```
1  ////////////////////////////////////////////////////////////////////////
2  //
3  // polygon.h
4  //
5  // contains everything that the polyclip algorithm needs for
6  // handling polygons
7  //
8  ////////////////////////////////////////////////////////////////////////
9
10 #include <fstream>
11 #include <iostream>
12
13 using namespace std;
14
15 ////////////////////////////////////////////////////////////////////////
16 //
17 // a simple class to handle polygon vertices
18 //
19 ////////////////////////////////////////////////////////////////////////
20
21 enum IntersectionLabel {      // for the classification of intersection vertices in the second ph
   ase
22   NONE,
23   CROSSING,
24   BOUNCING,
25   LEFT_ON,
26   RIGHT_ON,
27   ON_ON,
28   ON_LEFT,
29   ON_RIGHT,
30   DELAYED_CROSSING,
31   DELAYED_BOUNCING
32 };
33
34 enum EntryExitLabel {         // for marking intersection vertices as "entry" or "exit"
35   EXIT,
36   ENTRY,
37   NEITHER
38 };
39
40 class vertex
41 {
42 public:
43   point2D p;                  // coordinates of the vertex
44   vertex* prev;               // pointer to previous vertex
45   vertex* next;               // pointer to next vertex
46   vertex* neighbour;          // pointer to neighbouring vertex for intersection vertices
47   bool source;                // to mark source vertices of the polygon
48   bool intersection;          // to mark intersection vertices
49   double alpha;               // to describe relative edge position of an intersection vertex
50   IntersectionLabel label;    // type of intersection vertex
51   EntryExitLabel enex;        // entry/exit "flag"
52
53   //
54   // default-constructor for generating a vertex with coordinates (0,0)
55   //
56     vertex() :
57     p(), prev(NULL), next(NULL), neighbour(NULL),
58     intersection(false), source(false),
59     alpha(-1.0), label(NONE), enex(NEITHER)
60   {};
61
62   //
63   // constructor for generating a vertex with explicit coordinates
64   //
65   vertex(double x, double y) :
66     p(x,y), prev(NULL), next(NULL), neighbour(NULL),
67     intersection(false), source(false),
68     alpha(-1.0), label(NONE), enex(NEITHER)
69   {};
70
71   //
72   // constructor for generating a vertex from a 2D point;
73   // if alpha is given (and then in [0,1], thus > -1.0),
74   // then the vertex will be marked as an intersection vertex
75   //
76   vertex(point2D& q, double alpha = -1.0) :
77     p(q), prev(NULL), next(NULL), neighbour(NULL),
78     source(false),
79     alpha(alpha), label(NONE), enex(NEITHER)
80   {
81     intersection = (alpha > -1.0);
```

```
  };
};

//
// link vertex P to vertex Q, using the neighbour pointer
// and mark both vertices as intersection vertices
//
void link(vertex* P, vertex* Q) {
  P->neighbour = Q;
  Q->neighbour = P;
  P->intersection = true;
  Q->intersection = true;
}

//
// insert a vertex between two existing vertices
// if alpha is given (and then in [0,1], thus > -1.0),
// then the vertex will be inserted, based on alpha
//
void insertVertex (vertex* V, vertex* curr, double alpha = -1.0) {
  if (alpha > -1.0)
        do {
            curr = curr->next;
        } while (!curr->source && curr->alpha < alpha);
  else
    curr = curr->next;

    curr->prev->next = V;
    V->prev = curr->prev;
    V->next = curr;
    curr->prev = V;
}

////////////////////////////////////////////////////////////////////////
//
// a simple class to handle polygon edges
//
////////////////////////////////////////////////////////////////////////

class edge
{
public:
  vertex* one;
  vertex* two;

  //
  // constructor for generating an edge from vertex P to vertex Q
  //
  edge (vertex* P, vertex* Q) : one(P), two(Q) { };
};

//
// insert an intersection vertex between the endpoints of edge E
//
void insertVertex (vertex* V, edge& E) {
  insertVertex(V, E.one, V->alpha);
}

////////////////////////////////////////////////////////////////////////
//
// iterators to loop over vertices and edges, with the option to
// restrict to source vertices or edges and to intersection vertices
//
////////////////////////////////////////////////////////////////////////

enum IteratorType {
  SOURCE,
  INTERSECTION,
  CROSSING_INTERSECTION,
  ALL
};

class vertexIterator
{
private:
  class iterator {
  public:
    iterator(vertex* root, IteratorType IterType) :
      root(root), V(NULL), iterType(IterType)
    {
      if (root == NULL)
        return;
```

```
164
165       if (nextVertex() == NULL)          // no (source/intersection) vertex found
166         root = V = NULL;                 // -> mark iterator as "end"
167     }
168
169     const iterator& operator++() {
170       nextVertex();
171       return *this;
172     }
173
174     vertex* operator*() {
175       return V;
176     }
177
178     bool operator!=(const iterator& other) const {
179       return (root != other.root) || (V != other.V);
180     }
181
182   private:
183     vertex* root;
184     vertex* V;
185     IteratorType iterType;
186
187     //
188     // find the next vertex
189     // if iterType is ALL, then it is just the next vertex
190     // if iterType is SOURCE, then it is the next source vertex
191     // if iterType is INTERSECTION, then it is the next intersection vertex
192     // if iterType is CROSSING_INTERSECTION, then it is the next intersection vertex with CROSSIN
    G label
193     //
194     vertex* nextVertex() {
195       bool nextFound = false;
196
197       if (V == NULL) {                   // find first (source/intersection) vertex
198         V = root;
199         switch(iterType) {
200         case ALL:
201           nextFound = true;
202           break;
203         case SOURCE:
204           if (V->source)
205             nextFound = true;
206           break;
207         case INTERSECTION:
208           if (V->intersection)
209             nextFound = true;
210           break;
211         case CROSSING_INTERSECTION:
212           if (V->intersection && (V->label == CROSSING))
213             nextFound = true;
214           break;
215         }
216       }
217
218       while (!nextFound) {               // find next (source/intersection) vertex
219         switch(iterType) {
220         case ALL:
221           V = V->next;
222           break;
223         case SOURCE:
224           do {
225             V = V->next;
226           } while (!V->source && V != root);
227           break;
228         case INTERSECTION:
229           do {
230             V = V->next;
231           } while (!V->intersection && V != root);
232           break;
233         case CROSSING_INTERSECTION:
234           do {
235             V = V->next;
236           } while ( ( !V->intersection || (V->label != CROSSING) ) && V != root);
237           break;
238         }
239
240         if (V == root) {                 // back at the root vertex?
241           root = V = NULL;               // -> mark iterator as "end"
242           return(V);
243         }
244
```

```
245           switch(iterType) {
246           case ALL:
247             nextFound = true;
248             break;
249           case SOURCE:
250             if (V->source)
251               nextFound = true;
252             break;
253           case INTERSECTION:
254             if (V->intersection)
255               nextFound = true;
256             break;
257           case CROSSING_INTERSECTION:
258             if (V->intersection && (V->label == CROSSING))
259               nextFound = true;
260             break;
261           }
262         }
263         return(V);
264       }
265     };
266
267   public:
268     vertexIterator() : root(NULL) {};
269
270     iterator begin() { return iterator(root, iterType); }
271     iterator end()   { return iterator(NULL, iterType); }
272
273     vertex* root;
274     IteratorType iterType;
275   };
276
277
278   class edgeIterator
279   {
280   private:
281     class iterator {
282     public:
283       iterator(vertex* root, IteratorType IterType) :
284         root(root), one(NULL), two(NULL), iterType(IterType)
285       {
286         if (root == NULL)
287           return;
288
289         if (nextEdge() == NULL)           // no source edge found
290           root = one = two = NULL;        // -> mark iterator as "end"
291       }
292
293       const iterator& operator++() { nextEdge(); return *this; }
294
295       edge operator*() {
296         return edge(one,two);
297       }
298
299       bool operator!=(const iterator& other) const {
300         return (root != other.root) || (one != other.one) || (two != other.two);
301       }
302
303     private:
304       vertex* root;
305       vertex* one;
306       vertex* two;
307       IteratorType iterType;
308
309       //
310       // find the next vertex, starting at curr
311       // if iterType is ALL, then it is just the next vertex
312       // if iterType is SOURCE, then it is the next source vertex
313       //
314       vertex* nextVertex(vertex* curr) {
315         if (curr == NULL)
316           return(NULL);
317
318         switch(iterType) {
319         case ALL:
320           curr = curr->next;
321           break;
322
323         case SOURCE:
324           do {
325             curr = curr->next;
326           } while (!curr->source);
```

```
327           break;
328         }
329
330         return(curr);
331       }
332
333       //
334       // find the next edge
335       //
336       vertex* nextEdge() {
337         if (root == NULL)                  // empty polygon?
338           return (NULL);
339
340         if (one == NULL) {                 // find one (source) vertex
341           one = root;                      // note: root is always a (source) vertex
342           two = nextVertex(one);
343           if (two == one)                  // just one (source) vertex
344             return(NULL);                  // -> no (source) edges
345           return(one);
346         }
347
348         if (two == root) {                 // back at the root vertex?
349           root = one = two = NULL;         // -> mark iterator as "end"
350           return(NULL);
351         }
352
353         one = two;
354         two = nextVertex(one);
355
356         return (one);
357       }
358     };
359
360   public:
361     edgeIterator() : root(root) {};
362
363     iterator begin() { return iterator(root, iterType); }
364     iterator end()   { return iterator(NULL, iterType); }
365
366     vertex* root;
367     IteratorType iterType;
368   };
369
370
371   ////////////////////////////////////////////////////////////////////////
372   //
373   // a simple polygon class with essential functionality
374   //
375   ////////////////////////////////////////////////////////////////////////
376
377   class polygon
378   {
379   public:
380     //
381     // default-constructor for generating an empty polygon
382     //
383     polygon() : root(NULL) {}
384
385     //
386     // create a new vertex and append it to the polygon
387     //
388     void newVertex(point2D& v, bool source = false) {
389       vertex* V = new vertex(v);
390       V->source = source;
391
392       if (root == NULL) {
393         // add vertex as the very first vertex of the polygon
394         V->next = V;
395         V->prev = V;
396         root = V;
397       }
398       else {
399         // add vertex at the end
400         V->prev = root->prev;
401         V->next = root;
402         root->prev->next = V;
403         root->prev = V;
404       }
405     }
406
407     //
408     // removes the vertex V from the polygon
```

```
409     //
410     void removeVertex (vertex* V) {
411       if (root == V) {
412         root = V->next;
413         if (root->next == root)
414           root = NULL;
415       }
416       V->prev->next = V->next;
417       V->next->prev = V->prev;
418       delete(V);
419     }
420
421     //
422     // check, if the point R lies inside the polygon or not
423     // cf. Algorithm 6 in Hormann & Agathos [2001]
424     //
425     bool pointInPoly(point2D& R) {
426       int w = 0;
427       for (edge E : edges(ALL)) {
428               point2D& P0 = E.one->p;
429               point2D& P1 = E.two->p;
430
431               if ( (P0.y < R.y) != (P1.y < R.y) )
432                   if (P0.x >= R.x) {
433                       if (P1.x > R.x)
434                           w = w + 2 * (P1.y > P0.y) - 1;
435                       else
436                           if ( (A(P0,P1,R) > 0) == (P1.y > P0.y) )
437                               w = w + 2 * (P1.y > P0.y) - 1;
438                   }
439                   else
440                       if (P1.x > R.x)
441                           if ( (A(P0,P1,R) > 0) == (P1.y > P0.y) )
442                               w = w + 2 * (P1.y > P0.y) - 1;
443           }
444
445       return ( (w % 2) != 0 );
446     }
447
448     //
449     // check, if all vertices have the ON_ON label
450     //
451     bool allOnOn() {
452       for (vertex* V : vertices(ALL))
453         if (V->label != ON_ON)
454           return(false);
455       return(true);
456     }
457
458     //
459     // check, if the polygon does not contain any crossing intersection vertex
460     // or crossing intersection chain or (if we want to compute the union instead
461     // of the intersection) a bouncing vertex or a bouncing intersection chain
462     //
463     bool noCrossingVertex(bool union_case = false) {
464       for (vertex* V : vertices(ALL))
465         if (V->intersection) {
466           if ( (V->label == CROSSING) || (V->label == DELAYED_CROSSING) )
467             return(false);
468
469           if (union_case && ( (V->label == BOUNCING) || (V->label == DELAYED_BOUNCING) ) )
470             return(false);
471         }
472       return(true);
473     }
474
475     //
476     // return a non-intersection point
477     //
478     point2D getNonIntersectionPoint() {
479       for (vertex* V : vertices(ALL))
480         if (!V->intersection)
481           return(V->p);
482
483       // no non-intersection vertex found -> find suitable edge midpoint
484       for (vertex* V : vertices(ALL))
485         // make sure that edge from V to V->next is not collinear with other polygon
486         if ( (V->next->neighbour != V->neighbour->prev) && (V->next->neighbour != V->neighbour->nex
    t) )
487           // return edge midpoint
488           return(0.5*(V->p + V->next->p));
489     }
```

```
490
491     //
492     // return and insert a non-intersection vertex
493     //
494     vertex* getNonIntersectionVertex() {
495       for (vertex* V : vertices(ALL))
496         if (!V->intersection)
497           return(V);
498
499       // no non-intersection vertex found -> generate and return temporary vertex
500       for (vertex* V : vertices(ALL))
501         // make sure that edge from V to V->next is not collinear with other polygon
502         if ( (V->next->neighbour != V->neighbour->prev) && (V->next->neighbour != V->neighbour->nex
    t) ) {
503           // add edge midpoint as temporary vertex
504           point2D p = 0.5*(V->p + V->next->p);
505           vertex* T = new vertex(p);
506           insertVertex(T, V);
507           return(T);
508         }
509       return(NULL);
510     }
511
512     //
513     // return iterator to loop over (source/intersection) vertices,
514     // starting at first (by default, it starts at root)
515     //
516     vertexIterator& vertices(IteratorType iterType, vertex* first = NULL) {
517       vertexIter.iterType = iterType;
518
519       if (first == NULL)
520         vertexIter.root = root;
521       else
522         vertexIter.root = first;
523
524       return vertexIter;
525     }
526
527     //
528     // return iterator to loop over (source) edges
529     //
530     edgeIterator& edges(IteratorType iterType) {
531       edgeIter.iterType = iterType;
532       edgeIter.root = root;
533       return edgeIter;
534     }
535
536     vertex* root;               // root vertex of the polygon
537
538   protected:
539     vertexIterator vertexIter;  // vertex iterator
540     edgeIterator edgeIter;      // edge iterator
541   };
542
543   //
544   // read polygon from a stream
545   //
546   inline istream& operator>>(istream& s, polygon& P) {
547     point2D v;
548     char c;
549     bool readOn = true;
550
551     do {
552       s >> v >> c;
553       if (s.fail())
554           readOn = false;
555       else {
556         P.newVertex(v, true);
557
558         if (c == ';')
559           readOn = false;
560       }
561     } while (readOn && !s.eof());
562
563     return (s);
564   }
565
566   //
567   // write polygon to a stream
568   //
569   inline ostream& operator<<(ostream& s, polygon& P) {
570     if (P.root == NULL)
```

```
571       return (s);
572     for (vertex* V : P.vertices(ALL)) {
573       s << V->p;
574       if (V != P.root->prev)
575         s << "," << endl;
576     }
577     return (s << ";\n");
578   }
579
580   //
581   // toggle status from ENTRY to EXIT or vice versa
582   //
583   void toggle(EntryExitLabel& status) {
584     if (status == ENTRY) {
585       status = EXIT;
586       return;
587     }
588     if (status == EXIT) {
589       status = ENTRY;
590       return;
591     }
592   }
```