IMPLEMENTATION OF SWEEPLINE SEGMENT INTERSECTION ALGORITHM IN GEOLAB GEOMETRICAL WORKBENCH

Giacomo Gattorna* Alper Üngör*

* *Department of Computer Science*
Rensselaer Polytechnic Institute, Troy, NY 12180
gattorng@cs.rpi.edu
ungora@cs.rpi.edu

Abstract. In this paper, we describe the work we have done to extend GeoLab's environment (a Computational Geometry Programming environment). We discuss the implementation, testing and animation of a Sweepline algorithm for finding segment intersections. The assumptions usually made during the implementation of Sweepline algortihm are relaxed using perturbation methods. A new algorithm based on a simple idea is also proposed. We discuss the other segment intersection algorithms and make a comparison of them. A programmer's manual for the Geolab, whose original documentation is in Potuguese, is also available in the paper.

Keywords. Line segment intersection, Computational geometry, Plane Sweep Algorithm

1. INTRODUCTION

In this study we investigated one of the basic problems in computational geometry. As stated in the thesis work of Shamos [6], the basic motivation for the intersection problems comes from the fact that two objects can not occupy the same place at the same time. These problems arise in many applications such as printed circuit design, architectural databases, computer graphics, cartography and robotics. A Very Large Scale Integrated circuit may contain millions of components. To produce a fast computer graphics system where there may be thousands of intersections, one should use an efficient intersection algorithm.

In this paper, we will specifically look at the line intersection problem. However, keep in mind that there exist other important problems that are reducible to the segment intersection problem as for instance the intersection of simple polygons. The reader is referred to Shamos [6] for further detail. We should also note that segment intersection problem is different than line intersection problem. Hence, when we refer to a segment through out the paper, we will always have two end points to represent it. The segment intersection problem can be defined as:
Given a collection of n segments in the plane, report all pairs of segments that intersect and their intersection points.

In the following section you will find a survey of segment intersection algorithms. Section 3 tells about the Sweepline algorithm in detail. We also stated the difficulties in the implementation and the degenerate cases and our solutions to them. Section 4 explains a new simple algorithm on segment intersection (Ordered Brute Force). In section 5, you will find a comparison of Ge-

[1] The code for the implementation is available on-line: http://www.cs.rpi.edu/~ungora/

olab's Brute Force algorithm and our implementation of Sweepline and Ordered Brute Force algorithms. A computaional analysis of the three algorithm is given. Section 6 stands as a programmer's manual to Geolab's environment.

2. SEGMENT INTERSECTION ALGORITHMS

The Brute-Force solution tests all pairs of line segments for intersection. This takes $O(n^2)$ time, since there are $n(n-1)/2$ pairs and each intersection test can be performed in constant time.

A plane sweep algorithm can be used to solve the problem in $O((k+n)log(n))$ time, where n is equal to the number of lines and k is equal to the number of intersections reported ([1] Bentley and Ottman). The key idea of the plane sweep algorithm is to avoid testing the pairs of segments that lie so far apart that they cannot possibly intersect. This algortihm is further discussed in next section. Originally, the space requirement for the algorithm was $O(n+k)$, but Brown reduced the space requirement of the plane sweep algorithm to $O(n)$, proposing a new insertion rule for the sweepline data structure ([2] Brown).

An optimal algorithm for the segment intersection problem is given by Chazelle and Edelsbrunner. Their starting point is Bentley and Ottman's solution. Two main ideas of their optimal algorithm are: (i) use of the lazy evaluation/call-by-need paradigm and (ii) a new combinatorial lemma on line arrangements. ([3] Chazelle and Edelsbrunner) The algorithm is optimal because, it doesn't stop at each intersection point but, instead, only at the extreme points of the segments. Therefore the $log(n)$ step is repeated n times instead of $n+k$ times. This is accomplished through a lazy update of the sweepline structure (only when needed) which takes into account segments' order swapping caused by the intersection points already found. Although the algorithm is theoretically optimal, it is complicated and not easy to implement.

Franklin et al.[8] used a uniform grid technique to compute the intersection of segments. Optimal grid size is dependent on the statistics of the input segments. Basically, for each grid cell, line segments that pass through the cell are found and kept in an ordered pair list. Then for each cell, all pairs are tested to find the intersections in that cell. This technique works well especially for telling which pairs of a large number of short segments intersect. A neat property of the uniform grid technique is that it is suitable for parallel processinng.

3. SWEEPLINE ALGORITHM FOR LINE INTERSECTION

3.1 *Algorithm description*

The Sweep line paradigm is powerful and simple: a two dimensional problem is reduced to a set of (easier) one dimensional problems along a line (sweepline) which is moved over the other dimension at certain meaningfull points called events. In the case of segments intersection this line is conventionally chosen to be vertical and the event schedule corresponds to the ordered list of all the x coordinates of all the segments' extreme points and the currently found intersection points. Then the sweepline is moved from the first event to the last and at each step the current event must be one of the following:

(1) The sweepline has reached the left extreme of one segment.

(2) The sweepline has reached one intersection point previously found.

(3) The sweepline has reached the right extreme of one segment.

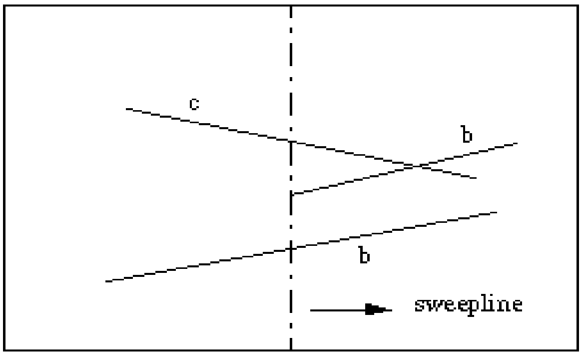

Fig. 1. Left end event point

In the first case the new segment encountered is introduced in the y-ordered active segments list (those crossed by the sweepline) according to the y coordinate of its first extreme. Then it is checked for a possible intersection with the two segments immediatly above and below it on the active list (a and c c in Figure 1).

In the second case the two segments which generated the intersection points have their position in the active list swapped and they are checked for intersection against their new neighbors (c against a and b against d in Figure 2).

In the third case, the segment whose second extreme is the current event is deleted from the active list and its ex neighbors are checked against each other for possible intersections (a and c as in Figure 3).

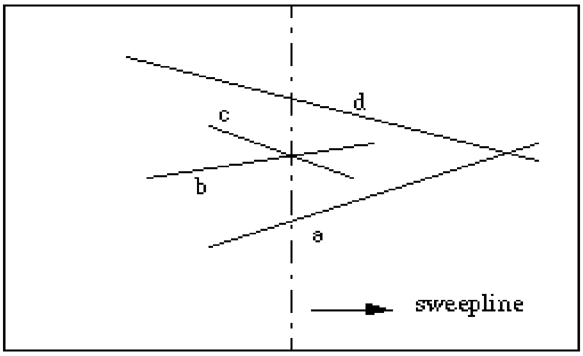

Fig. 2. Intersection event

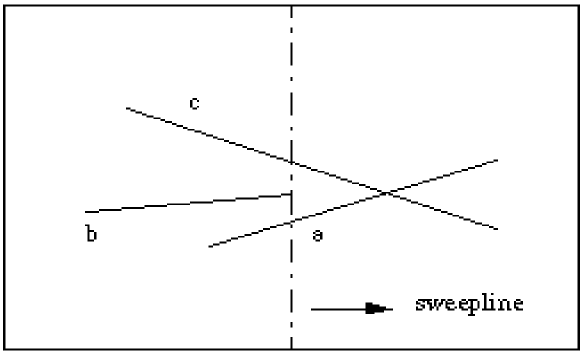

Fig. 3. Right end event point

Finally, if in any of the above cases an intersection is found, the intersection point is added to the ordered list of events (schedule) if its x coordinate is greater than the current sweepline position and it has not been inserted already. Then, each time the sweepline event is an intersection, the corresponding point is inserted in the result list of the algorithm.

The rationale behind the algorithm is that the necessary condition for two segments to intersect is to be adjacent on the active list at some time. Therefore we simply need to check the possible intersection of all those segments which happen to become neighbors in the active list.

3.2 *Computational complexity analysis*

Let n be the number of segments and k the total number of intersections found. From the above description it is easy to see that the number of events is exactly $2n + k$. Then, at each event, no more than two intersections are found and inserted in the active segments list and in the event schedule. Now, since no more than n segments can be active at the same time, there are no more than $n-1$ adjacent active pairs and, therefore, no more than $n-1$ new intersections added to the events schedule. So, both the active segments list and the events schedule have $O(n)$ elements and, with adequate data structures (AVL trees, randomized trees, etc), all insertions, removals, and ordered retrieval operations can be done in time $O(log(n))$. Hence the worst case time complexity of the algorithm is $((n + k)log(n))$.

3.3 *Algorithm implementation*

Our implementation is a modified version of the C++ algorithm presented in Lazlo's book [7] which take full advantage of the template definition for a randomized tree data structure which implements both the events schedule and the active segments list. Many modification have been required in order to integrate it in the Geolab environment. In particular we tried to use Geolab's LEDA 2.01 data structures and geometric objects definitions in order to avoid duplications and the runtime overhead of the conversion process (Geolab needs the output of the algorithm in its own format to display it in the *Editable area*. See Geolab sections 6.2 and 6.3). After the creation of the events schedule inserting all the extreme points in an instance of the randomized tree data structure, the algorithm basically loops over all the events calling the corresponding handle-transition method. These methods are then different depending on the current subclass of the considered event (handle-transition for the generic event is a pure virtual method which is implemented in the subclasses LeftEndpoint, RightEndpoint and Crossing). These three methods basically implement the actions described in section 3.1. The only implementation details which need to be mentioned are those concerning the treatment of vertical segments and roundoff problems.

3.4 *Special Cases*

Vertical segments can not be treated as normal segments by the sweepline algorithm because they have no ordering on the active segments list. The solution we adopted was to introduce a local perturbation on vertical segments so that they are not any more vertical. This is done during the construction of the event schedule when all segments are processed. Another solution (suggested in Laszlo[7]) is to rotate all the segments. The problem with this solution is that it is difficult to find an angle such that no other segment is becoming vertical as a side effect of the global rotation. Moreover, such rotation would have added a noticeable overhead on the processing time and would have created additional problems for the animation phase. Adopting this solution, in fact, would have forced us to deal with a virtual (rotated) representation of the segments and the original set given in input (and automatically displayed) by the Geolab environment.

3.5 *Roundoff Problems*

The sweepline algorithm is prone to roundoff errors because it is based on two ordered sets (events and active segments) which can be topologically modified because of roundoff errors. Such a modification in the order of these sets can cause many subsequent steps of the algorithm to fail without an apparent reason. The possible roundoff error belong basically to the following three classes:

(1) Segments crossing the sweepline almost in the same point.

(2) Almost vertical segments.

(3) A segment with one extreme almost on another segment.

For the first class we introduced a factor $EPSILON1$ to evaluate the two (or more) segments crossing the sweepline a little bit before current sweepline position x. This allows to obtain y values for these segments different enough from each other to be unaffected by roundoff errors. Unfortunately as the chosen $EPSILON1$ gets bigger (and the chance of these roundoffs errors smaller), problems of class 2 and 3 get worse (see Figure 4).

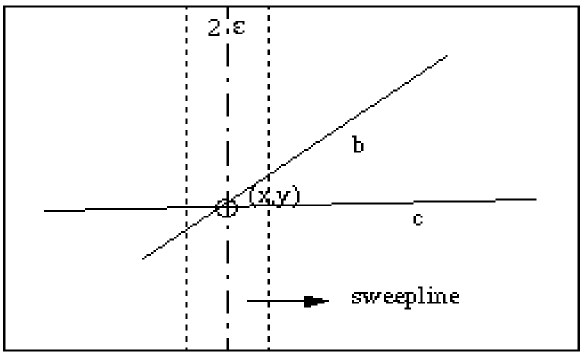

Fig. 4. Two lines crossing almost on the sweepline

For the second class we decided to perturbate all segments whose verticality did not satisfy a certain threshold as we did with the vertical segments. Unfortunately, as for the real vertical segments, the needed perturbation can be pretty big because of the way sweepline active segments order is enforced. In fact, given the $EPSILON1$ introduced above and a current sweepline position x, if the slope of the perturbated segment is still too close to infinity, the y value of the perturbated segment at $x - EPSILON1$ can be very far away from the y value at x. If then we are trying to evaluate the position of the perturbated segment relative to a pair of segments very close to each other, this y can give us a wrong result as seen in Figure 5.

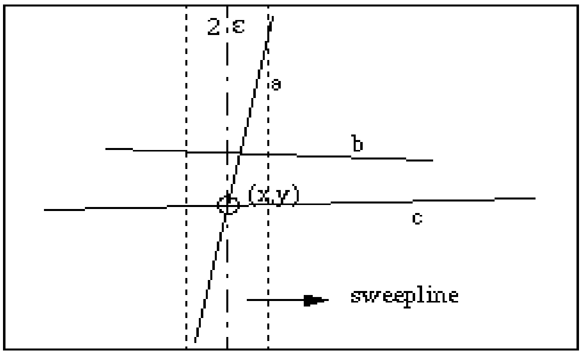

Fig. 5. Almost vertical line segment (labelled as a)

Finally the third class consists of the typical problem of segments which intersect in one of the extreme points. At every intersection point $(x1, y1)$ the order of the y coordinates of the crossing segments for all x's greater than $x1$ is the opposite of the order of the y's for all x's less than $x1$. Hence, if we report an intersection we would also like to keep the above property. Unfortunately Geolab's segments intersection method is making different calculations than the function used for ordering the y's and the active segments along the sweepline. This can bring to inconsistence between intersections and active segments positions. The problem is then worsened by the $EPSILON1$ introduced above. In fact, even if we have found an intersection, the evaluation of the order of the active segments at $x - EPSILON1$ instead of x can make the segment disappear from the active list when it shouldn't (see Figure 6). A more systematic solution to this problem would try to use the same mathematical operations for both the ordering function and the intersection method. Then the evaluation of the active segments list should be done twice at x and $x - EPSILON1$ discarding the mistaken result (if any) due to the presence of a particular case of class 1 or 3.

As easily seen roundoff problems for the Sweepline algorithm can not be overlooked and need to be taken into account for real world applications because they affect both the efficiency and the complexity of the code (easier approaches as the brute-force and ordered brute-force algorithm are almost "immune" to the problem).

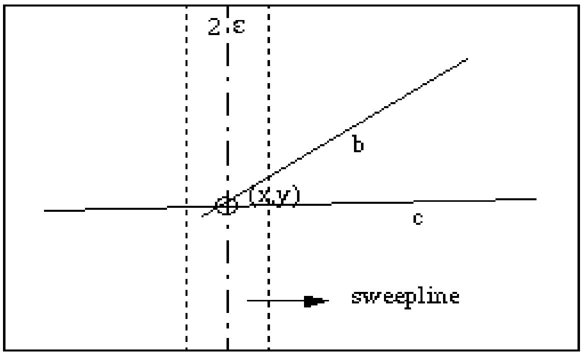

Fig. 6. A segment with one extreme almost on another segment.

3.6 *Animation*

As we described in section 6, Geolab's environment allows to easily animate geometrical algorithms. Our animation of the sweepline algorithm shows at each event point the "interesting" active segments. In particular we can have up to four segments shown if the sweepline is at an intersection event point. We used the red color to represent the segment (or couple of segments if its a crossing point) which "generated" the event and the blue color for the (eventual) immediate upper and lower neighbor(s) of the red segment(s). If a new intersection is found, the corresponding point is evidentiated by a circle. (See Figures at the end of the paper).

4. A SIMPLE IDEA TO IMPROVE BRUTE-FORCE

Brute-Force algorithm checks all pairs of lines to find the intersections. We wanted to reduce the number of tests. To achieve that we used the fact that "two segments can intersect only if their projections on the x-axis overlap". The algorithm can be explained as in the following steps:

- Step 1. Order the segments according to the left end points in the x-axis. Take the first segment in the list as the currently active segment.
- Step 2. Perform the intersection test of the currently active segment against all the following segments until the left endpoint of one of them is no longer within the range of its rightmost extreme.
- Step 3. Discard the previously active segment and select the following instead Then repeat step 2 and 3 until no segments are left in the list.

In Figure 7, all sorted segments are labelled lexicographically to the clarify the Ordered Brute Force Algorithm. In the example, segment a is already processed and its intersections are reported (a circle at a and d intersection point). Segment b is the currently active segment: intersection test for segment b is performed against c and

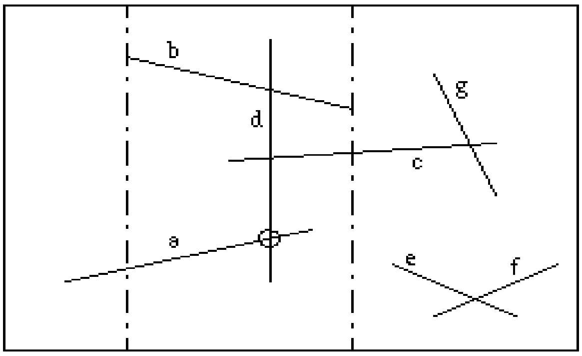

Fig. 7. Ordered Brute Force Algorithm

d, but not not against e and the others. Checking if the left end point of segment e is in the range of segment b, stops the rest of intersection tests for segment b. Worst case complexity for the algorithm is still $O(n^2)$ as in Brute Force. However, as we will see in the next section, the algorithm performs very well on short segments

5. COMPARISON OF SEGMENT INTERSECTION ALGORITHMS

Geolab environment gave us the chance to easily test the performance of the algorithms. We have designed a new input module to create random segments which allowed us to test the algorithm on sets with different number of segments and different maximum lenghts. Random lengths are created defining a box centered on the first extreme and generating random coordinates for the second extreme within this box). These random boxes have sizes which are a fraction (we call this fraction f) of Geolab's *Editing Area* sizes. For the test we analyzed the influence of both the number and the lengths of the segments on the execution time of the algorithms. In Table 1 we report the results optained with different number of segments (50, 100, 500, 1000, 2000) and with f equal to 1/10. In Table 2 we fixed the size of the problem equal to 200 segments and we see how the executions times change with f equal to (1, 1/2, 1/5, 1/10, 1/20).

The test with fixed maximum length and different number of segments basically confirm the theoretical analysis: the pure brute force approach has a quadratic complexity which makes it impractical when the input is too large. On the other hand Sweepline algorithm time is increasing a bit more than linearly confirming the $(n + k)log(n)$ complexity. Surprisingly, the simple improvement over the Brute Force approach in the Ordered Brute Force algorithm make it the fastest of the three although its growing rate is definitely higher than the Sweepline algorithm. The almost equal result with 2000

Number of Segments	Sweep Line	Brute Force	Ordered BF	Number of Intersections
50	0.13	0.15	0.03	3
100	0.36	0.83	0.11	8
200	0.96	3.24	0.30	24
500	4.06	18.63	1.75	143
1000	11.50	82.03	6.29	554
1500	21.44	174.21	13.98	1198
2000	33.94	409.12	28.73	2209

Table 1. Execution time for different number of segments, keeping f factor constant. (f=1/10)

f	Sweep Line	Brute Force	Ordered BF	Number of Intersections
1	26.63	3.50	2.49	2231
1/2	6.40	3.33	1.29	548
1/5	1.60	3.25	0.51	82
1/10	0.96	3.24	0.30	24
1/20	0.69	3.12	0.21	8

Table 2. Execution time for various f values, keeping the number of segments constant ($N = 200$)

segments (though its time on less number of segments was definitely smaller) let us guess that on larger input its inherent quadratic complexity will make it less efficient than the theoretically better sweepline. The second test helps in understanding which algorithm and how much is output sensitive. The Brute Force showed almost constant time where more intersections added just a little overhead for some additional calculations. On the other hand both Ordered Brute Force and Sweepline algorithms are very dependant on the number of intersections reported. In particular the Sweepline algorithm confirms its more than quadratic worse case (on a quadratic number of intersections) penalized also by the considerable overhead needed for its complex data structures. Ordered Brute Force is only indirectly dependent on the output: the more are the intersections with respect to the input segments, the more these are longer on the average and the less is the gain of ordering the segments. As a final consideration we note that the simplicity and wider range of applicability of the Ordered Brute Force approach, make it preferable whenever the amount of data in input is not in the order of thousands of segments or if it is known a priori that the segments are more likely to be short.

6. GEOLAB ENVIRONMENT

Geolab is a programming environment for the development of algorithms in Computational Geometry (See [4] and [5]). Geolab binds together support for software and algorithm development and support for real-time interaction. Essentially, these consist of:

Support for development:

- Built-in abstract data types for representaion of geometric (and other) entities.
- Data structures and general purpose algorithms.
- Basic algorithms and some complex geometric data structures (as building blocks).
- Mechanisms for creation and incorporation of new components without requiring the recompilation of the environment.
- Tools to allow the production of reusable shared libraries.
- Methods to convert between data representations.

Support for interaction:

- An interactive graphical environment for the manipulation of geometric models.
- Facilities for the construction of sets of input data to test average performance and degenarate cases.
- Support for debugging and obtaining statistical data on run-time performances.
- tools for handling secondary storage.
- Customization of algorithm animation (like speed).

Geolab is written in C++ and makes extensive use of object oriented programming for a hierarchical modeling of geometric objects and of geometric algorithms. It currently has about 40 algorithms in its dynamically linked libraries which are made up of roughly 45000 lines of C++ code (plus an additional 50000 lines for the environment itself). It runs on SparcStations under Sun/OS using XView graphics library, following the OpenLook graphical user interface guidelines.

6.1 *Graphical Interface*

Geolab's graphical interface consists of an *Editing area* onto which objects can be manually or automatically generated, a *Palette of Operation Modes* (with two user definable buttons) and a set of cascade-menu buttons with pins to confortably stick the menu in separate windows and move them around the screen (See Figure 9 at the end of the paper).

From the palette of *Operation Modes* the user can pick a tool to create, select, move, reshape, zoom or scroll the objects on the *Editing area*, while from the various menus, the user can choose a variety of actions.

Once objects have been created in the *Editing area*, the user may select some or all of them and perhaps change their properties. Among various graphical appearance properties (e.g. line thickness or color) there are other manipulation properties like selecting if objects are deletable or not. All the algorithms selected in *Algorithm menu* are applied to the currently selected objects in the *Editing Area*. Whenever the selected object can not be handled by one algorithm, its corresponding entry in the menu is temporarily disabled. When an algorithm is executed and its result is a set of geometric objects with graphical representation, these objects are placed in the *Editing area*.

Additionally, two user-chosen externally implemented modes can be installed in the palette of *Operation Modes* at any given time. One of those is an *Interactive Creation Mode* and the other is a *Functional Mode*. These can be just about anything, such as an interactive simple polygon creation tool or a query type for the shortest path mode among obstacles.

Note that some of the input tools may require objects to work on (e.g. Random Polygon create a polygon out of a given set of points), but are not disabled on empty inputs as they are legally handled by the corresponding procedure.

6.2 *Geolab Programming Guide*

To add new algorithms to Geolab environment is necessary to do the following steps which describe how to create the required dynamic linked library with the stubs functions needed to communicate with Geolab.

STEP 1: Algorithm Implementation.

The first obvious step is to create a procedure in C++ (Geolab's programming language) which implements the algorithm. The constraints to the implementation are that it must be compilable by CC-3.0.1 compiler (for creating a compatible symbol table for the dynamic library) and must return all geometric objects to display as a set of Geolab geometrical objects (Point2D, Segment2D, etc.). Obviously a better integration and an easier development is done if one uses Geolab specific libraries of geometric objects and data structures (LEDA: Library of Efficient Data Structures and Algorithms).

STEP 2: Mapping of the algorithm on Geolab's class *Algorithm*

First thing to be done is to define a class for the new algorithm derived from Geolab's *Algorithm* class:

```
class IntersectionProblems : public Algorithm
{
 public :
      bool animation() { return true; }
};

class InterSegments : public IntersectionProblems
{
 public :
      bool is_able_to_handle(const SelectedObjects &,
                             const GraphicObjects &);
};

class AllIntersectionsReportSWEEP : public InterSegments
{
   ResultObjects binder(const SelectedObjects &,
                        const GraphicObjects &,
                        World *w = 0,
                        int animation_depth = 0,
                        int reference_depth = 0);
};
```

Then a link function has to be defined to create an instance of the new class:

```
Algorithm* all_intersections_report_sweep_link()
{
   return new AllIntersectionsReportSWEEP();
}
```

Finally methods *binder* and *is_able_to_handle* have to be redefined. The first one should include the actual call to the new algorithm (with a conversion function of the input parameters from their graphical object representation), and the second one should define which input parameters are valid for the algorithm:

```
ResultObjects  AllIntersectionsReportSWEEP::binder(
                         const SelectedObjects &SO,
                         const GraphicObjects &GO,
                         World *w,
                         int animation_depth,
                         int reference_depth)
{
   ResultObjects result;
   list(Point2D_ptr) all_inter;
   Point2D_ptr p;

   if (!SO.empty()) {
      all_inter = all_intersections_report_sweep(
                          extract_segments(SO, GO),
                          w, animation_depth,
                          reference_depth);

      forall(p, all_inter) {
        result.push(p);
      }
   }
   return result;
}

bool InterSegments::is_able_to_handle(
                     const SelectedObjects &SO,
                     const GraphicObjects &GO)
{
   list_item it;

   forall(it, SO) {
```

```
      if (GO[it]->type() != Segment2D_t) {
        SO.init_iterator();
        return false;
      }
   }
   return true;
}

static list(Segment2D_ptr) extract_segments(
                    const SelectedObjects &SO,
                    const GraphicObjects &GO)
{
    list(Segment2D_ptr) res;
    list_item it;
    forall(it, SO) {
       res.push((Segment2D *)GO[it]->obj);
    }
    return res;
}
```

STEP 3: The inclusion of the required include files.

Basically the include files required are those for the definition of external algorithms:

```
#include "External.h"
#include "Algorithm.h"
```

and all those for the definitions of LEDA geometric objects and data strucures:

```
#include "basic.h"
#include "List.h"
#include "Point2D.h"
#include "Segment2D.h"
```

etc.

STEP 4: Makefile

The makefile for the creation of dynamic linking modules to be used by Geolab requires specific flags for accessing the correct libraries and include files and for the creation of a pure relocatable module: [2]

```
DLIB = $(GEOLAB_HOME)/lib/
DOBJ = obj/
CFLAGS = -I$(GUIDEHOME)/include
 -I$(OPENWINHOME)/include -PIC
LDFLAGS =
CC = CC-3.0.1

OBJS=  \
               $(DOBJ)Integer.o \
               $(DOBJ)Intersections.o \
               $(DOBJ)intersections.o

$(DLIB)Intersections.so: $(OBJS)
        ld $(LDFLAGS) -assert pure-text
             -o $(DLIB)Intersections.so $(OBJS)
```

`GEOLAB_HOME`, `GUIDEHOME` and `OPENWINHOME` are environment variable which must be set to use Geolab to the directory where respectively Geolab, Guide1.1 and OpenWindows3 are. (see Geolab's `HOW-TO-COMPILE` file).

STEP 5: Geolab's menus extension

Finally the new algorithm execution must be binded to Geolab's *Algorithm* menu. This is easily done editing `.geolab-menu` file which define new menu entries between the keywords `MENU` and `END`. Each menu entry [3] is composed by a string to appear in the menu itself, a string identifying the algorithm to call and a string defining the dynamic library where the algorithm is. [4]

```
MENU "Segments"
   "Intersections Counting (brute force)"
          "all_intersections_counting_bf"
                       "Intersections.so"

   "Intersections Report (brute force)"
          "all_intersections_report_bf"
                     "Intersections.so"

   "Intersections Report (ordered brute force)"
                 "all_intersections_report_obf"
                             "Intersections.so"

   "Intersections Report (sweep line)"
          _intersections_report_sweep"
                    "Intersections.so"
END
```

STEP 6: Geolab's animation extensions

After steps 1 to 5 the algorithm should be working fine in Geolab environment, but without any animation capability. This can be added through animation function calls enclosed by two animation macros:

```
ANIMATION_BEGIN(w, animation_depth, reference_depth)
<animation code>
ANIMATION_END(w, animation_depth, reference_depth)
```

Parameters `w`, `animation_depth`, and `reference_depth` are passed to the algorithm by Geolab and represent the graphic world to operate in the *Editing Area*. The depth parameters are used to create animations whose depth can be changed by the user through the animation options window. In between the two macro is possible to use any C++ code (in order to manipulate the graphi-

[2] Unfortunately there are some limitations due to the incompatibility of Geolab with compilers other than Sun C++ 3.0.1. This compiler has some troubles with templates when it comes to build a dynamic library. This forced us to use only inline defintions for all template classes

[3] Although for convenience they are written separately , each group of three lines has to stand as one line in the actual code.

[4] An algorithm's main call has to be defined as the name of the instance of type *Algorithm* defined in step 2 (i.e. `all_intersections_report_sweep_link()`) without the final `_link()` part. Usually, for clearness, this name is the same used for the algorithm defined in step 1.

cal objects in the Geolab's graphical world `w`) plus two animation functions: [5]

```
frame_stop_mark(World *w);
// put breakpoints to do step by step animations.

frame_delay_mark(World *w);
// set a mark to apply the frame delay
// specified by the user.
```

7. CONCLUSION

In this paper we analyzed the problem of segments intersection making a short survey of existing techniques and evaluating the implementation of one of the most commonly used (sweepline algorithm) within the framework of an extensible computational geometry workbench (Geolab).

The implementation of the algorithm raised some practical issues concerning roundoff problems and particular cases which demonstrated how this algorithm (as many other in computational geometry) is prone to approximations errors due to the finite representation of floating points numbers. A robust implementation of the algorithm would have required a considerable development time which should be taken in account whenever this approach is used for real world data. On the other hand a comparative test showed that for a reasonable amount of data (or for data with particular properties) a slightly improved version of the Brute Force approach has obtained better performances with an insignificant development time and error proof robustness.

Finally, this investigation has been possible thanks to the use of Geolab's environment which has been an indispensable tool for easily developing, visualizing and evaluating all the algorithms. The presence of a large library of C++ geometric objects and data structures (LEDA) reduced the development time considerably. The weakest part of Geolab is the lack of a complete documentation in English. Some important parts as the programming guide are described only in Portuguese. Furthermore, LEDA library version included in the package is the quite old 1990 release (2.01).

[5] For finding the definitions of graphical methods of Geolab's objects look in files "Obj.h" and "Obj.C".

REFERENCES

[1] Bentley J.L., Ottman T.
Algorithms for reporting and counting geometric intersections,
IEEE Transactions on Computers C-28 (1979), 643-647.

[2] Brown K.Q.
Comments on "Algorithms for reporting and counting geometric intersections"
IEEE Transactions on Computers C-30 (1981), 147-148.

[3] Chazelle B., Edelsbrunner H.
An optimal algorithm for intersecting line segments in the plane.
In Foundations of Computer Science - 29th Annual Symposium, White Plains, NY, Oct 1988.

[4] Rezende P. J., Jacometti W. R.
Animation of Geometric algorithms using Geolab.
In Proc. 9th Annual ACM Symposium Computational Geometry, 1993.

[5] Rezende P. J., Jacometti W. R.
Video: Animation of Geometric algorithms using Geolab.
In Video Review of the 9th Annual ACM Symposium Computational Geometry, 1993.

[6] Shamos M.I.
Computational Geometry, Dissertaion for Ph.D. Yale University, 1978.

[7] Lazslo M.J.
Computational Geometry and Computer Graphics in C++
Prentice Hall, 1996.

[8] Akman V., Franklin W. R., Kankanhalli M., Narayanaswami C.
Geometric computing and uniform grid technique.
In Computer-Aided Design, vol. 21, number 7, september 1989.