Improving Worst-Case Optimal Delaunay Triangulation Algorithms

Geoff Leach
Department of Computer Science
Royal Melbourne Institute of Technology
Melbourne, Australia.
gl@cs.rmit.oz.au

June 15, 1992

Abstract

We present results of an empirical investigation into the performance of two $O(nlogn)$ worst-case optimal Delaunay triangulation algorithms: a divide-and-conquer algorithm and a plane-sweep algorithm. We present improvements which give a factor of 4-5 speed-up to the divide-and-conquer algorithm and a factor of 13-16 speed-up to the plane-sweep algorithm. Experiments using our improved implementations of both algorithms show the plane-sweep algorithm to be slightly faster (about 20%) than the divide-and-conquer algorithm across a range of distributions. Using our fastest implementation of the plane-sweep algorithm a set of points can be triangulated in 7-8 times the time it takes to (merge) sort them.

1 Introduction

The Voronoi diagram of a set $S = \{p_1, p_2, \ldots, p_n\}$ of points in the plane, called *sites*, is a partitioning of the plane into n convex regions, one per site. Each Voronoi region V_i contains all points in the plane closer to p_i than to any other site. The straight line dual of the Voronoi diagram, obtained by adding a line segment between each pair of sites of S whose Voronoi regions share an edge, is called the Delaunay triangulation. Given the Voronoi diagram of a set of sites, $V(S)$, the Delaunay triangulation of those sites, $D(S)$, can be obtained in $O(n)$ time — and *vice versa*. An example Voronoi diagram and Delaunay triangulation are shown in Figure 1.

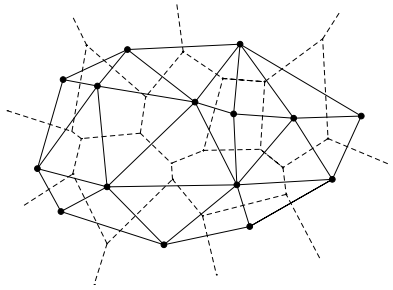

Figure 1: Voronoi diagram (dashed) and Delaunay triangulation (solid).

The Voronoi diagram and Delaunay triangulation are amongst the most useful data structures of computational geometry [4, 11]. Accordingly, there is considerable interest in their efficient computation. Shamos proved $\Omega(nlogn)$ time and $\Omega(n)$ space are required for their computation in the worst-case [13] and with Hoey gave a divide-and-conquer algorithm with matching upper bounds [14]. Their approach was improved by Lee and Schachter who simplified the merge step by constructing the Delaunay triangulation, rather than the Voronoi diagram [10]. Guibas and Stolfi further refine this approach, presenting a comprehensive description of their algorithm in one page [7]. The first $O(nlogn)$ time and $O(n)$ space plane-sweep algorithm is by Fortune [5]. He presents a geometric transformation to overcome a difficulty with adopting the plane sweep paradigm for construction of Voronoi diagrams — that a Voronoi region may be encountered by the sweepline long before the site itself. A plane-sweep algorithm based on a different geometric transformation is presented in [2].

In this paper we discuss improvements which give about a 4–5 fold speed-up to the Guibas-Stolfi algorithm and improvements which give about a 14–16 fold speed-up to to the Fortune algorithm — based on the relative speeds of our fastest and slowest (first) implementations of both algorithms.

2 The Guibas-Stolfi Algorithm

The Guibas-Stolfi algorithm conforms to the standard divide-and-conquer paradigm of solving a problem by recursively breaking it into smaller subproblems and then merging the results of the subproblems to solve the original problem. First the points are sorted lexigraphically (by x-coordinate, with ties resolved by y-coordinate). The sorted sites are then partitioned into two halves, a left half L and a right half R, and the Delaunay triangulation of each half recursively computed. The triangulation is completed by merging the triangulation of L, $D(L)$, and the triangulation of R, $D(R)$. The recursion terminates at either two or three sites, in which case either an edge or a triangle is created.

The merge step is the most complicated and expensive part of the algorithm. It can be thought of as a bottom-to-top "stitching" operation, in which some existing edges of $D(L)$ and $D(R)$ are removed and new L–R *cross edges* are added, as shown in Figure 2. The

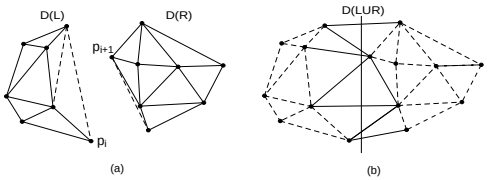

Figure 2: Merging two triangulations. In (a) $D(L)$ and $D(R)$ are shown. Edges which are deleted in the merge are shown dashed. In (b) the merged triangulation is shown. New cross edges are shown solid and existing triangulation edges are shown dashed.

cross edges are determined in vertical order, starting with the lower convex hull edge which is determined in an initialisation step. From then on successive cross edges are found by a three step process: (1) find the best site in L for a cross edge connected to the origin of the topmost cross edge (cross edges are regarded as oriented from R to L), (2) find the best site in R for a cross edge connected to the destination of the topmost cross edge and (3) choose the best site between the two chosen in (1) and (2) and add the next cross edge. The process finishes when the upper convex hull edge has been added.

The best candidate site from L is found by evaluating, in counter-clockwise (ccw) order, the suitability of the sites in $D(L)$ *adjacent* to the L-vertex of the topmost cross edge. Initially the first ccw site is assumed to be the best site. The next ccw site is a better site if it lies inside the circumcircle of the first ccw site and the endpoints of the topmost cross edge, as shown in Figure 3. If D is a better candidate than

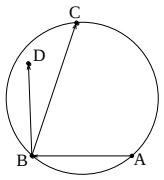

Figure 3: `InCircle` test for better candidates. A and B are the endpoints of the topmost cross edge, C is the first ccw site and D the next ccw site.

C edge $\overline{BC}$ is deleted, C is updated to refer to D and D is updated to refer to the next ccw adjacent site. The iteration stops as soon as a site D lies outside of the circumcircle of A, B and C, or when the site D becomes *invalid*, that is, lies to the left (below) the edge $\overline{AB}$.

The choice between the best candidate site from L and and the best candidate site from R — found symmetrically to the best candidate site from L — is made as follows: if either site is invalid then the other site is chosen, otherwise the `InCircle` test is applied and the site with the smallest circumcircle is chosen. The merge loop terminates when there are no valid sites in either $D(L)$ or $D(R)$.

3 Improvements: Guibas-Stolfi

Our implementation of the Guibas-Stolfi algorithm began as the detailed description given in [7] and went through eighteen versions as we investigated various improvements. Our efforts were directed by timing tests using up to a million points sampled randomly from the unit square and detailed statistics provided by the UNIX profiler gprof. We stopped when our improvements were making little or negative difference to the speed of the algorithm and when we felt we had exhausted the main potential areas. In this section we present the most signficant improvements.

In our first version the break-up of time according to gprof is: 55% in various *quad-edge* (see below) navigation and manipulation routines, 16% calculating the `InCircle` predicate, 12% calculating the `Valid` predicate and the remaining 17% spread across a number of activities — including 2% sorting the sites. We present our improvements in that order (rather than the "order" in which they were implemented).

The Guibas-Stolfi algorithm uses the quad-edge data structure [7] to represent the Delaunay trian-

gulation. We replace it with the winged-edge structure [1]. Both structures are edge based representations of graphs but the quad-edge structure has some extra features, particularly the ability to switch easily from the primal graph to the dual graph, which although elegant require more complicated addressing.

Our second improvement is to the `InCircle` test. Guibas and Stolfi give the following definition for this fundamental predicate of the merge step (A, B, C and D are sites):

$$\mathcal{D}(A,B,C,D) = \begin{vmatrix} x_A & y_A & x_A^2+y_A^2 & 1 \\ x_B & y_B & x_B^2+y_B^2 & 1 \\ x_C & y_C & x_C^2+y_C^2 & 1 \\ x_D & y_D & x_D^2+y_D^2 & 1 \end{vmatrix} > 0.$$

The predicate returns true if site D lies inside the circumcircle of sites A, B and C. Expansion of the determinant gives an expression which, assuming common sub-expression elimination, requires 45 (arithmetic) operations.

We instead use a test based on y-coordinates of circumcircle centres. As shown in Figure 4, for two fixed sites A and B the centre of the circumcircle of A, B and a third point lies on the bisector of A and B. As

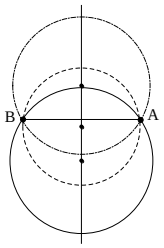

Figure 4: Circumcircle centre test.

the merge step of the algorithm always seeks the best candidate site *above* the topmost cross edge, a site D is a better site than a site C if its circumcircle centre lies lower on the bisector. The best candidate site is the one whose circumcircle centre is the lowest.

Substituting $(x_A, y_A), (x_B, y_B)$, and (x_C, y_C) into

$$(x-p)^2 + (y-q)^2 = r^2$$

and solving for q gives

$$q = \frac{(x_A^2+y_A^2)(x_C-x_B)-(x_B^2+y_B^2)(x_C-x_A)+(x_C^2+y_C^2)(x_B-x_A)}{2((x_B-x_A)(y_C-y_A)-(x_C-x_A)(y_B-y_A))}.$$

This calculation requires 23 operations (ignoring the division by 2 and assuming common subexpression elimination). Two such calculations are needed to compare the first two candidate sites; thereafter only one is needed per candidate site, for which the cost of the `InCircle` test is reduced by approximately half.

The cost is further reduced by *caching*. The topmost cross edge remains fixed for one iteration of the merge loop. Identifying sites A and B as its endpoints, the quantities $x_B - x_A$, $y_B - y_A$, $x_A^2 + y_A^2$ and $x_B^2 + y_B^2$ from the above expression for q are constant for one iteration of the loop. By precalculating them the cost of `InCircle` is reduced to 15 operations.

Another saving made possible by reformulating the `InCircle` test as a comparison of circumcircle centre y-coordinates is to the final `InCircle` test to decide between the best candidate from L and the best candidate from R. This test can be reduced to a simple comparison of two circumcircle y-coordinates, both of which have already been calculated and stored.

Our third improvement involves the `Valid` test. This test determines whether a point lies to the right of a line defined by two points and it's definition is given as

$$\mathcal{D}(A,B,C) = \begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} > 0.$$

Expanding the determinant and rearranging gives

$$\mathcal{D}(A,B,C) = (x_B-x_A)(y_C-y_A)-(x_C-x_A)(y_B-y_A)$$

which is just the cross product of two vectors formed from three points. The cost of the test, assuming caching of $x_B - x_A$ and $y_B - y_A$ is 5 operations. We use the test as given, but use it in two ways to reduce the cost of the `InCircle` test. The first is to use the quantity calculated in the `Valid` test as the denominator in the circumcircle y-coordinate calculation — notice the two expressions are the same. The cost of the `InCircle` test is then 10 operations. The second is as a short-circuit. Restructuring the loops which find the best candidates to always test for validity of a site before performing an `InCircle` test meets the need to detect and exclude invalid sites (handled automatically by the original test) with a cheaper test than `InCircle`.

Our fourth improvement, which occured throughout development, is to code the winged-edge navigation and manipulation routines, and the `InCircle` and `Valid` tests as macros to save on function calls. For the same reason we eliminated recursion from our fastest implementation, but, to our surprise, found it to be no quicker — and less clear.

Taken together the improvements increase the speed of the algorithm by a factor of about 4–5. We discuss the impact of the improvements on space and the issue of robustness in section 6.

4 The Fortune Algorithm

The Fortune algorithm combines the plane-sweep paradigm [4, 11] with a geometric transformation. The transformation overcomes the difficulty for plane-sweep approaches that a Voronoi region may start before the corresponding site is encountered or, in the case of the Delaunay triangulation, that a site may invalidate many Delaunay triangles formed so far.

The transformation adds the distance to the closest member of S to a point's y-coordinate. Some of its effects are as follows. A site p is mapped to itself, as the closest member of S is p and the distance from p to p is zero. A bisector B_{pq} between sites p and q, with $p_y > q_y$, is mapped into a hyperbola B^*_{pq} that has it's lowest point at p and which has two arms, or *boundaries*, that extend upwards, or into a vertical line if B_{pq} is vertical, as shown in Figure 5. A

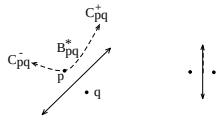

Figure 5: Transformed bisectors

Voronoi edge is mapped to a hyperbola segment and a Voronoi vertex is mapped to an intersection of three (or more) hyperbolae. A complete Voronoi diagram and its image under the transformation together with an example sweepline are shown in Figure 6.

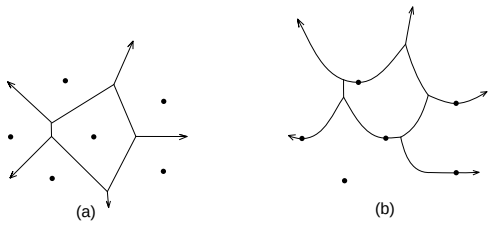

Figure 6: (a) Original Voronoi Diagram. (b) Transformed Voronoi diagram with sweepline (S).

The plane-sweep technique has two characteristic data structures: the sweep line status L and the event queue Q. The exact content and requirements of these data structures depends on the application. In this case L maintains the horizontal order (for a vertical sweep) of the boundaries of $V(S)^*$ intersected by the horizontal sweep line, as shown in Figure 6, and the event queue Q stores two types of events: site events and intersection events.

The actions required for a site event are to insert into L the boundaries of the associated transformed Voronoi region and to update Q by deleting intersection events between boundaries which are no longer neighbours and inserting intersection events between boundaries which have just become neighbours. The actions required for an intersection event, which marks the top of the circumcircle of three sites, are to replace the intersecting boundaries in L with one new boundary, to update Q as for a site event and to mark the bisectors corresponding to the intersecting boundaries with the Voronoi vertex corresponding to the untransformed intersection point, or, if the Delaunay triangulation is sought, to output a Delaunay triangle.

The sweep line status must thus support insert, find, delete, successor and predecessor operations — the operations of an (augmented) *dictionary*. The event point queue must must support delete-min, insert and delete operations — the operations of a *priority-queue*.

5 Improvements: Fortune

The Fortune algorithm is presented at a higher level than the Guibas-Stolfi algorithm. Details of the data structures are left largely in the implementors' hands. After getting our first $O(nlogn)$ version working we proceeded as for the Guibas-Stolfi algorithm. In this section we discuss the most significant improvements.

In our first version L is implemented as a red-black tree [6], augmented with successor and predecessor pointers for $O(1)$ neighbour access and Q is implemented as an indirect implicit heap [12]. According to gprof 83% of the total time is spent in tree operations and 9% in heap operations. Tree searches, including those performed as part of insertions and deletions, account for 78% of total time. This includes 25% of total time for square root calculations.

Our improvements to tree operations fall into two categories: improvements to reduce their number and improvements to reduce their cost. There are three improvements to reduce their number. The first is to insert a pair of boundaries with only one tree search. The second is to delete only one of the two boundaries involved in an intersection event whilst reusing the other with updated information (constituting a *change* operation). The third is perform deletions without a tree search by storing a back-pointer with intersection events. These changes reduce the number of tree searches by about four-fifths. To allow tree rebalancing these changes require bottom-up, rather than top-down, red-black tree insertion and parent pointers.

Before discussing improvements which reduce the

cost of tree searches we elaborate on the calculations involved. The boundaries stored in L are ordered along the x-axis. They divide the horizontal sweep line into segments as shown in Figure 6. The segments' endpoints vary with y but their order remains the same between any two events. A (binary) tree search, in which *site-boundary* comparisons are performed, is used to determine the segment whose x-coordinates contain s_x, the x-coordinate of a site s (the only tree searches performed are for insertions associated with site events). The most direct way to perform to perform a site-boundary comparison is to compute the x-coordinate of a boundary given s_y, the y coordinate of a site. The equation of a transformed bisector is

$$B^*_{pq} = \{(x, y^*) : y^* = y + ((y - p_y)^2 + (x - p_x)^2)^{1/2}\}$$

where a point on the bisector has the distance from it to the site p added to it's y-coordinate. Eliminating y by using the equation $y = mx+c$ of the untransformed bisector, substituting s_y for y^* and rearranging gives a quadratic equation for the x-coordinate of the boundary at s_y, which can solved and the result compared with s_x. This is how we first performed site-boundary comparisons.

Our first improvement to reduce the cost of site-boundary comparisons is to eliminate the square root operation required to solve the quadratic equation. Rearrangement of the equation of a transformed bisector gives

$$(s_y - (mx + c))^2 = ((mx + c) - p_y)^2 + (x - p_x)^2.$$

If we replace x by s_x we have an expression on the left which can be compared with the expression on the right to see if s_y lies below, above or on the boundary, and hence whether, assuming a negative boundary, the site lies to the left, to the right or on the boundary. The test amounts to a comparison of two (squared) distances: the vertical distance from the site to the untransformed bisector B_{pq} and the distance from the point on B_{pq} at s_x to p by which the untranformed bisector is transformed. If m and c are calculated and stored with each boundary the cost of the test is 9 operations.

Our second improvement to reduce the cost of tree searches is to use short-circuit tests. We give two. The first is that a site s lies to the left of the right boundary C^+_{pq} (where $p_y > q_y$) if $s_x < p_x$ and to the right of the left boundary C^-_{pq} if $s_x > p_x$. This short-circuits about half of the site-boundary calculations. The second is that a site s lies to the left of the boundary C^-_{pq} if $s_x < i_x$, where i_x is the x-coordinate of the boundary's intersection point with either of it's neighbours, and vice-versa for C^+_{pq}. With both of these short-circuits in place we find that the more expensive test is needed in only about 10% of cases.

With the above improvements to the tree operations, the main event queue operations take about 43% of the total time, compared with about 9% in our first implementation. We give two improvements to event queue operations. The first is to separate the event queue into a site event queue and an intersection event queue and to use different representations and algorithms. The sites form a static set of events, all of which are known at the outset and all of which must be processed. The site event queue is built by simply sorting the sites. The intersection events, however, form a dynamic set of events, discovered on the fly, and need to be stored in a priority queue. By storing only intersection events in the priority queue its size is reduced and operations are speeded. The next event is determined by comparing the minimum event of both queues. Separating the queues also allows the sites to be stored in single (or double) precision whilst storing intersection points in double precision — we found intersection points had to be stored in double precision to improve robustness to an acceptable level.

The second improvement to event queue operations is to store only one intersection event, the lowest, per boundary. As presented in [5], the algorithm finds and stores two intersection events per boundary, one for each neighbouring boundary. However only the lowest is ever processed — the other is always deleted.

Our final improvement is to recode some small functions as macros, which we did throughout the investigation, although we did not make a special effort to reduce function calls to a minimum, preferring to adhere to a programming style we are comfortable with.

Overall, the improvements gave a speed-up factor of about 13-16. We discuss the impact of the improvements on space and the issue of robustness in section 6.

6 Experiment Results

In Table 1 we give results of our timing experiments for points sampled from the unit square. These results are from a lightly loaded Sun Sparcstation 2 with 64Mbytes of main memory. The programs are written in C. Gaps in the table represent experiments where our slowest implementations failed. Using the largest experiments for which we have results the speed-up factors are 4.9 for the Guibas-Stolfi algorithm and 16.3 for the Fortune algorithm. In part, the difference in speed-up is because we started with a less polished implementation of the Fortune algorithm. The fastest

	GS		F	
	s	f	s	f
2^{10}	0	0	2	0
2^{11}	1	0	6	0
2^{12}	4	0	12	1
2^{13}	9	2	27	2
2^{14}	20	4	57	5
2^{15}	44	9	120	9
2^{16}		20	277	17
2^{17}	204	42		38
2^{18}		91		76
2^{19}		195		164
2^{20}		410		333

Table 1: Experiment results for our slowest (s) and fastest (f) implementation of the Guibas-Stolfi (GS) algorithm and the Fortune (F) algorithm.

implementation of the Fortune algorithm is about 25% faster than the fastest implementation of the Guibas-Stolfi algorithm.

We also ran experiments using a number of other distributions, summarised in the following table. Distributions B and C are intended to approxi-

Key	Distribution
A	uniform within unit square
B	A with a x scale factor of 100
C	A with a y scale factor of 100
D	normal$(0,1)^2$
E	clustered normal: five normal$(p,1)^2$, $p \epsilon A$

Table 2: Distributions

mate the best-case and worst-case for the divide-and-conquer algorithm and the worst-case and best-case for the plane-sweep algorithm.

In Table 3 we give results for the above distributions. In each case we give the times for the largest experiment for which the slowest implementation succeeded. The results indicate that the improvements we made on the basis of results obtained from distribution A extended to the other distributions.

In Table 4 we give results for experiments of 2^{20} sites using our fastest implementations of both algorithms. Results for distributions B and C are swapped around for the Fortune algorithm so that the fastest result for the Guibas-Stolfi algorithm is compared with the fastest result for the Fortune algorithm, and likewise for the slowest results. On average the plane-sweep

	GS				F			
	n	s	f	s/f	n	s	f	s/f
A	2^{17}	204	42	4.9	2^{16}	277	17	16
B	2^{16}	57	13	4.4	2^{13}	32	2	16
C	2^{16}	112	28	4.0	2^{16}	224	16	14
D	2^{15}	45	10	4.5	2^{17}	494	39	13
E	2^{16}	95	19	5.0	2^{16}	227	17	13

Table 3: Ratios of slowest (s) to fastest (f) times.

	GS	F	GS/F
A	410	333	1.2
B	288	290	1.0
C	535	376	1.4
D	412	329	1.3
E	398	325	1.2

Table 4: Ratios of fastest implementations.

algorithm is faster by a factor of 1.2. Experiments on a Silicon Graphics Personal Iris (4D/20) gave similar results — to within a nearly constant scale factor. This is a small difference, particularly in view of the speed-up factors of 4-5 and 13-16.

We turn now to space requirements. Both algorithms store the n sites. In addition the Guibas-Stolfi algorithm stores the planar graph, which has at most $3n - 6$ edges. There are 24 bytes/edge in our representation and we maintain an edge free list at 4 bytes/edge. So just under $84n$ bytes is required to store the edges. The plane-sweep algorithm stores the sweep line status and the event queue. In the worst-case for the plane-sweep algorithm there could be $2(n-1)$ boundaries and $n-2$ intersection events present at the one time (Figure 7). There are 80

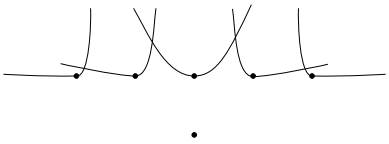

Figure 7: Worst-case for plane-sweep algorithm.

bytes/boundary and 32 bytes/event in our representation. We also maintain three free lists (intersection events, boundaries and tree nodes) at 4 bytes/item. Together, these structures require just under $212n$ bytes in the worst-case. However, the average case re-

quirements are considerably less. We use an estimate of $an^{1/2}$ for the number of boundaries and intersection events, where a is a scale factor, which had a maximum value of 10 in our experiments and thus for 2^{20} sites about 1.5 Mbytes, in addition to the 8 Mbytes for the sites, was required.

As to ease of implementation our subjective assessment is that the Guibas-Stolfi algorithm is slightly easier to implement. Our fastest implementation of the Guibas-Stolfi algorithm is 1087 lines of code and our fastest implementation of the Fortune algorithm is 2490 lines of code. Numerical robustness is a common problem. Fortuitously, the improvements discussed in Sections 3 and 5, increased robustness as well as speed (although our most robust (fastest) versions still occasionally fail). The impact of the improvements on numerical robustness needs further investigation (see [9] for discussion of a rational arithmetic implementation of the Guibas-Stolfi algorithm and references to other results relating to robustness and correctness of geometric algorithms).

7 Future Work

In investigations of this sort there is always possibility of further improvements. We suggest one: replacement of the implicit heap structure used for the priority queue in the plane-sweep algorithm with a newer heap structure such as a splay-heap [8, 15].

Another area of future investigation is improvement of the average case behaviour whilst retaining worst-case optimality. For instance, Dwyer shows how the a gridding technique using $O(n)$ space can be employed to improve the average case complexity to $O(nloglogn)$ for a large range of distributions whilst retaining $O(nlogn)$ worst-case complexity [3].

References

[1] B. G. Baumgart. A Polyhedron Representation for Computer Vision. *National Computer Conference*. AFIPS Conference Proceedings, vol. 44, pp. 589-596, AFIPS Press, Arlington, Va., 1975.

[2] F. Dehne and R. Klein. A Sweepcircle Algorithm for Voronoi Diagrams. *Proceedings of Graph-Theoretical Concepts in Computer Science*, published as Lecture Notes in Computer Science pp. 59-69, Springer-Verlag, 1987.

[3] R. A. Dwyer. A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations. *Algorithmica*, 2:137-151, 1987.

[4] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, 1987.

[5] S. Fortune. A Sweepline Algorithm for Voronoi Diagrams. *Algorithmica*, 2:153-174, 1987.

[6] L. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. *19th Annual Symposium on Foundations of Computer Science*. IEEE, 1978.

[7] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Transactions on Graphics*, 4:74-123, April, 1985.

[8] D. W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300-311, April, 1986.

[9] M. Karasick, D. Lieber and L. R. Nackman. Efficient Delaunay Triangulation Using Rational Arithmetic. *ACM Transactions on Graphics*, 10(1):71-91, Jan., 1990.

[10] D. T. Lee and B. Schachter. Two Algorithms for Constructing Delaunay Triangulations. *Int. J. Comput. Inform. Sci.*, 9(3):219-242, 1980.

[11] F. P. Preparata and M. I. Shamos. *Computational Geometry - an Introduction*. Springer-Verlag, New York, 1985.

[12] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, Mass., 1988.

[13] M. I. Shamos. Computational Geometry. Ph.D. Dissertation, Yale University, New Haven, Conn., 1977.

[14] M. I. Shamos and D. Hoey. Closest-point problems. *Proc. 16th IEEE Symposium on Foundations of Computer Science*, pp. 151-162, 1977.

[15] D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal on Computing*, 15(1): pp. 52-69, Feb., 1986.