◊.1

Incremental Delaunay Triangulation

Dani Lischinski
580 ETC Building
Cornell University
Ithaca, NY 14850, USA
danix@graphics.cornell.edu

◊ Introduction ◊

This gem gives a simple algorithm for the incremental construction of the Delaunay triangulation (DT) and the Voronoi diagram (VD) of a set of points in the plane. A triangulation is called *Delaunay* if it satisfies the empty circumcircle property: the circumcircle of a triangle in the triangulation does not contain any input points in its interior. DT is the straight-line dual of the *Voronoi diagram* of a point set, which is a partition of the plane into polygonal cells, one for each point in the set, so that the cell for point p consists of the region of the plane closer to p than to any other input point (Preparata and Shamos 1985,Fortune 1992).

Delaunay triangulations and Voronoi diagrams, which can be constructed from them, are a useful tool for efficiently solving many problems in computational geometry (Preparata and Shamos 1985). DT is optimal in several respects. For example, it maximizes the minimum angle and minimizes the maximum circumcircle over all possible triangulations of the same point set (Fortune 1992). Thus, DT is an important tool for high quality mesh generation for finite elements (Bern and Eppstein 1992). It should be noted, however, that standard DT doesn't allow edges that must appear in the triangulation to be specified in the input. Thus, in order to mesh general polygonal regions the more complicated *constrained* DT should be used (Bern and Eppstein 1992).

The incremental DT algorithm given in this gem was originally presented by Green and Sibson (Green and Sibson 1978), but the implementation is based entirely on the quad-edge data structure and the pseudocode from the excellent paper by Guibas and Stolfi (Guibas and Stolfi 1985). I will briefly describe the data structures and the algorithm, but the reader is referred to Guibas and Stolfi for more details.

ISBN 0-12-XXXXXX-X

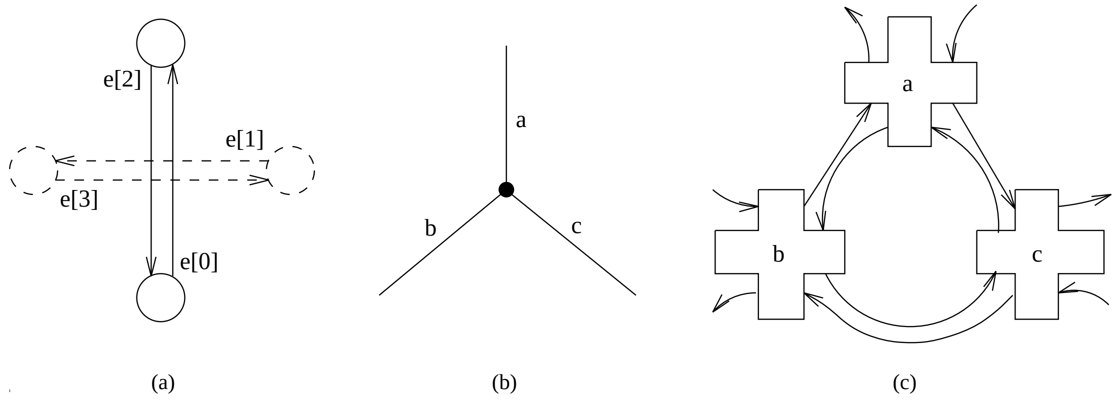

Figure 1. The quad-edge data structure.

◊ The Quad-Edge Data Structure ◊

The quad-edge data structure (Guibas and Stolfi 1985) was designed for representing general subdivisions of orientable manifolds. It is similar to the winged-edge data structure (Baumgart 1975), but it simultaneously represents both the subdivision and its dual. Each quad-edge record groups together four directed edges corresponding to a single undirected edge in the subdivision and to its dual edge (Figure 1a). Each directed edge has two pointers: a `next` pointer to the next counterclockwise edge around its origin, and a `data` pointer to geometrical and other nontopological information (such as the coordinates of its origin.)

Figures 1b and 1c illustrate how three edges incident on the same vertex are represented using the quad-edge data structure: the vertex itself corresponds to the inner cycle of pointers in Figure 1c. The remaining three cycles correspond to the three faces meeting at the vertex.

Aside from a primitive to create an edge (`MakeEdge`), a single topological operator `Splice` is defined that can be used to link disjoint edges together as well as to break two linked edges apart. This operator is its own inverse and together with `MakeEdge` it can be used to construct any subdivision.

◊ The Incremental Algorithm ◊

The incremental DT algorithm starts with a triangle large enough to contain all of the points in the input. Points are added into the triangulation one by one, maintaining the invariant that the triangulation is Delaunay. Figure 2 illustrates the point insertion process. First, the triangle containing the new point p is located (2a). New edges are created to connect p to the vertices of the containing triangle (2b). The old edges of the

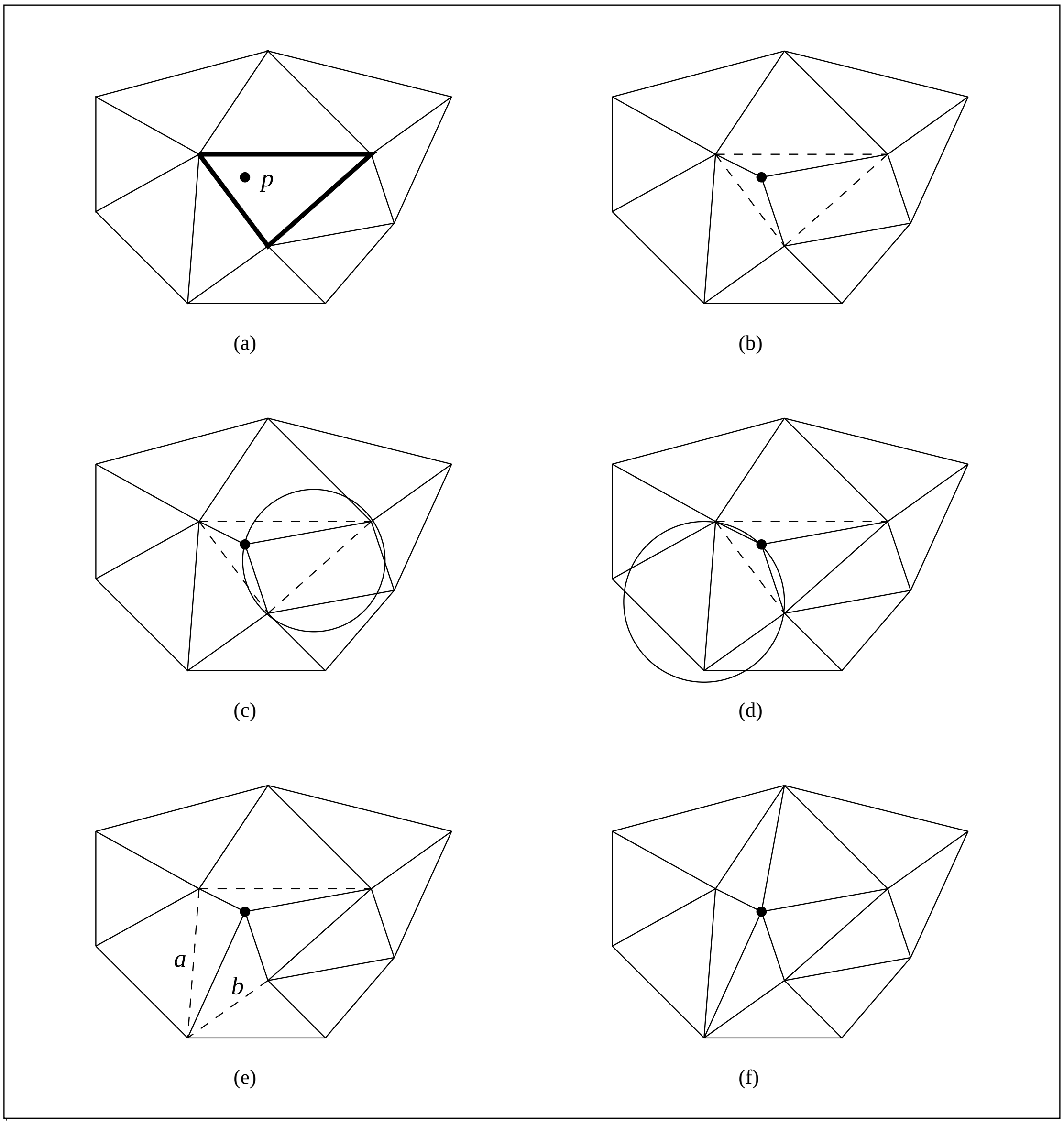

Figure 2. Inserting a point into the triangulation. Dashed lines indicate edges that need to be inspected by the algorithm.

triangle are inspected to verify that they still satisfy the empty circumcircle condition. If the condition is satisfied (2c) the edge remains unchanged. If it is violated (2d) the offending edge is flipped, that is, replaced by the other diagonal of the surrounding quadrilateral. In this case two more edges become candidates for inspection (edges a and b in Figure 2e.) The process continues until no more candidates remain, resulting in the triangulation shown in Figure 2f.

In the worst case the insertion of a point can require $O(n)$ edges to be flipped. However, in practice the average number of edges tested per insertion is small (< 9). Guibas, Knuth, and Sharir have shown that if the insertion order is randomized, the expected time is $O(1)$ per insertion (Guibas et al. 1990).

Locating the containing triangle can be done in an optimal $O(\log n)$ time, but this requires maintaining complicated data structures. Alternatively, the triangle can be located by starting from an arbitrary place in the triangulation and moving in the direction of p until the containing triangle is reached. This requires $O(n)$ time, but if the inserted points are uniformly distributed, the expected number of operations to locate a point is only $O(n^{1/2})$. A simple improvement is to always resume the search from the triangle that was found last: in this way, when the points to be located are near each other, the containing triangles are determined quickly.

Figure 3 shows the DT and the corresponding VD produced by this algorithm from 250 random points in the unit square. Note that because the quad-edge data structure represents both the triangulation and its dual, the topology of the Voronoi diagram is readily available from the DT constructed by the algorithm. To have a complete VD one only needs to compute the circumcenters of all the triangles (i.e., the locations of the Voronoi vertices.)

♢ Robustness ♢

In order to produce a practical implementation of a geometric algorithm, one typically needs to address two problems: geometric degeneracies and numerical errors. For DT, four or more cocircular points in the input constitute a geometric degeneracy and the resulting DT is not unique. In such a case this algorithm will produce one of the possible triangulations as output.

Dealing with numerical errors is more difficult. Various applications in which the need for DT or VD arises differ in the nature of their input and in their output accuracy requirements. Therefore, it is very difficult to come up with a single efficient solution to the problem. Karasick, Lieber and Nackman suggest a solution that uses rational arithmetic as well as survey other approaches (Karasick et al. 1991).

In this implementation all the computations are performed using standard floating point arithmetic. Epsilon tolerances are used to determine whether two point coincide and whether a point falls on an edge. No other special measures to ensure robust-

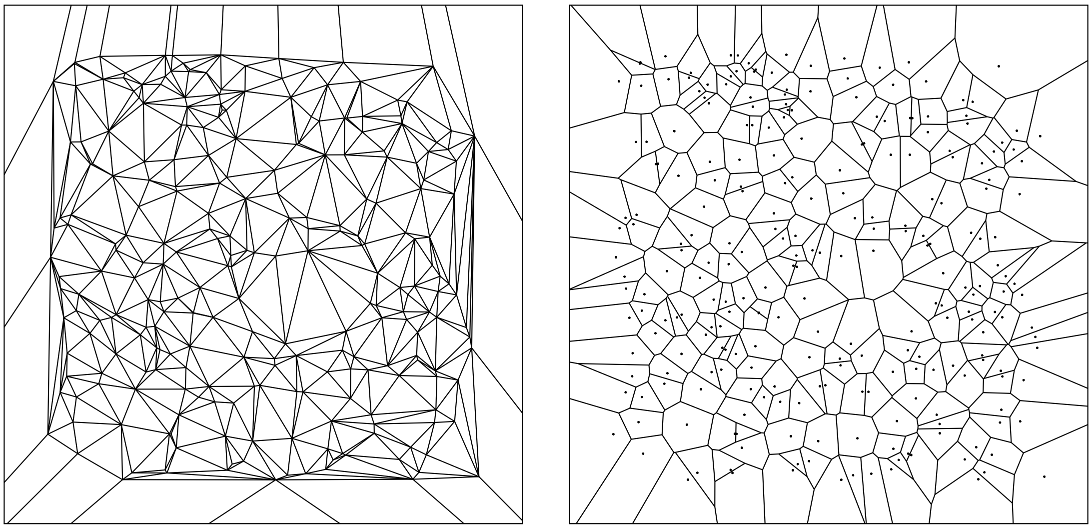

Figure 3. The DT (left) and the VD (right) of 250 random points uniformly distributed in the unit square.

ness were taken. Nevertheless, largely because of the simplicity of the algorithm, the implementation has proven to be very robust.

◊ C++ Code ◊

The code listed below is the C++ implementation of the quad-edge data structure and the incremental Delaunay triangulation algorithm. In addition, the disk that comes with this book contains code for 2D vectors, points, and lines and a test program. This program constructs and displays a triangulation, allowing the user to add more points into the triangulation interactively by clicking a mouse button at the place of insertion. The code should compile and execute on SGI graphics workstations.

```
#include <geom2d.h>

class QuadEdge;

class Edge {
    friend QuadEdge;
    friend void Splice(Edge*, Edge*);
  private:
    int num;
    Edge *next;
    Point2d *data;
  public:
    Edge() { data = 0; }
    Edge* Rot();
```

```
    Edge* invRot();
    Edge* Sym();
    Edge* Onext();
    Edge* Oprev();
    Edge* Dnext();
    Edge* Dprev();
    Edge* Lnext();
    Edge* Lprev();
    Edge* Rnext();
    Edge* Rprev();
    Point2d* Org();
    Point2d* Dest();
    const Point2d& Org2d() const;
    const Point2d& Dest2d() const;
    void  EndPoints(Point2d*, Point2d*);
    QuadEdge* Qedge()   { return (QuadEdge *)(this - num); }
};

class QuadEdge {
    friend Edge *MakeEdge();
  private:
    Edge e[4];
  public:
    QuadEdge();
};

class Subdivision {
  private:
    Edge *startingEdge;
    Edge *Locate(const Point2d&);
  public:
    Subdivision(const Point2d&, const Point2d&, const Point2d&);
    void InsertSite(const Point2d&);
    void Draw();
};

inline QuadEdge::QuadEdge()
{
    e[0].num = 0, e[1].num = 1, e[2].num = 2, e[3].num = 3;
    e[0].next = &(e[0]); e[1].next = &(e[3]);
    e[2].next = &(e[2]); e[3].next = &(e[1]);
}

/************************ Edge Algebra *************************************/

inline Edge* Edge::Rot()
// Return the dual of the current edge, directed from its right to its left.
{
    return (num < 3) ? this + 1 : this - 3;
}

inline Edge* Edge::invRot()
// Return the dual of the current edge, directed from its left to its right.
```

```
{
    return (num > 0) ? this - 1 : this + 3;
}

inline Edge* Edge::Sym()
// Return the edge from the destination to the origin of the current edge.
{
    return (num < 2) ? this + 2 : this - 2;
}

inline Edge* Edge::Onext()
// Return the next ccw edge around (from) the origin of the current edge.
{
    return next;
}

inline Edge* Edge::Oprev()
// Return the next cw edge around (from) the origin of the current edge.
{
    return Rot()->Onext()->Rot();
}

inline Edge* Edge::Dnext()
// Return the next ccw edge around (into) the destination of the current edge.
{
    return Sym()->Onext()->Sym();
}

inline Edge* Edge::Dprev()
// Return the next cw edge around (into) the destination of the current edge.
{
    return invRot()->Onext()->invRot();
}

inline Edge* Edge::Lnext()
// Return the ccw edge around the left face following the current edge.
{
    return invRot()->Onext()->Rot();
}

inline Edge* Edge::Lprev()
// Return the ccw edge around the left face before the current edge.
{
    return Onext()->Sym();
}

inline Edge* Edge::Rnext()
// Return the edge around the right face ccw following the current edge.
{
    return Rot()->Onext()->invRot();
}

inline Edge* Edge::Rprev()
```

```
// Return the edge around the right face ccw before the current edge.
{
    return Sym()->Onext();
}
/************* Access to data pointers *************************************/

inline Point2d* Edge::Org()
{
    return data;
}

inline Point2d* Edge::Dest()
{
    return Sym()->data;
}

inline const Point2d& Edge::Org2d() const
{
    return *data;
}

inline const Point2d& Edge::Dest2d() const
{
    return (num < 2) ? *((this + 2)->data) : *((this - 2)->data);
}

inline void Edge::EndPoints(Point2d* or, Point2d* de)
{
    data = or;
    Sym()->data = de;
}

/********************** Basic Topological Operators ***********************/

Edge* MakeEdge()
{
    QuadEdge *ql = new QuadEdge;
    return ql->e;
}

void Splice(Edge* a, Edge* b)
// This operator affects the two edge rings around the origins of a and b,
// and, independently, the two edge rings around the left faces of a and b.
// In each case, (i) if the two rings are distinct, Splice will combine
// them into one; (ii) if the two are the same ring, Splice will break it
// into two separate pieces.
// Thus, Splice can be used both to attach the two edges together, and
// to break them apart. See Guibas and Stolfi (1985) p.96 for more details
// and illustrations.
{
    Edge* alpha = a->Onext()->Rot();
    Edge* beta  = b->Onext()->Rot();
```

```
    Edge* t1 = b->Onext();
    Edge* t2 = a->Onext();
    Edge* t3 = beta->Onext();
    Edge* t4 = alpha->Onext();

    a->next = t1;
    b->next = t2;
    alpha->next = t3;
    beta->next = t4;
}

void DeleteEdge(Edge* e)
{
    Splice(e, e->Oprev());
    Splice(e->Sym(), e->Sym()->Oprev());
    delete e->Qedge();
}

/************* Topological Operations for Delaunay Diagrams *****************/

Subdivision::Subdivision(const Point2d& a, const Point2d& b, const Point2d& c)
// Initialize a subdivision to the triangle defined by the points a, b, c.
{
    Point2d *da, *db, *dc;
    da = new Point2d(a), db = new Point2d(b), dc = new Point2d(c);
    Edge* ea = MakeEdge();
    ea->EndPoints(da, db);
    Edge* eb = MakeEdge();
    Splice(ea->Sym(), eb);
    eb->EndPoints(db, dc);
    Edge* ec = MakeEdge();
    Splice(eb->Sym(), ec);
    ec->EndPoints(dc, da);
    Splice(ec->Sym(), ea);
    startingEdge = ea;
}

Edge* Connect(Edge* a, Edge* b)
// Add a new edge e connecting the destination of a to the
// origin of b, in such a way that all three have the same
// left face after the connection is complete.
// Additionally, the data pointers of the new edge are set.
{
    Edge* e = MakeEdge();
    Splice(e, a->Lnext());
    Splice(e->Sym(), b);
    e->EndPoints(a->Dest(), b->Org());
    return e;
}

void Swap(Edge* e)
// Essentially turns edge e counterclockwise inside its enclosing
// quadrilateral. The data pointers are modified accordingly.
```

```
{
    Edge* a = e->Oprev();
    Edge* b = e->Sym()->Oprev();
    Splice(e, a);
    Splice(e->Sym(), b);
    Splice(e, a->Lnext());
    Splice(e->Sym(), b->Lnext());
    e->EndPoints(a->Dest(), b->Dest());
}

/*************** Geometric Predicates for Delaunay Diagrams *****************/

inline Real TriArea(const Point2d& a, const Point2d& b, const Point2d& c)
// Returns twice the area of the oriented triangle (a, b, c), i.e., the
// area is positive if the triangle is oriented counterclockwise.
{
    return (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
}

int InCircle(const Point2d& a, const Point2d& b,
             const Point2d& c, const Point2d& d)
// Returns TRUE if the point d is inside the circle defined by the
// points a, b, c. See Guibas and Stolfi (1985) p.107.
{
    return (a.x*a.x + a.y*a.y) * TriArea(b, c, d) -
           (b.x*b.x + b.y*b.y) * TriArea(a, c, d) +
           (c.x*c.x + c.y*c.y) * TriArea(a, b, d) -
           (d.x*d.x + d.y*d.y) * TriArea(a, b, c) > 0;
}

int ccw(const Point2d& a, const Point2d& b, const Point2d& c)
// Returns TRUE if the points a, b, c are in a counterclockwise order
{
    return (TriArea(a, b, c) > 0);
}

int RightOf(const Point2d& x, Edge* e)
{
    return ccw(x, e->Dest2d(), e->Org2d());
}

int LeftOf(const Point2d& x, Edge* e)
{
    return ccw(x, e->Org2d(), e->Dest2d());
}

int OnEdge(const Point2d& x, Edge* e)
// A predicate that determines if the point x is on the edge e.
// The point is considered on if it is in the EPS-neighborhood
// of the edge.
{
    Real t1, t2, t3;
    t1 = (x - e->Org2d()).norm();
```

```
    t2 = (x - e->Dest2d()).norm();
    if (t1 < EPS || t2 < EPS)
        return TRUE;
    t3 = (e->Org2d() - e->Dest2d()).norm();
    if (t1 > t3 || t2 > t3)
        return FALSE;
    Line line(e->Org2d(), e->Dest2d());
    return (fabs(line.eval(x)) < EPS);
}

/************* An Incremental Algorithm for the Construction of *************/
/************************ Delaunay Diagrams *********************************/

Edge* Subdivision::Locate(const Point2d& x)
// Returns an edge e, s.t. either x is on e, or e is an edge of
// a triangle containing x. The search starts from startingEdge
// and proceeds in the general direction of x. Based on the
// pseudocode in Guibas and Stolfi (1985) p.121.
{
    Edge* e = startingEdge;

    while (TRUE) {
        if (x == e->Org2d() || x == e->Dest2d())
            return e;
        else if (RightOf(x, e))
                 e = e->Sym();
        else if (!RightOf(x, e->Onext()))
                 e = e->Onext();
        else if (!RightOf(x, e->Dprev()))
                 e = e->Dprev();
        else
            return e;
    }
}

void Subdivision::InsertSite(const Point2d& x)
// Inserts a new point into a subdivision representing a Delaunay
// triangulation, and fixes the affected edges so that the result
// is still a Delaunay triangulation. This is based on the
// pseudocode from Guibas and Stolfi (1985) p.120, with slight
// modifications and a bug fix.
{
    Edge* e = Locate(x);
    if ((x == e->Org2d()) || (x == e->Dest2d()))  // point is already in
        return;
    else if (OnEdge(x, e)) {
        e = e->Oprev();
        DeleteEdge(e->Onext());
    }

    // Connect the new point to the vertices of the containing
    // triangle (or quadrilateral, if the new point fell on an
    // existing edge.)
```

```
    Edge* base = MakeEdge();
    base->EndPoints(e->Org(), new Point2d(x));
    Splice(base, e);
    startingEdge = base;
    do {
        base = Connect(e, base->Sym());
        e = base->Oprev();
    } while (e->Lnext() != startingEdge);

    // Examine suspect edges to ensure that the Delaunay condition
    // is satisfied.
    do {
        Edge* t = e->Oprev();
        if (RightOf(t->Dest2d(), e) &&
            InCircle(e->Org2d(), t->Dest2d(), e->Dest2d(), x)) {
                Swap(e);
                e = e->Oprev();
        }
        else if (e->Onext() == startingEdge)  // no more suspect edges
            return;
        else  // pop a suspect edge
            e = e->Onext()->Lprev();
    } while (TRUE);
}

/*****************************************************************************/
```

◊ Bibliography ◊

(Baumgart 1975) G. Baumgart, B. A polyhedron representation for computer vision. In *1975 National Computer Conference*, volume 44 of *AFIPS Conference Proceedings*, pages 589–596, Arlington, Va, 1975. AFIPS Press.

(Bern and Eppstein 1992) Marshall Bern and David Eppstein. Mesh generation and optimal triangulation. In F. K. Hwang and D.-Z. Du, editors, *Computing in Euclidean Geometry*, pages 23–90. World Scientific, Singapore, 1992.

(Fortune 1992) Steven Fortune. Voronoi diagrams and Delaunay triangulations. In F. K. Hwang and D.-Z. Du, editors, *Computing in Euclidean Geometry*, pages 193–233. World Scientific, Singapore, 1992.

(Green and Sibson 1978) P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *Computer Journal*, 21(2):168–173, 1978.

(Guibas and Stolfi 1985) Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

(Guibas et al. 1990) L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Proc. 17th Int. Colloq. — Automata, Languages, and Programming*, volume 443 of *Springer-Verlag LNCS*, pages 414–431, Berlin, 1990. Springer-Verlag.

(Karasick et al. 1991) Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

(Preparata and Shamos 1985) Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.