Guidelines for using the code

In this document you will find the guidelines for using the code on a Linux system.

1 Comparative of algorithms for computing Boolean operations

If you want to compare the execution times of the different implemented algorithms do the following:

```
$ make
$ ./clip file_subject file_clipping file_result [I|U|D|X]
```

where:

- `file_subject` is the file containing the subject polygon
- `file_clipping` is the file containing the clipping polygon
- `file_result` will be used to save the result of the operation
- the last parameter is optional and indicates the kind of Boolean operation:
  - I stands for Intersection (the default operation)
  - U stands for Union
  - D stands for Difference between the subject and clipping polygons
  - X stands for eXclusive OR (Symmetric difference) between the subject and clipping polygons

2 Format of polygon files

You can find polygons in the directory *polygons*. In the directory *polygons/random* you will find random non-self-intersecting polygons up to 10000 vertices. We suggest to try:

```
$ ./clip polygons/random/p10000-0 polygons/random/p1000-0 /dev/null
```

You can also create your own polygons. A file polygon is an ASCII file with the following structure:

```
<number of contours>
<number of vertices of first contour>
<vertex-list of first contour>
<number of vertices of second contour> <level of second contour>
<vertex-list of second contour>
...
<contourId>: <firstHole_contourId> <secondHole_contourId> ...
...
```

Next, you can see the content of the file *polygonwithtwocontours*, that belongs to the directory *polygons/samples*:

```
2
3
0.1 0.1
0.3 0.1
0.2 0.3
3
0.6 0.1
0.8 0.1
0.7 0.3
```

Next, you can see the content of the file *polygonwithholes*, that belongs to the directory *polygons/samples*:

```
3
3
-0.15 -0.15
0.45 -0.15
0.15 0.45
3
-0.05 -0.05
0.15 0.35
0.35 -0.05
3
0.05 0.05
0.25 0.05
0.15 0.25
0: 1
1: 2
```

3 Admitted input polygons

The following features are allowed in input polygons:

- Contours can be described in clockwise or counterclockwise order.
- Holes.
- A vertex of a polygon can touch (in a point) a vertex or edge of the same polygon.
- Self-intersecting polygons.

The following features are not allowed in input polygons:

- Overlapping edges (the intersection of two edges of the same polygon can be a point, but cannot be a segment).

4 Rendering the result of Boolean operations

If you want to render the result of a Boolean operation you should have installed glut and opengl and type:

```
$ make -f makefile.guiglut
$ ./guiglut file_subject file_clipping G|V|M [I|U|D|X]
```

where,

- `file_subject` is the file containing the subject polygon
- `file_clipping` is the file containing the clipping polygon
- the third parameter indicates the kind of algorithm used to compute the Boolean operation
  - G stands for Greiner-Hormann
  - V stands for Alan Murta's implementation of Vatti's algorithm
  - M stands for our algorithm
- the last parameter is optional, and indicates the kind of Boolean operation
  - I stands for Intersection (the default operation)
  - U stands for Union
  - D stands for set the Difference subject-clipping polygons
  - X stands for eXclusive OR (Symmetric difference) between the subject and clipping polygons

Note that when you run the program you can see the subject and clipping polygons. The subject polygon is shaded green, while the clipping polygon is rendered as translucent red. You can type the following keys:

- R: for toggling on/off rendering the result of the operation

- S: for toggling on/off rendering the subject polygon.
- C: for toggling on/off rendering the clipping polygon.
- W: for switching rendering the polygons from filled to wireframe.
- z: for zoom out.
- Z: for zoom in.
- Arrow keys: to translate up/down/left/right the polygons.

5 Implementation files

You can take a look to the implementation of the algorithms. Greiner and Hormann's algorithm is implemented in the *greiner.h* and *greiner.cpp* files. The implementation does not support the degenerate case of a vertex lying on an edge of the other polygon. We have not neither extended the algorithm to compute the union and difference operation on polygons with several contours (the intersection can be computed).

We have included the Alan Murta's implementation of Vatti's algorithm (files *gpc.h* and *gpc.cpp*). Please, note that this code has its own license (included in the file *gpc.cpp*). Alan Murta's implementation is very robust, it includes extensions to the algorithm to deal with edges parallel to the sweep line, and to deal with overlapping edges.

Finally, the implementation of our algorithm can be found in the files *martinez.h* and *martinez.cpp*.

6 Polygons with several components and with holes

Suppose two polygons such that both polygons have one single contour and neither of the polygons has a hole. The result of a Boolean operation on the two polygons can be a polygon with several contours and/or with holes. For these polygons, the algorithm described in the paper "A new algorithm for computing Boolean operations on polygons" computes correctly the different contours and holes of the Boolean operation. However, the algorithm does not compute the topological information describing which contours—holes—lie whithin which contours. In order to compute this information I have developed the member function computeHoles of the class Polygon. This function also gives an orientation—clockwise or counterclockwise—to the different contours taking into account the contour depth. The following program shows how the computeHoles member function can be used:

```
int main ()
{
   Polygon p ("polygons/samples/polygonwithholes");
   cout << "Number of contours: " << p.ncontours () << '\n';
   cout << "Number of vertices: " << p.nvertices () << '\n';
   p.computeHoles ();
   // show information
   for (int i = 0; i < p.ncontours (); i++) {
      cout << "--- new contour ---\n";
      cout << "Identifier: " << i << '\n';
      cout << (p.contour (i).external () ? "External" : "Internal") << " contour\n";
      cout << "Orientation: " << (p.contour (i).clockwise () ? "clockwise" : "counterclockwise")
           << '\n';
      cout << "Holes identifiers: ";
      for (int j = 0; j < p.contour (i).nholes (); j++)
         cout << p.contour (i).hole (j) << "  ";
      cout << '\n';
      cout << "Vertices: ";
      for (int j = 0; j < p.contour (i).nvertices (); j++)
         cout << p.contour (i).vertex (j) << "  ";
      cout << '\n';
   }
   return 0;
}
```

The computeHoles function makes a second plane sweep in order to compute the holes associated to each contour. Its execution time is $O(n \log n)$, where n is the number of edges of the polygon—of all its contours. Although, the computeHoles function is quite efficient the computation of holes can be integrated in the Connector class in order to make a single plane sweep—I hope to program this in the future.

7 Robustness

The implementation is based on the C++ double type. So, the implementation is not robust.

8 Acknowledgements

I would like to thank Christian Woltering for his corrections in the function Polygon::**operator**>> and for formulating a correct definition of the field poss of the SweeEvent structure.