# Contents

# 10

# Geometry Algorithms

We discuss convex hulls, triangulations, the verification of geometric structures, Delaunay triangulations and Delaunay diagrams, Voronoi diagrams, applications of Delaunay and Voronoi diagrams, geometric dictionaries, line segment intersection, polygons, and close with a glimpse at higher-dimensional computation geometry. For each problem we introduce the required mathematics and derive algorithms and their implementations. The books [Meh84b, Ede87, PS85, Mul94, Kle97, BY98, dBKOS97] provide a wider view of computational geometry.

The chapter uses results of all preceding chapters and is, in this sense, the culmination point of the book, e.g., we use lists and arrays from the basic data types, integers and rationals from the number types, dictionaries, maps and sorted sequences from the advanced data types, graphs and graph algorithms, embedded graphs, and the geometry kernels.

Computational geometry is a very rich area and LEDA certainly does not provide everything there is to it. Other good sources of geometric software are CGAL [CGA] and the LEDA extension packages [LEP].

## 10.1 Convex Hulls

The convex hull problem in the plane is one of the simplest geometric problems and hence a good starting point for our exploration of geometry algorithms. It will allow us to address five important themes in a simple setting:

- The *sweep paradigm*: In this paradigm the input points are first sorted according to the lexicographic order and then the desired geometric structure is constructed incrementally during a single sweep over the points. We will derive and implement a

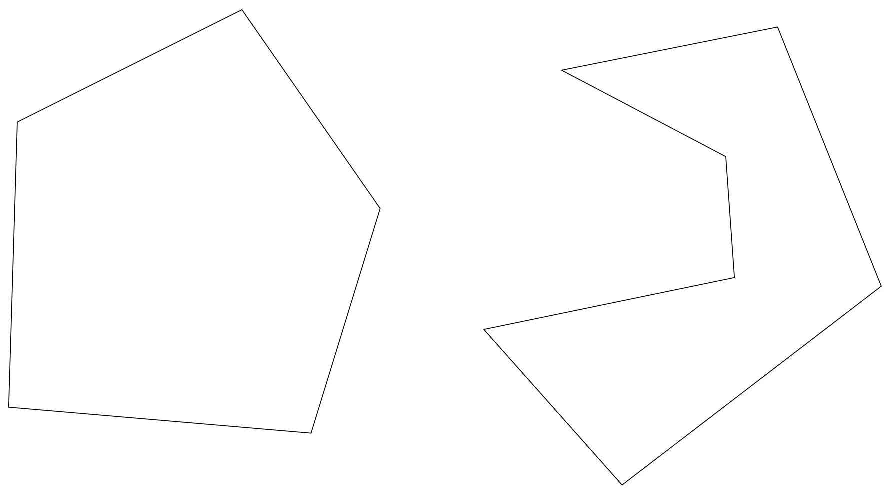

**Figure 10.1** A convex and a non-convex set.

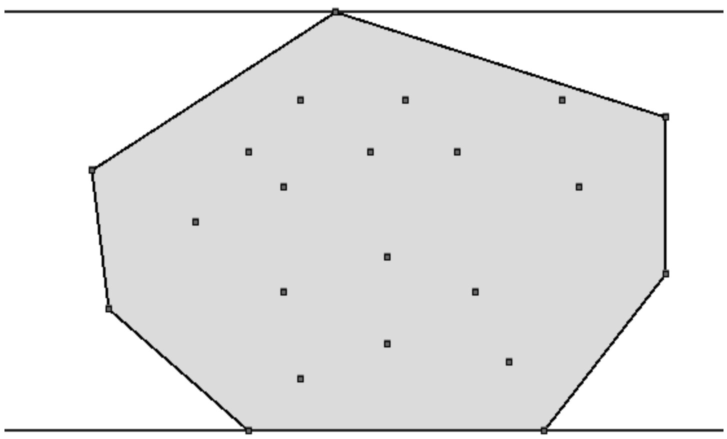

**Figure 10.2** A point set, its convex hull, and its width. The figure was generated with the xlman-demo voronoi_demo. The width of point sets is discussed in Section 10.1.3.

sweep algorithm for convex hulls. We will see more applications of the sweep paradigm in later sections.

- The *(randomized) incremental construction paradigm*: In this paradigm the input points are considered one by one in either arbitrary or random order and the desired geometric structure is constructed incrementally. We will derive and implement an incremental algorithm for convex hulls.
- The careful handling of *degeneracies*: The literature on computational geometry frequently makes the so-called *general position assumption* which states that only inputs are considered for which none of the geometric predicates required by the algorithm (recall that the evaluation of a geometric predicate calls for the evaluation of the sign of an expression) ever evaluates to zero. For example, the incremental

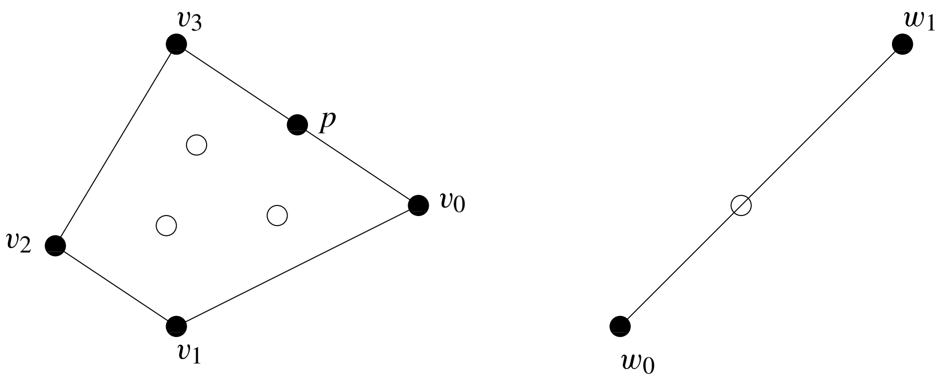


**Figure 10.3** Two point sets and their convex hulls. The hulls are represented as cyclic lists of points, namely $v_0$, $v_1$, $v_2$, $v_3$ for the example on the left and $w_0$, $w_1$ for the example on the right.

algorithm for convex hulls uses the orientation predicate and hence the general position assumption excludes all inputs containing three collinear points. Of course, we do not want to exclude any inputs and hence cannot make the general position assumption. Dropping the general position assumption typically requires a more careful formulation of the algorithms. The sweep as well as the incremental algorithm for convex hulls will work for all inputs. In fact, all algorithms in this chapter do.

- *Verification of geometric structures*: Geometric programs require checking. Although the convex hull problem is one of the simplest geometric problems, the programs derived in this section will be non-trivial. We will see how to partially check the output of convex hull programs in Section 10.3.
- The importance of *exact geometric primitives*: In the preceding chapter we introduced the rational geometry kernel; in this section we will profit from it.

A set $C$ is called *convex* if for any two points $p$ and $q$ in $C$ the entire line segment $pq$ is contained in $C$, see Figure 10.1. The *convex hull* conv $S$ of a set $S$ of points is the smallest (with respect to set inclusion) convex set containing $S$. A point $p \in S$ is called an *extreme point* of $S$ if there is a closed halfspace containing $S$ such that $p$ is the only point in $S$ that lies in the boundary of the halfspace. A point $p \in S$ is called a *weak extreme point* of $S$ if there is a closed halfspace containing $S$ such that $p$ lies in the boundary of the halfspace. Clearly, an extreme point is also a weak extreme point, but there may be weak extreme points that are not extreme points. The point $p$ in Figure 10.3 is an example.

From now on we restrict our discussion to the plane. If $S$ contains no three collinear points then every weak extreme point is also extreme, i.e., under the general position assumption there is no need to distinguish between weak extreme points and extreme points. We define the convex hull problem as the problem of computing the extreme points of a finite set of points as a cyclically ordered list of point, see Figure 10.3. The cyclic order is the clockwise order in which the extreme points appear on the hull.

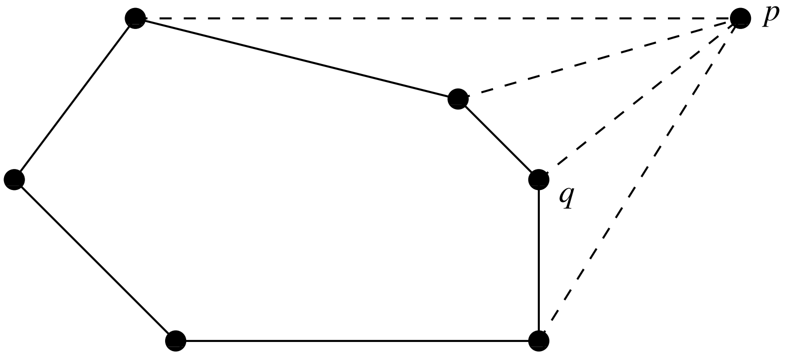


**Figure 10.4** Adding point $p$. We determine the two tangents from $p$ by a clockwise and counter-clockwise walk along the current hull starting at the most recently added point $q$.

The function

```
list<POINT> CONVEX_HULL(const list<POINT>& L);
```

computes the convex hull of the points in $L$ and returns its list of vertices. The cyclic order of the vertices in the result corresponds to the clockwise order of the vertices on the hull. The algorithm uses randomized incremental construction and its expected running time is $O(n \log n)$.

### 10.1.1 *The Sweep Algorithm*

The sweep algorithm for convex hulls consists of the following three steps:

- The input points are sorted in increasing lexicographic order.
- The convex hull is initialized with the two lexicographically smallest points in $L$.
- The remaining points are considered in increasing lexicographic order and the convex hull is updated for each point. Assume that $p$ is the next point to be considered and that we have already constructed the convex hull of the preceding points. The new hull can be obtained from the old hull by constructing the two tangents from $p$. The construction of the tangents is simple since $p$ is guaranteed to see the point $q$ added just before $p$. We only have to walk from $q$ in clockwise and counter-clockwise direction along the hull in order to determine the other endpoints of the tangents, see Figure 10.4.

We now turn this strategy into a program. We assume that the set $S$ is given as a list *L0* of points. We allow multiple occurrences of points. We follow the general outline above and proceed in three steps. We first make a local copy $L$ of *L0* and sort $L$. Next we initialize the list of hull vertices with the first two points (in the sorted version) of $L$, and finally, we add all other points of $L$. We call the resulting program CONVEX_HULL_S since it uses the sweep paradigm to compute convex hulls.

⟨*convex_hull.c*⟩≡
```
list<POINT>  CONVEX_HULL_S(const list<POINT>& L0)
{ list<POINT> CH;
  list<POINT> L = L0;
  L.sort();
  ⟨initialize hull with two points⟩
  ⟨add all other points⟩
  return CH;
}
```

We prepare for the sweep by sorting the points according to the lexicographic order. A point $p$ precedes a point $q$ in the *lexicographic order* if either its $x$-coordinate is smaller or the two $x$-coordinates are equal and its $y$-coordinate is smaller. The default ordering on points is the lexicographic ordering and hence *L.sort*( ) rearranges $L$ in the desired way.

We can now start building the hull. We begin with the first two points in $L$ and make them the vertices of the first hull. As said above we represent the hull as a linear list *CH* that contains the hull vertices in *clockwise* order. The list is to be interpreted as a cyclic list. We maintain an item *last_vertex* into the list; it contains the point added last.

⟨*initialize hull with two points*⟩≡
```
if ( L.empty() ) return CH;
POINT last_p;
CH.append(last_p = L.pop());
// remove duplicates of first point
while ( !L.empty() && last_p == L.head() ) L.pop();
if ( L.empty() ) return CH;
list_item last_vertex = CH.append(last_p = L.pop());
```

We process the remaining points. If the next point $p$ is equal to the last point added we do nothing. If the current hull consists of only two vertices and the new point $p$ is collinear with these vertices we replace the second vertex by $p$. Otherwise, we determine two items *up_item* and *down_item* in *CH* which correspond to the other endpoints of the two tangents starting at $p$. To determine *up_item* we scan the hull in counter-clockwise direction starting at *last_vertex*. If the point stored at the predecessor of *up_item*, the point stored at *up_item*, and $p$ do not form a right turn we move *up_item* to its predecessor vertex. We determine *down_item* by the symmetric procedure.

After having determined *up_item* and *down_item* we update the hull. We delete all items strictly between *up_item* and *down_item* and insert $p$ instead of them. Note that *up_item* and *down_item* are guaranteed to be different since $p$ sees at least one of the edges incident to the most recently added vertex.

⟨*add all other points*⟩≡
```
POINT p;
forall(p,L)
{ if ( p == last_p ) continue; // duplicate point
```

```
    last_p = p;
    if (CH.length() == 2 && collinear(CH.head(),CH.tail(),p))
      { CH[last_vertex] = p; continue; }
    // the interesting case
    // compute up_item
    list_item up_item = last_vertex;
    while (!right_turn(CH[CH.cyclic_pred(up_item)], CH[up_item], p))
    { up_item = CH.cyclic_pred(up_item); }
    // compute down_item
    list_item down_item = last_vertex;
    while (!left_turn(CH[CH.cyclic_succ(down_item)], CH[down_item], p))
    { down_item = CH.cyclic_succ(down_item); }
    // update hull
    while (down_item != CH.cyclic_succ(up_item))
    { CH.del_item(CH.cyclic_succ(up_item)); }
    last_vertex = CH.insert(p,up_item,after);
  }
```

The running time of the convex hull program is $O(n \log n)$. It takes time $O(n \log n)$ to sort the points lexicographically. After that everything is linear as the following amortization argument shows. Adding a point to the hull takes constant time plus time proportional to the number of points removed from the hull. Since any point can disappear from the hull at most once, the total time to add all points is linear. The running time of the algorithm is never better than $n \log n$ since it takes $\Theta(n \log n)$ time to sort the points. The sweep algorithm for convex hulls is due to Andrew ([And79]); it refines an earlier algorithm of Graham ([Gra72]).

The convex hull program makes use of the primitives provided by the geometry kernels. The rational kernel guarantees that all geometric primitives behave according to their mathematical specification and hence binding the program with the rational kernel will yield a correct executable. The program may behave incorrectly if bound with the floating point kernel. Consider the following example.

We compute the convex hull of the set $\{(-M+1, -M), (0,0), (M, M+1), (0,-2)\}$ for $M = 2^m$ and increasing values of $m$. All four points are extreme and hence the following program will print "everything went fine", when executed with the rational kernel.

⟨*convex hull and kernel*⟩≡

```
  for (int m = 20; m < 50; m++)
  {
    double M = ldexp(1.0,m);
    INT_TYPE IM(M);
    POINT p(-IM + 1, -IM) , q(0, 0), r(IM, IM + 1), s(0, -2);
    list<POINT> L;
    L.append(p); L.append(q); L.append(r); L.append(s);
    list<POINT> CH = CONVEX_HULL_S(L);
    if ( CH.length() != 4 )
```

```
    { cout << "\n\nlength = " << CH.length() << " for  m = " << m;
      return 0;
    }
  }
  cout << "\n\neverything went fine";
```

However, when executed with the floating point kernel the program will print

```
  length = 3 for  m = 27,
```

since the floating point kernel believes that the triple $(p, q, r)$ is collinear for $m \geq 27$.

### 10.1.2 *Incremental Construction*

We will next describe an alternative algorithm to compute convex hulls. The algorithm is based on the paradigm of *(randomized) incremental construction*. The algorithm has a worst case running time of $O(n^2)$, an average running time of $O(n \log n)$, and a best case running time of $O(n)$.

The algorithm starts by searching for three non-collinear points $a$, $b$, and $c$. If there are none, then all points are collinear and the vertices of the hull are simply the lexicographically smallest and largest point.

⟨*convex_hull.c*⟩+≡

```
  ⟨ch_edge_class⟩
  list<POINT>  CONVEX_HULL_IC(const list<POINT>& L)
  {
    if (L.length() < 2) return L;
    list<POINT> CH;
    POINT a = L.head(), b = L.tail();
    POINT c, p;
    if ( a == b ) { forall(p,L) if (p != a) { b = p; break; } }
    if ( a == b ) { // all points are equal
                    CH.append(a);
                    return CH;
                  }
    int orient;
    forall(c,L) if ( (orient = orientation(a,b,c)) != 0 ) break;
    if ( orient == 0 )
    { // all points are collinear
      forall(p,L) { if ( compare(p,a) < 0 ) a = p;
                    if ( compare(p,b) > 0 ) b = p;
                  }
      CH.append(a); CH.append(b);
      return CH;
    }
    // a, b, and c are not collinear
    if ( orient < 0 ) leda_swap(b,c);
    ⟨full-dimensional case: initialization⟩
```

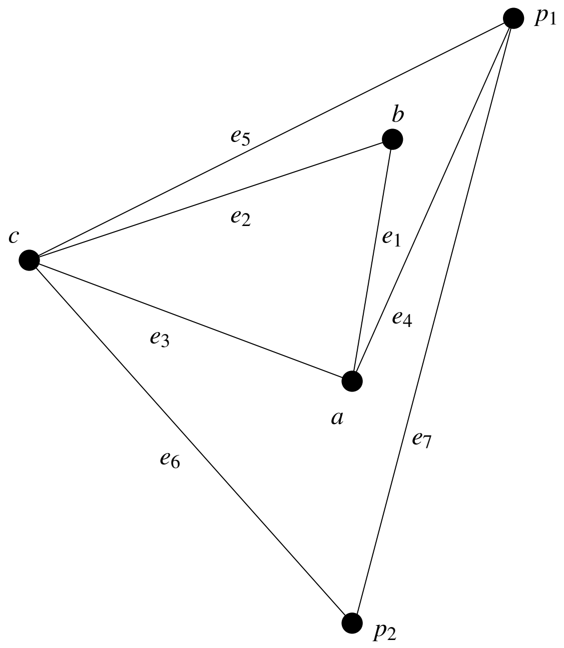


**Figure 10.5** The initial convex hull consists of the points $a$, $b$, and $c$. When point $p_1$ is added the edges $e_1$ and $e_2$ are deleted from the hull and the edges $e_4$ and $e_5$ are added, and when $p_2$ is added to the hull the edges $e_3$ and $e_4$ are deleted from the hull and the edges $e_6$ and $e_7$ are added. The boundary of the current hull consists of edges $e_7$, $e_5$, and $e_6$ in counter-clockwise order. Every edge ever deleted from the hull points to the two edges that replaced it, e.g., $e_3$ and $e_4$ point to $e_6$ and $e_7$.

```
  forall(p,L) { ⟨full-dimensional case: insertion of p⟩ }
  ⟨full-dimensional case: prepare result and clean-up⟩
  return CH;
}
```

We come to the interesting case that not all points in $L$ are collinear. We have already determined three non-collinear points $a$, $b$, and $c$. Their orientation is positive, i.e., the three points form a counter-clockwise oriented triangle.

The algorithm maintains the current hull as a cyclically linked list of edges and also keeps all edges that ever belonged to a hull. Every edge that is not on the current hull anymore points to the two edges that replaced it. More precisely, assume that $S$ is the set of points already seen and that $p$ is a point outside the current hull $CH(S)$. There is a chain $C$ of edges of the boundary of $CH(S)$ that do not belong to the boundary of $CH(S \cup p)$. The chain is replaced by the two tangents from $p$ to the previous hull. All edges in $C$ are made to point to the two new edges, see Figure 10.5.

We use a class *ch_edge* to represent convex hulls. Every edge stores its two endpoints, three links *succ*, *pred*, and *link* to other edges, and a boolean flag *outside*. We use *link* to collect all edges into a linear list in the order of their creation; every edge points to the edge created just before it and *last_edge* points to the edge created last. The only purpose of this linear list is to help in the destruction of edges.

The boolean flag *outside* indicates whether an edge belongs to the current hull or not. All edges in the current hull form a cyclic doubly linked list with *succ* pointing to the clockwise successor and *pred* pointing to the clockwise predecessor. All edges that do not belong to the current hull anymore use their *succ* and *pred* fields to point to the two replacement edges.

⟨*ch_edge_class*⟩≡

```
class ch_edge;
static ch_edge* last_edge = nil;
class ch_edge {
public:
  POINT    source, target;
  ch_edge* succ, pred, link;
  bool     outside;
  ch_edge(const POINT& a, const POINT& b) : source(a), target(b)
  { outside = true;
    link = last_edge;
    last_edge = this;
  }
~ch_edge() {}
};
```

In order to initialize the data structure we create the edges $(a,b)$, $(b,c)$ and $(c,a)$, store them in an array $T$, and turn them into a doubly-linked cyclic list. We initialize *last_edge* to *nil* before doing any of this, such that the list of all edges has the correct anchor.

⟨*full-dimensional case: initialization*⟩≡

```
last_edge = nil;
ch_edge* T[3];
T[0] = new ch_edge(a,b);
T[1] = new ch_edge(b,c);
T[2] = new ch_edge(c,a);
int i;
for(i = 0; i < 2; i++)  T[i]->succ = T[i+1];
T[2]->succ = T[0];
for(i = 1; i < 3; i++)  T[i]->pred = T[i-1];
T[0]->pred = T[2];
```

We are now ready to deal with the insertion of a point $p$. We proceed in two steps. We first determine whether $p$ is outside the current hull and then update the hull (if $p$ is outside).

In order to find out whether $p$ lies outside the current hull, we walk through the history of hulls. We first find out whether $p$ can see one of the edges of the initial triangle: $p$ lies outside the initial triangle if there is an edge $e$ of the initial triangle such that $p$ lies to the right of the edge.

More generally, $p$ is outside one of the intermediate hulls $CH(S)$ if there is an edge $e$ on

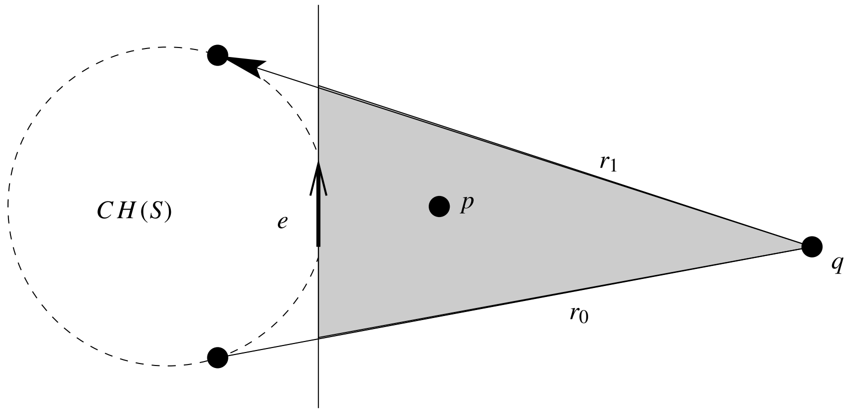


**Figure 10.6** $e$ is a (counter-clockwise) edge of the current hull and $p$ lies to the right of it; $e$ is replaced by $r_0$ and $r_1$ when the point $q$ is added. If $p$ lies neither to the right of $r_0$ nor to the right of $r_1$ then $p$ lies in the shaded region and hence in $CH(S \cup q)$.

its boundary such that $p$ lies to the right of the edge. If $e$ is an edge on the boundary of the current hull then $p$ lies outside the current hull. If $e$ is not an edge on the boundary of the current hull, let $r_0$ and $r_1$ be the two edges that replaced $e$ when $CH(S)$ was enlarged to $CH(S \cup q)$. $p$ is outside $CH(S \cup q)$ if it lies to the right of either $r_0$ or $r_1$, see Figure 10.6.

⟨*full-dimensional case: insertion of p*⟩≡

```
int i = 0;
while (i < 3 && !right_turn(T[i]->source,T[i]->target,p) ) i++;
if (i == 3) { // p inside initial triangle
              continue;
            }
ch_edge* e = T[i];
while (! e->outside)
{ ch_edge* r0 = e->pred;
  if ( right_turn(r0->source,r0->target,p) ) e = r0;
  else { ch_edge* r1 = e->succ;
         if ( right_turn(r1->source,r1->target,p) ) e = r1;
         else { e = nil; break; }
       }
}
if (e == nil) continue;  // p inside current hull
```

⟨*insertion of p: p is outside current hull*⟩

Assume now that $p$ lies outside the current hull and to the right of the counter-clockwise hull edge $e$. We determine all edges visible from $p$ by walking along the hull in both directions. This is exactly as in the previous algorithm. Let *low* be the first predecessor of $e$ that is not visible and let *high* be the first successor that is not visible.

We then add the new tangents between *low* and *high* and mark all edges that were deleted from the hull as inside and make the two new tangents the replacement edges of all deleted edges.

⟨*insertion of p: p is outside current hull*⟩≡

```
// compute "upper" tangent (p,high->source)
ch_edge* high = e->succ;
while (orientation(high->source,high->target,p) <= 0) high = high->succ;
// compute "lower" tangent (p,low->target)
ch_edge* low = e->pred;
while (orientation(low->source,low->target,p) <= 0) low = low->pred;
e = low->succ;  // e = successor of low edge
// add new tangents between low and high
ch_edge* e_l = new ch_edge(low->target,p);
ch_edge* e_h = new ch_edge(p,high->source);
e_h->succ = high;
e_l->pred = low;
high->pred = e_l->succ = e_h;
low->succ  = e_h->pred = e_l;
// mark edges between low and high as "inside"
// and define refinements
while (e != high)
{ ch_edge* q = e->succ;
  e->pred = e_l;
  e->succ = e_h;
  e->outside = false;
  e = q;
}
```

Having computed the hull we prepare the output and delete all edges. We prepare the output by running around the hull once and we clean up by deleting all edges.

⟨*full-dimensional case: prepare result and clean-up*⟩≡

```
ch_edge* l_edge = last_edge;
CH.append(l_edge->source);
for(ch_edge* e = l_edge->succ; e != l_edge; e = e->succ)
  CH.append(e->source);
// clean up
while (l_edge)
{ ch_edge* e = l_edge;
  l_edge = l_edge->link;
  delete e;
}
```

What is the running time of the incremental construction of convex hulls?

The worst case running time is $O(n^2)$ since the time to insert a point is $O(n)$. The time to insert a point is $O(n)$ since there are at most $2(k+1)$ edges after the insertion of $k$ points and since every edge is looked at at most once in the insertion process.

The best case running time is $O(n)$. An example for the best case is when the points $a$, $b$, and $c$ span the hull.

The average case running time is $O(n \log n)$ as we will show next. What are we averaging over? We consider a fixed but arbitrary set $S$ of $n$ points and average over the $n!$ possible insertion orders. The following theorem is a special case of the by now famous *probabilistic analysis of incremental constructions* started by Clarkson and Shor [CS89]. The books [Mul94, BY98, MR95, dBKOS97] contain detailed presentations of the method. The reader may skip the proof of Theorem 1. Why do we include a proof at all given the fact that the method is already well treated in textbooks? We give a proof because the cited references prove the theorem only for points in general position. We want to do without the general position assumption in this book.

**Theorem 1** *The average running time of the incremental construction method for convex hulls is* $O(n \log n)$.

*Proof* We assume for simplicity that the points in $S$ are pairwise distinct. The theorem is true without this assumption; however, the notation required in the proof is more clumsy.

The running time of the algorithm is linear iff all points in $S$ are collinear. So let us assume that $S$ contains three points that are not collinear. In this case we will first construct a triangle and then insert the remaining points. Let $p$ be one of the remaining points. When $p$ is inserted, we first determine the position of $p$ with respect to the initial triangle (time $O(1)$), then search for a hull edge $e$ visible by $p$, and finally update the hull. The time to update the hull is $O(1)$ plus some bounded amount of time for each edge that is removed from the hull. We conclude that the total time (= time summed over all insertions) spent outside the search for a visible hull edge is $O(n)$.

In the search for a visible hull edge we perform tests *rightturn*$(x, y, p)$ where $x$ and $y$ are previously inserted points. We call a test *successful* if it returns true and observe that in each iteration of the while-loop at most two rightturn tests are performed and that in all iterations except the last at least one rightturn test is successful. It therefore suffices to bound the number of successful rightturn tests.

For an ordered pair $(x, y)$ of distinct points in $S$ we use $K_{x,y}$ to denote the set of points $z$ in $S$ such that *rightturn*$(x, y, z)$ is true plus[1] the set of points on the line through $(x, y)$ but not between $x$ and $y$, see Figure 10.7. We use $k_{xy}$ to denote the cardinality of $K_{x,y}$, $F_k$ to denote the set of pairs $(x, y)$ with $k_{xy} = k$, $F_{\le k}$ to denote the set of pairs $(x, y)$ with $k_{xy} \le k$, and $f_k$ and $f_{\le k}$ to denote the cardinalities of $F_k$ and $F_{\le k}$, respectively. We have

**Lemma 1** *The average number $A$ of successful rightturn tests is bounded by* $\sum_{k \ge 1} 2 f_{\le k}/k^2$.

*Proof* Consider a pair $(x, y)$ with $k_{xy} = k$. If some point in $K_{x,y}$ is inserted before both $x$ and $y$ are inserted then $(x, y)$ is never constructed as a hull edge and hence no rightturn

[1] The set to be defined next is empty if $S$ is in general position. The probabilistic analysis of incremental constructions usually assumes general position. We do not want to assume it here and hence have to modify the proof somewhat.

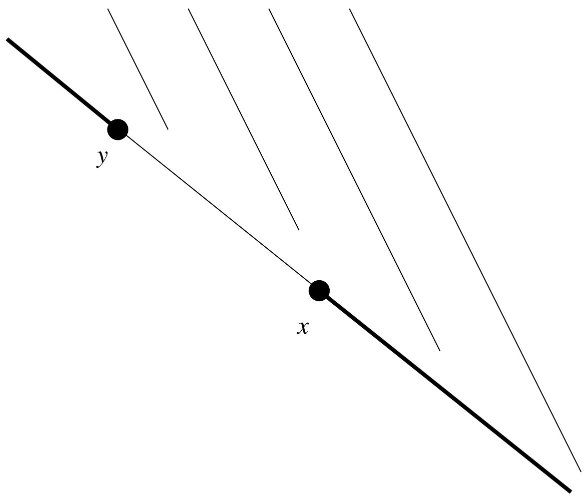


**Figure 10.7** $K_{x,y}$ consists of all points in the shaded region plus the two solid rays.

tests $(x, y, -)$ are performed. However, if $x$ and $y$ are inserted before all points in $K_{x,y}$ then up to $k$ successful rightturn tests $(x, y, z)$ are performed.
The probability that $x$ and $y$ are inserted before all points in $K_{x,y}$ is

$$2!k!/(k+2)!$$

since there are $(k+2)!$ permutations of $k+2$ points out of which $2!k!$ have $x$ and $y$ as their first two elements. Thus the expected number of successful rightturn tests $(x, y, z)$ is bounded by

$$2!k!/(k+2)! \cdot k = 2 \cdot k/(k+1)(k+2) < 2/(k+1).$$

The argument above applies to any pair $(x, y)$ and hence the average number of successful rightturn tests is bounded by

$$\sum_{k \geq 1} 2 f_k/(k+1).$$

We next write $f_k = f_{\leq k} - f_{\leq k-1}$ and obtain

$$\begin{aligned} A &\leq \sum_{k\geq 1} 2(f_{\leq k} - f_{\leq k-1})/(k+1) = \sum_{k\geq 1} 2 f_{\leq k}(1/(k+1) - 1/(k+2)) \\ &= \sum_{k \geq 1} 2 f_{\leq k}/((k+1)(k+2)). \end{aligned}$$

□

It remains to bound $f_{\leq k}$. We use random sampling to derive a bound.

**Lemma 2** *$f_{\leq k} \leq 2e^2 n \cdot k$ for all $k$, $1 \leq k \leq n$.*

*Proof* There are only $n^2$ pairs of points of $S$ and hence we always have $f_{\leq k} \leq n^2$. Thus, the claim is certainly true for $n \leq 10$ or $k \geq n/4$.
So assume that $n \geq 10$ and $k \leq n/4$ and let $R$ be a random subset of $S$ of size $r$. We will

fix $r$ later. Clearly, the convex hull of $R$ consists of at most $r$ edges. On the other hand, if for some $(x, y) \in F_{\leq k}$, $x$ and $y$ are in $R$ but none of the points in $K_{x,y}$ is in $R$, then $(x, y)$ will be an edge of the convex hull of $R$. The probability of this event is

$$\frac{\binom{n-i-2}{r-2}}{\binom{n}{r}} \geq \frac{\binom{n-k-2}{r-2}}{\binom{n}{r}},$$

where $i = k_{x,y}$. Observe that the event occurs if $x$ and $y$ are chosen and the remaining $r-2$ points in $R$ are chosen from $S \setminus \{x, y\} \setminus K_{x,y}$. The expected number of edges of the convex hull of $R$ is therefore at least

$$f_{\leq k} \cdot \frac{\binom{n-k-2}{r-2}}{\binom{n}{r}}.$$

Since the number of edges is at most $r$ we have

$$f_{\leq k} \cdot \binom{n-k-2}{r-2} \Big/ \binom{n}{r} \leq r$$

or

$$f_{\leq k} \leq r \cdot \binom{n}{r} \Big/ \binom{n-k-2}{r-2} = r \cdot \frac{n(n-1)}{r(r-1)} \cdot \frac{[n-2]_{r-2}}{[n-k-2]_{r-2}},$$

where $[n]_i = n(n-1)\cdots(n-i+1)$. Next observe that

$$\begin{aligned} \frac{[n-2]_{r-2}}{[n-k-2]_{r-2}} &\leq \frac{[n]_r}{[n-k]_r} = \prod_{i=0}^{r-1} \frac{n-i}{n-k-i} = \prod_{i=0}^{r-1} \left(1 + \frac{k}{n-k-i}\right) \\ &= \exp\left(\sum_{i=0}^{r-1} \ln(1 + k/(n-k-i))\right) \leq \exp\left(rk/(n-k-r)\right), \end{aligned}$$

where the last inequality follows from $\ln(1+x) \leq x$ for $x \geq 0$ and the fact that $k/(n-k-i) \leq k/(n-k-r)$ for $0 \leq i \leq r-1$. Setting $r = n/(2k)$ and using the fact that $n-k-r \geq n/4$ for $k \leq n/4$ and $n \geq 10$, we obtain

$$f_{\leq k} \leq e^2 n^2/r = 2e^2 nk.$$

□

Putting our two lemmas together completes the proof of Theorem 1

$$A \leq 4e^2 \sum_{k \geq 1} nk/k^2 = O(n \log n).$$

□

There are two important situations when the assumptions of the theorem above are satisfied:

- When the points in $S$ are generated according to a probability distribution for points in the plane.
- When the points are randomly permuted before the incremental construction process is started. We then speak about a *randomized incremental construction*.

CONVEX_HULL_RIC realizes the randomized incremental construction of convex hulls.

⟨*convex_hull.c*⟩+≡

```
list<POINT> CONVEX_HULL_RIC(const list<POINT>& L)
{ list<POINT> L1 = L;
  L1.permute();
  return CONVEX_HULL_IC(L1);
}
list<POINT> CONVEX_HULL(const list<POINT>& L)
{ return CONVEX_HULL_RIC(L); }
```

It is important to understand the difference between _IC and _RIC. The former is a *deterministic* procedure whose average running time is $O(n \log n)$ if the assumptions of Theorem 1 are satisfied. The latter is a randomized algorithm whose expected running time for any input is $O(n \log n)$. Table 10.1 shows the difference. We generated a list $L$ of $n$ random points for each of three distributions: random points in the unit square, random points in the unit disk, and random points close to the boundary of the unit circle. We also generated a second input set *LS* by sorting $L$ lexicographically. On the random inputs _IC does slightly better than _RIC because the latter does something that is unnecessary for random inputs: it randomly permutes an input that is already random. However, for the sorted inputs the situation is completely different. _RIC behaves about the same as for random inputs. However, _IC behaves much worse. For the points on the circle the behavior seems to be quadratic and for the points in the square and the disk the behavior seems to be $n^{\delta}$ for some $\delta > 1$. For this reason RIC is to be preferred over IC.

We next compare the sweep line algorithm with the randomized incremental construction algorithm. Table 10.2 shows the results. Observe that we use much larger inputs sizes for this table. The randomized incremental algorithm is faster than the sweep algorithm for inputs with only few hull vertices and is somewhat slower for points on the unit circle. Observe that the proof of Theorem 1 implies that the running time of randomized incremental construction is $o(n \log n)$ if a random subset of the input points has a small convex hull.

There are many more convex hull algorithms than sweep and (randomized) incremental construction. Schirra [Sch98] discusses implementations.

### 10.1.3 *The Width of a Point Set*

The *width* of a point set $L$ is the minimal width of a stripe containing all points in $L$. A stripe is the region of the plane between two parallel lines. Minimum width stripes are illustrated in the xlman-demo voronoi-demo, see Figure 10.2. The function

```
RAT_TYPE WIDTH(const list<POINT>& L, LINE& l1, LINE& l2)
```

assumes that $L$ is non-empty and returns the square[2] of the minimum width stripe containing $L$ and the boundaries of the stripe.

We show how to compute the minimum width stripe by the so-called *rotating caliber method*. We start with a partial characterization of the minimum width stripe.

[2] We return the square of the width instead of the width because this choice avoids the use of square roots.

| | | | | IC | | RIC | |
|---|---|---|---|---|---|---|---|
| K | n | Gen | V | Random | Sorted | Random | Sorted |
| S | 4000 | 0.29 | 18 | 0.09 | 0.27 | 0.11 | 0.13 |
| S | 8000 | 0.64 | 23 | 0.16 | 0.76 | 0.22 | 0.21 |
| S | 16000 | 1.34 | 29 | 0.33 | 2.53 | 0.42 | 0.41 |
| D | 4000 | 0.27 | 59 | 0.1 | 0.45 | 0.11 | 0.1 |
| D | 8000 | 0.59 | 66 | 0.17 | 1.26 | 0.23 | 0.2 |
| D | 16000 | 1.25 | 87 | 0.43 | 3.48 | 0.5 | 0.41 |
| C | 4000 | 9.32 | 4000 | 0.32 | 15.57 | 0.34 | 0.37 |
| C | 8000 | 18.87 | 7995 | 0.7 | 65.93 | 0.75 | 0.71 |
| C | 16000 | 37.62 | 1.599e+04 | 1.47 | 253.4 | 1.53 | 1.57 |

**Table 10.1** A comparison of incremental and randomized incremental construction: We generated $n$ points according to one of three distributions, either points with random integer coordinates in $[-R\,..\,R]$, or random points with integer coordinates in the disc with radius $R$ centered at the origin, or random points with integer coordinates that lie approximately on the circle with radius $R$ centered at the origin. We used $R = 16000$. The columns show from left to right the kind of the point set (S for points in a square, D for points in the disc, and C for points on a circle), the number $n$ of points, the time to generate the $n$ points, the number of vertices of the hull, the running time of the incremental algorithm (_IC), and the running time of the randomized incremental algorithm (_RIC). For both algorithms the first column gives the time for random inputs and the second column gives the time for lexicographically sorted inputs. Observe the bad behavior of _IC on sorted inputs. Also observe that the time to compute the hull is usually smaller than the time to generate the points.

**Lemma 3** *Let $S$ be a minimum width stripe containing $L$. Then one of the boundaries contains an edge of the convex hull of $L$ and the other boundary contains at least one vertex of the convex hull of $L$.*

*Proof* Clearly, both boundaries of $S$ must contain at least one vertex of the convex hull of $S$. Assume that neither boundary contains an edge of the convex hull and let $p$ and $q$ be the two vertices of the convex hull of $L$ that are contained in the boundary of $S$. Since the boundary of $L$ contains no edge of the convex hull we can rotate both lines around $p$ and $q$, respectively. Let $\alpha$ be the acute angle between the segment $pq$ and the boundary of $S$ incident to $p$, see Figure 10.8. Then

$$width(S) = |pq| \cdot \sin\alpha$$

and hence the width decreases when $\alpha$ is decreased. □

| | | | | Sweep | | RIC | |
|---|---|---|---|---|---|---|---|
| K | n | Gen | V | Random | Sorted | Random | Sorted |
| S | 20000 | 1.72 | 25 | 1.68 | 1.54 | 0.55 | 0.55 |
| S | 40000 | 3.77 | 29 | 3.6 | 3.26 | 1.26 | 1.43 |
| S | 80000 | 7.92 | 31 | 7.72 | 6.98 | 2.06 | 2.07 |
| D | 20000 | 1.62 | 106 | 1.75 | 1.59 | 0.55 | 0.56 |
| D | 40000 | 3.49 | 109 | 3.76 | 3.33 | 1.17 | 1.25 |
| D | 80000 | 7.32 | 152 | 8 | 7.02 | 2.42 | 2.58 |
| C | 20000 | 47 | 1.999e+04 | 1.82 | 1.67 | 2.13 | 2.12 |
| C | 40000 | 94.68 | 3.994e+04 | 3.96 | 3.57 | 4.46 | 4.41 |
| C | 80000 | 188.8 | 7.979e+04 | 8.6 | 7.78 | 10.31 | 10.04 |

**Table 10.2** The running times of the sweep algorithm and the randomized incremental construction algorithm for convex hulls. The meaning of the columns is the same as for Table 10.1.

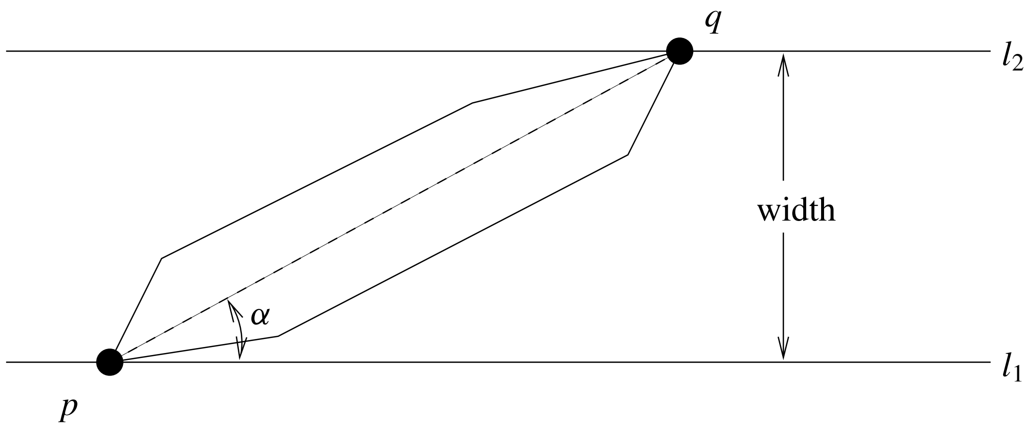


**Figure 10.8** The stripe $S$ with boundaries $l_1$ and $l_2$ contains all points of $L$, but neither boundary contains an edge of the convex hull of $L$. Rotating its boundaries decreases the width of the stripe.

We conclude from the lemma above that the minimum width stripe is defined by an edge of the convex hull and the vertex of maximum distance from the line supporting this edge. The next lemma constrains the part of the convex hull where this vertex of maximal distance may lie.

**Lemma 4** *Let $v_0, v_1, \ldots, v_{k-1}$ be the vertices of the convex hull of $L$, let $l = l(v_{k-1}, v_0)$ be the line passing through $v_{k-1}$ and $v_0$, and let $v_m$ be the vertex of maximal distance from $l$.*

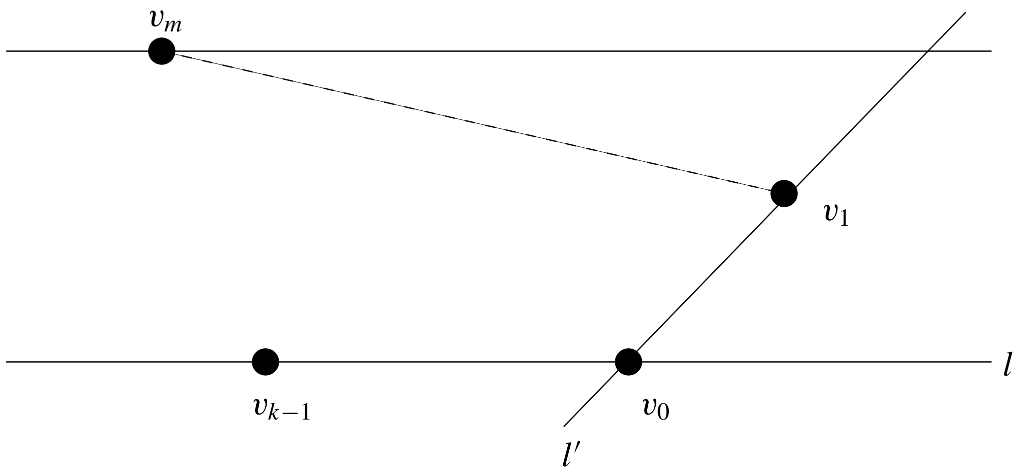


**Figure 10.9** Illustration of the proof of Lemma 4.

*Let* $l' = l(v_0, v_1)$*, let* $v_{m'}$ *be the vertex of maximal distance from* $l'$*. Then* $m \le m' \le k-1$*. Also* $m'$ *is minimal such that* $v_{m'+1}$ *has smaller distance to* $l'$ *than* $v_{m'}$*.*

*Proof* Consider Figure 10.9. All vertices $v_i$ with $1 \le i \le m$ are contained in the triangle with corners $v_1$, $v_m$, and the intersection between $l'$ and the line parallel to $l$ through $v_m$. Any point in this triangle has smaller distance to $l'$ than $v_m$. Thus $m \le m' \le k-1$.

For the second claim consider the distance between $l'$ and $v_i$ as a function of $i$ and as $i$ ranges from 1 to $k-1$. It follows from convexity that this function is first strictly increasing then reaches its maximum for either one or two vertices and is then again strictly decreasing. □

It is easy to derive an algorithm from the preceding lemma. We determine for each hull edge $pq$ the vertex $m$ of maximal distance from the line $l(p,q)$. We initialize $p$ and $q$ to the first two hull vertices and find $m$ by a search over all vertices. We then scan once around the convex hull of $L$ in order to check all other edges.

We maintain the square of the width of the currently best stripe in *min_sqr_width* and the boundaries of the stripe in *l1* and *l2*.

⟨*width.c*⟩≡

```
RAT_TYPE WIDTH(const list<POINT>& L, LINE& l1, LINE& l2)
{
  if ( L.empty() )
    error_handler(1,"WIDTH applies only to non-empty sets");
  list<POINT> CH = CONVEX_HULL(L);
  if ( CH.length() == 1 )
  { l1 = l2 = LINE(L.head(), VECTOR(INT_TYPE(1),INT_TYPE(1)));
    return 0;
  }
  if ( CH.length() == 2 )
```

```
{ l1 = l2 = LINE(CH.head(), CH.tail()); return 0; }
list_item p_it = CH.first();
list_item q_it = CH.cyclic_succ(p_it);
list_item m_it = q_it;
list_item it;
LINE l(CH[p_it],CH[q_it]);
RAT_TYPE min_sqr_width = 0; RAT_TYPE sqr_dist;
// find vertex with maximal distance from l
forall_items(it,CH)
{ if ( (sqr_dist = l.sqr_dist(CH[it])) > min_sqr_width )
  { min_sqr_width = sqr_dist;
    m_it = it;
  }
}
l1 = l; l2 = LINE(CH[m_it], CH[q_it] - CH[p_it]);
⟨rotate caliber around CH⟩
return min_sqr_width;
}
```

Let $r$ be the successor vertex of $q$. We want to determine the vertex $m'$ with maximal distance from $l' = l(q, r)$. The last sentence of the lemma above implies that $m'$ is the closest successor of $m$ (inclusive) such that the successor of $m'$ has smaller distance to $l'$ than $m'$.

*⟨rotate caliber around CH⟩*≡

```
do  // move caliber to next edge
{
  list_item r_it = CH.cyclic_succ(q_it);
  LINE l(CH[q_it],CH[r_it]);
  RAT_TYPE cur_sqr_dist = l.sqr_dist(CH[m_it]);
  list_item new_m_it = m_it;
  it = CH.cyclic_succ(m_it);
  while ( (sqr_dist = l.sqr_dist(CH[it])) >= cur_sqr_dist )
  { new_m_it = it; it = CH.cyclic_succ(it);
    cur_sqr_dist = sqr_dist;
  }
  if ( cur_sqr_dist < min_sqr_width )
  { min_sqr_width = cur_sqr_dist;
    l1 = l; l2 = LINE(CH[new_m_it], CH[r_it] - CH[q_it]);
  }
  p_it = q_it; q_it = r_it; m_it = new_m_it;
} while ( p_it != CH.first() );
```

The running time of the width computation is the time to compute the convex hull plus an amount of time that is linear in the number of vertices of the convex hull. It takes linear time to compute the vertex of maximal distance from the first hull edge and it takes linear time to compute the vertex of maximal distance for all other edges. The latter follows from the observation that both the edge and the vertex of maximal distance "travel around the convex hull once".

***Exercises for 10.1***

1 Design an example where the running time of CONVEX_HULL_IC is quadratic.
2 Design an example where the running time of CONVEX_HULL_IC is linear.
3 Redo the proof of Theorem 1 under the assumption that the expected number of hull edges in the convex hull of $r$ random points is $r^{1-\delta}$ for some $\delta > 0$.
4 Modify either convex hull algorithm such that it returns all points that lie on the boundary of the convex hull.
5 Let $P$ and $Q$ be two disjoint convex polygons given by their cyclic list of vertices. Write a program that computes the common tangents of $P$ and $Q$.
6 Use the solution of the previous exercise to compute the convex hull by divide-and-conquer. Sort the points lexicographically and split them into two halves. Compute the hull of both halves recursively. Merge the two hulls by constructing the common tangents.
7 Generate $n$ random points in the unit square and compute their convex hull. Do so for different values of $n$ and derive a conjecture concerning the expected number of extreme points in a set of $n$ random points. Try to prove your conjecture or do a literature search to find out what is known about the problem. Do the same for random points in the unit disk.

## 10.2 Triangulations

A *triangulation* of $S$ is a partition of the convex hull of $S$ into triangles. This assumes that not all points of $S$ are collinear. Each triangle in the partition has three points of $S$ as its vertices and any two triangles in the partition are either disjoint, or share a vertex, or an edge and two vertices. The union of all triangles is the convex hull of $S$, see Figure 10.10 for two examples. What is a triangulation of conv $S$ if all points of $S$ are collinear? It is simply a partition of conv $S$ into line segments[3], see Figures 10.10 and 10.11.

Triangulations are a versatile data structure. We will use them for point location queries, nearest neighbor queries, and range queries in Section 10.6 and describe their use in *interpolation* now. Assume that we are given the values of some function $f$ at some finite set $S$ of points and want to interpolate $f$ for all points in the convex hull of $S$. Triangulations offer an elegant way to approach this problem. We compute a triangulation $T$ of $S$ and lift it to three-dimensional space. More precisely, for every triangle $(p, q, r)$ of $T$ we define

[3] More generally, if $S$ has affine dimension $d$ then a triangulation of $S$ is a partition of conv $S$ into $d$-dimensional simplices. A $d$-dimensional *simplex* is the convex hull of $d + 1$ affinely independent points. Thus, triangles are two-dimensional simplices and line segments are one-dimensional simplices and hence a triangulation of a one-dimensional set $S$ is a partition of its convex hull into line segments, a triangulation of a two-dimensional set is a partition of its convex hull into triangles, and a triangulation of a three-dimensional set is the partition of its convex hull into tetrahedra.

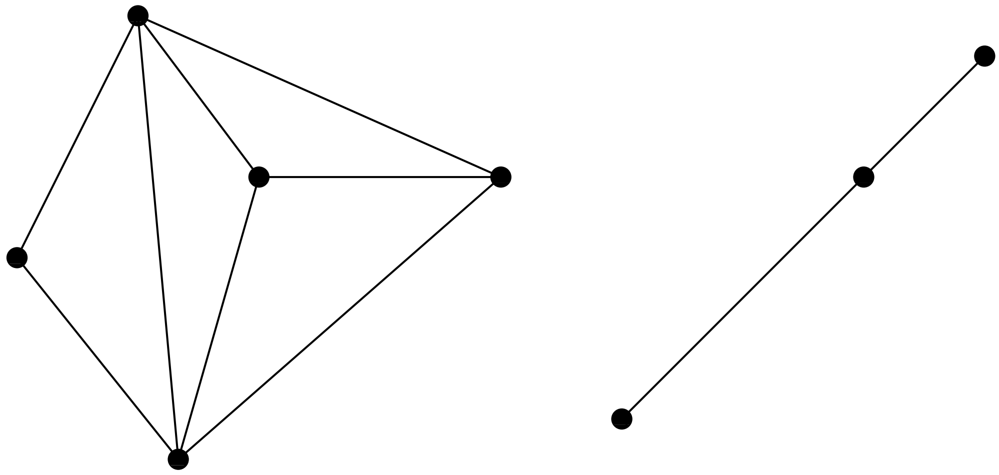

**Figure 10.10** A triangulation of a two-dimensional and of a one-dimensional point set.

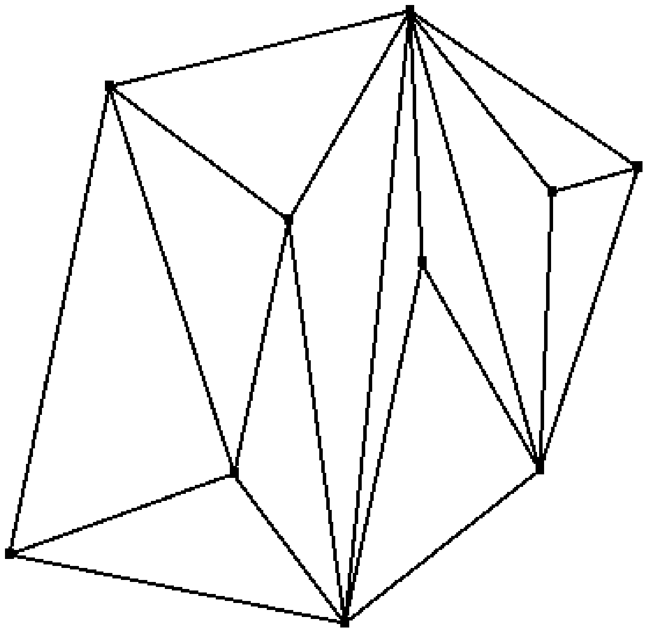

**Figure 10.11** A triangulation computed by the function TRIANGULATE_POINTS discussed in this section.

a triangle $((p, f(p)), (q, f(q)), (r, f(r)))$ in three-space, see Figure 10.12. In this way we obtain a surface in three-space. In order to determine the interpolating value at a point $x \in \text{conv}\, S$ we determine the height of the interpolating surface above $x$ and return it. This requires us to find the triangle of $T$ containing $x$ (a point location query) and to determine the height at $x$ by linear interpolation from the height at the vertices of the triangle containing $x$. Assume that $x$ lies in the triangle with vertices $p$, $q$, and $r$. We write $x$ as a convex

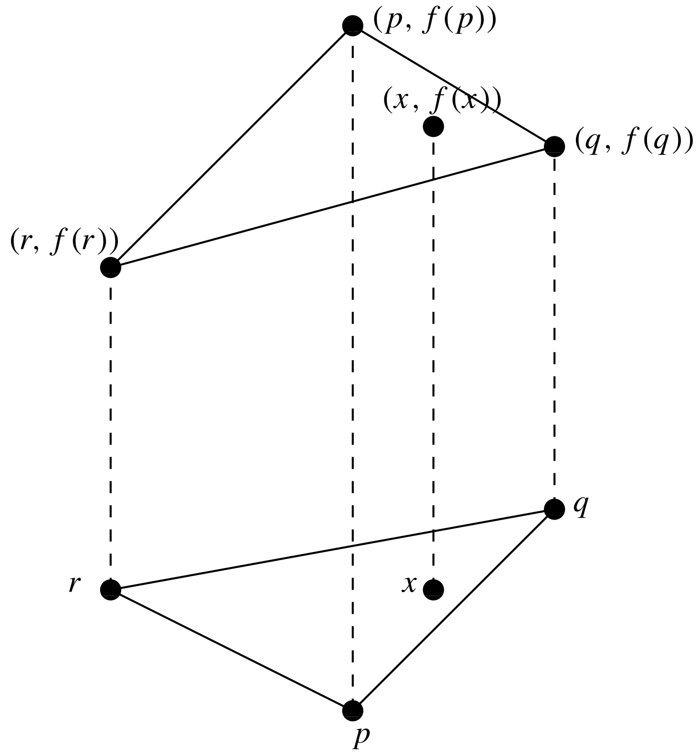


**Figure 10.12** A triangle in the plane and its lifting to three-space.

combination of $p$, $q$, and $r$, i.e.,

$$x = c_p p + c_q q + c_r r,$$

where $c_p + c_q + c_r = 1$ and compute $f(x)$ as

$$f(x) = c_p f(p) + c_q f(q) + c_r f(r).$$

The coefficients $c_p$, $c_q$, and $c_r$ are called the *barycentric coordinates* of $x$ with respect to the triangle $(p, q, r)$.

We next discuss how to represent triangulations. We represent triangulations as straight line embedded plane maps; embedded graphs are the subject of Chapter 8 and we recommend that you read the first four sections of that chapter before proceeding. Let $T$ be a triangulation of a set $S$ of points. We use a graph $G$ of type *GRAPH<POINT, int>* to represent $T$; $G$ has the following properties, see Figure 10.14:

- The nodes of $G$ are in one-to-one correspondence to the points in $S$. For a node $v$ of $G$ the point in $S$ corresponding to it is stored as $G[v]$.
- $G$ is a directed graph whose edges will be called *darts*. We use the word dart instead of edge in order to distinguish the edges of the representing graph from the edges of the represented geometric object. The darts of $G$ come in pairs. For every dart $e = (v, w)$ of $G$ the reversed dart $e^R = (w, v)$ is also a dart of $G$. Moreover, the member function *reversal* maps each dart to its reversal, i.e., *G.reversal*$(e) = e^R$ and

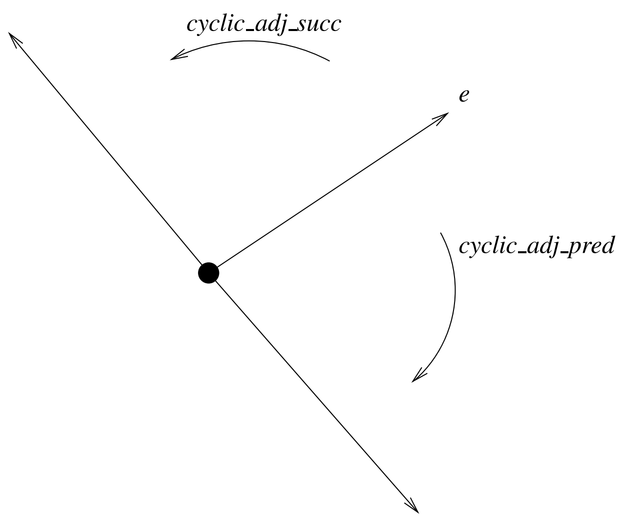


**Figure 10.13** The relationship between the cyclic ordering of the adjacency list $A(v)$ of a node $v$ and the counter-clockwise ordering of the edges incident to $G[v]$.

$G.reversal(e^R) = e$. We call a pair consisting of a dart and its reversal a uedge (= undirected edge). The uedges of $G$ correspond to the edges of $T$ and a dart $(v, w)$ of $G$ corresponds to the oriented edge $(G[v], G[w])$ of $T$.

- For each node $v$ of $G$ the list $A(v)$ of edges out of $v$ is ordered cyclically. For an edge $e$ with source $v$ the functions

  ```
  G.cyclic_adj_succ(e);
  G.cyclic_adj_pred(e);
  ```

  return the cyclic successor and the cyclic predecessor of $e$ in $A(v)$. The cyclic ordering of the edges in $A(v)$ agrees with the counter-clockwise ordering of the edges incident to $G[v]$ in the triangulation, i.e., *G.cyclic_adj_succ*$(e)$ is the next dart out of $v$ in counter-clockwise direction and *G.cyclic_adj_pred*$(e)$ is the next dart out of $v$ in clockwise direction, see Figure 10.13.

- The preceding items guarantee that the faces of the triangulation correspond to the face cycles of $G$. For each counter-clockwise triangle $(G[u], G[v], G[w])$ of the triangulation the edges $(u, v)$, $(v, w)$, $(w, u)$ form a face cycle of $G$. There is also a face cycle corresponding to the unbounded face of $T$. As a face cycle is traversed the face lies to the left of the face cycle. The functions

  ```
  G.face_cycle_succ(e);
  G.face_cycle_pred(e);
  ```

  support the convenient traversal of the face cycles of a map. They give the successor and predecessor of $e$ in the face cycle containing $e$, respectively. The face cycle successor is the cyclic adjacency predecessor of the reversal of $e$, see Figure 8.10.

- Each dart has an integer label (available as $G[e]$) that gives information about the dart. The labels come from the enumeration type

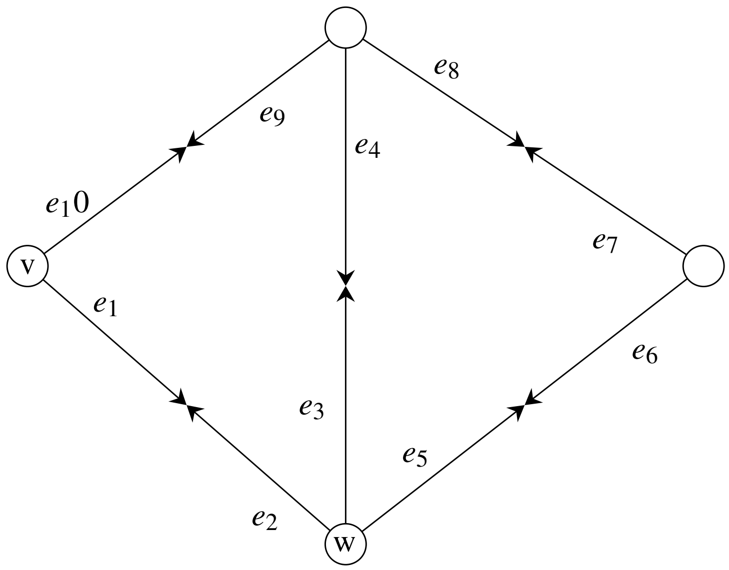


**Figure 10.14** A graph $G$ representing a triangulation. For each edge of the triangulation there are two darts in $G$, e.g., the edge $G[v]G[w]$ is represented by the darts $e_1 = (v, w)$ and $e_2 = (w, v)$. We have $G.reversal(e_1) = e_2$ and $G.reversal(e_2) = e_1$. For each dart its name is shown near the source of the dart and to the left of the dart. The list $A(w)$ of edges out of $w$ is a cyclic shift (it is not specified which) of $(e_5, e_3, e_2)$. The two triangles correspond to the face cycles $(e_1, e_3, e_9)$ and $(e_5, e_7, e_4)$. The unbounded face corresponds to the face cycle $(e_6, e_2, e_{10}, e_8)$.

```
enum delaunay_edge_info{ DIAGRAM_EDGE = 0, DIAGRAM_DART = 0,
                         NON_DIAGRAM_EDGE = 1, NON_DIAGRAM_DART = 1,
                         HULL_EDGE = 2, HULL_DART = 2
                       };
```

defined in <*LEDA/geo_global_enums.h*>. We discuss them in Section 10.4.

A dart is called a *hull dart* if the unbounded face of $G$ lies to its left. If *hull_dart* is any hull dart, the following lines of code traverse all hull darts.

```
edge e = hull_dart;
do { e = G.face_cycle_succ(e); } while (e != hull_dart);
```

We next extend the hull program of the preceding section to a triangulation program. This algorithm was first described in [Meh84a]. Again, we start by sorting the points lexicographically. Then we set up the triangulation of the first two points and finally add point by point to the triangulation.

⟨*triangulation.c*⟩≡

```
inline int left_bend(const POINT& p, const GRAPH<POINT,int>& G,
                     const edge& e)
{ return (orientation(p,G[source(e)],G[target(e)]) > 0); }
edge TRIANGULATE_POINTS(const list<POINT>& L0, GRAPH<POINT,int>& G)
{
  G.clear();
```

```
    if (L0.empty()) return nil;
    list<POINT> L = L0;
    L.sort();
    if ( L.empty() ) return nil;
    // initialize G with a single edge starting at the first point
    POINT last_p = L.pop();            // last visited point
    node  last_v = G.new_node(last_p); // last inserted node
    while (!L.empty() && last_p == L.head()) L.pop();
    if (!L.empty())
    { last_p = L.pop();
      node v = G.new_node(last_p);
      edge x = G.new_edge(last_v,v,0);
      edge y = G.new_edge(v,last_v,0);
      G.set_reversal(x,y);
      last_v = v;
    }
    ⟨triangulate points: scan remaining points⟩
  }
```

In order to facilitate the addition of points we maintain the dart *e_last*; it is the hull dart that leaves the most recently added vertex. Let $p$ be the point to be added and let *e_up* and *e_low* be hull darts such that exactly the hull vertices between the target of *e_up* and the source of *e_low* are visible from $p$, see Figure 10.15. All edges $e$ between *e_up* and *e_low* are such that $p$, the source of $e$, and the target of $e$ form a left turn, but *e_up* and *e_low* do not have this property. Moreover, *e_up* is a proper face cycle predecessor of *e_last*, and *e_low* is a face cycle successor of *e_last*. Thus it is easy to determine *e_up* and *e_low*. For example, the former is the first proper face cycle predecessor $e$ of *e_last* such that $p$, the source of $e$, and target of $e$ do not form a left turn.

Having determined *e_up* we walk to *e_low* and extend the triangulation by adding edges between $v$, where $v$ is a new node corresponding to point $p$, and the hull vertices visible from $p$. We must be careful to add the new edges in a way that reflects the triangulation. We iterate over the hull darts between *e_up* inclusive and *e_low* exclusive, starting at *e_up* and walking towards *e_low*. Consider any such $e$ and let *e_succ* be its face cycle successor. We add the dart $(source(e\_succ), v)$ after *e_succ* to $A(source(e\_succ))$ and we append the dart $(v, source(e\_succ))$ to $A(v)$. Observe that this way of adding darts builds $A(v)$ in counter-clockwise order and adds the dart $(source(e\_succ), v)$ at the proper position to $A(source(e\_succ))$.

The update step just described works correctly even if the new point is collinear with all preceding points. In this situation only a line segment is added to the triangulation.

⟨*triangulate points: scan remaining points*⟩≡

```
  POINT p;
  forall(p,L)
  { if (p == last_p) continue;
    edge e =  G.last_adj_edge(last_v);
```

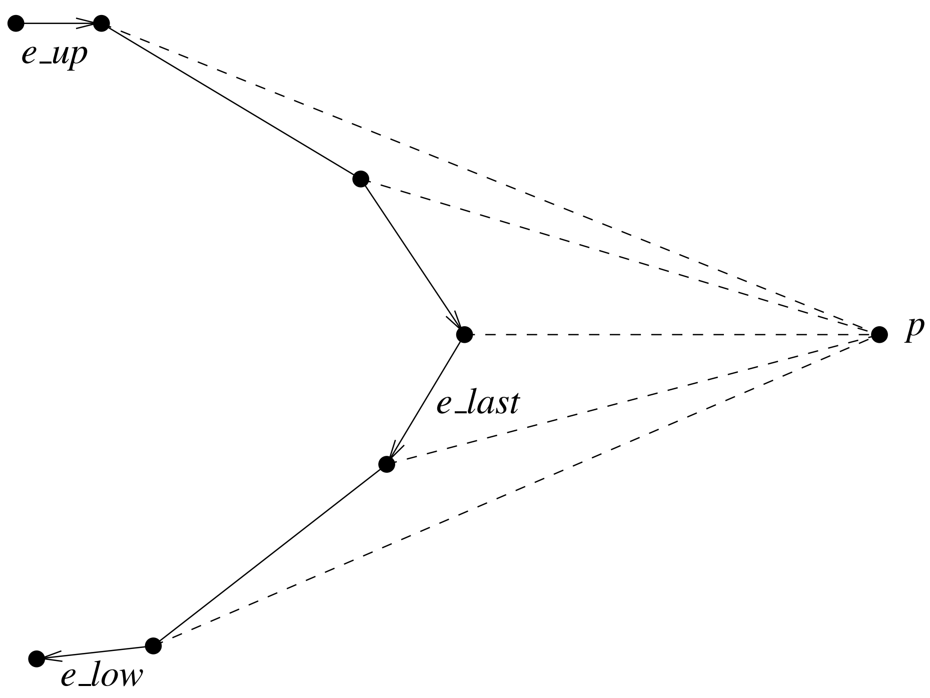


**Figure 10.15** Edges *e_last*, *e_up*, and *e_low*.

```
    last_v = G.new_node(p);
    last_p = p;
    // walk up to upper tangent
    do e = G.face_cycle_pred(e); while (left_bend(p,G,e));
    // now e = e_up
    // walk down to lower tangent and triangulate
    do { edge succ_e = G.face_cycle_succ(e);
         edge x = G.new_edge(succ_e,last_v,after,0);
         edge y = G.new_edge(last_v,source(succ_e),0);
         G.set_reversal(x,y);
         e = succ_e;
       } while (left_bend(p,G,e));
  }
```

⟨*mark edges of convex hull as HULL_DARTS*⟩

In the pieces of code above we labeled all new edges with zero. We now relabel all hull darts as such. The last edge added to the triangulation is a hull dart and all other hull darts are reached by tracing the face cycle containing it. The labeling of the hull darts will prove useful in the section on Delaunay diagrams.

We return a hull dart.

⟨*mark edges of convex hull as HULL_DARTS*⟩≡

```
  edge hull_dart = G.last_edge();
  if (hull_dart)
  { edge e = hull_dart;
    do { G[e] = HULL_DART;
```

| K | n | Gen | V | Hull | Hull check | Triang | Triang check |
|---|---|---|---|---|---|---|---|
| S | 20000 | 0.44 | 25 | 1.71 | 0 | 3.1 | 23.7 |
| S | 40000 | 0.92 | 29 | 3.65 | 0 | 6.43 | 47.35 |
| S | 80000 | 1.84 | 35 | 7.52 | 0 | 13.04 | 94.31 |
| D | 20000 | 0.41 | 91 | 1.9 | 0 | 3.13 | 24.72 |
| D | 40000 | 0.73 | 123 | 3.6 | 0 | 6.26 | 47.29 |
| D | 80000 | 1.47 | 147 | 7.72 | 0 | 13.15 | 94.36 |
| C | 20000 | 47.3 | 19992 | 1.69 | 0.17 | 2.62 | 21.19 |
| C | 40000 | 95.59 | 39958 | 3.59 | 0.42 | 5.47 | 42.01 |
| C | 80000 | 190.9 | 79756 | 8.08 | 1.32 | 11.65 | 86.39 |

**Table 10.3** The running times of the sweep algorithms for convex hulls and triangulations. We generated unsorted lists of $n$ points according to the same distributions as in Table 10.1. The meaning of the first four columns is as in Table 10.1. The column "Hull" shows the time to compute the convex hull, the column "Hull check" shows the time to verify that any three consecutive vertices of *CH* form a right turn, the column "Triang" shows the time to compute the triangulation, and the column "Triang check" shows the time to run *Is_Triangulation*$(G)$.

```
      e = G.face_cycle_succ(e);
    } while (e != hull_dart);
}
return hull_dart;
```

Table 10.3 compares the running times of the sweep algorithms for convex hulls and triangulations. We generated $n$ random points in a square, a disc, and on a circle, respectively, The triangulation algorithm takes about twice as long as the convex hull program. The table also shows the time for partially checking the output of either program. For the convex hull program we checked that any three consecutive vertices form a right turn and in the case of triangulations we called the checker *Is_Triangulation*$(G)$, which will be discussed in the next section.

Both checks are only partial. In the case of triangulations we do not check that exactly the input points appear as vertices of the triangulation. This omission could be corrected by the use of a dictionary. In the case of the convex hull program we do not verify that all input points lie inside the produced convex chain. This is an omission which is not easily corrected; the obvious approach takes quadratic time.

***Exercises for 10.2***

1 Write a program that verifies that the nodes of a *GRAPH*<*POINT*, *int*> agree with the points in a *list*<*POINT*>. Add this to the check of the triangulation program.

2 Extend the randomized incremental construction of convex hulls to an incremental construction of triangulations.

## 10.3 Verification of Geometric Structures, Basics

We have by now seen programs to compute convex hulls, minimum width stripes, and triangulations. The programs are non-trivial and we will see more complex programs in later sections. Although we wrote the documentation and the correctness proofs in parallel to the development of these programs, we nevertheless made mistakes, some minor, like testing for positive orientation instead of non-negative orientation, and some major, like assuming that every set of points contains three non-collinear points. *Visual debugging*, i.e., displaying the output of a geometric computation, was an indispensible aid in getting the programs correct, but visual debugging has its limits. Visual debugging is most useful in the plane; already displaying a partition of three-space is next to impossible. Also, the representation underlying a geometric object may be incorrect, although the object itself "looks correct".

One of our key experiences was the development of a program to compute convex hulls in arbitrary dimensions. It took some time to get the programs working for points in the plane, but after some time it produced convex hulls which "looked right". We moved to three-space and a few hours later the convex hulls in three-space looked right. We got adventurous and tried an example in seven-dimensional space. The program ran to completion and claimed that it had computed the convex hull. Given our past experience we had every reason to believe the contrary. At that time we had no way to check the result of the convex hull computation. We teamed up with some collegues and wrote [MNS$^+$96]. In this paper we discuss how to verify convex hulls, triangulations, Delaunay diagrams, and Voronoi diagrams. Alternative checkers are discussed in [DLPT97].

In this section and in Sections 10.4.3, 10.4.6, and 10.5.3 we derive procedures to verify properties of *geometric graphs*. A geometric graph is a straight line embedded map. Every node is mapped to a point in the plane and every dart is mapped to the line segment connecting its endpoints. We start with procedures to check that the edges around vertices are cyclically ordered, that face cycles define convex polygons, and that a graph defines a convex subdivision or a triangulation. In later sections we will extend these functions to check Delaunay triangulations, Delaunay diagrams, and Voronoi diagrams.

We use *geo_graph* as a template parameter for geometric graphs. Any instantiation *geo_graph_inst* of *geo_graph* must provide a function

```
VECTOR edge_vector(const geo_graph_inst& G, const edge& e)
```

that returns a vector from the source to the target of $e$. We will use two instantiations of *geo_graph* in this chapter: *GRAPH*<*POINT*, *int*> for triangulations, Delaunay triangulations, and Delaunay diagrams, and *GRAPH*<*CIRCLE*, *POINT*> for Voronoi diagrams. In the first case, the position of a node $v$ is given by the point $G[v]$ and hence the edge vector function can be realized as

⟨*GRAPH⟨POINT,int⟩: edge vector function*⟩≡

```
static VECTOR edge_vector(const GRAPH<POINT,int>& G, const edge& e)
{ return G[G.target(e)] - G[G.source(e)]; }
```

In the second case, the position of a node $v$ is given by the center of the circle $G[v]$. We will define the corresponding edge vector function in the section on Voronoi diagrams.

All functions that check properties of geometric graphs are collected in the file

⟨*geo_check.t*⟩≡

⟨*comparing edges by angle*⟩
⟨*cyclically ordered lists*⟩
⟨*verifying the order of adjacency lists and the convexity of faces*⟩

in directory LEDA/templates. This file must be included to use any of these functions.

### 10.3.1 *Monotone and Cyclically Monotone Sequences*

Let $x = (x_1, x_2, \ldots, x_n)$ be a sequence of elements from some ordered set; $x$ is called *non-decreasing* if $x_i \leq x_{i+1}$ for all $i$, $1 \leq i < n$, and $x$ is called *increasing* if $x_i < x_{i+1}$ for all $i$, $1 \leq i < n$, $x$ is called *cyclically non-decreasing* iff some cyclic shift of $x$ is non-decreasing, and $x$ is called *cyclically increasing* iff some cyclic shift of $x$ is increasing. The notions non-increasing, decreasing, cyclically non-increasing, and cyclically decreasing are defined analogously.

The functions *Is_C_Nondecreasing* and *Is_C_Increasing* check whether a sequence is cyclically non-decreasing or increasing. They take a list $L$ of elements of some type $T$ and a compare object *cmp* for type $T$.

The implementation is simple. We iterate over the elements of $L$ and compare every element with its cyclic successor. We count how often the successor is smaller (smaller or equal for the second function). If the count reaches two, the sequence violates the property.

⟨*cyclically ordered lists*⟩≡

```
template <class T>
bool Is_C_Nondecreasing(const list<T>& L, const leda_cmp_base<T>& cmp)
{ list_item it;
  int number_of_less = 0;
  forall_items(it,L)
    if ( cmp(L[L.cyclic_succ(it)],L[it]) < 0 ) number_of_less++;
  return (number_of_less < 2);
}
template <class T>
bool Is_C_Increasing(const list<T>& L, const leda_cmp_base<T>& cmp)
{ list_item it;
```

```
    int number_of_lesseq = 0;
    forall_items(it,L)
      if ( cmp(L[L.cyclic_succ(it)],L[it]) <= 0 ) number_of_lesseq++;
    return (number_of_lesseq < 2);
  }
```

The functions *Is_C_Nonincreasing* and *Is_C_decreasing* are defined analogously. We leave their implementation to the reader.

### 10.3.2 *Comparing Edges by Angle*

For a non-zero two-dimensional vector $v$ let $\alpha(v)$ be the angle between the positive $x$-axis and $v$, i.e., the angle by which the positive $x$-axis has to be turned counter-clockwise until it aligns with $v$. The geo kernels provide functions

```
int compare_by_angle(const VECTOR& v1,const VECTOR& v2)
```

that compare vectors by angle, i.e., the functions return $-1$ if *v1* precedes *v2*, 0 if *v1* and *v2* define the same angle, and $+1$ if *v1* succeeds *v2*. The zero vector precedes all non-zero vectors in the ordering by angle.

In a geometric graph $G$ the function *edge_vector*$(G, e)$ returns the vector from the source to the target of edge $e$. The compare object *cmp_edges_by_angle* compares the edges of any *geo_graph* $G$ according to the vectors defined by the edges of $G$. It is derived from *leda_cmp_base*<*edge*>, has a constructor that takes a geometric graph $G$ and stores a reference to it in the object, and a function operator that takes two edges $e$ and $f$ and compares them according to the vectors defined by them.

⟨*comparing edges by angle*⟩≡

```
  template <class geo_graph>
  class cmp_edges_by_angle: public leda_cmp_base<edge> {
    const geo_graph& G;
  public:
    cmp_edges_by_angle(const geo_graph& g): G(g){}
    int operator()(const edge& e, const edge& f) const
    { return compare_by_angle(edge_vector(G,e), edge_vector(G,f)); }
  };
```

### 10.3.3 *Counter-Clockwise Ordered Adjacency Lists*

The function

```
bool Is_CCW_Ordered(const geo_graph& G)
```

returns true if for all nodes $v$ the neighbors of $v$ are in increasing counter-clockwise order around $v$, and the function

```
bool Is_CCW_Ordered_Plane_Map(const geo_graph& G)
```

returns true if, in addition, $G$ is a plane map. The function

```
void SORT_EDGES(geo_graph& G)
```

reorders the adjacency lists such that for every node $v$ of $G$ the edges in $A(v)$ are in non-decreasing order by angle.

All three functions are very easy to implement. For the first function, we define a compare object *cmp* to compare the darts of $G$ by angle, and then check whether the darts out of every node $v$ are cyclically increasing. The second function calls the first and checks whether $G$ is a plane map, and the third function sorts the set of darts and then rearranges the adjacency lists.

⟨*verifying the order of adjacency lists and the convexity of faces*⟩≡

```
template <class geo_graph>
bool Is_CCW_Ordered(const geo_graph& G)
{ node v;
  cmp_edges_by_angle<geo_graph> cmp(G);
  forall_nodes(v,G)
    if ( !Is_C_Increasing(G.out_edges(v),cmp) ) return false;
  return true;
}
template <class geo_graph>
bool Is_CCW_Ordered_Plane_Map(const geo_graph& G)
{ return Is_Plane_Map(G) && Is_CCW_Ordered(G); }
template <class geo_graph>
bool Is_CCW_Weakly_Ordered(const geo_graph& G)
{ node v;
  cmp_edges_by_angle<geo_graph> cmp(G);
  forall_nodes(v,G)
    if ( !Is_C_Nondecreasing(G.out_edges(v),cmp) ) return false;
  return true;
}
template <class geo_graph>
bool Is_CCW_Weakly_Ordered_Plane_Map(const geo_graph& G)
{ return Is_Plane_Map(G) && Is_CCW_Weakly_Ordered(G); }
template <class geo_graph>
void SORT_EDGES(geo_graph& G)
{
  cmp_edges_by_angle<geo_graph> cmp(G);
  list<edge> L = G.all_edges();
  L.sort(cmp);
  G.sort_edges(L);
}
```

### 10.3.4 *Convex Faces*

We define functions that check for convexity of faces. Consider any face cycle $f$ of a geometric graph $G$; $f$ defines a closed polygonal chain $C$ in the plane. We want to know

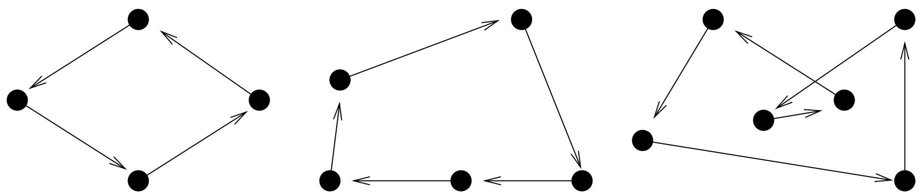

**Figure 10.16** A strictly convex counter-clockwise polygonal chain, a weakly convex clockwise polygonal chain, and a chain which is not simple.

whether the polygonal chain is the boundary of a convex region. More precisely, we call $C$ a *weakly convex counter-clockwise polygonal chain* if $C$ is simple, i.e., does not intersect itself, and the region to the left of $C$ is convex. We call $C$ a *strictly convex counter-clockwise polygonal chain* or simply *convex counter-clockwise polygonal chain* if, in addition, any two consecutive edges of $C$ do not have the same direction, see Figure 10.16. For clockwise chains the region to the right of $C$ must be convex.

In a convex subdivision, e.g., a triangulation, the face cycles of all bounded faces form convex counter-clockwise polygonal chains, and the face cycle of the unbounded face forms a weakly convex clockwise polygonal chain.

Let $p_0, p_1, \ldots, p_{k-1}$ be the points associated with the nodes of $C$.

**Lemma 5** *$C$ is a counter-clockwise weakly convex polygonal chain iff the sequence $s = (p_1 - p_0, p_2 - p_1, \ldots, p_0 - p_{k-1})$ is cyclically non-decreasing.*

*Proof* If $C$ is a counter-clockwise weakly convex polygonal chain then $s$ is clearly cyclically non-decreasing.

Assume next that $s$ is cyclically non-decreasing. Then no pair of consecutive vectors forms a right turn and the angles between all pairs of consecutive vectors sum to $2\pi$. We conclude that $C$ is simple, i.e, does not intersect itself, and that the region to the left of $C$ is convex. □

The functions

```
bool Is_CCW_Convex_Face_Cycle(const geo_graph& G,const edge e)
bool Is_CCW_Weakly_Convex_Face_Cycle(const geo_graph& G, const edge e)
bool Is_CW_Convex_Face_Cycle(const geo_graph& G, const edge e)
bool Is_CW_Weakly_Convex_Face_Cycle(const geo_graph& G, const edge e)
```

return true if the face cycle of $G$ containing $e$ has the stated property, i.e., if the face cycle forms a cyclically increasing, non-decreasing, decreasing, or non-increasing, respectively, sequence of edges according to the compare-by-angles ordering.

We give the implementation of the first function. We collect the edges of the face cycle in a list $L$, define a compare object *cmp* that compares edges of $G$, and then check whether $L$ is cyclically increasing.

⟨*verifying the order of adjacency lists and the convexity of faces*⟩+≡

```
template <class geo_graph>
bool Is_CCW_Convex_Face_Cycle(const geo_graph& G, const edge& e)
{
  list<edge> L;
  edge e1 = e;
  do { L.append(e1);
       e1 = G.face_cycle_succ(e1);
  } while ( e1 != e );
  cmp_edges_by_angle<geo_graph> cmp(G);
  return Is_C_Increasing(L,cmp);
}
```

### 10.3.5 *Convex Subdivisions*

A geometric graph $G$ is a *convex planar subdivision*, if $G$ is a plane map and if the positions assigned to the nodes of $G$ define a straight line embedding of $G$ in which all bounded faces are strictly convex and in which the unbounded face is weakly convex.

The function

```
bool Is_Convex_Subdivision(const GRAPH<POINT,int>& G)
```

returns true if $G$ is a convex planar subdivision, and the function

```
bool Is_Triangulation(const geo_graph& G)
```

returns true if $G$ is a convex planar subdivision in which every bounded face is a simplex. More precisely, if all nodes of $G$ lie on a common line, then every face cycle of a bounded face is simply a pair of anti-parallel edges, and if the nodes of $G$ do not lie on a common line, then every bounded face of $G$ is a triangle.

Both functions are implemented in terms of the function

```
bool Is_Convex_Subdivision(const GRAPH<POINT,int>& G,
                           bool& is_triangulated)
```

that returns true if $G$ is a convex subdivision and sets *is_triangulated* to true if, in addition, $G$ is a triangulation.

We discuss the theory behind the latter function and then give its implementation. If $G$ is a convex subdivision, then the following conditions are certainly satisfied:

- $G$ is a connected plane map.
- All nodes of $G$ have counter-clockwise ordered adjacency lists.
- If all vertices lie on a common line, i.e., the underlying point set has affine dimension less than 2, then $G$ is a path which reflects the ordering of its vertices on the line.
- If the underlying point set has affine dimension 2, then each face is either a bounded counter-clockwise oriented convex polygon or a clockwise oriented weakly convex polygon. There is only one face of the latter kind.

**Lemma 6** *If $G$ satisfies the four conditions above, then $G$ is a convex planar subdivision.*

*Proof* Assume first that all vertices of $G$ lie on a line $l$ and let $v_1, v_2, \ldots, v_n$ be the ordering of the vertices on $l$. Then the points assigned to adjacent vertices must be distinct, $v_1$ and $v_n$ must have degree one, and $v_i$ must have neighbors $v_{i-1}$ and $v_{i+1}$ for $1 < i < n$. The number of edges of $G$ is $2n - 2$ where $n$ is the number of nodes of $G$.

Assume next that not all vertices of $G$ lie on a common line. Let $R$ be the region that is enclosed by the unique face cycle $f$ which is a weakly convex clockwise polygon. We claim that all vertices that are not part of $f$ lie in the interior of $R$. Assume otherwise. Then there must be a vertex $v$ that is not part of $f$ and a direction $d$ such that $v$ is a maximal vertex of $G$ in direction $d$ (note that we said "a maximal vertex" and not "the maximal vertex"). Since $v$ is maximal there must be a pair of edges incident to $v$ which span an angle of at least $\pi$ and hence $v$ must be part of a weakly convex chain. Thus $v$ belongs to $f$, a contradiction.

Every face cycle of $G$ different from $f$ defines a counter-clockwise oriented convex polygonal region in the plane. We need to show that these regions form a partition of $R$. Consider a point $p$ moving in the plane such that it avoids vertices of $G$. Whenever $p$ crosses a directed edge $e$ it will enter another region (namely, the one to the left of *reversal*($e$)) except when *reversal*($e$) belongs to $f$. This shows that all points in the interior of $R$ are covered by the same number of regions. Also, since all vertices on the boundary of $R$ are part of $f$, exactly one bounded region is incident to each edge of $f$. Altogether we have shown that the regions defined by the face cycles different from $f$ partition $R$. The number of edges of $G$ must be at least $2n$ since every node must have degree at least two. □

We turn to the implementation. We first check whether $G$ is a connected plane map in which all adjacency lists are counter-clockwise ordered. Then we compare $m$ and $n$. If $m = 2n - 2$ we must be in the situation that all vertices of $G$ are collinear and if $m > 2n - 2$ we must be in the situation that the underlying point set has affine dimension 2.

⟨*subdivision_check.c*⟩+≡

```
static bool False(const string& s)
{ cerr << "Is_Convex_Subdivision: " << s; return false; }
bool Is_Convex_Subdivision(const GRAPH<POINT,int>& G,
                            bool& is_triangulated)
{
  is_triangulated = true;
  if ( !Is_Connected(G) ) return False("G is not connected");
  if ( !Is_CCW_Ordered_Plane_Map(G) )
    return False("G is not a CCW-ordered plane map");
  int n = G.number_of_nodes();
  int m = G.number_of_edges();
  cmp_edges_by_angle<GRAPH<POINT,int> > cmp(G);
  if ( m == 2*n - 2) { ⟨ICS: collinear points⟩ }
  ⟨ICS: affine dimension is two⟩
}
```

If $m = 2n - 2$, the fact that $G$ is a connected bidirected graph guarantees that $G$ is a tree. It therefore suffices to check that there is no vertex of degree three and that for every vertex of degree two the two incident edges point in opposite directions.

⟨*ICS: collinear points*⟩≡

```
node v;
if ( n <= 1 ) return true;
forall_nodes(v,G)
{ if ( G.outdeg(v) > 2 ) return False("G is a tree but not a chain");
  if (G.outdeg(v) == 1) continue;
  edge e1 = G.first_adj_edge(v), e2 = G.last_adj_edge(v);
  node w = G.target(e1);
  node u = G.target(e2);
  if ( G[v] == G[w] || G[v] == G[u] )
    return False("nodes at equal positions");
  if ( cmp(e1,G.reversal(e2)) != 0 )
     return False("direction not opposite");
}
return true;
```

It remains to deal with the situation that the affine dimension of the underlying point set is 2. We trace all face cycles of $G$. One face cycle must be a weakly convex clockwise oriented polygon and all other face cycles must be strongly convex counter-clockwise polygons. We make the distinction by considering three consecutive nodes of a face cycle and determining their orientation. If the orientation is positive, the face cycle must be a strongly convex counter-clockwise polygon, and if the orientation is non-positive, the face cycle must be the boundary of the unbounded face.

If the number of edges of the face cycle is three, the orientation test itself guarantees strong convexity and there is no need to trace the face cycle to check convexity.

⟨*ICS: affine dimension is two*⟩≡

```
edge e;
edge_array<bool> considered(G,false);
bool already_seen_unbounded_face = false;
forall_edges(e,G)
{ if ( !considered[e] )
  { // check the face to the left of e
    POINT a = G[source(e)];
    POINT b = G[target(e)];
    POINT c = G[target(G.face_cycle_succ(e))];
    int orient = orientation(a,b,c);
    int n = 0;
    edge e0 = e;
    do { considered[e] = true;
         e = G.face_cycle_succ(e);
         n++;
    } while ( e != e0);
    if ( orient > 0 )
```

```
      { if ( n > 3 )
        { is_triangulated = false;
          if ( !Is_CCW_Convex_Face_Cycle(G,e) )
            return False("non-convex bounded face");
        }
      }
      else
      { if ( already_seen_unbounded_face )
          return False("two faces qualify for unbounded face");
        already_seen_unbounded_face = true;
        if ( !Is_CW_Weakly_Convex_Face_Cycle(G,e) )
          return False("unbounded face is not weakly convex");
      }
    }
  }
  return true;
```

***Exercises for 10.3***

1 Improve the implementation of *Is_CWW_Ordered* and the functions checking convexity of faces. In our implementation we first construct a list of edges and then check this list for cyclic monotonicity. Avoid the construction of the list.

2 Improve the theory underlying *Is_Convex_Subdivision*. Is it necessary to check whether the edges in $A(v)$ are CCW-ordered or does this property follow from the condition that all bounded faces are counter-clockwise strongly convex polygonal chains?

3 Extend the function *Is_Convex_Subdivision* such that it works for *geo_graph* and not only for *GRAPH<POINT*, *int>*.

## 10.4 Delaunay Triangulations and Diagrams

A point set may in general be triangulated in many different ways. Depending on the application one triangulation is preferable over another. A triangulation that is useful in many contexts is the so-called *Delaunay triangulation*. A triangulation of a point set $S$ is called *Delaunay* if the interior of the circumcircle of any triangle in the triangulation contains no point of $S$. Figure 10.17 shows a Delaunay triangulation. The voronoi_demo and the point_set_demo in xlman illustrate Delaunay diagrams.

In this section we will first show the existence of Delaunay triangulations. The existence proof is constructive and yields a simple algorithms for the construction of Delaunay triangulation, the so-calling *flipping algorithm*. We give an implementation of the algorithm based on the so-called *incircle test*, a powerful geometric primitive. The Delaunay triangulation of a point set is in general not unique (if the point set contains co-circular points); it has, however, a substructure which is unique, the so-called *Delaunay diagram*. We characterize Delaunay diagrams and give some applications of Delaunay diagrams and triangulations.

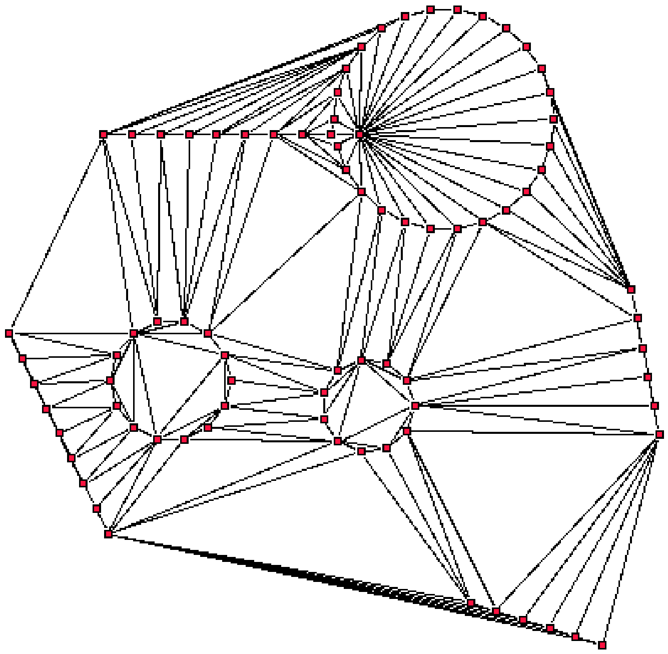

**Figure 10.17** A Delaunay triangulation. The figure was produced with the voronoi_demo in xlman.

### 10.4.1 *Delaunay Triangulations and the Flipping Algorithm*

Our immediate goal is to prove that Delaunay triangulations exist. Consider the simplest situation first, four points $p$, $q$, $r$, and $s$ forming the corners of a convex quadrilateral. There are two triangulations corresponding to the chords $pr$ and $qs$, respectively, see Figure 10.18. We show that at least one of the two triangulations is Delaunay. Assume that the triangulation corresponding to the chord $pr$ is not Delaunay, say because $s$ is contained in the circumcircle of triangle $\triangle(p, q, r)$. Then $q$ is also contained in the circumcircle of triangle $\triangle(p, r, s)$. We can obtain the circumcircle of triangle $\triangle(p, q, s)$ from the circumcircle of $\triangle(p, q, r)$ by reducing the size of the circle while simultaneously insisting that it passes through $p$ and $q$. This shows that $r$ is outside the circumcircle of triangle $\triangle(p, q, s)$ and that the radius of the circumcircle of $\triangle(p, q, s)$ is smaller than the radius of the circumcircle of $\triangle(p, q, r)$. The symmetric argument shows that $p$ is outside the circumcircle of triangle $\triangle(q, r, s)$ and that the radii of both circles in the Delaunay triangulation are smaller than the radii of the circles in the other triangulation.

Let us next turn to point sets of larger cardinality. We show that any triangulation which is not Delaunay contains two adjacent triangles, i.e., triangles sharing an edge, that form a convex quadrilateral and such that the circumcircles of both triangles contain the third vertex of the other triangle. Clearly, a triangulation which is not Delaunay contains a triangle, say $\triangle(p, q, r)$ whose circumcircle is non-empty. Assume w.l.o.g. that there is a point $s$ contained in the region $R$ formed by the chord $pq$ and the circular arc connecting $p$ and $q$ and not containing $r$, see Figure 10.19. Consider the other triangle incident to edge $pq$. If

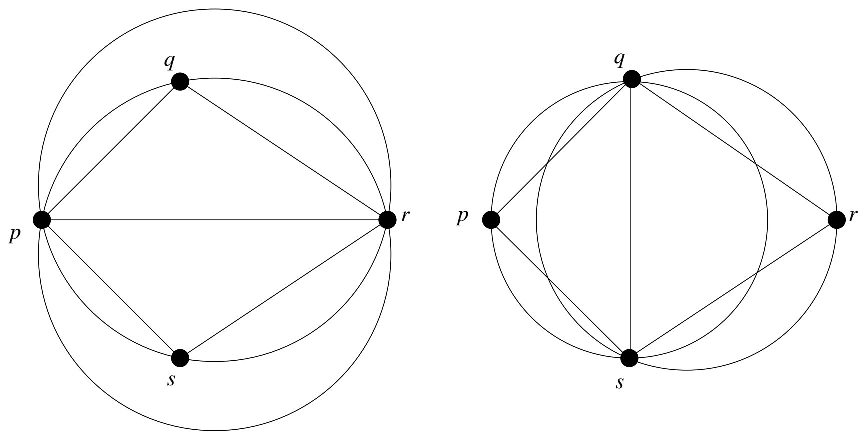


**Figure 10.18** The two triangulations of a convex quadrilateral.

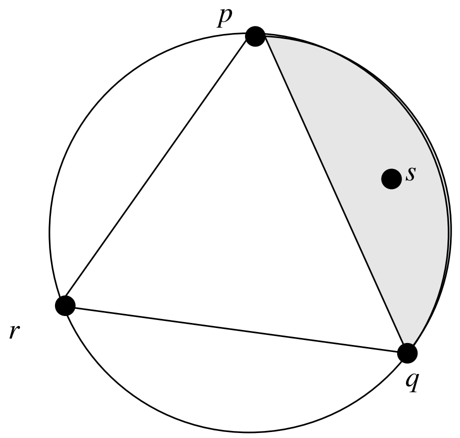


**Figure 10.19** A triangle $\triangle(p, q, r)$ with non-empty circumcircle. Region $R$ is shown shaded.

the third vertex of this triangle is also contained in $R$, we have identified the desired pair of triangles. If the third vertex, say $t$, is outside $R$ then $s$ is also contained in the circumcircle of triangle $\triangle(p, q, t)$ and $s$ is closer to $\triangle(p, q, t)$ than to $\triangle(p, q, r)$. Here, the distance of a point to a triangle is the distance to the closest point of the triangle. We repeat the argument with triangle $\triangle(p, q, t)$ and point $s$. After a finite number of steps we must arrive at the first case.

We have now shown that any triangulation that is not Delaunay contains a convex quadrilateral formed by two adjacent triangles such that the triangulation of this quadrilateral is not Delaunay. The deletion of the common edge of both triangles and the insertion of the other diagonal of the quadrilateral is called a *diagonal-flip* or simply *flip*. A flip makes the triangulation locally Delaunay and also decreases the sum of the radii of the circumcircles of all triangles. We have thus arrived at the so-called flipping algorithm for Delaunay triangulations:

$T$ = some triangulation;
**while** ($T$ is not Delaunay)
{ find a pair of adjacent triangles that form a convex quadrilateral and whose triangulation is not Delaunay;

flip the diagonal of the quadrilateral;
}

The algorithm terminates since every flip reduces the sum of the radii of all circumcircles and hence no triangulation can repeat. The maximal number of flips performed by the flipping algorithm is $\Theta(n^2)$. We ask you in the exercises to construct a worst case point set. The upper bound follows from the fact that once a segment $pq$ is flipped away it will never be reintroduced into the triangulation. The flipping algorithm is due to Lawson ([Law72]).

For points in convex position[4] there is also a so-called *furthest site Delaunay triangulation*. In a furthest site Delaunay triangulation of a set $S$ the circumcircle of any triangle has no point of $S$ in its exterior. The flipping algorithm can also be used to construct furthest site Delaunay triangulation. We start with an arbitrary triangulation of a set of points in convex position and flip as long as the triangulation is not furthest site Delaunay. Of course, this time we flip the diagonal of a convex quadrilateral if the third vertex of the other triangle is outside the circumcircle.

When it is necessary to emphasize the difference between ordinary Delaunay triangulations and furthest site Delaunay triangulations we call the former nearest site Delaunay triangulations. Some algorithms work for nearest and furthest site Delaunay triangulations. In these algorithms we use the enumeration type

```
enum delaunay_voronoi_kind { NEAREST, FURTHEST };
```

defined in LEDA/geo_global_enums.h to distinguish between the two kinds of triangulations.

As in the preceding section we use the type *GRAPH<POINT, int>* to represent triangulations. For every node $v$ of $G$ the associated point is given by $G[v]$. For every edge $e$ of $G$, $G[e]$ is an integer in the enumeration type *delaunay_edge_info*. In the Delaunay triangulation all hull darts are labeled HULL_DART, and every other dart is labeled either DIAGRAM_DART or NON_DIAGRAM_DART. A non-hull dart is labeled DIAGRAM_DART if the circumcircles of the triangles incident to it are distinct and is labeled NON_DIAGRAM_DART otherwise. The reversals of hull darts are labeled DIAGRAM_DART.

The functions

```
void DELAUNAY_TRIANG(  const list<POINT>& L, GRAPH<POINT,int>& G);
void F_DELAUNAY_TRIANG(const list<POINT>& L, GRAPH<POINT,int>& G);
```

compute the nearest site and the furthest site Delaunay triangulation of a list $L$ of points.

[4] A set $S$ of points is in convex position if every point in $S$ is a vertex of the convex hull of $S$.

### 10.4.2 *The Flipping Algorithm*

We turn the flipping algorithm into a program[5]. The flipping algorithm works for nearest and furthest site Delaunay triangulations.

We assume that we start with a triangulation $G$ in which all hull darts are labeled with the label HULL_DART and in which all other darts have a label different from HULL_DART. The algorithm terminates with a Delaunay triangulation and returns the number of flips performed. For furthest site triangulations we assume further that the vertices of $G$ are in convex position.

The algorithm maintains a set $S$ of darts which may potentially violate the Delaunay property. Initially, $S$ consists of one dart in each uedge of $G$. The algorithm terminates when $S$ is empty. As long as $S$ is non-empty, an arbitrary dart $e$ of $S$ is chosen. If it violates the Delaunay property, a flip is performed.

We define the integer $f$ to be $+1$ if we are aiming for a nearest site diagram and to be $-1$ if we are aiming for a furthest site diagram. It will be used in the test for the Delaunay property.

⟨*flip_delaunay.c*⟩≡

```
int DELAUNAY_FLIPPING(GRAPH<POINT,int>& G, delaunay_voronoi_kind kind)
{
  if (G.number_of_nodes() <= 3) return 0;
  int f = ( kind == NEAREST ? +1 : -1);
  list<edge> S;
  edge e;
  forall_edges(e,G) if ( index(e) < index(G.reversal(e)) ) S.append(e);
  int flip_count = 0;
  while ( !S.empty() )
  { edge e = S.pop();
    edge r = G.reversal(e);
    ⟨check e for the Delaunay property and flip if necessary⟩
  }
  return flip_count;
}
```

Let $e$ be a dart of the current triangulation. If $e$ is a hull dart or the reversal of a hull dart, then no action is required as hull darts belong to every Delaunay triangulation. If $e$ is not a hull dart, define edges $r$, $e_1$, and $e_3$, and points $a$, $b$, $c$, and $d$ as in Figure 10.20; $r$ is the reversal of $e$, $e_1$ is the face cycle successor of $r$, $e_3$ is the face cycle successor of $e$, $a$ and $b$ are source and target of $e_1$, and $c$ and $d$ are source and target of $e_3$. The quadrilateral $(a, b, c, d)$ is convex if and only if the interior angles at vertices $a$ and $c$ are less than $180^\circ$, i.e., if $(d, a, b)$ and $(b, c, d)$ are left turns.

[5] The program delaunay_flip_anim in LEDAROOT/book/Geo animates the algorithm.

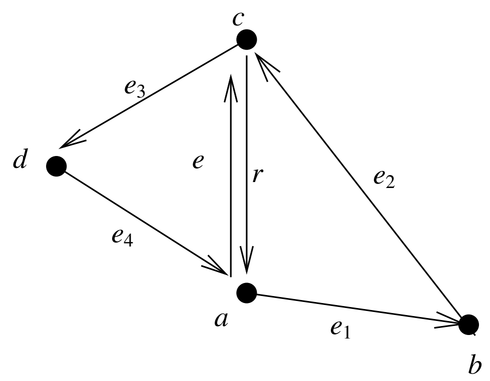


**Figure 10.20** The edges $e$, $r$, $e_1$, $e_2$, $e_3$, and $e_4$, and the points $a$, $b$, $c$, and $d$.

⟨*check e for the Delaunay property and flip if necessary*⟩≡

```
if (G[e] == HULL_DART || G[r] == HULL_DART) continue;
G[e] = DIAGRAM_DART;
G[r] = DIAGRAM_DART;
// e1,e2,e3,e4: edges of quadrilateral with diagonal e
edge e1 = G.face_cycle_succ(r);
edge e3 = G.face_cycle_succ(e);
// flip test
POINT a = G[source(e1)];
POINT b = G[target(e1)];
POINT c = G[source(e3)];
POINT d = G[target(e3)];
if ( left_turn(d,a,b) && left_turn(b,c,d) )
{ // the quadrilateral is convex
  ⟨check circle property and flip if necessary⟩
}
```

Assume now that the quadrilateral $(a, b, c, d)$ is convex. The triangulation is locally Delaunay if $d$ does not lie inside the circle defined by $(a, b, c)$, and can be improved by a flip if $d$ lies inside the circle. For the furthest site triangulation the situation is reversed. The test

```
side_of_circle(a,b,c,d)
```

returns

$+1$ if $d$ is left of the oriented circle through $a$, $b$, and $c$,
$0$ if $|\{a, b, c\}| \leq 2$ or $d$ lies on the oriented circle through $a$, $b$, and $c$,
$-1$ if $d$ is right of the oriented circle through $a$, $b$, and $c$.

Let $soc = f \cdot side\_of\_circle(a, b, c, d)$. If $soc$ is zero, the four points are co-circular, and no flip is required. However, $e$ and $r$ have to be relabeled with NON_DIAGRAM_DART. If $soc$ is positive, $d$ lies inside the circumcircle of the triangle $(a, b, c)$ (outside for furthest site triangulations) and a flip is required. Let $e_2$ and $e_4$ be the other two edges of the quadrilateral $(a, b, c, d)$. We move $e$ and $r$ to the other diagonal of the quadrilateral. More precisely, we insert $e$ after $e_2$ into $A(source(e_2))$[6] and make $source(e4)$ the target of $e$, and we insert $r$ after $e_4$ into $A(source(e_4))$ and make $source(e2)$ the target of $r$. We also add all four sides of the quadrilateral to $S$ to make sure that their Delaunay property is rechecked. Observe that flipping $e$ may affect the "Delaunay-ness" of the sides of the quadrilateral.

⟨*check circle property and flip if necessary*⟩≡

```
int soc = f * side_of_circle(a,b,c,d);
if (soc == 0) // co-circular quadrilateral(a,b,c,d)
{ G[e] = NON_DIAGRAM_DART;
  G[r] = NON_DIAGRAM_DART;
}
if (soc > 0) // flip
{ edge e2 = G.face_cycle_succ(e1);
  edge e4 = G.face_cycle_succ(e3);
  S.push(e1);
  S.push(e2);
  S.push(e3);
  S.push(e4);
  // flip diagonal
  G.move_edge(e,e2,source(e4));
  G.move_edge(r,e4,source(e2));
  flip_count++;
}
```

In order to construct the Delaunay triangulation for a set of points we first triangulate the set of points and then call the flipping algorithm to turn the triangulation into a Delaunay triangulation.

In the case of the furthest site Delaunay triangulation we first extract the vertices of the convex hull, then construct a triangulation of them, and finally use the flipping algorithm to obtain a furthest site Delaunay triangulation.

⟨*flip_delaunay.c*⟩+≡

```
int DELAUNAY_FLIP(const list<POINT>& L, GRAPH<POINT,int>& G)
{ TRIANGULATE_POINTS(L,G);
  if (G.number_of_edges() == 0) return 0;
  return DELAUNAY_FLIPPING(G,NEAREST);
}
int F_DELAUNAY_FLIP(const list<POINT>& L, GRAPH<POINT,int>& G)
{
  list<POINT> H = CONVEX_HULL(L);
  TRIANGULATE_POINTS(H,G);
  if (G.number_of_edges() == 0) return 0;
  return DELAUNAY_FLIPPING(G,FURTHEST);
}
```

[6] Recall that for a node $v$, $A(v)$ is the counter-clockwise ordered cyclic list of darts out of $v$.

### 10.4.3 *Verifying Delaunay Triangulations*

The function

```
bool Is_Delaunay_Triangulation(const GRAPH<POINT,int>& G,
                               delaunay_voronoi_kind kind);
```

checks whether $G$ is a Delaunay triangulation of the points associated with its nodes. The flag *kind* allows us to choose between nearest and furthest site diagrams.

Let $S$ be the set of points associated with the nodes of $G$. $G$ is a Delaunay triangulation of $S$, if $G$ is a triangulation and every triangle of $G$ has the Delaunay property.

Thus the implementation is simple. First we check whether $G$ is a triangulation. If the affine dimension of $S$ is less than 2 this suffices; the affine dimension is less than 2 if $m = 2n - 2$. Otherwise, we walk over all edges. If an edge separates two triangles that form a convex quadrilateral we check the Delaunay property.

⟨*delaunay_check.c*⟩+≡

```
static bool False(const string& s)
{ cerr << "Is_Delaunay_Triangulation: " << s; return false; }
bool Is_Delaunay_Triangulation(const GRAPH<POINT,int>& G,
                               delaunay_voronoi_kind kind)
{ if ( !Is_Triangulation(G) ) return False("G is no triangulation");
  if (G.number_of_edges() == 2*G.number_of_nodes() - 2) return true;
  ⟨check Delaunay property⟩
  return true;
}
```

where

⟨*check Delaunay property*⟩≡

```
edge e;
edge_array<bool> considered(G,false);
forall_edges(e,G)
{ if (!considered[e])
  { // check the faces incident to e and reversal(e)
    considered[e] = considered[G.reversal(e)] = true;
    POINT a = G[source(e)];
    POINT b = G[target(G.cyclic_adj_pred(e))];
    POINT c = G[target(e)];
    POINT d = G[target(G.face_cycle_succ(e))];
    if (left_turn(a,b,c) && left_turn(b,c,d) &&
           left_turn(c,d,a) && left_turn(d,a,b) )
    { // the faces to the left and right of e are bounded
      int s = side_of_circle(a,b,c,d);
      /* +1 for inside, -1 for outside */
      if ( (kind == NEAREST && s > 0) || (kind == FURTHEST && s < 0) )
        return False("violated Delaunay property");
    }
  }
}
```

| K | n | Flipping | Guibas–Stolfi | Dwyer | Check |
|---|---|---|---|---|---|
| S | 20000 | 26.4 | 17.36 | 8.57 | 25.63 |
| S | 40000 | 56.89 | 37.45 | 17.44 | 51.66 |
| S | 80000 | 122.1 | 79.61 | 36.35 | 102.7 |
| D | 20000 | 26.13 | 17.22 | 8.71 | 25.53 |
| D | 40000 | 56.28 | 37.1 | 17.62 | 51.09 |
| D | 80000 | 120.8 | 78.49 | 36.92 | 102.7 |
| C | 20000 | 14.66 | 10.6 | 11.09 | 27.72 |
| C | 40000 | 29.74 | 21.73 | 22.89 | 55.87 |
| C | 80000 | 60.74 | 44.55 | 45.29 | 111 |

**Table 10.4** The running times of Delaunay triangulation algorithms. The first column designates the kind of input (S for random points in a square, D for random points in a disk, C for random points near a circle), and the other columns show the number of points, the running time of the flipping algorithm, the running time of the algorithm of Guibas and Stolfi, the running time of the algorithm of Dwyer, and the time to verify the correctness of the result, respectively.

### 10.4.4 *Other Algorithms for Delaunay Triangulations*

The flipping approach yields a simple but not the most efficient Delaunay triangulation algorithm. There are $O(n \log n)$ algorithms based on sweeping [For87], on divide-and-conquer [GS85, Dwy87], and on randomized incremental construction [BT93]. The paper [SD97] compares many Delaunay algorithms.

In LEDA the divide-and-conquer algorithms of Guibas and Stolfi and of Dwyer are available. Table 10.4 shows an experimental comparison of the flipping algorithm with the two divide-and-conquer algorithms. The algorithm of Dwyer is consistently the best and therefore we use it as our default implementation. For the furthest site diagram we only have the flipping algorithm.

⟨*delaunay.c*⟩≡

```
void DELAUNAY_TRIANG(const list<POINT>& L, GRAPH<POINT,int>& G)
{ DELAUNAY_DWYER(L,G); }

void F_DELAUNAY_TRIANG(const list<POINT>& L, GRAPH<POINT,int>& G)
{ F_DELAUNAY_FLIP(L,G); }
```

### 10.4.5 *Delaunay Diagrams*

The Delaunay triangulation of a set $S$ is in general not unique, e.g., if $S$ consists of the corners of a square, or more generally of four co-circular points, then both triangulations of $S$

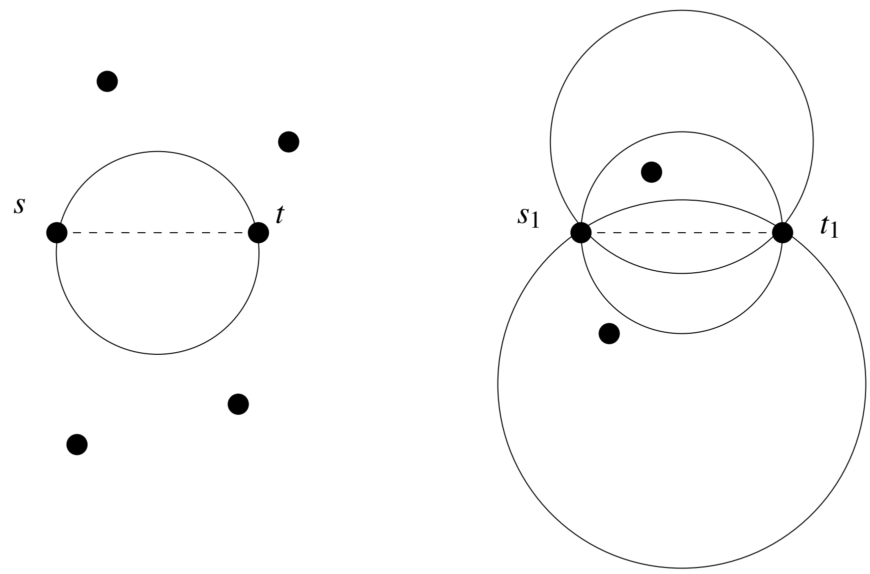


**Figure 10.21** $st$ is essential but $s_1t_1$ is not.

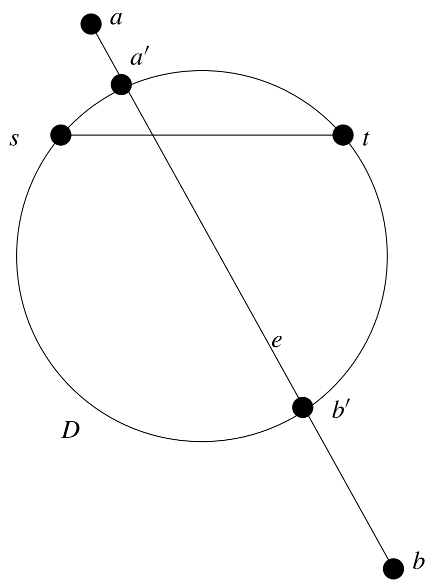


**Figure 10.22** An essential segment $st$ with its disk $D$ and an edge $e = (a, b)$ of a Delaunay triangulation intersecting $st$.

are Delaunay. We now characterize the segments that belong to all Delaunay triangulations. Let $s$ and $t$ be two distinct points in $S$. A segment $st$ is called *essential* if there is a closed disk $D$ with $S \cap D = \{s, t\}$. In other words, there is a circle passing through $s$ and $t$ such that $s$ and $t$ are the only points of $S$ contained in the closure of the circle, see Figure 10.21. We have

**Lemma 7** *Let $S$ be a finite set of points in the plane and let $s$ and $t$ be distinct points in $S$. The segment $st$ is essential if and only if it belongs to every Delaunay triangulation of $S$.*

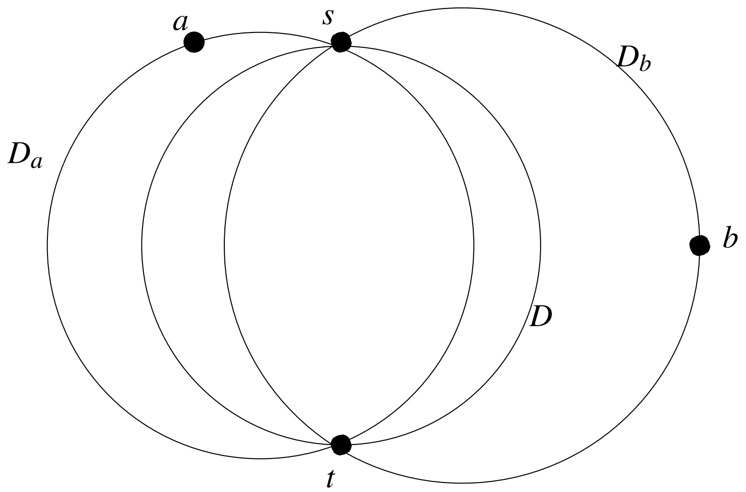


**Figure 10.23** The discs $D_a$, $D_b$, and $D$.

*Proof* We first show that essential segments belong to all Delaunay triangulations. Assume otherwise, say $st$ is essential but does not belong to some Delaunay triangulation $T$. Then $st$ cannot be an edge of the convex hull of $S$ because any such edge belongs to every triangulation. The open segment $st$ is therefore contained in the interior of conv $S$. Imagine travelling along the segment $st$ from $s$ to $t$. In the vicinity of $s$ the segment $st$ runs inside some triangle of $T$ and in the vicinity of $t$ it runs inside some other triangle of $T$. We conclude that the segment $st$ must intersect an edge $e = (a, b)$ of $T$. Since $st$ is essential there is a closed disk $D$ with $S \cap D = \{s, t\}$. Let $a'$ and $b'$ be the intersections of the boundary of $D$ with edge $e$, see Figure 10.22. The four points $a'$, $s$, $b'$, and $t$ form the corners of a convex quadrilateral and are co-circular. This implies that any closed disk containing the segment $a'b'$ must also contain either $s$ or $t$. Consider next any of the triangles of $T$ incident to $e$. The circumcircle of this triangle contains the segment $a'b'$ in its interior and hence also contains either $s$ or $t$ in its interior. The triangle is therefore not Delaunay, a contradiction. This proves that essential edges are part of every Delaunay triangulation.

To show the converse consider a non-essential segment $st$. We will construct a Delaunay triangulation that does not contain $st$. Let $T$ be any Delaunay triangulation of $S$. If $st$ is not an edge of $T$ we are done. Otherwise, consider the two triangles $\triangle$ and $\triangle'$ incident to $st$ in $T$; it is easy to see that $st$ is not a hull edge and hence the two triangles exist. Let $a$ and $b$ be the third vertices of $\triangle$ and $\triangle'$, respectively. If the four points $s$, $a$, $t$, $b$ are co-circular then we may replace $st$ by $ab$ and stay Delaunay. So, assume that the four points are not co-circular. Then $b$ is outside the closed disk $D_a$ having $s$, $a$, and $t$ on its boundary and $a$ is outside the closed disk $D_b$ having $s$, $b$, and $t$ on its boundary, see Figure 10.23. Consider the closed disk $D$ having $s$ and $t$ on its boundary and having its center at the midpoint of the centers of $D_a$ and $D_b$; all of $D$ (except for $s$ and $t$) is contained in the interior of $D_a \cup D_b$. Thus, $D \cap S \subseteq \{s, t\}$ and $st$ is essential, a contradiction. $\square$

We can now define the *Delaunay diagram* of a set $S$ of points. It consists of all essential segments defined by the points in $S$ and is denoted $DD(S)$. The Delaunay diagram is a subgraph of every Delaunay triangulation. The Delaunay diagram is a planar graph whose bounded faces are convex polygons all of whose vertices are co-circular. If no four points of $S$ are co-circular then all bounded faces are triangles and the Delaunay diagram is a triangulation.

It is trivial to construct the Delaunay diagram from a Delaunay triangulation. We only have to delete all edges that are labeled NON_DIAGRAM_DART.

⟨*delaunay.c*⟩+≡

```
void DELAUNAY_DIAGRAM(const list<POINT>& L, GRAPH<POINT,int>& DD)
{
  DELAUNAY_TRIANG(L,DD);
  list<edge> el;
  edge e;
  forall_edges(e,DD) if ( DD[e] == NON_DIAGRAM_DART) el.append(e);
  forall(e,el) DD.del_edge(e);
}
```

For furthest site diagrams the construction is completely analogous and therefore not shown.

### 10.4.6 *Verifying Delaunay Diagrams*

We show how to verify Delaunay diagrams. The function

```
bool Is_Delaunay_Diagram(const GRAPH<POINT,int>& G,
                         delaunay_voronoi_kind kind);
```

checks whether $G$ is a Delaunay diagram of the points associated with its nodes. The flag *kind* allows us to choose between nearest and furthest site diagrams. Let $S$ be the set of points associated with the nodes of $G$.

It is clearly necessary that $G$ is a convex subdivision in which the vertices of every bounded face (= a face whose face cycle is a convex counter-clockwise polygon) are co-circular. Assume this is the case. Then $G$ is a Delaunay diagram if an arbitrary triangulation of $G$ is a Delaunay triangulation. It therefore suffices to check the Delaunay property of all edges of $G$ as in ⟨*check Delaunay property*⟩.

⟨*delaunay_check.c*⟩+≡

```
static bool False_IDD(const string& s)
{ cerr << "Is_Delaunay_Diagram: " << s; return false; }
bool Is_Delaunay_Diagram(const GRAPH<POINT,int>& G,
                         delaunay_voronoi_kind kind)
{
  if ( !Is_Convex_Subdivision(G) )
    return False_IDD("G is no convex subdivision");
  edge e;
  edge_array<bool> considered(G,false);
  forall_edges(e,G)
```

```
  { if (!considered[e])
    { // check the face to the left of e
      POINT a = G[source(e)];
      POINT c = G[target(e)];
      POINT d = G[target(G.face_cycle_succ(e))];
      if ( left_turn(a,c,d) )
      { // face is bounded
        CIRCLE C(a,c,d);
        edge e0 = e;
        do { considered[e] = true;
             if ( !C.contains(G[source(e)]) )
             return False_IDD("face with non-co-circular vertices");
             e = G.face_cycle_succ(e);
        } while ( e != e0 );
      }
      else
      { // face is unbounded
        edge e0 = e;
        do { considered[e] = true;
             e = G.face_cycle_succ(e);
        } while ( e != e0 );
      }
    }
  }
  { ⟨check Delaunay property⟩ }
  return true;
}
```

### 10.4.7 ***Applications***

Delaunay triangulations have several useful properties. We mention three:

- For a triangulation $T$ let $\mu(T)$ be the smallest interior angle of any triangle in $T$. Delaunay triangulations maximize $\mu(T)$.
- Delaunay triangulations tend to produce "rounder" triangles than other triangulations, see Figure 10.24, a property desirable for numerical applications of triangulations. For example, the interpolation scheme presented at the beginning of Section 10.2 is numerically more stable if the triangulation contains no "skinny" triangles.
- The Euclidian minimum spanning tree of a set $S$ is a tree of minimum cost connecting all points in $S$, where the cost of an edge is its Euclidian length. The Euclidian minimum spanning tree is a subgraph of the Delaunay diagram.

The function

```
void MIN_SPANNING_TREE(const list<POINT>& L, GRAPH<POINT,int>& T)
```

computes the Euclidian minimum spanning tree for the points in $L$. It first constructs the Delaunay diagram $T$ for $L$, then runs the minimum spanning tree algorithm on $T$, and finally deletes all edges from $T$ that do not belong to the minimum spanning tree.

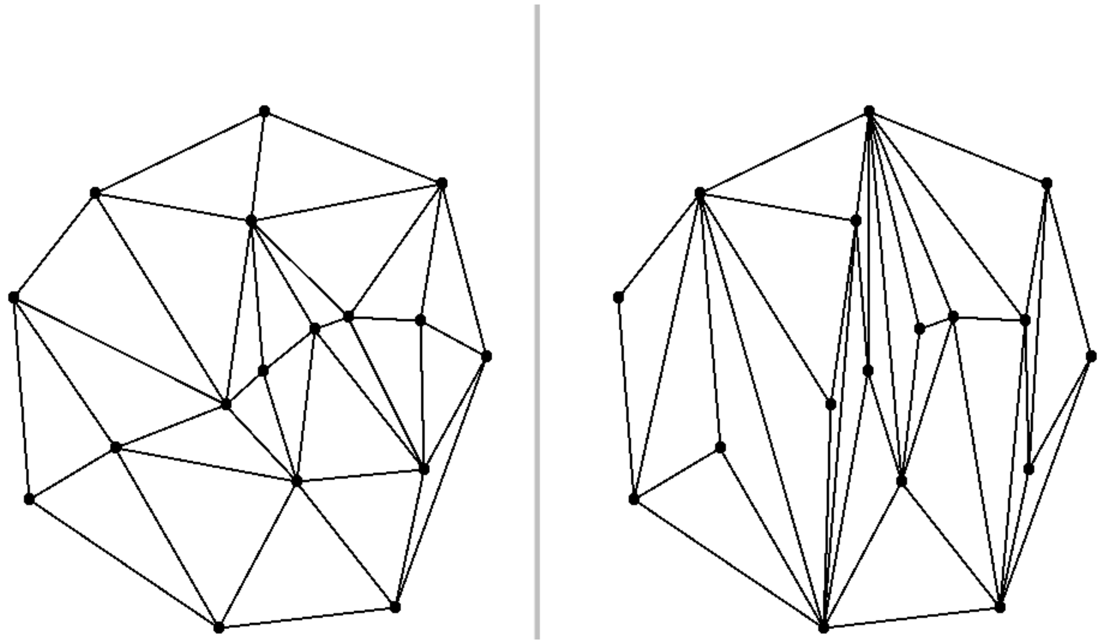

**Figure 10.24** A Delaunay triangulation and a triangulation produced by sweeping. The Delaunay triangulation is shown on the left. The triangles in the Delaunay triangulation are "rounder" than in the triangulation by sweeping. The figure was generated with the triangulation_demo (see LEDAROOT/demo/book/Geo).

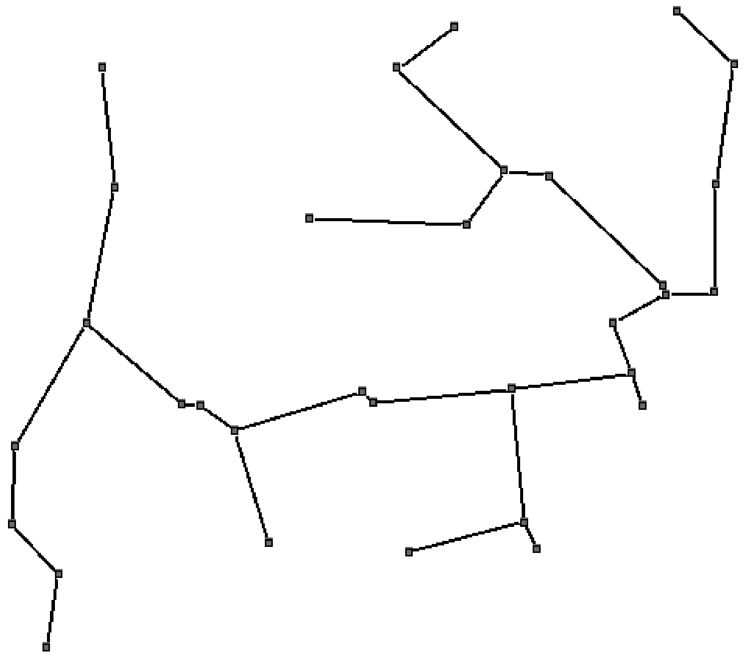

**Figure 10.25** A point set and its Euclidian minimum spanning tree. The figure was generated with the voronoi_demo in xlman.

***Exercises for 10.4***

1 Show that the flipping algorithm constructs a furthest site Delaunay triangulation for a set of points in convex position.

2 Extend the functions for checking Delaunay triangulations and Delaunay diagrams such that they also check the edge labels.

3 Write a program that takes a Delaunay triangulation and draws it into a graphics *window*. For each triangle the circumcircle should also be displayed.

4 Consider the points $(i, i^2)$, $0 \le i < n$. Show that the Delaunay triangulation of this point set has a fan-like shape. Show that the flipping algorithm may perform $\Omega(n^2)$ flips when starting with the "opposite fan".

5 (Euclidian minimum spanning tree (EMST)) For a set $S$ of points in the plane a tree $T$ of minimum cost connecting all points in $S$ is called a Euclidian minimum spanning tree of $S$. The cost of an edge is defined as its Euclidian length.
a) Show that every edge of an EMST is essential. (Hint: For an edge $e$ with endpoints $a$ and $b$ consider the circle centered at the midpoint of $e$ and passing through $a$ and $b$. Assume that it contains a point $c \in S \setminus \{a, b\}$. Show that a better tree can be obtained by removing $e$ and adding either $(a, c)$ or $(c, b)$.)
b) Conclude from part a) that an EMST is a subgraph of the Delaunay diagram. Write a program to compute an EMST. Make use of programs for Delaunay diagrams and minimum spanning trees. Try to work with the squared length of edges instead of their length.

6 For a triangulation $T$ let $\alpha(T)$ be the sorted tuple of all interior angles of all triangles in $T$. Consider Figure 10.18 and let $T_1$ and $T_2$ be the two triangulations shown with $T_2$ being Delaunay. Show that $\alpha(T_1) \le \alpha(T_2)$ where the ordering on tuples is the lexicographic one. Consider next any triangulation $T$ of a set $S$ that is not Delaunay and let $T'$ be obtained from $T$ by a diagonal flip. Show that $\alpha(T) \le \alpha(T')$. Conclude that Delaunay triangulations maximize the smallest interior angle.

7 Improve the implementation of the flipping algorithm by ensuring that, for any pair of darts in a uedge, at most one is in $S$. Observe that we ensure this property only at the time of initialization. Does the running time improve?

## 10.5 Voronoi Diagrams

We discuss Voronoi diagrams. We define them and discuss their representation by graphs. We relate them to Delaunay triangulations and show how to obtain Voronoi diagrams from Delaunay triangulations. Finally, we discuss applications and the verification of Voronoi diagrams.

### 10.5.1 ***Definition and Representation***

A structure closely related to the Delaunay diagram is the so-called *Voronoi diagram*. Let $S$ be a set of points in the plane. We will refer to the elements of $S$ as *sites*. For any point $p$ of the plane let $close(p)$ be the set of sites that realize the closest distance between $p$ and the sites in $S$, i.e., $s \in close(p)$ if $dist(s, p) \le dist(t, p)$ for all $t \in S$. In other words, there is a circle with center $p$ passing through all points in $close(p)$ and having no points of $S$ in its interior, see Figure 10.26. For most points $p$ of the plane $close(p)$ consists of only a

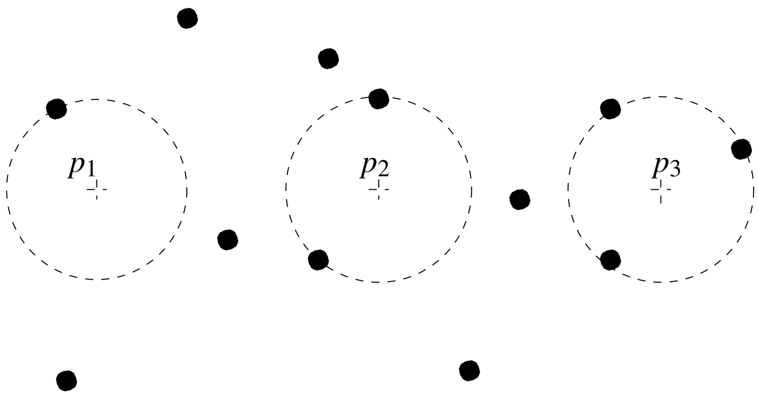


**Figure 10.26** Sites are shown as dots. The point $p_i$ has $i$ sites in $close(p_i)$.

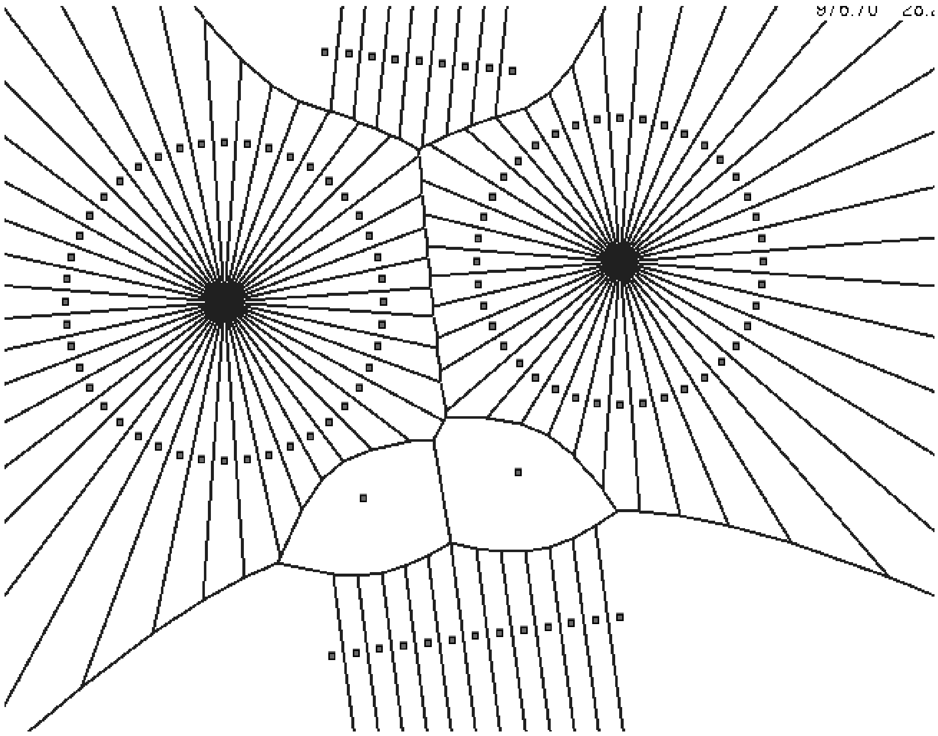

**Figure 10.27** A Voronoi diagram. The figure was generated with the voronoi_demo in xlman.

single site. For some points $p$, $close(p)$ contains two or more sites. These points form the so-called Voronoi diagram $VD(S)$ of $S$.

$$VD(S) = \{p \in I\!R^2; |close(p)| \geq 2\}.$$

Figure 10.27 shows a Voronoi diagram. The Voronoi diagram is a graph-like structure. Its vertices are all points $p$ with $|close(p)| \geq 3$, its edges are maximal connected sets of points $p$ with $|close(p)| = 2$, and its faces are maximal connected sets of points $p$ with $|close(p)| = 1$.

We derive some more properties of edges and faces. Consider any edge $e$ of the Voronoi diagram, and let $s$ and $t$ be the two sites of $S$ such that $close(p) = \{s, t\}$ for all points $p$

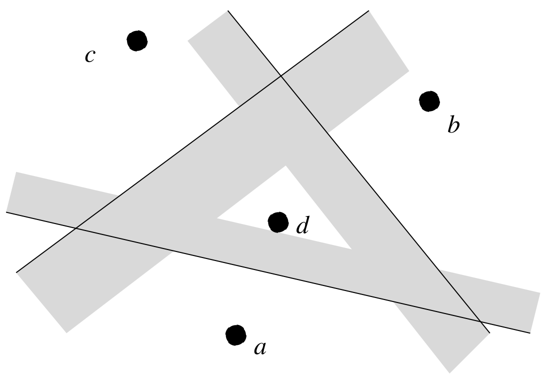


**Figure 10.28** The Voronoi region of $d$ is the intersection of three open halfspaces $VR(d, a)$, $VR(d, b)$, and $VR(d, c)$.

of $e$. Any such $p$ lies on the perpendicular bisector of $s$ and $t$ and hence $e$ is a straight line segment contained in the perpendicular bisector of $s$ and $t$.

Consider next any face $f$ of the Voronoi diagram and let $s$ be the site of $S$ such that $close(p) = \{s\}$ for all points $p$ of $f$. Then $dist(s, p) < dist(t, p)$ for all $t \in S \setminus \{s\}$ and hence $f$ is contained in the open halfplane bounded by the perpendicular bisector of $s$ and $t$ and containing $s$. We use $VR(s, t)$ to denote this halfplane, see Figure 10.28, and call it the halfplane where *$s$ dominates over $t$*. We have just shown that $f \subseteq VR(s, t)$ for all $t \in S \setminus \{s\}$ and hence

$$f \subseteq VR(s) := \bigcap_{t \in S \setminus \{s\}} VR(s, t).$$

We even have equality since $p \in VR(s)$ implies $p \in VR(s, t)$ for all $t \in S \setminus \{s\}$ which in turn implies that $p$ is closer to $s$ than to any other site in $S$. We call $VR(s)$ the *Voronoi region* of site $s$. It is the intersection of open halfspaces and hence an open convex polygonal region.

How are we going to represent Voronoi diagrams? We use plane maps of type

```
GRAPH<CIRCLE,POINT>.
```

We defined the Voronoi diagram $VD(S)$ as a set of points. We turn it into a graph $G$ by placing a "vertex at infinity" on every unbounded edge of $VD(S)$[7] and by deleting the portion of the edge that goes beyond the vertex at infinity, see Figure 10.29. A node $v$ of $G$ has either degree one or degree three or more. We call $v$ a node at infinity in the former case and a proper node in the latter case.

The node and edge labels give information about the positions of the node of $G$ in the plane and about the Voronoi regions:

- Every dart is labeled with the site whose region lies to its left.
- Every proper node $v$ is labeled by a circle $CIRCLE(a, b, c)$, where $a$, $b$, and $c$ are

[7] If all sites are collinear and hence $VD(S)$ consists of a set of parallel lines, we put two vertices at infinity on every line.

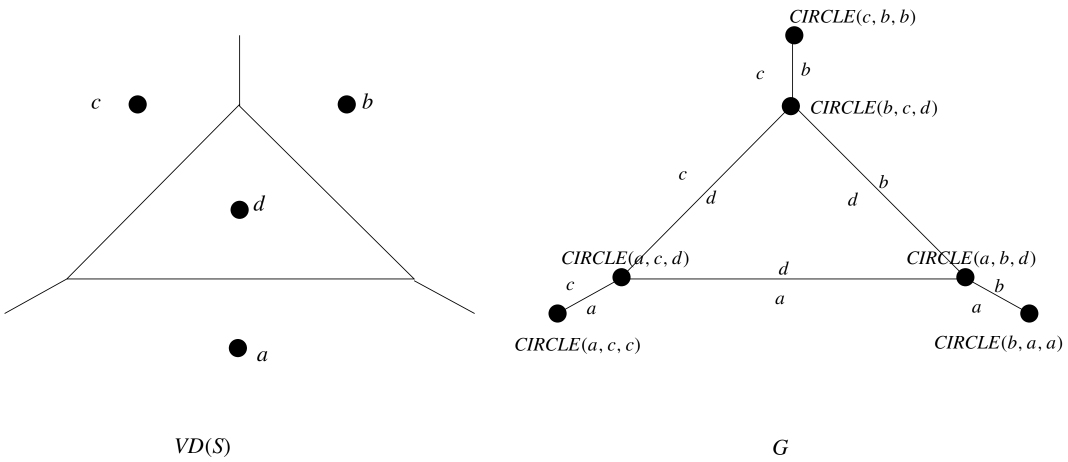


**Figure 10.29** A Voronoi diagram for a set of four sites and its graph representation.

distinct sites whose regions are incident to $v$. The center of this circle is the position of $v$ in the plane.

- Every node $v$ at infinity lies on the perpendicular bisector of two sites $a$ and $b$. We label $v$ by $\mathit{CIRCLE}(a, x, b)$, where $x$ is an arbitrary point collinear to $a$ and $b$ (e.g., $a$) and $v$ lies to the left of the oriented segment from $a$ to $b$.

The function

```
void VORONOI(const list<POINT>& L, GRAPH<CIRCLE,POINT>& VD);
```

computes the Voronoi diagram of the sites in $L$ in time $O(n \log n)$.

There is also a so-called *furthest site Voronoi diagram*, see Figure 10.30 for an example. Its definition is the same as for (nearest site) Voronoi diagrams except for replacing closest by furthest. For any point $p$ let $\mathit{furthest}(p)$ be the set of sites that realize the furthest distance between $p$ and the sites in $S$, i.e., $s \in \mathit{furthest}(p)$ if $\mathit{dist}(s, p) \geq \mathit{dist}(t, p)$ for all $t \in S$. In other words, there is a circle with center $p$ passing through all points in $\mathit{furthest}(p)$ and having no points of $S$ in its exterior. For most points $p$ of the plane $\mathit{furthest}(p)$ consists of only a single site. For some points $p$, $\mathit{furthest}(p)$ contains two or more sites. These points form the so-called furthest site Voronoi diagram $\mathit{FVD}(S)$ of $S$.

$$\mathit{FVD}(S) = \{p \in R^2; |\mathit{furthest}(p)| \geq 2\}.$$

The furthest site Voronoi region of a site $s$ is given by

$$\mathit{FVR}(s) := \bigcap_{t \in S \setminus \{s\}} \mathit{FVR}(t, s).$$

Only vertices of the convex hull have non-empty regions in the furthest site Voronoi digram. The rules for the graph representation of furthest site diagrams are the same as for nearest site diagrams.

The function

```
void F_VORONOI(const list<POINT>& L, GRAPH<CIRCLE,POINT>& FVD);
```

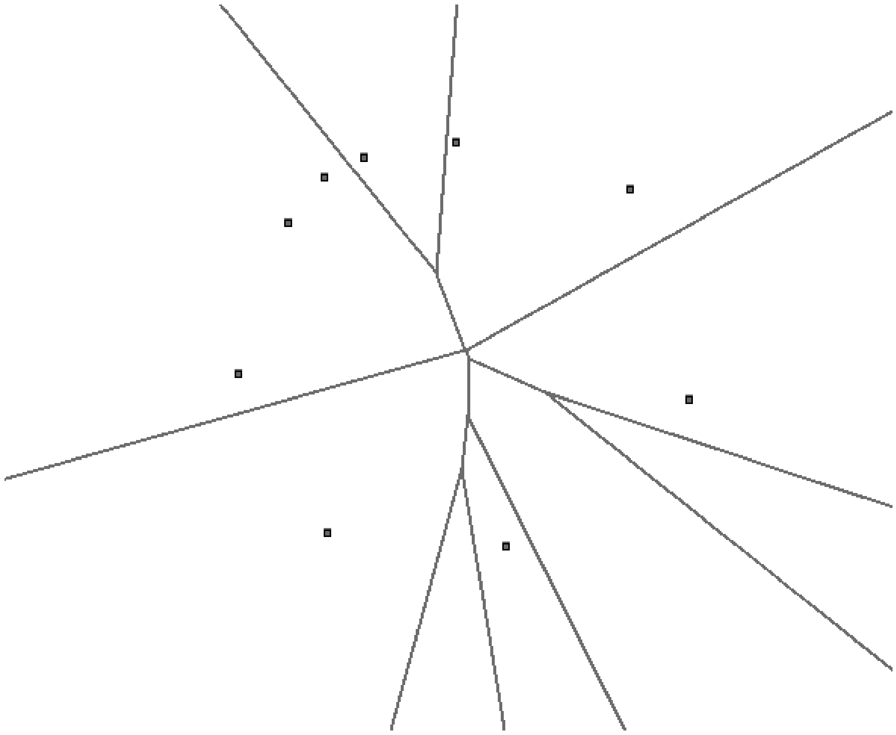

**Figure 10.30** A furthest site Voronoi diagram. The figure was generated with the voronoi_demo in xlman.

computes the furthest site Voronoi diagram of the points in $L$.

We recommend that the readers exercise the Voronoi demo in xlman before proceeding.

### 10.5.2 *The Duality between Voronoi and Delaunay Diagrams*

Voronoi diagrams and Delaunay diagrams are closely related structures. In fact, each one of them is easily obtained from the other. Let $S$ be a set of sites and let $VD(s)$ and $DD(S)$ be its Voronoi and Delaunay diagram, respectively. We show how to obtain $VD(S)$ from $DD(S)$.

**(1)** For every bounded face $f$ of $DD(S)$ there is a vertex $c(f)$ of $VD(S)$ located at the center of the circumcircle of $f$.

**(2)** Consider an edge $st$ of $DD(S)$ and let $f_1$ and $f_2$ be the faces incident to the two sides of the edge.

a) If $f_1$ and $f_2$ are both bounded, then the edge $c(f_1)c(f_2)$ belongs to $VD(S)$.

b) If $f_1$ is unbounded and $f_2$ is bounded, then a ray with source $c(f_2)$ and contained in the perpendicular bisector of $s$ and $t$ belongs to $VD(S)$. It extends into the halfplane containing the unbounded face.

c) If $f_1$ and $f_2$ are unbounded[8] and hence $f_1 = f_2$, then the entire perpendicular bisector of $s$ and $t$ belongs to $VD(S)$.

[8] Case $c$ arises only if all sites in $S$ are collinear. Then cases a) and b) do not arise.

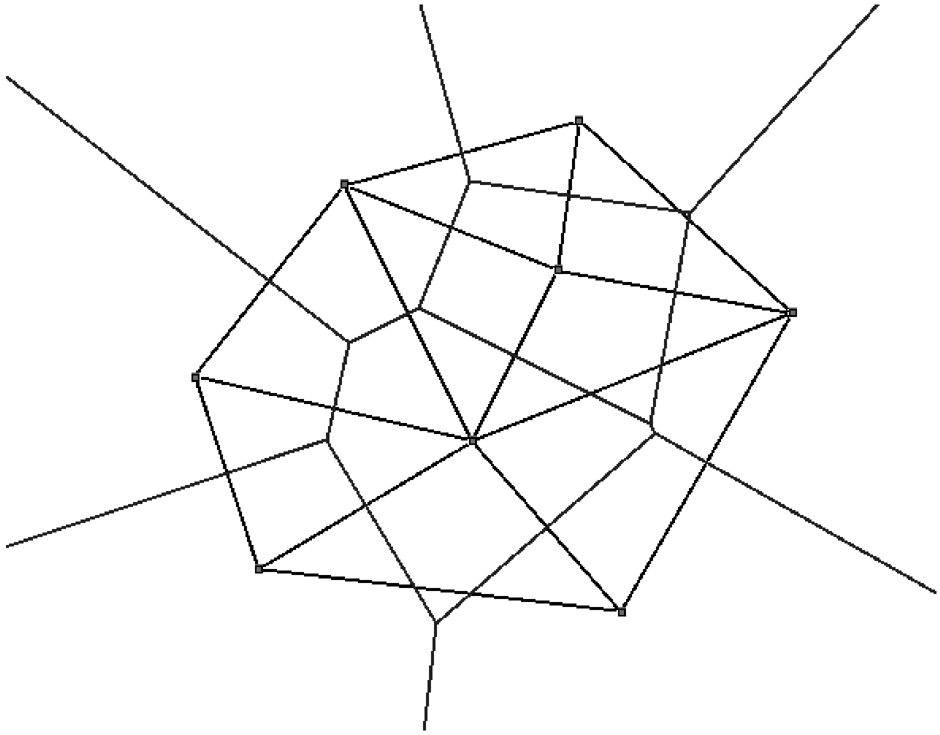

**Figure 10.31** A Voronoi diagram and a Delaunay diagram for the same set of sites. This figure was generated with the voronoi_demo in xlman.

**(3)** That's all.

Figure 10.31 shows a Delaunay and a Voronoi diagram for the same set of sites. Use the Voronoi demo to construct your own examples. The rules above are called a *duality* relation because they map faces (= 2-dimensional objects) into vertices (= 0-dimensional objects), edges into edges, and vertices into faces. The latter map is implicit. There is a corresponding set of rules that construct the Delaunay diagram from the Voronoi diagram. We leave them to the exercises.

**Theorem 2** *The rules above construct the Voronoi diagram from the Delaunay diagram.*

*Proof* We proceed in two steps. We first show that everything that is constructed by the rules does indeed belong to the Voronoi diagram and in a second step we show that the complete Voronoi diagram is obtained.

Consider any bounded face $f$ of $DD(S)$. The vertices of $f$ are co-circular and hence the circumcenter $c(f)$ is a point with $|close(p)| \geq 3$, i.e., a vertex of $VD(S)$.

Consider next any edge $st$ of $DD(S)$. View it as oriented from $s$ to $t$ and let $f_1$ and $f_2$ be the faces to its left and right, respectively. Assume first that $f_1$ and $f_2$ are both bounded. The centers $c(f_1)$ and $c(f_2)$ of the circumcircles of $f_1$ and $f_2$ both lie on the perpendicular bisector of $s$ and $t$ and any point between $c(f_1)$ and $c(f_2)$ is the center of a disk $D$ with $D \cap S = \{s, t\}$, see Figure 10.32. Thus, $c(f_1)c(f_2)$ is an edge of $VD(S)$.

Assume next that $f_1$ is unbounded and $f_2$ is bounded, i.e, $st$ is a clockwise convex hull

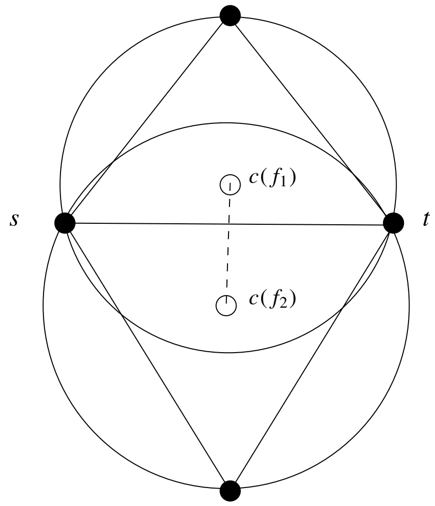


**Figure 10.32** An edge $e = (s, t)$ of $DD(S)$, the two incident faces $f_1$ and $f_2$ and the circumcircles of $f_1$ and $f_2$. Each point on the open line segment $c(f_1)c(f_2)$ is the center of an empty circle passing through $s$ and $t$.

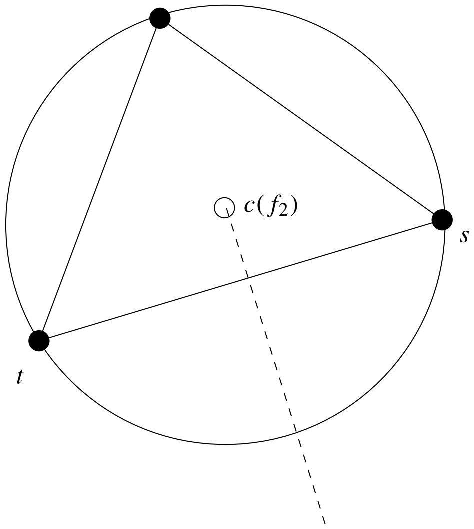


**Figure 10.33** $st$ is a clockwise convex hull edge and the face $f_2$ to its right is bounded.

edge, see Figure 10.33. Then the same argument shows that the ray starting in $c(f_2)$, contained in the perpendicular bisector of $s$ and $t$, and extending into the left halfplane with respect to $st$ belongs to $VD(S)$.

Finally, if $f_1$ and $f_2$ are both unbounded then the entire perpendicular bisector of $s$ and $t$ is an edge of $VD(S)$.

We have now shown that the rules above construct only features of the Voronoi diagram. We show next that the entire Voronoi diagram is constructed. Consider any edge $e$ of $VD(S)$, say separating the regions $VR(s)$ and $VR(t)$. Then $close(p) = \{s, t\}$ for every point $p \in e$, i.e., every $p \in e$ witnesses that the segment $st$ is essential and hence is an edge of $DD(S)$. Imagine a disk centered at $p$ and having $s$ and $t$ in its boundary as $p$ moves along $e$. When

$p$ moves into an endpoint of $e$ ($e$ may have 0, 1, or 2 endpoints), *close*($p$) grows to at least three points, namely the vertices of a face of *DD*($S$) incident to *st*. Thus, applying the appropriate rule 2a, 2b, or 2c to *st* yields $e$. Moreover, applying rule 1 to the bounded faces incident to *st* produces the endpoints of $e$ (if any). We have now shown that all edges of *VD*($S$) are constructed and since every vertex of *VD*($S$) is incident to at least one (actually three) edge we have also shown that all vertices are constructed. □

We next give the program that constructs a Voronoi diagram from a Delaunay diagram. The Voronoi diagram is empty if the number of sites is less than two. So assume that there are at least two sites. We first determine a hull edge, then create all nodes of the Voronoi diagram and finally all darts of the Voronoi diagram. We use an edge array *vnode* in order to associate with each dart $e$ of *DD* the node of *VD* that lies in the face to the left of $e$.

⟨*voronoi.c*⟩≡

```
void DELAUNAY_TO_VORONOI(const GRAPH<POINT,int>& DD,
                         GRAPH<CIRCLE,POINT>& VD)
{
  VD.clear();
  if (DD.number_of_nodes() < 2) return;
  // determine a hull dart
  edge e;
  forall_edges(e,DD) if (DD[e] == HULL_DART) break;
  edge hull_dart = e;
  edge_array<node> vnode(DD,nil);
  ⟨DD to VD: create Voronoi nodes⟩
  ⟨DD to VD: create Voronoi darts⟩
}
```

We create the Voronoi nodes in two phases. We first create the nodes at infinity and then the proper nodes.

There is one node at infinity for each hull dart. If $e$ is a hull dart and $a$ and $b$ are the sites associated with the source and target of $e$, respectively, then the label of the node at infinity is *CIRCLE*$(a, x, b)$, where $x$ is any point collinear with $a$ and $b$. We use the midpoint of $a$ and $b$ for $x$.

If $e$ is not a hull dart then there is a proper node $v$ associated with the face cycle of $e$. We label $v$ with *CIRCLE*$(a, b, c)$, where $a$, $b$, and $c$ are any three vertices of the face cycle, and associate $v$ with every dart of the face cycle.

⟨*DD to VD: create Voronoi nodes*⟩≡

```
// create Voronoi nodes for outer face
POINT a = DD[source(e)];
do { POINT b = DD[target(e)];
     vnode[e] =  VD.new_node(CIRCLE(a,center(a,b),b));
     e = DD.face_cycle_succ(e);
     a = b;
   } while ( e != hull_dart );
```

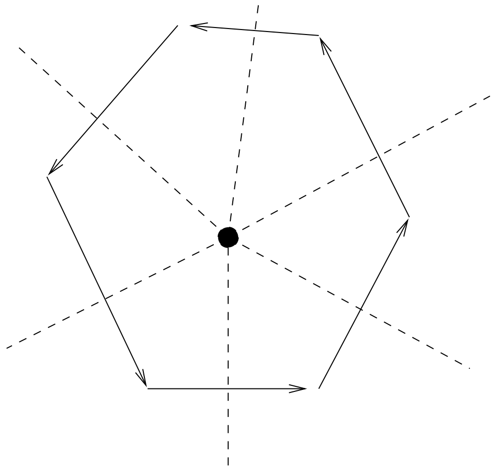

**Figure 10.34** Tracing a face cycle in forward direction generates the darts incident to the node dual to the face in counter-clockwise order.

```
// and for all other faces
forall_edges(e,DD)
{ if (vnode[e]) continue;
  edge  x = DD.face_cycle_succ(e);
  POINT a = DD[source(e)];
  POINT b = DD[target(e)];
  POINT c = DD[target(x)];
  node  v = VD.new_node(CIRCLE(a,b,c));
  vnode[e] = v;
  do { vnode[x] = v;
       x = DD.face_cycle_succ(x);
     } while( x != e );
}
```

We come to the construction of the Voronoi darts. Let $e$ be a dart of $DD$, let $r$ be its reversal, and let $p$ be the point associated with the target of $e$. The dart dual to $e$ starts at the node associated with $e$, ends at the node associated with $r$, and is labeled by $p$.

We want to construct the darts incident to any node of $VD$ in their proper counter-clockwise order. For the nodes at infinity this is no problem since they have degree one. We therefore construct the Voronoi darts in two phases. We first construct the Voronoi darts out of the nodes at infinity and then the Voronoi darts out of the proper nodes. Finally, we link the two darts in each. For each dart $e$ of $DD$ we record the dart dual to it in the edge array *vedge*.

Consider a proper node $v$. It corresponds to a bounded face of $DD$ and has one incident dart for each dart of the face cycle. We construct the darts in their proper counter-clockwise order if we trace the face cycle in forward direction, see Figure 10.34.

⟨*DD to VD: create Voronoi darts*⟩≡

```
edge_array<edge> vedge(DD,nil);
// construct Voronoi darts out of nodes at infinity
e = hull_dart;
do { edge r = DD.reversal(e);
     POINT p = DD[target(e)];
     vedge[e] = VD.new_edge(vnode[e],vnode[r],p);
     e = DD.cyclic_adj_pred(r); // same as DD.face_cycle_succ(e)
   } while ( e != hull_dart );
// and out of all other nodes.
forall_edges(e,DD)
{ node v = vnode[e];
  if (VD.outdeg(v) > 0) continue;
  edge x = e;
  do { edge  r = DD.reversal(x);
       POINT p = DD[target(x)];
       vedge[x] = VD.new_edge(v,vnode[r],p);
       x = DD.cyclic_adj_pred(r);
     } while ( x != e);
}
// assign reversal edges
forall_edges(e,DD)
{ edge r = DD.reversal(e);
  VD.set_reversal(vedge[e],vedge[r]);
}
```

This completes the construction of Voronoi diagrams from Delaunay diagrams. The construction runs in linear time.

In order to construct the Voronoi diagram for a set $L$ of points we first construct the Delaunay diagram and then the Voronoi diagram from the Delaunay diagram.

⟨*voronoi.c*⟩+≡

```
void VORONOI(const list<POINT>& L, GRAPH<CIRCLE,POINT>& VD)
{ GRAPH<POINT,int> DD;
  DELAUNAY_DIAGRAM(L,DD);
  DELAUNAY_TO_VORONOI(DD,VD);
}
```

The construction of furthest site Voronoi diagrams from furthest site Delaunay triangulations is completely analogous. We leave it to the exercises.

### 10.5.3 *Verifying Voronoi Diagrams*

Let $G$ be a graph of type *GRAPH<CIRCLE*, *POINT>*. We want to verify that $G$ is the Voronoi diagram of the sites associated with its nodes. The procedure to be described is fairly complicated and we wished we had a simpler one. The procedure is probably the least elegant piece of code contained in this book. We considered to drop this section, but decided against it for two reasons. We had invested a lot of time in it, and more importantly, the check discovered several mistakes.

$G$ must satisfy the following conditions:

- $G$ is a CCW-ordered plane map.
- The site information associated with edges is consistent, i.e., if $e$ and $e'$ are consecutive edges on some face cycle then both edges have the same associated site.
- The sites associated with $e$ and *reversal*$(e)$ are distinct.
- Call a vertex whose associated circle is non-degenerate non-trivial and call it trivial otherwise. Every non-trivial vertex has degree at least three and every trivial vertex has degree one.
- For each non-trivial vertex each of the three points defining the associated circle is associated with one of the incident edges and the sites associated with all incident edges lie on the associated circle.
- Each trivial vertex has an associated circle of the form *CIRCLE*$(a, \_, c)$, where $a$ and $c$ are distinct. Let $e$ be the unique outgoing edge. In a nearest site diagram the site associated with the face to the left of $e$ is $c$ and the site associated with the face to the right of $e$ is $a$ and in a furthest site diagram the roles of $a$ and $c$ are interchanged.
- For every edge $e = (v, w)$ such that $v$ and $w$ are non-trivial, the centers of the circles associated with $v$ and $w$ are distinct. Let $p$ and $q$ be these centers and let $a$ be the site associated with $e$. In a nearest site diagram $a$ lies to the left of the segment $pq$ and in a furthest site diagram $a$ lies to the right of the segment $pq$.
- Each face is a convex polygonal region and the regions associated with the different sites partition the plane.

In the implementation we first check the first six conditions and then distinguish cases according to whether $G$ is connected or not. For the first item we want to use the function *Is_CCW_Ordered_Plane_Map* and therefore we need to define the *edge_vector* function for circle-points. Let $e$ be an edge and let $C$ and $D$ be the circles associated with the source and the target of $e$, respectively. If both circles are non-degenerate the edge vector is simply the vector from the center of $C$ to the center of $D$. So assume that one of the circles is degenerate. If $D$ is degenerate then $D$ = *CIRCLE*$(a, \_, c)$ and $D$ represents a point at infinity on the perpendicular bisector of $a$ and $c$ and to the right of the line segment $ac$. Let $m$ be the midpoint of $a$ and $c$ and let $a_1$ be the point obtained by rotating $a$ by $90^\circ$ in a clockwise direction about $m$. We may return the vector $m - a_1$. The case that $C$ is degenerate is symmetric.

⟨*voronoi_check: edge vector function*⟩≡

```
static VECTOR edge_vector(const GRAPH<CIRCLE,POINT>& G, const edge& e)
{ const CIRCLE& C = G[G.source(e)];
  const CIRCLE& D = G[G.target(e)];
  if ( D.is_degenerate() ) { POINT a = D.point1();
                             POINT c = D.point3();
                             POINT m = midpoint(a,c);
                             return m - a.rotate90(m);
                           }
  if ( C.is_degenerate() ) { POINT a = C.point1();
                             POINT c = C.point3();
                             POINT m = midpoint(a,c);
                             return a.rotate90(m) - m;
                           }
  // both circles are non-degenerate
  return D.center() - C.center();
}
```

and

⟨*voronoi_check.c*⟩+≡

⟨*voronoi_check: edge vector function*⟩

```
static bool False_IVD(const string& s)
{ cerr << "Is_Voronoi_Diagram: " << s; return false; }

bool Is_Voronoi_Diagram(const GRAPH<CIRCLE,POINT>& G,
                        delaunay_voronoi_kind kind)
{ if ( G.number_of_nodes() == 0 ) return true;
  node v,w; edge e;

  if ( !Is_CCW_Ordered_Plane_Map(G) )
     return False_IVD("G is not CCW-ordered plane map");

  forall_edges(e,G)
  { if ( G.outdeg(target(e)) != 1 )
    { // e does not end at a vertex at infinity
      if ( G[e] != G[G.face_cycle_succ(e)] )
        return False_IVD("inconsistent site labels");
    }
    if ( G[e] == G[G.reversal(e)] )
        return False_IVD("same site on both sides");
  }

  forall_nodes(v,G)
  { CIRCLE C = G[v];
    if ( C.is_degenerate() )
    { // vertex at infinity

      if ( G.outdeg(v) != 1 )
        return  False_IVD("degree of vertex at inf");

      edge e = G.first_adj_edge(v); edge r = G.reversal(e);
      POINT a = C.point1(); POINT c = C.point3();
      if ( (kind == NEAREST) && (c != G[e] || a != G[r]) ||
           (kind == FURTHEST) && (a!= G[e] || c != G[r]) )
        return False_IVD("vertex at inf: wrong edge labels");
```

```
    }
    else
    { // finite vertex
      if ( G.outdeg(v) < 3 )
        return False_IVD("degree of proper vertex");
      forall_adj_edges(e,v)
      { if ( !C.contains(G[e]) )
          return False_IVD("label of proper vertex");
      }
      for (int i = 1; i <= 3; i++)
      {
        POINT a = ( i == 1 ? C.point1() :
                   (i == 2 ? C.point2() : C.point3() ) );
        bool found_a = false;
        forall_adj_edges(e,v) if ( a == G[e] ) found_a = true;
        if ( !found_a ) return False_IVD("wrong cycle");
      }
      forall_adj_edges(e,v)
      { w = G.target(e);
        if ( G.outdeg(w) == 1 ) continue;
        if ( C.center() == G[w].center() )
          return False_IVD("zero length edge");
        int orient = orientation(C.center(),G[w].center(),G[e]);
        if ( kind == NEAREST && orient <= 0 ||
             kind == FURTHEST && orient >= 0 )
          return False_IVD("orientation");
      }
    }
  }
  if ( Is_Connected(G) )
    { ⟨G is connected⟩ }
  else
    { ⟨G is not connected⟩ }
  return true;
}
```

When $G$ has passed all tests above we can construct a geometric object from it as follows. We assign a position $pos(v)$ to each non-trivial vertex $v$ and a segment, ray, or line $geo(e)$ to each edge $e$. For a non-trivial vertex $v$ let $pos(v)$ be the center of the circle associated with $v$. For an edge $e = (v, w)$ let $a$ and $c$ be the sites separated by $e$, i.e., one of $a$ and $c$ is associated with $e$ and the other with $reversal(e)$. If $v$ is non-trivial then $a$ and $c$ lie on the circle associated with $v$ and hence $pos(v)$ lies on the perpendicular bisector of $a$ and $b$. Define $geo(e)$ as follows. First assume that $v$ and $w$ are both non-trivial. Then $geo(e)$ is the segment directed from $pos(v)$ to $pos(w)$. Note that this segment has non-zero length and is part of the perpendicular bisector of $a$ and $c$. Next assume that exactly one of $v$ and $w$ is non-trivial. Assume w.l.o.g. that the triple of points associated with the trivial vertex is of the form $(a, \_, c)$. If $w$ is trivial then $geo(e)$ is the ray starting at $pos(v)$, running along the perpendicular bisector of $a$ and $c$, and extending to infinity to the right of the segment

$ac$. If $w$ is trivial then *geo*$(e)$ is the ray ending in *pos*$(v)$, running along the perpendicular bisector of $a$ and $c$, and coming from infinity to the right of the segment $ac$. Finally, assume that $v$ and $w$ are trivial and assume w.l.o.g. that the triple of points associated with $v$ is of the form $(a, \_, c)$. Then *geo*$(e)$ is the bisector of $a$ and $c$ oriented such that $a$ lies to its left.

Now we distinguish cases according to whether $G$ is connected or not.

**$G$ is connected:** Define a face chain as a minimal sequence $e_0, e_1, \ldots, e_k$ of edges such that $e_{i+1}$ is the face cycle successor of $e_i$ for all $i$, $0 \le i < k$, and either $target(e_k) = source(e_0)$ or $source(e_0)$ and $target(e_0)$ have degree one. We call face chains of the former kind closed and face chains of the latter kind open. All face chains are strictly convex counter-clockwise oriented. Moreover, the rays going to infinity wind around the origin once and open face chains cover only a half-circle. There is no need to check the second half-sentence as it is implied by the first half-sentence.

Below, we first search for a vertex of degree one and then check the open face chains one by one. Simultaneously we build the list of all rays; note that they will wind clockwise around the origin. Having checked all open face chains we turn to the closed face chains.

⟨*G is connected*⟩≡

```
cmp_edges_by_angle<GRAPH<CIRCLE,POINT> > cmp(G);
node v;
forall_nodes(v,G) if ( G.outdeg(v) == 1 ) break;
edge_array<bool> considered(G,false);
list<edge> rays;
edge e = G.first_adj_edge(v);
do { rays.push(e);
     list<edge> D;
     do { considered[e] = true;
          D.append(e);
          e = G.face_cycle_succ(e);
        } while ( G.outdeg(source(e)) != 1);
     if ( !Is_C_Increasing(D,cmp)  ) return False_IVD(": wrong order");
} while ( G.source(e) != v);
if ( !Is_C_Nondecreasing(rays,cmp) )
  return False_IVD("wrong order, rays");
forall_edges(e,G)
{ if ( !considered[e] )
  { edge e0 = e;
    do { considered[e] = true;
         if ( G.outdeg(target(e)) == 1 )
           return False_IVD("unexpected vertex of degree one");
         e = G.face_cycle_succ(e);
       } while ( e != e0);
    if ( !Is_CCW_Convex_Face_Cycle(G,e) )
      return False_IVD("wrong order");
  }
}
```

We claim that we are done at this point. Let us see why. Consider any face chain $f$. All edges on the boundary of $f$ have the same associated site, say $a$, the circles associated with all non-trivial vertices of $f$ pass through $a$, for each edge $e$ of $f$, $geo(e)$ is part of the perpendicular bisector of $a$ and the site associated with the other side of $e$, and $a$ lies to the left of $geo(e)$ if *kind* is *NEAREST* and to the right of it if *kind* is *FURTHEST*. Define

$$reg(f) = \bigcap_{e;\ e \text{ is an edge of } f} H(a, site\_of\_reversal(e)),$$

where $b = site\_of\_reversal(e)$ is the site associated with the reversal of $e$ and $H(a,b)$ is the halfplane defined by $a$ and $b$ and containing $a$ if *kind* is *NEAREST* and not containing $a$ otherwise. Then $reg(f)$ is a convex polygonal region which contains the Voronoi region of site $a$ (since in the definition of a Voronoi region the intersection is over all sites different from $a$). We still need to show that the regions partition the plane. Consider a point moving in the plane and avoiding vertices of regions. Such a point is always covered by the same number of regions. Moreover, when the point travels along a cycle at infinity it is always covered by exactly one region since the rays of the diagram wind around the origin once. Altogether we have shown that the regions partition the plane.

**$G$ is not connected:** If $G$ is not connected it can only be the Voronoi diagram of a set of collinear sites. As such it must have the following additional properties:

- All nodes have out-degree one.
- All sites are collinear.
- No site is associated with three edges of $G$.
- The number of distinct sites is equal to $m/2 + 1$.

We show that these conditions suffice. Clearly, the geometric interpretation of $G$ is a set of parallel line segments. Consider the placement of the sites on their common underlying line. For each site $s$ which is associated with two edges, it is guaranteed that the two adjacent sites (= sites for which there is an edge having $s$ on one of its sides) lie on opposite sides of $s$; this follows from the fact that we have already checked that each edge incident to a trivial node separates the sites it is supposed to separate. We conclude that the conditions above suffice.

⟨*G is not connected*⟩≡

```
forall_nodes(v,G)
  if ( G.outdeg(v) > 1 ) return False_IVD("degree larger than 1");
d_array<POINT,int> count(0);
int n_dual = 0;
edge e = G.first_edge();
LINE l(G[e],G[G.reversal(e)]);
forall_edges(e,G)
{ if ( !l.contains(G[e]) )
```

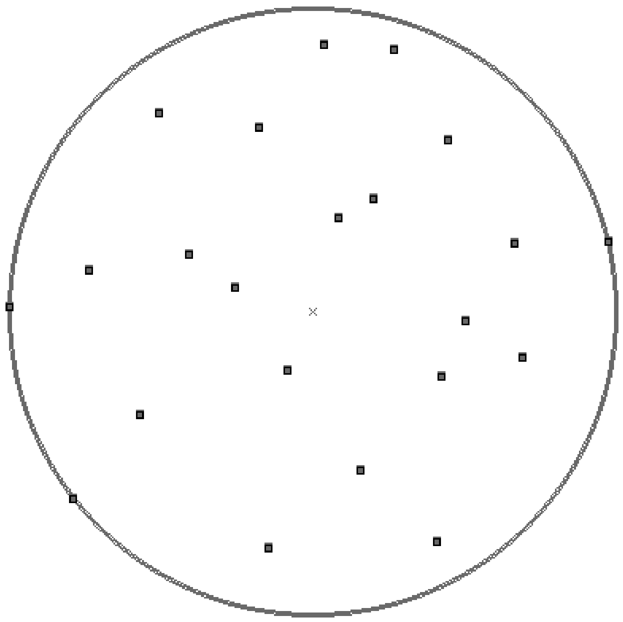

**Figure 10.35** The smallest circle enclosing a set of points. The figure was generated with the voronoi_demo in xlman.

```
      return False_IVD("non-collinear sites");
    int& pc = count[G[e]];
    if (pc == 0) n_dual++;
    pc++;
    if (pc == 3) return False_IVD(": site mentioned thrice");
  }
  if ( n_dual != (G.number_of_edges()/2 + 1) )
    return False_IVD(": two many sites");
```

### 10.5.4 *Applications of Voronoi Diagrams*

We discuss some applications of Voronoi diagrams. All of them are illustrated in the voronoi-demo of xlman.

**Extremal Circles:** The *smallest enclosing circle* for a set $L$ of points is the circle with the smallest radius containing all points in $L$, see Figure 10.35. The smallest enclosing circle is the best approximation of $L$ by a circle. It is easy to see that such a circle has at least two points in $L$ on its boundary and hence its center lies on the furthest site Voronoi diagram of $L$.

We conclude that the center of the minimum enclosing circle is either a vertex of the furthest site diagram (and then has three points in $L$ on its boundary) or lies on an edge of the furthest site diagram (and then is the circle of minimum radius passing through the two sites defining the edge). In this way each edge and vertex of the furthest site Voronoi

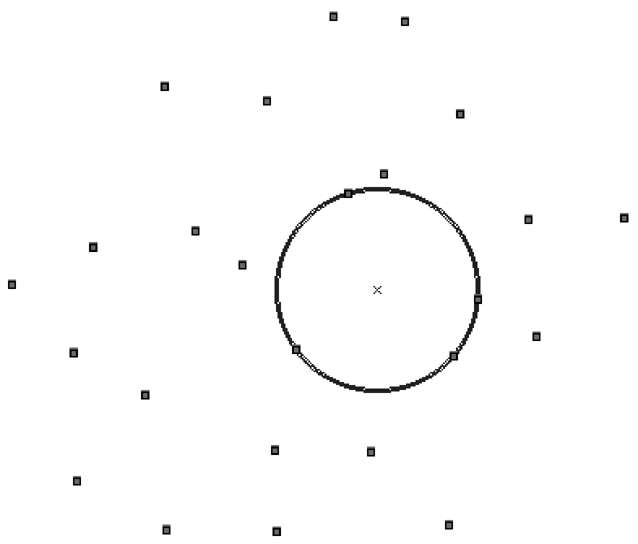

**Figure 10.36** The largest empty circle for a set of points. The figure was generated with the voronoi_demo in xlman.

diagram defines a candidate circle. The minimum enclosing circle is the smallest of these circles.

The function

```
CIRCLE SMALLEST_ENCLOSING_CIRCLE(const list<POINT>& L);
```

computes a smallest enclosing circle according to the strategy just described.

The *largest empty circle* for a set $L$ of points is the circle with the largest radius whose interior is void of points in $L$ and whose center lies inside the convex hull of $L$, see Figure 10.36. We know of no good motivation for considering largest empty circles. It is easy to see that such a circle has at least two points in $L$ on its boundary and hence its center lies on the nearest site Voronoi diagram of $L$.

We conclude that the center of the largest empty circle is either a vertex of the nearest site diagram (and then has three points in $L$ on its boundary) or lies on an edge of the nearest site diagram (and then is the circle of maximum radius passing through the two sites defining the edge and having its center inside the convex hull). In this way each edge and vertex of the nearest site Voronoi diagram defines a candidate circle. The largest empty circle is the largest of these circles.

The function

```
CIRCLE LARGEST_EMPTY_CIRCLE(const list<POINT>& L);
```

computes a largest empty circle according to the strategy just described.

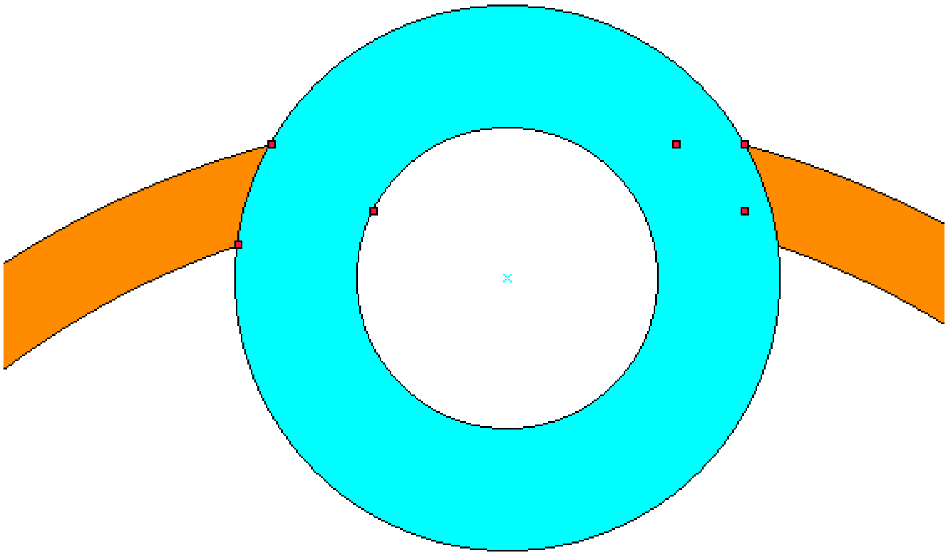

**Figure 10.37** The minimum width and the minimum area annulus for a set of points. The figure was generated with the voronoi_demo in xlman.

The application of Voronoi diagrams to find enclosing and empty circles is due to Shamos and Hoey ([SH75]).

**Minimum Width and Minimum Area Annuli:** An *annulus* $A$ is the region between two concentric circles. When the common center of the circles is a point at infinity, an annulus degenerates to a stripe between parallel lines. Annuli are closed sets. An annulus covers a set $L$ of points if all points in $L$ are contained in the annulus. The *width* of an annulus is the difference between the radius of the outer circle and the radius of the inner circle of the annulus (in the case of a stripe the width is the distance between the two boundaries of the stripe). The *area* of an annulus is the area of the region between the outer and the inner circle (it is infinite in the case of a stripe of non-zero width and is zero in the case of a stripe of width zero). We are interested in computing minimum width and minimum area annuli covering a given set $L$ of points, see Figure 10.37 for an example. Minimum width and minimum area annuli are used to estimate the "roundness" of a set of points.

It can be shown that there is always a minimum annulus covering a given set $L$ of points that is either:

- the minimum width stripe covering the points, or
- a pair of concentric circles whose center is either a vertex of the nearest site Voronoi diagram, or a vertex of the furthest site diagram, or an intersection between an edge of

the nearest site diagram and an edge of the furthest site diagram. This observation was made in [SH75].

The idea for the proof is as follows. Consider an annulus covering the points in $L$. Clearly, if one of the boundaries does not contain a point in $L$ then the annulus can be improved. So both boundaries must contain at least one point in $L$. If the two boundaries together contain a total of four points of $L$ then the center of the annulus is either a vertex of one the diagrams (if one boundary contains three points and the other contains one) or an intersection between edges (if both boundaries contain two points). So assume that the boundaries together contain less than four points, say there are two points $p$ and $q$ on one of the boundaries and one point $r$ on the other boundary. Then the center $c$ lies on the perpendicular bisector of $p$ and $q$. Let $d$ be a vector in the direction of the perpendicular bisector and consider the annulus $A(\epsilon)$ with center $c + \epsilon \cdot d$ and having $p$, $q$ and $r$ on its boundaries. For small enough $\epsilon$, $A(\epsilon)$ covers $L$. Consider the optimization criterion as a function of $\epsilon$ and conclude that the center can be moved either in the direction $+d$ or the direction $-d$ without increasing the objective value. Move until a further point lies on one of the boundaries. For example, if the objective value is the area, the area of $A(\epsilon)$ is proportional to

$$dist(p, c + \epsilon \cdot d)^2 - dist(r, c + \epsilon \cdot d)^2 = (p - c)^2 - (r - c)^2 + 2\epsilon(p - r) \cdot d,$$

i.e., is a linear function of $\epsilon$. If $(p-r) \cdot d \neq 0$ then the annulus can be improved by moving the center, and if $(p - r) \cdot d = 0$ then the center can be moved in either direction without increasing the area of the annulus.

The two items above suggest a strategy to compute minimum width and minimum area annuli. One simply checks all the candidates listed. This results in quadratic algorithms.

The functions

```
bool MIN_AREA_ANNULUS(const list<POINT>& L, POINT& center,
                      POINT& ipoint, POINT& opoint, LINE&  l1);
bool MIN_WIDTH_ANNULUS(const list<POINT>& L, POINT& center,
                       POINT& ipoint, POINT& opoint,
                       LINE& l1, LINE& l2);
```

compute minimum area and minimum width annuli covering the points in $L$, respectively. The functions return *true*, if the optimal annulus is the region between two circles, and return *false* if the optimal annulus is a stripe. In the former case the center of the annulus and a point on the inner and the outer circle are returned in *center*, *ipoint* and *opoint*, respectively. In the latter case the boundaries of the stripe are returned in *l1* and *l2*. In the case of the a minimum area annulus a stripe can only be optimal if it has width zero. Hence only one line is returned in the former function.

Both functions have quadratic running time and hence should be used only for small input size. There are much faster algorithms: the minimum area annulus can be computed in linear time by linear programming ([Sei91]) and the minimum width annulus can be computed in time $O(n^{8/5+\epsilon})$ by parametric search ([AST94]).

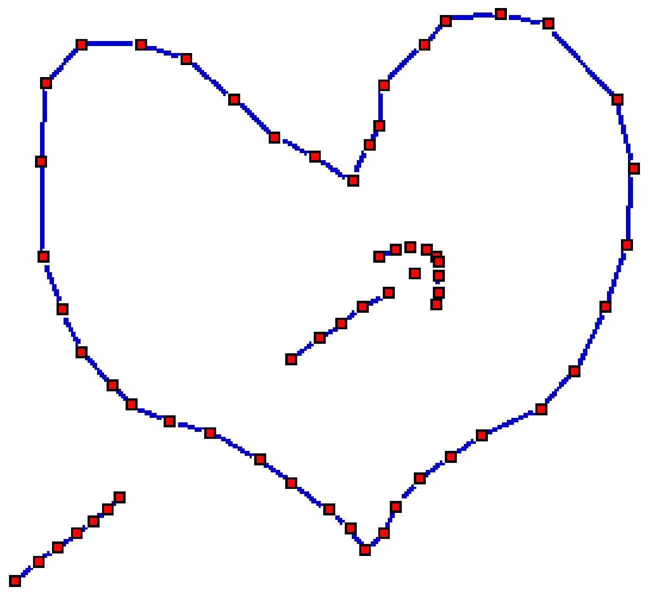

**Figure 10.38** A set of points in the plane and the curve reconstructed by *CRUST*. The figure was generated by the Voronoi-demo in xlman.

**Curve Reconstruction:** The reconstruction of a curve from a set of sample points is an important problem in computer vision. We describe a reconstruction algorithm due to Amenta, Bern, and Eppstein [ABE98]. Figure 10.38 shows a point set and the curves reconstructed by their algorithm.

The precise problem formulation is as follows. Let $F$ be a smooth curve in the plane and let $S \subset F$ be a finite set of sample points from $F$. A *polygonal reconstruction* of $F$ is a graph that connects every pair of samples adjacent along $F$, and no others.

The algorithm *CRUST* to be described takes a list $S$ of points and returns a graph $G$. The graph $G$ is guaranteed to be a polygonal reconstruction of $F$ if $F$ is sufficiently densely sampled by $S$. We refer the reader to [ABE98] to the definition of sufficient dense sampling density.

The algorithm proceeds in three steps:

- It first constructs the Voronoi diagram *VD* of the points in $S$.
- It then constructs a set $L = S \cup V$, where $V$ consists of all proper vertices of *VD*.
- Finally, it constructs the Delaunay triangulation *DT* of $L$ and makes $G$ the graph of all edges of *DT* that connect points in $L$.

The algorithm is very simple to implement[9].

[9] In 1997 the authors attended a conference, where Nina Amenta presented the algorithm. We were supposed to give a presentation of LEDA later in the day. We started the presentation with a demo of algorithm *CRUST*.

⟨*crust.c*⟩+≡

```
void CRUST(const list<POINT>& S, GRAPH<POINT,int>& G)
{
  list<POINT> L = S;
  map<POINT,bool> voronoi_vertex(false);
  GRAPH<CIRCLE,POINT> VD;
  VORONOI(L,VD);
  // add Voronoi vertices and mark them
  node v;
  forall_nodes(v,VD)
  { if (VD.outdeg(v) < 2) continue;
    POINT p = VD[v].center();
    voronoi_vertex[p] = true;
    L.append(p);
  }
  DELAUNAY_TRIANG(L,G);
  forall_nodes(v,G)
    if (voronoi_vertex[G[v]]) G.del_node(v);
}
```

The program above owes much of its elegance to the fact that we use graphs to represent Delaunay diagrams and hence have the full power of the graph data type available to us. Observe that after having constructed the Delaunay triangulation of $L$ in $G$, we treat $G$ as an “ordinary graph”. We simply delete all auxiliary nodes from it, a step that does not make sense on the level of Delaunay triangulations.

### 10.5.5 *Voronoi Diagrams of Line Segments*

The Voronoi diagram of a set of point sites under the Euclidian metric is just one instance in a wide class of Voronoi diagrams. Other diagrams are obtained by choosing a different metric and/or a different class of sites.

Figure 10.39 shows a Voronoi diagram of line segments. In such a diagram the sites are points and open line segments; the endpoints of every line segment must belong to the point sites. The edges of a Voronoi diagram of line segments are part of angular bisectors between line segments, of parabola, and of lines perpendicular to segments at their endpoints.

Michael Seel [See97] has written a package to compute Voronoi diagrams of line segments. It is available as a LEDA extension package.

The Voronoi diagram of line segments has played an important role in the development of the number types in LEDA, see Section 4.4. Our first program for Voronoi diagrams of line segments used floating point arithmetic in a naive way and worked only for a small number of examples. The main difficulty was a correct implementation of the incircle test. Observe that the coordinates of Voronoi vertices are non-rational algebraic numbers and hence the incircle test requires to compute the sign of certain algebraic numbers. This computation is very error-prone when executed with floating point arithmetic.

In [Bur96, BMh94, BFMh97] we laid the theoretical basis for an efficient and correct

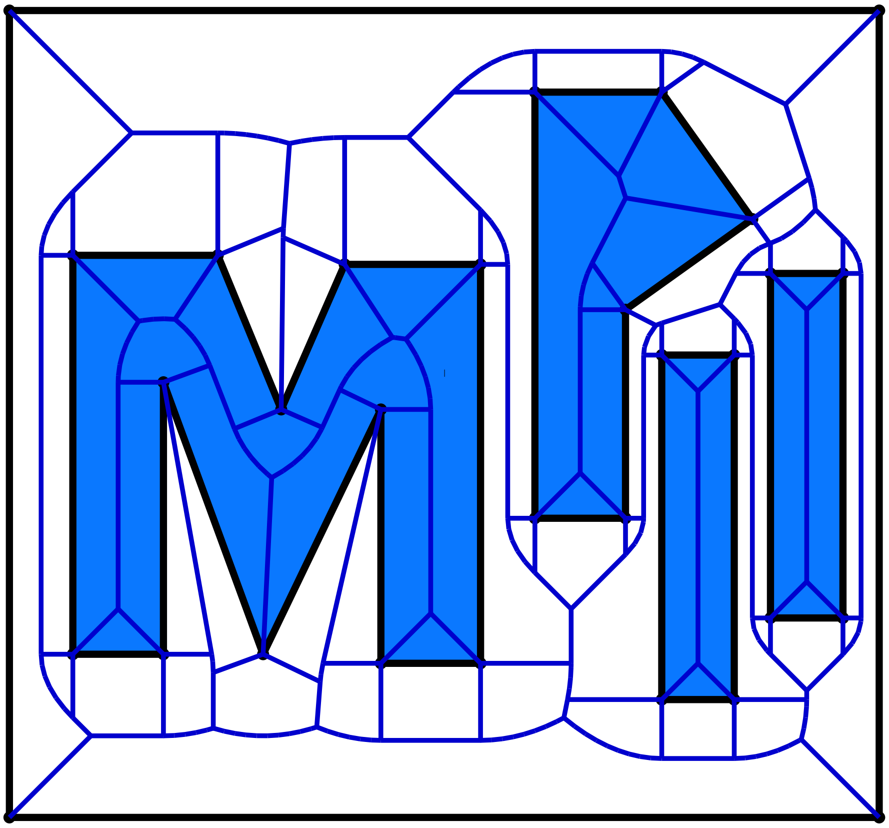

**Figure 10.39** A Voronoi diagram of line segments. The figure was generated with Michael Seel's extension package for Voronoi diagrams of line segments.

sign test of simple algebraic numbers which is used in [BMh96] to implement the number type *real*. Michael Seel uses this number type in his implementation.

### *Exercises for 10.5*

1. Construct a set $S$ where the Voronoi diagram contains no vertices and $S$ has at least three points. What is the Delaunay diagram of $S$?
2. Give the rules for obtaining the Delaunay diagram from the Voronoi diagram for the same set of sites.
3. Write a program that constructs the Delaunay diagram of a set $S$ given its Voronoi diagram.
4. Write a program to compute the largest empty circle.
5. Write a program to compute the smallest enclosing circle.

## 10.6 Point Sets and Dynamic Delaunay Triangulations

The class *POINT_SET*[10] maintains a set of points in the plane under insertions and deletions. It offers dictionary operations, nearest neighbor queries, point location queries, and circular, triangular and rectangular range queries. A point set is maintained as a Delaunay triangulation of its elements and hence the class may equally well be called dynamic Delaunay triangulation[11]. The class is derived from *GRAPH*<*POINT*, *int*> and hence all graph algorithms and all operations for graphs are available for point sets[12].

In this section we will first give an impression of the functionality and then give part of the implementation. The full implementation can be found in [MN98]. We close the section with some experimental data. POINT_SETS are illustrated by the point_set_demo in xlman, see Figure 10.40.

### 10.6.1 *Functionality*

The constructors

```
point_set T;                  // set of points
point_set T(list<point> L);
rat_point_set RT;             // set of rat_points
rat_point_set RT(list<rat_point> L);
```

create a point set for the empty set and the set of points in $L$, respectively. We mentional already that *POINT_SET* is derived from *GRAPH*<*POINT*, *int*>. Every instance of class *POINT_SET* is an embedded planar map. The position of a vertex $v$ is given by $T.pos(v)$ and also by $T[v]$ and we use

$$S = \{T.pos(v) \mid v \in T\}$$

to denote the underlying point set. Each edge is labeled by an element in the enumeration type *delaunay_edge_info* defined in Section 10.2. If the list $L$ in the constructor contains multiple occurrences of equal points, only the last occurrence of each point is retained and the others are discarded.

The function

```
int T.dim()
```

returns the affine dimension of the point set, i.e., $-1$ if $S$ is empty, 0 if $S$ consists of only one point, 1 if $S$ consists of at least two points and all points in $S$ are collinear, and 2 otherwise.

The functions *lookup*, *insert* and *del* give point sets the functionality of a *dictionary for points*.

```
node T.lookup(POINT p)
```

[10] The instantiations are *point_set* for *points* and *rat_point_set* for *rat_points*.

[11] In an earlier version of LEDA we called the class *delaunay_triang*. We found, however, that the typical use of the class emphasizes the query operations and hence we now find the name point set more appropriate.

[12] Only *const* graph operations and graph algorithms should be used as others may destroy the additional invariants imposed by POINT_SET.

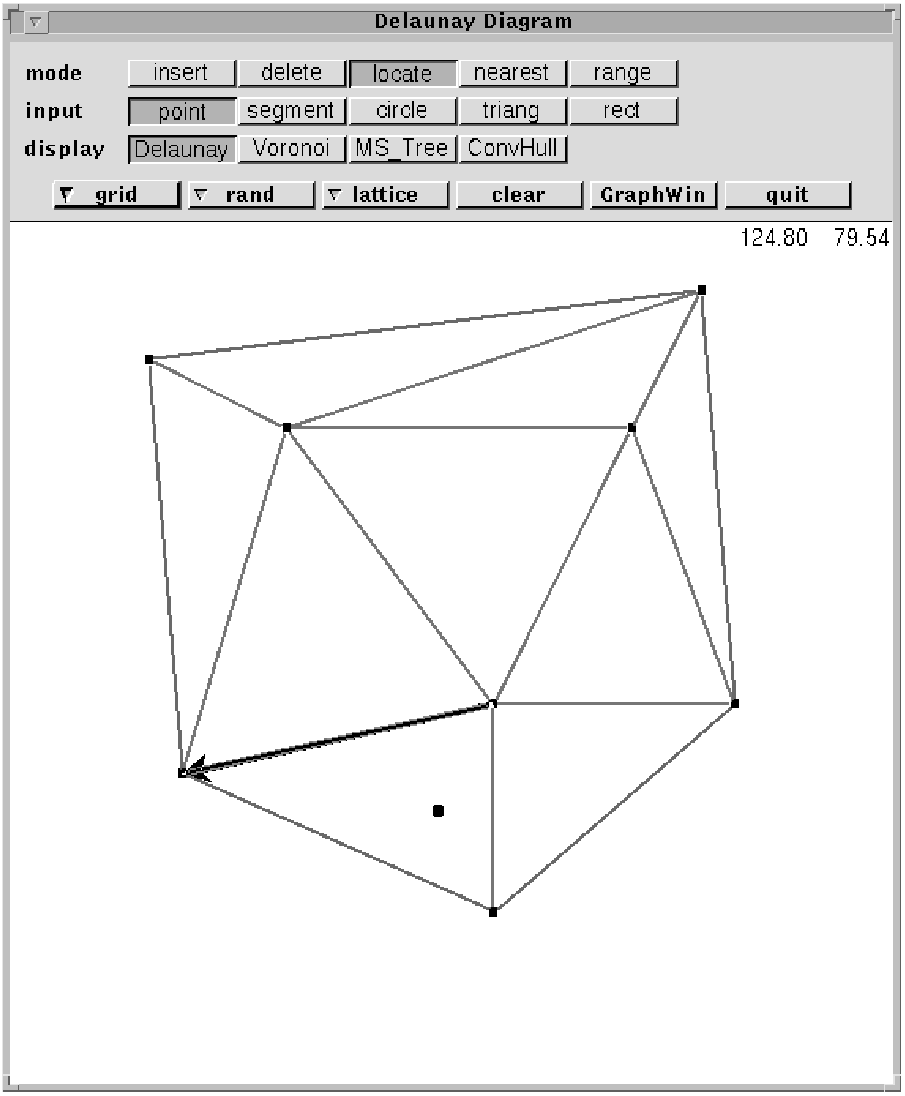


**Figure 10.40** A screenshot of the point-set-demo in xlman. A locate query for the highlighted point was performed. The edge returned by the query is highlighted.

returns a node $v$ of $T$ with $T.pos(v) = p$, if there is such a node, and returns *nil* otherwise.

```
node  T.insert(POINT p)
```

inserts $p$ into $T$ and returns the corresponding node. More precisely, if there is already a node $v$ in $T$ positioned at $p$ (i.e., $pos(v)$ is equal to $p$) then $pos(v)$ is changed to $p$ (i.e., $pos(v)$ is made identical to $p$) and if there is no such node then a new node $v$ with $pos(v) = p$ is added to $T$. In either case, $v$ is returned.

```
void T.del(node v)
```

removes node $v$, i.e., makes $T$ a point set for $S \setminus \{pos(v)\}$.

We come to *point location* and *nearest neighbor* queries. The function

```
edge T.locate(POINT p)
```

performs point location. It returns a dart $e$ (*nil* if $T$ has no edge) such that $p$ lies in the closure of the face to the left of $e$, see Figure 10.40.

The functions

```
node       T.nearest_neighbor(POINT p);
list<node> T.k_nearest_neighbors(POINT p, int k);
```

return a node $v$ of $T$ that is closest to $p$, i.e.,

$$dist(p, pos(v)) = \min\{dist(p, pos(u)) \; ; u \in T\}$$

and the list of the $\min(k, |S|)$ closest points to $p$, respectively. The points in the result list are ordered by distance from $p$. One can also ask for the nearest neighbor(s) of a node.

```
node       T.nearest_neighbor(node w);
list<node> T.k_nearest_neighbors(node w, int k);
```

return a node $v$ of $T$ that is closest to $T[w]$, i.e.,

$$dist(p, pos(v)) = \min\{dist(p, pos(u)) \; ; u \in T \setminus w\}$$

and the list of the $\min(k, |S| - 1)$ closest points to $T[w]$, respectively. The points in the result list are ordered by distance from $T[w]$. Figure 10.41 illustrates nearest neighbor queries and the deletion of nodes.

The next three functions concern *range queries*.

```
list<node> T.range_search(const CIRCLE& C);
list<node> T.range_search(node v,const POINT& b);
list<node> T.range_search(const POINT& a,const POINT& b,const POINT& c);
list<node> T.range_search(const POINT& a,const POINT& b);
```

return the list of points contained in the closure of disk $C$, in the closure of the disk centered at $T[v]$ and having $b$ in its boundary, in the closure of the triangle $(a, b, c)$, and in the closure of the rectangle with diagonal $(a, b)$, respectively. Figure 10.42 illustrates circular range queries.

```
list<edge> T.minimum_spanning_tree()
```

returns a list of edges of $T$ that comprise a minimum spanning tree of $S$ and

```
void   T.compute_voronoi(GRAPH<CIRCLE,POINT>& V)
```

computes the Voronoi diagram $V$ for the sites in $S$. Each node of $V$ is labeled with its defining circle and each edge is labeled with the site lying in the face to its left.

The class POINT_SET also provides functions that support the drawing of Delaunay triangulations, Delaunay diagrams, and Voronoi diagrams. For example,

```
void T.draw_nodes(void (*draw_node)(const POINT&))
```

calls *draw_node*$(pos(v))$ for every node $v$ of $T$.

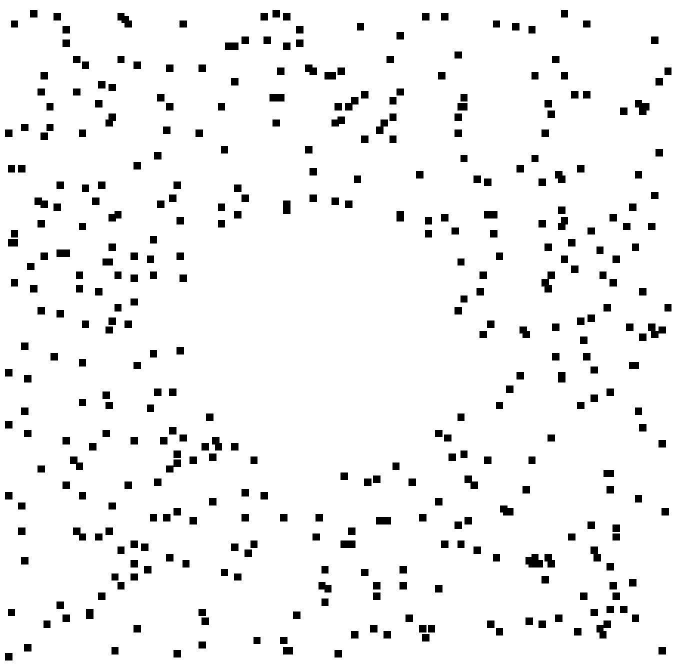

**Figure 10.41** Illustration of nearest neighbor searching plus deletion. We generated a point set of 500 random point and then performed the following operation about thirty times: Locate the nearest neighbor of a point in the center of the screen and delete it. The resulting point set is displayed.

### 10.6.2 *Implementation*

We start with an overview and explain how point sets are represented.

⟨*POINT_SET.h*⟩+≡

```
class __exportC POINT_SET : public  GRAPH<POINT,int>
{
private:
   edge cur_dart;
   edge hull_dart;
   bool check; // functions are checked if true
   // for marking nodes in search procedures
   int cur_mark;
   node_map<int> mark;
   ⟨handler functions for animation⟩
   ⟨functions to mark nodes⟩
   ⟨auxiliary functions⟩
public:
   ⟨public member functions⟩
   ⟨public member functions for checking⟩
};
⟨inline functions⟩
```

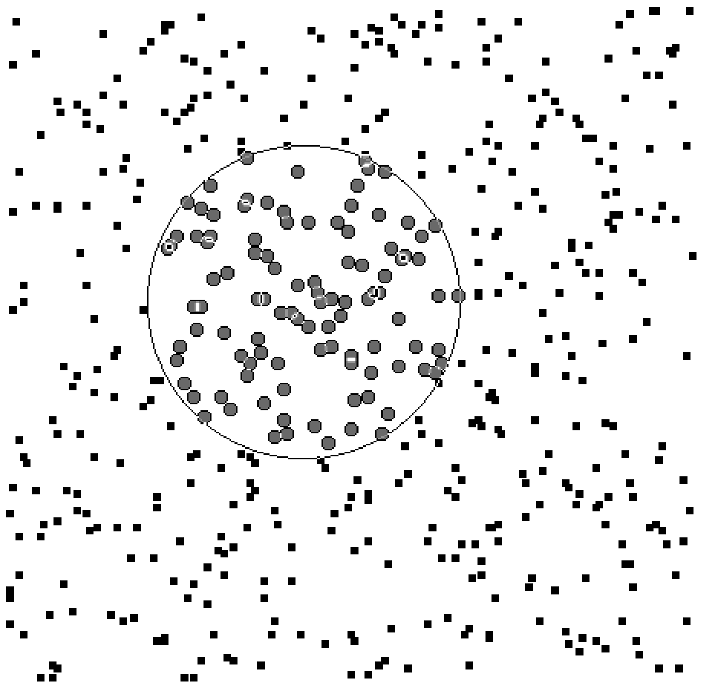

**Figure 10.42** We generated a point set of 500 random points and then performed a circular range query. The points returned by the query are highlighted.

We store a POINT_SET as a planar map *GRAPH<POINT, int> T* plus two edges *cur_dart* and *hull_dart*. For each node $v$ of $T$ we store its position in the plane in $T[v]$ and for each edge $e$ we store its type in $T[e]$. The edge type is an element of the global enumeration type *delaunay_edge_info* defined in geo_global_enums.

```
enum delaunay_edge_info { DIAGRAM_EDGE = 0,  DIAGRAM_DART = 0,
                          NON_DIAGRAM_EDGE = 1, NON_DIAGRAM_DART = 1,
                          HULL_EDGE = 2, HULL_DART = 2
                        }
```

The darts of $T$ are labeled as defined in Section 10.4 on static Delaunay diagrams. Hull darts are labeled *HULL_DART* and non-hull darts are labeled either *DIAGRAM_DART* or *NON_DIAGRAM_DART*. The former label is used for non-hull darts that belong to the Delaunay diagram.

In *hull_dart* we always store a dart of the convex hull and in *cur_dart* we store an arbitrary dart of the triangulation. We use *cur_dart* as the starting point for searches.

Many member functions of *POINT_SET* come with a checker. The boolean *check* controls whether checking is done or not.

Most query operations require graph searches. We use a *node_map<int> mark* and an integer *cur_mark* to mark visited nodes in these searches. More precisely, a node $v$ is marked if $mark[v] == cur\_mark$ and in order to unmark all nodes we increase *cur_mark* by one. We start with *cur_mark* equal to zero and all node marks equal to $-1$ and hence this solution is safe as long as *cur_mark* does not wrap around by overflow. Overflow occurs after MAXINT search operations. Assuming that a query takes at least 100 instructions one can do at most $10^6$ (about $2^{20}$) queries per second. Thus the solution would work for at least $2^{12}$ seconds or about an hour. We conclude that we should guard against this error, in particular, since it will be very difficult to locate once it occurs. The solution is simple. Whenever *cur_mark* reaches MAXINT we reinitialize.

⟨*functions to mark nodes*⟩≡

```
void init_node_marks() { mark.init(*this,-1);
                         cur_mark = 0;
                       }
void mark_node(node v) const { ((node_map<int>&)mark)[v] = cur_mark; }
void unmark_node(node v) const
{ ((node_map<int>&)mark)[v] = cur_mark - 1; }
bool is_marked(node v)  const { return mark[v] == cur_mark; }
void unmark_all_nodes() const
{ ((int&)cur_mark)++;
  if ( cur_mark == MAXINT)
  ((POINT_SET*)this) -> init_node_marks(); //cast away constness
}
```

**Checking:** We have two general routines for purposes of checking:

- *save_state*(*POINT p*) saves the current state of the data structure and the point *p* (which is typically the argument of a query operation) to a file, and
- *check_state*(*string loc*) checks the state of the data structure and prints diagnostic information to *cerr* if an error is found.

Checking is controlled by the boolean flag *check*, i.e., if *check* is *true*, *save_state* and *check_state* perform as described, and if *check* is *false*, they do nothing and *check_state* returns *true*.

A typical function *F* of class POINT_SET has a body of the following form.

```
if ( check ) save_state(POINT p);
/* proper body of F */
if ( check && !check_state("POINT_SET::F") )
{  cerr << additional information ; }
```

Assume now that check is set to *true* and that some function *F* contains an error. The error will be caught by *check_state*. Since the state before the execution of *F* was saved, the error is reproducible. We added this feature to POINT_SET because an earlier version of POINT_SET contained errors which arose very infrequently. For example, at one point we ran a test program for more than an hour before it failed.

**Auxiliary Functions:** The function *mark_edge* is used to assign a *delaunay_edge_info* to an edge. The call to *mark_edge_handler* is for the purposes of animation which we do not discuss here. Readers interested in the animation of the point set class should read [MN98].

⟨*auxiliary functions*⟩≡

```
void mark_edge(edge e, delaunay_edge_info k)
{ assign(e,k);
  if (mark_edge_handler) mark_edge_handler(e);
}
```

**The Constructors:** The constructors allow us to construct a point set for either the empty set of points or for a set $S$ of points. In the latter case the Delaunay triangulation algorithm of Section 10.4 is used, i.e., an arbitrary triangulation is constructed by plane sweep and then Delaunay flips are performed to obtain a Delaunay triangulation. The work horse for the second step is a member function *make_delaunay*$(E)$ that takes a list of edges (it is required that all edges not in $E$ have the Delaunay property) and turns the current triangulation into a Delaunay triangulation.

**Locate:** The function

```
edge T.locate(POINT p)
```

is the basis for all query functions. It returns an edge $e$ of $T$ (*nil* if $T$ has no edge) with the following properties:

- If there is an edge of $T$ containing $p$, such an edge is returned. If $p$ lies on the boundary of the convex hull then a hull dart is returned (and not the reversal of a hull dart).
- If $p$ lies in the interior of a face $f$ of $T$ (if $p$ lies outside the convex hull of $S$, $f$ is the unbounded face) then a dart on the boundary of $f$ is returned. This dart has $p$ to its left, except if all points in $S$ are collinear and $p$ lies on the line passing through the points in $S$. In this case, *target*$(e)$ is the point in $S$ closest to $p$.

The implementation of *locate* is non-trivial. We therefore define a function *check_locate* that checks the output of *locate*.

⟨*auxiliary functions*⟩+≡

```
void check_locate(edge answer,const POINT& p) const;
```

The implementation of *check_locate* is left to the reader; it can be found in [MN98]. We turn to the implementation of *locate*. We distinguish cases according to the dimension of the triangulation.

⟨*POINT_SET.c*⟩+≡

```
edge POINT_SET::locate(POINT p) const
{
  if (number_of_edges() == 0) return nil;
```

```
  if (dim() == 1) { ⟨locate: one-dimensional case⟩ }
  ⟨locate: two-dimensional case⟩
}
```

If the dimension is less than one we return nil.

Let us assume next that the affine dimension of $S$ is one. If $p$ does not lie in the affine hull of $S$, i.e., $p$ does not lie on the line supporting *hull_dart*, we return either *hull_dart* or its reversal. If $p$ lies on the line supporting *hull_dart* we determine the answer by a walk in the triangulation. triangulations!walk through a triangulation

We initialize $e$ to either *hull_dart* or its reversal such that $p$ lies in the halfspace orthogonal[13] to $e$. We walk in the direction of $e$. Let *e1* be the face cycle successor of $e$. As long as *e1* points into the same direction as $e$, i.e., is not the reversal of $e$, and contains $p$ in the halfspace orthogonal to it, we advance $e$ to *e1*.

The walk ends when *e1* is either the reversal of $e$ or does not contain $p$ in the halfspace orthogonal to it. In the former case $p$ lies on $e$ or *target*($e$) is the point in $S$ closest to $p$ and in the latter case $p$ lies on $e$. In either case we may therefore return $e$.

⟨*locate: one-dimensional case*⟩≡

```
edge e = hull_dart;
int orient = orientation(e,p);
if (orient != 0) { if (orient < 0) e = reversal(e);
                   if (check) check_locate(e,p);
                   return e;
                 }
// p is collinear with the points in S. We walk
if ( !IN_HALFSPACE(e,p) ) e = reversal(e);
// in the direction of e. We know IN_HALFSPACE(e,p)
edge e1 = face_cycle_succ(e);
while ( e1 != reversal(e) && IN_HALFSPACE(e1,p) )
{ e = e1;
  e1 = face_cycle_succ(e);
}
if (check) check_locate(e ,p);
return e;
```

We come to the two-dimensional case. Assume w.l.o.g that *cur_dart* is not a hull dart (otherwise, replace *cur_dart* by its reversal).

If $p$ is equal to the source of *cur_dart*, we are done and return the reversal of *cur_dart*; recall that we want to return a hull dart if $p$ lies on the boundary of the convex hull.

So assume that $p$ is distinct from the source of *cur_dart*. The face cycle containing *cur_dart* is a triangle since *cur_dart* is not a hull dart and hence $p$ either does not lie on the line supporting *cur_dart* or the line supporting *face_cycle_pred*(*cur_dart*). Let $e$ be the

[13] The halfspace orthogonal to $e$ has normal vector $e$, has *source*($e$) in its boundary, and contains the target of $e$. We need this definition only for this paragraph.

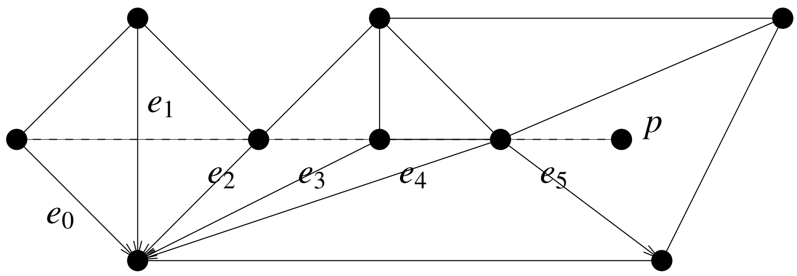


**Figure 10.43** In order to locate $p$ we walk along the segment $s$ from $source(e_0)$ to $p$; $s$ intersects the half-closures of the darts $e_0, e_1, \ldots, e_5$; $e_0, \ldots, e_5$ are directed downwards.

appropriate dart and assume that $p$ lies in the positive halfspace[14] of $e$ (replace $e$ by its reversal otherwise).

We walk along the ray $s$ starting in the source of $e$ and ending in $p$, see Figure 10.43. We will maintain the following invariant during the walk:

- $p$ lies in the positive subspace with respect to $e$.
- $s$ intersects the half-closure of $e$, where the half-closure of $e$ consists of the interior of $e$ plus its source. However, the target of the dart does not belong to the half-closure.

⟨*locate: two-dimensional case*⟩≡

```
edge e = is_hull_dart(cur_dart) ? reversal(cur_dart) : cur_dart;
if (p == pos_source(e) ) return reversal(e);
int orient = orientation(e,p);
if (orient == 0) { e = face_cycle_pred(e);
                   orient = orientation(e,p);
                 }
if (orient < 0) e = reversal(e);
SEGMENT s(pos_source(e),p);
while ( true )
{
  if (is_hull_dart(e)) break;
  ⟨locate: determine the next edge e or break from the loop⟩
}
if (check) check_locate(e ,p);
((edge&)cur_dart) = e;
return e;
```

The while-loop performs the walk. We distinguish cases according to whether $e$ is a hull dart or not. If $e$ is a hull dart, we stop and return $e$.

Otherwise, let $e$, $e_1$, $e_2$ be the face cycle of the triangle $F$ to the left of $e$. We need to find out whether the walk ends in $F$ or whether we are leaving the triangle through $e_1$ or through

[14] The positive halfspace with respect to $e$ is the halfspace to the left of the oriented line supporting $e$.

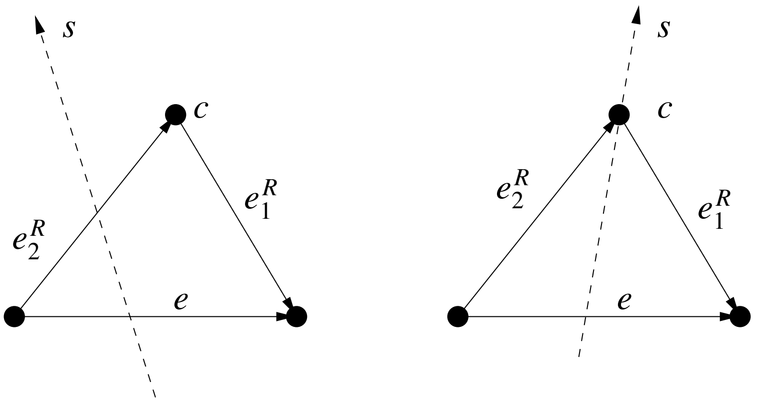


**Figure 10.44** A step of the walk through the triangulation: In the left part of the figure, $c$ lies to the right of $s$ and in the right part it does not.

$e_2$. Let $c$ be the common endpoint of $e_1$ and $e_2$. We distinguish cases according to whether $c$ lies to the right of $s$ or not.

Assume first that $c$ lies to the right of $s$, i.e., $s$ intersects the half-closure of the reversal $e_2^R$ of $e_2$, see Figure 10.44. If $p$ lies to the left of $e_2^R$, we replace $e$ by $e_2^R$ and continue. If $p$ lies on $e_2^R$, we return $e_2^R$, and if $p$ lies to the right of $e_2^R$ and hence in the interior of $F$, we return $e$.

Assume next that $c$ does not lie to the right of $s$, i.e., $s$ intersects the half-closure of the reversal $e_1^R$ of $e_1$, see Figure 10.44. If $p$ lies to the left of $e_1^R$, we replace $e$ by $e_1^R$ and continue. If $p$ lies on $e_1^R$, we return $e_1^R$, and if $p$ lies to the right of $e_1^R$ and hence in the interior of $F$ or on $e_2$ (the latter case can only occur when $s$ passes through the source of $e$ and $p$ lies on $e_2$), we return $e$ in the former case and $e_2^R$ in the latter.

⟨*locate: determine the next edge e or break from the loop*⟩≡

```
edge e1 = face_cycle_succ(e);
edge e2 = face_cycle_pred(e);
int d = ::orientation(s,pos_target(e1));
edge e_next = reversal( (d < 0) ? e2 : e1 );
int orient = orientation(e_next,p);
if ( orient > 0 )  { e = e_next; continue; }
if ( orient == 0 ) { e = e_next; break; }
if ( d == 0 && orient < 0 && orientation(e2,p) == 0 ) e = reversal(e2);
break;
```

This completes the description of *locate*. We still need to argue termination. We clearly make progress when the new dart $e$ intersects $s$ closer to $p$ than the old dart $e$. It may, however, be the case that the intersections are the same. In this situation the new dart $e$ forms a smaller angle with $s$ than the old one.

Having *locate*, we can easily implement the *lookup* operation.

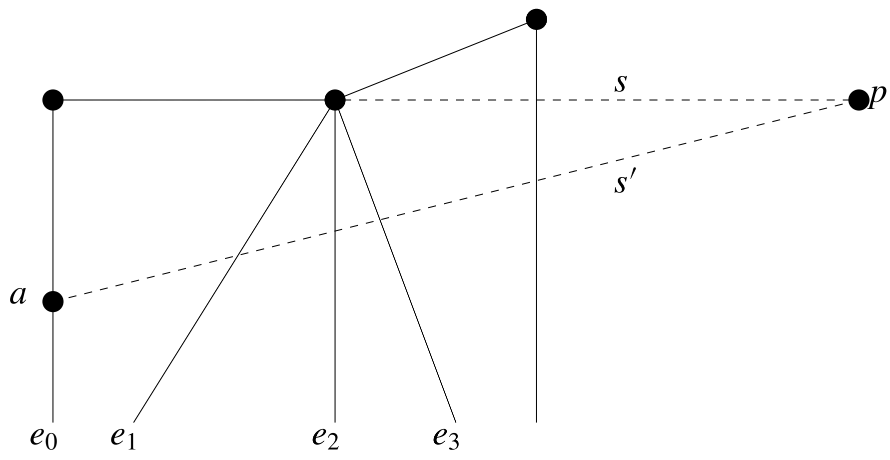


**Figure 10.45** The node $a$ lies in the interior of dart $e_0$ but infinitesimally close to the source node of $e_0$. The darts $e_0, e_1, \ldots$ have $p$ on their left and are directed downwards. The ray $s'$ intersects only the interior of darts.

⟨*POINT_SET.c*⟩+≡

```
node POINT_SET::lookup(POINT p) const
{ if (number_of_nodes() == 1) { node v = first_node();
                                 return (pos(v) == p) ? v : nil;
                               }
  edge e = locate(p);
  if (pos(source(e)) == p) return source(e);
  if (pos(target(e)) == p) return target(e);
  return nil;
}
```

It took us a long time to come up with the short and elegant inner loop for *locate* given above. Earlier attempts were longer and less elegant (and some were plain wrong). Why did we have such difficulties and how did we finally arrive at the program given above? The difficulties stemmed from degeneracies; we had difficulties handling the case that the ray $s$ passes through some node of the triangulation or even runs on top of an edge of the triangulation. Under the additional assumption that there are no degeneracies, i.e., that $s$ enters and leaves triangles through relative interiors of edges, it was easy to write a correct program. We had difficulties extending the solution to the case where $s$ enters and/or leaves through a vertex. Our original solution was clumsy because we used the weaker invariant that $s$ intersects the closure of $e$ (and not only the half-closure as we stated above). This resulted in a lengthy case distinction.

The key to the simpler program was a thought experiment using *perturbation*. Recall that we locate $p$ by a walk through the triangulation starting at the source node of some dart $e_0$. The idea of perturbation is to simulate the walk along a perturbed ray $s'$ that starts in a node $a$ that lies in the interior of $e_0$ but infinitesimally close to the source of $e_0$, see Figure 10.45. The perturbed ray will only pass through the interior of darts (except maybe at $p$); it may pass infinitesimally close to the source of a dart but not infinitesimally close to the target. We concluded that source nodes of darts play a different role than target nodes of darts and came up with the concept of the half-closure of a dart. Once we had the concept of a half-closure, we arrived at a correct program within an hour.

We close this section with a remark about the efficiency of *locate*. Clearly, the running time of locate is proportional to the number of darts of the Delaunay triangulation crossed by the segment $s$. Bose and Devroye [BD95] have shown that the expected number of edges of a Delaunay triangulation of random points crossed by a line segment of length $l$ is $O(l\sqrt{\gamma})$, where $\gamma$ is the point density.

**Insert:** The function

```
node T.insert(POINT p);
```

inserts the point $p$ into $T$ and returns the corresponding node. More precisely, if there is already a node $v$ in $T$ positioned at $p$ (i.e., $pos(v)$ is equal to $p$) then $pos(v)$ is changed to $p$ (i.e., $pos(v)$ is made identical to $p$) and if there is no such node then a new node $v$ with $pos(v) = p$ is added to $T$. In either case, $v$ is returned.

We first define our return statement

⟨*insert::check and return v*⟩≡

```
if ( check && !check_state("POINT_SET::insert") )
{ cerr << "The point inserted was " << p;
  exit(1);
}
return v;
```

and then give an overview. We first deal with the case that $T$ has at most one node. If $T$ has more than one node, we locate $p$ in the triangulation. Let $e$ be the edge returned by *locate*$(p)$. If $p$ is equal to an endpoint of $e$, we replace the endpoint by $p$ and return.

Otherwise, we determine whether $p$ lies on $e$ and then distinguish cases according to the dimension of the triangulation after the insertion. The dimension is one if the current dimension is one and $p$ lies in the affine subspace of $S$.

⟨*POINT_SET.c*⟩+≡

```
node POINT_SET::insert(POINT p)
{ if ( check ) save_state(p);
  node v;
  ⟨T has zero or one node⟩
  edge e = locate(p);
  if (p == pos_source(e)) { assign(source(e),p); return source(e); }
  if (p == pos_target(e)) { assign(target(e),p); return target(e); }
  bool p_on_e = seg(e).contains(p);
  if ( dim() == 1 && orientation(e,p) == 0 )
  { ⟨dimension is one after the insertion⟩ }
  ⟨dimension is two after the insertion⟩
}
```

Assume first that $T$ has at most one node. If $T$ has no node, we create a node, label it with $p$ and return it, if $T$ has one node, we either relabel this node with $p$ or we create a new node with label $p$ and connect it to the old node.

⟨*T has zero or one node*⟩≡

```
if (number_of_nodes() == 0)
{ v = new_node(p); ⟨insert::check and return v⟩ }
if (number_of_nodes() == 1)
{ node w = first_node();
  if (p == pos(w))
  { assign(w,p);
    v = w;
    ⟨insert::check and return v⟩
  }
  else
  { v = new_node(p);
    edge x = new_edge(v,w); edge y = new_edge(w,v);
    mark_edge(x,HULL_DART); mark_edge(y,HULL_DART);
    set_reversal(x,y);
    hull_dart = cur_dart = x;
    ⟨insert::check and return v⟩
  }
}
```

If *dim* is one and $p$ lies in the affine hull of $S$ there are two cases. If $p$ is on $e$ then we split $e$ into two edges and if $p$ does not lie on $e$ we simply add new edges between $p$ and $target(e)$.

⟨*dimension is one after the insertion*⟩≡

```
v = new_node(p);
edge x = new_edge(v,target(e)); edge y = new_edge(target(e),v);
mark_edge(x,HULL_DART);         mark_edge(y,HULL_DART);
set_reversal(x,y);
if (p_on_e)
{ x = new_edge(v,source(e));
  y = new_edge(source(e),v);
  mark_edge(x,HULL_DART);
  mark_edge(y,HULL_DART);
  set_reversal(x,y);
  hull_dart = cur_dart = x;
  del_edge(reversal(e));
  del_edge(e);
}
⟨insert::check and return v⟩
```

In the remaining case the hull is guaranteed to be two-dimensional after the insertion. We now have to triangulate the face that contains $p$. $p$ lies in the interior of the convex hull iff $e$ is not a hull dart.

If $p$ lies in a bounded face (= triangle), we connect it to all (three) nodes of the face.

One of the three new triangles could have height zero. We made sure that *make_delaunay* handles this case correctly.

If $p$ lies in the outer face or on its boundary, we first determine the set of hull darts visible from $p$ by walking in both directions along the hull starting in $e$. We call the two extreme darts reached by these walks *e1* and *e2*. We then add an edge for each visible vertex, i.e. for all vertices from *target*(*e1*) to *source*(*e2*).

There is one subtle point. It is important how ties are broken when $p$ lies on a hull dart. Only one triangle should be added to the triangulation and not three (the latter would be the case if we break the tie in favor of the triangle incident to the hull dart). In order to guarantee that ties are broken correctly, we have *locate* return a hull dart if $p$ does not lie in the interior of the triangulation.

In the implementation we retriangulate the outer face and bounded faces in a uniform way; we add new edges for all nodes from *target*(*e1*) to *source*(*e2*) for two darts *e1* and *e2*. In the case of a bounded face we choose *e1* = *e2* = *e* and in the case of the outer face we set *e1* and *e2* to the extreme (tangent) darts as described above.

⟨*dimension is two after the insertion*⟩≡

```
v  = new_node(p);
edge e1 = e;
edge e2 = e;
list<edge> E;
bool outer_face = is_hull_dart(e);

if (outer_face)
{ //  move e1/e2 to compute upper/lower tangents
  do e1 = face_cycle_pred(e1); while (orientation(e1,p) > 0);
  do e2 = face_cycle_succ(e2); while (orientation(e2,p) > 0);
}

// insert edges between v and target(e1) ... source(e2)
e = e1;
do { e = face_cycle_succ(e);
     edge x = new_edge(e,v);
     edge y = new_edge(v,source(e));
     set_reversal(x,y);
     mark_edge(e,DIAGRAM_DART);
     E.append(e);
     E.append(x);
   } while (e != e2);

if (outer_face)
{ // mark last visited and new edges as hull edges
  mark_edge(face_cycle_succ(e1),HULL_DART);
  mark_edge(face_cycle_pred(e2),HULL_DART);
  mark_edge(e2,HULL_DART);
  hull_dart = e2;
}

make_delaunay(E); // restores Delaunay property
```

⟨*insert::check and return v*⟩

**Deletion:** The functions

```
void T.del(node v)
void T.del(POINT p)
```

remove the node $v$ and the point $p$, respectively, i.e., make $T$ a Delaunay triangulation for $S \setminus \{pos(v)\}$ and $S \setminus p$, respectively.

The strategy to remove a node is simple. Removal of a node from the interior of a two-dimensional triangulation (of course, the program also has to handle the removal of a node from a triangulation that is not two-dimensional or of a node which lies on the boundary of the convex hull) creates a cavity in the triangulation. The cavity is retriangulated in an arbitrary way and then *make_delaunay*$(E)$ is called to restore the Delaunay property, where $E$ is the set of new edges and the set of edges on the boundary of the cavity.

After this general outline we define our return statement and give an overview of the deletion procedure.

⟨*del: check and return*⟩≡

```
if ( check && !check_state("POINT_SET::del(node v)") )
{ cerr << "deleted the node with position " << pos(v);
  exit(1);
}
return;
```

⟨*POINT_SET.c*⟩+≡

```
void POINT_SET::del(node v)
{
  if (v == nil) error_handler(1,"POINT_SET::del: nil argument.");
  if (number_of_nodes() == 0)
    error_handler(1,"POINT_SET::del: graph is empty.");
  if (check) save_state(inf(v));
  if ( dim() < 2 )
  {
    if ( outdeg(v) == 2)
    { node s = target(first_adj_edge(v));
      node t = target(last_adj_edge(v));
      edge x = new_edge(s,t);  edge y = new_edge(t,s);
      mark_edge(x,HULL_DART);  mark_edge(y,HULL_DART);
      set_reversal(x,y);
    }
    del_node(v);
    cur_dart = hull_dart = first_edge();
    ⟨del: check and return⟩
  }
  ⟨removal of v from a two-dimensional triangulation⟩
  ⟨del: check and return⟩
}
```

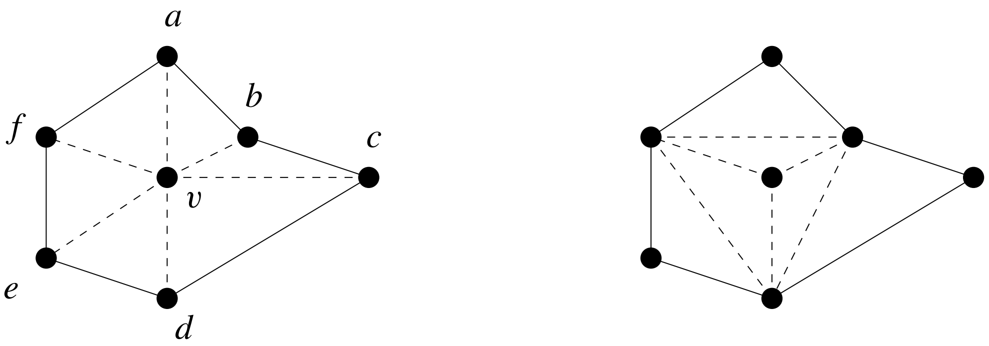


**Figure 10.46** The right part of the figure shows the effect of flipping the edges $(v, a)$, $(v, c)$ and $(v, e)$.

If the dimension of the triangulation is less than two, the removal of $v$ is trivial. If the dimension is zero or the dimension is one and $v$ is an extreme node of the triangulation (i.e., the outdegree of $v$ is one), we simply remove $v$. If $v$ has outdegree two, we connect the two neighbors of $v$ by a new edge and then delete $v$. Of course, *cur_dart* or *hull_dart* could have been incident to $v$ and hence have to be given new values.

We come to the interesting case, the removal of $v$ from a two-dimensional triangulation. We first discuss the case that $v$ lies in the interior of the triangulation. We will later see that the same strategy also handles the case where $v$ lies on the boundary of the convex hull.

Removal of $v$ creates a face $P$ that is, in general, not a triangle. It is only a triangle if the degree of $v$ is three. We need to retriangulate this face. A natural approach would be to remove $v$ and to retriangulate after the removal of $v$. However, this approach does not exploit the fact that $P$ is a so-called *star-shaped polygon* with respect to $v$, i.e., that $v$ can see all vertices of $P$. We will exploit this fact as follows in the retriangulation process. We will show below that there is always an edge $e$ incident to $v$ such that the two triangles incident to $v$ form a convex quadrilateral. We "flip $e$ away from $v$" by replacing it by the other diagonal of the triangle. In this way the degree of $v$ is decreased by one. We continue until the degree of $v$ is three. At this point, $v$ is removed and the created face is a triangle, see Figure 10.46.

We now give the details. We need a slightly more general definition of star-shapedness than was alluded to in the text above. The more general definition is needed to cope with the case that three or more points of $S$ lie on a common line.

We call a polygon $P$ *star-shaped* with respect to a point $v$ if either:

- $v$ lies in the interior of $P$ and for every vertex $p$ of $P$ the open line segment $vp$ is contained in the interior of $P$, or

- $v$ lies in the relative interior of an edge $e$ of $P$ and for every vertex $p$ of $P$ that is not an endpoint of $e$ the open line segment $vp$ is contained in the interior of $P$.

**Lemma 8** *Let $P$ be a polygon which has at least four vertices and is star-shaped with respect to some point $v$. Then there are three consecutive vertices $p$, $q$, $r$ of $P$ such that*

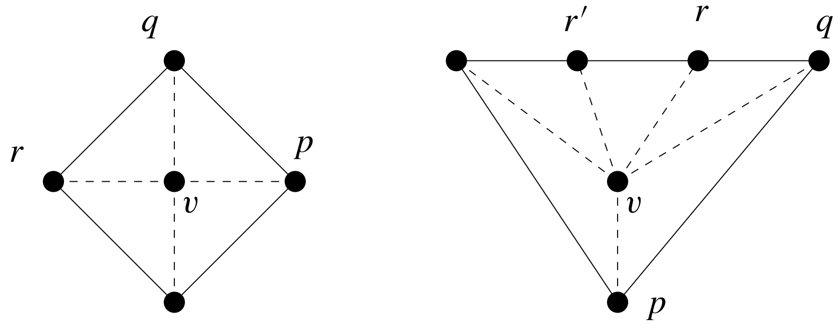


**Figure 10.47** $(p, q, r, v)$ forms a convex quadrilateral. In the situation on the left $v$ will lie on an edge of $P'$ after the flip of edge $(v, q)$ and in the situation on the right it will still lie in the interior of $P'$. The quadruple $(q, r, r', v)$ does not qualify for a flip.

*$(v, p, q, r)$ form a convex quadrilateral. In this quadrilateral the angle at $v$ maybe equal to $\pi$. The angle at $v$ can be equal to $\pi$ only if $v$ lies in the interior of $P$, see Figure 10.47.*

*Let $P'$ be the polygon obtained from $P$ by replacing the edges $pq$ and $qr$ by the edge $pr$. Then $P'$ is star-shaped with respect to $v$.*

*Proof* Consider any triangulation $T$ of $P$. $T$ consists of at least two triangles. Since the dual of a triangulation is a tree and every tree has at least two leaves, there must be at least two triangles in $T$ whose edges consist of two consecutive edges of $P$ plus the chord connecting the source of the first edge with the target of the second edge and hence there must be at least one such triangle which, in addition, does not contain $v$ in its interior. Consider one such triangle, say $t$, and let $e_1 = (p, q)$ and $e_2 = (q, r)$ be the edges of $P$ that are contained in its boundary. Since $(p, q, r)$ is a triangle of $T$ the angle at $q$ is less than $\pi$.

Since $v$ is not contained in the interior of $t$, $(v, p, q, r)$ forms a convex quadrilateral. In this quadrilateral the angles at $p$ and $r$ must be less than $\pi$ since $P$ is star-shaped with respect to $v$. Also by the star-shapedness, the angle at $v$ can be equal to $\pi$ only if $v$ lies in the interior of $P$.

$P'$ is clearly star-shaped with respect to $v$. □

Call an edge incident to $v$ *flipable* if the two triangles incident to it form a convex quadrilateral. As long as there is a flipable edge incident to $v$ flip it. The lemma above guarantees that the process does not terminate before $v$ has degree three.

How can we find flipable edges quickly? We scan through the edges incident to $v$. Let $e$ be the current edge. If $e$ is not flipable, we advance $e$ to the cyclic successor of $e$, and if $e$ is flipable, we flip it and set $e$ to the cyclic predecessor of $e$.

When do we terminate? We terminate when $v$ has degree three. Since we want to use the same procedure also for nodes on the hull we develop a more general termination condition. We terminate when the degree of $v$ reaches *min_deg*, where *min_deg* is three for nodes in the interior and is two for hull nodes. We also keep a counter *count* which is a lower bound on the number of edges out of $v$ that are certainly not flipable. We increment *count* whenever a non-flipable edge is found, we decrement *count* by two whenever a flip is performed, as this may make the two neighbors of the flipped edge flipable, and we terminate if *count* reaches *outdeg*($v$).

Why is this correct? Call an edge *certified non-flipable* if it has been tested for flipping and its two neighbors have not changed since. In the procedure just outlined the edges that are certified non-flipable are consecutive in the cyclic adjacency list of $v$ and *count* is a lower bound on their number. This shows correctness.

The running time of retriangulation is linear in the initial degree of $v$. This follows from the fact that the total decrement of *count* is bounded by twice the initial degree of $v$ and hence the total increase of *count* is bounded by thrice the initial degree of $v$.

We obtain the following code.

⟨*removal of v from a two-dimensional triangulation*⟩≡

```
list<edge> E;
int min_deg = 3;
edge e;
forall_adj_edges(e,v)
{ E.append(face_cycle_succ(e));
  if (is_hull_dart(e)) min_deg = 2;
}
int count = 0;
e = first_adj_edge(v);
while ( outdeg(v) > min_deg && count < outdeg(v) )
{ edge e_pred = cyclic_adj_pred(e);
  edge e_succ = cyclic_adj_succ(e);
  POINT a = pos_target(e_pred); POINT c = pos_target(e_succ);
  if ( !right_turn(a,c,pos(v)) && right_turn(a,c,pos_target(e)) )
  { // e is flipable
    edge r = reversal(e);
    move_edge(e,reversal(e_succ),target(e_pred));
    move_edge(r,reversal(e_pred),target(e_succ),LEDA::before);
    mark_edge(e,DIAGRAM_DART);
    mark_edge(r,DIAGRAM_DART);
    E.append(e);
    e = e_pred;
    count = count - 2;
    if ( count < 0 ) count = 0;
  }
  else
  { e = e_succ;
    count++;
  }
}
if ( min_deg == 2 )
{ ⟨adjust marks of new hull darts and their reversals⟩ }
```

```
cur_dart = E.head();
del_node(v);
make_delaunay(E);
```

We give some more explanations. The while-loop performs the retriangulation. During the retriangulation we build up a list $E$ of edges whose Delaunay property needs to be checked: $E$ consists of all edges in the boundary of the cavity created by the removal of $v$ and all edges created during retriangulation.

After retriangulation we remove $v$ and add call *make_delaunay*($E$) to restore the Delaunay property.

We also have to take care of *cur_dart*. It may have been incident to $v$. We set it to an arbitrary edge in the boundary of the cavity created by the removal of $v$.

This completes the discussion of the case when a node in the interior of the triangulation is removed. We will next argue that the same retriangulation strategy works when $v$ is a node in the boundary of the triangulation.

Again we flip edges away from $v$ until no further edges are flipable. When this is the case, the neighbors of $v$ form a chain that is concave as seen from $v$ and hence removal of $v$ leaves us with a triangulation of the remaining nodes. Removal of $v$ also turns some darts into hull darts. Their labels have to be changed to *HULL_DART* and the edges of their reversal have to be changed to *DIAGRAM_DART*. There is a small exception to the latter rule, namely when a reversal is a hull dart itself. This will be the case when the removal of $v$ reduces the dimension of the triangulation from two to one.

⟨*adjust marks of new hull darts and their reversals*⟩≡

```
edge e,x;
forall_adj_edges(e,v)
{ x = face_cycle_succ(e);
  mark_edge(x,HULL_DART);
  if ( !is_hull_dart(reversal(x)) ) mark_edge(reversal(x),DIAGRAM_DART);
}
hull_dart = x;
```

**Nearest Neighbor Searching:** The functions

```
node       T.nearest_neighbor(POINT p);
list<node> T.k_nearest_neighbors(POINT p, int k);
```

return a node $v$ closest to $p$, i.e., $dist(p, pos(v)) = \min\{dist(p, pos(u))\ ;\ u \in T\}$, and the list of the $\min(k, |S|)$ closest points to $p$, respectively. The points in the result list are ordered by distance from $p$. One can also ask for the nearest neighbor(s) of a node.

```
node       T.nearest_neighbor(node w);
list<node> T.k_nearest_neighbors(node w, int k);
```

return a node $v$ different from $w$ that is closest to $T[w]$ and the list of the $\min(k, |S| - 1)$ closest points to $T[w]$, respectively.

The following observation paves the way for a simple algorithm for both problems and is also the basis of the range query algorithms to be discussed in the next section.

**Lemma 9** *Let $s$ and $t$ be two nodes of a Delaunay triangulation $T$ and let $d$ be their distance. Then there is a path from $s$ to $t$ in $T$ such that all intermediate nodes have distance less than $d$ from $s$.*

*Proof* We use induction on $d$. Let $D$ be the disk with radius $d$ centered at $s$. If $st$ is an edge of $T$, we are done. Otherwise let $a$ and $b$ be the two neighbors of $t$ such that the segment $st$ runs between the edges $ta$ and $tb$ of $T$. The points $a$, $b$, and $t$ form a triangle of $T$. If one of $a$ and $b$ has distance less than $d$ from $s$, we can apply the induction hypothesis and are done. So assume otherwise, i.e., neither $a$ nor $b$ lies in the interior of $D$. The segments $st$ and $ab$ intersect (since $s$ cannot lie in the interior of the triangle with corners $a$, $b$, and $t$) and hence $(s, a, t, b)$ is a convex quadrilateral. The disk $D$ proves that the segment $ab$ does not belong to the Delaunay triangulation of $\{a, b, s, t\}$ and hence cannot be an edge of $T$. □

The lemma suggests a simple strategy to find the $k$-nearest neighbors of $p = T[v]$. If the number of points in $T$ is no more than $k$, we simply return all nodes in $T$. So assume otherwise. We start a graph search starting in $v$. We keep all reached nodes in a priority queue according to their (squared) distance from $v$ and always continue the exploration from a node with smallest distance. The lemma above guarantees that this strategy explores the nodes of $T$ in order of increasing distance from $v$.

⟨*POINT_SET.c*⟩+≡

```
#include <LEDA/p_queue.h>
list<node> POINT_SET::nearest_neighbors(node v, int k) const
{ list<node> result;
  int n = number_of_nodes();
  if ( k <= 1 ) return result;
  if ( n + 1 <= k )
  { node w;
    forall_nodes(w,*this) if ( w != v ) result.append(w);
    return result;
  }
  POINT p = pos(v);
  unmark_all_nodes();
  p_queue<RAT_TYPE,node> PQ;
  PQ.insert(0,v); mark(v);
  while ( k > 0 )
  { pq_item it = PQ.find_min();
    node w = PQ.inf(it); PQ.del_item(it);
    if ( w != v ) { result.append(w); k--; }
    node z;
    forall_adj_nodes(z,w)
```

```
    { if ( !is_marked(z) ) { PQ.insert(p.sqr_dist(pos(z)),z);
                             mark(z);
                           }
    }
  }
  return result;
}
```

We come to the case where we want to search for the nearest neighbors of a point $p$. We simply insert $p$ into $T$ and then use the procedure above.

A small complication arises from the fact that $p$ may lie on a node of $T$. We test for this case by performing a lookup for $p$. If $p$ does not lie on a node of $v$, we insert it. Of course, it has to removed again after calling the procedure above and $p$ has to be removed from the list of answers.

⟨*POINT_SET.c*⟩+≡

```
list<node> POINT_SET::nearest_neighbors(POINT p, int k)
{ list<node> result;
  int n = number_of_nodes();
  if ( k <= 0 ) return result;
  if ( n <= k ) return all_nodes();
  // insert p and search neighbors graph starting at p
  node v = lookup(p);
  bool old_node = true;
  if ( v == nil ) { v = ((POINT_SET*)this)->insert(p);
                    old_node = false;
                  }
  else k--;
  result = nearest_neighbors(v,k);
  if ( old_node )
    result.push_front(v);
  else
    ((POINT_SET*)this)->del(v);
  return result;
}
```

The nearest neighbor of a node $v$ in a Delaunay diagram is a node adjacent to $v$. Thus one only has to find the minimum (squared) distance between $v$ and its neighboring nodes.

⟨*POINT_SET.c*⟩+≡

```
node POINT_SET::nearest_neighbor(node v) const
{
  if (number_of_nodes() <= 1) return nil;
  POINT p = pos(v);
  edge e = first_adj_edge(v);
  node min_v = target(e);
  while ((e = adj_succ(e)) != nil)
```

| n | I | NN | NNA |
|---|---|---|---|
| 50000 | 128.2 | 2.32 | 18.08 |

**Table 10.5** We constructed a point set of $n$ random points in the unit square and performed a nearest neighbor query for each node in the triangulation. NN shows the time for the function *nearest_neighbor* and NNA shows the time with alternative implementation of the inner loop. Column I shows the time for the $n$ insertions. The table was made with the rational kernel.

```
  { node u = target(e);
    if ( p.cmp_dist(pos(u),pos(min_v)) < 0 ) min_v = u;
  }
  return min_v;
}
```

An alternative way to write the inner loop is:

⟨*alternative inner loop*⟩≡

```
node min_v = target(e);
RAT_TYPE min_d = p.sqr_dist(pos(min_v));
while ((e = adj_succ(e)) != nil)
{ node u = target(e);
  RAT_TYPE d_u = p.sqr_dist(pos(u));
  if ( d_u < min_d ) { min_v = u;
                       min_d = d_u;
                     }
}
```

This is much slower, see Table 10.5. Why is the alternative so much slower; aren't the two programs doing exactly the same thing? Both programs compute the squared distance from $v$ to all its neighbors and find the minimum.

The difference is that the alternative version computes all squared distances *exactly* as rational numbers[15] and finds the minimum of these rational numbers. The original version asks the kernel to compare distances. The kernel first computes floating point approximations to the squared distances and uses them in the comparisons. If the floating point approximation suffices to decide the comparison, the exact squared distance is never computed and a lot of work is saved.

**Range Searches:** We have functions for circular, triangular, and rectangular range searches.

In order to perform a circular range query with center $v$ we perform a DFS starting at $v$. The search is restricted to the nodes that lie in the circular range. Correctness follows from Lemma 9.

[15] We assume for this paragraph that the rational kernel is used.

⟨*POINT_SET.c*⟩+≡

```
void POINT_SET::dfs(node s, const POINT& pv,
                    const POINT& p, list<node>& L) const
{ L.append(s);
  mark_node(s);
  node u;
  forall_adj_nodes(u,s)
      if (!is_marked(u) && pv.cmp_dist(pos(u),p) <= 0 ) dfs(u,pv,p,L);
}

list<node> POINT_SET::range_search(node v,const POINT& p) const
{
  list<node> L;
  POINT pv = pos(v);
  unmark_all_nodes();
  dfs(v,pv,p,L);
  return L;
}
```

The other two kind of queries can be reduced to circular queries by first performing a range query with the circumcircle of the triangle or rectangle and then filtering the returned list of points with the triangle or rectangle, respectively. We leave the implementation of the other queries to the reader.

**Experimental Data:** Table 10.6 contains running times. The table shows that nearest neighbor queries for nodes are very efficient in comparison to nearest neighbor queries for points. This comes from the fact that the latter involve a lookup, an insertion, a deletion, as well as a nearest neighbor query for a node. For queries that ask for the ten nearest neighbors the difference is not as pronounced. This stems from the fact that $k$-nearest neighbor queries involve rational arithmetic.

***Exercises for 10.6***

1 Implement circular range queries.

2 Implement triangular and rectangular range queries. You may use circular range queries.

3 Animate the Delaunay class such that the actions performed after the insertion of a point are visualized.

4 The *nearest_neighbors* algorithm uses a *p_queue*<*RAT_TYPE*, *node*>. The code becomes slightly simpler if a *node_pq*<*RAT_TYPE*> is used. Why is it better to use a *p_queue* instead of a *node_pq*? Time both programs and explain.

5 Develop a version of the $k$-nearest neighbor search that cuts down on the use of rational arithmetic.

6 Our implementation of *nearest_neighbor*(*POINT p*) modifies the Delaunay triangulation by an insertion and a deletion. It is not guaranteed that the original Delaunay triangulation is restored. Can you modify the implementation such that it becomes a const-operation? Try to determine the set $L$ of all edges of the current triangulation whose

| K | n | I | L | NNP | NNV | NNP(10) | NNV(10) | D |
|---|---|---|---|---|---|---|---|---|
| S | 1000 | 1.14 | 0.66 | 2.15 | 0.05 | 10.22 | 7.07 | 1.16 |
| | 2000 | 2.79 | 1.83 | 4.92 | 0.09 | 21.25 | 14.06 | 2.77 |
| | 4000 | 6.83 | 5.36 | 11.68 | 0.2 | 44.4 | 28.29 | 6.65 |
| D | 1000 | 1.15 | 0.68 | 2.22 | 0.03999 | 10.27 | 7.03 | 1.18 |
| | 2000 | 2.78 | 1.89 | 4.99 | 0.11 | 21.21 | 14.04 | 2.75 |
| | 4000 | 6.76 | 5.23 | 11.53 | 0.2 | 44.25 | 28.25 | 6.65 |
| C | 1000 | 0.82 | 0.41 | 0.99 | 0.03 | 5.43 | 4.65 | 2.84 |
| | 2000 | 1.75 | 0.9 | 2.08 | 0.06 | 11.09 | 9.31 | 8.2 |
| | 4000 | 3.78 | 2.03 | 4.42 | 0.13 | 22.35 | 18.48 | 29.09 |

**Table 10.6** The performance of point sets. As in the other tables of this chapter we used three kinds of inputs: random points in the unit square, random points in the unit disk, and random points on the unit circle. We generated two sets $L$ and $LQ$ of $n$ points, built a point set $T$ by inserting the points in $L$ (I), performed $n$ lookups for the points in $LQ$ (L), performed nearest neighbor queries for the points in $LQ$ (NNP), performed nearest neighbor queries for the nodes of $T$ (NNV), computed the ten nearest neighbor queries for the points in $LQ$ (NNP(10)), computed the ten nearest neighbor queries for nodes of $T$ (NNV(10)), and finally deleted all points.

Delaunay property is destroyed by $p$. The nearest neighbor of $p$ must be a vertex of the triangle containing $p$ or an endpoint of an edge in $L$.

## 10.7 Line Segment Intersection

The line segment intersection problem asks to compute the set of intersections of a set $S$ of line segments in the plane. It is one of the basic geometric problems and has numerous applications, e.g., in computer aided design, geographic information systems, and cartography. We will see an application to boolean operations on polygons in Section 10.8. Many different algorithms have been designed for the problem and several of them are available in LEDA. The line segment intersection problem comes in many different flavors as different applications have different output requirements. One may be interested in the number of intersections, or one may want to trigger an action for every pair of intersecting segments, or one may want to compute the graph induced by the segments, or one may want to compute the trapezoidal decomposition induced by the set of segments. In LEDA we provide functions for several output conventions which we survey in Section 10.7.1. We also give

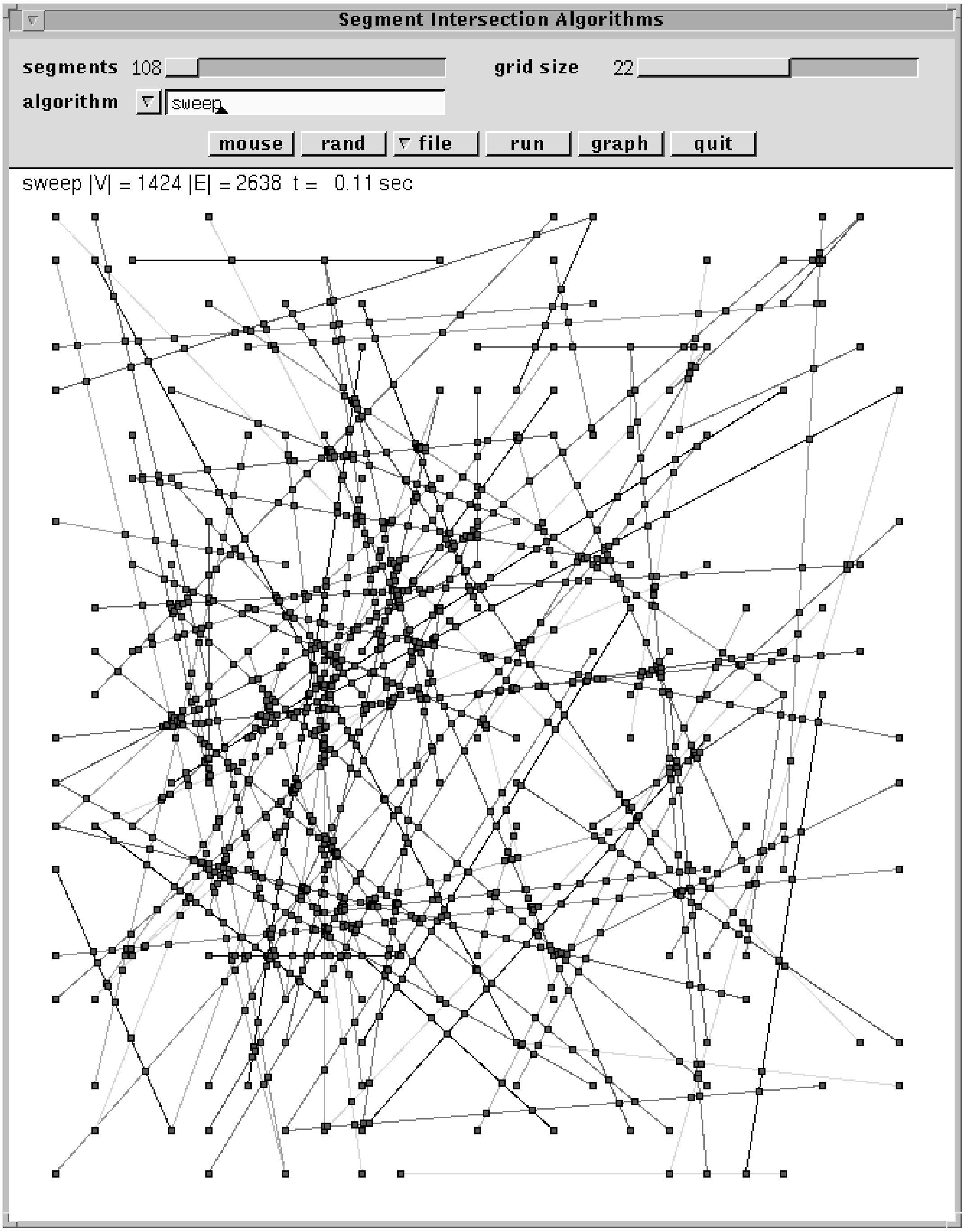


**Figure 10.48** A screen shot of the intersect_segments demo in xlman. The sweep line algorithm was used to compute the graph induced by a set of 203 random segments. The induced graph has 1424 nodes and 2638 edges.

some experimental data in this section. In the remaining sections we discuss the sweep line algorithm for segment intersection.

The algorithms discussed in this section are illustrated by the intersect_segments demo in xlman. Figure 10.48 shows a screen shot.

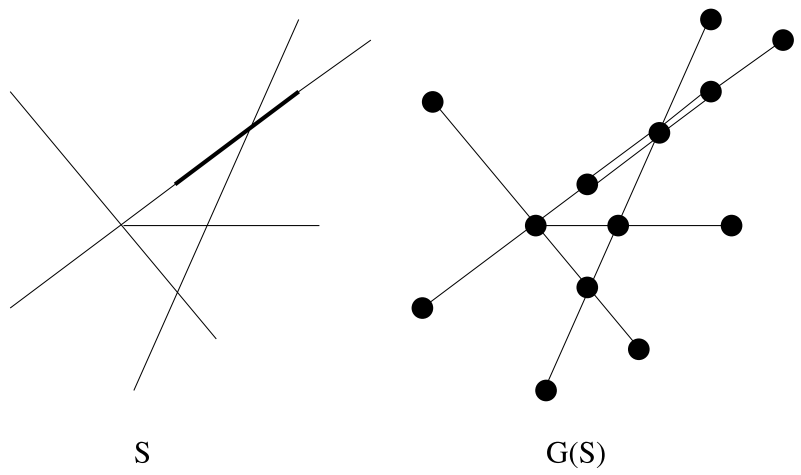


**Figure 10.49** A set $S$ of segments and the induced planar graph.

### 10.7.1 *Functionality*

We first introduce some terminology. Two segments $s_1$ and $s_2$ *intersect* if they have at least one point in common and *overlap* if they have more than one point in common. Two segments $s_1$ and $s_2$ are said to have a *proper intersection* if they share exactly one point and this point lies in the relative interior of both segments. A segment of length zero is called a trivial segment.

The undirect graph $U(S)$ induced by $S$ is defined as follows. The vertices of $U(S)$ are all endpoints of segments and all proper intersection points between segments in $S$. The edges of $U$ are the maximal relatively open and connected subsets of segments in $S$ that contain no vertex of $U(S)$. Figure 10.49 shows an example. Note that the graph $U(S)$ contains parallel edges if $S$ contains segments that overlap. We use $n$ to denote the number of segments in $S$, $s$ to denote the number of nodes of $U$, $m$ to denote the number of edges of $U$, and $k$ to denote the number of pairs of intersecting segments. If $S$ contains no overlapping segments, $m = O(n+s)$. If $S$ contains overlapping segments, $m$ may be as large as $n(n+s)$ since an input segment may be divided into $n+s$ pieces by the endpoints and intersection points. The number of nodes of $U$ is at most $n+k \le n+n(n-1)/2$. If many segments have a common intersection, $k$ may be much larger than $s$. For example, if all $n$ segments pass through a common point then $s = n+1$ and $k = n(n-1)/2$.

The function

```
void SEGMENT_INTERSECTION(const list&lt;SEGMENT&gt;& S,
                          GRAPH&lt;POINT,SEGMENT&gt;& G, bool embed = false)
```

computes a directed graph $G(s)$ representing $U(S)$. The algorithm makes no assumption about the segments in $S$. They may be overlapping, they may have multiple intersections, they may share endpoints, they may have length zero, … .

$G$ and $U$ have the same set of nodes; each node of $G$ is labeled by its position in the plane.

The edges of $G$ correspond to the edges of $U$. If *embed* is false, there is exactly one dart in $G$ for each edge in $U$; the dart is labeled by the segment in $S$ containing it and inherits

its direction from the segment containing it, i.e, if $e = (v, w)$ is a dart of $G$ then $G[e]$ is directed from $G[v]$ to $G[w]$.

If *embed* is true, $G$ is a plane map. For each edge of $U$ there are two darts in $G$ and the two darts are reversal of each other. For each node $v$ of $G$ the cyclic list of darts out of $v$ are counter-clockwise ordered.

The function

```
void SEGMENT_INTERSECTION(const list&lt;SEGMENT&gt;& S, list&lt;POINT&gt;& P)
```

returns the list of points that correspond to nodes of $G$ of degree two or more and the function

```
SEGMENT_INTERSECTION(const list&lt;SEGMENT&gt;& S,
                     void (*report)(const SEGMENT&, const SEGMENT&) )
```

calls *report*$(s_1, s_2)$ for every pair $(s_1, s_2)$ of intersecting segments. Observe that the points in $P$ are a subset of the points for which *report* is called. For example, if $S$ consists of two identical trivial segments, then $G(S)$ consists of a single node and no edge and hence $P$ will be empty. On the other hand, *report* will be called for this pair of segments.

For all functions above several implementations are available. The implementations are based on the algorithms of Bentley and Ottmann ([BO79]), Mulmuley ([Mul90]), and Balaban ([Bal95]). For the reporting version of segment intersection we also have the trivial implementation which simply checks every pair of segments in $S$ for an intersection.

```
void MULMULEY_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT,SEGMENT&gt;& G,
                       bool embed = false);
void SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT,SEGMENT&gt;& G,
                    bool embed = false, bool use_optimization = true);
void SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S , list&lt;POINT&gt;& P);
void BALABAN_SEGMENTS(const list&lt;SEGMENT&gt;& S,
                      void (*report)(const SEGMENT&, const SEGMENT&));
void TRIVIAL_SEGMENTS(const list&lt;SEGMENT&gt;& S,
                      void (*report)(const SEGMENT&, const SEGMENT&));
```

The asymptotic running time of the Bentley–Ottmann algorithm is $O(m+(n+s)\log n)$, the asymptotic running time of the Mulmuley algorithm is $O(m+s+n\log n)$. Both algorithms can be used for all functions above. If *embed* is true the running time of the Bentley–Ottmann algorithm increases by $O(m\log m)$, since an additional sorting step is required. The asymptotic running time of the Balaban algorithm is $O(n\log^2 n + k)$. It can only be used for the functions that report intersections. The asymptotic running time of the trivial implementation is $O(n^2)$.

Table 10.7 compares the running time of our various implementations. In the examples, Balaban's algorithm is always better than the trivial algorithm. Mulmuley's algorithm is better than the Bentley–Ottmann algorithm when the number of intersections is large. It also incurs a smaller additional cost for turning $G(S)$ into a planar map (as it always computes an undirected planar map). When the number of intersections is small, the Bentley–Ottmann algorithm and Mulmuley's algorithm behave about the same.

| n | d | V | E | S | S + E | M | M + E | T | B |
|---|---|---|---|---|---|---|---|---|---|
| 2000 | 22 | 4007 | 2014 | 1.14 | 1.3 | 1.74 | 1.76 | 15.13 | 1.94 |
| 2000 | 23 | 4026 | 2052 | 1.18 | 1.37 | 2.25 | 2.29 | 14.87 | 2.07 |
| 2000 | 24 | 4136 | 2272 | 1.25 | 1.42 | 2.91 | 2.91 | 15.26 | 2.17 |
| 2000 | 25 | 4428 | 2856 | 1.39 | 1.63 | 3.44 | 3.47 | 15.06 | 2.33 |
| 2000 | 26 | 5857 | 5714 | 1.81 | 2.37 | 4.44 | 4.5 | 15.31 | 2.48 |
| 2000 | 27 | 10954 | 15908 | 3.03 | 5.02 | 5.93 | 6.02 | 15.41 | 2.74 |
| 2000 | 28 | 29683 | 53366 | 7.57 | 16.43 | 9.71 | 10.02 | 16.01 | 3.22 |
| 2000 | 29 | 91789 | 177578 | 22.84 | 58.31 | 20.04 | 20.94 | 16.62 | 5.38 |
| 2000 | 30 | 267045 | 528090 | 70.24 | 193.7 | 48.96 | 51.95 | 18.42 | 11.54 |

**Table 10.7** The running time of the functions related to segment intersections. S stands for the sweep line algorithm of Bentley and Ottmann ([BO79]), M and B stand for the algorithms of Mulmuley and Balaban ([Mul90, Bal95]), and T stands for the trivial algorithm that checks every pair of segments for an intersection. The "+ E" indicates that the graph $G(S)$ is returned as a planar map. The first three columns contain the number of input segments, the number of nodes of $G$, and the number of edges of $G$, respectively.
We chose $n$ segments. For each segment we chose random $k$ bit integer for the Cartesian coordinates of the first endpoint and obtained the second endpoint from the first by adding a vector with random $d$ bit integer coordinates. We used $k = 30$ and different values of $d$. The number of intersections is an increasing function of $d$.

Let us interpret the experimental findings in terms of asymptotic running time. When the number of intersections is very large, the $O(k \log n)$ term[16] in the time bound of the sweep algorithm dominates the $O(k)$ term in the time bound of the other algorithms. The trivial algorithm has a running time $\Theta(n^2 + k \cdot T_{report})$, where $T_{report}$ is the cost of calling the function *report*. In our tests, *report* increases a counter and hence does minimal work. Thus the constant factor in the big-O expression is small. This explains why the running time of the trivial algorithm depends very little on the number of intersections and why the trivial algorithm is competitive when the number of intersections is large. When the number of intersections is small the Bentley–Ottmann algorithm and Mulmuley's algorithm have running time $O(n \log n)$ and Balaban's algorithm has running time $O(n \log^2 n)$. We should therefore expect that the former two algorithms are superior when the number of intersections is small. This is confirmed by the experiments.

### 10.7.2 *The Sweep Line Algorithm*

We discuss the Bentley–Ottmann sweepline algorithm for line segment intersection and give an implementation of the function

16 In our examples, there are hardly any intersections of three or more segments and hence $s \approx k$. Observe that if all nodes are endpoints or proper intersections of exactly two segments then $E = n + 2(V - 2n)$, as $U(S)$ contains $2n$ nodes of degree one and $(V - 2n)$ nodes of degree four. In our examples we have $E \approx n + 2(V - 2n)$. We will therefore replace $s$ by $k$ in the discussion to follow.

```
SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT,SEGMENT&gt;& G,
               bool embed, bool use_optimization)
```

that takes a list $S$ of segments and computes the graph $G$ induced by it. For each vertex $v$ of $G$ it also computes its position in the plane, and for each edge $e$ of $G$ it computes the segment containing it.

If *embed* $=$ *true*, the algorithm turns $G$ into a planar map, i.e., $G$ is made bidirected and the adjacency lists are sorted according to the geometric embedding in clockwise order.

If *use_optimization* $=$ *true*, an optimization described below is used.

The algorithm runs in time $O((n+s)\log(n+m)+m)$, where $n$ is the number of segments, $s$ is the number of nodes of $G$, and $m$ is the number of edges of $G$. If $S$ contains no overlapping segments then $m = O(n+s)$. If *embed* is *true*, the running time is increased by an additive factor of $O(m \log m)$. Note that $s \leq 3(n+k)$ and that $k$ can be as large as $s^2$.

We want to stress that the implementation makes no assumptions about the input, in particular, segments may have length zero, may be vertical or may overlap, several segments may intersect in the same point, endpoints of segments may lie in the interior of other segments, … .

We achieve this generality by reformulating the plane sweep algorithm so that it can handle all geometric situations. The reformulation makes the description of the algorithm shorter and it also makes the algorithm faster, since $k$ is replaced by $s$ in the time bound[17]. The only previous algorithm that could handle all degeneracies is due to Myers [Mye85]. Its expected running time for random segments is $O(n \log n + k)$ and its worst case running time is $O((n+k)\log n)$.

In the sweepline paradigm a vertical line is moved from left to right across the plane and the output (here the graph $G(S)$) is constructed incrementally as it evolves behind the sweep line. One maintains two data structures to keep the construction going: the so-called *Y-structure* contains the intersection of the sweep line with the scene (here the set $S$ of line segments) and the so-called *X-structure* contains the events where the sweep has to be stopped in order to add to the output or to update the X- or Y-structure. In the line segment intersection problem an event occurs when the sweep line hits an endpoint of some segment or an intersection point. When an event occurs, some nodes and edges are added to the graph $G(S)$, the Y-structure is updated, and maybe some more events are generated. When the input is in general position (no three lines intersecting in a common point, no endpoint lying on a segment, no two endpoints or intersections having the same $x$-coordinate, no vertical lines, no overlapping segments, …) then at most one event can occur for each position of the sweep line and there are three clearly distinguishable types of events (left endpoint, right endpoint, intersection) with easily describable associated actions, cf. [Meh84b, VII.8]. We want to place no restrictions on the input and therefore need to proceed slightly differently. We now describe the required changes.

We define the sweep line by a point $p\_sweep = (x\_sweep, y\_sweep)$. Let $\epsilon$ be a positive

[17] Bentley and Ottmann formulated their algorithm for line segments in general position and stated a time bound of $O((n+k)\log n)$.

infinitesimal (readers not familiar with infinitesimals may think of $\epsilon$ as an arbitrarily small positive real). Consider the directed line $L$ consisting of a vertical upward ray ending in point $(x\_sweep+\epsilon^2, y\_sweep+\epsilon)$ followed by a horizontal segment ending in $(x\_sweep-\epsilon^2, y\_sweep+\epsilon)$ followed by a vertical upward ray. We call $L$ the *sweep line*. Note that[18] no endpoint of any segment lies on $L$, that no two segments of $S$ intersect $L$ in the same point except if the segments overlap, and that no non-vertical segment of $S$ intersects the horizontal part of $L$. All three properties follow from the fact that $\epsilon$ is arbitrarily small but positive. Figure 10.50 illustrates the definition of $L$ and the main data structures used in the algorithm: the Y-structure, the X-structure, and the graph $G$.

The Y-structure contains all segments intersecting the sweep line $L$ ordered as their intersections with $L$ appear on the directed line $L$. Overlapping segments are ordered by their *ID-numbers*. Every segment has an associated ID-number; distinct segments are guaranteed to have distinct IDs.

The X-structure contains all endpoints that are to the right of the sweep line and also some intersection points between segments in the Y-structure. More precisely, for each pair of segments adjacent in the Y-structure their intersection point is contained in the X-structure (if it exists and is to the right of the sweep line). The X-structure may contain other intersection points. The graph $G$ contains the part of $G(S)$ that is to the left of the sweep line.

Initially, the Y-structure and the graph $G$ are empty and the X-structure contains all endpoints of all input segments. The events in the X-structure are then processed in left to right order. Events with the same $x$-coordinate are processed in bottom to top order.

Assume that we need to process an event at point $p$ and that the X-structure and Y-structure reflect the situation for $p\_sweep = (p.x, p.y - 2\epsilon)$. Note that this is true initially, i.e., before the first event is removed from the X-structure. We now show how to establish the invariants for $p\_sweep = p$. We proceed in seven steps.

1. We add a node $v$ at position $p$ to our graph $G$.
2. We determine all segments in the Y-structure containing the point $p$. These segments form a possibly empty subsequence of the Y-structure.
3. For each segment in the subsequence we add an edge to the graph $G$.
4. We delete all segments ending in $p$ from the Y-structure.
5. We update the order of the subsequence in the Y-structure. This amounts to moving the sweep line across the point $p$.
6. We insert all segments starting in $p$ into the Y-structure.
7. We generate events for the segments in the Y-structure that become adjacent by the actions above and insert them into the X-structure.

This completes the description of how to process the event $p$. The invariants now hold for $p\_sweep = p$ and hence also for $p\_sweep = (p'.x, p'.y - 2\epsilon)$ where $p'$ is the new first element of the X-structure.

[18] We defined the sweep line in this seemingly complicated way in order to be able to write this "Note that". The note will allow us to define a linear order on the segments intersecting the sweep line.

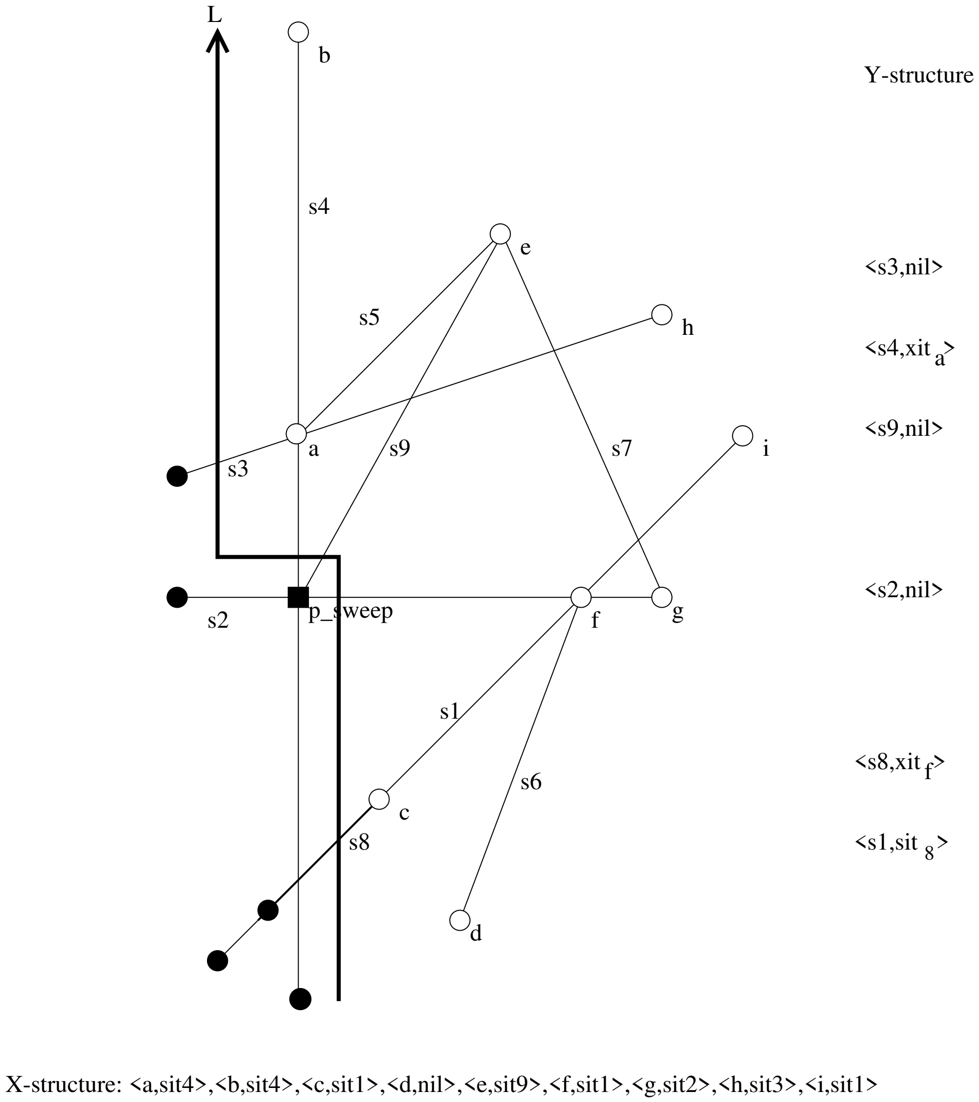


**Figure 10.50** A scene of nine segments. The segments $s_1$ and $s_8$ overlap. The sweep line is shown in bold. The part of $G(S)$ to the left of the sweep line is already constructed. Its nodes are shown filled. The sweep line intersects the segments $s_1$, $s_8$, $s_2$, $s_9$, $s_4$, and $s_3$ and in this order. The Y-structure contains one item for each one of them. The X-structure contains points $a$, $b$, $c$, $d$, $e$, $f$, $g$, $h$, and $i$ and in this order.
The information associated with the items in the X- and Y-structure will be explained in the next section.

### 10.7.3 *The Implementation of the Sweep Line Algorithm*

*This section is joint work with Ulrike Bartuschka.*

The implementation follows the algorithm closely. It makes use of several data types discussed in earlier chapters. The main "ingredients" are the basic geometric objects and primitives, sorted sequences for the X- and Y-structure, priority queues for storing events, and graphs for representing the output.

To make this section self-contained we briefly review the data types used.

**Points and Segments:** The types *rat_point* and *rat_segment* realize points and segments in the plane with rational coordinates and are part of the rational kernel. A *rat_point* is specified by its homogeneous coordinates of type *integer* – the type of arbitrary precision integers. If $p$ is a *rat_point* then $p.X(\,)$, $p.Y(\,)$, and $p.W(\,)$ return its homogeneous coordinates and *p.xcoord*( ) and *p.ycoord*( ) return its Cartesian coordinates. If $x$, $y$, and $w$ are of type *integer* with $w \neq 0$ then *rat_point*$(x, y)$ and *rat_point*$(x, y, w)$ create the *rat_point* with homogeneous coordinates $(x, y, 1)$ and $(x, y, w)$, respectively. Two points are equal (*operator==*) if they agree in their Cartesian coordinates. A *rat_segment* is specified by its two endpoints; so if $p$ and $q$ are *rat_point*s then *rat_segment*$(p, q)$ is the directed segment with source $p$ and target $q$. If $s$ is a *rat_segment* then *s.source*( ) and *s.target*( ) return the source and target of $s$, respectively.

There are also points (class *point*) with coordinates of type *double*. The corresponding segment class is called *segment*. The classes *point* and *segment* have the same interface as *rat_point* and *rat_segment*. However, the internal representation is different: instead of storing the homogeneous coordinates as *integers*, the Cartesian coordinates are stored as *doubles*.

The sweep program can be executed with either the rational or the floating point geometry kernel. Be aware, however, that the instantiation with the floating point kernel is not fully reliable, see Section 10.7.2. In the sequel we use POINT to denote the point class and SEGMENT to denote the segment class.

POINTS and SEGMENTS come with a large number of geometric primitives. In the sweep program the following primitives are used:

- *int compare*(*POINT p*, *POINT q*)
  compares points by their lexicographic order; $p$ precedes $q$ if either
  *p.xcoord*( ) $<$ *q.xcoord*( ) or
  *p.xcoord*( ) $=$ *q.xcoord*( ) and *p.ycoord*( ) $<$ *q.ycoord*( ). The function returns $-1$ if $p$ precedes $q$, returns 0 if $p$ and $q$ are equal, and returns $+1$ otherwise. The lexicographic order of points is the default order on points.

- *int orientation*(*POINT p*, *POINT q*, *POINT r*)
  computes the orientation of points $p$, $q$, and $r$ in the plane, i.e., 0 if the points are collinear, $-1$ if they define a clockwise oriented triangle, and $+1$ if they define a counter-clockwise oriented triangle.

- *int orientation*(*SEGMENT s*, *POINT p*)
  computes *orientation*(*s.source*( ), *s.target*( ), $p$).

- *int cmp_slopes*(*SEGMENT s1*, *SEGMENT s2*)
  compares the slopes of *s1* and *s2*. If one of the segments is degenerate, i.e., has length zero, the result is zero. Otherwise, the result is the sign of *slope*(*s1*) $-$ *slope*(*s2*).

- *bool intersection_of_lines*(*SEGMENT s1*, *SEGMENT s2*, *POINT& p*)
  returns *false* if segments *s1* and *s2* are parallel or one of them is degenerate. Otherwise, it computes the point of intersection of the two straight lines supporting the segments, assigns it to the third parameter $q$, and returns *true*.

Our program maintains its own set of segments which we call *internal segments* or simply segments and store in the list *internal*; input segments are called input segments or original segments when the need for distinction arises. Internal segments are directed from left to right; vertical segments are directed upwards. There is one internal segment for every non-trivial input segment. The *map*<*SEGMENT*, *SEGMENT*> *original* stores for each internal segment the corresponding original segment.

⟨*local declarations*⟩≡

`list<SEGMENT>          internal;`

`map<SEGMENT,SEGMENT> original;`

**Sorted Sequences:** The type *sortseq*<$K$, $I$> realizes sorted sequences of pairs in $K \times I$, see Section 5.6; $K$ is called the key type and $I$ is called the information type of the sequence. The key type must be linearly ordered, i.e., the function *int compare*(*const* $K$&, *const* $K$&) must be defined for the type $K$ and the relation $<$ on $K$ defined by $k_1 < k_2$ iff $compare(k_1, k_2) < 0$ must be a linear order on $K$. An object of type *sortseq*<$K$, $I$> is a sequence of items (type *seq_item*) each containing a pair in $K \times I$. We use <$k$, $i$> to denote an item containing the pair $(k, i)$ and call $k$ the key and $i$ the information of the item. The keys in a sorted sequence $\langle k_1, i_1\rangle$, $\langle k_2, i_2\rangle$, $\ldots$, $\langle k_m, i_m\rangle$ form an increasing sequence, i.e., $k_l < k_{l+1}$ for $1 \le l < m$.

Let $S$ be a sorted sequence of type *sortseq*<$K$, $I$> and let $k$ and $i$ be of type $K$ and $I$, respectively. The operation *S.lookup*($k$) returns the item $it = \langle k, .\rangle$ in $S$ with key $k$ if there is such an item and returns *nil* otherwise. If *S.lookup*($k$) == *nil* then *S.insert*($k$, $i$) adds a new item $\langle k, i\rangle$ to $S$ and returns this item. If *S.lookup*($k$) == *it* then *S.insert*($k$, $i$) changes the information in the item *it* to $i$. If $it = \langle k, i\rangle$ is an item of $S$ then *S.key*(*it*) and *S.inf*(*it*) return $k$ and $i$, respectively, and *S.succ*(*it*) and *S.pred*(*it*) return the successor and predecessor item of *it*, respectively; the latter operations return *nil* if these items do not exist. The operation *S.min*( ) returns the first item of $S$, *S.empty*( ) returns *true* if $S$ is empty and *false* otherwise. Finally, if *it1* and *it2* are items of $S$ with *it1* before *it2* then *S.reverse_items*(*it1*, *it2*) reverses the subsequence of $S$ starting at item *it1* and ending at item *it2*.

In our implementation the X-structure has type *sortseq*<*POINT*, *seq_item*> and the Y-structure has type *sortseq*<*SEGMENT*, *seq_item*>. The Y-structure has one item for each segment intersecting the sweep line. The information field in the Y-structure is used for cross-links with the X-structure and for linking overlapping segments.

The X-structure is ordered according to the default order of points and the Y-structure is ordered according to the intersections of the segments with the directed sweep line $L$. The position of the sweep line is determined by *p_sweep* and the comparison object *cmp* realizes the order in the Y-structure. The class *sweep_cmp* will be defined below.

$\langle$*local declarations*$\rangle+\equiv$

```
POINT p_sweep;
sweep_cmp cmp(p_sweep);
sortseq<POINT,seq_item>     X_structure;
sortseq&lt;SEGMENT, seq_item&gt;  Y_structure(cmp);
```

In the example of Figure 10.50 the sweep line intersects the segments $s_1$, $s_8$, $s_2$, $s_9$, $s_4$, and $s_3$. The Y-structure therefore consists of six items, one each for segments $s_1$, $s_8$, $s_2$, $s_9$, $s_4$, and $s_3$.

The X-structure contains an item for each endpoint of an input segment that is to the right of the sweep line and an item for each intersection point between segments that are adjacent in the Y-structure and that intersect to the right of the sweep line. It may also contain intersection points between segments that are not adjacent in the Y-structure.[19] The points in the X-structure are ordered according to the lexicographic ordering of their Cartesian coordinates. As mentioned above this is the default order on points.

In the example of Figure 10.50 the X-structure contains items for the endpoints $b$, $c$, $d$, $e$, $g$, $h$, $i$ and for intersections $a$ and $f$. Here, $a$ and $f$ are the intersections between segments $s_4$ and $s_3$, and $s_1$ and $s_2$, respectively.

The informations associated with the items of both structures serve as cross-links between the two structures: the information associated with an item in the X-structure is either *nil* or an item in the Y-structure; the information associated with an item in the Y-structure is either *nil* or an item of either structure. The precise definition follows: consider first an item $\langle s, it\rangle$ in the Y-structure and let $s'$ be the segment associated with the successor item $it'$ in the Y-structure. If $s$ and $s'$ overlap then $it = it'$. If $s$ and $s'$ do not overlap and $s \cap s'$ exists and lies to the right of the sweep line then $it$ is the item in the X-structure with key $s \cap s'$. In all other cases we have $it = nil$.

Consider next an item $\langle p, sit\rangle$ in the X-structure. If $sit \neq nil$ then $sit$ is an item in the Y-structure and the segment associated with it contains $p$. Moreover, if there is a pair of adjacent segments in the Y-structure that intersect in $p$ then $sit \neq nil$. We may have $sit \neq nil$ even if there is no pair of adjacent segments intersecting in $p$.

In our example, the Y-structure contains the items $\langle s_1, sit_8\rangle$, $\langle s_8, xit_f\rangle$, $\langle s_2, nil\rangle$, $\langle s_9, nil\rangle$, $\langle s_4, xit_a\rangle$, and $\langle s_3, nil\rangle$ where $sit_8$ is the item of the Y-structure with associated segment $s_8$ and $xit_a$ and $xit_f$ are the items of the X-structure with associated points $a$ and $f$, respectively. Let's turn to the items of the X-structure next. All items except $\langle d, nil\rangle$ point back to the Y-structure. If $sit_i$ denotes the item $\langle s_i, \ldots\rangle$, $i \in \{1, 2, 9, 4, 3\}$, of the Y-structure then the items of the X-structure are $\langle a, sit_4\rangle$, $\langle b, sit_4\rangle$, $\langle c, sit_1\rangle$, $\langle d, nil\rangle$, $\langle e, sit_9\rangle$, $\langle f, sit_1\rangle$, $\langle g, sit_2\rangle$, $\langle h, sit_3\rangle$, and $\langle i, sit_1\rangle$.

**The Order on the Y-structure:** The segments in the Y-structure are ordered according to their intersection with the sweep line. Overlapping segments are ordered according to their

[19] Our X-structure may contain intersection points between segments that are no longer adjacent in the Y-structure. These events could be removed from the X-structure. Removing these events would guarantee an X-structure of linear size, however, at the cost of complicating the code. Since the size of the X-structure is always bounded by the size of the output graph we do not remove these events.

ID-number. All segments in the Y-structure are non-trivial and the position of the sweep line is determined by *p_sweep*.

The Y-structure is realized as a sorted sequence. In a sorted sequence comparisons between keys are only made during insertions and lookups and then one of the keys involved in the comparison is an argument of the operation. We conclude that compare is only called for segments $s_1$ and $s_2$ where one of the segments has its source point equal to *p_sweep*. Also, at least one of the segments is non-trivial and if one of the segments is trivial it has both endpoints equal to *p_sweep*. Let us assume first that both segments are non-trivial.

Assume $s_i$ has its source point equal to *p_sweep*. If *p_sweep* does not lie on $s_{1-i}$, i.e., $orientation(s_{1-i}, p\_sweep) \neq 0$, then the orientation test is also the outcome of compare.

If both segments contain *p_sweep* we compare the slopes of $s_1$ and $s_2$ (*orientation*(*s2*, *s1.target*( ))). Only overlapping segments are equal after this comparison. They are ordered according to their ID-numbers. Since only internal segments are stored in the Y-structure and since internal segments are pairwise non-identical, any two internal segments have different ID-numbers.

The compare class *sweep_cmp* is derived from *leda_cmp_base*, see Section 2.10. It has a private data member *p_sweep* whose value will always be equal to the position of the sweep line; in the constructor the data member is initialized to the initial position of the sweep line and *set_position* is used to inform the compare object about any advance of the sweep line.

⟨*geometric primitives*⟩≡
```
class sweep_cmp : public leda_cmp_base&lt;SEGMENT&gt;
{
  POINT p_sweep;
public:
 sweep_cmp(const POINT& p) : p_sweep(p) {}
 void set_position(const POINT& p) { p_sweep = p; }
 int operator()(const SEGMENT& s1, const SEGMENT& s2) const
 { // Precondition:
   // p_sweep is identical to the left endpoint of either s1 or s2.
   if (identical(s1,s2)) return 0;
   int s = 0;
   if ( identical(p_sweep,s1.source()) )  s = orientation(s2,p_sweep);
   else
    if ( identical(p_sweep,s2.source()) ) s = orientation(s1,p_sweep);
    else error_handler(1,"compare error in sweep");
   if (s || s1.is_trivial() || s2.is_trivial()) return s;
   s = orientation(s2,s1.target());
   // overlapping segments will be ordered by their ID_numbers :
   return s ? s : (ID_Number(s1) - ID_Number(s2));
 }
};
```

We still need to explain the purpose of the tests *is_trivial*. We will also have to locate trivial segments in the Y-structure. These segments will have both endpoints equal to *p_sweep*. We want the search to be successful iff the Y-structure contains a segment passing through *p_sweep*. In the order defined above, the trivial segment (*p_sweep*, *p_sweep*) is larger than all segments intersecting the sweep line before *p_sweep*, is equal to all segments passing through *p_sweep*, and is larger than all segments intersecting the sweep line after *p_sweep*. We conclude that a search for the trivial segment will return a segment passing through *p_sweep* if there is one.

It is important to observe that the compare function for segments changes as the sweep progresses. What does it mean then that the keys of the items in a sorted sequence form an increasing sequence? The requirement is that whenever a lookup or insert operation is applied to a sorted sequence, the sequence must be sorted with respect to the current compare function. The other operations may be applied even if the sequence is not sorted.

**The Graph $G$:** The graph $G$ has type *GRAPH<POINT*, *SEGMENT>*, i.e., it is a directed graph where a *POINT*, respectively *SEGMENT*, is associated with each node, respectively edge, of the graph. The graph $G$ is the part of $G(S)$ that is left of the sweep line. The point associated with a vertex defines its position in the plane and the segment associated with an edge is an input segment containing the edge. We use two operations to extend the graph $G$. If $p$ is a *POINT* then *G.new_node*$(p)$ adds a new node to $G$, associates $p$ with the node, and returns the new node. If $v$ and $w$ are nodes of $G$ and $s$ is a *SEGMENT* then *G.new_edge*$(v, w, s)$ adds the edge $(v, w)$ to $G$, associates $s$ with the edge, and returns the new edge. In order to facilitate the addition of edges we maintain a *map<SEGMENT*, *node>* *last_node*: it gives for each segment in the Y-structure the rightmost vertex lying on the segment.

$\langle$*local declarations*$\rangle+\equiv$
```
map&lt;SEGMENT,node&gt;                last_node(nil);
```

**The Priority Queue:** We use a priority queue *seg_queue* to drive the insertion of segments into the Y-structure. The queue contains all internal segments that are ahead of the sweep line ordered according to their left endpoint. In particular, the first segment in *seg_queue* is always the segment that is encountered next by the sweep line. *Seg_queue* has type *p_queue<POINT*, *SEGMENT>*.

The data type *p_queue*$<P, I>$ realizes priority queues with priority type $P$ and information type $I$. $P$ must be linearly ordered. Priority queues are an item-based data type. Every item (of type *pq_item*) stores a pair $(p, i)$ from $P \times I$, $p$ is called the priority and $i$ is called the information of the item. The usual operations on priority queues (*insert*, *delete_min*, *find_min*) are available.

$\langle$*local declarations*$\rangle+\equiv$
```
p_queue<POINT,SEGMENT>      seg_queue;
```

We are now ready for the program. It has the following structure:

⟨*sweep_segments.c*⟩+≡

```
⟨geometric primitives⟩
⟨embedding⟩
void SWEEP_SEGMENTS(const list&lt;SEGMENT&gt;& S, GRAPH&lt;POINT, SEGMENT&gt;& G,
                    bool embed, bool use_optimization)
{ ⟨local declarations⟩
  ⟨initialization⟩
  ⟨sweep⟩
  ⟨post processing⟩
}
```

**Initialization:** We describe the initialization of the data structures. We clear the graph $G$, we compute a coordinate *Infinity* that is larger than the absolute value of the coordinates of all endpoints and that plays the role of $\infty$ in our program, we insert the endpoints of all input segments into the X-structure, and we create for each non-trivial input segment an internal segment with the same endpoints, insert this segment into *seg_queue* and link the input segment to it (through map *original*), we create two sentinel segments at $-\infty$ and $+\infty$, respectively, and insert them into the Y-structure, we put the sweep line at its initial position by setting *p_sweep* to $(-\infty, -\infty)$, and we add a stopper point with coordinates $(+\infty, +\infty)$ to *seg_queue*. The sentinels avoid special cases and thus simplify the code. Finally, we introduce a variable *next_seg* that always contains the first segment in *seg_queue*.

⟨*initialization*⟩≡

```
G.clear();
COORD Infinity = 1;
SEGMENT s;
forall(s,S)
{
  COORD x1 = s.xcoord1(), y1 = s.ycoord1();
  COORD x2 = s.xcoord2(), y2 = s.ycoord2();
  if (x1 < 0) x1 = -x1;
  if (y1 < 0) y1 = -y1;
  if (x2 < 0) x2 = -x2;
  if (y2 < 0) y2 = -y2;
  while (x1 >= Infinity || y1 >= Infinity ||
         x2 >= Infinity || y2 >= Infinity )   Infinity *= 2;
  seq_item it1 = X_structure.insert(s.source(), seq_item(nil));
  seq_item it2 = X_structure.insert(s.target(), seq_item(nil));
  if (it1 == it2) continue;  // ignore zero-length segments
  POINT p = X_structure.key(it1);
  POINT q = X_structure.key(it2);
  SEGMENT s1 = ( compare(p,q) < 0 ? SEGMENT(p,q) : SEGMENT(q,p) );
  original[s1] = s;
  internal.append(s1);
  seg_queue.insert(s1.source(),s1);
}
SEGMENT lower_sentinel(-Infinity,-Infinity,Infinity,-Infinity);
```

```
SEGMENT upper_sentinel(-Infinity, Infinity,Infinity, Infinity);
p_sweep = lower_sentinel.source();
cmp.set_position(p_sweep);
Y_structure.insert(upper_sentinel,seq_item(nil));
Y_structure.insert(lower_sentinel,seq_item(nil));
POINT pstop(Infinity,Infinity);
seg_queue.insert(pstop,SEGMENT(pstop,pstop));
SEGMENT next_seg = seg_queue.inf(seg_queue.find_min());
```

There is one subtle point in the code above. An insert operation into a sorted sequence with a key that is already present in the sorted sequence returns the item containing the key; it does not add a new item to the sequence and its does not change the key of the item returned. We exploit this feature of sorted sequences to ensure that internal segments share endpoints. Assume for concreteness that $s_1$ and $s_2$ are two input segments with a common source point and assume that $s_1$ is processed first. When the source point of $s_2$ is inserted into the X-structure, the item containing the source point of $s_1$ will be returned and hence the internal segments corresponding to $s_1$ and $s_2$ have the same (not just equal) source point[20].

**Processing Events:** We now come to the heart of procedure sweep: processing events. Let $event = \langle p, sit\rangle$ be the first event in the X-structure and assume inductively that our data structure is correct for $p\_sweep = (p.x,\ p.y - 2\epsilon)$. Our goal is to change *p_sweep* to $p$, i.e., to move the sweep line across point $p$. As long as the X-structure is not empty we perform the following actions.

We first extract the next event point *p_sweep* from the X-structure by assigning the minimal key in the X-structure to *p_sweep*, adjusting the compare function for segments to the new position of the sweep line, and adding a vertex $v$ with position *p_sweep* to the output graph $G$. Then, we handle all segments passing through or ending at *p_sweep*. Finally, we insert all segments starting at *p_sweep* into the Y-structure, check for possible intersections between pairs of segments now adjacent in the Y-structure, and update the X-structure. Finally, we delete the event from the X-structure.

⟨*sweep*⟩≡

```
while ( !X_structure.empty() )
{ seq_item event = X_structure.min();
  p_sweep = X_structure.key(event);
  cmp.set_position(p_sweep);
  node v = G.new_node(p_sweep);
  ⟨handle passing and ending segments⟩
  ⟨insert starting segments⟩
  ⟨compute new intersections and update X-structure⟩
  X_structure.del_item(event);
}
```

[20] A point is realized as a pointer to a representation class. Two points are equal if they have the same Cartesian coordinates and two points are identical if they share the representation. Testing two points for identity is faster than testing them for equality.

**Handling Passing and Ending Segments:** We first determine the segments passing through or ending in *p_sweep* and then handle them by reversing their order in the Y-structure.

⟨*handle passing and ending segments*⟩≡
```
seq_item sit = X_structure.inf(event);
if (sit == nil) sit = Y_structure.lookup(SEGMENT(p_sweep,p_sweep));
seq_item sit_succ  =     nil;
seq_item sit_pred  =     nil;
seq_item sit_pred_succ = nil;
seq_item sit_first =     nil;
if (sit != nil)
{ ⟨determine passing and ending segments⟩
  ⟨reverse order of passing segments⟩
}
```

We first determine whether there is any segment passing through or ending in *p_sweep*. Recall that the current event is ⟨*p_sweep*, *sit*⟩.

If $sit \neq nil$, the segment associated with *sit* contains *p_sweep*. If $sit = nil$, there is no pair of adjacent non-overlapping segments in the Y-structure intersecting in *p_sweep*. However, there may be a bundle of overlapping segments in the Y-structure that contain *p_sweep*. We can decide whether there is such a bundle and determine some segment in the bundle by locating the point *p_sweep* in the Y-structure[21]. We defined the comparison function for segments such that a search for the trivial segment (*p_sweep*, *p_sweep*) in the Y-structure is successful iff the Y-structure contains a segment containing *p_sweep*.

If there is no segment in the Y-structure containing *p_sweep*, there is nothing to do. Assume otherwise. Then *sit* points to one such segment. We determine all such segments. The corresponding items form a subsequence of the Y-structure, see Figure 10.51. We compute the first (*sit_first*) and last (*sit_last*) item of this bundle of items and also the predecessor (*sit_pred*) and successor (*sit_succ*) item of the bundle. We also store in *sit_pred_succ* the successor of *sit_pred* before the insertion, i.e, *sit_first*.

The items in the bundle are easily recognized by their informations. The information of every item in the bundle except for the last is either equal to the current event item *event* or equal to the successor item in the Y-structure (in the case of a segment overlapping with its successor). The information of the last item in the bundle is either *nil* or an item in the X-structure different from *event* (such an item stands for an intersection with *sit_succ*).

We determine the items in the bundle as follows. Starting at *sit* we first walk up until *sit_succ* is reached. Then we walk down to *sit_pred*. During the downward walk we also start to update the data structures. For every segment $s$ in the bundle we do the following:

- We add an edge to $G$ connecting *last_node*[$s$] and $v$ and label it with $s$. The new edge gets its direction from the original segment containing it, if *embed* is *false*, and is directed from $v$ to *last_node*[$s$], if *embed* is *true*.

[21] The Y-structure contains segments and hence only segments can be located in it. In order to locate the point *p_sweep* in the Y-structure, we locate the zero-length segment (*p_sweep*, *p_sweep*) instead.

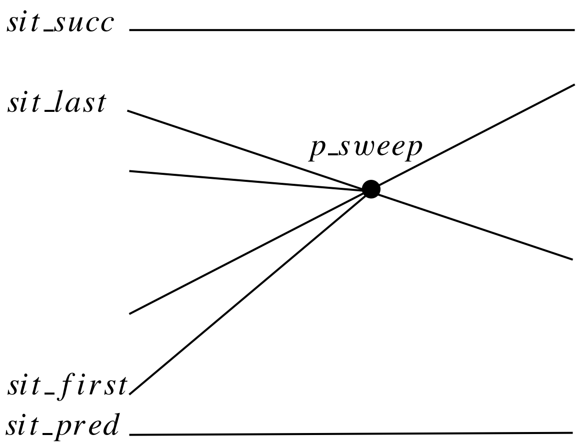


**Figure 10.51** The items *sit_pred*, *sit_first*, *sit_last*, and *sit_succ*.

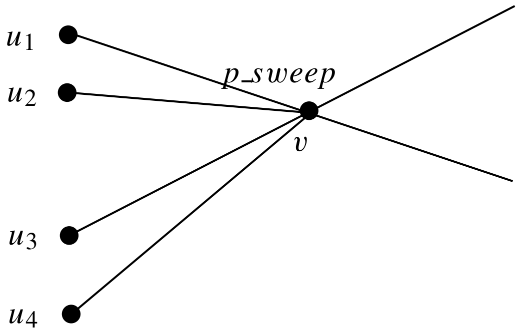


**Figure 10.52** The edges out of $v$ are constructed in the order $(v, u_1)$, $(v, u_2)$, $(v, u_3)$, $(v, u_4)$.

- If $s$ ends at *p_sweep* then we delete it from the Y-structure. If the predecessor segment overlaps with $s$, we copy the information about the successor segment of $s$ (if any) to the predecessor and set a flag that the downward walk is not finished yet.
- If $s$ continues through *p_sweep* then we change the intersection information associated with it to *nil* and set *last_node* to $v$.

We explain why we direct the edge constructed for a segment $s$ from $v$ to *last_node*$[s]$ if *embed* is *true*. Since *new_edge* appends the edge constructed to the list of outgoing edges of $v$ and since we construct edges during the downward walk the edges out of $v$ will be constructed in their proper counter-clockwise order, see Figure 10.52. We will exploit this fact when we construct the planar embedding of $G$ in the post-processing step.

The identification of the subsequence of segments incident to *p_sweep* takes constant time per element of the sequence. Moreover, the constant is small since the test of whether $p$ is incident to a segment involves no geometric computation but only identity tests between items. The code is particularly simple due to our sentinel segments: *sit* can never be the first or last item of the Y-structure.

⟨*determine passing and ending segments*⟩≡

```
// walk up
while ( Y_structure.inf(sit) == event ||
        Y_structure.inf(sit) == Y_structure.succ(sit) )
  sit = Y_structure.succ(sit);
sit_succ = Y_structure.succ(sit);
seq_item sit_last = sit;
if ( use_optimization ) { ⟨optimization, part 1⟩ }
// walk down
bool overlapping;
do
{ overlapping = false;
  s = Y_structure.key(sit);
  if ( !embed && s.source() == original[s].source() )
    G.new_edge(last_node[s], v, s);
  else
    G.new_edge(v, last_node[s], s );
  if ( identical(p_sweep,s.target()) )   // ending segment
  {
     seq_item it = Y_structure.pred(sit);
     if ( Y_structure.inf(it) == sit )
     { overlapping = true;
       Y_structure.change_inf(it, Y_structure.inf(sit));
     }
     Y_structure.del_item(sit);
     sit = it;
   }
   else  // passing segment
   {
     if ( Y_structure.inf(sit) != Y_structure.succ(sit) )
       Y_structure.change_inf(sit, seq_item(nil));
     last_node[s] = v;
     sit = Y_structure.pred(sit);
   }
} while ( Y_structure.inf(sit) == event || overlapping ||
          Y_structure.inf(sit) == Y_structure.succ(sit) );
sit_pred = sit;
sit_first = Y_structure.succ(sit_pred);
sit_pred_succ = sit_first;
```

All segments in the bundle starting with *sit_first* and ending in *sit_last* pass through node $v$ and moving the sweep line through *p_sweep* changes the order of these segments in the Y-structure. More precisely, if $s$ and $s'$ are two segments passing through *p_sweep* then moving the sweep line through *p_sweep* reverses their order iff $s$ and $s'$ do not overlap.

If the bundle is non-empty, we update its order as follows: first we reverse all subsequences of overlapping segments and then we reverse the entire bundle, see Figure 10.53.

The bundle of segments passing through *p_sweep* is empty iff *sit_first* is equal to *sit_succ*.

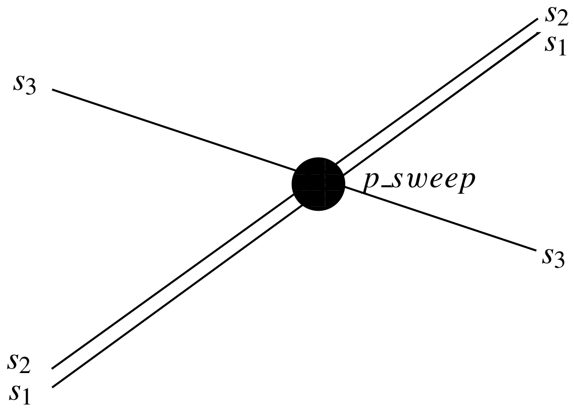


**Figure 10.53** Three segments passing through *p_sweep*, two of them overlapping. The order of the segments is reversed, but the order within the sub-bundle of overlapping segments is retained.

⟨*reverse order of passing segments*⟩≡

```
sit = sit_first;
// reverse subsequences of overlapping segments  (if existing)
while ( sit != sit_succ )
{ seq_item sub_first = sit;
  seq_item sub_last  = sub_first;
  while (Y_structure.inf(sub_last) == Y_structure.succ(sub_last))
    sub_last = Y_structure.succ(sub_last);
  if ( sub_last != sub_first )
    Y_structure.reverse_items(sub_first, sub_last);
  sit = Y_structure.succ(sub_first);
}
// reverse the entire bundle
if ( sit_first != sit_succ )
  Y_structure.reverse_items(Y_structure.succ(sit_pred),
                            Y_structure.pred(sit_succ));
```

**Insertion of Starting Segments:** The last step in handling the event point *p_sweep* is to insert all segments starting at *p_sweep* into the Y-structure and to test the new pairs of adjacent items (*sit_pred*, ...) and (..., *sit_succ*) for possible intersections. If there were no segments passing through or ending in *p_sweep* then the items *sit_succ* and *sit_pred* still have the value *nil* and we have to compute them now.

We use the priority queue *seg_queue* to find the segments to be inserted. As long as the first segment in *seg_queue* starts at *p_sweep*, i.e., *next_seg.source*( ) is identical[22] to *p_sweep*, we remove it from the queue and locate it in the Y-structure. Let *s_it* be the item returned by *locate* and let *p_sit* be its predecessor.

We insert *next_seg* after *s_it* into the Y-structure; this will add an item *sit* to the Y-structure. We set the information of *sit* to *s_sit* if the new segment overlaps with the segment associated

[22] Recall that we ensured that endpoints of internal segments that are equal are identical.

with *s_sit* and we set it to *nil* otherwise. Similarly, if the new segment overlaps with the segment associated with *p_sit* we change the information of *p_sit* to *sit*.

We associate the new item *sit* with the right endpoint of *next_seg* in the X-structure; note that the point is already there but it does not have its link to the Y-structure yet. We also set *last_node*[$s$] to $v$, and if *sit_succ* and *sit_pred* are still undefined, i.e, there was no segment passing through or ending in *p_sweep*, we set them to the successor and predecessor of the new item, respectively, and we set *sit_pred_succ* to *sit_succ*.

⟨*insert starting segments*⟩≡

```
while ( identical(p_sweep,next_seg.source()) )
{ seq_item s_sit = Y_structure.locate(next_seg);
  seq_item p_sit = Y_structure.pred(s_sit);

  s = Y_structure.key(s_sit);

  if ( orientation(s, next_seg.start()) == 0 &&
       orientation(s, next_seg.end()) == 0 )
     sit = Y_structure.insert_at(s_sit, next_seg, s_sit);
  else
     sit = Y_structure.insert_at(s_sit, next_seg, seq_item(nil));

  s = Y_structure.key(p_sit);

  if ( orientation(s, next_seg.start()) == 0 &&
       orientation(s, next_seg.end()) == 0 )
    Y_structure.change_inf(p_sit, sit);

  X_structure.insert(next_seg.end(), sit);
  last_node[next_seg] = v;

  if ( sit_succ == nil )
  { sit_succ = s_sit;
    sit_pred = p_sit;
    sit_pred_succ = sit_succ;
  }

  // delete minimum and assign new minimum to next_seg

  seg_queue.del_min();
  next_seg = seg_queue.inf(seg_queue.find_min());
}
```

**Computing New Intersections:** If *sit_pred* still has the value *nil*, there were no ending, passing or starting segments and hence *p_sweep* is an isolated point and we are done. Isolated points result from segments of length zero.

So assume that *sit_pred* exists. We have to update its information field (which still has the value from before the event). We set it to *nil* if there is no intersection between *sit_pred* and its successor. If the intersection exists, we insert it into the X-structure and set the information field of *sit_pred* to it. If there are segments leaving *p_sweep*, i.e, *sit_pred* is not the predecessor of *sit_succ*, we also check for an intersection between *sit_succ* and its predecessor.

⟨*compute new intersections and update X-structure*⟩≡

```
if ( sit_pred != nil )
{ if ( !use_optimization )
  { Y_structure.change_inf(sit_pred,seq_item(nil));
    compute_intersection(X_structure, Y_structure, sit_pred);
    sit = Y_structure.pred(sit_succ);
    if ( sit != sit_pred )
      compute_intersection(X_structure, Y_structure, sit);
  }
  else
  { ⟨optimization, part 2⟩ }
}
```

The function *compute_intersection* takes an item *sit0* of the Y-structure and determines whether the segment associated with *sit0* intersects the segment associated with its successor item *sit1* to the right of the sweep line. If so, it updates the X- and the Y-structure. Let $s_0$ and $s_1$ be the segments associated with *sit0* and *sit1*, respectively, and let $\ell_0$ and $\ell_1$ be the supporting lines of $s_0$ and $s_1$, respectively.

We know that $s_0$ intersects the sweep line $L$ before $s_1$. Thus $s_0$ and $s_1$ intersect right of the sweep line if the right endpoint of $s_1$ lies below or on $\ell_0$ (*orientation*(*s0*, *s1.target*( )) $\geq 0$) and the right endpoint of $s_0$ lies above or on $\ell_1$ (*orientation*(*s1*, *s0.target*( )) $\leq 0$).

If the segments intersect, we compute the point of intersection, call it $q$, by a call of *s0.intersection_of_lines*(*s1*, $q$), insert a new pair ($q$, *sit0*) into the X-structure and associate this pair with *sit0* in the Y-structure.

⟨*geometric primitives*⟩+≡

```
static void compute_intersection(sortseq<POINT,seq_item>& X_structure,
                    sortseq&lt;SEGMENT,seq_item&gt;& Y_structure, seq_item sit0)
{ seq_item sit1 = Y_structure.succ(sit0);
  SEGMENT  s0   = Y_structure.key(sit0);
  SEGMENT  s1   = Y_structure.key(sit1);

  if ( orientation(s0,s1.target()) <= 0 &&
       orientation(s1,s0.target()) >= 0 )
  { POINT q;
    s0.intersection_of_lines(s1,q);
    Y_structure.change_inf(sit0, X_structure.insert(q,sit0));
  }
}
```

**Post Processing:** We associate with each edge of $G$ an input segment containing it. This is easily done as each edge has an internal segment associated with it. Thus we only have to replace $G[e]$ by *original*[$G[e]$].

The graph $G$ constructed during the sweep is planar but is not in the form of a planar map yet. In particular, the order of the adjacency lists depends on the insertion order.

When *embed* is *true*, we turn $G$ into a planar map.

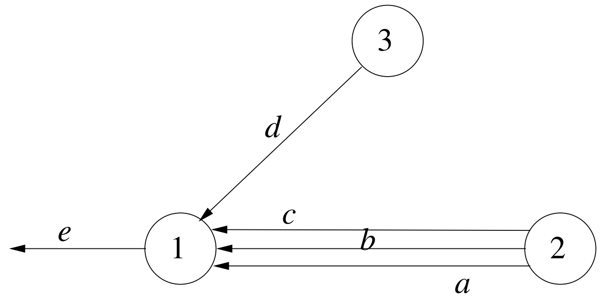


**Figure 10.54** Before the call of embedding there is only one edge leaving node 1, namely, the edge $e$. There are three parallel edges (2, 1); their counter-clockwise order around node 2 is in decreasing order of ID-number. We need to add the reversals of the edges $a$, $b$, $c$, and $d$ to the list of edges out of 1. Sorting the edges by increasing slope and edges of equal slope by ID-number gives the desired order.

⟨*post processing*⟩≡

```
if (embed) construct_embedding(G);
edge e;
forall_edges(e, G) G[e] = original[G[e]];
```

When *embed* is *true* all edges of $G$ are directed from right to left (vertical edges are directed downwards). Moreover, the edges out of any node are already in their proper counter-clockwise order.

In order to turn $G$ into a planar map we need to add the reversal of every edge and to insert the new edges at their proper position into the adjacency lists.

Edge reversals are directed from left to right (the reversal of a vertical edge is directed upwards). The proper order of edge reversals is therefore by slope. Reversals of parallel edges should be ordered by ID-number. Consider Figure 10.54.

Let $R$ be a copy (!!!) of the set of all edges of $G$. We use $R$ instead of $E$ to indicate that $R$ represents the set of edge reversals. We sort the edges in $R$ according to slope and then add for each edge $e$ in $R$ the edge $(target(e), source(e))$ to $G$. Since new edges are appended to the lists of outgoing edges, this will result in properly ordered adjacency lists.

⟨*embedding*⟩≡

```
class sweep_cmp_edges : public leda_cmp_base<edge>
{
  const GRAPH<POINT,SEGMENT>& G;
public:
  sweep_cmp_edges(const GRAPH<POINT,SEGMENT>& g): G(g) {}
  int operator()(const edge& e1, const edge& e2) const
  { SEGMENT s1 = G[e1];
    SEGMENT s2 = G[e2];
    int c = cmp_slopes(s1,s2);
    if (c == 0) c = compare(ID_Number(s1),ID_Number(s2));
    return c;
  }
};
```

```
static void construct_embedding(GRAPH<POINT,SEGMENT>& G)
{
  list<edge> R = G.all_edges();
  sweep_cmp_edges cmp(G);
  R.sort(cmp);
  edge e;
  forall(e,R)
  { edge r = G.new_edge(target(e),source(e),G[e]);
    G.set_reversal(e,r);
  }
}
```

In the post-processing step we first compute the embedding and then replace internal segments by input segments. It would be incorrect to change the order of two steps: first, the ordering of the Y-structure is an ordering on internal segments and we must use the same ordering in the embedding step. Second, the input may contain multiple occurrences of the same segment and the ordering by ID-number does not break ties between identical segments.

**An Optimization:** The running time of SWEEP_SEGMENTS is $O((n+s)\log(n+m)+m)$ where $n$ is the number of segments, $s$ is the number of nodes of $G$ and $m$ is the number of edges of $G$. If there are no overlapping segments then $m = O(n+s)$ since $G$ is planar. In the presence of overlapping segments, $m$ may be as large as $n(n+s)$. The time bound can be seen as follows. There are $O(n+k)$ lookups, insertions, and deletions in the X- and Y-structure, each for a cost of $O(\log(n+m))$. Observe that $n+m$ is an upper bound on the number of items in the Y-structure and that $n+s$ is an upper bound on the number of items in the X-structure. Since $s \le n^2$ we have $\log(n+s+m) = O(\log(n+m))$. The total number of items handled by the *reverse_items* operations on the Y-structure is $O(m)$. Since the cost of *reverse_items* is proportional to the number of items reversed, the total cost for all *reverse_items* operations is $O(m)$. The number of operations on $G$ is $O(n+k+m)$, each for a cost of $O(1)$.

Experiments show that a significant fraction of the running time is spent in the geometric primitives *sweep_cmp* and *compute_intersection*, in particular, if the rational kernel is used (which we recommend). The rational kernel has a built-in floating point filter, i.e., all geometric tests are first performed in floating point arithmetic, the rounding error is estimated, and only if the error estimation indicates that the result of the floating point computation may be wrong, the computation is repeated with exact arithmetic. The floating point filter is discussed in detail in Section 8.7.

The function *compute_intersection* performs orientation tests and computes an intersection point. The floating point filter applies to the orientation tests but does not apply to the computation of intersection points since constructions of new points are always performed with exact arithmetic.

The function *compute_intersection* is called whenever two segments become adjacent in

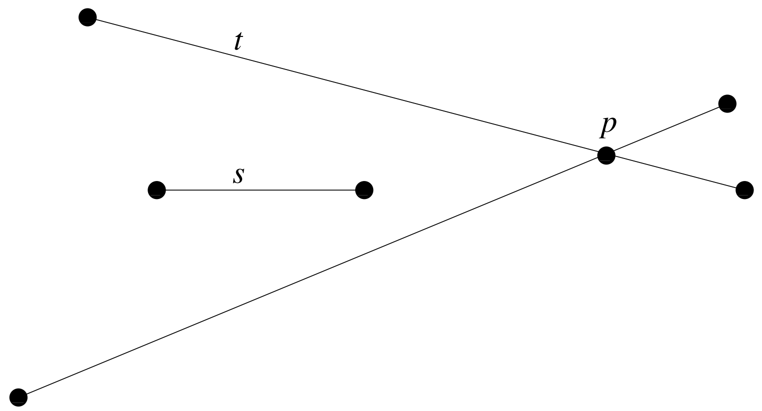


**Figure 10.55** The intersection $p$ is first discovered when $t$ is inserted into the Y-structure and is rediscovered when $s$ is removed from the Y-structure.

the Y-structure. Segments may become adjacent in the Y-structure more than once, see Figure 10.55. We show how to avoid the recomputation of intersections.

We maintain a dictionary *inter_dic* which maps pairs of segments to items in the X-structure. The appropriate data type is a two-dimensional map.

⟨*local declarations*⟩+≡

```
map2&lt;SEGMENT,SEGMENT,seq_item&gt;   inter_dic(nil);
```

Whenever a pair of segments that is adjacent in the Y-structure becomes non-adjacent we store their intersection in the dictionary and whenever a pair of segments becomes adjacent we consult the dictionary to find out whether their intersection was already computed.

When processing an event two intersections may get lost. Consider the sequence of items corresponding to segments passing through or ending in *p_sweep*. Let *sit_last* be the last item in this sequence and let *sit_pred* and *sit_succ* be the items before and after the sequence, respectively; *sit_last* does not exist if there are no segments passing through or ending in *p_sweep*.

Sweeping through *p_sweep* reverses the subsequence starting with *sit_first* and ending with *sit_last* and hence two intersections can get lost, the intersection stored in *sit_last* and the intersection stored in *sit_pred*. The intersection stored in *sit_last* is with the segment associated with *sit_succ* and the intersection stored in *sit_pred* is with the segment associated with the successor of *sit_pred*. This is the item *sit_pred_succ*.

⟨*optimization, part 1*⟩≡

```
seq_item xit = Y_structure.inf(sit_last);
if (xit) { SEGMENT s1 = Y_structure.key(sit_last);
           SEGMENT s2 = Y_structure.key(sit_succ);
           inter_dic(s1,s2) = xit;
         }
```

⟨*optimization, part 2*⟩≡

```
seq_item xit = Y_structure.inf(sit_pred);
```

```
if ( xit )
{ SEGMENT s1 = Y_structure.key(sit_pred);
  SEGMENT s2 = Y_structure.key(sit_pred_succ);  // sit_first
  inter_dic(s1,s2) = xit;
  Y_structure.change_inf(sit_pred, seq_item(nil));
}
compute_intersection(X_structure, Y_structure,inter_dic,sit_pred);
sit = Y_structure.pred(sit_succ);
if ( sit != sit_pred )
  compute_intersection(X_structure, Y_structure,inter_dic,sit);
```

We also need to change the function *compute_intersection*. Before computing an intersection point we check whether the two segments already have an intersection event in the X-structure by a lookup in *inter_map*. If the lookup fails we compute the intersection and add it to the X-structure.

⟨*geometric primitives*⟩+≡

```
static void compute_intersection(sortseq<POINT,seq_item>& X_structure,
                  sortseq&lt;SEGMENT,seq_item&gt;& Y_structure,
                  const map2&lt;SEGMENT,SEGMENT,seq_item&gt;& inter_dic,
                  seq_item sit0)
{ seq_item sit1 = Y_structure.succ(sit0);
  SEGMENT  s0   = Y_structure.key(sit0);
  SEGMENT  s1   = Y_structure.key(sit1);
  if ( orientation(s0,s1.target()) <= 0 &&
       orientation(s1,s0.target()) >= 0 )
  {
    seq_item it = inter_dic(s0,s1);
    if ( it == nil)
    { POINT q;
      s0.intersection_of_lines(s1,q);
      it = X_structure.insert(q,sit0);
    }
    Y_structure.change_inf(sit0, it);
  }
}
```

### 10.7.4 *Experimental Evaluation of the Sweep Line Algorithm*

We report about tests for *three kinds of test data*, namely random, difficult, and highly degenerate inputs, *three different implementations of points and segments*, namely the floating point kernel (FK), the rational kernel (RK) and the rational kernel with turned-off floating point filter ($FK^-$), and *with and without the optimization*. We describe the test data, list running times, and comment on the results.

**Random Inputs:** The random data set consists of $n$ segments whose endpoints have random $k$ bit coordinates. Table 10.8 gives the number of nodes and edges of the output graph and the running time for $n = 200$ and different values of $k$. The experiments indicate that the optimization described above and the floating point filter are effective. The optimization

| $k$ | $V$ | $E$ | RK$^-$ | RK$^-$O | RK | RKO | FK | FKO |
|---|---|---|---|---|---|---|---|---|
| 10 | 4813 | 9028 | 2.27 | 2 | 1.2 | 1.09 | 0.73 | 0.67 |
| 20 | 4742 | 8884 | 2.63 | 2.19 | 1.31 | 1.1 | 0.7 | 0.67 |
| 30 | 5467 | 10334 | 3.07 | 2.57 | 1.52 | 1.26 | 0.8 | 0.77 |
| 40 | 5478 | 10356 | 3.78 | 3.13 | 1.69 | 1.38 | 0.81 | 0.77 |
| 50 | 5168 | 9736 | 3.66 | 3.13 | 1.62 | 1.3 | 0.76 | 0.73 |
| 60 | 5558 | 10516 | 4.36 | 3.59 | 1.81 | 1.43 | 0.82 | 0.79 |
| 70 | 5909 | 11218 | 5.2 | 4.23 | 2.13 | 1.6 | 0.86 | 0.83 |
| 80 | 5174 | 9748 | 4.75 | 3.78 | 1.86 | 1.43 | 0.78 | 0.74 |
| 90 | 4808 | 9016 | 4.86 | 3.82 | 1.77 | 1.34 | 0.71 | 0.68 |
| 100 | 5080 | 9560 | 5.92 | 4.5 | 2.12 | 1.54 | 0.75 | 0.73 |

**Table 10.8** 200 random segments, coordinates are random $k$-bit integers. An "O" indicates the use of the optimization.

is more effective for the rational kernels because the computation of intersections is more costly in exact arithmetic. Floating point arithmetic is faster than exact arithmetic but the difference is never more than a factor of two in running time. We have to admit though that the difference can be made arbitrarily larger by choosing larger values of $k$.

**Difficult Inputs:** Let $size = 2^k$ and let $y = 2size/(n-1)$. The random data set consists of $n$ segments where the $i$-th segment has endpoints $(size + rx1,\ size + i \cdot y + ry1)$ and $(3 \cdot size + rx2,\ 3 \cdot size - i \cdot y + ry2)$ and $rx1$, $rx2$, $ry1$, $ry2$ are random integers in $[-s, s]$ for some small integer $s$. For $s = 0$ all segments in the difficult data set pass through the point $(2 \cdot size,\ 2 \cdot size)$, and for small but non-zero values of $s$ they intersect in the neighborhood of this point. Table 10.9 gives the results for the difficult data set with $s = 10$, $k = 10, 20, \ldots$ , 100, and $n = 200$. The floating point filter and the optimization are again quite effective. The floating point implementation produced incorrect results for all values of $k$; the floating point implementation does, however, work correctly for smaller values of $n$ and/or larger values of $s$.

**Highly Degenerate Inputs:** The highly degenerate test set consists of $n$ segments with random coordinates in a small grid with side length $s$. For example, for $n = 100$ and $s = 10$ one should expect a large number of degeneracies. We used this test set to support our claim that the algorithm handles all degeneracies. We do not report running times for the highly degenerate inputs.

The readers may perform their own experiments by running either the sweep-segments-demo in xlman or the sweep_time program in the demo directory.

We were surprised by two outcomes of our experiments.

First, we expected the implementation using the rational kernel to be much slower than the floating point computation and not just by a factor of two. We achieve the small factor by the use of the floating point filter, by the optimization which avoids the costly recomputation

| $k$ | $V$ | $E$ | RK$^-$ | RK$^-$O | RK | RKO | FK | FKO |
|---|---|---|---|---|---|---|---|---|
| 10 | 20134 | 39669 | 9.84 | 8.29 | 5.07 | 4.46 | error | error |
| 20 | 20298 | 39997 | 11.75 | 9.71 | 5.65 | 4.64 | error | error |
| 30 | 20296 | 39994 | 12.33 | 10.5 | 6.04 | 4.88 | error | error |
| 40 | 20298 | 39997 | 14.79 | 11.71 | 6.5 | 5.13 | error | error |
| 50 | 20300 | 40000 | 16.12 | 12.5 | 6.7 | 5.22 | error | error |
| 60 | 20298 | 39997 | 16.32 | 12.95 | 6.91 | 5.45 | error | error |
| 70 | 20300 | 40000 | 18.77 | 14.84 | 7.51 | 5.69 | error | error |
| 80 | 20300 | 40000 | 19.82 | 15.91 | 7.62 | 5.72 | error | error |
| 90 | 20298 | 39997 | 21.27 | 16.25 | 7.68 | 5.71 | error | error |
| 100 | 20296 | 39994 | 24.61 | 18.39 | 8.58 | 6.24 | error | error |

**Table 10.9** The difficult example with 200 segments. An "O" indicates the use of the optimization and error indicates that the computation with the floating point kernel gave the incorrect result.

of intersections, and by the observation that many equality tests for points can be replaced by tests for identity of points.

Second, we expected the floating point implementation to have difficulties with the difficult example. However, we were surprised by the fact that it never crashed. It always produced an output, albeit an incorrect one. We try to explain this phenomenon by arguing that the program does not crash as long as the sentinels are handled correctly, i.e, the segments *lower_sentinel* and *upper_sentinel* have all segments between them and all intersection points precede *pstop*. We do not care what the geometric tests do with segments that are not sentinels. If sentinels are handled correctly, every lookup in the Y-structure will return an item different from the first item in the Y-structure[23]. Also the walks performed in the Y-structure will determine a subsequence that does not include the sentinel items. For this reason none of the operations on the Y-structure will fail; i.e., it will never happen that we ask for the successor of the last or the predecessor of the first item. Also since *pstop* is handled correctly, we will never attempt to extract *next_seg* from an empty *seg_queue*.

***Exercises for 10.7***

1 Let $G_0$ and $G_1$ be graphs of type *GRAPH<POINT, SEGMENT>*. Write a function that checks whether the graphs are isomorphic, i.e., whether there are bijections $i_V : V_0 \rightarrow V_1$ and $i_E : E_0 \rightarrow E_1$ such that $G_0[v] = G_1[i_V(v)]$ for all nodes of $G_0$ and such that $i_E(e) = (i_V(v), i_V(w))$ and $G_0[e] = G_1[i_E(e)]$ for all edges $e = (v, w)$ of $G_0$.

2 Use the solution to the previous exercise to write a function that runs two implementations of SEGMENT_INTERSECTION and then checks the computed graphs for isomorphism.

[23] This sentence requires knowledge of the implementation of sorted sequences. The implementation is such that if the comparisons with the first and the last element of the sorted sequence are correct and the outcome of any other comparison is arbitrary, lookup will not return the first element.

3 Write a trivial implementation of *SEGMENT_INTERSECTION*($G$, *report*) that simply checks every pair of segments for an intersection.
4 Extend the sweep line algorithm or any of the other algorithms such that it computes the trapezoidal decomposition induced by a set of segments.

## 10.8 **Polygons**

We define the types polygon and generalized polygon. A polygon is an open region of the plane whose boundary is a closed polygonal chain[24] and a generalized polygon is anything that can be obtained from polygons by regularized set operations. Both classes offer functions for point location, for intersection with lines and segments, and for moving objects around. Generalized polygons offer, in addition, the regularized set operations complement, union, intersection, difference, and symmetric difference.

This section is structured as follows: in Section 10.8.1 we discuss the functionality of polygons and generalized polygons, in Section 10.8.2 we give the essentials of the implementation of polygons, in Section 10.8.3 we give the mathematics underlying the representation of generalized polygons, and in Section 10.8.4 we give the highlights of the implementation of generalized polygons.

We advise you to exercise the polygon demo in xlman before reading this section, see Figure 10.56.

### 10.8.1 ***Functionality***

A closed polygonal chain $P$ is a cyclic sequence $(p_0, p_1, \ldots, p_{n-1})$ of points. The points are called the vertices of the chain and the number of vertices is called the size of the chain. The vertices of a closed polygonal chain are indexed modulo the size $n$ of the chain, in particular, $p_n = p_0$. A closed polygonal chain induces a set $S(P)$ of segments, namely the set of segments $p_i p_{i+1}$, $0 \le i \le n-1$, connecting consecutive vertices. A closed polygonal chain is called *simple* if all nodes of the graph $G(S(P))$ defined by the segments in $S(P)$ have degree equal to two, i.e., if no two segments in $S(P)$ except for consecutive segments share a point. A closed polygonal chain $P$ is called *weakly simple* if the segments in $S(P)$ are disjoint except for common endpoints[25] and if the chain does not cross itself. Figure 10.57 shows some examples.

A weakly simple polygonal chain splits the plane into an unbounded region and one or more bounded regions. For a simple polygonal chain there is just one bounded region. When a weakly simple polygonal chain $P$ is traversed either the bounded region is consistently to the left of $P$ or the unbounded region is consistently to the left of $P$; this follows from the fact that a weakly simple chain does not cross itself. We say that $P$ is positively oriented in the former case and negatively oriented in the latter case. We call the region

[24] A precise definition is given below.
[25] It is allowed that segments that are not consecutive on $P$ share an endpoint.

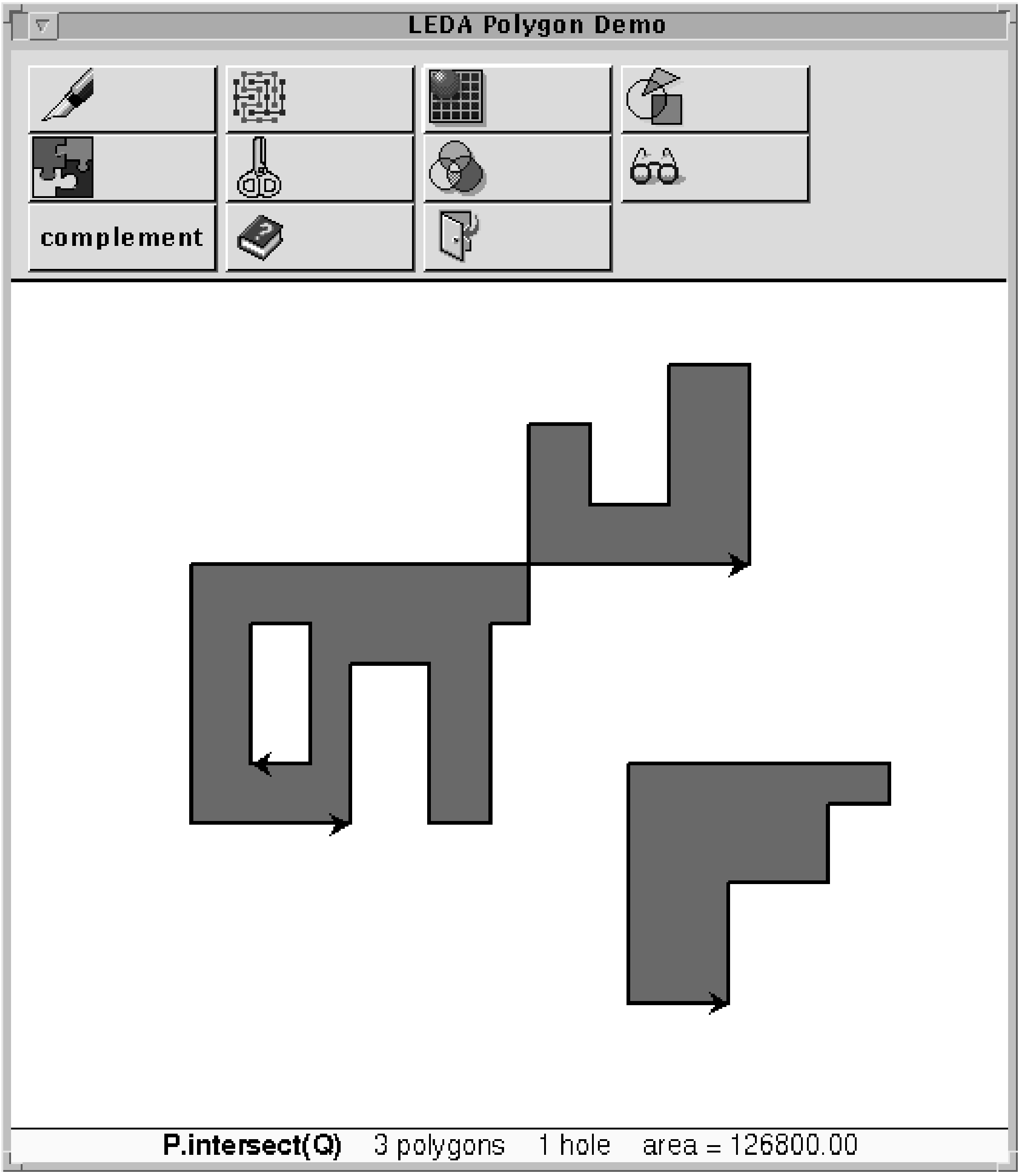


**Figure 10.56** A screen shot of the polygon demo in xlman. The display shows a generalized polygon. The boundary cycles are indicated by arrows and the inside of the polygon is shaded. The various buttons allow the user to construct polygons by mouse input or by calling generators, to force vertices to a grid, to compute intersections, unions, differences, and symmetric differences, to perform point location queries, and to compute complements.

to the left of $P$ the positive side of $P$. We overload notation and use $P$ also to denote the positive side of $P$, see Figure 10.58. The positive side of $P$ is an open set and $P$ is its boundary.

Frequently, we do not want to distinguish between a polygonal chain and the polygonal region defined by it. We use the word *polygon* to cover both aspects.

We have two classes of polygons: *rat_polygons* have *rat_points* as their vertices and *polygons* have *points* as their vertices. Both classes offer essentially the same function-

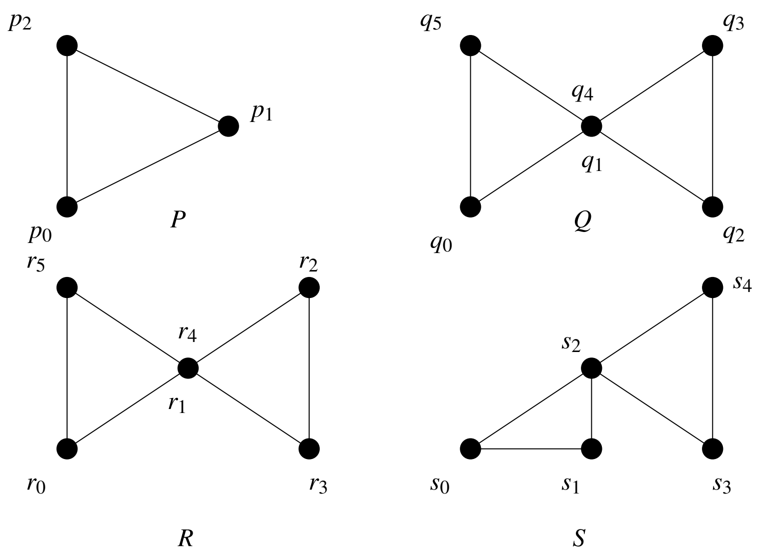


**Figure 10.57** $P$ is simple and $Q$ is weakly simple but not simple. $R$ is not weakly simple because it crosses itself at $r = r_1 = r_4$, and $S$ is not weakly simple since $s_2$ lies in the interior of another segment.

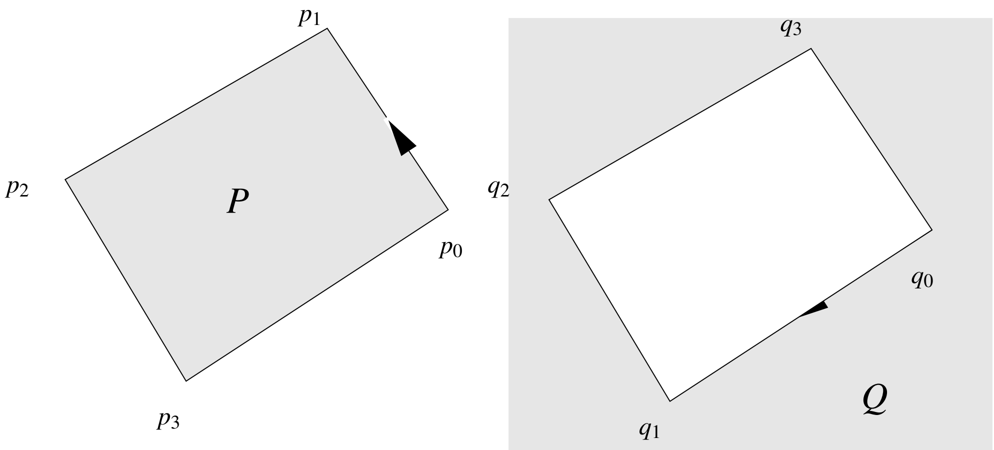


**Figure 10.58** The bounded region is to the left of $P$; $P$ is positively oriented. The unbounded region is to the left of $Q$, $Q$ is negatively oriented.

ality, but, of course, only *rat_polygons* guarantee correct results. We use *rat_polygons* in this section.

The declarations

```
rat_polygon P1;
rat_polygon P2(const list<rat_point>& pl,
               CHECK_TYPE check = rat_polygon::SIMPLE,
               bool respect_orientation =
                                   rat_polygon::RESPECT_ORIENTATION);
```

introduce polygons *P1* and *P2*; *P1* is initialized to the empty polygon and *P2* is initialized to the polygon with vertex sequence *pl*. The second argument takes one of the values

NO_CHECK, SIMPLE, WEAKLY_SIMPLE of a local enumeration type CHECK_TYPE. If *check* is SIMPLE, the polygon must be simple, and if *check* is WEAKLY_SIMPLE, the polygon must be weakly simple. The third argument takes one of the values RESPECT_ORIENTATION or DISREGARD_ORIENTATION. If *respect_orientation* is DISREGARD_ORIENTATION, the orientation of *pl* is chosen such that the bounded region with respect to *pl* lies to the left of *pl*. The meaning of this flag is undefined if *pl* is not weakly simple.

Simplicity and weak simplicity can also be checked by the functions

```
bool P.is_simple();
bool P.is_weakly_simple();
```

Assignment and copy constructor are available for polygons. The functions

```
list<rat_point>   P.vertices();
list<rat_segment> P.edges();
```

return the list of vertices and the list of segments of $P$, respectively. The second function is also available as *P.segments*( ).

Let $l$ be a line and let $s$ be a segment. The functions

```
list<rat_point> P.intersection(l);
list<rat_point> P.intersection(s);
```

return the crossings between the chain $P$ and $l$ or $s$, respectively. The function

```
rat_polygon P.complement()
```

returns the polygon whose list of vertices is the reversal of $P$'s list. If $P$ is weakly simple, the positive side of the complement is the negative side of $P$ and vice versa.

*The remaining functions for polygons assume that $P$ is weakly simple. Their meaning is undefined if $P$ is not weakly simple.* Recall that a weakly simple polygon $P$ splits the plane in an unbounded region and one or more bounded regions. Also recall that we designated the region(s) to the left of $P$ as the positive side of $P$ and use $P$ also for the positive side of $P$.

Let $p$ be a point. The function

```
int P.side_of(p);
```

returns the side of $P$ to which $p$ belongs, i.e., $+1$ if $p$ belongs to the positive side, 0 if $p$ lies on $P$, and $-1$ if $p$ belongs to the negative side, see Figure 10.59. The function

```
region_kind P.region_of(p);
```

returns the region with respect to $P$ to which $p$ belongs, i.e., BOUNDED_REGION if $p$ lies in the bounded region of $P$, ON_REGION if $p$ lies on $P$, and UNBOUNDED_REGION if $p$ lies in the unbounded region. One can also ask for the containment in a specific region by

```
bool P.inside(p);
bool P.on_boundary(p);
bool P.outside(p);
```

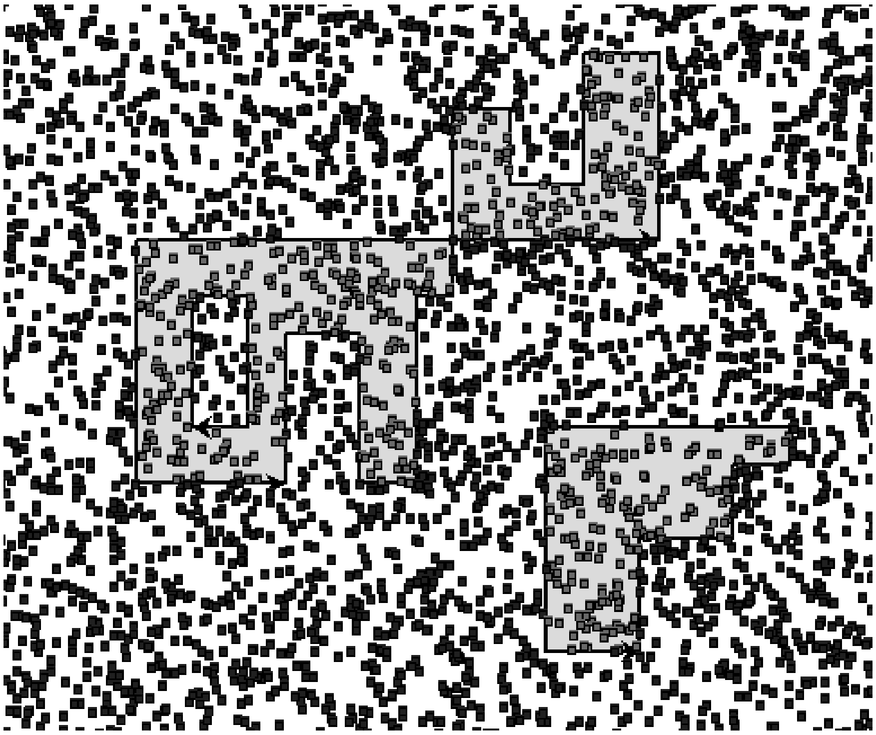

**Figure 10.59** Side-of tests: We performed side-of tests with respect to the generalized polygon of Figure 10.56 for 5000 random points. The points on the different sides are shown at different grey level.

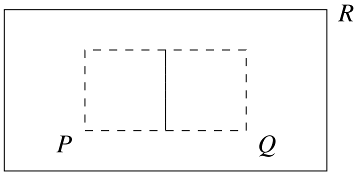


**Figure 10.60** The intersection of $P$ and $Q$ is a line segment; $R \setminus (P \cap Q)$ is a rectangle minus a line segment.

The function

```
RAT_TYPE P.area();
```

returns the signed area of the bounded region of $P$. The sign of the area is positive if $P$ is positively oriented and is negative if $P$ is negatively oriented.

We come to generalized polygons. The class of polygons is not closed under boolean operations. In fact, very strange objects can be generated from polygons by boolean operations, see Figure 10.60. The class of generalized polygons encompasses all sets that can be constructed from polygons by the so-called *regularized set operations*, see [Req80, TR80, Hof89]. We refer the reader to [Nef78] for the general case.

In order to define the regularized set operations we need to review some elementary concepts of topology. For a set $X$ we use int $X$, cl $X$, bd $X$, and cpl $X$ to denote its *interior*, *closure*, *boundary*, and *complement*, respectively. An open set $X$ is called *regular* if $X =$

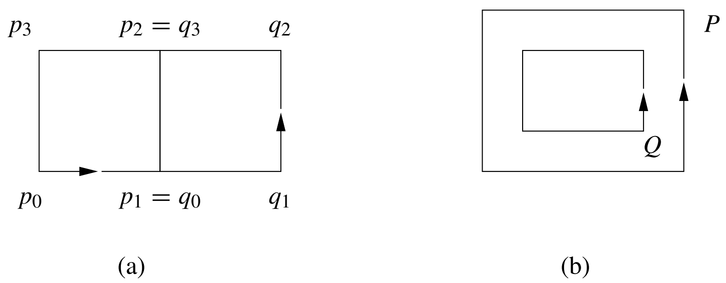


**Figure 10.61** In (a), the polygons $P$and $Q$ share an edge, $P \cap Q$ is a closed line segment, and $\text{reg}(P \cap Q)$ is the empty set. In (b), $P \setminus Q$ is the half-closed region between the cycles $P$ and $Q$; the chain $P$ does not belong to $P \setminus Q$ and the chain $Q$ belongs to it. The regularized set difference $\text{reg}(P \setminus Q)$ is the open region with boundaries $P$ and $Q$.

$\text{int}\,\text{cl}\,X$. The following sets are non-regular: the plane minus a single point or the plane minus a line. A set is called *polygonal* if its boundary consists of a finite number of points and open line segments. The regularization of a set $X$ is defined as $\text{int}\,\text{cl}\,X$; we use $\text{reg}\,X$ as a shorthand for $\text{int}\,\text{cl}\,X$. We show that regularization generates regular sets and that the regularized set operations[26] applied to regular polygonal regions generate regular polygonal regions, see Figure 10.61.

**Lemma 10**

***(a)*** *Let $X$ be any set. Then* $\text{reg}\,X$ *is regular.*
***(b)*** *Let $X$ be any open set. $X$ is regular iff $X$ and* $\text{int}\,\text{cpl}\,X$ *have the same boundary.*
***(c)*** *Let $P$ be a weakly simple polygonal chain. Then the bounded region and the unbounded region with respect to $P$ are regular polygonal sets.*
***(d)*** *If $P$ and $Q$ are regular polygonal regions then so are* $\text{reg}\,\text{cpl}\,P$, $\text{reg}(P \cap Q)$, $\text{reg}(P \cup Q)$, $\text{reg}(P \setminus Q)$, *and* $\text{reg}(P \oplus Q)$.

*Proof* We start with part (a). Let $X$ be any set and let $Y = \text{reg}\,X$. We need to show that $Y$ is regular. We have $Y \subseteq \text{cl}\,Y$ and hence $Y \subseteq \text{int}\,\text{cl}\,Y$ since $Y$ is open. We have $Y \subseteq \text{cl}\,X$ by definition of $Y$ and hence $\text{cl}\,Y \subseteq \text{cl}\,\text{cl}\,X = \text{cl}\,X$. Thus $\text{int}\,\text{cl}\,Y \subseteq \text{int}\,\text{cl}\,X = Y$.

We turn to part (b). Assume first that $X$ is regular, i.e., $X = \text{int}\,\text{cl}\,X$, and let $x$ be any point in the boundary of $X$. Then $x \in \text{cl}\,X \setminus X$ since $X$ is open. Assume that there is a neighborhood $U$ of $x$ such that $U \cap \text{int}\,\text{cpl}\,X = \emptyset$. Then $U \subseteq \text{cl}\,X$ and hence $x \in \text{int}\,\text{cl}\,X$, a contradiction to the regularity of $X$.

To prove the converse we observe that $X \subseteq \text{int}\,\text{cl}\,X$ since $X$ is open. We need to show that the containment is not proper. Consider any point $x \in \text{bd}\,X$. By assumption every neighborhood $U$ of $x$ has $U \cap \text{int}\,\text{cpl}\,X \neq \emptyset$. Thus $x \notin \text{int}\,\text{cl}\,X$ and hence $\text{int}\,\text{cl}\,X \subseteq X$.

For part (c) we observe that the boundary of the bounded as well as the unbounded region with respect to $P$ is equal to $P$ and hence both regions are certainly polygonal. The regularity of both regions follows from part (b) and the fact that $P$ is weakly simple.

[26] The regularized union of two sets $X$ and $Y$ is defined as $\text{reg}(X \cup Y)$; the definition of the other regularized set operations is analogous.

The results of the regularized set operations are certainly polygonal; regularity follows from part (a). □

The classes *rat_gen_polygon* and *gen_polygon* represent regular polygonal regions over the rational and the floating point kernel, respectively. In our examples we use *rat_gen_polygons*; *gen_polygon* stands for generalized polygon.

The constructors

```
rat_gen_polygon P;
rat_gen_polygon Q(rat_polygon R);
```

construct the empty generalized polygon and the generalized polygon corresponding to $R$, respectively. The second constructor requires that $R$ is a weakly simple polygon. There are two special generalized polygons, the empty one and the full one. The *full polygon* is the entire plane.

The functions

```
bool P.is_empty();
bool P.is_full();
```

return true if $P$ is the empty set or the entire plane, respectively.

If $p$ is a point and $P$ is a generalized polygon then

```
bool P.side_of(p)
```

returns $+1$ if $p \in P$, returns 0 if $p$ lies on $P$, and returns $-1$ otherwise, see Figure 10.59.

The function

```
region_kind P.region_of(p);
```

returns the region with respect to $P$ to which $p$ belongs, i.e., BOUNDED_REGION if $p$ lies in the bounded region of $P$, ON_REGION if $p$ lies on $P$, and UNBOUNDED_REGION if $p$ lies in the unbounded region. The bounded region of the empty polygon is empty and the bounded region of the full polygon is the entire plane.

The function

```
RAT_TYPE P.area();
```

returns the signed area of the bounded region of $P$. The sign of the area is positive if $P$ is bounded and is negative if $P$ is unbounded. This function cannot be applied to the full polygon.

For the following operations let $P$ and $Q$ be generalized polygons.

```
rat_gen_polygon P.complement()
```

returns the regularized complement of $P$ and

```
gen_rat_polygon P.unite(Q);
gen_rat_polygon P.intersection(Q);
gen_rat_polygon P.diff(Q);
gen_rat_polygon P.sym_diff(Q);
```

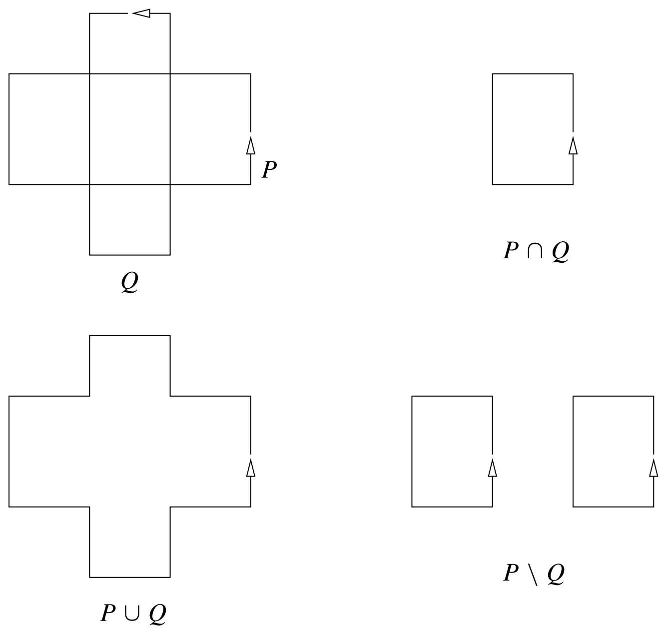


**Figure 10.62** Two polygons $P$ and $Q$ and the results of the three boolean operations $\cap$, $\cup$, and $\setminus$.

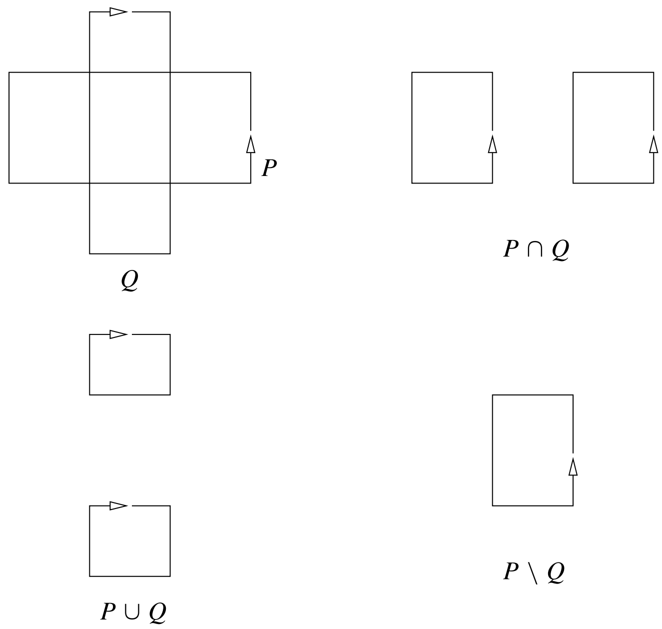


**Figure 10.63** Two polygons $P$ and $Q$ and the results of the three boolean operations $\cap$, $\cup$, and $\setminus$. Observe that the positive side of $Q$ is unbounded.

return reg($P \cup Q$), reg($P \cap Q$), reg($P \setminus Q$), and reg($P \oplus Q$), respectively. The word *union* is a reserved word of C++, hence the name *unite* for the union-operation. Figures 10.62 and 10.63 show some examples.

A generalized polygon can be represented by its boundary cycles as will be explained in Section 10.8.3. The function

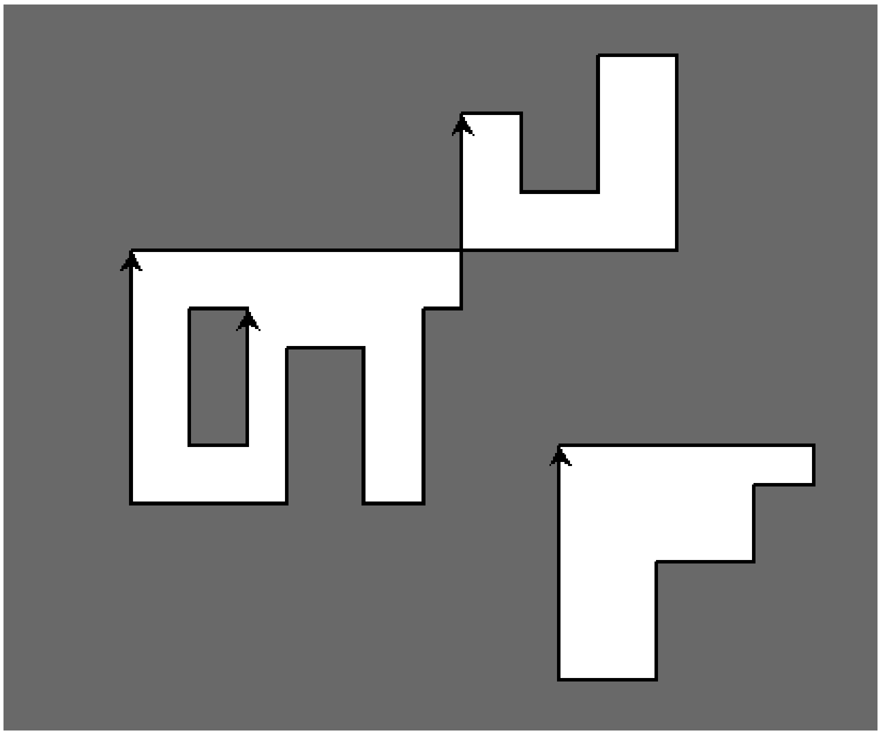

**Figure 10.64** The complement of the generalized polygon of Figure 10.56. Observe that the orientation of all boundary cycles is reversed.

```
list<rat_polygon> P.polygons();
```

returns the list of boundary cycles of $P$. The list is ordered according to nesting, i.e., if a boundary cycle $D$ is nested in a boundary cycle $C$, then $C$ is before $D$ in the list of boundary cycles.

### 10.8.2 *The Implementation of Polygons*

Polygons are a handle type, i.e., a polygon is realized as a pointer to a representation class (called *polygon_rep* and *rat_polygon_rep*, respectively) which contains the actual representation. The member function *ptr*( ) of class polygon returns the pointer to the representation object.

The representation consists of a list of points, a list of segments, four extreme points, and an integer which stores the orientation of the polygon. The orientation is positive if the bounded region is to the left of the polygon and is negative otherwise.

```
list<POINT>   pt_list;
list&lt;SEGMENT&gt; seg_list;
POINT xmin, ymin, xmax, ymax;
int orient;
```

Here, *pt_list* contains the list of points, *seg_list* contains the list of segments (the $i$-th segment in *seg_list* connects the $i$-th point in *pt_list* to the $i+1$-th point in *pt_list*), and *xmin*, *ymin*, *xmax*, and *ymax* are vertices with minimal $x$-coordinate, minimal $y$-coordinate, maximal $x$-coordinate, and maximal $y$-coordinate, respectively.

We will next discuss some of the member functions of *polygon*.

**The Signed Area of a Simple Polygon:** Assume that *seg_list* is the list of boundary segments of a simple polygon $P$. We show how to compute the signed area $A(P)$ of the bounded face of $P$. The sign of the area is positive if the bounded face lies to the left of $P$ and is negative otherwise.

**Lemma 11** *Let $P$ be a simple polygon and let $n$ be the number of segments in the boundary of $P$. For $0 \leq i < n$, let $p_i$ be the source point of the $i$-th boundary segment. Let $p$ be an arbitrary point in the plane and let $A_i = A(\Delta_i)$ be the signed area of the triangle $\Delta_i = (p, p_i, p_{i+1})$. Then*

$$A(P) = \sum_{0 \leq i < n} A_i$$

*is the signed area of $A$.*

*Proof* We use induction on $n$ and assume w.l.o.g. that the signed area is positive. Assume first that $P$ is a triangle, see Figure 10.65. If $p$ lies in the bounded face of $P$ or on $P$, the bounded face of $P$ is partitioned by the triangles $\triangle_0$, $\triangle_1$, and $\triangle_2$, and hence $A(P) = A(\triangle_0) + A(\triangle_1) + A(\triangle_2)$. If $p$ lies in the unbounded face of $P$, then $p$ can see either one or two edges of $P$. If $p$ can see one edge of $P$, say $p_0 p_1$, then

$$A(P) = |A(\triangle_1)| + |A(\triangle_2)| - |A(\triangle_0)| = A(\triangle_1) + A(\triangle_2) + A(\triangle_0),$$

where the second equality follows from the fact that $\triangle_1$ and $\triangle_2$ are positively oriented and $\triangle_0$ is negatively oriented. If $p$ can see two edges of $P$, say $p_0 p_1$ and $p_1 p_2$, then

$$A(P) = |A(\triangle_2)| - |A(\triangle_1)| - |A(\triangle_0)| = A(\triangle_2) + A(\triangle_1) + A(\triangle_0),$$

where the second equality follows from the fact that the orientation of $\triangle_2$ is positive and the orientations of $\triangle_0$ and $\triangle_1$ are negative. This completes the base step of the induction.

Assume next that $n \geq 4$. Then there is an $i$ such that the segment $p_i p_{i+2}$ is contained in the interior of $P$ [27]. Let $Q$ be the polygon obtained from $P$ by replacing the segments $p_i p_{i+1}$ and $p_{i+1} p_{i+2}$ by the segment $p_i p_{i+2}$. Then

$$A(P) = A(Q) + A(\triangle)$$

where $\triangle = (p_i, p_{i+1}, p_{i+2})$. Applying the induction hypothesis to $Q$ yields

$$A(Q) = \sum_{j=0}^{i-1} A(\triangle_j) + A(p, p_i, p_{i+2}) + \sum_{j=i+2}^{n-1} A(\triangle_j)$$

and applying the induction hypothesis to $\triangle$ yields

$$A(\triangle) = A(\triangle_i) + A(\triangle_{i+1}) + A(p_{i+2}, p_i, p) = A(\triangle_i) + A(\triangle_{i+1}) - A(p, p_i, p_{i+2}).$$

[27] Consider an arbitrary triangulation of $P$. The dual of the triangulation is a tree and hence there is at least one triangle in the triangulation which has two edges of $P$ in its boundary. The two edges are $p_i p_{i+1}$ and $p_{i+1} p_{i+2}$ for some $i$.

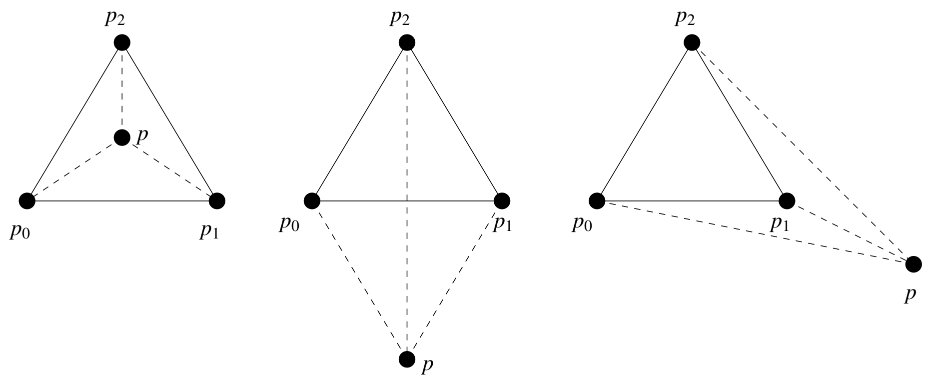


**Figure 10.65** Let $\triangle_i = (p, p_i, p_{i+1})$ for $i = 0, 1, 2$, and let $P = (p_0, p_1, p_2)$. Then $A(P) = A(\triangle_0) + A(\triangle_1) + A(\triangle_2)$ in all three cases.

Adding the two equations completes the induction step. □

The implementation follows directly from the lemma above.

⟨*polygon: compute area*⟩≡

```
static RAT_TYPE compute_area(const list&lt;SEGMENT&gt;& seg_list)
{
  if (seg_list.length() < 3) return 0;

  list_item it = seg_list.get_item(1);
  POINT     p  = seg_list[it].source();

  it = seg_list.succ(it);

  RAT_TYPE A  = 0;

  while (it)
  { SEGMENT s = seg_list[it];
    A += ::area(p,s.source(),s.target());
    it = seg_list.succ(it);
   }

  return A;
}
```

The time to compute the signed area of a polygon is $O(n)$. The constant factor in the $O$-expression is fairly large, in particular, with the rational kernel. Observe that the areas of $n$ triangles are computed and that an area computation of a triangle amounts to the evaluation of a $3 \times 3$ determinant.

**Determining the Orientation:** The simplest way to compute the orientation of a polygon $P$ is to take the sign of the area. This takes linear time but is slow; see the remark at the end of the preceding section. A faster approach is as follows.

Let $q$ be the lexicographically smallest vertex of $P$ and let $p$ and $r$ be the predecessor and successor vertices of $q$ on $P$. Then the orientation of $P$ is equal to the orientation of the triple $(p, q, r)$, see Figure 10.66. Observe that this statement is not true for an arbitrary vertex $q$; it is only true for a vertex that is extreme in some direction.

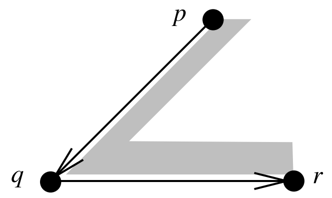


**Figure 10.66** The triple $(p, q, r)$ has positive orientation. If $q$ is the lexicographically smallest vertex of the polygon, the region to the left of the polygonal chain is bounded. This conclusion cannot be drawn for an arbitrary vertex.

The implementation of *orientation* follows directly from the preceding paragraph.

⟨*polygon: compute orientation*⟩≡

```
static int compute_orientation(const list&lt;SEGMENT&gt;& seg_list)
{ list_item q_it = seg_list.first();
  POINT q = seg_list[q_it].source();

  list_item it;
  forall_items(it,seg_list)
    if ( compare(seg_list[it].source(),q) < 0 )
    { q_it = it;
      q = seg_list[q_it].source();
    }

  POINT p = seg_list[seg_list.cyclic_pred(q_it)].source();
  POINT r = seg_list[seg_list.cyclic_succ(q_it)].source();

  return ::orientation(p,q,r);
}
```

**Point Containment:** Let $P$ be a weakly simple polygon. The function

```
region_kind P.region_of(const POINT& p) const
```

returns the region of $P$ containing $p$. In order to decide containment we first use the extreme vertices for a quick test. If $p$ lies to the left of *xmin* or to the right of *xmax* or below *ymin* or above *ymax*, we return UNBOUNDED_REGION. Next we check whether $p$ lies on $P$. Assume this is not the case, i.e., $p$ lies either in the bounded face or the unbounded face of $P$.

We use the following observation. Consider a vertical upward ray $r_p$ starting in $p$ and assume that $r_p$ does not pass through any vertex of $P$. Then $r_p$ intersects an odd number of segments of $P$ iff $p$ lies in the bounded region of $P$. The observation solves the problem iff $r_p$ does not pass through any vertex of $P$.

We use *perturbation* to extend the solution to arbitrary points $p$. If $p$ does not lie on $P$, the point $q$ obtained from $p$ by moving $p$ by an infinitesimal amount to the right belongs to the same face with respect to $P$ as $p$. Moreover, the vertical upward ray $r_q$ starting at $q$ does not pass through any vertex of $P$. In particular, $r_q$ does not intersect any vertical edge of $P$.

Consider a segment $s$ of $P$. If $s$ is vertical, $r_q$ does not intersect it. So assume that $s$ is not vertical. Let $a$ be the endpoint of $s$ with the smaller $x$-coordinate and let $b$ be the other endpoint of $s$. Then $r_q$ intersects $s$ if $x_a < x_q < x_b$ and $q$ lies to the right of the oriented line $\ell$ through $a$ and $b$. Here, we used $x_z$ to denote the $x$-coordinate of a point $z$. Since $x_q = x_p + \epsilon$ for an infinitesimal $\epsilon$, the first condition is equivalent to $x_a \le x_p < x_b$ and the second condition is equivalent to $p$ being to the right of $\ell$.

We obtain the following code.

⟨*polygon: region_of and side_of*⟩≡

```
region_kind POLYGON::region_of(const POINT& p) const
{
  // use extreme vertices for a quick test.
  int cx1 = POINT::cmp_xy(p,ptr()->xmin);
  int cx2 = POINT::cmp_xy(p,ptr()->xmax);
  int cy1 = POINT::cmp_yx(p,ptr()->ymin);
  int cy2 = POINT::cmp_yx(p,ptr()->ymax);
  if (cx1 < 0 || cx2 > 0 || cy1 < 0 || cy2 > 0) return UNBOUNDED_REGION;
  list&lt;SEGMENT&gt;& seglist = ptr()->seg_list;
  // check boundary segments
  list_item it;
  forall_items(it,seglist)
  { SEGMENT s = seglist[it];
    if (s.contains(p)) return ON_REGION;
  }
  // count intersections with vertical ray starting in p
  int count  = 0;
  forall_items(it,seglist)
  { SEGMENT s = seglist[it];
    POINT a = s.source();  POINT b = s.target();
    int orient = POINT::cmp_x(a,b);
    if ( orient == 0 ) continue;
    if ( orient > 0 ) { // a is right of b
                        leda_swap(a,b);
                      }
    if ( POINT::cmp_x(a,p) <= 0 && POINT::cmp_x(p,b) < 0
        && ::orientation(a,b,p) < 0 )
      count++;
  }
  return ( count % 2 == 0 ? UNBOUNDED_REGION : BOUNDED_REGION );
}
```

Given the function *region_of* it is easy to implement *side_of*. The positive side of $P$ is equal to the bounded region if $P$ is positively oriented and is equal to the unbounded region otherwise.

⟨*polygon: region_of and side_of*⟩+≡

```
int POLYGON::side_of(const POINT& p) const
{ region_kind k = region_of(p);
  switch (k) {
    case ON_REGION:        return 0;
    case BOUNDED_REGION:   return   ptr()->orient;
    case UNBOUNDED_REGION: return -(ptr()->orient);
    default:               assert( 0 == 1); return 0;
  }
}
```

**The Complement of a Polygon:** The complement of a weakly simple polygon is easy to compute. We simply reverse the list of segments. The complement has the opposite orientation.

⟨*polygon: complement*⟩≡

```
POLYGON POLYGON::complement() const
{ list&lt;SEGMENT&gt; R;
  SEGMENT s;
  forall(s,ptr()->seg_list) R.push(SEGMENT(s.target(),s.source()));
  return POLYGON(R, - orientation());
}
```

### 10.8.3 *The Mathematics of Generalized Polygons*

The purpose of this section is to give the mathematical underpinning for the representation of regular polygonal sets. We show that a regular polygonal set can be represented by its list of boundary cycles.

If $X$ is a regular polygonal set and $p$ is an arbitrary point in the plane the intersection $U \cap X$ for $U$ a sufficiently small neighborhood of $p$ has one of the following three forms:

- If $p$ is contained in (the interior of) $X$ then $U \cap X \subseteq X$.
- If $p$ is contained in the interior of the complement of $X$ then $U \cap X = \emptyset$.
- If $p$ is contained in the boundary of $X$ then $U \cap X$ and $U \cap \operatorname{int} \operatorname{cpl} X$ are unions of "pieces of pie" as shown in Figure 10.67.

We call a set $X$ trivial if either $X = \emptyset$ or $X = \mathbb{R}^2$. Let $X$ be a non-trivial polygonal set. We call a collection $P_1, \ldots, P_k$ of weakly simple polygons a *representation* of $X$ if:

- the set of segments in the boundary of $X$ is the disjoint union of the set of segments of the $P_i$'s, and
- the orientation of each $P_i$ is such that $X$ is locally to the left of $P_i$, and
- the $P_i$ are pairwise non-crossing, i.e., there are no consecutive segments $pq$ and $qr$ on some $P_i$ and $xq$ and $qy$ on some $P_j$ with $i \neq j$ and the segments interleaving around $q$, see Figure 10.68.

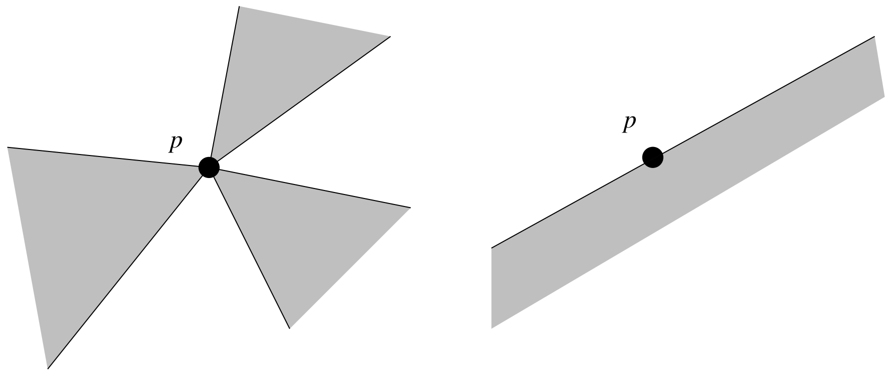


**Figure 10.67** The shaded part of the plane belongs to the polygonal region $X$ and $p$ lies in the boundary of $X$. If $p$ is a vertex of $X$ and $U$ is a sufficiently small neighborhood of $p$ then $U \cap X$ and $U \cap \operatorname{int} \operatorname{cpl} X$ are unions of pieces of pie. If $p$ lies in the relative interior of a boundary segment of $X$ then $X$ looks like an open half-plane in the vicinity of $p$.

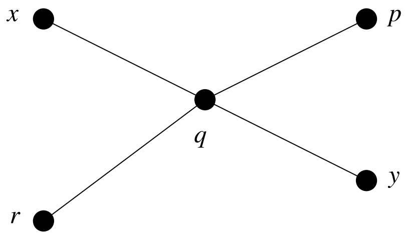


**Figure 10.68** The chains $(\ldots, p, q, r, \ldots)$ and $(\ldots, x, q, y, \ldots)$ cross in $q$.

Figure 10.69 shows an example.

**Lemma 12** *Every non-trivial polygonal set has a representation.*

*Proof* Consider a boundary segment $s$ of $X$. Since $X$ is regular, $X$ lies on only one of the sides of $s$ and hence $s$ can be oriented such that $X$ is locally to the left of $s$.

Consider next a point $p$ as shown in Figure 10.67. Since $X$ is the union of pieces of pie in the neighborhood of $p$ we can join the boundary segments of $X$ incident to $p$ such that any two consecutive segments define one of the pieces of the pie. In this way no crossings are introduced. Also, since none of the pieces of the complement of $X$ is degenerated to a line, every boundary segment incident to $p$ is used only once.

The construction guarantees that the polygons formed are weakly simple and satisfy the two properties of a representation stated above. □

The representation of a polygonal set is not unique as Figure 10.69 shows. We still need to justify the choice of the name representation. In what sense does a representation of a polygonal set "represent" the set?

We start with the observation that the polygons in a representation form a so-called nested family. Let $P_i$ and $P_j$ be two polygons in a representation. Since $P_i$ and $P_j$ do not cross, we have either $\mathrm{bR}\, P_i \cap \mathrm{bR}\, P_j = \emptyset$ or $\mathrm{bR}\, P_i \subset \mathrm{bR}\, P_j$ or $\mathrm{bR}\, P_j \subset \mathrm{bR}\, P_i$, where $\mathrm{bR}\, P$

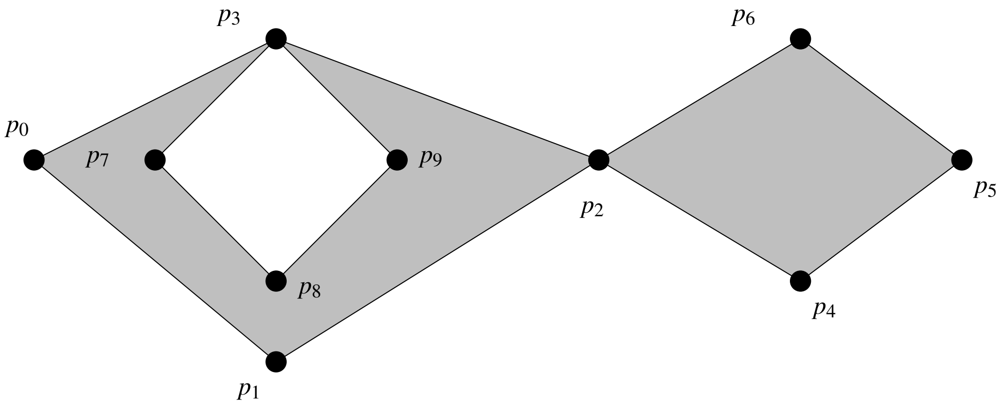


**Figure 10.69** The open shaded region consists of two connected sets, one of which is simple. $X$ can be represented by $(p_0, p_1, p_2, p_4, p_5, p_6, p_2, p_3, p_9, p_8, p_7, p_3)$ or by $(p_0, p_1, p_2, p_3)$, $(p_2, p_4, p_5, p_6)$, $(p_3, p_9, p_8, p_7)$, or by $(p_0, p_1, p_2, p_4, p_5, p_6, p_2, p_3)$, $(p_9, p_8, p_7, p_3)$.

denotes the bounded region with respect to a polygon $P$. We say that $P_j$ is *nested* in $P_i$ if $\mathrm{bR}\, P_j \subset \mathrm{bR}\, P_i$.

We can now define a forest $F$ on the polygons in a representation. A polygon $P_j$ is a child of a polygon $P_i$ if $P_j$ is nested in $P_i$ and there is no $P_k$ such that $P_j$ is nested in $P_k$ and $P_k$ is nested in $P_i$. If $P_j$ is a child of $P_i$ in $F$, we say that $P_j$ is directly nested in $P_i$. We have:

**Lemma 13** *If $P_j$ is a child of $P_i$ in $F$ then $P_j$ and $P_i$ have different orientations. All roots of $F$ have the same orientation.*

*Proof* If $P_i$ is positively oriented then $\mathrm{bR}\, P_i$ belongs to $X$ in the vicinity of $P_i$ and to the left of $P_i$. Since $P_j$ is directly nested in $P_i$ and since it is part of the boundary of $X$, $P_j$ must be negatively oriented. If $P_i$ is negatively oriented then $\mathrm{bR}\, P_i$ belongs to $\mathrm{int\,cpl}\, X$ in the vicinity of $P_i$ and to the left of $P_i$. Since $P_j$ is directly nested in $P_i$ and since it is part of the boundary of $X$, $P_j$ must be positively oriented.

If $X$ is bounded, all roots of $F$ are positively oriented and if $X$ is unbounded, all roots of $P$ are negatively oriented. □

It is convenient to turn the forest $F$ into a tree by adding an artificial root. The polygon associated with the root represents the "circle at infinity". The circle at infinity is positively oriented if $X$ is unbounded and is negatively oriented if $X$ is bounded. We use $P_0$ to denote the artificial polygon representing the circle at infinity. Every point of the plane is contained in the bounded region with respect to the circle at infinity.

We assume from now on that the polygons $P_0, P_1, \ldots, P_k$ in a representation are ordered such that no $P_i$ is nested in a $P_j$ for $i < j$. In other words, parents precede their children.

**Lemma 14** *Let $P_0, P_1, \ldots, P_k$ be a representation of a polygonal set $X$ and let $p$ be a point in the plane that does not lie on any of the polygons in the representation. Let $i$ be maximal* *such that* $p \in \mathrm{bR}\, P_i$. *If* $P_i$ *is positively oriented then* $p \in X$ *and if* $P_i$ *is negatively oriented then* $p \notin X$.

*Proof* Observe first that $i$ exists since every point is contained in the bounded region of the circle at infinity. Assume w.l.o.g. that $P_i$ is positively oriented. Let $P_{j_1}$ to $P_{j_l}$ be the children of $P_i$ in $F$. We have

$$\mathrm{bR}\, P_i \setminus (\mathrm{bR}\, P_{j_1} \cup \ldots \cup \mathrm{bR}\, P_{j_l}) \subseteq X$$

and $i < j_1, \ldots, i < j_l$. Thus $p \notin (\mathrm{bR}\, P_{j_1} \cup \ldots \cup \mathrm{bR}\, P_{j_l})$ by the definition of $i$. This shows that $p \in X$. □

### 10.8.4 *The Implementation of Generalized Polygons*

Generalized polygons are a handle type, i.e., a generalized polygon is realized as a pointer to a representation class (called *gen_polygon_rep* and *rat_gen_polygon_rep*, respectively) which contains the actual representation. The member function *ptr*( ) returns the pointer to the representing object.

The representation consists of a flag $k$ which indicates whether the polygon is trivial and a list *pol_list* of polygons. More precisely, we have a local enumeration type *kind* with elements EMPTY, FULL, and NON_TRIVIAL and $k$ is equal to EMPTY or FULL iff the polygon is empty or full and is equal to NON_TRIVIAL, otherwise. If the polygon is trivial, *pol_list* is empty, and if the polygon is non-trivial, *pol_list* is the list of boundary cycles.

```
enum kind { EMPTY, FULL, NON_TRIVIAL };
kind k;
list<rat_polygon> pol_list;
```

We next discuss some member function of generalized polygons.

**Checking a Representation:** We define a function *check_representation* that applies to a list *pol_list* of polygons. It returns true if *pol_list* is a legal boundary representation, i.e., if:

- the segments of the polygons in *pol_list* meet only at endpoints, i.e, the planar map $G$ defined by them has $2m$ edges, where $m$ is the number of segments, and no parallel edges.
- there are no crossings between polygons,
- if $D$ is directly nested in $C$ then $D$ and $C$ have alternate orientations, and $C$ is before $D$ in the list of polygons, and
- all outermost polygons have the same orientation.

In the following program we check only the first two items. We know of no method to check the other items that is substantially different from our method to compute boundary representations. The latter method will be described in Section 10.8.4.

⟨*gen_polygon: check representation*⟩≡

```
static bool check_rep(const list<POLYGON>& pol_list)
{ GRAPH<POINT,SEGMENT> G;
  list&lt;SEGMENT&gt; seg_list;
  POLYGON P;
  forall(P,pol_list)
  { list&lt;SEGMENT&gt; SL = P.segments();
    seg_list.conc(SL);
  }
  SEGMENT_INTERSECTION(seg_list,G,true);
  if ( G.number_of_edges() != 2*seg_list.length() )
    return False("check_rep: wrong number of edges");
  // no parallel edges
  node v; edge e;
  forall_edges(e,G)
   if ( target(e) == target(G.cyclic_adj_succ(e)) )
     return False("check_rep: parallel edges");
  ⟨check_representation: check for crossings⟩
  return true;
}
bool GEN_POLYGON::check_representation() const
{ if ( trivial() ) return polygons().empty();
  return check_rep(polygons());
}
```

We describe how to check for crossings. Consider any node $v$ of $G$. Each edge $e$ out of $v$ corresponds to a segment $s$ of one of the polygons in *pol_list*. The polygons running through $v$ introduce a pairing on the edges incident to $v$, where two edges are paired if they correspond to consecutive edges of one of the polygons. We number the pairs and replace each edge by the label of its pair. Then it must not happen that we have distinct labels $a$ and $b$ interlacing around $v$, i.e., the cyclic sequence of labels induced by the edges out of $v$ must not contain a subsequence of the form $a, \ldots, b, \ldots, a, \ldots, b$. This is easily checked by means of a push down store $S$. We iterate over the edges $e$ out of $v$. If the edge label of $e$ agrees with the label on the top of $S$, we pop $S$, if it does not agree, we push the label of $e$. There is no crossing at $v$ iff the push down store is empty at the end of the iteration.

⟨*check_representation: check edge labels*⟩≡

```
forall_nodes(v,G)
{ stack<int> S;
  forall_adj_edges(e,v)
  { if ( S.empty() || label[e] != S.top() )
      S.push(label[e]);
    else
```

```
      S.pop();
    }
    if ( !S.empty() ) return False("check_rep: crossing");
  }
```

It remains to compute the edge labels. We do so in a two step process. We first construct a dictionary that stores for every segment $s$ the edge $e(s)$ in $G$ corresponding to it, i.e., having the same source and sink. We then iterate over all pairs $(s,t)$ of consecutive segments and give $e(s)^{rev}$ and $e(t)$ the same label.

⟨*check_representation: check for crossings*⟩≡

```
map&lt;SEGMENT,edge&gt; segment_to_edge;
forall_edges(e,G)
{ SEGMENT s = G[e];
  node v = G.source(e);
  segment_to_edge[s] = ( s.source() == G[v] ? e : G.reversal(e) );
}
edge_array<int> label(G);
int count = 0;
forall(P,pol_list)
{ list_item it;
  const list&lt;SEGMENT&gt;& seg_list = P.segments();
  forall_items(it,seg_list)
  { edge e = segment_to_edge[seg_list[it]];
    e = G.reversal(e);
    edge f = segment_to_edge[seg_list[seg_list.cyclic_succ(it)]];
    label[e] = label[f] = count++;
  }
}
```

⟨*check_representation: check edge labels*⟩

**Point Containment:** The implementation of *side_of* follows directly from Lemma 14. If $P$ is either empty or full, the answer is obvious. If $P$ is non-trivial, we scan through the list of polygons in the representation. If $p$ lies on one of the polygons, we return ON_REGION. Otherwise, we find the last $P_i$ such that $p$ lies in the bounded region of $P_i$; $P_i$ might not exist, i.e., be equal to the fictitious polygon $P_0$. We return the orientation of $P_i$.

⟨*gen_polygon: side_of*⟩≡

```
int GEN_POLYGON::side_of(const POINT& p) const
{ if ( empty() ) return -1;
  if ( full() ) return +1;
  POLYGON P, P_i;
  bool P_i_exists = false;
  forall(P,polygons())
  { region_kind k = P.region_of(p);
    if ( k == ON_REGION ) return 0;
    if ( k == BOUNDED_REGION ) { P_i = P; P_i_exists = true; }
  }
```

```
    if ( P_i_exists ) return P_i.orientation();
    P = (ptr()->pol_list).front();
    return -P.orientation(); // = P0.orientation()
  }
```

**Boolean Operations:** We only discuss the binary boolean operations and leave the implementation of *complement* as an exercise. The implementations of all binary boolean operations follow a common principle. Let $P_0$ and $P_1$ be two generalized polygons and let $R$ be the result of the boolean operation. We construct $R$ in stages:

**(1)** We first deal with the case that either $P_0$ or $P_1$ is trivial. The remaining stages are not needed if this is the case.
**(2)** We construct the planar map $G$ induced by $P_0$ and $P_1$.
**(3)** We classify the face cycles of $G$, i.e., compute for each face its status with respect to $P_1$ and $P_2$.
**(4)** Given the classification of the edges computed in the preceding stage, we mark all edges of $G$ that are relevant for the result $R$ of the boolean operation. An edge is relevant if the face to its left belongs to $R$.
**(5)** We simplify the graph $G$ by deleting edges. We keep only those edges that separate a face belonging to $R$ from a face belonging to the complement of $R$.
**(6)** We trace the face cycles of $G$ and compute the representation of $R$.

Only the first and the fourth stage depend on the boolean operation. All other stages are generic and apply to all boolean operations. In the sequel we concentrate on the *intersection* routine.

We define constants `P0_face`, `non_P0_face`, `P1_face`, and `non_P1_face` which we use to label edges in stages two and three. The constants are chosen such that boolean operations are possible on them. After stages two and three every edge $e$ of $G$ will have a label describing the status of the face to its left with respect to $P_0$ and $P_1$.

The functions defined in ⟨*construct labeled map*⟩ realize stages two and three, the functions defined in ⟨*simplify graph*⟩ realize stage five, and the functions defined in ⟨*collect polygon*⟩ realize stage six. We will discuss them below.

Stage one is easy. If either argument is empty the intersection is empty, and if either argument is full the result is the other argument.

In stage four we label those edges as relevant which border a face of $G$ which belongs to $P_0$ and $P_1$. These are precisely the edges whose label is equal to $P0\_face + P1\_face$.

⟨*gen_polygon: boolean operations*⟩+≡

```
  static int P0_face     = 1;
  static int not_P0_face = 2;
  static int P1_face     = 4;
  static int not_P1_face = 8;
  ⟨construct labeled map⟩
  ⟨simplify graph⟩
```

⟨*collect polygon*⟩
```
GEN_POLYGON GEN_POLYGON::intersection(const GEN_POLYGON& P1) const
{ // stage I
  if ( empty() || P1.empty() )
    return GEN_POLYGON(GEN_POLYGON_REP::EMPTY);
  if ( full() ) return P1;
  if ( P1.full() ) return *this;
  // stages II and III
  ⟨gen boolean operations: set up labeled map⟩
  // label relevant edges, stage IV
  edge_array<bool> relevant(G,false);
  int d = P0_face + P1_face;
  edge e;
  forall_edges(e,G) if (label[e] == d) relevant[e] = true;
  // stages V and VI
  ⟨gen boolean operations: extract result⟩
}
```

We come to stages two and three. We define the graph $G$, we introduce *P0* as a synonym for the *this*-argument of the intersection, we define an edge array *label*, and call *construct_labeled_map*. It computes the planar map defined by the segments of $P_0$ and $P_1$ and labels all edges of this map.

⟨*gen boolean operations: set up labeled map*⟩≡
```
GRAPH<POINT,SEGMENT> G;
const GEN_POLYGON& P0 = *this;
edge_array<int> label;
construct_labeled_map(P0,P1,G,label);
```

The function *construct_labeled_map* realizes stages two and three. It first calls *construct_initial_map* for stage two and then uses *extend_labeling* for stage three. A call of *extend_labeling* with argument $e$ labels the edges of the face cycle of $G$ containing $e$.

⟨*construct labeled map*⟩≡
```
⟨construct initial map⟩
⟨extend labeling⟩
static void construct_labeled_map(const GEN_POLYGON& P0,
                                  const GEN_POLYGON& P1,
                                  GRAPH<POINT,SEGMENT>& G,
                                  edge_array<int>& label)
{ construct_initial_map(P0,P1,G,label);
  edge_array<bool> visited(G,false);
  edge e;
  forall_edges(e,G)
  { if (visited[e]) continue;
```

```
    extend_labeling(P0,P1,G,e,visited,label);
  }
}
```

Stage two is realized by *construct_initial_map*. It takes two generalized polygons $P_0$ and $P_1$ and computes the planar map $G$ induced by their segments using the segment intersection algorithm of Section 10.7. It also computes a label for every dart of $G$. The label of a dart $e = (v, w)$ of $P_0$ is `P0_face` if $P_0$ is locally to the left of $e$ and is `non_P0_face` otherwise. The analogous statement holds true for darts of $P_1$.

We proceed in several steps. In the first step we collect the segments of $P_0$ and $P_1$ into a list *seg_list* and label each segment with the gen_polygon to which it belongs. Note that a segment may belong to $P_0$ and $P_1$. We therefore use the labels 1, 2 and 3, where 3 indicates that a segment belongs to both polygons and label $i$, $1 \le i \le 2$, indicates that the segment belongs to $P_{i-1}$.

In a second step we compute the planar map induced by the segments in *seg_list*. In this planar map every node must have even degree. If the floating point kernel is used the map returned by SEGMENT_INTERSECTION may be non-plane or have a vertex of odd degree; if this is the case we recommend use of the rational kernel.

In the third step we compute the label of each dart. We discuss it below.

⟨*construct initial map*⟩+≡

```
static void construct_initial_map(const GEN_POLYGON& P0,
                                  const GEN_POLYGON& P1,
                                  GRAPH<POINT,SEGMENT>& G,
                                  edge_array<int>& label)
{
  list&lt;SEGMENT&gt; seg_list;
  map&lt;SEGMENT,int&gt; seg_label(0);
  const list&lt;SEGMENT&gt;& L0 = P0.edges();
  const list&lt;SEGMENT&gt;& L1 = P1.edges();
  SEGMENT s;
  forall(s,L0) { seg_label[s] = 1;
                 seg_list.append(s);
               }
  forall(s,L1) { seg_label[s] += 2;
                 seg_list.append(s);
               }
  SEGMENT_INTERSECTION(seg_list,G,true);
  node v;
#if ( KERNEL == FLOAT_KERNEL )
  if ( Genus(G) != 0 ) error_handler(1,mes + "Genus(G) != 0.");
  forall_nodes(v,G)
  { int deg = G.outdeg(v);
    if (deg % 2 != 0) error_handler(1,mes + "odd degree vertex.");
  }
```

```
#endif
  ⟨construct_initial_map: compute dart labels⟩
}
```

It remains to compute the dart labels.

Consider a dart $e$ and its reversal. We assign a polygon to $e$ as follows. If the segment $s = G[e]$ belongs to a unique polygon, $e$ inherits the polygon from $G[e]$. Otherwise, either the cyclic adjacency predecessor or the cyclic adjacency successor of $e$ must be parallel to $e$, i.e., have the same target as $e$. We arbitrarily assign $e$ to $P_0$ in the former case and to $P_1$ in the latter case.

The polygon $P_i$ is locally to the left of $e$ if $s$ and $e$ point into the same direction, i.e., if the dot product of the underlying vectors is positive.

⟨*construct_initial_map: compute dart labels*⟩≡

```
label.init(G,0);
edge e0;
forall_edges(e0,G)
{ if ( label[e0] != 0 ) continue;
  edge e = e0; edge e_rev = G.reversal(e);
  POINT   a = G[source(e)];
  POINT   b = G[target(e)];
  SEGMENT s = G[e];
  if ( (b - a) * (s.target() - s.source()) <= 0 )
    leda_swap(e,e_rev);
  // now s and e point into the same direction
  switch ( seg_label[s] )
  { case 1: label[e] = P0_face;
            label[e_rev] = not_P0_face;
            break;
    case 2: label[e] = P1_face;
            label[e_rev] = not_P1_face;
            break;
    case 3: { edge f = G.cyclic_adj_pred(e);
              if ( target(f) != target(e) ) f = G.cyclic_adj_succ(e);
              label[e] = P0_face;
              label[e_rev] = not_P0_face;
              label[f] = P1_face;
              label[G.reversal(f)] = not_P1_face;
            }
  }
}
```

The function *extend_labeling* classifies the face $F$ to the left of dart $e$. It scans the face cycle containing $e$, marks all darts of the cycle as visited, and computes the "or" of all dart labels on the cycle in $d$.

If all darts of the face cycle originate from either $P_0$ ($d$ is less than four) or $P_1$ ($d$ is divisible by four), we still have to classify the face cycle with respect to the other polygon and update $d$ accordingly. This will be discussed below.

Finally, the label $d$ is propagated to all darts of the cycle. If the label is contradictory, i.e., claims that the face is a $P_i$-face and a not-$P_i$-face, we raise an error.

⟨*extend labeling*⟩≡

```
static void extend_labeling(const GEN_POLYGON& P0,const GEN_POLYGON& P1,
                            const GRAPH<POINT,SEGMENT>& G, edge e,
                            edge_array<bool>& visited,
                            edge_array<int>& label)
{ int d = 0; int length = 0;
  edge x = e;
  do { visited[x] = true;   length++;
       //node v = source(x);
       //if (G.outdeg(v) == 2) v2 = v;
       d |= label[x];
       x = G.face_cycle_succ(x);
  } while (x != e);
  if ( d % 4 == 0 || d < 4 )
    { ⟨extend_labeling: face cycle has only darts from one polygon⟩ }
  x = e;
#if ( KERNEL == FLOAT_KERNEL )
  if ( d % 4 == P0_face + not_P0_face ||
      (d/4)*4 == P1_face + not_P1_face )
    error_handler(1,mes + "contradicting edge labels.");
#endif
  do { label[x] = d;
       x = G.face_cycle_succ(x);
  } while (x != e);
}
```

It remains to deal with the case that all darts of the face cycle $F$ belong to the same *gen_polygon*, say $P_i$. Let $v$ be the source of $e$. We distinguish two cases: either no dart out of $v$ has a determined status with respect to $P_{1-i}$ or this is not the case. In the former case $v$ cannot lie on the boundary of $P_{1-i}$ and hence $v$'s side with respect to $P_{i-1}$ determines the status of $F$ with respect to $P_{i-1}$. In the latter case let $f$ be the nearest adjacency predecessor of $e$ such that the status of $f$ with respect to $P_{1-i}$ is already known. For all darts between $e$ and $f$ the status is still unknown and hence none of them can be contained in the boundary of $P_{1-i}$; $f$ may be contained in the boundary of $P_{1-i}$ or not (in the latter case, $f$ belongs to a face cycle which was already considered and hence its status with respect to both polygons is known). In either case the status of $F$ with respect to $P_{1-i}$ is given by the status of $f$ with respect to $P_{1-i}$, see Figure 10.70.

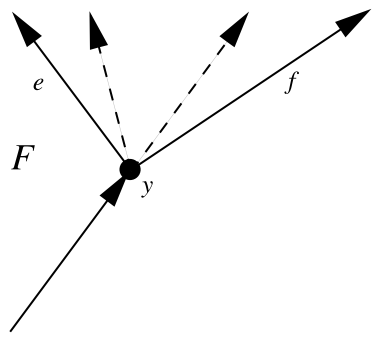


**Figure 10.70** The dart $f$ is the nearest adjacency predecessor of $e$ whose status with respect to $P_{1-i}$ is known. The edges between $e$ and $f$ do not belong to the boundary of $P_{1-i}$ and hence $F$ and the face to the left of $f$ have the same status with respect to $P_{1-i}$.

⟨*extend_labeling: face cycle has only darts from one polygon*⟩≡

```
edge f;
for ( f = G.cyclic_adj_pred(e); f != e; f = G.cyclic_adj_pred(f) )
{ if ( d % 4 == 0 && label[f] % 4 != 0 || d < 4  && label[f] > 4 )
    break;
}
if ( f == e )
{ node v = source(e);
  if ( d % 4 == 0 )
      d |= ( P0.side_of(G[v]) == 1 ? P0_face : not_P0_face );
  if ( d < 4 )
      d |= ( P1.side_of(G[v]) == 1 ? P1_face : not_P1_face );
}
else
{ if ( d % 4 == 0 ) d |= ( label[f] % 4 );
  if ( d < 4 )      d |= ( ( label[f] / 4 ) * 4 );
}
```

We come to stage five. At this point all darts of $G$ are labeled as relevant or non-relevant. A dart is labeled relevant if the face to its left belongs to the result $R$ of the boolean operation.

We simplify the graph by removing darts. We proceed in two steps. In the first step we remove parallel darts that come from overlapping segments in the two arguments of the boolean operation, see Figure 10.71. This turns all face cycles of $G$ into weakly simple polygons. In the second step we remove all edges from the graph that do not separate $R$ from its complement.

The details of the first step are as follows. Let $e$ and $f$ be two parallel darts and assume that $f$ is the cyclic adjacency successor of $e$. This implies that we have a face cycle $(e, f^{rev})$ of length two. This face cycle defines a polygon of area zero which we can remove. We remove the face cycle by removing its two constituent darts and making $f$ and $e^{rev}$ reversals of each other. There cannot be a set of three parallel darts and hence the target of $f$ should be different from the target of its cyclic adjacency successor.

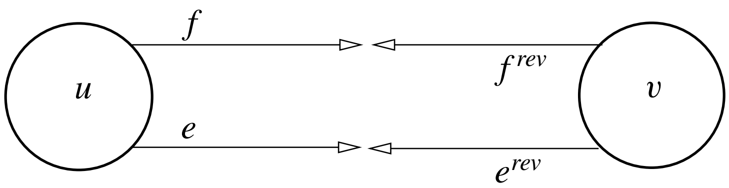


**Figure 10.71** The darts $e$ and $f$ come from a segment of $P_0$ and $P_1$, respectively. The face cycle $(e, f^{rev})$ consists of only two darts. We remove $e$ and $f^{rev}$ and make $f$ and $e^{rev}$ reversals of each other.

The first simplification step leaves us with a planar map without parallel darts. This implies that all face cycles are weakly simple polygons. In the second step we merge adjacent faces that belong to the same side of the result polygon.

A dart $e$ does not separate $R$ from its complement if $e$ and $e^{rev}$ are either both relevant or both irrelevant. In the former case $R$ exists on both sides of the edge and in the latter case the complement of $R$ lives on both sides of the edge.

The second step may remove all edges from the graph. This will be the case if the result is either empty or full. We need to distinguish these cases. We have the former case if there are no relevant edges before simplification and we have the latter case if all edges are relevant before simplification. We return true in the latter case.

⟨*simplify graph*⟩≡
```
static bool simplify_graph(GRAPH<POINT,SEGMENT>& G,
                           edge_array<bool>& relevant)
{ edge e; node v;
  forall_nodes(v,G)
  { list<edge> E = G.out_edges(v);
    forall(e,E)
    { edge f = G.cyclic_adj_succ(e);
      if ( target(e) != target(f) ) continue;
      edge e_rev = G.reversal(e);
      G.del_edge(e); G.del_edge(G.reversal(f));
      G.set_reversal(e_rev,f);
    }
  }
  bool non_trivial_result = false;
  forall_nodes(v,G)
  { list<edge> E = G.out_edges(v);
    forall(e,E)
    { if ( relevant[e] || relevant[G.reversal(e)] )
         non_trivial_result = true;
      if ( relevant[e] == relevant[G.reversal(e)] )
      { G.del_edge(G.reversal(e)); G.del_edge(e); }
    }
  }
  return non_trivial_result;
}
```

After simplification every uedge of $G$ separates $R$ from its complement and hence belongs to the boundary representation. Also all face cycles are weakly simple polygons. We conclude that the face cycles of $G$ form the representation of the result of the boolean operation.

The following function *collect_polygon* takes a dart $e$, marks all darts in the face cycle of $e$ as visited, and collects the segments corresponding to the face cycle in a list *pol*.

⟨*collect polygon*⟩≡
```
static void collect_polygon(const GRAPH<POINT,SEGMENT>& G, edge e,
                            edge_array<bool>& visited,
                            list&lt;SEGMENT&gt;& pol)
{ pol.clear();
  edge x = e;
  do { visited[x] = true;
       node v = source(x);
       node w = target(x);
       POINT   a = G[v];
       POINT   b = G[w];
       pol.append(SEGMENT(a,b));
       x = G.face_cycle_succ(x);
  } while (x != e);
}
```

The function above is the main ingredient for the last stage. We first simplify $G$. If this trivializes $G$, i.e., removes all edges from it, we either return the full gen_polygon or the empty gen_polygon; the return value of *simplify_graph* tells us which.

⟨*gen boolean operations: extract result*⟩≡
```
bool non_trivial_result = simplify_graph(G,relevant);
if (G.number_of_edges() == 0 )
{ if ( non_trivial_result )
    return GEN_POLYGON(GEN_POLYGON_REP::FULL);
  else
    return GEN_POLYGON(GEN_POLYGON_REP::EMPTY);
}
edge_array<bool> visited(G,false);
list<POLYGON> result;
⟨gen boolean operations: form boundary cycles⟩
return GEN_POLYGON(result,GEN_POLYGON::NO_CHECK);
```

So assume that $G$ is non-trivial. We cycle over all darts of $G$ and collect all face cycles consisting of relevant darts.

⟨*gen boolean operations: form boundary cycles, first try*⟩≡
```
forall_edges(e,G)
{ if ( visited[e] || !relevant[e] ) continue;
  list&lt;SEGMENT&gt; pol;
```

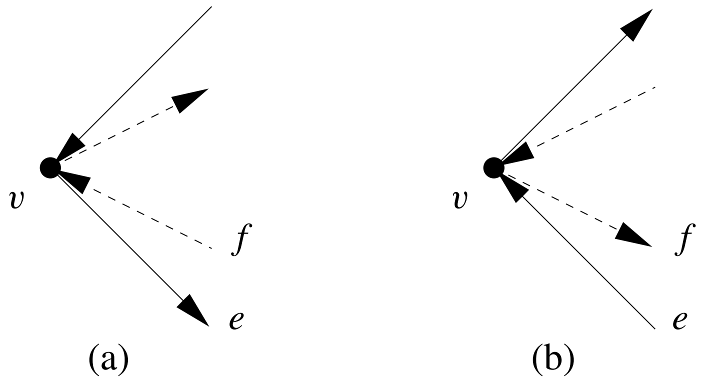


**Figure 10.72** The dashed boundary cycle is nested in the solid cycle and both cycles have $v$ as their leading node. In situation (a) the leading dart of the solid cycle is $e$ and the leading dart of the dashed cycle is $f^{rev}$. In situation (b) the leading dart of the solid cycle is $e^{rev}$ and the leading dart of the dashed cycle is $f$. In either case the leading dart of the solid cycle has smaller slope.

```
    collect_polygon(G,e,visited,pol);
    POLYGON P(pol);
    result.append(P);
  }
```

The code above generates the boundary cycles in no particular order. We want an order that reflects nesting, i.e., no polygon should be nested in a polygon following it.

There are several ways to achieve a proper ordering. Our first solution took time $O(n + k \log k)$ and, moreover, was burdened with a fairly large constant factor. We exploited the fact that if $D$ is nested in $C$ then $D$ has smaller unsigned area than $C$. We generated the polygons in an arbitrary order and then sorted the polygons in decreasing order of their unsigned area.

We describe an alternative approach. We show that one can rearrange the darts of $G$ such that the code above generates the polygons in the proper order. Our approach is based on the following definition and observation. Define the leading node and dart of a boundary cycle as follows:

- The leading node $v(C)$ of a boundary cycle $C$ is the lexicographically smallest node of the boundary cycle.
- The leading dart $e(C)$ of a boundary cycle is the shallowest (= smallest slope) dart of $C$ starting in $v(C)$ if $C$ is positively oriented, and is the reversal of the shallowest dart in $C$ ending in $v(C)$ if $C$ is negatively oriented.

**Lemma 15** *If $D$ is nested in $C$ then either:*

- *$v(C)$ is lexicographically smaller than $v(D)$ or*
- *$v(C)$ is equal to $v(D)$ and $e(C)$ has smaller slope than $e(D)$.*

*Proof* Clearly the leading node of $C$ cannot be lexicographically larger than the leading node of $D$. If $C$ and $D$ have the same leading node, the situation is as shown in Figure 10.72 and the leading dart of $C$ has smaller slope than the leading dart of $D$. □

Consider the following order on darts. A dart $e = (v, w)$ precedes a dart $f = (x, y)$ if either $v$ lexicographically precedes $x$ or $v$ is equal to $x$ and $e$ has smaller slope than $f$. This order has the following properties:

- For any boundary cycle $C$ the leading dart of $C$ precedes all darts of $C$.
- If $D$ is nested in $C$ then the leading dart of $C$ precedes the leading dart of $D$.

The following compare class realizes the dart ordering; the base class *leda_cmp_base* is discussed in Section 2.10.

⟨*collect polygon*⟩+≡

```
template <class POINT, class SEGMENT>
class cmp_for_cycle_tracing : public leda_cmp_base<edge> {
const GRAPH<POINT,SEGMENT>& G;
public:
  cmp_for_cycle_tracing(const GRAPH<POINT,SEGMENT>& g): G(g) {}
  int operator()(const edge& e1, const edge& e2) const
  { node v = G.source(e1);
    node w = G.source(e2);
    if ( v != w ) return compare(G[v],G[w]);
    SEGMENT s1 = G[e1];
    SEGMENT s2 = G[e2];
    return cmp_slopes(s1,s2);
  }
};
```

It is now easy to generate the boundary cycles in the appropriate order. We sort the darts of $G$ according to the ordering above and then iterate over all darts of $G$. Whenever we encounter a uedge that is not contained in a boundary cycle yet, we collect the boundary cycle. The uedge is a pair $\{e, e^{rev}\}$ and either $e$ or its reversal is relevant (but not both). If $e$ is relevant, the cycle to be traced is positively oriented, and if $e^{rev}$ is relevant, the cycle to be traced is negatively oriented, see Figure 10.72. Thus there is no need to compute the orientations of the boundary cycles; our method of generating boundary cycles in an ordered fashion yields the orientations as a by-product.

We obtain:

⟨*gen boolean operations: form boundary cycles*⟩≡

```
cmp_for_cycle_tracing<POINT,SEGMENT> cmp(G);
list<edge> E = G.all_edges();
E.sort(cmp);
edge e0;
forall(e0,E)
```

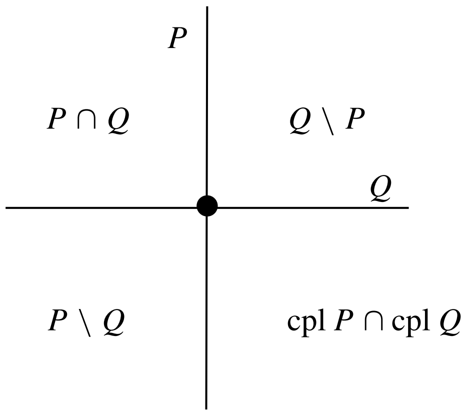


**Figure 10.73** The vertex $v$ is an intersection between the boundaries of $P$ and $Q$. There are four faces incident to $v$ and at least one but not all of them belong to the result of the boolean operation.

```
{ edge e = e0;
  if ( visited[e] || visited[G.reversal(e)]) continue;
  int orient;
  if ( relevant[e] )
    { orient = +1; }
  else
    { e = G.reversal(e); orient = -1; }
  list&lt;SEGMENT&gt; pol;
  collect_polygon(G,e,visited,pol);
  POLYGON P(pol,orient);
  result.append(P);
}
```

We conclude our treatment of boolean operations on polygons with a discussion of their asymptotic running time. Consider a boolean operation with input polygons $P$ and $Q$ and result polygon $R$. Let $n$ be the total number of vertices of $P$, $Q$, and $R$, and let $G$ be the graph induced by the two input polygons. Any vertex of $G$ is either a vertex of one of the input polygons or is an intersection between the boundaries of the input polygons. In the latter case it will be a vertex of the result polygon, as Figure 10.73 shows. We conclude that $G$ has at most $n$ vertices and hence can be computed in time $O(n \log n)$. The time required to sort the edges before tracing the boundary cycles is also $O(n \log n)$. Let $f$ be the number of face cycles of $G$ which have darts from only one of the polygons; $f$ can be as large as $O(n)$. For each such face cycle we spend time $O(n)$ to classify it with respect to the other polygon for a total time of $O(fn)$ (this time bound could be reduced to $O(f \log n)$ by using a more refined data structure for point location). All other steps take time $O(n)$. We conclude that the total time to compute boolean operations is $O(n + n \log n + fn)$.

**A Demo Program:** We give a small demo program. We construct an n-gon $P$ with vertices near the unit circle. We also construct an affine transformation $T$ that rotates the plane by

| $n$ | $m$ | $P$ | $T$ | $Q$ | $P \cap Q$ | $\lvert P \cap Q \rvert$ |
|---|---|---|---|---|---|---|
| 5000 | 6.175e+06 | 1.35 | 0 | 0.36 | 12.92 | 20000 |
| 5000 | 2.47e+07 | 1.33 | 0 | 0.37 | 13.06 | 20000 |
| 5000 | 9.88e+07 | 1.35 | 0.01 | 0.39 | 13.44 | 20000 |
| 5000 | 3.952e+08 | 1.35 | 0 | 0.35 | 13.71 | 20000 |
| 5000 | 1.581e+09 | 1.35 | 0 | 0.36 | – | – |
| 20000 | 2.47e+07 | 5.65 | 0 | 1.47 | 56.13 | 80000 |
| 20000 | 9.88e+07 | 5.71 | 0 | 1.61 | – | – |

**Table 10.10** Execution times with floating point kernel: The first two columns show $n$ and $m$, respectively, the next four columns show the time to construct $P$, $T$, $Q = T(P)$, and $P \cap Q$, respectively, and the last column shows the number of vertices of $P \cap Q$. A dash in the next to last column indicates that the program produced an error message and recommended use of rat_polygons.

an angle $\alpha = 2\pi/(2nm) \pm eps$ about the origin, where $eps = 1/(10nm)$. Let $Q = T(P)$ be the result of turning $P$ by angle $\alpha$ and let $R$ be the union of $P$ and $Q$.

$\langle n\_gon\_time \rangle \equiv$

```
double eps = 1/(10.0*n*m);
POLYGON P = N_GON(n,C,eps);
GEN_POLYGON PG(P,GEN_POLYGON::NO_CHECK);
report_time("time to generate P = ");
TRANSFORM T = rotation(ORIGIN, LEDA_PI/(n * m), eps);
report_time("time to generate the transformation T = ");
POLYGON Q = T(P);
GEN_POLYGON QG(Q,GEN_POLYGON::NO_CHECK);
report_time("time to compute T(P) = ");
GEN_POLYGON R = PG.unite(QG);
report_time("time to compute P union T(P) = ");
```

Tables 10.10 and 10.11 show the execution times for the floating point and the rational kernel and different values of $n$ and $m$. Observe that we ran extreme examples. We took 5000-gons and 20000-gons and rotated them by angles $2\pi/(2 * n * m)$, where $m$ ranges between $10^6$ and $10^9$. This amounts to rotations by angles between $10^{-8}$ and $10^{-10}$ degrees.

The floating point kernel did not always obtain a result. In the two cases where it did not obtain a result, it discovered that there is a problem. For $n = 5000$ and $n = 1.581 \cdot 10^9$ it reported that the map computed by SEGMENT_INTERSECTION is not planar and for $n = 20000$ and $m = 9.88 \cdot 10^7$ it reported that there is a node of odd degree in the map.

| $n$ | $m$ | $P$ | $T$ | $Q$ | $P \cap Q$ | $\|P \cap Q\|$ |
|---|---|---|---|---|---|---|
| 5000 | 6.175e+06 | 1.69 | 0 | 1.4 | 30.8 | 20000 |
| 5000 | 2.47e+07 | 1.73 | 0 | 1.41 | 31.45 | 20000 |
| 5000 | 9.88e+07 | 1.74 | 0.01 | 1.4 | 33.93 | 20000 |
| 5000 | 3.952e+08 | 1.77 | 0 | 1.41 | 34.01 | 20000 |
| 5000 | 1.581e+09 | 1.78 | 0.009995 | 1.41 | 34.7 | 20000 |
| 20000 | 2.47e+07 | 7.25 | 0 | 5.66 | 140.9 | 80000 |
| 20000 | 9.88e+07 | 7.37 | 0 | 5.69 | 141.6 | 80000 |
| 20000 | 3.952e+08 | 7.45 | 0 | 5.66 | 143.2 | 80000 |
| 20000 | 1.581e+09 | 7.52 | 0.01001 | 5.58 | 145.1 | 80000 |
| 20000 | 6.323e+09 | 7.53 | 0 | 5.6 | 149.2 | 80000 |

**Table 10.11** Execution times with rational kernel: The meaning of the columns is the same as for Table 10.10.

It is instructive to study the output of the program when the test for the planarity of $G$ is not made. The graph $G$ constructed by SEGMENT_INTERSECTION had 19994 nodes (and so 6 nodes are missing) and 59952 edges, 10012 nodes had degree two (12 too many) and 9982 nodes had degree four (18 too few). The genus of $G$ was one. $G$ had face cycles of length two and three and *only one* face cycle of length larger than three (there should be two). *All* edges of the graph were declared relevant and hence removed by *simplify_graph*. The full polygon was returned. It took several hours of detective work to discover this explanation for the behavior of the floating point implementation. The detective work was considerably helped by the fact that the execution with the rational kernel produced the correct result and hence we *knew* that the error must be in the floating point arithmetic.

It would be fantastic if the floating point implementation would always degrade gracefully, i.e., either compute the correct result or tell that the problem is too difficult for a floating point computation. We are not making this claim.

Although the floating point implementation did not always obtain the correct result it can handle surprisingly difficult cases.

The rational kernel always worked correctly, as it is supposed to do. There is about a factor three overhead for the use of the rational kernel.

***Exercises for 10.8***

1 Implement the function *complement* for generalized polygons.

2 Implement the function *unite* for generalized polygons. Start from the implementation of *intersection* and describe the required modifications.

## 10.9 A Glimpse at Higher-Dimensional Geometric Algorithms

We give an overview of the extension package for higher-dimensional computational geometry, exhibit a relationship between convex hulls and Delaunay triangulations, and use it to derive the formula for the side-of-sphere test. For a detailed treatment of higher-dimensional geometry we refer the reader to [Ede87].

### 10.9.1 *The Extension Package for Higher-Dimensional Geometry*

The extension package [MMN$^+$98] features a higher-dimensional kernel, simplicial complexes, convex hulls and Delaunay diagrams.

The *higher-dimensional kernel* offers points, lines, segments, rays, vectors, hyperplanes, spheres, affine transformations, and geometric operations and predicates in $d$-dimensional Euclidian space for arbitrary dimension $d$. Examples for geometric predicates are the orientation test, the side-of-sphere test, the test of whether a point is contained in a simplex, and the computation of the affine rank of a set of points. Examples for geometric constructions are the construction of a hyperplane from a set of points, or the computation of the intersection of a line and a hyperplane.

The extension package offers three geometric data structures: regular simplicial complexes, convex hulls and Delaunay diagrams.

A *simplicial complex* is a collection of simplices in which the intersection of any two simplices in the collection is a face of both[28]. A simplicial complex is *regular* iff all maximal simplices of the collection[29] have the same dimension and if its maximal simplices are connected under the neighboring relation[30]. The data type *regl_complex* realizes regular simplicial complexes. It supports navigation in the complex (go to the $i$-th neighbor) and update operations on the complex (add a new simplex and make it the neighbor of some existing simplices). Regular simplicial complexes generalize triangulations to arbitrary dimension.

*Convex hulls* are represented as regular simplicial complexes, namely by a complex arising from a triangulation of the hull. Figure 10.11 shows an example in two-dimensional space.

The convex hull complex is built by a natural generalization of the incremental hull algorithm of Section 10.1.2. Whenever a point $p$ is added to a convex hull, a simplex with peak $p$ is added to the convex hull for every facet of the hull visible from $p$.

The data type *convex_hull* supports navigation through the underlying triangulation, navigation over the boundary of the hull, visibility queries (find all facets visible from a point $p$), point location queries (does a point $p$ lie in the interior, on the boundary, or in the exterior of the hull) and insertion of new points.

*Delaunay triangulations* are also represented as simplicial complexes. The data type

[28] The empty set is a face of any simplex.

[29] A simplex is maximal if it is not contained in any other simplex.

[30] Two simplices of dimension $k$ are neighbors if they share a face of dimension $K - 1$.

*delaunay* extends the functionality of the type *point_set* of Section 10.6 to higher dimensions. It supports navigation in the complex, insertion of new points, point location queries (return the simplex containing a query point $p$), nearest neighbor queries (return the point closest to a query point $p$), and range searches with spheres and simplices (return all points contained in a query sphere or query simplex, respectively).

### 10.9.2 *Delaunay Diagrams and Convex Hulls*

The implementation of Delaunay diagrams in higher-dimensional space is based on a powerful relationship between Delaunay diagrams, Voronoi diagrams, and convex hulls in one higher dimension.

Let $d$ be a positive integer. We use $x_0, x_1, \ldots, x_{d-1}$, and $z$ for the Cartesian coordinates of a $d+1$-dimensional space. Our Delaunay triangulations live in the $d$-dimensional subspace with coordinates $x_0, x_1, \ldots, x_{d-1}$ and the corresponding convex hulls will live in the $d+1$-dimensional space with coordinates $x_0, x_1, \ldots, x_{d-1}$, and $z$. We call the former space the *base space*.

The *paraboloid of revolution* $P$ is defined by

$$z = x_0^2 + x_1^2 + \ldots + x_{d-1}^2.$$

It is obtained by rotating the two-dimensional parabola $z = x_0^2$ about the $z$-axis. The key for the entire section is the following observation.

**Lemma 16** *The intersection between $P$ and any hyperplane $h$ that is not parallel to the $z$-axis is a curve $C$ whose projection into the base space is a sphere and any sphere in the base space can be obtained in that way.*

*Proof* Since $h$ is not parallel to the $z$-axis it is defined by an equation

$$z = a_0 x_0 + a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d.$$

Any point $(x_0, x_1, \ldots, x_{d-1}, z)$ in the intersection between $P$ and $h$ satisfies

$$x_0^2 + x_1^2 + \ldots + x_{d-1}^2 = z = a_0 x_0 + a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$$

and hence

$$(x_0 - a_0/2)^2 + \ldots + (x_{d-1} - a_{d-1}/2)^2 = a_d + (a_0^2 + \ldots + a_{d-1}^2)/4.$$

This is the equation of a sphere in base space with center $c$ and radius $r$ where

$$c = (a_0/2, \ldots a_{d-1}/2) \text{ and } r = \sqrt{a_d + (a_0^2 + \ldots + a_{d-1}^2)/4}.$$

Thus the projection of $P \cap h$ into base space is a sphere. Conversely, if we start with any sphere $B$ with center $c$ and radius $r$ in base space and define coefficients $a_0, a_1, \ldots, a_d$ through $c = (a_0/2, \ldots a_{d-1}/2)$ and $r^2 = a_d + (a_0^2 + \ldots + a_{d-1}^2)/4$ then the hyperplane $z = a_0 x_0 + a_1 x_1 + \ldots + a_{d-1} x_{d-1} + a_d$ will intersect $P$ in a curve projecting into $B$. $\square$

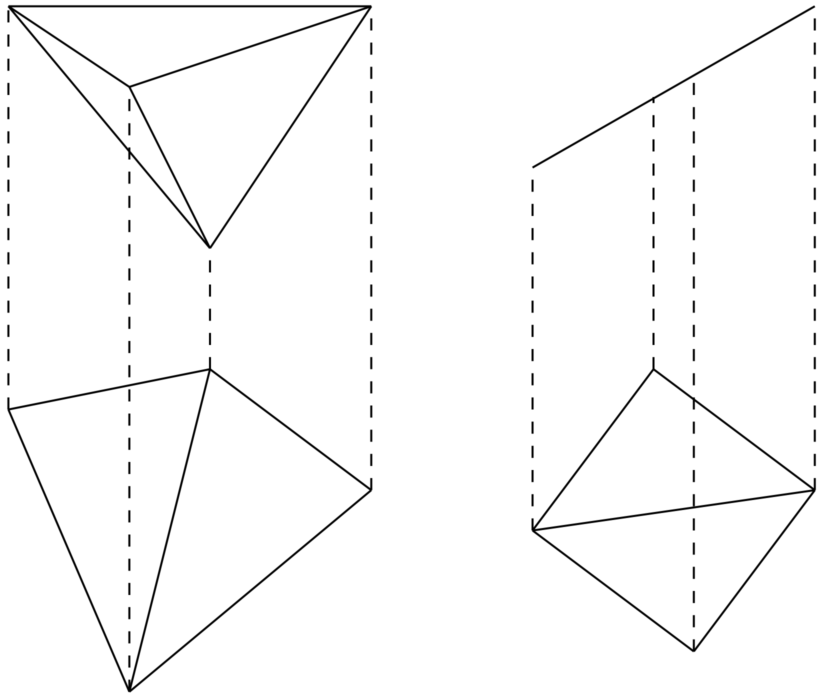

**Figure 10.74** The connection between Delaunay diagrams in the plane and convex hulls in three-space. The lifting map is indicated by dashed lines. The four points on the left are not co-circular and hence the convex hull of the lifted points is a tetrahedron. The Delaunay diagram is the projection of the lower part of the tetrahedron.
The four points on the right are co-circular and hence the lifted points lie in a common plane. The convex hull of the lifted points is a rectangle contained in this plane. The Delaunay diagram is the projection of the rectangle and the projection of any triangulation of the rectangle is a Delaunay triangulation.

For a point $p = (x_0, x_1, \ldots, x_{d-1})$ in base space we call

$$\mathit{lift}(p) = (x_0, x_1, \ldots, x_{d-1}, x_0^2 + x_1^2 + \ldots + x_{d-1}^2)$$

its *lifting* onto $P$, i.e., the intersection of $P$ with a vertical upward ray starting in $p$. We use the lifting map to establish a surprising connection between Delaunay diagrams and convex hulls.

Let $S$ be any full-dimensional finite set of points in base space and let $p_0, p_1, \ldots, p_d$ be $d+1$ affinely independent points in $S$. The lifted points $\mathit{lift}(p_0), \mathit{lift}(p_1), \ldots, \mathit{lift}(p_d)$ define a hyperplane $h$. By the above, this hyperplane intersects $P$ in a curve $C$ whose projection into the base space is a sphere $B$. Of course, $B$ passes through $p_0, p_1, \ldots, p_d$. In other words, $B$ is the circumsphere of the simplex spanned by $p_0, p_1, \ldots, p_d$.

Next consider an arbitrary additional point $p$ in base space. If $p$ lies inside $B$ then $\mathit{lift}(p)$ lies below $h$, if $p$ lies on $B$ then $\mathit{lift}(p)$ lies on $h$, and if $p$ lies outside $B$ then $\mathit{lift}(p)$ lies above $h$. We conclude that the interior of the circumsphere of $p_0, p_1, \ldots, p_d$ is void of points of $S$ if and only if no point of

$$\mathit{lift}(S) = \{\mathit{lift}(p) \mid p \in S\}$$

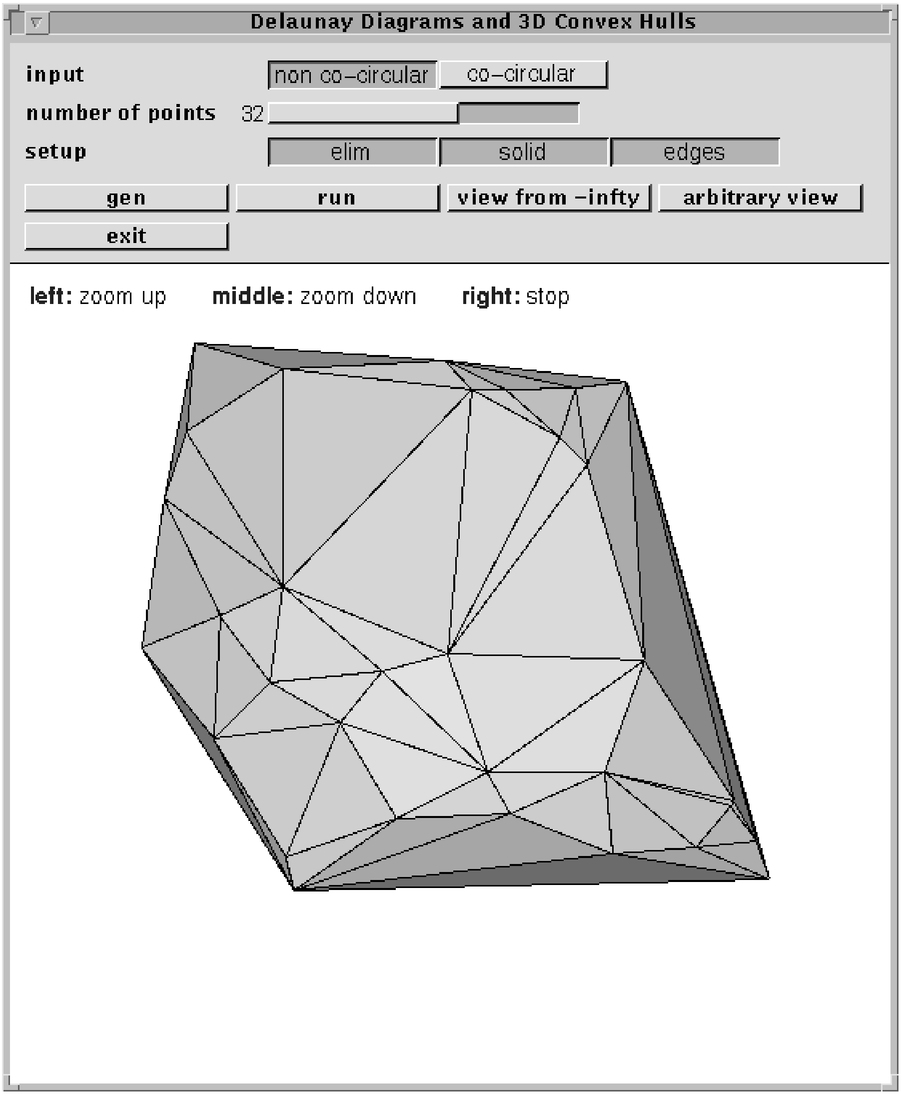


**Figure 10.75** A screen shot of the delaunay_and_convex_hull_demo (in demo/book/Geo). The screen shot shows the lower convex hull of 32 random points in the unit square lifted to the paraboloid of revolution.

lies below $h$, or in other words, if $h$ supports the lower convex hull of *lift*($S$). The *lower convex hull* of a point set consists of all points of the convex hull which are visible from $z = -\infty$.

Let us take a closer look at the lower convex hull. We need to distinguish cases according to whether the points in $S$ are co-spherical or not, see Figures 10.74 and 10.75.

If the points in $S$ are not co-spherical, the dimension of *list*($S$) is one higher than the dimension of $S$ and hence *list*($S$) is full-dimensional. The convex hull of *lift*($S$) is a $d+1$-dimensional object. The lower convex hull consists of all facets with a downward normal.

If the points in $S$ are co-spherical, the points in *lift*($S$) lie in a common hyperplane and the dimension of *lift*($S$) is the same as the dimension of $S$. The Delaunay diagram of $S$ is identical to the convex hull of $S$ and any triangulation of the convex hull is a Delaunay triangulation. The convex hull of $lift(S)$ is a $d$-dimensional object; it is simply the lifting of the convex hull of $S$ to a plane in $d+1$-dimensional space.

We summarize.

**Theorem 3** *For any finite point set $S$ in base space the Delaunay diagram $DD(S)$ is the vertical projection of the lower convex hull of $lift(S)$ into base space*[31]. *A Delaunay triangulation is the vertical projection of a triangulation of the lower hull.*

The preceding theorem is the basis for the implementation of Delaunay diagrams. We maintain the convex hull of the lifted points. All queries about Delaunay diagrams are translated into queries about the corresponding hull.

### 10.9.3 *Sidedness and Orientation*

In this section we show how the results of the preceding section can be used to define the orientation, side-of, and region-of predicate for spheres.

Let $p_0, p_1, \ldots, p_d$ be $d+1$ points in base space and let $p$ be an additional point in base space and let $S$ be the sphere passing through $p_0, p_1, \ldots, p_d$. Define *orientation*$(S)$, *side_of_sphere*$(S, p)$, and *region_of_sphere*$(S, p)$ by

$$\begin{aligned}
orientation(S) &= orientation(p_0, p_1, \ldots, p_d),\\[1em]
side_of_sphere(S, p) &= -orientation(lift(p_0), lift(p_1), \ldots, lift(p_d), lift(p)),\\[1em]
region_of_sphere(S, p) &= \begin{cases} bounded_region & \text{if } o(S)\cdot o(S,p) > 0\\ on_region & \text{if } o(S)\cdot o(S,p) = 0\\ unbounded_region & \text{if } o(S)\cdot o(S,p) < 0 \end{cases}
\end{aligned}$$

where we used $o$ as an abbreviation for *orientation* in the last formula to save space.

We will next show that *side_of_sphere*$(S, p)$ and *region_of_sphere*$(S, p)$ have their intended meaning.

**Lemma 17** *Let $p_0, p_1, \ldots, p_d$ be $d+1$ affinely independent points in base space and let $p$ be an additional point in base space. Then we have*

$$side\_of\_sphere(S, p) = \begin{cases} +1 & \textit{if } p \textit{ lies inside } S\\ 0 & \textit{if } p \textit{ lies on } S\\ -1 & \textit{if } p \textit{ lies outside } S \end{cases}$$

*if orientation$(S) > 0$ and*

$$side\_of\_sphere(S, p) = \begin{cases} +1 & \textit{if } p \textit{ lies outside } S\\ 0 & \textit{if } p \textit{ lies on } S\\ -1 & \textit{if } p \textit{ lies inside } S \end{cases}$$

[31] In the discussion above we assumed that $S$ is full-dimensional. If $S$ is contained in a lower dimensional subspace, we only need to restrict the discussion to this subspace. More precisely, assume that $S$ is contained in a $k$-dimensional subspace. We may assume w.l.o.g that the first $k$ coordinates span this subspace and can then use the argument above with $d$ replaced by $k$.

*if orientation*$(S) < 0$. *Also*

$$\textit{region\_of\_sphere}(S, p) = \begin{cases} \textit{bounded\_region} & \textit{if } p \textit{ lies inside } S \\ \textit{on\_region} & \textit{if } p \textit{ lies on } S \\ \textit{unbounded\_region} & \textit{if } p \textit{ lies outside } S \end{cases}$$

*Proof* Observe first that the assumption that $p_0, p_1, \ldots, p_d$ are affinely independent implies that

$$\textit{orientation}(S) = \textit{orientation}(p_0, p_1, \ldots, p_d) \neq 0.$$

Furthermore, by symmetry, we may assume without loss of generality that the points $p_0, p_1, \ldots, p_d$ are positively oriented. Under the assumption that $p_0, p_1, \ldots, p_d$ are positively oriented the following three statements are equivalent:

**(a)** $p$ is inside (on, outside) the sphere $S$.
**(b)** $\textit{lift}(p)$ lies below (on, above) the hyperplane through points $\textit{lift}(p_0), \textit{lift}(p_1), \ldots, \textit{lift}(p_d)$.
**(c)** $(\textit{lift}(p_0), \textit{lift}(p_1), \ldots, \textit{lift}(p_d), \textit{lift}(p))$ is negatively oriented.

We argued the equivalence of the first two items in the preceding section. The equivalence between the last two items follows from Lemma 3 in Section 8.2.2. This establishes the first claim. The second claim follows directly from the first. □

***Exercises for 10.9***

1 Let $p_0, p_1, \ldots, p_d$ be $d+1$ affinely dependent points ($\textit{orientation}(p_0, p_1, \ldots, p_d) = 0$) in base space and let $p$ be an additional point. Discuss the possible values of *side_of_sphere* and *region_of_sphere* for the $d+2$ tuple $(p_0, p_1, \ldots, p_d, p)$.
2 Assume that the base space is two-dimensional and that all points in $S$ lie on the line $x_0 + x_1 = 1$. What does the convex hull of $\textit{lift}(S)$ look like?
3 Assume that the base space is two-dimensional and that all points in $S$ lie on a circle. What does the convex hull of $\textit{lift}(S)$ look like?
4 Consider a circular range query with a square $C$ in a set $S$. Translate the query by the lifting map. What is the result?
5 Show how to implement a nearest neighbor query by use of the lifting map.

## 10.10 **A Complete Program: The Voronoi Demo**

We discuss the voronoi_demo in xlman. The demo illustrates many of the geometric algorithms available in LEDA and we have already seen several screen shots. The demo is also a representative example for the design of geometric demos in LEDA and useful as a starting point for the development of further demos. We start with an overview, then give the details of the implementation, and end with a discussion of what can go wrong when the demo is run with the floating point kernel.

It is best to have the demo running while reading this section. Figure 10.76 shows yet another screen shot of the demo. The window consists of a panel part and a display part. The panel part in turn is structured in four parts. There is a list of eleven choice items which control which geometric structures are to be displayed; in the situation shown only the button for the Delaunay diagram is pressed and hence only the Delaunay diagram is shown. There is a list of three choice items which control how mouse clicks in the display part of the window are to be interpreted. In the situation shown every click of the left mouse button adds a point. The other two buttons allow the user to input points and circles respectively. There is a choice item which allows the user to switch between the rational kernel and the floating point kernel, and there is a boolean item and a slider item that control whether the input points are rounded to a grid and how many grid lines there are. Finally, there are six buttons for opening sub-menus, for clearing the window, for asking for help, and for exiting the demo.

### 10.10.1 ***Overview***

The Voronoi demo allows the user to construct a scene of points and to visualize several fundamental geometric data structures for it: the nearest and furthest site Delaunay diagram, the nearest and furthest site Voronoi diagram, the convex hull and the width, the minimum spanning tree, the minimum enclosing and the maximum empty circle, the minimum width and the minimum area annuli, and the crust of the point set.

The point set is constructed either by mouse input or by calling one of the generators (sub-menu points). For mouse input there is the choice between single points, points on a line segment, and points on a circle. The current set of points is maintained as a list *p_list* of *rat_points*. The list is initially empty and is cleared by the clear-button. Any newly constructed point is added to it. It is important to remember that adding a line segment or adding a circle adds points that lie *exactly* on a line or a circle.

The geometric structures to be displayed can be computed with the use of three different geometry kernels: the rational kernel with the built-in floating point filter (this is the default), the rational kernel without the built-in floating point filter, and the floating point kernel. This allows the user to compare the relative speeds of the kernels and also to check visually whether the floating point kernel worked correctly. *When the floating point kernel is used, the program may abort or produce incorrect results.*

The geometric structures are not computed directly for the points in *p_list* but for a derived set of points. The derived set of points is called *rp_list* for use with the rational kernel and is called *fp_list* for use with the floating point kernel. The following procedure adds a point to *rp_list* and *fp_list*.

⟨*manipulate p_list, rp_list, and fp_list*⟩≡

```
void move_point(const rat_point& p)
{ point fp = p.to_point();
  if ( !round_to_grid )
  { fp_list.append(fp); rp_list.append(p); return; }
  double x = truncate(fp.xcoord(),truncation_prec);
  double y = truncate(fp.ycoord(),truncation_prec);
```

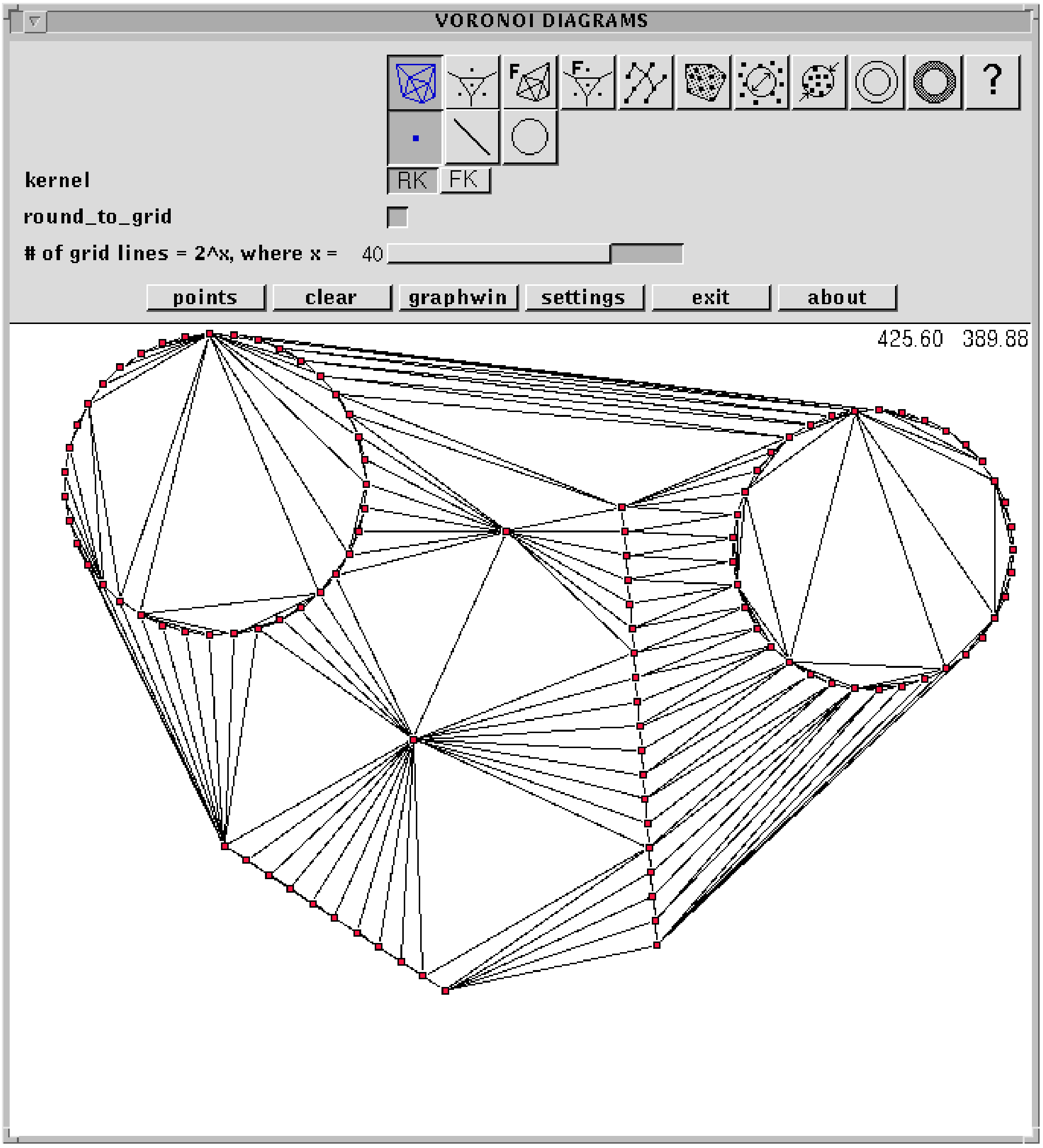


**Figure 10.76** A screen shot of the Voronoi demo. A Delaunay triangulation is displayed.

```
    point tp(x,y);
    fp_list.append(tp);
    rp_list.append(rat_point(tp));
  }
```

The addition of a point is controlled by variables *round_to_grid* and *truncation_prec*. Let $p$ be a *rat_point*. If *round_to_grid* is false, $p$ is added to *rp_list* and $fp = p.to\_point(\,)$ is added to *fp_list*; the Cartesian coordinates of *fp* are the optimal approximations of the rational coordinates of $p$ by *doubles*. Observe that when *round_to_grid* is false, the points $p$ and *fp* are in general distinct. In particular, if *p_list* contains points on a circle or segment, the corresponding points in *fp_list* will lie close to the circle or segment but not exactly on it.

Such inputs will frequently overburden the floating point kernel, e.g., try to construct the crust of co-circular points.

When *round_to_grid* is true, the mantissae of the Cartesian coordinates of *fp* are truncated to *truncation_prec* binary places, i.e., all but the first *truncation_prec* bits are set to zero. This moves the points on a grid with $2^{truncation\_prec}$ grid lines. The point with the truncated coordinates is then added to *fp_list* and *rp_list*. Truncation with small values of *truncation_prec* will visibly move the points. When *round_to_grid* is true, *rp_list* and *fp_list* contain the same set of points.

The demo also gives a feeling for the running time of the various algorithms. Whenever the user requests to change the display (for example, by requesting for an additional geometric structure, by dropping a request, or by switching to another kernel) *all* requested structures are recomputed.

The demo can make mistakes when run with the floating point kernel. When using the floating point kernel, set *round_to_grid* to true and play with *truncation_prec* to get a feeling for the limits of the floating point kernel. You can always switch to the rational kernel for a visual comparison of the result. We want to point out one frequently occurring mistake. When the crust of points on a circle is constructed and a high value of *truncation_prec* is used, the output is frequently completely wrong. This comes from the fact that crust constructs the Delaunay diagram of $fp\_list \cup VD(fp\_list)$, where $VD(fp\_list)$ denotes the set of vertices of the Voronoi diagram of *fp_list*. The latter set contains many points crowding near the center of the circle and this confuses the computation of the Delaunay diagram.

When the scene contains many points on circles or segments, the running time with the rational kernel may go up sharply. The reason is that these inputs are very difficult, because our generators guarantee that the points lie exactly on a circle or line, respectively.

### 10.10.2 *Implementation*

We start with the global structure of the program.

We use a global variable *p_list* to store the current set of points, a list *fp_list* to store the corresponding list of float points, a pointer *Wp* to the display window, and integers *display* and *input* that govern which geometric structure to display and which kind of geometric object is selected for input. The variable *kernel* controls which kernel is used and the variable *use_filter* controls whether the filter is used in the rational kernel (it can be changed in the settings menu). We have already explained the role of *use_to_grid* and *truncation_prec*.

In the main program we first set up the display window $W$ and then go into an infinite loop. At the beginning of the loop we wait for a mouse button to be pressed. The mouse button is either pressed on one of the seven buttons in the lower row of the panel section (cases zero to six) or in the display part of the window (case MOUSE_BUTTON(1)); the buttons in the top row of the display part are handled elsewhere as will be explained below.

In case of the event MOUSE_BUTTON(1) we put back the event, so that the mouse click can be processed again, and call $get\_input(W, input)$ to further process the mouse click.

At the end of the inner loop we draw the window as governed by the variable *display*.

⟨*voronoi_demo.c*⟩≡

```
#include <LEDA/plane_alg.h>
#include <LEDA/vector.h>
#include <LEDA/rat_vector.h>
#include <LEDA/window.h>
#include <LEDA/graphwin.h>
#include <LEDA/bitmaps/button32.h>
#include <math.h>
#include <LEDA/rat_window.h>
⟨definition of bit maps⟩
⟨definition of display mask⟩
static list<rat_point>  p_list, rp_list;
static list<point> fp_list;
static window* Wp;
static int display = 0;
static int input = 0;
enum { RK = 0, FK = 1};
static int kernel = RK;
static bool use_filter = true;
static int truncation_prec = 40;
static bool round_to_grid = true;
⟨further global variables⟩
⟨manipulate p_list, rp_list, and fp_list⟩
#include <LEDA/rat_kernel_names.h>
⟨displaying geometric structures⟩
⟨graph edit for graphwin⟩
#include <LEDA/kernel_names_undef.h>
#include <LEDA/float_kernel_names.h>
⟨displaying geometric structures⟩
⟨graph edit for graphwin⟩
#include <LEDA/kernel_names_undef.h>
⟨global drawing functions⟩
⟨action functions⟩
⟨point generators⟩
⟨adding a geometric object⟩
int main()
{
  window W(630,720,"VORONOI DIAGRAMS");
  Wp = &W;
  ⟨set up window⟩
for(;;)
{
 int but = W.read_mouse();
  rat_point::use_filter = use_filter;
  if (but == 0) break;
```

```
    switch (but) {
    case MOUSE_BUTTON(1): put_back_event();
                          get_input(W,input);
                          break;
    case 1: { ⟨generate points menu⟩; break; }
    case 2: { ⟨settings menu⟩; break; }
    case 3:  clear_all(); break;
    case 4:  // start GraphWin
             if ( kernel == FK )
                graph_edit(display,fp_list);
             else
                graph_edit(display,rp_list);
             break;
    case 5:  // help
             help_win.open(W); break;
    }
    draw(display);
   }
  rat_point::print_statistics();
  return 0;
 }
```

The drawing functions are needed for both kernels and hence are included twice. We comment below why we did not use templates.

We give more details.

**Setting up the Window:** We start by defining a *help_string* and the panel *help_win* that pops up when the "about"-button is pressed. We then define the panel section of $W$. It consists of three sets of *choice_items*, a boolean item, a slider item, and a set of six buttons. We come back to them below.

Having defined the panel part we open the display, state that window coordinates for the $x$-coordinate are between 0 and 1000 and that they start at 0 for the $y$-coordinate (the upper bound for the $y$-coordinate depends on the actual geometry of $W$, state that nodes are drawn with width two, and that coordinates are to be shown.

⟨*set up window*⟩≡

```
string help_string;
help_string += "This program demonstrates some of the algorithms ";
help_string += "for two dimensional geometry of points based on ";
help_string += "Delaunay triangulations and Voronoi Diagrams.";

panel help_win;

help_win.text_item("\\bf Voronoi Demo");
help_win.text_item("");
help_win.text_item("K. Mehlhorn and S. Naeher (1997)");
help_win.text_item("");
help_win.text_item("see LEDAROOT/demo/documentation/voronoi_demo.ps");
```

```
help_win.text_item(help_string);
help_win.button("ok");
W.set_bitmap_colors(black,blue);
W.choice_mult_item("",display,11,32,32,display_bits,draw);
W.choice_item("",input,3,32,32,input_bits);
list<string> kernel_choices;
kernel_choices.append("RK"); kernel_choices.append("FK");
W.choice_item("kernel",kernel,kernel_choices,change_kernel);
W.bool_item("round_to_grid",round_to_grid,change_round_to_grid);
W.int_item("# of grid lines = 2^x, where x =",truncation_prec,
            2,52,change_truncation_prec);
W.button("points",   1, "Opens a point generator panel.");
W.button("clear",    3, "Clears point set and window.");
W.button("graphwin", 4, "Loads graph into GraphWin.");
W.button("settings", 2, "Opens an option setting dialog.");
W.button("exit",     0, "Exits the program.");
W.button("about",    5, "Displays information about this program.");
W.display();
W.init(0,1000,0);
W.set_redraw(redraw);
W.set_node_width(2);
W.set_show_coordinates(true);
```

We need to say a few more words about the panel part of the window. The first choice item controls the variable *display* and consists of eleven items. Whenever the $i$-th button is pressed the $i$-th bit of *display* is flipped and the function call *draw*(*display*) is made. Each item is drawn as a 32x32 pixel map taken from the collection of pixel maps defined in LEDA/bitmaps/button32.h. The pixel maps selected are defined by the array *display_bits*. The pixel maps are shown black when the corresponding button is released and are shown in blue when the button is pressed.

The second choice item controls the variable *input*. The effect of pressing one of the buttons in this collection of buttons is to set *input* to the number of the button.

The third choice item controls the use of the filter, the boolean item controls whether the input is rounded to a grid, and the slider item controls the number of grid lines.

The other buttons are added by the seven *button* statements. Each button is given a name, a number, and a help string that is displayed when the mouse rests over the button for an extended period of time.

⟨*definition of bit maps*⟩≡

```
static char* input_bits [] = { point_bits, line_bits, circle_bits };
static char* display_bits [] = { triang_bits, voro_bits, f_triang_bits,
  f_voro_bits, tree_bits, hull_bits, empty_circle_bits,
  encl_circle_bits, w_annulus_bits, a_annulus_bits, help_bits };
```

**Action Functions:** Some of the items in the menu part of the window have action functions associated with them. Recall that action functions are called with the new value of the variable associated with the item (the value of the variable itself is only changed after return from the action function such that new and old values of the variable are available during the action). All action functions follow the same scheme. They set the corresponding variable to the new value (since we want the new value during the execution of the action), clear the window and redraw the sites, recompute *rp_list* and *fp_list*, and recompute the display. The function *draw* will be discussed below.

⟨*action functions*⟩≡
```
void change_truncation_prec(int new_prec)
{ truncation_prec = new_prec;
  Wp->clear();
  draw_sites(p_list);
  recompute_rp_and_fp_list();
  draw(display);
}
void change_round_to_grid(int new_mode)
{ round_to_grid = new_mode;
  Wp->clear();
  draw_sites(p_list);
  recompute_rp_and_fp_list();
  draw(display);
}
void change_kernel(int new_kernel)
{ kernel = new_kernel;
  Wp->clear();
  draw_sites(p_list);
  draw(display);
}
```

The function *recompute_rp_and_fp_list* clears both lists and then moves all points from *p_list*. The function *add_point* will be called whenever a new point is added to *p_list* and *clear_all* clears the window and all lists.

⟨*manipulate p_list, rp_list, and fp_list*⟩+≡
```
void add_point(const rat_point& p)
{ p_list.append(p);
  move_point(p);
}
void recompute_rp_and_fp_list()
{ fp_list.clear(); rp_list.clear();
  rat_point p;
  forall(p,p_list) move_point(p);
}
void clear_all()
{ Wp->clear();
  p_list.clear(); fp_list.clear(); rp_list.clear();
}
```

**Global Drawing Functions:** The function *draw_area*(*disp*, *x0*, *y0*, *x1*, *y1*, *L*) draws the part of *W* covered by the rectangle with lower corner (*x0*, *y0*) and upper corner (*x1*, *y1*). It is our master drawing function. The geometric structures shown are governed by *disp* and *L* is either *p_list* or *fp_list*. If *L* is *p_list* the drawing functions use the rational kernel and if *L* is *fp_list* the drawing functions use the floating point kernel.

⟨*global drawing functions*⟩≡

```
template <class POINT>
void draw_area(int disp, double x0, double y0, double x1, double y1,
               const list<POINT>& L)
{
  if (L.empty()) return;

  Wp->start_buffering();
  Wp->clear();

  if (disp & MWA_MASK)   draw_min_width_annulus(L);
  if (disp & MAA_MASK)   draw_min_area_annulus(L);
  if (disp & HULL_MASK)  draw_convex_hull(L);
  if (disp & DT_MASK)    draw_delaunay(L);
  if (disp & VD_MASK)    draw_voronoi(L);
  if (disp & FDT_MASK)   draw_f_delaunay(L);
  if (disp & FVD_MASK)   draw_f_voronoi(L);
  if (disp & LEC_MASK)   draw_max_empty_circle(L);
  if (disp & SEC_MASK)   draw_min_encl_circle(L);
  if (disp & MST_MASK)   draw_min_span_tree(L);
  if (disp & CRUST_MASK) draw_crust(L);

  draw_sites(L);

  Wp->flush_buffer(x0,y0,x1,y1);
  Wp->stop_buffering();
}
```

If our current set of sites is empty, *draw_area* has nothing to do. Otherwise we clear the window, draw the selected geometric structures (the constants MWA_MASK, MAA_MASK, ... are defined in an enumeration type and denote $2^0$, $2^1$, $2^2$, ...), and draw the sites. The appearance of the window is better if the sites are displayed after the selected geometric structures. We want the new drawing to appear in a single blow and therefore put the window in buffering mode before constructing the drawings of the selected geometric structures.

Once all drawings are constructed we flush the buffer and stop the buffering mode.

⟨*definition of display mask*⟩≡

```
enum display_mask {
  DT_MASK    =    1,  VD_MASK    =    2,  FDT_MASK   =    4,
  FVD_MASK   =    8,  MST_MASK   =   16,  HULL_MASK  =   32,
  LEC_MASK   =   64,  SEC_MASK   =  128,  MWA_MASK   =  256,
  MAA_MASK   =  512,  CRUST_MASK = 1024
};
```

The master drawing function is used by the functions *draw_area*, *draw* and *redraw*.

*Draw_area* (now without the *list<POINT>*-argument) makes the distinction between the use of the rational kernel and the floating point kernel.

*Draw* is called whenever one of the choice items changing *display* is called and at the end of each iteration of the main loop and redraw is called whenever the geometry of the window is changed. Accordingly, we redraw either only the display part of the window (in *draw*) or the entire window (in *redraw*).

⟨*global drawing functions*⟩+≡

```
void draw_area(int disp, double x0, double y0, double x1, double y1)
{
  if ( kernel == FK ) draw_area(disp,x0,y0,x1,y1,fp_list);
  else                draw_area(disp,x0,y0,x1,y1,rp_list);
}
void draw(int disp)
{ draw_area(disp,Wp->xmin(),Wp->ymin(),Wp->xmax(),Wp->ymax()); }
void redraw(window* wp, double x0, double y0, double x1, double y1)
{ draw_area(display,x0,y0,x1,y1); }
```

**Displaying Specific Geometric Structures:** For each of our geometric structures we have a function that displays it. We discuss only a representative sample of the functions.

We draw each site as a filled node of color *site_color*, where *site_color* is a global variable defined in ⟨*further global variables*⟩. This code is not shown. The default value of *site_color* is red; the color can be changed in the settings menu.

⟨*displaying geometric structures*⟩≡

```
void draw_sites(const list<POINT>& L)
{ POINT p;
  forall(p,L) Wp->draw_filled_node(p.to_point(),site_color);
}
```

Most of our geometric structures are graphs. We have to deal with two kinds of graphs. Voronoi diagrams have type *GRAPH<CIRCLE, POINT>* and Delaunay diagrams have type *GRAPH<POINT, int>*. We define a drawing function for each kind of graph. Recall that we use bidirected graphs to represent Delaunay diagrams and Voronoi diagrams. We therefore have to draw uedges and not edges.

In order to draw a *GRAPH<POINT, int>* we simply draw each uedge as the segment defined by the endpoints of the edge.

⟨*displaying geometric structures*⟩+≡

```
void draw_graph_edges(const GRAPH<POINT,int>& T, color col)
{ edge_array<bool> drawn(T,false);
  edge e;
  forall_edges(e,T)
    if (!drawn[e])
    { drawn[e] = true;
      edge r = T.reversal(e);
```

```
      if (r) drawn[r] = true;
      POINT p = T[source(e)];
      POINT q = T[target(e)];
      Wp->draw_edge(p.to_point(),q.to_point(),col);
    }
}
```

Voronoi diagrams are a bit harder to draw. The positions of the nodes are determined by the circles associated with them. A proper node, i.e., a node of degree at least three, is positioned at the center of the circle associated with it. A node of degree one is positioned at the circle at infinity. If its circle is *CIRCLE*$(a, \_, b)$ then the node lies on the perpendicular bisector of $a$ and $b$, and to the left of the oriented segment from $a$ to $b$. Each edge is labeled by the site owning the region to the left of the edge. An edge $e$ is part of the perpendicular bisector of sites $a$ and $b$, where $a = G[e]$ and $b = G[G.reversal(e)]$.

After these preliminaries it is clear how to draw a Voronoi edge $(v, w)$. An edge connecting two improper nodes is drawn as the perpendicular bisector of the points $a$ and $b$, an edge connecting a proper node and an improper node is drawn as a ray starting at the proper node, running along the perpendicular bisector of points $a$ and $b$ and extending towards the position of the improper node at the circle at infinity, and an edge connecting two proper nodes is drawn as a segment connecting the nodes. We obtain the following code.

⟨*draw_voro_edges*⟩≡

```
void draw_voro_edges(const GRAPH<CIRCLE,POINT>& VD, color col)
{
  edge_array<bool> drawn(VD,false);
  edge e;
  forall_edges(e,VD)
  { if (drawn[e]) continue;
    drawn[VD.reversal(e)] = drawn[e] = true;
    node v = source(e);
    node w = target(e);
    POINT a = VD[e];
    POINT b = VD[VD.reversal(e)];
    VECTOR vec = (b - a).rotate90();
    line l = p_bisector(a,b).to_line();
    if (VD.outdeg(v) == 1 && VD.outdeg(w) == 1){ Wp->draw_line(l,col); }
    else
      if (VD.outdeg(w) == 1)
      { POINT cv = VD[v].center();
        VECTOR vec = VD[w].point3() - VD[w].point1();
        POINT  rp  = cv + vec.rotate90();
        Wp->draw_ray(cv.to_point(),rp.to_point(),col);
      }
      else
        if (VD.outdeg(v) == 1)
        { POINT cw = VD[w].center();
          VECTOR vec = VD[v].point3() - VD[v].point1();
          POINT  rp  = cw + vec.rotate90();
```

```
          Wp->draw_ray(cw.to_point(),rp.to_point(),col);
        }
        else
        { POINT cv = VD[v].center();
          POINT cw = VD[w].center();
          Wp->draw_segment(cv.to_point(),cw.to_point(),col);
        }
    }
  }
```

The procedure above has serious numerical differences. Consider the following example. Assume that we compute the Voronoi diagram of three points that lie almost on a common line. The Voronoi diagram consists of one vertex and three rays. The vertex has very large coordinates and even if its coordinates are computed exactly (as they will be with the rational kernel) the conversion to point in *draw_ray* will suffer some loss of accuracy. We are now drawing a ray from a distant point. It is unlikely that this ray intersects the window in the desired form.

The window class offers drawing functions that are appropriate for this situation as discussed in Section 8.1. The modified drawing functions have an additional argument $l$ of type *line*, which is supposed to be the line underlying the segment $s$ or ray $r$ to be drawn. In our case $l$ is the bisector of $a$ and $b$ and hence determined with high precision. The additional argument is used as follows.

If the source of $r$ lies in $W$ or the two endpoints of $s$ lie in $W$, $l$ is ignored. Otherwise, the intersection $t$ between $l$ and the window is determined and the part of $t$ which also belongs to $r$ or $s$ is drawn.

⟨*displaying geometric structures*⟩+≡

```
  // template <class POINT, class CIRCLE, class VECTOR, class LINE>
  void draw_voro_edges(const GRAPH<CIRCLE,POINT>& VD, color col)
  {
    edge_array<bool> drawn(VD,false);
    edge e;
    forall_edges(e,VD)
    { if (drawn[e]) continue;
      drawn[VD.reversal(e)] = drawn[e] = true;
      node v = source(e);
      node w = target(e);
      POINT a = VD[e];
      POINT b = VD[VD.reversal(e)];
      line l = p_bisector(a,b).to_line();
      if (VD.outdeg(v) == 1 && VD.outdeg(w) == 1){ Wp->draw_line(l,col); }
      else
        if (VD.outdeg(w) == 1)
        { POINT cv = VD[v].center();
          VECTOR vec = VD[w].point3() - VD[w].point1();
          POINT  rp  = cv + vec.rotate90();
          Wp->draw_ray(cv.to_point(),rp.to_point(),l,col);
```

```
        }
        else
          if (VD.outdeg(v) == 1)
          { POINT cw = VD[w].center();
            VECTOR vec = VD[v].point3() - VD[v].point1();
            POINT  rp  = cw + vec.rotate90();
            Wp->draw_ray(cw.to_point(),rp.to_point(),1,col);
          }
          else
          { POINT cv = VD[v].center();
            POINT cw = VD[w].center();
            Wp->draw_segment(cv.to_point(),cw.to_point(),1,col);
          }
    }
  }
```

The function above uses points, lines, circles, and vectors and hence would require four template arguments. Moreover, we would have to add artificial arguments of type LINE and VECTOR such that the appropriate type inference can be made by the compiler. We decided to use our primitive renaming mechanism instead. An alternative would be to introduce a class *rat_kernel*

```
class rat_kernel{
  typedef rat_point   POINT;
  typedef rat_segment SEGMENT;
  // and so on
}
```

and a similar class *float_kernel*, to use a single template argument called *kernel*, and to use qualified type names such as *kernel*::*POINT* and *kernel*::*SEGMENT* in *draw_voro_edges*. This design is used extensively in CGAL [CGA].

We come to the drawing functions for the individual geometric structures. Nearest and furthest sites Delaunay diagrams, crusts, and minimum spanning trees are drawn by first computing the structure and then calling *draw_graph_edges*. For example,

⟨*displaying geometric structures*⟩+≡

```
void draw_delaunay(const list<POINT>& L)
{ GRAPH<POINT,int>  DT;
  DELAUNAY_TRIANG(L,DT);
  draw_graph_edges(DT,triang_color);
}
```

Nearest and furthest site Voronoi diagrams are drawn by computing the structure and calling *draw_voro_edges*. For example,

⟨*displaying geometric structures*⟩+≡

```
void draw_voronoi(const list<POINT>& L)
{ GRAPH<CIRCLE,POINT> VD;
  VORONOI(L,VD);
  draw_voro_edges(VD,voro_color);
}
```

In order to display the convex hull and the width of our set of points we compute the convex hull (a list of POINTs), convert the list to a list of *points*, and draw the list of points as a filled polygon of *hull_color* and as a black polygonal line. We also compute the minimum width slab containing our set of points and display the two lines bounding the slab.

⟨*displaying geometric structures*⟩+≡

```
void draw_convex_hull(const list<POINT>& L)
{ list<POINT> CH = CONVEX_HULL(L);
  list<point> pol;
  POINT p;
  forall(p,CH) pol.append(p.to_point());
  Wp->draw_filled_polygon(pol,hull_color);
  Wp->draw_polygon(pol,black);
  // width
  LINE l1,l2;
  WIDTH(L,l1,l2);
  Wp->draw_line(l1.to_line(),blue);
  Wp->draw_line(l2.to_line(),blue);
}
```

In order to draw a minimum width annulus we either draw the two circles or the two parallel lines defining the annulus. In the first case we want the annulus to be shown in orange. We therefore draw the larger disk in orange first and then the smaller disk in white. This leaves the annulus in orange.

⟨*displaying geometric structures*⟩+≡

```
void draw_min_width_annulus(const list<POINT>& L)
{ POINT a,b,c; LINE l1,l2;
  if ( MIN_WIDTH_ANNULUS(L,a,b,c,l1,l2) )
  { // proper annulus
    circle c1(a.to_point(),b.to_point());
    circle c2(a.to_point(),c.to_point());
    Wp->draw_disc(c2,orange);
    Wp->draw_disc(c1,white);
    Wp->draw_circle(c1,black);
    Wp->draw_circle(c2,black);
    Wp->draw_point(a.to_point(),orange);
  }
  else
  { // strip
    Wp->draw_line(l1.to_line(),black);
```

```
    Wp->draw_line(l2.to_line(),black);
  }
}
```

**Adding a Geometric Object:** We come to the mouse input of points, lines, and circles. The function *get_input*($W$, *input*) reads either a point, or a segment, or a circle and then calls the appropriate insertion function.

⟨*adding a geometric object*⟩≡

⟨*adding a point, segment or circle*⟩

```
void get_input(window& W, int inp)
{ rat_point p; rat_segment s; rat_circle c;
  switch (inp) {
    case 0: if (W >> p) insert_point(p);   break;
    case 1: if (W >> s) insert_segment(s); break;
    case 2: if (W >> c) insert_circle(c);  break;
  }
}
```

⟨*adding a point, segment or circle*⟩≡

```
void insert_point(rat_point p)
{ Wp->draw_filled_node(p.to_point(),site_color);
  add_point(p);
}
```

Addition of a point does the obvious. In order to add points on a segment we generate $n$ points on the segment, where $n$ is determined by the ratio between the length of the segment and the global variable *point_dist*.

In order to add a circle we generate $n$ uniformly spaced points on the circle, where $n$ is determined by the ratio between the circumference of the circle and the global variable *point_dist*.

⟨*adding a point, segment or circle*⟩+≡

```
void insert_segment(rat_segment s)
{
  double l = s.to_segment().length();
  int n = Wp->real_to_pix(l)/point_dist + 1;
  list<rat_point> L;
  points_on_segment(s,n,L);
  rat_point p;
  forall(p,L)
  { add_point(p);
    Wp->draw_filled_node(p.to_point(),site_color);
  }
}
void insert_circle(rat_circle C)
```

```
{
  double L = 2 * C.to_circle().radius() * LEDA_PI;
  int n = Wp->real_to_pix(L)/point_dist + 1;
  double d = (2*LEDA_PI)/n;
  double eps = 0.001;
  double a = 0;
  for(int i = 0; i < n; i++)
  { rat_point q = C.point_on_circle(a,eps);
    add_point(q);
    Wp->draw_filled_node(q.to_point(),site_color);
    a += d;
  }
}
```

**Point Generators:** The point generator menu allows the user to select between three generators. A generator for random points in a square, a generator for regularly spaced points, and a generator for random points near a circle. The third generator produces inputs which are useful to illustrate the computation of annuli.

⟨*generate points menu*⟩≡

```
panel P;
P.text_item("\\bf Generate input points");
P.text_item("");
P.choice_item("",k_gen,"random","lattice","near circle");
P.int_item("",n_gen,0,500);
P.button("create",0);
P.button("cancel",1);
if (P.open(W) == 0)
{ switch (k_gen) {
  case 0: random_square(n_gen); break;
  case 1: lattice_points(n_gen); break;
  case 2: near_circle(n_gen,point_dist); break;
  }
}
```

We only show the *near_circle* generator. It generates points in an annulus with inner radius *rmin* and outer radius *rmax*; *rmax* is chosen such that the annulus fits nicely on the screen and *rmin* is chosen as 90% of *rmax*.

For each point to be generated we generate a random point on a circle of radius *r* where *r* is randomly chosen between *rmin* and *rmax*.

⟨*point generators*⟩+≡

```
void near_circle(int n, int point_dist)
{
  double x0 = Wp->xmin(), y0 = Wp->ymin();
  double x1 = Wp->xmax(), y1 = Wp->ymax();
  point cent((x0+x1)/2,(y0+y1)/2);
  int rmax = int(0.35 * (x1-x0));
```

```
    int rmin = int(0.9 * rmax);
    clear_all();
    for(int i=0; i < n; i++)
    { //circle C(cent,rand_int(rmin,rmax));
      circle C(cent,(double)rand_int(rmin,rmax));
      double a;
      rand_int >> a;
      point q = C.point_on_circle(2*a*LEDA_PI);
      int x = (int)q.xcoord();
      int y = (int)q.ycoord();
      add_point(rat_point(x,y,1));
      Wp->draw_filled_node(x,y,site_color);
    }
  }
```

**Calling GraphWin:** The function *graph_edit* visualizes the graphs underlying our geometric structures. We do not discuss it here.

**Settings:** The settings menu allows the user to set some of the global variables. It is self-explanatory.

⟨*settings menu*⟩≡

```
  panel SP("SETTINGS");
  SP.bool_item("use filter in rat kernel", use_filter);
  SP.bool_item("draw lines with width 2",thick_lines);
  SP.int_item("grid", grid_width,0,50,10);
  SP.int_item("pix dist", point_dist,1,64);
  SP.color_item("sites ", site_color);
  SP.color_item("voro  ", voro_color);
  SP.color_item("triang", triang_color);
  SP.color_item("hull",   hull_color);
  SP.color_item("tree",   tree_color);
  SP.button("continue");
  SP.open(W);
  W.set_grid_mode(grid_width);
  W.clear();
  W.set_line_width( thick_lines ? 2 : 1);
  draw_sites(p_list);
  recompute_rp_and_fp_list();
  draw(display);
```

### 10.10.3 *Floating Point Errors*

What can go wrong when the demo is executed with the floating point kernel?

When a segment or circle is added a certain number of points on the segment or circle are added to *p_list*. The rational kernel guarantees that these points lie exactly on the segment or circle, respectively. When the *rat_points* are converted to *points*, they will lie only almost on the circle or segment.

Consider now a scene that consists of points on two segments. The Delaunay triangulation will contain extremely flat triangles. This can cause the computation of the Delaunay diagram and the Voronoi diagram to fail.

Crust is also a good source of error. It computes the Delaunay diagram of the points is *fp_list* plus the vertices of the Voronoi diagram of *p_list*. When *fp_list* contains points that lie almost on a circle there will be many Voronoi vertices near the center of the circle and the Delaunay diagram computation will get confused. This can lead to strange crusts.

# Bibliography


[ABE98] N. Amenta, M. Bern, and D. Eppstein. The crust and the $\beta$-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing*, pages 125–135, 1998.

[And79] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.

[AST94] P.K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. *Journal of Algorithms*, 17:292–318, 1994.

[Bal95] Balaban. An optimal algorithm for finding segment intersections. In *Proceedings of the 11th Annual ACM Symposium on Computational Geometry (SCG'95)*, 1995.

[BD95] P. Bose and L. Devroye. Intersections with random geometric objects. manuscript, 1995.

[BFMh97] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 702–709, 1997. `www.mpi-sb.mpg.de/~mehlhorn/ftp/sep-bound.ps`.

[BMh94] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimenta l results. In *Proceedings of the 2nd Annual European Symposium on Algorithms - ESA'94*, volume 855 of *Lecture Notes in Computer Science*, pages 227–239. Springer, 1994.

[BMh96] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, January 1996.

[BO79] J.L. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transaction on Computers C 28*, pages 643–647, 1979.

[BT93] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. *Theoretical Computer Science*, 112(2):339–354, 1993.

[Bur96] C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. PhD thesis, Max-Planck-Institut für Informatik, 1996.

[BY98] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, Cambridge, 1998.

[CGA] CGAL (Computational Geometry Algorithms Library). `www.cs.ruu.nl/CGAL`.

[CS89] K.L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Journal of Discrete and Computational Geometry*, 4:387–421, 1989.

[dBKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[DLPT97] O. Devillers, G. Liotta, F.P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. Technical report, Center for Geometric Computing, Department of Computer Science, Brown Universi ty, 1997.

[Dwy87] R.A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, 1987.

[For87] S.J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hulls of a finite point set. *Information Processing Letters*, 1:132–133, 1972.

[GS85] L.J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[Hof89] C.M. Hoffmann. *Geometric and solid modeling : an introduction. 2nd pr. 1993.* Series in computer graphics and geometric modeling. Morgan Kaufmann, 1989.

[Kle97] R. Klein. *Algorithmische Geometrie*. Addison-Wesley, 1997.

[Law72] C.L. Lawson. Transforming triangulations. *Discrete Mathematics*, pages 365–372, 1972.

[LEP] LEDA Extension Packages. `www.mpi-sb.mpg.de/LEDA/friends/leps.html`.

[Meh84a] K. Mehlhorn. *Data Structures and Algorithms 1,2, and 3.* Springer, 1984.

[Meh84b] K. Mehlhorn. *Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry*. Springer, 1984.

[MMN+98] K. Mehlhorn, M. Müller, S. Näher, S. S. Schirra, M. Seel, C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and its applications. *Computational Geometry: Theory and Applications*, 10:289–303, 1998. `www.mpi-sb.mpg.de/~seel`.

[MN98] K. Mehlhorn and S. Näher. Dynamic Delaunay triangulations, 1998. `www.mpi-sb.mpg.de/~mehlhorn`.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (SCG'96)*, pages 159–165, 1996.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Mul90] K. Mulmuley. A fast planar partition algorithm, I. *Journal of Symbolic Computation*, 10(3-4):253–280, 1990.

[Mul94] K. Mulmuley. *Computational Geometry*. Prentice Hall, 1994.

[Mye85] E. Myers. An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem. *SIAM Journal of Computing*, 14(3):625–636, 1985.

[Nef78] W. Nef. *Beitraege Zur Theorie Der Polyeder Mit Anwendungen in der Computergraphik (Contributions to the Theory of Polyhedra, with Applications in Computer Graphics).* Verlag H. Lang & Cie. AG, Bern, 1978.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.

[Req80] A.A.G. Requicha. Representations for rigid solids: Theory, methods and systems. *Computing Surveys*, 12(4):437–464, 1980.

[Sch98] S. Schirra. Parameterized implementations of classical planar convex hull algorithms and extreme point computations. Technical Report MPI-I-98-1-003, Max-Planck-Institut für Informatik, 1998.

[SD97] P. Su and R.L.S. Drysdale. A comparison of seqential Delaunay triangulation algorithms. *Computational Geometry: Theory and Applications*, 7:361–385, 1997.

[See97] M. Seel. The computation of abstract Voronoi diagrams. `www.mpi-sb.mpg.de/~seel`, 1997.

[Sei91] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete and Computational Geometry*, 6:423–434, 1991.

[SH75] M.I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS'75)*, pages 151–165, 1975.

[TR80] R.B. Tilove and A.A.G. Requicha. Closure of boolean operations on geometric entities. *Computer-Aided Design*, 12:219–222, 1980.

# Index