Contents

0.1 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[1] 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 0.1.1 we discuss the functionality of polygons and generalized polygons, in Section 0.1.2 we give the essentials of the implementation of polygons, in Section 0.1.3 we give the mathematics underlying the representation of generalized polygons, and in Section 0.1.4 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 0.1.

[1] A precise definition is given below.

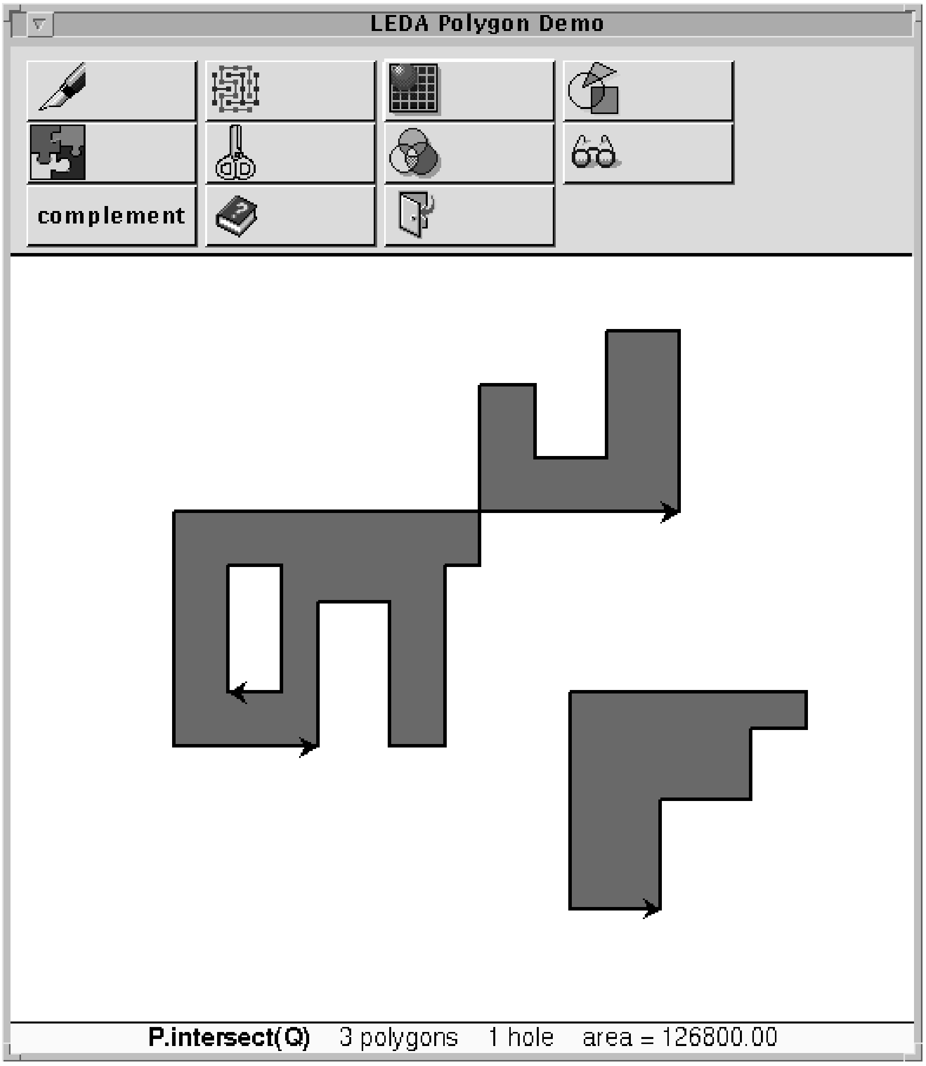

Figure 0.1 A screen shot of the polygon demo in xlman. The display shows a generalized polygon. The boundary cycles are indicated by arrows and the inside of the polygon is shaded. The various buttons allow the user to construct polygons by mouse input or by calling generators, to force vertices to a grid, to compute intersections, unions, differences, and symmetric differences, to perform point location queries, and to compute complements.

0.1.1 *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 $S(P)$

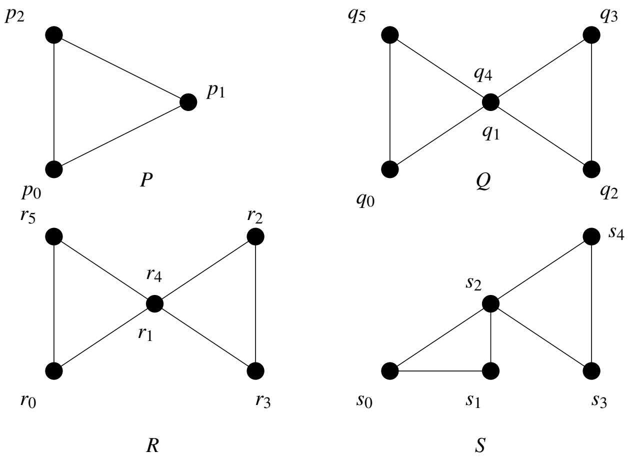

Figure 0.2 P is simple and Q is weakly simple but not simple. R is not weakly simple because it crosses itself at $r = r_1 = r_4$, and S is not weakly simple since s_2 lies in the interior of another segment.

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[2] and if the chain does not cross itself. Figure 0.2 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 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 0.3. 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 functionality, but, of course, only *rat_polygons* guarantee correct results. We use *rat_polygons* in this section.

The declarations

[2] It is allowed that segments that are not consecutive on P share an endpoint.

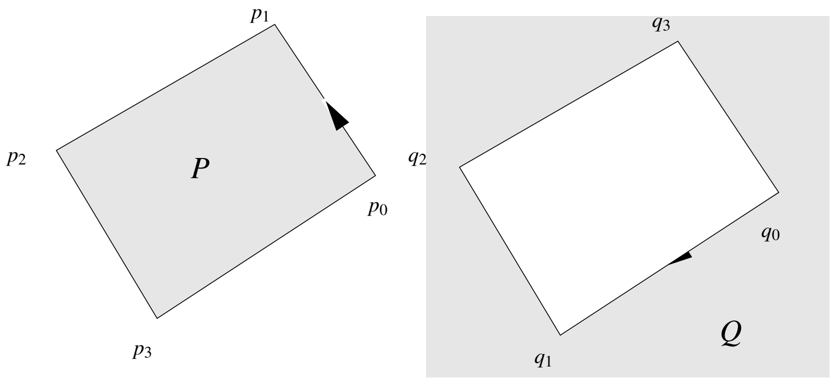

Figure 0.3 The bounded region is to the left of P; P is positively oriented. The unbounded region is to the left of Q, Q is negatively oriented.

```
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

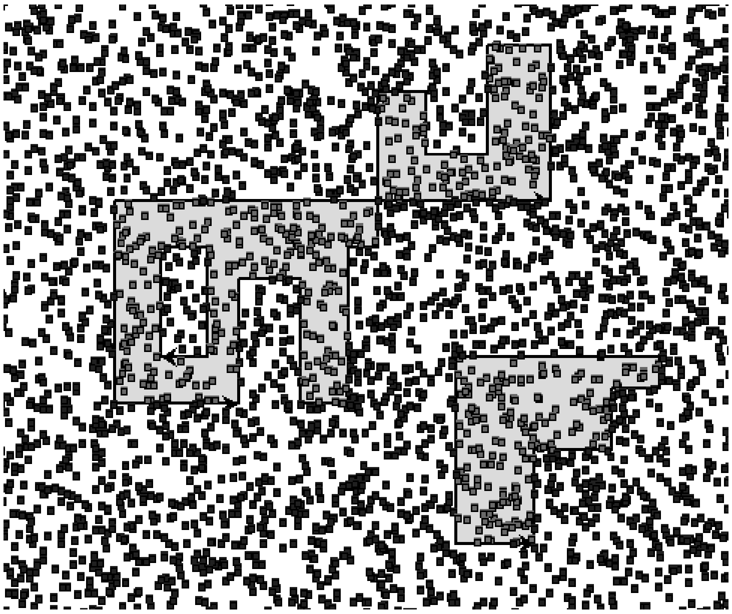

Figure 0.4 Side-of tests: We performed side-of tests with respect to the generalized polygon of Figure 0.1 for 5000 random points. The points on the different sides are shown at different grey level.

```
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 0.4. 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);
```

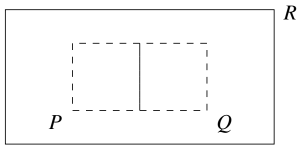

Figure 0.5 The intersection of P and Q is a line segment; $R \setminus (P \cap Q)$ is a rectangle minus a line segment.

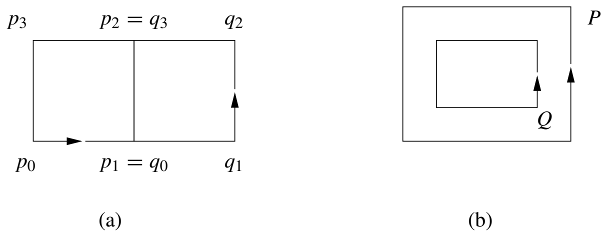

Figure 0.6 In (a), the polygons $P$and Q share an edge, $P \cap Q$ is a closed line segment, and $\text{reg}(P \cap Q)$ is the empty set. In (b), $P \setminus Q$ is the half-closed region between the cycles P and Q; the chain P does not belong to $P \setminus Q$ and the chain Q belongs to it. The regularized set difference $\text{reg}(P \setminus Q)$ is the open region with boundaries P and Q.

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 0.5. 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 $\text{int}\, X$, $\text{cl}\, X$, $\text{bd}\, X$, and $\text{cpl}\, X$ to denote its *interior*, *closure*, *boundary*, and *complement*, respectively. An open set X is called *regular* if $X = \text{int}\,\text{cl}\, X$. 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[3] applied to regular polygonal regions generate regular polygonal regions, see Figure 0.6.

Lemma 1

[3] 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.

(a) *Let X be any set. Then* $\operatorname{reg} X$ *is regular.*
(b) *Let X be any open set. X is regular iff X and* $\operatorname{int}\operatorname{cpl} X$ *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* $\operatorname{reg}\operatorname{cpl} P$, $\operatorname{reg}(P \cap Q)$, $\operatorname{reg}(P \cup Q)$, $\operatorname{reg}(P \setminus Q)$, *and* $\operatorname{reg}(P \oplus Q)$.

Proof We start with part (a). Let X be any set and let $Y = \operatorname{reg} X$. We need to show that Y is regular. We have $Y \subseteq \operatorname{cl} Y$ and hence $Y \subseteq \operatorname{int}\operatorname{cl} Y$ since Y is open. We have $Y \subseteq \operatorname{cl} X$ by definition of Y and hence $\operatorname{cl} Y \subseteq \operatorname{cl}\operatorname{cl} X = \operatorname{cl} X$. Thus $\operatorname{int}\operatorname{cl} Y \subseteq \operatorname{int}\operatorname{cl} X = Y$.

We turn to part (b). Assume first that X is regular, i.e., $X = \operatorname{int}\operatorname{cl} X$, and let x be any point in the boundary of X. Then $x \in \operatorname{cl} X \setminus X$ since X is open. Assume that there is a neighborhood U of x such that $U \cap \operatorname{int}\operatorname{cpl} X = \emptyset$. Then $U \subseteq \operatorname{cl} X$ and hence $x \in \operatorname{int}\operatorname{cl} X$, a contradiction to the regularity of X.

To prove the converse we observe that $X \subseteq \operatorname{int}\operatorname{cl} X$ since X is open. We need to show that the containment is not proper. Consider any point $x \in \operatorname{bd} X$. By assumption every neighborhood U of x has $U \cap \operatorname{int}\operatorname{cpl} X \neq \emptyset$. Thus $x \notin \operatorname{int}\operatorname{cl} X$ and hence $\operatorname{int}\operatorname{cl} X \subseteq X$.

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.

The results of the regularized set operations are certainly polygonal; regularity follows from part (a). $\square$

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 0.4.

The function

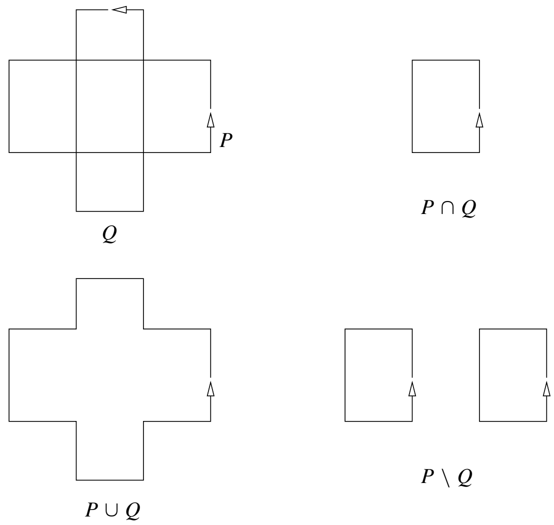

Figure 0.7 Two polygons P and Q and the results of the three boolean operations $\cap$, $\cup$, and $\setminus$.

```
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);
```

return $\text{reg}(P \cup Q)$, $\text{reg}(P \cap Q)$, $\text{reg}(P \setminus Q)$, and $\text{reg}(P \oplus Q)$, respectively. The word *union* is a reserved word of C++, hence the name *unite* for the union-operation. Figures 0.7 and 0.8 show some examples.

A generalized polygon can be represented by its boundary cycles as will be explained in Section 0.1.3. The function

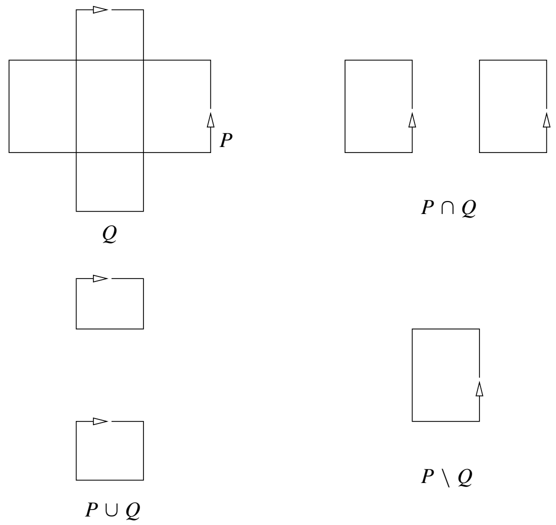

Figure 0.8 Two polygons P and Q and the results of the three boolean operations $\cap$, $\cup$, and $\setminus$. Observe that the positive side of Q is unbounded.

```
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.

0.1.2 *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*.

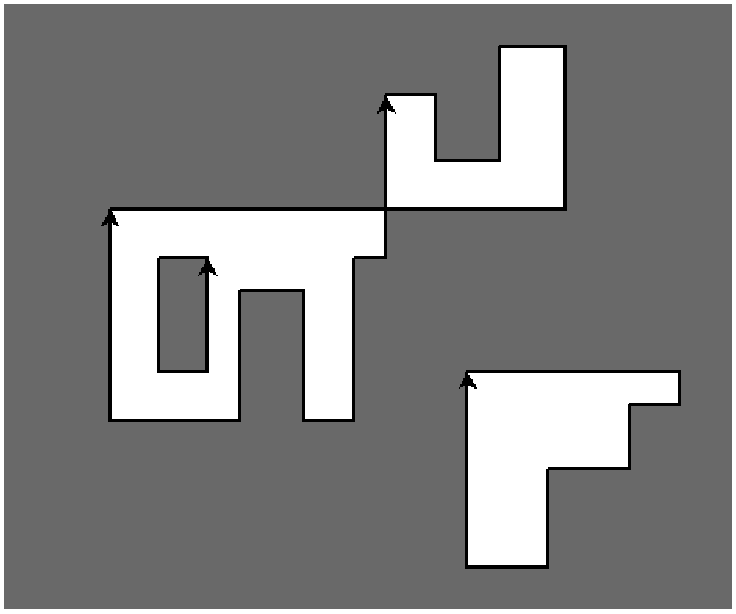

Figure 0.9 The complement of the generalized polygon of Figure 0.1. Observe that the orientation of all boundary cycles is reversed.

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 2 *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 0.10. 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

$$A(P) = |A(\triangle_1)| + |A(\triangle_2)| - |A(\triangle_0)| = A(\triangle_1) + A(\triangle_2) + A(\triangle_0),$$

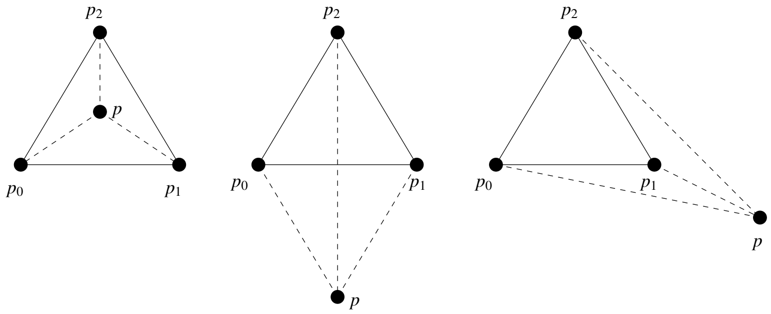

Figure 0.10 Let $\triangle_i = (p, p_i, p_{i+1})$ for $i = 0, 1, 2$, and let $P = (p_0, p_1, p_2)$. Then $A(P) = A(\triangle_0) + A(\triangle_1) + A(\triangle_2)$ in all three cases.

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 [4]. 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

$$A(\triangle) = A(\triangle_i) + A(\triangle_{i+1}) + A(p_{i+2}, p_i, p) = A(\triangle_i) + A(\triangle_{i+1}) - A(p, p_i, p_{i+2}).$$

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();
```

[4] 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.

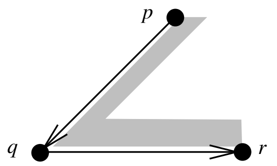

Figure 0.11 The triple (p, q, r) has positive orientation. If q is the lexicographically smallest vertex of the polygon, the region to the left of the polygonal chain is bounded. This conclusion cannot be drawn for an arbitrary vertex.

```
  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 3×3 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 0.11. 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.

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 ℓ 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 ϵ, 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 ℓ.

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());
}
```

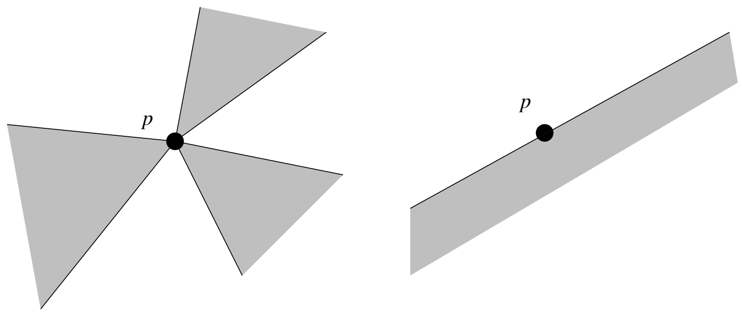

Figure 0.12 The shaded part of the plane belongs to the polygonal region X and p lies in the boundary of X. If p is a vertex of X and U is a sufficiently small neighborhood of p then $U \cap X$ and $U \cap \text{int cpl } X$ are unions of pieces of pie. If p lies in the relative interior of a boundary segment of X then X looks like an open half-plane in the vicinity of p.

0.1.3 *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 $U \cap \text{int cpl } X$ are unions of "pieces of pie" as shown in Figure 0.12.

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 0.13.

Figure 0.14 shows an example.

Lemma 3 *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.

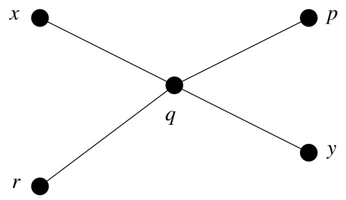

Figure 0.13 The chains $(\ldots, p, q, r, \ldots)$ and $(\ldots, x, q, y, \ldots)$ cross in q.

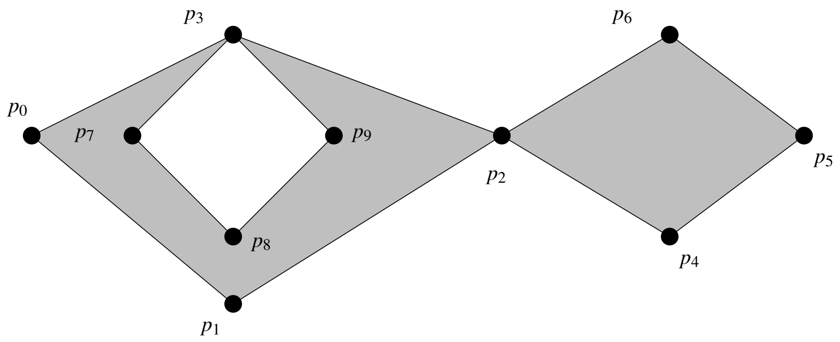

Figure 0.14 The open shaded region consists of two connected sets, one of which is simple. X can be represented by $(p_0, p_1, p_2, p_4, p_5, p_6, p_2, p_3, p_9, p_8, p_7, p_3)$ or by (p_0, p_1, p_2, p_3), (p_2, p_4, p_5, p_6), (p_3, p_9, p_8, p_7), or by $(p_0, p_1, p_2, p_4, p_5, p_6, p_2, p_3)$, (p_9, p_8, p_7, p_3).

Consider next a point p as shown in Figure 0.12. 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. $\square$

The representation of a polygonal set is not unique as Figure 0.14 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 $\mathrm{bR}\, P$ 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 4 *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 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 bR P_i belongs to 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 5 *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 $p \in$ 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

$$\text{bR } P_i \setminus (\text{bR } P_{j_1} \cup \ldots \cup \text{bR } P_{j_l}) \subseteq X$$

and $i < j_1, \ldots, i < j_l$. Thus $p \notin (\text{bR } P_{j_1} \cup \ldots \cup \text{bR } P_{j_l})$ by the definition of i. This shows that $p \in X$. □

0.1.4 *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 0.1.4.

⟨*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 5. 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 **??**. 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.");
```

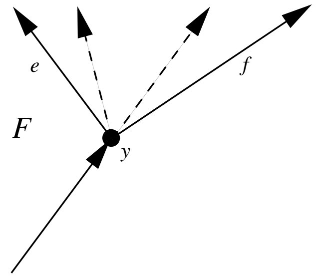

Figure 0.15 The dart f is the nearest adjacency predecessor of e whose status with respect to P_{1-i} is known. The edges between e and f do not belong to the boundary of P_{1-i} and hence F and the face to the left of f have the same status with respect to P_{1-i}.

```
#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 0.15.

⟨*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
```

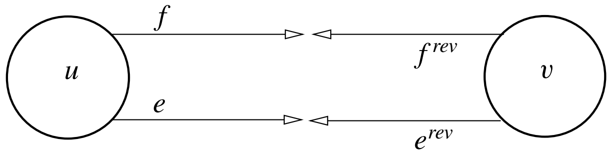

Figure 0.16 The darts e and f come from a segment of P_0 and P_1, respectively. The face cycle (e, f^{rev}) consists of only two darts. We remove e and f^{rev} and make f and e^{rev} reversals of each other.

```
{ 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 0.16. 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.

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&lt;POINT,SEGMENT&gt;& 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;
  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 6 *If D is nested in C then either:*

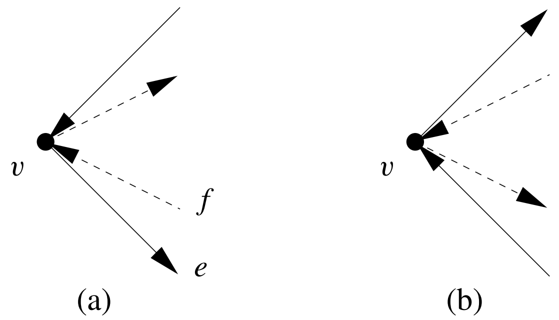

Figure 0.17 The dashed boundary cycle is nested in the solid cycle and both cycles have v as their leading node. In situation (a) the leading dart of the solid cycle is e and the leading dart of the dashed cycle is f^{rev}. In situation (b) the leading dart of the solid cycle is e^{rev} and the leading dart of the dashed cycle is f. In either case the leading dart of the solid cycle has smaller slope.

- *$v(C)$ 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 0.17 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 **??**.

⟨*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 0.17. 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)
{ 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 0.18 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 $O(n + n \log n + fn)$.

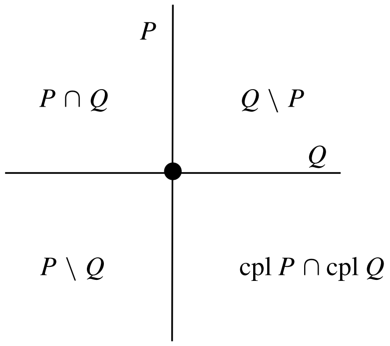

Figure 0.18 The vertex v is an intersection between the boundaries of P and Q. There are four faces incident to v and at least one but not all of them belong to the result of the boolean operation.

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 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 α 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 0.1 and 0.2 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.

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

n	m	P	T	Q	$P \cap Q$	$\|P \cap Q\|$
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 0.1 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.

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 0.2 Execution times with rational kernel: The meaning of the columns is the same as for Table 0.1.

(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 0.1

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.

Bibliography

[Hof89] Christoph M. Hoffmann. *Geometric and solid modeling : an introduction. 2nd pr. 1993*. Series in computer graphics and geometric modeling. Morgan Kaufmann, San Mateo, CA, 1989. 1-55860-067-1.

[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.

[Req80] A. A. G. Requicha. Representations for rigid solids: Theory, methods and systems. *Computing Surveys*, 12(4):437–464, December 1980.

[TR80] R. B. Tilove and A. A. G. Requicha. Closure of boolean operations on geometric entities. *Computer-Aided Design*, 12:219–222, September 1980.

Index