finds the intersections between road segments and river segments.
3. Two farmers are having a dispute over land. Each farmer brings a survey of the land she believe to own to the land claims office. The officer takes the two surveys and determines where the overlap is, then tries to figure out what went wrong.

Problem 1 requires a solution to what is known as the general line segment intersection problem:

General Segment Intersection Problem: Given a collection of n line segments in the plane, determine which pairs of segments intersect.

In some applications, the segments are separated into two layers, called *red* and *blue* to distinguish the layers. Problem 2 does so, and thus requires a solution to what is known as the red-blue line segment intersection problem:

Red-Blue Segment Intersection Problem: Given two collections of line segments in the plane, where

- one collection is coloured red and the other is coloured blue, and
- there are no intersections of a single colour, except possibly at an endpoint,

determine which pairs of segments intersect.

In the red-blue intersection problem, there are n segments in total.

A variant of the red-blue segment intersection problem allows intersections within the layers, but only the bichromatic (*purple*) intersections are reported. This variant is considered by Basch et al. [3], but we will not use this alternate definition.

It is also possible to define the general and red-blue intersection problems in terms of objects other than segments. For example, we may wish to determine the intersections of circular arc segments or monotonic curves. Many of the existing algorithms work for objects other than line segments, as long as the proper comparison predicates are defined and run in $O(1)$ time.

In both the general and red-blue intersection problems, the input is in the form of segments specified by their endpoints, with the coordinates of the endpoints expressed as decimal numbers that can be mapped into a finite integer range. The output usually consists of a list of the intersecting pairs, but in certain cases we may only be interested in the number of intersections:

Counting (Red-Blue) Intersections Problem: Given a collection of n line segments in the plane (or two collections of red and blue line segments), return the number of intersecting pairs.

For example, the engineer in Problem 2 may want to know the number of bridges, so that he can give a cost estimate to the town.

Using the algebraic decision-tree computation model, we can transform the Element Uniqueness Problem into a segment intersection detection problem [24], showing that the lower bound for detecting if two segments intersect is $\Omega(n \log n)$. Trivially, the lower bound for reporting k intersections is $\Omega(k)$, and thus we can see that the general and red-blue intersection problems will require $\Omega(n \log n + k)$ time. (Note that k may be as large as $O(n^2)$.)

Algorithms exist that meet the optimal time bound for both the general [1, 9] and red-blue [8, 10, 19, 23] intersection problems. When counting intersections, we do not need to report individual intersections. Algorithms [10, 23] exist that count red-blue intersections in optimal $\Theta(n \log n)$ time.

When the input data is rounded to a finite grid, as is often the case in geographic information systems, the algebraic decision-tree model may not provide a correct lower bound running time.

In Problem 3, we need to know the arrangement of segments before we can perform Boolean operations on the polygons. The arrangement of segments can be mathematically defined as follows:

Arrangement of Segments: An arrangement of segments is a decomposition of the plane into vertices, edges, and faces, where

- vertices are intersection points of segments or endpoints of segments,
- edges are maximal connected (open) portions of segments that do not contain any vertices, and
- faces are maximal connected (open) portions of the plane not meeting any vertex or edge.

Thus, to compute the arrangement of segments, we need to find all intersections and a partial ordering of the intersections. (We need to know the order of intersections along each segment.) The segments may be a general collection of segments, or they may be divided into two layers (red and blue), as in the third problem. Once we find the arrangement of red and blue segments, we can label faces as red, blue, both, or neither, for the purposes of performing Boolean operations.

Computing the arrangement of red and blue line segments in the plane has many applications in vector-based geographic information systems (GIS). In GIS, computing the arrangement is also called *map overlay*. Geographic data are separated into layers representing various features. Users frequently want to perform Boolean queries on the polygons represented in these layers, such as union, intersection or subtraction.

1.2 Degeneracies

One problem with geographic data sets is that they contain degeneracies. In the past, many intersection algorithms ignored degeneracies and assumed the segments were in *general position* [1, 4, 9, 10]:

General Position: For the purposes of segment intersection problems, a collection of segments is in general position if:

- all endpoints and intersections have distinct x -coordinates,
- no segment endpoint lies on another segment, and

- any two segments intersect in finitely many isolated points.

Note that having distinct x -coordinates also excludes vertical segments, zero-length segments, and multiple segments incident to a single endpoint. Any configuration which is not allowed by the definition of general position is a degeneracy.

Some algorithms resolve degeneracies by using perturbation techniques [19]. Handling degeneracies can significantly increase the difficulty of implementing the algorithms [23], and perturbing the data can cause problems, such as dramatically increasing the number of intersections.

Our intersection algorithm, SPAGHETTISWEEP, is designed to operate correctly on data where degeneracies are expected.

1.2.1 Non-distinct x -coordinates

We do not actually calculate the coordinates of intersections in our algorithm, and thus we are not affected when intersections have the same x -coordinates as endpoints or other intersections. We allow segments to have duplicate x -coordinates by using a lexicographic ordering.

In sorting, there are two orderings we are concerned with: the ordering of the endpoints, and the ordering of the segments. In ordering the list of endpoints (which we use as *events* in a sweep line algorithm), we use the standard lexicographical order and remove any duplicate endpoints from the list. When sorting the segments, we sort on the (bottom) leftmost endpoint, and for any segments that share this endpoint, we order the segments counter-clockwise, starting with the bottom segment.

Because we use a lexicographic ordering, we do not need to treat vertical segments as degeneracies in our algorithm, although the above/below predicate must take vertical segments into account.

1.2.2 Zero-length Segments

Depending on whether or not zero-length segments contain essential information, we can choose to either filter out or handle degenerate zero-length segments. We outline how to correctly handle zero-length segments, but the code for this special case can be easily left out.

Zero-length segments require some special treatment in our algorithm for two reasons. The first reason is that zero length segments can cause problems with some sorting algorithms unless the comparison predicate is written to handle zero-length segments. Using only a counter-clockwise determinant test, a zero-length segment, s_0 , would test as being “equal” to any segment that started at s_0 ’s location. Having

two different and unequal segments test as “equal” to the same zero-length segment can cause problems in sort algorithms, such as quicksort.

The second reason is that in our event driven sweep line algorithm, zero-length segments start and end at the same event, and so we must both “add” them to and “remove” them from our data structure before continuing to the next event.

Note that there may be many zero-length segments at the same point; since there is no interior to a zero-length segment, and multiple segments of the same colour may share an endpoint, our definition of the red-blue intersection problem allows two or more zero-length segments of the same colour to have the same location.

1.2.3 Shared Endpoint

A degenerate endpoint intersection occurs when many segments share an endpoint:

All of the pairwise intersections at the endpoint must be reported. In our algorithm, we process all of the segments at the endpoint at the same time, reporting the participating segments in counter-clockwise order.

1.2.4 Point on a Line Segment

It is possible that an endpoint of one segment will fall exactly on a second segment:

We deal with this degeneracy by changing the representation of the problem. When an endpoint of one segment falls in the interior of another segment we can break the second segment exactly at that endpoint. This does not require additional precision, because the endpoint at which we are breaking the segment is represented exactly with the given precision. By breaking the segment, we convert this type of degeneracy into the shared endpoint degeneracy.

Breaking segments increases the number of segments by at most a constant factor, since every breakpoint can be traced to an endpoint of the original data set, and an endpoint can break at most one segment of each colour.

By breaking segments we reduce the number of degenerate cases, and thus reduce the complexity of implementing the intersection algorithm.

1.2.5 Overlapping Segments

Two segments, ab and cd may intersect in a line rather than a point:

The definition of the red-blue intersection problem allows this type of degenerate intersection only if one segment is red and the other is blue.

We again reduce the number of cases to deal with by breaking off the sections of the segments that do not overlap. In the above illustration, ac and bd would be broken off. In our algorithm, we detect the remaining section (cb) as a degenerate intersection, and we use a special representation to deal with these two segment fragments efficiently.

1.3 Thesis Overview

In this thesis we describe an algorithm, SPAGHETTISWEEP, for finding the pairs of intersecting red and blue line segments. Our algorithm is able to handle all degeneracies easily, and we use low degree predicates (described in Chapter 2) so that numerical errors are not a problem. Our algorithm not only finds all pairs of intersecting segments, but also determines the order of intersections along each segment.

In Chapter 2 we survey previous work in the intersection problems and predicate degree. Chapter 3 explains the concepts of pseudo-segments, events, and witnesses. Algorithm SPAGHETTISWEEP is described and its properties are proved in Chapter 4. We explain the comparisons in detail in Chapter 5, and outline possible extensions in Chapter 6.

We have implemented a demonstration version of our algorithm as a Java applet, which can be found at <http://www.cs.unc.edu/~mantler/redblue>.

Chapter 2

Previous Work

In this chapter we outline some of the previous work done on the segment intersection problems. We start with general segment intersection, followed by red-blue segment intersection, and finish with papers that address issues of degenerate segment positions and errors in calculations.

2.1 Summary of Previous Work

2.1.1 General Segment Intersection

Starting with an algorithm by Shamos and Hoey [25] that reports if an intersection occurs in a set of n line segments, Bentley and Ottmann [4] develop the first non-trivial algorithm for reporting all k intersecting pairs of segments. They produce a simple algorithm that runs in $O((n + k) \log n)$ time, somewhat slower than the optimal bound of $\Theta(n \log n + k)$.

Bentley and Ottmann use a sweep line that passes from left to right, recording intersections as they are encountered. They use two data structures, one stores the segments in order along the sweep line, and the other maintains a priority queue, Q , of upcoming endpoints and intersections. As their algorithm processes the items from Q , they maintain the following two invariants:

Invariant 1 *The segments intersecting the sweep line are sorted according to their order along the sweep line.*

Invariant 2 *All of the intersections to the left of the sweep line have been recorded, and those to the right have not.*

Intersections are recorded when segments swap order along the sweep line. The algorithm terminates when all items from Q have been processed.

In their paper, Bentley and Ottmann also give two variants of their algorithm. The first can report the intersections in $O(n \log n + k)$ time when the segments are all either vertical or horizontal, and the second can count intersections in $O(n \log n)$ time assuming the same input conditions. Given the correct predicates, the Bentley and Ottmann algorithm also works for more general x -monotone objects.

Chazelle and Edelsbrunner [9] give an $O(n \log n + k)$ algorithm for general segment intersection, but their algorithm requires $O(n + k)$ space. They use the Bentley-Ottmann algorithm as a starting point, and achieve optimal running time after a series of modifications.

Balaban [1] improves on the results of Chazelle and Edelsbrunner. Balaban starts with a relatively simple $O(n \log^2 n + k)$ time and $O(n)$ space algorithm, then optimizes one of the steps to reduce the running time to $O(n \log n + k)$. Balaban's algorithm also works on x -monotone 2D objects when the correct predicates exist.

2.1.2 Red-Blue Segment Intersection

The red-blue segment intersection problem is somewhat easier than the general case, because algorithms can take advantage of the extra structure in the data. Mairson and Stolfi [19] were able to find an optimal $\Theta(n \log n + k)$ time and $O(n)$ space algorithm for the red-blue intersection problem before the general case was solved.

The Mairson and Stolfi algorithm is very similar to the Bentley-Ottmann algorithm, but they use two "active lists" instead of a single sweep line structure. This allows them to use a static event queue, removing the log factor from k . To ensure all intersections are found, Mairson and Stolfi use a recursive cone breaking method. Unlike the Bentley-Ottmann algorithm, their algorithm does not return all the intersections in sorted order, but the intersections are locally sorted along each segment.

The Mairson and Stolfi algorithm can also count intersections in $O(n^{1.5} \log n)$ time, or find intersecting pairs of x -monotone curve segments in $O(n \log n + k)$ time and $O(n + k)$ space. They handle degeneracies by using either real or virtual perturbations of the input segments.

Chazelle et al. [10] start with a straightforward algorithm that reports intersections in $O(n \log^2 n + k)$ time and $O(n \log n)$ space, or counts intersections in $O(n \log^2 n)$ time and $O(n \log n)$ space, using a hereditary segment tree that is described in their paper. They then reduce the time and space to $O(n \log n + k)$ and $O(n)$ for reporting intersections, and $O(n \log n)$ and $O(n)$ to count intersections.

Palazzi and Snoeyink [23] simplify the algorithm of Chazelle et al. [10]. They replace the segment tree with a linear list of slabs. Their algorithm can benefit from preprocessing, and the data access is sequential, localizing memory references.

Although Palazzi and Snoeyink handle degeneracies, their method is complex and may change the number of intersections reported, requiring them to adjust the count.

Chan [8] gives a simple $O(n \log n + k)$ time algorithm for red and blue segment intersection that requires only $O(n)$ space. The algorithm can be modified to find intersecting pairs of x -monotone curve segments, given the necessary predicates. In order to simplify presentation, the paper did not include a discussion of degenerate cases.

Chan's algorithm uses a *trapezoid sweep* process, which is a variation of the standard line sweep algorithm. A blue trapezoidation of the plane is formed from the blue segments and the vertical extensions of all endpoints to the blue segments immediately above and below. As the sweep line moves from left to right, the blue trapezoids are added to the sweep front. Intersections are reported as the sweep line encounters new blue trapezoids.

Our algorithm is based on Chan's algorithm, but we use a more symmetric approach. We do not form a blue trapezoidation of the plane, but instead, as described in Chapter 4, treat both of the colours of segments in essentially the same way. Chan's algorithm depends on treating the segments differently, whereas we introduce asymmetry for efficiency reasons. The asymmetries we introduce are not fundamental to our algorithm.

2.1.3 Robustness Issues in Segment Intersection

Segment intersection algorithms are the core of GIS overlay processing, so robustness issues cannot be ignored. Many of the intersection algorithms are sensitive to numerical errors and degenerate cases. As previously mentioned, degeneracies are frequently found in GIS data. Additionally, topological or combinatorial errors caused by rounding the results of calculations cannot be ignored. From the theoretical side, two methods are commonly used to deal with such degeneracies and errors: exact arithmetic and restricted predicates. (Section 2.2 provides more details on bounded degree predicates.) In practice, however, the algorithms are frequently implemented using ϵ comparisons and crossed fingers.

Exact Arithmetic

Bartuschka et al. [2] use LEDA [20] to implement a variation of the Bentley-Ottmann sweep algorithm that is robust and efficient. They are able to handle all position degeneracies, and use exact arithmetic with a floating point filter to handle numerical errors. Their algorithm runs in $O((n + s) \log n)$ time, where s is the size of the graph storing the arrangement.

Resorting to the Original Data

Brinkmann and Hinrichs [7] address the issue of exact line segment intersection in map overlay without assuming the data are in general position. Their predicates handle the degeneracies frequently found in GIS data. They define a set of predicates that use exact arithmetic when necessary, and maintain “parent sets” in order to represent generated spatial objects only in terms of the input data.

Restricted Predicates

Boissonnat and Preparata [5] examine the degrees of predicates used in previously published intersection algorithms, and use the degrees of these predicates to determine the degree of the problems. They define the degree of a predicate as the maximum degree of the irreducible factors (over the rationals) of the polynomials that occur in the predicate and that do not have a constant sign. The degree of a problem is the minimum degree of any algorithm that is able to solve the problem, and the degree of an algorithm is equal to the maximum degree of any predicates used by the algorithm. Predicate degree is discussed in further detail in Section 2.2.

Boissonnat and Preparata consider three problems: finding the intersecting segments, finding the arrangement, and finding the decomposition of the plane into trapezoids (draw vertical line segments extending from each segment endpoint and intersection point to the segment above and below the points, or to infinity if no such segment exists). Finding the intersections is a degree two problem, and they modify the Bentley-Ottmann algorithm to use degree three predicates instead of degree five. Computing the arrangement in the general case is degree four, and finding the trapezoid map is degree five.

Boissonnat and Snoeyink [6] consider whether restricted sets of geometric predicates can be used to find line and curve segment intersections efficiently. They show that a general intersection algorithm for monotone curves must perform at least $\Omega(n\sqrt{k})$ evaluations of bounded degree predicates. Their version of Balaban’s algorithm [1] for line segments runs in $O(n \log^2 n + k \log n)$ time, and Chan’s algorithm [8] for red and blue line and curve segments, after modification, still runs in optimal $O(n \log n + k)$ time.

Geometric Rounding of Output

When the output arrangement is calculated using additional precision, we may need to round the output before writing the data to a file. If we simply round the coordinates to the nearest value, we may introduce inconsistencies in the data. *Snap rounding* [12, 14] is a geometric method for rounding line segments.

Hobby [15] uses snap rounding and the Bentley-Ottmann algorithm to compute a rounded arrangement in $O((n + k) \log n + k')$ time, where k' is the number of intersections between segments and “tolerance squares”. In practice, k' tends to be small, but can approach $O(n(n + k))$ if the lines are nearly collinear. The paper also outlines how to deal with degenerate cases.

2.2 Bounded Degree Calculations

Normally, while doing computations by hand, we do not think about the precision necessary to represent a real number. The number of decimal places expands as necessary (or we use symbols) as we add, subtract, and multiply values. In a computer, however, real numbers are stored as floating point numbers with finite precision. When adding two floating point numbers together, we can express the result using the same amount of precision as the input numbers. If we multiply the two numbers, we need twice the precision to represent the result accurately. There are two cases when we are concerned with the precision of calculations: geometric tests and constructing new geometric objects.

If the precision required is higher than that available, we lose information. In some cases, the information lost with these bits does not affect the result in any significant manner. However, where the result of a calculation is close to zero, we may get different results for geometric sign tests. If we make incorrect decisions in our algorithm, the output may not truly reflect the input data.

Even if using high degree polynomial calculations is not a problem when a geometric algorithm runs once, we may encounter problems if the algorithm is run multiple times, using the constructed geometric objects of one run as input in the next. Each subsequent time we run an algorithm the precision necessary increases. The amount of precision needed can become exponentially high, or cause geometric tests to fail where previously we were able to perform the calculations accurately.

Two methods to reduce or eliminate the errors caused by rounding include exact arithmetic and bounded degree calculations. In a way, bounded degree calculations are a form of exact arithmetic. We choose the allowed precision at compile time, and restrict ourselves to input and calculations that can be computed exactly within the specified precision.

There are several approaches to achieving exact geometric computation [17, 20], many of which are based on multiple precision and precision sensitivity. The multiple precision method uses abstract data types to keep track of the digits as the precision expands. The problem with exact arithmetic is that the calculations are usually significantly slower than regular floating point operations, and the abstract

data types use more memory than hardware supported types. One way to reduce the cost of using exact arithmetic is to use filters on the calculations, and only use exact arithmetic when necessary.

Bounded degree calculations restrict the number of terms that are multiplied together. If we know the precision of the input data and the maximum number of terms multiplied together, we can determine the maximum precision needed to store the results accurately. If the precision necessary is less than that provided by a built in data type, we can safely use that data type.

2.2.1 Reasons For Using Low Degree Predicates

We have chosen to implement our algorithm using bounded, low degree predicates. With the method outlined in Chapter 4, we can get sufficient information using at most degree 2 predicates to determine both the intersecting pairs of segments, and the arrangement of segments.

By restricting ourselves to low degree predicates, we increase the allowable input precision, without incurring the overhead of extended precision exact calculations.

2.2.2 Bounded Degree Predicates

We are restricting ourselves to degree two predicates. In degree two predicates, we are allowed to add and subtract terms as much as we want, but we can only multiply at most two terms together. An example of a degree two predicate is the standard *orientation test*, found using the sign of the following determinate:

$$\text{CCW}(p, q, r) = \text{sign} \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

As an example of a predicate we may not use, consider the predicate that compares an intersection point with an endpoint. Determining the x -order of an endpoint and an intersection is degree three [5]. If we define $t \times u$ as:

$$t \times u = \begin{vmatrix} t_x & t_y \\ u_x & u_y \end{vmatrix} = t_x u_y - t_y u_x,$$

then the intersection a of pq and rs can be computed using:

$$a = p + (q - p) \frac{(r - p) \times (s - p)}{(q - p) \times (r - s)}$$

Predicate test	Degree
x -order of endpoints	1
endpoint above/below segment	2
segment intersection test	2
x -order endpoint and intersection	3
order of intersections on segment	4
x -order of intersections	5

Table 2.1: Degree of predicates for line segments [6].

Comparing the above intersection with an endpoint b , we get:

$$\begin{aligned}
 p + (q - p) \frac{(r - p) \times (s - p)}{(q - p) \times (r - s)} &\stackrel{a}{=} b \\
 p((q - p) \times (r - s)) + (q - p)((r - p) \times (s - p)) &\stackrel{b}{=} b((q - p) \times (r - s))
 \end{aligned}$$

This predicate involves multiplying three terms together, and thus the predicate can be evaluated as a degree three polynomial. To establish that the degree is exactly three, we must show that the polynomial of the predicate is irreducible over the rationals. (This is shown in the appendix of the paper by Boissonnat and Preparata [5].)

Table 2.1 summarizes the precision necessary for various predicate tests. As we can see, it is necessary to use at least degree two predicates in an intersection algorithm, because we need to be able to determine if two segments intersect.

Theorem 1 *The red-blue intersection problem requires at least degree two predicates.*

Proof: If an algorithm existed that solved the red-blue intersection problem using only degree one predicates, we would be able to determine if two segments intersected in degree one, contradicting that the segment intersection test is a degree two polynomial. $\square$

With our restriction of using degree two predicates, of the predicates listed in Table 2.1, we are only able to sort endpoints, test to see if endpoints are above or below a segment, or test if two segments intersect. In Chapter 4, we provide an algorithm to compute the intersections using only these predicates.

2.2.3 Computing the Arrangement Using Low Degree Predicates

Boissonnat and Preparata [5] show that in the general line segment intersection case, degree four predicates are necessary to compute the arrangement. This is because we need to be able to compare the x -order of intersections in able to correctly determine the orientation of a triangle formed by three intersecting segments:

Using only degree two predicates, we can determine that each of the above segments has two intersections, but we cannot determine the orientation of the triangle formed by these three segments. Thus we cannot distinguish between the two possible arrangements.

There are no triangles in the red-blue line segment intersection case. Because of this, our algorithm can correctly determine the arrangement using only degree two predicates. After we run our algorithm, each segment knows the order of its intersections. This gives us sufficient information to determine the vertices, edges, and faces of the arrangement, without actually computing the coordinates of the intersections.

In Section 6.2, we discuss how to compute the arrangement of segments by making small modifications to the algorithm in Chapter 4.

Chapter 3

Pseudo-Segments, Events, and Witnesses

This chapter defines the concepts of pseudo-segments, events, and witnesses that will be used in the algorithm. Boissonnat and Snoeyink [6] were the first to describe pseudo-segments and witnesses, although witnesses are related to ideas used by Boissonnat and Preparata [5].

We use pseudo-segments as a conceptual tool, to help visualize the state of the segments intersecting the sweep line in algorithm SPAGHETTISWEEP. Events and witnesses are specifically used by the algorithm.

3.1 Pseudo-Segments

Pseudo-segments grew out of the idea of pseudo-lines, introduced by Levi [18]. Pseudo-lines are defined in terms of an arrangement of pseudo-lines [13]:

An *arrangement of pseudo-lines* $\mathcal{P}$ in the real projective plane P is any family of simple closed curves $P_1, \dots, P_n$ in P such that every two curves have precisely one point in common, at which they cross each other.

An arrangement of lines is also an arrangement of pseudo-lines, although an arrangement of pseudo-lines may not be homeomorphic to an arrangement of lines.

However, we are dealing with segments in the plane. We are interested in arrangements of curve segments that are homeomorphic to arrangements of segments.

Pseudo-Segment: A pseudo-segment is a closed, continuous curve segment that is indistinguishable from a line segment under a specified set of predicates. All pseudo-segments have the following properties:

- a pseudo-segment has exactly two fixed endpoints, which are the endpoints of the original line segment, and
- a pseudo-segment has an intersection with another (pseudo-)segment only if and only if the corresponding line segment would have such an intersection.

Besides ordering the endpoints, the two predicates from Table 2.1 we are concerned with are:

- endpoint above/below segment, and
- segment intersection test.

As with pseudo-lines, an arrangement of segments is also an arrangement of pseudo-segments. There is a continuous deformation from an arrangement of segments to an equivalent arrangement of pseudo-segments. The intersection properties do not change when the arrangement is deformed.

Essentially, a pseudo-segment is a floppy line segment. The endpoints are fixed, but we do not know (or care about) the exact path from the starting point to the ending point. The paths of these floppy segments can be conceptually nudged to a position that is advantageous to the task at hand.

For our intersection algorithm, we treat the red and blue segments as pseudo-segments in order to push the detection of intersections as far to the right as possible. When drawing pseudo-segments, we will draw them as being monotone in both x and y : in the sweep algorithm, we only really “notice” that the segments change height when they are forced to go around an endpoint. This happens to give the pseudo-segments a “staircase” appearance:

These “staircases” may go up or down depending on the slope of the original segment.

3.2 Events

To find the intersections, we use a left to right sweep of the data, processing “events” as the sweep line encounters them.

According to WordNet [21], one definition of *event* is “something that happens at a given place and time.” Informally, the “something that happens” is that the main data structure in the sweep algorithm needs updating. The “given place” is an (x, y) location in the plane, and the “given time” refers to the event’s lexicographical ordering in the list of events. More formally:

Event: An event is an (x, y) coordinate encountered by the sweep line at which the data structure for the sweep becomes invalid.

In SPAGHETTISWEEP, all events happen at segment endpoints. Every endpoint corresponds to an event at that location in the plane. Although events are located at segment endpoints, there may be fewer events than endpoints: even if multiple segments share an endpoint, there will be only one event for the (x, y) location. By only having one event per location, we are able to batch process all of the changes that happen at one location.

3.3 Witnesses

Because we are treating segments as pseudo-segments, the exact intersection locations are unknown. We need some way to tell that an intersection has occurred during our sweep of the segments. For this we use *witnesses*:

Witness: A witness is the (lower) leftmost endpoint (event) that certifies that the order of segments along the sweep line has changed from the initial order [6].

When the order of segments change, an intersection occurs, and thus the event witnesses an intersection with the change in order.

Figure 3.1 illustrates an intersection being witnessed by the event w . The dashed segment tests as being below w , the solid segment tests as being above w , but before we encounter the witness in the sweep, the dashed segment is above the solid one. The event w witnesses that there is a change in the order, and SPAGHETTISWEEP processes the intersection before continuing with the sweep. Table 3.1 indicates the cases where an event witnesses an intersection.

Theorem 2 *Each intersection has a witness.*

Proof: The witness for an intersection will be the lexicographically first event in the set of potential witnesses. The set of potential witnesses contains all events between the two segments after their intersection, and the (upper) rightmost endpoint of each segment.

Figure 3.1: Pseudo-segments and witnesses. The two curves are pseudo-segments, and the point w is the witness for their intersection.

		<i>upper segment</i>		
		–	=	+
<i>lower segment</i>	–	unknown	unknown	unknown
	=	WITNESS	DEGENERATE	unknown
	+	WITNESS	WITNESS	unknown

Table 3.1: When an event witnesses an intersection. The symbols $+$, $=$ and $-$ indicate if the segment is above, equal to, and below the event respectively.

If there is no event between the two segments after the intersection, then one of the endpoints of the two segments will be the event that witnesses the intersection. $\square$

Witnesses are a subset of the events. The witness of an intersection may be an endpoint of one of the two participating segments, or it may be an endpoint of a different segment. In Figure 3.1, all endpoints are potential witnesses, but only w is actually a witness for the intersection illustrated. The point w is an endpoint of a segment that has not been drawn.

3.4 Delaying Intersections

SPAGHETTISWEEP conceptually treats the line segments as pseudo-segments. By pretending that the segments travel horizontally for as long as possible, we delay the intersections in our left to right sweep, as illustrated in Figure 3.1. Because we delay the intersections, SPAGHETTISWEEP performs a *lazy* sweep:

Lazy Sweep Property: Intersections are not processed until they are witnessed.

Because all intersections have a witness (by Theorem 2) and all witnesses are events, as long as we process events correctly, all intersections will be reported. We describe how we process events in Chapter 4.

The benefit of moving the recording of intersections to the right is that we need look for intersections only when we encounter a witness. This allows us to batch process intersections. Additionally, the witnesses tell us precisely where in our data structure to look for the intersections, since they will always involve segments immediately above and below the witness.

Chapter 4

Algorithm Description

In this chapter, we describe SPAGHETTISWEEP, our degree two algorithm for the red-blue intersection problem. We start by listing the input and output of the algorithm, and the main invariants the algorithm must maintain. Then we outline the data structures used to maintain these invariants, and how the algorithm processes events. Once all of the events have been processed, the final state of the intersection-recording data structures represent the intersections found. If we do not wish to store the intersections, we can output them as they are found.

4.1 Input and Output

As specified in the introduction, the input to our algorithm is two sets of segments, a “red” set and a “blue” set. Intersections between two segments of a single colour are not allowed, unless the intersection is at the endpoint of one of the two segments. The segments are specified by the coordinates of their endpoints.

Algorithm SPAGHETTISWEEP outputs all intersecting pairs of segments. By modifying one of the subroutines, we get COUNTINTERSECTIONS (discussed in Section 6.1), which outputs the number of intersecting pairs of segments. With a small modification, our algorithm can also output the arrangement of segments.

4.2 Invariants

4.2.1 Regular Sweep

In the standard Bentley-Ottmann [4] sweep algorithm for the general line segment intersection problem, a line sweeps across the plane from left to right, processing endpoints and intersections as they are encountered by the sweep line. As the sweep line moves, the following two invariants are maintained:

Invariant 1 *The segments intersecting the sweep line are sorted according to their order along the sweep line.*

Invariant 2 *All of the intersections to the left of the sweep line have been recorded, and those to the right have not.*

An equivalent way of stating Invariant 2 is that the arrangement to the left of the sweep line is known.

As described in Chapter 3, an event is a location where the ordering of segments along the sweep line changes. In the Bentley-Ottmann sweep, the events happen at both segment endpoints and intersections. The endpoints are known in advance, but the intersections are calculated while the algorithm is running. The upcoming intersections can be determined by whether neighbouring segments on the sweep line intersect at some later point. If they do, the intersection is added to the list of events.

When an event is processed, segments may enter or leave the sweep line structure, or two segments may swap order. After updating the order of the segments to reflect this change, the algorithm must check to see if newly neighbouring segments have an intersection at a later point.

In order to maintain the two invariants, the regular sweep algorithm uses two data structures. One structure acts as a priority queue for the events, and the second stores the order of the segments along the sweep line.

4.2.2 Red-Blue Pseudo-Segment Sweep

When the segments are treated as pseudo-segments, the processing of intersections is delayed, and Invariants 1 and 2 are replaced by:

Invariant 3 *The segments intersecting the sweep line are sorted according to the order determined by the witnesses to the left of the sweep line.*

Invariant 4 *All of the intersections whose witnesses are to the left of the sweep line have been processed, and those whose witnesses are to the right have not.*

The order of segments along the sweep line is now consistent with the ordering that would exist if all of the intersections were pushed as far right as possible while keeping the pseudo-segments monotone, as shown in Figure 4.1. We do not detect an intersection until it has been witnessed.

As opposed to the regular sweep algorithm, events now only occur at endpoints of segments. Since all of the events are known in advance, the structure storing the events is static and can be presorted.

Figure 4.1: The processing of intersections is delayed.

4.3 Data Structures

By using specially designed data structures, we can make the handling of events more efficient. The segments intersecting the sweep line are stored in a multi-layered data structure, and there are two presorted lists that are used to support the updating process.

4.3.1 Supporting Lists

The first of the two supporting lists contains all of the events with the duplicates removed.

The second list contains all of the segments lexicographically sorted by their (lower) leftmost endpoint, where the endpoints are stored as (x, y) coordinate pairs. (The exact order is described in Section 5.1.1.) The algorithm uses this list to determine which segments will need to be added next to the sweep line structure.

4.3.2 Sweep Line Data Structures

The main data structure in our algorithm stores the segments that are currently intersecting the sweep line. This data structure is designed to handle efficient updates at events. Our data structure needs to be able to handle three types of queries: what is the order of segments along the sweep line, what are the closest red segments to an event, and what are the closest blue segments to a red segment.

Our motivation for these queries is as follows. There may be many intersections that are witnessed by a single event. Before the event is processed, the red and blue segments will be out of order with respect to that event, and thus we cannot find a single location in the sweep line structure to correspond to the event. We can find the location of the event with respect to the red segments, then search

for the blue segments that intersect the red segments near the event. These blue segments may actually be far from the segment nearest the event if there are many red segments in between:

We need to be able to skip past the intervening red segments to find the next blue segment.

Thus we group maximal consecutive collections of red and blue segments into *bundles*, and link the bundles into a list using *purple pointers*. Purple pointers are used to maintain the above/below relationship between the bundles. As described in Section 4.3.3, we store the red bundles in a *bundle tree*. The bundle tree provides fast access to the list structure representing the segments along the sweep line.

Because the bundles change as events are processed they should be easily modified. Note that we can choose to store the segments within a bundle in a threaded tree to allow for easier traversal when recording intersections.

Our sweep line data structure is illustrated in Figure 4.2.

4.3.3 Asymmetry

Storing bundles of only one colour in the bundle tree introduces some asymmetry into the algorithm. We do this because most search data structures depend on the data being ordered, and red and blue bundles may swap order along the sweep line at events. Bundles of a single colour have a consistent order.

We could make the algorithm more symmetric and use two bundle trees, one for red bundles and one for blue bundles, but this is unnecessary. We can access the red bundles through the bundle tree, and the blue bundles through the purple pointers. Using only one bundle tree saves memory without increasing the running time.

4.3.4 Underlying Tree Structure

Both the bundles and bundle tree use tree structures to store what they contain (segments and bundles). We need efficient find and insert methods for the trees, and also logarithmic time merge and split operations. We have chosen to implement the underlying trees using Sleator and Tarjan's splay trees [26], because splay trees have the necessary $O(\log n)$ amortized running time for each insert, delete, find, merge and split operation.

Figure 4.2: Segments along the sweep line are bundled, and the bundles of one colour are stored in a bundle tree.

4.4 Processing Events

To maintain Invariants 3 and 4, every time the sweep line encounters an event point we need to update the sweep line structure and record any witnessed intersections. To maintain Invariant 3, we need to swap any segments that are out of order, remove any old segments, and insert any new segments. To maintain Invariant 4, if any segments swap while updating the structure for Invariant 3, we need to record the intersections caused by the swap. Additionally, if multiple segments have an endpoint at the event, we need to record the degenerate intersection.

To determine if segments need to be swapped at the event, we classify the segments according to the event location. Compared to the event, segments fall into one of three classes: *above*, *equal*, or *below*. A segment is “equal to” the event if and only if the event is on the segment. Above and below have their standard intuitive meaning. We give a more detailed explanation in Chapter 5.

Now that we have classified the segments, we can also label the bundles. A bundle is an *above* bundle if all of the segments contained in the bundle are *above* segments. Likewise, a *below* bundle contains only *below* segments, and an *equal* bundle will contain *equal* segments. At most two bundles of each colour will contain more than one type of segment, and we will need to split this type of bundle.

In the figures we will abbreviate *above*, *equal*, and *below* as +, =, and -. In order to make it clear when we are using the classes *above*, *equal*, and *below*, versus the relations above, equal, and below, we use italics when we are referring to the classes. Pseudo-code for SPAGHETTISWEEP is given in Figure 4.3.

4.4.1 Main Event Loop

The main loop of SPAGHETTISWEEP iterates through the list of events, updating the sweep line structure. At each event the algorithm takes five steps:

1. Reorder the segments according to the event, recording intersections in the process.
2. Split the bundle tree into three parts, corresponding to the segments above, equal to, or below the event.
3. Make a list of any zero-length segments at the event, and create a new bundle tree with the segments that begin at the event.
4. Record the intersections between all segments that end or begin at the event.
5. Put the bundle tree back together, replacing the old *equal* section with the new section created in Step 3.

Algorithm SPAGHETTISWEEP()*Input.* The sorted list of segments, and the sorted list of events.*Output.* The intersections along each line segment.

```

bTree  $\leftarrow$  INITSENTINELS();
while (e  $\leftarrow$  NEXTEVENT())
1.1.1    $r_+ \leftarrow$  bTree.FINDSMALLESTBUNDLEABOVE(e);
1.1.2    $r^- \leftarrow$  SEPARATEEQ( $r_+$ .down.down, e);
1.1.3   if ( $r^- \neq \text{null} \wedge r^-.$ up.colour = red)  $r_+ = r^-.$ up;
1.1.4   if ( $r_+$ .up  $\neq$  null)
1.1.5       while (WITNESSED( $r_+$ ,  $r_+$ .up, e)) SWAPMERGE( $r_+$ ,  $r_+$ .up);
1.1.6       SEPARATEEQ( $r_+$ .up, e);
1.1.7       if (WITNESSED( $r_+$ ,  $r_+$ .up, e)) SWAPMERGE( $r_+$ ,  $r_+$ .up);
1.1.8   if ( $r_+$ .down.colour = red) GROUPBUNDLES( $r_+$ ,  $r_+$ .down);

1.2.1    $r^- \leftarrow$  bTree.FINDLARGESTBUNDLEBELOW(e);
1.2.2    $r_- \leftarrow$  SEPARATEEQ( $r^-.$ up.up, e);
1.2.3   if ( $r_- \neq \text{null} \wedge r_-.$ down.colour = red)  $r^- = r_-.$ down;
1.2.4   if ( $r_- \neq \text{null} \wedge r_-.$ up  $\neq$  null  $\wedge r_-.$ up.colour = blue)
1.2.5       while (WITNESSED( $r_-$ ,  $r_-.$ up, e)) SWAPMERGE( $r_-$ ,  $r_-.$ up);
1.2.6       SEPARATEEQ( $r_-.$ up, e);
1.2.7       if (WITNESSED( $r_-$ ,  $r_-.$ up, e)) SWAPMERGE( $r_-$ ,  $r_-.$ up);

1.3.1   if ( $r^-.$ down  $\neq$  null)
1.3.2       while (WITNESSED( $r^-.$ down,  $r^-$ , e)) SWAPMERGE( $r^-.$ down,  $r^-$ );
1.3.3       SEPARATEEQ( $r^-.$ down, e);
1.3.4       if (WITNESSED( $r^-.$ down,  $r^-$ , e)) SWAPMERGE( $r^-.$ down,  $r^-$ );
1.3.5   if ( $r^-.$ up.colour = red) GROUPBUNDLES( $r^-$ ,  $r^-.$ up);

1.4.1    $r_+ \leftarrow$  bTree.FINDSMALLESTBUNDLEABOVE(e);
1.4.2    $r^- \leftarrow$  SEPARATEEQ( $r_+$ .down.down, e);
1.4.3   if ( $r^- \neq \text{null} \wedge r^-.$ down  $\neq$  null  $\wedge r^-.$ down.colour = blue)
1.4.4       while (WITNESSED( $r^-.$ down,  $r^-$ , e)) SWAPMERGE( $r^-.$ down,  $r^-$ );
1.4.5       SEPARATEEQ( $r^-.$ down, e);
1.4.6       if (WITNESSED( $r^-.$ down,  $r^-$ , e)) SWAPMERGE( $r^-.$ down,  $r^-$ );
1.4.7   if ( $r^- \neq \text{null} \wedge r^-.$ up.colour = red) GROUPBUNDLES( $r^-$ ,  $r^-.$ up);

2       {bTree $_+$ , bTree $_-,$ old, bTree $_-$ }  $\leftarrow$  SPLIT(bTree, e);
3       bTree $_-,$ new  $\leftarrow$  MAKENEWEQTREE(FINDCONTSEGS(bTree $_-,$ old, e), e);
4       REPORTENDINTERSECTIONS(bTree $_-,$ old, bTree $_-,$ new);
5       bTree  $\leftarrow$  MERGE(MERGE(bTree $_+$ , bTree $_-,$ new), bTree $_-$ );

```

Figure 4.3: Pseudo-code for algorithm SPAGHETTISWEEP.

Step 1

In the first step, we reorder the segments by swapping bundles if the event witnesses intersections. When a non-degenerate intersection occurs, the intersecting segments change order along the sweep line. We continue to swap bundles until all of the witnessed non-degenerate intersections are taken care of, and the segments are grouped by whether they are above, equal to, or below the event. We may need to swap partial bundles if some of the bundles contain mixed classes of segments.

Swapping bundles is described in Section 4.4.2, and Section 4.4.3 describes how to compute the intersections.

Step 2

Once the new correct ordering is achieved, we split the bundle tree into three parts. When we split the tree, we may need to split up to four bundles at the boundaries between the *above*, *equal*, and *below* sections. By splitting the tree, we are able to remove all the segments that end at the event simultaneously. After splitting the tree, we need to check that the *equal* section does not contain any segments that continue past the event. If such segments exist, we break them, and add the continuing sections to the segments that are used in Step 3.

Step 3

In Step 3 we make a list of any zero-length segments at the event, and create a new bundle tree of the segments that start at the event. This new bundle tree will be used to replace the old *equal* section of the tree. We do not add the zero-length segments to the new *equal* tree, because they would have to be removed immediately anyway. To create this new section of the bundle tree, we use any continuing fragments from the old *equal* tree, and any waiting segments from the sorted list of segments that start at the event.

Step 4

In Step 4 we record any degenerate endpoint intersection. The degenerate endpoint intersection will involve all segments in the old *equal* tree, the zero-length segment list, and the new *equal* tree. This step is described in more detail in Section 4.4.3.

Step 5

In the last step, we reassemble the bundle tree. The existing *above* and *below* sections are merged onto the new *equal* section created in Step 3.

Note that we have not yet defined the comparison predicates. We are relying on an intuitive definition here, and we provide the details in Chapter 5.

4.4.2 Swapping Bundles

When an event witnesses non-degenerate intersections, the bundles involved need to be swapped to correct their order along the sweep line.

We define a *witnessed* predicate that takes two neighbouring bundles and the event as input, and returns true if the bundles need to be swapped (the event is a witness to the segment intersections). The order of the two bundles is correct with respect to a previous event. To determine whether the bundles need to be swapped, the witnessed predicate compares the top and bottom segments of the bundles against the new event. For example, if the bottom segment of the lower bundle tests as being above the event, and the top segment of the upper bundle tests as being below the event, the bundles are in the wrong order with respect to the new event. We can summarize the cases for swapping bundles with the following table, where +, = or - indicate the bundle is above, equal to, or below the event respectively:

		<i>upper</i>		
		-	=	+
<i>lower</i>	-	okay	okay	okay
	=	SWAP	okay	okay
	+	SWAP	SWAP	okay

Note that this table is essentially the same as Table 3.1, but in this case, the event is witnessing all of the intersections caused by two bundles swapping, not just a single intersection.

After two bundles are swapped, we merge adjacent bundles if they are the same colour.

The algorithm uses four while loops to make sure all of the bundles are properly swapped to their new position. (We show that these while loops are correct in Section 4.5.1.) We swap whole bundles while the loop iterates. After each individual loop terminates, there may be one last partial blue bundle that needs to be swapped. We check for this case after each of the while loops.

As shown in Figure 4.4, we are interested in four key red bundles for the while loops: the lowest *above* bundle (r_+), the highest and lowest *equal* bundles ($r^=$ and $r_=$), and the highest *below* bundle (r^-).

Note that we split off extra segments only on the side away from the blue bundles we want to swap: we need to have a position to swap the blue bundles

Figure 4.4: The four while loops executed by SPAGHETTISWEEP. The dashed line, where a red bundle may have been split, indicates where the blue bundles are swapped to.

into, and the witness predicate will only work properly if these extra segments are not present in the bundle. We do not split off the extra segments neighbouring the blue bundle to be swapped, because the blue segments will also intersect these red segments on the way past.

If, after splitting the red bundles, there are no blue segments to swap, we merge the adjacent red bundles back together.

The four while loops executed in Step 1 are as follows, where e is the current event:

- swap the blue *below* and *equal* bundles below r_+ :
`while (WITNESSED( $r_+$ ,  $r_+$ .up,  $e$ )) SWAPMERGE( $r_+$ ,  $r_+$ .up);`
- swap the blue *below* bundles below $r_=:$
`while (WITNESSED( $r_=:$ ,  $r_=:$ .up,  $e$ )) SWAPMERGE( $r_=:$ ,  $r_=:$ .up);`
- swap the blue *above* and *equal* bundles above r^- :
`while (WITNESSED( $r^-$ ,  $r^-$ .down,  $e$ )) SWAPMERGE( $r^-$ ,  $r^-$ .down);`
- swap the blue *above* bundles above $r^=$:
`while (WITNESSED( $r^=$ ,  $r^=$ .down,  $e$ )) SWAPMERGE( $r^=$ ,  $r^=$ .down);`

The loops are illustrated in Figure 4.4.

Figure 4.5: The witness w causes the bundles A and B to swap.

4.4.3 Recording Intersections

Intersections are witnessed when one of two things happens: two bundles swap order along the sweep line, or multiple segments start or end at a common point. Non-degenerate intersections between two segments are witnessed when the two bundles swap. Degenerate intersections are detected at a common endpoint.

Non-degenerate Intersections

When an event witnesses two bundles swapping, we get non-degenerate intersections, as illustrated in Figure 4.5. If the red bundle contains r segments, and the blue bundle contains b segments, there will be rb intersections witnessed by the event point p .

Theorem 3 *Assuming the red and blue segments are ordered correctly within each colour, we can correctly determine the order of non-degenerate intersections along each segment when two bundles swap.*

Proof: Let the two bundles being swapped be called A and B , where A is initially above B , as in Figure 4.5.

Consider the bottom segment of A . The first segment it intersects is the top segment of B . The next segment it intersects is the next segment down in B . Each segment of A will intersect the segments of B in top to bottom order.

The segments in B will intersect the segments in A in the reverse order, starting with the lowest segment in A and ending with the highest. $\square$

Corollary 4 *Assuming the red and blue segments are ordered correctly within each colour, we can correctly determine the order of all non-degenerate intersections witnessed by the current event w .*

Proof: The while loops swap only the bundles that are next to each other. As in the proof of Theorem 3, the bottom segment of the top bundle A must intersect every

segment of the bottom bundle B before it can intersect segments in a lower bundle. The bundle B must be swapped above A before the lower bundle can be swapped. Thus the order in which the bundles are swapped gives the correct intersection order. $\square$

In our implementation of the algorithm, we record non-degenerate intersections by storing a linked list of pointers to the other segments intersected for each segment. This allows us to easily walk around faces in the final arrangement.

Degenerate Intersections

Degenerate intersections occur when multiple segments share the same endpoint. If r red segments and b blue segments meet at an endpoint, then there are rb pairwise bicoloured intersections. (Or, if we wish to report all intersections including those of a single colour, then there are $\binom{r+b}{2}$ pairwise intersections.) Instead of storing all $O(rb)$ pairwise intersections, the algorithm creates a counter-clockwise ordered circular linked list of all the segments, and each segment points to its position in the list. This allows us to reduce the storage and time required to record the degenerate intersections, yet each segment knows the counter-clockwise order of all segments around the intersection. Because of the way we record degenerate intersections, the running time of our algorithm is in terms of the number of intersection points, which is less than the number of intersecting pairs of segments.

Theorem 5 *Segments around a degenerate endpoint intersection are reported correctly in counter-clockwise order.*

Proof: The algorithm uses three lists to compute degenerate endpoint intersections: the list of segments ending at the event, the list of zero-length segments, and the list of segments starting at the event. Except for the zero-length segments, the lists are stored as portions of bundle trees—the old and new *equal* sections. The counter-clockwise order can be determined by walking once from top to bottom on the *ending* list and then back up the *starting* list:

As will be proved later, the segments intersecting the sweep line are sorted according to the order determined by the witnesses to the left of the sweep line. The four while loops in Step 1 do not affect the order of the *equal* segments, and thus the ending

segments will be in counter-clockwise order when starting at the top of the old *equal* tree. Because of the way the segments are initially sorted, the starting segments are stored in lexicographic counter-clockwise order when starting at the bottom of the new *equal* tree. The zero-length segments may be legitimately placed anywhere in this counter-clockwise list. We choose to place them after the old *equal* section and before the new *equal* section. $\square$

The algorithm sets each segment to point to its corresponding record as the record in the circular linked list is created. A segment can participate in a maximum of two degenerate endpoint intersections, since continuing segments are broken.

In our applet the additional structural information we store when recording intersections makes the next step of computing the arrangement easier. (Discussed further in Section 6.2.) Additionally, storing the intersections allows us to display the intersecting segments graphically.

4.4.4 Ending and Starting Bundles

Segments end and start at events. Instead of removing and cleaning up after individual segments, we divide the bundle tree into three sections (the sections corresponding to segments above, equal to, and below the event), and deal with them all at once.

The while loops in Step 1 ensure that the *above*, *equal*, and *below* sections are contiguous and distinct when we consider the red and blue segments together. We call split twice, splitting the bundle tree into three trees. One split removes the section consisting of bundles above the event, and the other removes the section consisting of bundles below the event. After we record any degenerate intersections at the event and find the at most two segments continuing after the intersection, we can discard this section of the tree.

Because the zero-length segments are sorted as coming before any of the other segments starting at an event, we can remove all of the zero-length segments from the ordered list of segments before we go on to create the new bundle tree. After the degenerate endpoint intersection is recorded, the zero-length segments can be discarded.

Then, from the sorted list of all segments, we extract the segments that start at the event and form a new bundle tree from these segments plus any continuing segments. Because the segments are in sorted order, we can use a special "insert at top" method. Insertion into the new bundle tree section is simpler when the insertion is always at the top, because we do not need to deal with splitting bundles, checking for null pointers, or handling cases when a segment is inserted between bundles. We start at the bottom with a new bundle, and continue insertion while the segments

	Adjacency	Allowed	Not Allowed
1	Above r_+	+	=, -
2	Below r_+	any	
3	Above r_-	+, =	-
4	Below r_-	=, -	+
5	Above r_+	+, =	-
6	Below r_+	=, -	+
7	Above r_-	any	
8	Below r_-	-	+, =

Table 4.1: Allowed adjacencies when the *above*, *equal*, and *below* sections are disjoint. The symbols +, = and - stand for *above*, *equal*, and *below* bundles (or segments).

are the same colour. When the segment colour changes, we create a new bundle, link the purple pointers, and continue adding segments.

After the new bundle tree is created, we once again have three trees of bundles: one above the event, one starting at the event, and one below the event. We merge these trees together using two merge operations to reform the bundle tree.

4.5 Proof of Correctness

In this section, we prove that algorithm SPAGHETTISWEEP is correct.

4.5.1 While Loops

The first step is to prove that the while loops in Step 1 of the algorithm correctly reorder the segments so that, from top to bottom, all of the *above* segments come before all of the *equal* segments, and all of the *equal* segments come before all of the *below* segments.

Theorem 6 *Assuming the ordering of segments is correct with respect to the events to the left of the sweep line before Step 1 executes, then after Step 1 is complete, the above, equal, and below segments are ordered into disjoint above, equal, and below sections with respect to the current event.*

Proof: Since the segments of a single colour do not intersect, their order does not change. Subsequently, we can assume that the order within a single colour is correct. What we need to show is that, after the while loops have executed, the ordering between the red and blue segments is correct.

Consider the four “bundles” (they are allowed to contain extra segments in this discussion) described in Section 4.4.2—the lowest *above* bundle, r_+ , the highest and lowest *equal* bundles, $r^=$ and $r_=$, and the highest *below* bundle, r^- —and what bundles may be adjacent to them. For example, only blue *above* bundles may be above r_+ , but any blue bundle can validly be below. The full list of allowed adjacencies is given in Table 4.1.

Note that if the order of bundles is correct with respect to the red bundles, the order will also be correct with respect to the blue. If there is a red *above* bundle incorrectly below a blue *below* bundle, then the opposite will also hold true: a blue *below* bundle will also be incorrectly located above a red *above* bundle. By ensuring the order is correct with respect to the red bundles, we are also ensuring the order is correct with respect to the blue.

When we swap bundles at an event, we are moving the blue bundles from a now incorrect position to the correct one. The first while loop takes the blue *equal* and *below* bundles, and moves them below r_+ , making the order correct with respect to Property 1 of Table 4.1. Properties 2 and 7 will always hold, since any bundle may appear below r_+ or above r^- . Properties 3 and 5 are both covered by the second while loop: moving all blue *below* bundles below $r_=$ means that the blue *below* bundles will also be below $r^=$, since $r^=$ is above $r_=$. The third while loop moves blue *above* and *equal* bundles above r^- , ensuring Property 8 holds. Finally, the fourth while loop takes care of Properties 4 and 6 by moving all blue *above* bundles above $r^=$. The while loops correct the order with respect to all the required properties in Table 4.1, and thus the order of the segments after Step 1 will have three disjoint sections: the *above*, *equal*, and *below* sections. $\square$

Note that the relative order of the segments within the three sections is undisturbed.

4.5.2 Algorithm Correctness

To prove that SPAGHETTISWEEP is correct, we must show that Invariants 3 and 4 hold at every iteration.

Theorem 7 *The segments intersecting the sweep line are sorted according to the order determined by the witnesses to the left of the sweep line. (Invariant 3)*

Proof: Invariant 3 holds trivially before SPAGHETTISWEEP starts: no segments are intersecting the sweep line, and thus the order along the sweep line is correct.

The only time pseudo-segments change order along the sweep line is at an event. In addition, segments may only enter and leave the sweep line at an event.

For the induction step, assume that the order of segments along the sweep line is correct with respect to all events that are to the left of the sweep line, and consider the processing which occurs when the next event is reached.

When the sweep line encounters an event, by Theorem 6, Step 1 correctly reorders the segments according to the event. Step 2 splits the bundle tree into the *above*, *equal*, and *below* sections, and Step 5 replaces the old *equal* section with the new. Thus, after the event has been processed and is to the left of the sweep line, the order along the sweep line is correct with respect to that event: all of the old segments have been removed, all of the new segments have been added, and the current segments have been arranged so that they have the correct order.

Because the segments are reordered correctly each time the sweep line hits an event, the order of segments along the sweep line will be correct with respect to the events to the left of the sweep line. $\square$

Theorem 8 *All of the intersections whose witnesses are to the left of the sweep line have been processed, and those whose witnesses are to the right have not. (Invariant 4)*

Proof: Invariant 4 holds at the start of SPAGHETTISWEEP: all witnesses are to the right of the sweep line, and no intersections have been reported.

By the lazy sweep property, intersections are only processed at events.

At an event, Step 1 reorders only those segments whose change of order has been witnessed by the event. After the swaps are complete, by Theorem 6, the segments will be correctly divided into *above*, *equal*, and *below* sections, and thus all of the non-degenerate intersections witnessed by the event will be reported. Step 4 records the witnessed endpoint intersection. Thus, after each event has been processed, any intersections witnessed by the event have been reported.

Since intersections are not reported until they are witnessed, and they are only witnessed when the sweep line processes the event that witnesses them, Invariant 4 holds after every event. $\square$

Corollary 9 *Algorithm SPAGHETTISWEEP correctly determines all of the intersecting pairs of segments.*

Proof: The sweep line only stops once it has passed to the right of all of the segments. Since the intersection witnesses are a subset of the endpoints, all witnesses will be to the left of the sweep line. Thus by Theorem 8, all of the intersections have been reported $\square$

Figure 4.6: The lexicographically earlier event will witness the lexicographically earlier intersection along the segment. In neither case is w_2 actually a witness.

4.5.3 Partial Ordering of Intersections

Now we prove that the intersections will be ordered along each segment.

Theorem 10 *If two intersections along a segment are witnessed by two separate events, then the lexicographically earlier event will witness the lexicographically earlier intersection along the segment.*

Proof: Without loss of generality, let the three segments in question consist of one red segment, r , being intersected by two blue segments, b_1 and b_2 . Let b_1 intersect r earlier than b_2 . The two blue segments cannot intersect because they are the same colour. Let the two events acting as witnesses be w_1 and w_2 , with $w_1 < w_2$.

For the sake of forming a contradiction, suppose w_2 witnesses the earlier intersection, and w_1 the later. There are two cases: w_1 and w_2 are on the same side of r , or w_1 and w_2 are on opposite sides of r . (The cases where one of w_1 or w_2 is on r are equivalent, and the case where both w_1 and w_2 are on r is trivial.)

Consider the first case, where both w_1 and w_2 are on the same side of r . In order to be potential witnesses, w_2 must be between b_1 and r after the intersection, and w_1 must be between b_2 and r after the intersection. An example of this case is illustrated by the diagram on the left of Figure 4.6. Note that, under these conditions, w_1 must also be between b_1 and r after their intersection. But if w_1 is between b_1 and r after their intersection and is lexicographically before w_2 , then w_1 will witness the intersection of b_1 and r before w_2 can. Thus w_2 is not the witness in this case, and the theorem holds.

The second case is where w_1 and w_2 are on opposite sides of r , as illustrated on the right of Figure 4.6. In order to be potential witnesses, w_2 must be between b_1 and r after the intersection, and w_1 must be between b_2 and r after the intersection. Since b_2 intersects r after b_1 , and b_2 and b_1 cannot intersect, the start point of b_2 must also be between b_1 and r after the intersection. The start point of b_2 must be lexicographically less than w_1 , and since $w_1 < w_2$, the start point of b_2 must be

also lexicographically less than w_2 . Thus, the start point of b_2 is the witness of the intersection between b_1 and r , and w_2 is not. Since the start point of b_2 is less than w_1 , the first intersection is witnessed by a lexicographically earlier event, and the theorem holds. $\square$

Theorem 11 *After Step 1 has executed for an event w , we know the order of non-degenerate intersections along each segment up to and including those witnessed by w .*

Proof: The theorem is trivially true if no intersections have been witnessed.

For the induction hypothesis, assume that each intersection whose witness is lexicographically before w has been reported in the correct order along each segment. By Theorem 10, any intersections that have been witnessed by events lexicographically before w occur before any intersections witnessed by w .

By Corollary 4, the non-degenerate intersections at w are reported in the correct order, and thus the theorem holds. $\square$

Corollary 12 *The intersections reported by algorithm SPAGHETTISWEEP are partially ordered: the order of intersections along each segment is known.*

Proof: The sweep line stops once all of the intersections have been witnessed. Thus by Theorems 5 and 11, all of the degenerate and non-degenerate intersections have been correctly reported in a partial order. $\square$

4.6 Algorithm Analysis

In this section we show that algorithm SPAGHETTISWEEP runs in $O(n \log n + k)$ time and $O(n)$ space, where n is the total number of segments, and k is the number of intersection points. Because of the way we detect and report endpoint intersections, our k is in terms of the number of *intersection points*, not the number of *intersecting pairs of segments*.

4.6.1 Time

Our analysis assumes we are using splay trees [26], which have amortized $O(\log n)$ operations for find, insert, merge, and split.

Theorem 13 *If there are a total of n segments, and k intersection points, then algorithm SPAGHETTISWEEP runs in $O(n \log n + k)$ time.*

Proof: After the initial sorting of the data, there are five steps in the algorithm that are repeated at every event:

1. Swap bundles, recording intersections in the process.
2. Split the bundle tree into three parts.
3. Create a new bundle tree with the segments that begin at the event.
4. Record the intersections between all segments that end or begin at the event.
5. Merge the three parts of the bundle tree back together.

Events correspond to the endpoints of segments, and thus there are $O(n)$ events.

There are four while loops that swap the bundles in the first step. At each iteration of the while loop, two bundles are swapped, and at least one intersection is recorded. The swapping of two bundles can take up to $O(rb)$ steps, where r and b are the number of red and blue segments in the bundles, but each of those is to record an intersection, so the total for computing the swaps is $O(k)$.

After two bundles are swapped, the algorithm may need to merge bundles. Since the merge operation can be done in $O(\log n)$ amortized time, and there are at most $O(n)$ merges, it follows that all of the merges can be done in $O(n \log n)$ time.

The number of bundle merges is bounded above by $O(n)$, because the total number of bundles is bounded. The number of bundles increases only when segments are added to the tree, or bundles are split. There are n segments, so the maximum number of bundles created by adding segments is $O(n)$. Bundles are split in two cases: when the entire tree is being split, and when only part of a bundle needs to be swapped. The entire tree is split at most twice for each event, and thus the number of splits due to bundle tree splits is $O(n)$. Partial bundles are swapped when we reach the division point between *above*, *equal*, or *below* parts of the bundle tree. Equivalently, each of the while loops splits at most one red bundle and one blue bundle per event. The total number of splits due to swapping is $O(n)$. Since at most $O(n)$ bundles are created, we can have at most $O(n)$ bundles merged, totaling $O(n \log n)$ time.

The operations in Step 1 consist of recording intersections, or splitting and merging bundles, and total $O(n \log n + k)$ for all iterations.

In the second step, the bundle tree is split a constant number of times for each event. Since the split operation takes $O(\log n)$ amortized time, the total for all the events is $O(n \log n)$.

Step 3 depends on the insertion time. Every segment is inserted only once, and thus a $O(\log n)$ amortized insertion time gives $O(n \log n)$ time total.

Step 4 can be done in $O(n)$ time using an implicit representation of the intersections. Each segment is involved in at most two degenerate intersections, although each degenerate intersection may involve many segments. If all of the separate pairs of segment intersections are recorded individually, this step may take up $O(rb)$ time, where r and b are the number of red and blue segments. We are able to save time by recording all of the separate segment intersections at once. The algorithm walks down the list of segments that are ending, and up the list of new segments. Each segment is visited at most twice, and thus the total over all events is $O(n)$.

Like Step 2, Step 5 depends on the time to merge two bundle trees. Since the merge operation takes $O(\log n)$ amortized time and there are two merges per event, the total for all the events is $O(n \log n)$.

The total for all five steps is: $O(n \log n + k) + O(n \log n) + O(n \log n) + O(n) + O(n \log n) = O(n \log n + k)$. $\square$

4.6.2 Space

Theorem 14 *If there are a total of n segments, and we do not count the optional storing of intersections, then algorithm SPAGHETTISWEEP uses $O(n)$ space.*

Proof: Storing representations of the segments and events takes $O(n)$ space.

The space taken by the trees storing segments in bundles and bundles in the bundle tree is also linear in n : since every bundle contains at least one segment, there are no more than n bundles. $\square$

Chapter 5

Comparisons

In this chapter we outline the comparison methods used by SPAGHETTISWEEP. We also outline some potential pitfalls, which need to be taken into consideration in order that the comparison predicates return the correct results.

5.1 Types of Comparisons

Recall that because we are limiting ourselves to degree two predicates, we cannot calculate exact intersection coordinates for comparison. Thus, all of the comparison routines used are based in part on taking the following determinant:

$$\begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix} = (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x), \quad (5.1)$$

Using the right hand rule, the sign of the result tells us which side of the directed line $\overrightarrow{pq}$ the point r lies. If the sign of the determinant is positive, then r is counter-clockwise from $\overrightarrow{pq}$. If the sign is negative, then r is clockwise from $\overrightarrow{pq}$. Otherwise r is on the line that passes through $\overrightarrow{pq}$.

5.1.1 Initial Sort

Initially the segments are sorted lexicographically by their (lower) leftmost endpoint. That is, the segments are sorted by x coordinate of their (lower) leftmost endpoint, and if a pair of segments have the same x coordinate, they are then sorted by y coordinate.

If both the x and y coordinates of the leftmost endpoints are identical, the segments are sorted in a counter-clockwise order. This comparison is done by evaluating the sign of the determinant in Equation 5.1 where p and q are the endpoints of one segment, and r is the (upper) rightmost endpoint of the other segment:

We will get the same top to bottom ordering if we compare r to $\overline{pq}$ as q to $\overline{pr}$.

It is possible that two segments will lie exactly on top of each other. In this case, the two segments are treated as being equal for the initial sort, unless one of the segments has zero length.

Since a zero-length segment will test *equal* to multiple, unequal segments that start at the same point, we add a condition that zero length segments will test as *below* non-zero-length segments with the same starting point. This serves two purposes: it prevents problems with the sorting algorithm, and it groups all the zero-length segments at each event.

Note that it is not strictly necessary to sort segments with the same starting point in counter-clockwise order in the initial sort. The reason for this additional constraint is that it allows us to skip a “sorting” step when we bundle all the new segments that are to be inserted into the bundle tree at the same time.

5.1.2 Ends After

When we are processing events, we wish to know if any of the old *equal* segments continue past the event, or *end after* the event.

If the x coordinate of the (upper) rightmost endpoint of the segment is greater than that of the event, the segment is considered to *end after* the event. To correctly handle vertical segments, if the two x coordinates are equal then the y coordinates are compared; if the segment’s y coordinate is greater than the event’s y coordinate then the segment is also considered to *end after* the event.

5.1.3 Above, On or Below

The above/below relation is only defined when two items have overlapping horizontal ranges:

Comparable: Two items are *comparable* if they share some horizontal range.

Above/below is not well defined otherwise.

Comparing an Endpoint and a Segment

To determine whether an event is above, on, or below a segment, we use Equation 5.1. We only compare an event to a segment when the segment is intersecting

the sweep line, and thus when event is in the same horizontal range as the segment. Orienting the segment left to right, the counter-clockwise test corresponds to an up/down test (a positive sign indicates the event is above the segment), except when the segment is vertical.

When the segment is vertical, the event will be collinear with the segment. Thus we test to see whether the event falls within the vertical range of the endpoints of the segment, above the top endpoint, or below the bottom endpoint.

Comparisons with Bundles

We perform comparisons against bundles by comparing the item against the top and bottom segments. If the item is below the top segment, and above the bottom segment, then it is considered *equal*. If the item is above the top segment, it is above the bundle. Similarly if the item is below the bottom segment, it is below the bundle.

The only time two bundles are compared is when the underlying splay tree storing the bundles in the bundle tree performs a balancing operation, or we delete a discarded bundle after merging the segments with another bundle. The bundle tree only stores bundles of a single colour, and two bundles do not overlap in their vertical range, so it suffices to compare the two top segments to find the sort order of the two bundles.

Comparing Two Segments

The cases for comparing two segments are not obvious. Since segments may intersect, we may wish to get different answers based on how far the sweep line has moved.

Because we are dealing with pseudo-segments, we cannot test above/below when the two segments are a different colour: the two segments may intersect, but we do not know when.

Since all of the comparisons between different coloured segments (after the initial sort) happen at an event, we are able to infer the current order of the pseudo-segments by comparing the two segments to the event. Comparisons with an event

are unambiguous, and can be used to determine the order of the two segments when they lie on opposite sides of the event. Therefore we need only consider the case where the two segments are the same colour and thus do not intersect. We need to be able to compare segments of the same colour in order to compare bundles.

We can unambiguously order two segments of the same colour, but testing above/below is subtly different from a counter-clockwise test on the endpoints. In the following diagram, a is counter-clockwise from $\overline{cd}$, and c is counter-clockwise from $\overline{ab}$:

In this case, the correct answer is that segment $\overline{cd}$ is above segment $\overline{ab}$.

Theorem 15 *If segment $\overline{cd}$ starts after segment $\overline{ab}$ (and c is not on $\overline{ab}$), then comparing c to $\overline{ab}$ (assigning $r = c$, $p = a$ and $q = b$ in Equation 5.1) will correctly determine if segment $\overline{cd}$ is above or below $\overline{ab}$.*

Proof: We know that when a point is in the same horizontal range as the segment, the counter-clockwise test corresponds to an above/below test. If one point of a segment is above the other segment and the two segments do not intersect, we know that the entire segment will be above the other segment. $\square$

Thus we choose the labeling of the points for use in Equation 5.1 such that $\overline{pq}$ is the segment that starts lexicographically earlier, and r is the starting point of the segment that starts later:

Theorem 16 *If segment $\overline{cd}$ starts after segment $\overline{ab}$, and c is on $\overline{ab}$, then comparing d to $\overline{ab}$ (assigning $r = d$, $p = a$ and $q = b$ in Equation 5.1) will correctly determine if segment $\overline{cd}$ is above, equal to or below $\overline{ab}$.*

Proof: The *equal* case is trivial: if d is collinear with a and b , then all four points are collinear, and since the horizontal ranges of the segments overlap (in order to be comparable), the two segments are *equal*. (Neither segment is considered *above* the other.)

When d is counter-clockwise from $\overline{ab}$, we need to show that $\overline{cd}$ is *above* $\overline{ab}$, even if d is not in the horizontal range of $\overline{ab}$. Since $\overline{cd}$ is a straight line, if d is counter-clockwise from $\overline{ab}$, then every point internal to $\overline{cd}$ (except c) will be counter-clockwise from $\overline{ab}$. This implies that there will be some point d' internal to $\overline{cd}$ that is within the horizontal range of $\overline{ab}$, and thus is comparable to $\overline{ab}$. (SPAGHETTISWEEP does not need to consider the case where $b = c$.) d' is counter-clockwise from $\overline{ab}$, and thus is *above* $\overline{ab}$. Therefore, $\overline{cd}$ is *above* $\overline{ab}$.

The case when d is clockwise from $\overline{ab}$ is similar, and $\overline{cd}$ will be *below* $\overline{ab}$. $\square$

Thus, when c is on $\overline{ab}$, we choose the labeling of the points for use in Equation 5.1 such that $\overline{pq}$ is the segment that starts lexicographically earlier, and r is the ending point of the segment that starts later:

When comparing two segments, we also need to consider the case of comparing two vertical segments. If we simply use the counter-clockwise test of Equation 5.1, the two vertical segments will always test as being equal. A simple check to see if the two segments have overlapping y ranges will handle this case.

Since zero-length segments do not stay in the bundle tree, we do not need to consider the case of zero-length segments after the initial sort.

Note that the above/below comparison defines a partial order on the segments. The segments are totally ordered within a single colour, but the segments are only partially ordered when both colours are considered.

Chapter 6

Extensions

In this chapter we consider several possible extensions to our algorithm.

6.1 Counting Intersections

With a minor modification to SPAGHETTISWEEP, we get an algorithm, COUNTINTERSECTIONS, that counts the intersections efficiently. Instead of reporting all of the intersections when two bundles are swapped, COUNTINTERSECTIONS multiplies the sizes of the two bundles together. For endpoint intersections, the total number of pairwise intersections for m segments is $m(m-1)/2$, which is also a constant time calculation per event.

6.1.1 Number of Segments in a Bundle

To count intersections, we must know the number of segments in a bundle. We use splay trees to store segments in bundles. During any tree operation in a splay tree, we can maintain the number of segments in all subtrees without increasing the amortized running time of the tree operations.

The three splay operations are shown in Figure 6.1. Consider the zig case. Let ξ , η , α , β and γ be the number of segments in x , y , A , B and C , respectively. We can see that the sizes of A , B and C do not change. Before the zig operation we have:

$$\begin{aligned}\xi &= \alpha + \beta + 1 \\ \eta &= \xi + \gamma + 1 = \alpha + \beta + \gamma + 2\end{aligned}$$

After the zig operation, ξ and η become:

$$\begin{aligned}\xi' &= \alpha + \eta' + 1 = \alpha + \beta + \gamma + 2 = \eta \\ \eta' &= \beta + \gamma + 1\end{aligned}$$

Figure 6.1: Keeping track of bundle size during a splay operation. Each case has a symmetric variant (not shown). The operations, from top to bottom are: (a) Zig, (b) Zig-zig, and (c) Zig-zag [26].

The tree sizes in the other splay operations are adjusted similarly.

A find operation is composed of a series of splay operations, and thus the tree sizes are correctly computed. When we split a bundle, the segment on which we split is first splayed to the root, and then we remove one subtree: to adjust the counts, we subtract the size of the removed subtree. Likewise, on a merge, we add the size of the new subtree. Insertions and deletions can be broken down into split and merge operations [26].

To determine the number of segments in a bundle, we simply query the root of the splay tree.

6.1.2 Analysis

The analysis for counting intersections is similar to that for reporting intersections, but a few changes add up to potentially big savings. Reporting all the intersections takes $O(n \log n + k)$ time, but counting the intersections only takes $O(n \log n)$ time even if there are $O(n^2)$ intersections. The main difference in the analysis occurs in Steps 1 and 4 of the main processing loop.

Theorem 17 *Algorithm COUNTINTERSECTIONS counts the number of intersections in optimal $\Theta(n \log n)$ time.*

Proof: When the bundles know the number of segments they contain, the analysis for COUNTINTERSECTIONS differs from SPAGHETTISWEEP only in Steps 1 and 4.

In Step 1, the algorithm uses while loops to swap bundles into place. We are no longer interested in reporting all of the intersections that take place when bundles are swapped. Instead of needing as much as $O(k)$ time to count the intersections in a swap, each swap takes only $O(1)$ time to multiply the two bundle sizes together. Since there are as many as $O(n)$ swaps, Step 1 uses $O(n)$ time to count the intersections, and $O(n \log n)$ to merge the bundles.

In Step 4 the degenerate endpoint intersections are calculated. Because we need only to count the number of intersections, this step now takes constant time per event. The number of events is less than $2n$ if segments share endpoints, and thus the amount of time necessary is $O(n)$.

The amount of time taken in Steps 2, 3 and 5 remain the same at $O(n \log n)$ each. The total for all five steps is now: $O(n \log n) + O(n \log n) + O(n \log n) + O(n) + O(n \log n) = O(n \log n)$. $\square$

Figure 6.2: Making splay trees persistent. Copy root to x path before splaying x .

6.1.3 Recording Without Reporting

Persistent Data Structures

If we do not wish to report each intersecting pair, it is possible to record the intersections in $O(n \log n)$ time and space. This is accomplished by using persistent data structures.

Driscoll et al. [11] define a *persistent* data structure as one that supports access to multiple versions of the data structure. Most data structures are *ephemeral* in that when an update is performed, the old version is destroyed. A persistent data structure allows access to previous versions as well as the current version.

To make the splay trees persistent, we need to copy any node that is modified, and we need a mechanism that allows us to access the previous copy. Any nodes that are not modified do not need to be copied, since they are the same before and after the change to the data structure.

When a node x is splayed to the root, all the nodes on the path from x to the root are modified, and any node not on the path remains unchanged. We copy all the nodes on the path from x to the root, including both x and the root. Child pointers that pointed to unchanged nodes continue to point to those nodes in the old tree. Child pointers that pointed to nodes that will be changing point to the new copies of those nodes. The copy procedure is illustrated in Figure 6.2. In the history we are maintaining, we keep a pointer to the old root.

Once the path has been copied, we may splay x up to the root position. If we now access the tree through the new root x , the tree appears to be a normal splay

tree, consistent with the operations that have been performed so far. If we access the splay tree through the old root, the tree appears to be the splay tree before x was splayed to the root.

When two bundles are swapped, we record the intersections by keeping a pointer to the versions of each bundle that were swapped. For degenerate endpoint intersections, we keep a pointer to each bundle that was involved, and store the pointers in a way that indicates the order of the bundles at that time.

Analysis

Theorem 18 *Recording intersections in a persistent data structure takes $O(n \log n)$ space when splay trees are used as the underlying tree structure.*

Proof: To show that the intersections can be recorded in $O(n \log n)$ space, we need to consider all operations that modify the bundles. The segments in the bundles are stored internally in splay trees. Insert, find, merge, and split all cause a node to be splayed to the root, and thus modify the tree. As shown in Section 4.6, there are $O(n)$ of these operations altogether. The node that is being splayed to the root is at an amortized depth of $O(\log n)$, and thus for each splay operation we need to copy amortized $O(\log n)$ nodes. Therefore, the total space after $O(n)$ operations is $O(n \log n)$. $\square$

Theorem 19 *Recording intersections in a persistent data structure takes $O(n \log n)$ time when splay trees are used as the underlying tree structure.*

Proof: The time analysis for recording intersections is the same as that for counting intersections, except that now we need to take into account the time to copy the root to node path for each operation that involves splaying a node. There are $O(n)$ such operations, and the copy time for each node in the amortized $O(\log n)$ length path from the root to the node being splayed is constant. Thus, the total time for copying is $O(n \log n)$. Since the remainder of the analysis stays the same as for counting intersections, we only need $O(n \log n)$ time to record intersections. $\square$

Analysis Using Different Tree Structures

Splay trees use more than the optimal amount of space when they are made persistent. Because there are amortized $O(\log n)$ rotations every time the splay tree is modified, the total space used is $O(n \log n)$. If we used a different balanced tree structure with only a constant number of rotations per operation, we could reduce the space requirements to $O(n)$.

Figure 6.3: Unconnected groups of segments may nest. For example, the shaded region contains three connected components of segments.

6.2 Finding the Arrangement

In this section we briefly discuss how one could extend SPAGHETTISWEEP to compute the arrangement of segments.

The arrangement allows us to perform Boolean operations on the areas represented by the red and blue polygons. As described in the introduction, an arrangement of a collection of segments is a decomposition of the plane into vertices, edges, and faces, where:

- vertices are intersection points of segments or endpoints of segments,
- edges are maximal connected (open) portions of segments that do not contain any vertices, and
- faces are maximal connected (open) portions of the plane not meeting any vertex or edge.

Although computing the arrangement has not been implemented in the demonstration applet, it is not conceptually difficult. Our applet already provides most of the information we need to determine the arrangement. The intersection information from SPAGHETTISWEEP gives us the order of intersections along each segment, and the order of segments around each endpoint. All that remains to be found is the nesting structure of connected groups of segments. We need to know the nesting structure in order to determine when there are “holes” in the regions, as shown in Figure 6.3.

Figure 6.4: Regions may split or join as the sweep line moves. We know a connected component C is contained in the region R when R splits and then later rejoins.

We can find the nesting structure using the same sweep as we used to find the intersections. As the sweep line moves from left to right, we keep track of regions as they begin, split, join, and terminate.

As shown in Figure 6.4, regions split when the sweep line encounters an event where segments start but no segments end. Similarly, two regions join when segments end at an event, but no new segments start. The number of splits and joins is bounded by $O(n)$. Note that in the case of a split, we need to remember that what later appears to be two regions from a local perspective is actually one region that has split.

As we can see in Figure 6.4, a nested connected component of segments will cause a region to split and later rejoin. By keeping track of when a region joins with itself, we can determine which components of segments were contained within the region.

Regions begin when multiple segments start at the same event, and end when multiple segments end at the same event:

If m segments start at a point, then there will be at most $m - 1$ regions that begin at the point. Likewise, m segments ending will cause at most $m - 1$ regions to end. Thus, the number of regions that begin or end at endpoint intersections is bounded above by $O(n)$.

Regions also begin and end when segments cross:

For every non-degenerate intersection, at most one region starts (ends). Thus we will have no more than $O(k)$ regions starting in this manner, where k is the number of intersection points.

Thus, we will have at most $O(n + k)$ regions in total. Since regions can split and join at most $O(n)$ times, as long as we do not spend more than logarithmic time per split and join, and constant time when regions start and end, we can compute the arrangement in $O(n \log n + k)$ time, where k is the number of intersection points. The space needed will depend on the size of the representation of the arrangement.

6.3 Intersections with More Than Two Colours

In this section, we describe how our algorithm can be modified to find all pairwise intersecting segments or count the number of intersections when there are more than two colours. As we demonstrate, unlike the two colour case, the arrangement cannot be computed using our witness method. Although it is possible to implement the algorithm described in this section, we do not recommend it. The equivalent information can be obtained by running the existing SPAGHETTISWEEP algorithm on the $\binom{c}{2}$ pairwise combinations of the input data layers, where c is the number of layers.

The input consists of sets of segments, “coloured” for reference purposes. As in the two colour case, two segments of the same colour cannot intersect, except at an endpoint of a segment. The number of colours, c , must be constant or the running time will increase by a factor of c .

When we change the algorithm to use three or more colours, we need a bundle tree for each colour of segment. The event being processed is located in each of the bundle trees, then swaps are performed at the boundaries of each of the *equal* sections. At the edges of each *equal* section, *below* bundles of other colours are swapped down, and *above* bundles are swapped up. This isolates the *equal* sections of all the colours. The rest of the algorithm works in essentially the same way as in the two colour case.

The reason we cannot simply use a single purple pointer to access the *above* and *below* bundles as in the two colour case is because, when we search for the nearest segment of one colour, there may be many “between” bundles of the other

colours, and the search along this linked list cannot be done in guaranteed amortized logarithmic time. There are cases where the bundle immediately above the current bundle will not have intersections with the current bundle, but a bundle from much further above will. Thus we need to be able to check the “neighbouring” bundles of all the other colours before we can determine all of the witnessed intersections.

We cannot find the arrangement when there are more than two colours because we do not have enough witnesses:

Theorem 20 *Witnesses cannot be used to determine the order of intersections along a line segment.*

Proof: When a single event witnesses intersections between more than two colours of segments, we cannot determine the order in which the segments intersect along any of the segments. A counter-example is illustrated by the following diagram:

In the two cases, the three segments have the same order along the sweep line before and after the event, but the order of intersections along each segment has been reversed. Two of the three intersections are witnessed in both cases. The order in which those two intersections occurred cannot be determined simply by looking at the order of the segments before and after the event. $\square$

Along a segment, it is possible to determine the order of intersections from segments of one colour, but the interleaving order for two or more colours cannot be determined. Predicates of a higher order than orientation tests are needed: Boissonnat and Preparata [5] show that determining the order of intersections on a segment is a degree 4 test. With double precision predicates, the algorithm can only determine how many intersections occurred, and which pairs of segments intersected.

6.4 Future Work

The algorithm described in this thesis has been test implemented as an application in Java, which has been converted to run as an applet on a web page. This section discusses some potential future directions.

6.4.1 Finding the Arrangement

The current implementation finds the segment intersections, and reports how many intersections were found. Once the intersections have been found, the arrangement can be found. Extending the current applet to find the arrangement would give an indication of the level of difficulty of creating a commercial or library implementation to do this, as well as work out the details for doing so.

6.4.2 Underlying Tree Structure

In the test implementation, splay trees form the underlying structure for storing segments in a bundle and bundles in the bundle tree. A benefit of splay trees is that certain operations (disjoint merge, split) can be performed efficiently. The trees remain balanced, and merges and splits take amortized $O(\log n)$ time.

Splay trees are somewhat complex, and not all library versions have the split and merge operations implemented. It is possible that in practice the trees are small enough that using regular binary trees would not have had a significant effect on the running time. There may also be other types of trees that have or can be modified to have the amortized $O(\log n)$ merging and splitting properties that are simpler, faster, or have less overhead than splay trees. By comparing the performance of various types of trees on real data we could determine which type would be particularly well suited to the implementation in question.

6.4.3 Library Implementation

Currently the algorithm is implemented in Java, and the size of datasets the Java application can handle is limited. Although the current implementation is sufficient to demonstrate how the algorithm works, to make the algorithm usable as a library, it should be written in C/C++ in such a way that it can be used with existing GIS and other programs.

Due to the memory restrictions in Java, the current implementation is limited in the size of data sets it can handle. If there are similar restrictions in C/C++, some modifications in the data structures used will need to be made so that the library can handle sufficiently large data sets. Currently the data is read into contiguous arrays for sorting purposes. The segments are sorted by their leftmost endpoint, and a sorted list of all the endpoints (events) is stored as well. If the segments are stored pre-sorted, they can be read as they intersect the sweep line, and will not need to be in memory simultaneously unless they all intersect the sweep line. The list of events can either be stored pre-sorted in a file, or they can be added to a priority queue as the segments are processed.

There has been some interest in a CGAL [22] implementation, possibly using an iterator style approach [16].

6.4.4 Compare Running Times

Our algorithm has a theoretical running time equivalent to other algorithms that already exist, such as Chan's algorithm [8]. To determine whether this algorithm is faster than existing algorithms, the actual running times should be compared using the same data, and similarly implemented versions on the same hardware.

Whether or not our algorithm is faster than Chan's algorithm, algorithm SPAGHETTISWEEP has advantages in having a lower algorithmic degree and a more symmetric approach.

Bibliography

- [1] I. J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the 11th Annual ACM Symposium on Computational Geometry*, pages 211–219, 1995.
- [2] U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proceedings of the Workshop on Algorithm Engineering*, pages 124–135, 1997.
- [3] J. Basch, L. J. Guibas, and G. D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proceedings of the 4th European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, Berlin, 1996.
- [4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, September 1979.
- [5] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM Journal on Computing*, 29(5):1401–1421, 2000.
- [6] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 370–379, 1999.
- [7] A. Brinkmann and K. Hinrichs. Implementing exact line segment intersection in map overlay. In *Proceedings of the 8th International Symposium on Spatial Data Handling*, pages 569–579. International Geographic Union, 1998.
- [8] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proceedings of the 6th Canadian Conference on Computational Geometry*, pages 263–268, 1994.
- [9] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992.

- [10] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.
- [11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.
- [12] M. T. Goodrich, L. J. Guibas, J. Hershberger, and P. J. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proceedings of the 13th Annual ACM Symposium on Computational Geometry*, pages 284–293, 1997.
- [13] B. Grünbaum. *Arrangements and spreads*. Number 10 in Regional Conference Series in Mathematics. American Mathematical Society, Providence, Rhode Island, 1972.
- [14] L. J. Guibas and D. H. Marimont. Rounding arrangements dynamically. *International Journal of Computational Geometry & Applications*, 8(2):157–178, 1998.
- [15] J. D. Hobby. Practical segment intersection with finite precision output. *Computational Geometry: Theory and Applications*, 13(4):199–214, October 1999.
- [16] V. Kapoor, D. Köhl, and A. Wolff. A tutorial for designing flexible geometric algorithms. In preparation.
- [17] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 351–359, 1999.
- [18] F. Levi. Die Teilung der projektiven Ebene durch Gerade oder Pseudogerade. In *Berichte über die Verhandlungen der sächsischen Akademie der Wissenschaften*, volume 78, pages 256–267. Mathematisch-physikalische Klasse, Leipzig, 1926. (As cited in [13]).
- [19] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series F*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.
- [20] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

- [21] G. A. Miller et al. Wordnet 1.7. <http://www.cogsci.princeton.edu/~wn/>, Cognitive Science Laboratory, Princeton University.
- [22] M. H. Overmars. Designing the computational geometry algorithms library CGAL. In M. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes in Computer Science*, pages 53–58. Springer-Verlag, 1996.
- [23] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP: Graphical Models and Image Processing*, 56(4):304–310, 1994.
- [24] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
- [25] M. I. Shamos and D. J. Hoey. Geometric intersection problems. In *Proceedings of the 17th Annual IEEE Conference on Foundations of Computer Science*, pages 208–215, October 1976. (As cited in [4]).
- [26] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.