```
1  ////////////////////////////////////////////////////////////////////////
2  //
3  // polyclip.cpp
4  //
5  // computes the intersection (or union) of two non-self-intersecting complex
6  // polygons, with possibly multiple and nested components, even in case of
7  // degenerate intersections (vertex on edge, overlapping edges, etc.)
8  //
9  // compile:
10 //
11 // g++ -std=c++11 polyclip.cpp -o polyclip.exe
12 //
13 // usage:
14 //
15 // polyclip [-union] input1.poly input2.poly output.poly
16 //
17 ////////////////////////////////////////////////////////////////////////
18
19 #include <iostream>
20 #include <cstdlib>
21 #include <cmath>
22 #include <string>
23 #include <vector>
24 #include <set>
25
26 using namespace std;
27
28 #include "point2D.h"
29 #include "polygon.h"
30
31 ////////////////////////////////////////////////////////////////////////
32
33 #define EPSILON  0.000000001            // tolerance
34
35 vector<polygon> PP, QQ;                 // two input polygons
36 vector<polygon> RR;                     // output polygon
37
38 bool UNION = false;                     // global switch for computing union instead of intersect
   ion
39
40 ////////////////////////////////////////////////////////////////////////
41 //
42 // compute intersection of [P1,P2] and [Q1,Q2]
43 //
44 enum IntersectionType {                 // types of intersection (detected in the first phase)
45   NO_INTERSECTION,
46   X_INTERSECTION,
47   T_INTERSECTION_Q,
48   T_INTERSECTION_P,
49   V_INTERSECTION,
50   X_OVERLAP,
51   T_OVERLAP_Q,
52   T_OVERLAP_P,
53   V_OVERLAP
54 };
55
56 IntersectionType intersect(const edge& edgeP, const edge& edgeQ, double& alpha, double& beta) {
57   const point2D& P1 = edgeP.one->p;
58   const point2D& P2 = edgeP.two->p;
59   const point2D& Q1 = edgeQ.one->p;
60   const point2D& Q2 = edgeQ.two->p;
61
62     double AP1 = A(P1,Q1,Q2);
63     double AP2 = A(P2,Q1,Q2);
64
65     if (fabs(AP1-AP2) > EPSILON) {
66         // from here: [P1,P2] and [Q1,Q2] are not parallel
67
68         //
69         // analyse potential intersection
70         //
71
72         double AQ1 = A(Q1,P1,P2);
73         double AQ2 = A(Q2,P1,P2);
74
75         // compute alpha and beta
76         alpha = AP1 / (AP1-AP2);
77         beta  = AQ1 / (AQ1-AQ2);
78
79         // classify alpha
80         bool alpha_is_0 = false;
81         bool alpha_in_0_1 = false;
```

```
 82
 83          if ( (alpha > EPSILON) && (alpha < 1.0-EPSILON) )
 84              alpha_in_0_1 = true;
 85          else
 86              if (fabs(alpha) <= EPSILON)
 87                  alpha_is_0 = true;
 88
 89          // classify beta
 90          bool beta_is_0 = false;
 91          bool beta_in_0_1 = false;
 92
 93          if ( (beta > EPSILON) && (beta < 1.0-EPSILON) )
 94              beta_in_0_1 = true;
 95          else
 96              if (fabs(beta) <= EPSILON)
 97                  beta_is_0 = true;
 98
 99      //
100          // distinguish intersection types
101      //
102
103          if (alpha_in_0_1 && beta_in_0_1)
104              return (X_INTERSECTION);
105
106          if (alpha_is_0 && beta_in_0_1)
107              return (T_INTERSECTION_Q);
108
109          if (beta_is_0 && alpha_in_0_1)
110              return (T_INTERSECTION_P);
111
112          if (alpha_is_0 && beta_is_0)
113              return (V_INTERSECTION);
114      }
115      else
116          if (fabs(AP1) < EPSILON) {
117              // from here: [P1,P2] and [Q1,Q2] are collinear
118
119              //
120              // analyse potential overlap
121              //
122
123              point2D dP = P2-P1;
124              point2D dQ = Q2-Q1;
125              point2D PQ = Q1-P1;
126
127              // compute alpha and beta
128              alpha = (PQ*dP) / (dP*dP);
129              beta = -(PQ*dQ) / (dQ*dQ);
130
131              // classify alpha
132              bool alpha_is_0 = false;
133              bool alpha_in_0_1 = false;
134              bool alpha_not_in_0_1 = false;
135
136              if ( (alpha > EPSILON) && (alpha < 1.0-EPSILON) )
137                  alpha_in_0_1 = true;
138              else
139                  if (fabs(alpha) <= EPSILON)
140                      alpha_is_0 = true;
141                  else
142                      alpha_not_in_0_1 = true;
143
144              // classify beta
145              bool beta_is_0 = false;
146              bool beta_in_0_1 = false;
147              bool beta_not_in_0_1 = false;
148
149              if ( (beta > EPSILON) && (beta < 1.0-EPSILON) )
150                  beta_in_0_1 = true;
151              else
152                  if (fabs(alpha) <= EPSILON)
153                      beta_is_0 = true;
154                  else
155                      beta_not_in_0_1 = true;
156
157        //
158              // distinguish intersection types
159        //
160
161              if (alpha_in_0_1 && beta_in_0_1)
162                  return (X_OVERLAP);
163
```

```
164             if (alpha_not_in_0_1 && beta_in_0_1)
165                 return (T_OVERLAP_Q);
166
167             if (beta_not_in_0_1 && alpha_in_0_1)
168                 return (T_OVERLAP_P);
169
170             if (alpha_is_0 && beta_is_0)
171                 return (V_OVERLAP);
172         }
173
174     return (NO_INTERSECTION);
175 }
176 //
177 ////////////////////////////////////////////////////////////////////////
178
179
180 ////////////////////////////////////////////////////////////////////////
181 //
182 // compute all intersections
183 //
184 void computeIntersections() {
185   cout << "Computing intersections...\n\n";
186
187   point2D I;
188   double alpha;
189   double beta;
190   vertex* I_P;
191   vertex* I_Q;
192
193   int count[9] = {0,0,0,0,0,0,0,0,0};
194
195   //
196   // loop over the source edges of P and Q
197   //
198   for (polygon& P : PP)
199     for (edge edgeP : P.edges(SOURCE))
200       for (polygon& Q : QQ)
201         for (edge edgeQ : Q.edges(SOURCE)) {
202           //
203                 // determine intersection or overlap type
204           //
205           IntersectionType i = intersect(edgeP, edgeQ, alpha, beta);
206           count[i]++;
207
208           vertex* P1 = edgeP.one;
209           vertex* Q1 = edgeQ.one;
210
211           switch(i) {
212                 //
213                 // X-intersection
214                 //
215           case X_INTERSECTION:
216             I = (1.0-alpha)*edgeP.one->p + alpha*edgeP.two->p;
217             I_P = new vertex(I,alpha);
218             I_Q = new vertex(I,beta);
219             insertVertex(I_P, edgeP);
220             insertVertex(I_Q, edgeQ);
221             link(I_P, I_Q);
222             break;
223
224           //
225                 // X-overlap
226                 //
227           case X_OVERLAP:
228                       I_Q = new vertex(P1->p, beta);
229             insertVertex(I_Q, edgeQ);
230             link(P1, I_Q);
231
232                       I_P = new vertex(Q1->p, alpha);
233             insertVertex(I_P, edgeP);
234             link(I_P, Q1);
235             break;
236
237                 //
238                 // T-intersection or T_overlap on Q
239                 //
240           case T_INTERSECTION_Q:
241           case T_OVERLAP_Q:
242                       I_Q = new vertex(P1->p, beta);
243             insertVertex(I_Q, edgeQ);
244             link(P1, I_Q);
245             break;
```

```
246
247                 //
248                 // T-intersection or T-overlap on P
249                 //
250           case T_INTERSECTION_P:
251           case T_OVERLAP_P:
252                     I_P = new vertex(Q1->p, alpha);
253             insertVertex(I_P, edgeP);
254             link(I_P, Q1);
255             break;
256
257                 //
258                 // V-intersection or V-overlap
259                 //
260           case V_INTERSECTION:
261           case V_OVERLAP:
262             link(P1,Q1);
263             break;
264                 }
265         }
266
267   cout << "... " << count[1] << " non-degenerate and ";
268   cout << count[2]+count[3]+count[4]+count[5]+count[6]+count[7]+count[8] << " degenerate intersec
    tions found" << endl;
269   cout << "       (" << count[2]+count[3] << " T-intersections, ";
270   cout << count[4] << " V-intersections,\n        ";
271   cout << count[5] << " X-overlaps, ";
272   cout << count[6]+count[7] << " T-overlaps, ";
273   cout << count[8] << " V-overlaps)" << endl;
274   cout << "... " << count[1]+count[5]+count[3]+count[7] << " vertices added to P" << endl;
275   cout << "... " << count[1]+count[5]+count[2]+count[6] << " vertices added to Q" << endl;
276 }
277 //
278 ////////////////////////////////////////////////////////////////////////
279
280
281 ////////////////////////////////////////////////////////////////////////
282 //
283 // checks the relative position of Q with respect to (P1,P2,P3)
284 //
285 enum RelativePositionType {             // types of relative positions of Q w.r.t. (P1,P2,P3)
286   LEFT,
287   RIGHT,
288   IS_P_m,
289   IS_P_p
290 };
291
292 RelativePositionType oracle(vertex* Q, vertex* P1, vertex* P2, vertex* P3) {
293
294     // is Q linked to P1 ?
295     if ( P1->intersection && (P1->neighbour == Q) )
296         return(IS_P_m);
297
298     // is Q linked to P2 ?
299     if ( P3->intersection && (P3->neighbour == Q) )
300         return(IS_P_p);
301
302     // check relative position of Q with respect to chain (P1,P2,P3)
303     double s1 = A( Q->p, P1->p, P2->p);
304     double s2 = A( Q->p, P2->p, P3->p);
305     double s3 = A(P1->p, P2->p, P3->p);
306
307     if (s3 > 0) {
308         // chain makes a left turn
309         if (s1 > 0 && s2 > 0)
310             return(LEFT);
311         else
312             return(RIGHT);
313     }
314     else {
315         // chain makes a right turn (or is straight)
316         if (s1 < 0 && s2 < 0)
317             return(RIGHT);
318         else
319             return(LEFT);
320     }
321 }
322 //
323 ////////////////////////////////////////////////////////////////////////
324
325
326 ////////////////////////////////////////////////////////////////////////
```

```
327  //
328  // label all intersections
329  //
330  void labelIntersections() {
331    cout << "\nLabelling intersections...\n\n";
332
333      //
334      // 1) initial classification
335    //
336
337    int count[2] = {0,0};
338
339    // loop over intersection vertices of P
340    for (polygon& P : PP)
341      for (vertex* I : P.vertices(INTERSECTION)) {
342
343          // determine local configuration at this intersection vertex
344          vertex* P_m = I->prev;                // P-, predecessor of I on P
345          vertex* P_p = I->next;                // P+, successor of I on P
346          vertex* Q_m = I->neighbour->prev;     // Q-, predecessor of I on Q
347          vertex* Q_p = I->neighbour->next;     // Q+, successor of I on P
348
349          // check positions of Q- and Q+ relative to (P-, I, P+)
350          RelativePositionType Q_m_type = oracle(Q_m, P_m, I, P_p);
351          RelativePositionType Q_p_type = oracle(Q_p, P_m, I, P_p);
352
353          //
354          // check non-overlapping cases
355          //
356          if ((Q_m_type == LEFT  && Q_p_type == RIGHT) ||
357                  (Q_m_type == RIGHT && Q_p_type == LEFT )) {
358              I->label = CROSSING;
359          count[0]++;
360        }
361
362          if ((Q_m_type == LEFT  && Q_p_type == LEFT ) ||
363                  (Q_m_type == RIGHT && Q_p_type == RIGHT)) {
364              I->label = BOUNCING;
365          count[1]++;
366        }
367          //
368          // check overlapping cases
369          //
370          if ( ( (Q_p_type == IS_P_p) && (Q_m_type == RIGHT) ) ||
371               ( (Q_m_type == IS_P_p) && (Q_p_type == RIGHT) ) )
372              I->label = LEFT_ON;
373
374          if ( ( (Q_p_type == IS_P_p) && (Q_m_type == LEFT) ) ||
375               ( (Q_m_type == IS_P_p) && (Q_p_type == LEFT) ) )
376              I->label = RIGHT_ON;
377
378          if ( ( (Q_p_type == IS_P_p) && (Q_m_type == IS_P_m) ) ||
379               ( (Q_m_type == IS_P_p) && (Q_p_type == IS_P_m) ) )
380              I->label = ON_ON;
381
382          if ( ( (Q_m_type == IS_P_m) && (Q_p_type == RIGHT) ) ||
383               ( (Q_p_type == IS_P_m) && (Q_m_type == RIGHT) ) )
384              I->label = ON_LEFT;
385
386          if ( ( (Q_m_type == IS_P_m) && (Q_p_type == LEFT) ) ||
387               ( (Q_p_type == IS_P_m) && (Q_m_type == LEFT) ) )
388              I->label = ON_RIGHT;
389      }
390
391    cout << "... " << count[0] << " crossing and " << count[1] << " bouncing intersection vertices"
     << endl;
392
393      //
394      // 2) classify intersection chains
395      //
396
397    count[0] = count[1] = 0;
398
399    // loop over intersection vertices of P
400    for (polygon& P : PP)
401      for (vertex* I : P.vertices(INTERSECTION)) {
402
403        // start of an intersection chain ?
404          if (I->label == LEFT_ON ||
405                  I->label == RIGHT_ON) {
406
407              // remember status of the first chain vertex and vertex itself
```

```
408             RelativePositionType x;
409             if (I->label == LEFT_ON)
410                 x = LEFT;
411             else
412                 x = RIGHT;
413         vertex* X = I;
414
415             // proceed to end of intersection chain and mark all visited vertices as NONE
416             do {
417                 I->label = NONE;
418                 I = I->next;
419             } while (I->label == ON_ON);
420
421             RelativePositionType y;
422             if (I->label == ON_LEFT)
423                 y = LEFT;
424             else
425                 y = RIGHT;
426
427         // determine type of intersection chain
428         IntersectionLabel chainType;
429         if (x != y) {
430             chainType = DELAYED_CROSSING;
431           count[0]++;
432         }
433         else {
434             chainType = DELAYED_BOUNCING;
435           count[1]++;
436         }
437
438         // mark both ends of an intersection chain with chainType (i.e., as DELAYED_*)
439         X->label = chainType;
440         I->label = chainType;
441         }
442     }
443
444   cout << "... " << count[0] << " delayed crossings and " << count[1] << " delayed bouncings" <<
    endl;
445
446     //
447     // 3) copy labels from P to Q
448     //
449
450   // loop over intersection vertices of P
451   for (polygon& P : PP)
452     for (vertex* I : P.vertices(INTERSECTION))
453       I->neighbour->label = I->label;
454
455   //
456   // 3.5) check for special cases
457   //
458
459   set<polygon*> noIntersection[2];
460   set<polygon*> identical[2];
461
462   count[0] = count[1] = 0;
463
464   for (int i=0; i<2; ++i) {
465     vector<polygon>* P_or_Q = &PP;      // if i=0, then do it for P w.r.t. Q
466     vector<polygon>* Q_or_P = &QQ;
467
468     if (i==1) {                         // if i=1, then do it for Q w.r.t. P
469       P_or_Q = &QQ;
470       Q_or_P = &PP;
471     }
472
473     // loop over all components of P (or Q)
474     for (polygon& P : *P_or_Q)
475       if (P.noCrossingVertex(UNION)) {
476         //
477         // P_ has no crossing vertex (but may have bounces or delayed bounces, except for UNION),
478         // hence it does not intersect with Q_or_P
479         //
480         noIntersection[i].insert(&P);   // remember component, and ignore it later in step 4
481
482         // is P identical to some component of and Q_or_P?
483         if (P.allOnOn()) {
484           identical[i].insert(&P);      // -> remember for further processing below
485         }
486         else {
487           // is P inside Q_or_P?
488           bool isInside = false;
```

```
489           point2D p = P.getNonIntersectionPoint();
490           for (polygon& Q : *Q_or_P)
491             if ( Q.pointInPoly(p) )
492               isInside = !isInside;
493           if (isInside ^ UNION) {
494             RR.push_back(P);             // -> add P to the result
495             count[0]++;
496           }
497         }
498       }
499   }
500
501   // handle components of P that are identical to some component of Q
502   for (polygon* P : identical[0]) {
503     // is P a hole?
504     bool P_isHole = false;
505     for (polygon& P_ : PP)
506       if ( ( P_.root != P->root ) && (P_.pointInPoly(P->root->p)) )
507         P_isHole = !P_isHole;
508
509     for (polygon* Q : identical[1])
510       for (vertex* V : Q->vertices(ALL))
511         if (V == P->root->neighbour) {  // found Q that matches P
512           // is Q a hole?
513           bool Q_isHole = false;
514           for (polygon& Q_ : QQ)
515             if ( ( Q_.root != Q->root ) && (Q_.pointInPoly(Q->root->p)) )
516               Q_isHole = !Q_isHole;
517
518           // if P and Q are both holes or both are not holes
519           if (P_isHole == Q_isHole) {
520             RR.push_back(*P);            // -> add P to the result
521             count[1]++;
522           }
523           goto next_P;
524         }
525     next_P: ;
526   }
527
528   cout << "... " << count[0] << " interior and " << count[1] << " identical components added to r
    esult\n";
529
530     //
531     // 4) set entry/exit flags
532     //
533
534   set<vertex*> split[2];                 // split vertex candidates for P and Q
535   set<vertex*> crossing[2];              // CROSSING vertex candidates for P and Q
536
537   for (int i=0; i<2; ++i) {
538     vector<polygon>* P_or_Q = &PP;       // if i=0, then do it for P w.r.t. Q
539     vector<polygon>* Q_or_P = &QQ;
540
541     if (i==1) {                          // if i=1, then do it for Q w.r.t. P
542       P_or_Q = &QQ;
543       Q_or_P = &PP;
544     }
545
546     // loop over all components of P (or Q)
547     for (polygon& P : *P_or_Q) {
548
549       // ignore P if it does not intersect with Q_or_P (detected in step 3.5 above)
550       if(noIntersection[i].find(&P) != noIntersection[i].end())
551         continue;
552
553       // start at a non-intersection vertex of P
554       vertex* V = P.getNonIntersectionVertex();
555
556       // check if it is inside or outside Q (or P)
557       // and set ENTRY/EXIT status accordingly
558       EntryExitLabel status = ENTRY;
559       for (polygon& Q : *Q_or_P)
560         if (Q.pointInPoly(V->p))
561           toggle(status);
562
563       //
564       // starting at V, loop over those vertices of P, that are either
565       // a crossing intersection or marked as ends of an intersection chain
566       //
567       bool first_chain_vertex = true;     // needed for dealing with crossing chains
568
569       for (vertex* I : P.vertices(INTERSECTION, V)) {
```

```
570           //
571           // in the case of normal crossings, we...
572           //
573           if (I->label == CROSSING) {
574             // mark vertex with current ENTRY/EXIT status
575                   I->enex = status;
576             // toggle status from ENTRY to EXIT or vice versa
577                   toggle(status);
578           }
579
580           //
581           // identify split vertex candidates (INTERIOR bouncing vertices)
582           //
583           if ( (I->label == BOUNCING) && ((status == EXIT) ^ UNION) )
584             split[i].insert(I);
585
586           //
587           // in the case of a delayed crossing chain, we
588           // mark both end points of the chain with the current ENTRY/EXIT status,
589           // toggling the status only at the end last chain vertex,
590           // and, in case of a delayed EXIT  crossing, the first vertex
591           //  or, in case of a delayed ENTRY crossing, the last  vertex,
592           // of the chain as CROSSING
593           //
594           if (I->label == DELAYED_CROSSING) {
595             // mark vertex with current ENTRY/EXIT status
596             I->enex = status;
597
598             if (first_chain_vertex) {       // are we at the first vertex of a delayed crossing cha
    in?
599               if ((status == EXIT) ^ UNION)
600                 I->label = CROSSING;        // mark first vertex as CROSSING
601               first_chain_vertex = false;
602             }
603             else {                          // here we are at the last vertex of a delayed crossing
     chain
604               if ((status == ENTRY) ^ UNION)
605                 I->label = CROSSING;        // mark last vertex as CROSSING
606               first_chain_vertex = true;
607
608               // toggle status from ENTRY to EXIT or vice versa (only for last chain vertex)
609               toggle(status);
610             }
611           }
612
613           //
614           // in the case of a delayed bouncing chain, we
615           // mark both end points of the chain with the current ENTRY/EXIT status
616           // toggling the status at both end points of the chain,
617           // and, in case of a delayed INTERIOR bouncing, both end points
618           // of the chain as CROSSING candidates
619           //
620           if (I->label == DELAYED_BOUNCING) {
621             // mark vertex with current ENTRY/EXIT status
622             I->enex = status;
623
624             if (first_chain_vertex) {       // are we at the first vertex of a delayed crossing cha
    in?
625               if ((status == EXIT) ^ UNION)
626                 crossing[i].insert(I);      // mark first EXIT vertex as CROSSING candidate
627               first_chain_vertex = false;
628             }
629             else {                          // here we are at the last vertex of a delayed crossing
     chain
630               if ((status == ENTRY) ^ UNION)
631                 crossing[i].insert(I);      // mark last ENTRY vertex as CROSSING candidate
632               first_chain_vertex = true;
633
634             }
635             // toggle status from ENTRY to EXIT or vice versa (for first AND last chain vertex)
636             toggle(status);
637           }
638           }
639       }
640     }
641
642       //
643       // 5) handle split vertex pairs
644       //
645
646     count[0] = 0;
647
```

```
648     // loop over P's split candidates
649     for (vertex* I_P : split[0]) {
650       vertex* I_Q = I_P->neighbour;
651
652       // check if the neighbour on Q is also a split candidate
653       if (split[1].find(I_Q) != split[1].end()) {
654         count[0]++;
655         //
656         // split vertex pair
657         //
658
659         // duplicate vertices
660               vertex* V_P = new vertex(I_P->p);
661               vertex* V_Q = new vertex(I_Q->p);
662
663         // compute areas to compare local orientation
664         double sP = A( I_P->prev->p, I_P->p, I_P->next->p);
665         double sQ = A( I_Q->prev->p, I_Q->p, I_Q->next->p);
666
667         // link vertices correctly
668         if (sP*sQ > 0) {                  // same local orientation
669           link(I_P, V_Q);
670           link(I_Q, V_P);
671         }
672         else {                            // different local orientation
673           link(V_P, V_Q);
674         }
675
676         // add duplicate vertices to P and Q
677         insertVertex(V_P, I_P);
678         insertVertex(V_Q, I_Q);
679
680         // mark all four vertices correctly
681         if (!UNION) {
682           I_P->enex = EXIT;
683           V_P->enex = ENTRY;
684           I_Q->enex = EXIT;
685           V_Q->enex = ENTRY;
686         }
687         else {
688           I_P->enex = ENTRY;
689           V_P->enex = EXIT;
690           I_Q->enex = ENTRY;
691           V_Q->enex = EXIT;
692         }
693
694         I_P->label = CROSSING;
695         V_P->label = CROSSING;
696         I_Q->label = CROSSING;
697         V_Q->label = CROSSING;
698       }
699     }
700
701     cout << "... " << count[0] << " bouncing vertex pairs split" << endl;
702
703       //
704       // 6) handle CROSSING vertex candidates
705       //
706
707     // loop over P's CROSSING candidates
708     for (vertex* I_P : crossing[0]) {
709       vertex* I_Q = I_P->neighbour;
710
711       // check if the neighbour on Q is also a CROSSING candidate
712       if (crossing[1].find(I_Q) != crossing[1].end()) {
713         //
714         // mark CROSSING candidate pair as such
715         //
716         I_P->label = CROSSING;
717         I_Q->label = CROSSING;
718       }
719     }
720   }
721   //
722   ////////////////////////////////////////////////////////////////////////
723
724   ////////////////////////////////////////////////////////////////////////
725   //
726   // create the result polygon
727   //
728   void createResult() {
729     cout << "\nCreating result...\n";
```

```
730   //
731   // for all crossing vertices
732   //
733   // NOTE: all crossing vertices that are visited while contructing a
734   //       component of the result polygon are marked as "not intersection",
735   //       so that they cannot serve as start vertex of another component
736   //
737
738   for (polygon& P : PP)
739     for (vertex* I : P.vertices(CROSSING_INTERSECTION)) {
740       polygon R;                          // result polygon component
741
742       vertex* V = I;                      // start traversal at I
743       V->intersection = false;            // mark visited vertices
744
745       do {
746         EntryExitLabel status = V->enex;
747         toggle(status);
748             do {                              // traverse P (or Q) and add vertices to R, until..
.
749           if ((status == EXIT) ^ UNION)
750             V = V->next;                  // move forward  from an ENTRY vertex to the next EXIT
 vertex
751           else
752             V = V->prev;                  // move backward from an EXIT  vertex to the next ENTRY
 vertex
753           V->intersection = false;        // mark visited vertices
754
755                 // add vertex to result polygon
756           R.newVertex(V->p);
757         } while ( !(V->enex == status)    // ... we arrive at a vertex with opposite entry/exit f
lag, or
758                   && (V != I) );          // at the initial vertex I
759
760         if (V != I) {
761           V = V->neighbour;               // switch from P to Q or vice versa
762           V->intersection = false;        // mark visited vertices
763         }
764       } while (V != I);                   // the result polygon component is complete,
765                                           // if we are back to the initial vertex I
766       RR.push_back(R);
767     }
768 }
769 //
770 ////////////////////////////////////////////////////////////////////////
771
772 ////////////////////////////////////////////////////////////////////////
773 //
774 // cleans up the result polygon by removing unnecessary collinear vertices
775 //
776 void cleanUpResult() {
777   cout << "\nPost-processing...\n\n";
778   int count = 0;
779   for (polygon& R : RR) {
780     while ( (R.root != NULL) && (fabs(A(R.root->prev->p,R.root->p,R.root->next->p)) < EPSILON) )
{
781       R.removeVertex(R.root);
782       count++;
783     }
784     if (R.root != NULL)
785       for (vertex* V : R.vertices(ALL))
786         if (fabs(A(V->prev->p,V->p,V->next->p)) < EPSILON) {
787           R.removeVertex(V);
788           count++;
789         }
790   }
791   cout << "... " << count << " vertices removed\n\n";
792 }
793 //
794 ////////////////////////////////////////////////////////////////////////
795
796 ////////////////////////////////////////////////////////////////////////
797 //
798 // prints statistics (#components, #vertices) for a complex polygon
799 //
800 void printInfo(vector<polygon>& PP) {
801   int sum = 0;
802   cout << "has "<< PP.size() << " component";
803   if (PP.size() > 1)
804     cout << "s";
805   cout << " with ";
806   for (int i=0; i<PP.size(); i++) {
```

```
807       int count = 0;
808       for (vertex* V : PP[i].vertices(ALL))
809         count++;
810       cout << count;
811       if (i<PP.size()-1)
812         cout << " + ";
813       sum += count;
814     }
815     if (PP.size() > 1)
816       cout << " = " << sum;
817     cout << " vertices\n\n";
818   }
819   //
820   ////////////////////////////////////////////////////////////////////////
821
822   ////////////////////////////////////////////////////////////////////////
823   //
824   // load complex polygon P from file "s" and print statistics
825   //
826   void loadPolygon(vector<polygon>& PP, string s) {
827     ifstream from(s);
828     do {
829       polygon P;
830       from >> P;
831       if (!from.fail())
832         PP.push_back(P);
833     } while (!from.eof());
834     from.close();
835     printInfo(PP);
836   }
837   //
838   ////////////////////////////////////////////////////////////////////////
839
840   ////////////////////////////////////////////////////////////////////////
841   //
842   // save complex polygon P to file "s" and print statistics
843   //
844   void savePolygon(vector<polygon>& PP, string s) {
845     ofstream to(s);
846     for (polygon& P : PP)
847       to << P;
848     to.close();
849     printInfo(PP);
850   }
851   //
852   ////////////////////////////////////////////////////////////////////////
853
854
855   int main(int argc, char* argv[])
856   {
857     // check input parameters
858     if (argc < 4) {
859       cout << "insufficient number of parameters" << endl;
860       exit(0);
861     }
862
863     int argn = 1;
864     if (string(argv[1]) == "-union") {
865       cout << "\n!!! computing UNION instead of INTERSECTION !!!\n";
866       UNION = true;
867       argn++;
868     }
869
870     // read input polygons
871     cout << "\nP "; loadPolygon(PP,string(argv[argn++]));
872     cout <<   "Q "; loadPolygon(QQ,string(argv[argn++]));
873
874     // phase 1
875     computeIntersections();
876
877     // phase 2
878     labelIntersections();
879
880     // phase 3
881     createResult();
882
883     // post-processing
884     cleanUpResult();
885
886     // write output polygon
887     cout << "R "; savePolygon(RR,string(argv[argn]));
888   }
```