Fully Dynamic Constrained Delaunay Triangulations

Marcelo Kallmann[1], Hanspeter Bieri[2], and Daniel Thalmann[3]

[1] USC Robotics Research Lab, Computer Science Department, University of Southern California[†] *kallmann@usc.edu*
[2] Institute of Computer Science and Applied Mathematics, University of Bern *bieri@iam.unibe.ch*
[3] Virtual Reality Lab – VRlab, Swiss Federal Institute of Technology – EPFL *daniel.thalmann@epfl.ch*

Summary. We present algorithms for the efficient insertion and removal of constraints in Delaunay Triangulations. Constraints are considered to be points or any kind of polygonal lines. Degenerations such as edge overlapping, self-intersections or duplicated points are allowed and are automatically detected and fixed on line. As a result, a fully Dynamic Constrained Delaunay Triangulation is achieved, able to efficiently maintain a consistent triangulated representation of dynamic polygonal domains. Several applications in the fields of data visualization, reconstruction, geographic information systems and collision-free path planning are discussed.

Key words: Constrained Delaunay Triangulation, Dynamic Constraints.

1 Introduction and Related Work

Delaunay Triangulations and Constrained Delaunay Triangulations are popular tools used for the representation of planar domains. Applications include, for instance, data visualization [21], reconstruction [3], mesh generation [20], and geographic information systems [23].

The Constrained Delaunay Triangulation (CDT) is an extension of the Delaunay Triangulation (DT) to handle constraints. A CDT can be seen as the triangulation closest to the DT that respects given constraints.

The computation of the DT of a set of points in $\mathbb{R}^2$ is a classical problem in computational geometry [15]. Asymptotically optimal algorithms are known, as the *O(n log n)* divide-and-conquer [12] and sweepline [9] algorithms. For CDTs, an optimal *O(n log n)* divide-and-conquer algorithm is also known [5].

However the most popular implementations for both DT and CDT are those which are incremental and do not require the use of complicated data structures in addition to the triangulation itself. In general, such incremental algorithms take worst-case time of $O(n^2)$ mainly due to point location, which

[†] Work done while at VRlab-EPFL

usually takes $O(n)$ to locate each point to be inserted. However the *jump-and-walk* method of point location [14] allows the incremental computation of DTs within an expected time of only $O(n^{4/3})$ for randomly distributed points. Simple incremental algorithms can thus be competitive with optimal (but more complex) *O(n log n)* approaches. Note that, if dedicated data structures are maintained to optimize point location, the incremental algorithm reaches the expected time of *O(n log n)* [6].

Along this paper we mainly discuss complexity analysis results from existing incremental DT algorithms, expecting that similar asymptotic times are obtained for CDTs without "special cases". Therefore, constraints should have few intersections and not make the CDT become too different from the DT of the vertices of the constraints. Note that constraints can be defined to make a CDT match any given triangulation, thus invalidating time complexity analysis relying on the Delaunay property, as it is the case with the jump-and-walk method [14].

This work presents the implementation of incremental algorithms permitting to efficiently update CDTs in case of online insertions and removals of constraints. We consider constraints to be defined by any set of points and line segments. They may describe open polygonal lines, and simple or non-simple polygons. Edge overlapping, self-intersections and duplicated points are allowed, and are automatically detected and handled online.

Constraints are identified by ids at insertion time so that they can be removed later on. Note that, as edge overlapping and intersections are allowed, a constrained edge in the CDT can represent several input constraint segments. Therefore, we need to coherently keep track of ids during insertion and removal of constraints in order to keep updated which triangulation edges represent which input constraints.

Our implementation for constructing CDTs is strongly related to the approaches taken in previous works [17] [10] [1] [19]. Our main contributions in this paper are extensions to allow keeping track of overlapping and intersecting constraints during insertion and removal operations.

In order to achieve a fast and robust implementation, we follow a *topology-oriented* implementation approach [18], making use of floating-point arithmetic and *epsilons* to better treat degenerate input data. A similar approach has also been successfully used for the implementation of the Voronoi diagram of points and segments [13].

Starting from an initial (trivial) CDT, updates are done respecting all topological properties the CDT has to fulfill. Thus, topological consistency is guaranteed independently of errors in numerical computations. Our aim is that the algorithm never fails, giving always a topologically-valid output.

Our implementation has been successfully tested with several applications requiring representations of dynamic planar domains. Our first application which motivated this work was the navigation of characters in virtual environments. For this case we use the CDT as a cellular decomposition of the

free space, so that a simple search over the free triangles is sufficient to determine a free passage joining two input points, if one exists. Our approach becomes particularly interesting because we are able to efficiently update the environment description when obstacles (or constraints) move.

The ability to deal with degenerated constraints and to dynamically insert and remove them in CDTs, leads to a number of new possibilities in certain applications. We present and discuss several examples in different fields: data visualization, reconstruction, geographic information systems and collision-free path planning.

2 Background and Problem Definition

Let V be a finite set of vertices in $\mathbb{R}^2$. Let E be a set of edges where each edge has as its endpoints vertices in V. Edges are closed line segments. A triangulation $T = (V, E)$ is a planar graph such that: no edge contains a vertex other than its endpoints, no two edges cross, and all faces are triangles with their union being the convex hull of V.

Let $T = (V, E)$ be a triangulation. An edge $e \in E$, with endpoints $a, b \in V$, is a Delaunay edge if there exists a circle through a and b so that no points of V lie inside this circle. If T has only Delaunay edges, T is called a *Delaunay Triangulation* (DT) of V. Note that a Delaunay triangulation is not unique in case V has four or more co-circular points.

Let $S = \{C_1, C_2, ..., C_m\}$ be a set of m constraints, such that for each $i \in \{1, 2, \ldots, m\}$, constraint C_i is defined as a finite sequence of points in $\mathbb{R}^2$. C_i may contain a single point, otherwise it represents a polygonal line. Polygonal lines are allowed to be of any form: open or closed, with self-intersections, with overlapped portions or with duplicated points. For simplicity of notation, we call the segments or points of all the constraints in S by simply the segments or points in S.

Let S be a set of constraints as defined above. $T(S) = (V, E)$ is a *Constrained Delaunay Triangulation* (CDT) of S if:

- For each segment $s \in S$, there is a number of edges in E such that their union is equal to s. Such edges are called constrained.
- For each point $p \in S$, there is a vertex $v \in V$, such that $v = p$. Vertex v is said to be constrained, and if it has no adjacent constrained edges, we say that v is *isolated*. Isolated constrained vertices appear due to constraints defined as a single point.
- For every non-constrained edge $e \in E$, e is a Delaunay edge with respect only to the vertices connected with edges to the endpoints of e.

An edge $e \in E$ is constrained if it is equal to, or is a sub-segment of, some segment $s \in S$. All the other edges are non-constrained. Note that, as we allow segments in S to intersect, finding $CDT(S)$ implicitly means to determine all the intersection points, including them as additional vertices in V.

Segments having intersections are split at the intersection points, generating sub-segments which are inserted as edges of E (see Figure 1). Overlapped segments share a same constrained edge in $CDT(S)$.

The number and form of the constraints defined in S determines "how far" $CDT(S)$ is from $DT(V)$. Delaunay triangulations have the nice property of maximizing the minimum internal angle of triangles, and this property is kept in $CDT(S)$ as far as possible. Note that it is possible to define a set of constraints that forces the CDT to become any desired triangulation.

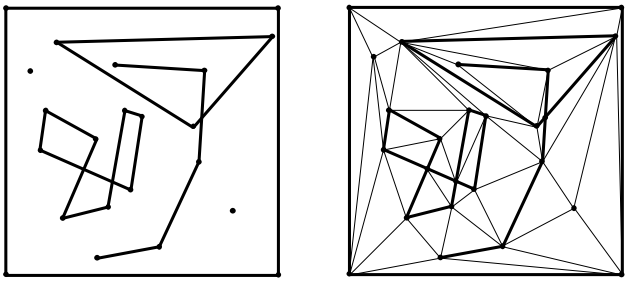

Fig. 1. Left: a set of constraints S composed of two points, a square, a triangle, a non-simple polygon, and an open polygonal line. Right: $CDT(S)$.

Problem definition. Let $S = \{C_1, C_2, ..., C_m\}$ be a set of constraints. The problem we want to solve is to construct and maintain a Dynamic Constrained Delaunay Triangulation of S, more precisely:

- Construct an initial $CDT(S) = (V, E)$, taking into account all possible degenerated cases in S.
- Insert constraints incrementally, i.e., given a new constraint C and a triangulation $CDT(S)$, we want to efficiently compute $CDT(S \cup \{C\})$.
- Remove constraints incrementally, i.e., given a constraint $C \in S$, and a triangulation $CDT(S)$, we want to efficiently compute $CDT(S \setminus \{C\})$.

3 Method Overview

Let $S = \{C_1, C_2, ..., C_m\}$ be a set of constraints. Our algorithm to construct $CDT(S)$ is incremental, and a box B strictly containing all constraints in S must be determined in advance. This is a common initialization requirement that allows considering only the case of inserting constraints in the interior of an existing CDT. B can be any large box that contains all possible constraints to be inserted, as for instance the enlarged (by any factor) bounding box of S. The triangulated B becomes the initial CDT. B constitutes the boundary of $CDT(S)$.

Once the CDT has been initialized we can insert, in any order, any constraint C_i, $i \in \{1, 2, \ldots, m\}$. In order to cope with overlapping segments and to identify constraints for later possible removals, we associate with each edge of the CDT a list containing the indices of the constraints it represents (an empty list means that the edge is not constrained):

```
struct edge_information { list<int> crep; };
```

For example, consider that segment $s_1 \in C_j$ has endpoints $\{(2,0),(3,0)\}$, and segment $s_2 \in C_k$ has endpoints $\{(1,0),(4,0)\}$. During the process of inserting segments s_1 and s_2 in the CDT, the overlapping is detected and a single edge e with vertices coordinates $\{(2,0),(3,0)\}$ is created. Edge e will therefore keep indices $\{j,k\}$ in its list `crep`. Note that segment s_2 will be represented in the CDT as the union of the three edges $\{(1,0),(2,0)\}, \{(2,0),(3,0)\}$ and $\{(3,0),(4,0)\}$. If s_1 is later on removed, s_2 becomes represented by the edge $\{(1,0),(4,0)\}$.

Vertices in the CDT must also keep track of the constraints it represents, however it is sufficient to count them. Therefore, each vertex in the CDT keeps track of a reference counter, and can be removed from the CDT only if this reference counter is 1. As this is a trivial procedure, we will not make further considerations concerning the management of such reference counters in the remainder of this paper.

Epsilon determination. Another initialization requirement is to determine the value of the epsilon to be considered in numerical computations. As we rely on standard floating-point precision arithmetic, epsilons are used to detect data inconsistencies. The epsilon mainly determines the threshold defining how far distinct points shall be considered the same, and its value reflects the precision of the measurement method. We assume that the user is aware of the nature of the input data and is able to determine the best epsilon to be used.

Held [13] proposes a more sophisticated approach to implement epsilon-based computations. A value is also required as input from the user, but it is considered as an upper bound. The lower bound is automatically determined by the floating-point precision of the used machine. In this way, geometric algorithms start by using the lower bound epsilon, and in case of problems, its value is incrementally grown until these problems are solved. In case the upper bound is reached before, data cleaning is performed and the computation restarts.

Data structure. The algorithms we will present require the triangulation to be represented by an efficient data structure, i.e. capable to give all adjacency relations in constant time. We implemented a data structure following the adjacency strategy of the *quad-edge* structure [12] and integrating element lists and operators as in the *half-edge* structure [27].

Our basic element is a structure representing an oriented edge. Each oriented edge is associated with only one vertex, one edge and one face. We call this basic element a `SymEdge`. This name is due to the fact that for each

`SymEdge`, there is another symmetrical one, which is associated with the other vertex and face sharing the same edge.

Each `SymEdge` structure keeps a pointer to the next `SymEdge` in the same face (`nxt` pointer), and a pointer to the next `SymEdge` rotating around the same vertex (`rot` pointer). A counter-clockwise orientation is used. The symmetrical element of a `SymEdge` is obtained by composing the `nxt` and `rot` pointers. In addition, three pointers are also stored in each `SymEdge` in order to retrieve the associated vertex, edge and face elements. The vertex, edge and face elements are organized in lists, and are used to store any application-specific data (as `crep` lists).

Construction and traverse operators are defined as a safe interface to manipulate the structure. We have implemented the same traverse operators as described in a previous paper [26]. In that work, a benchmark is presented where the mentioned "simplified quad-edge structure" is equivalent to the `SymEdge` structure described here. The benchmark indicates that the `SymEdge` structure is one of the fastest to describe general meshes. Note however that specific structures for describing triangulations may have better performance, as discussed in the work of Shewchuk [17].

4 Constraint Insertion

Given a constraint to be inserted in a CDT, we incrementally insert all its points and then all its segments, as shown in the following routine:

```
insert_constraint ( C_i, i )
   for all points p in C_i
       LocateResult lr = locate_point ( p );
       if ( lr is an existing vertex )
          v = lr.located_vertex;
       else if ( lr is on an existing edge )
          v = insert_point_in_edge ( lr.located_edge, p );
       else
          v = insert_point_in_face ( lr.located_face, p );
       add v to vertex_list;
   for all vertices v in vertex_list
       if ( v is not the last vertex in vertex_list )
          v_s = successor of v in vertex_list;
          insert_segment ( v, v_s, i );
```

Point location. The routine `insert_constraint` requires to locate points. For each point p to be inserted in the CDT, the `locate_point` routine searches where in the triangulation p is. Point location is an important issue for any incremental algorithm, and several approaches have been proposed. Most efficient methods rely on dedicated data structures, reaching the expected time of *O(log n)* to locate one point [6] [4] [11]. Alternatively,

bucketing has also been used [16] with good performances for well-distributed points. We follow the simpler *jump-and-walk* approach [14], which takes expected $O(n^{1/3})$ time (in DTs). It has the advantage that no additional data structures are needed, which is an important issue in our case: as constraints can be dynamically updated, the use of any additional data structure would also imply additional updates after each operation.

The jump-and-walk method of Mücke, Saias and Zhu [14] first defines a random sample of $O(n^{1/3})$ vertices from the triangulation (n being the current number of vertices), and then determines which one of these is closest to p. Finally, an oriented walk is performed, starting with one triangle t adjacent to the chosen vertex.

The oriented walk [24] [12] method selects one edge e of t, which separates the centroid of t and p in two distinct semi planes. Then, e is used to switch t to the other triangle adjacent to e. This simple process continues until t contains p. However it is only guaranteed to work for DTs [22] [8], and we have included an additional test to ensure its convergence in our CDT search. First, each visited triangle is marked. Then, whenever two edges exist separating the centroid of the current triangle and p in distinct semi planes, the one leading to a non-marked triangle is chosen. Marking triangles does not imply any overhead: we simply keep an integer for each triangle, and for each new search an incremented integer flag is used as a mark.

The geometric test required to determine if two points lie in the same side of a segment is implemented with a standard `CCW` (counter-clockwise) orientation test. However, as this test is not always robust, we switch to an epsilon-based walk mode when a loop is detected, i.e., when there are no unmarked triangles to switch to during the walk. The epsilon-based walk includes geometrical tests to explicitly check for each triangle t during the walk, if p is equal to some vertex of t, or if p lies on some edge of t (within the epsilon distance).

Point insertion. The point location routine determines (within the epsilon distance), if point p is already in the CDT, if it lies on an edge, or if it lies inside a face of the CDT. If p is already there, it is simply not inserted; otherwise a new vertex v is created in the located edge or face. If p is located in an edge it is first projected to that edge before insertion.

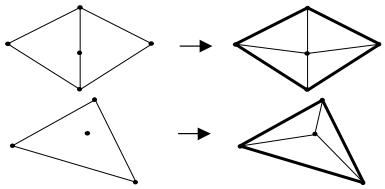

Fig. 2. Point insertion in an edge (top) and in a face (bottom). The edges outlined in bold (right side) constitute the initial edge set $F(p)$.

When p is inserted, non-Delaunay edges introduced in the triangulation need to be corrected. We follow the popular approach of flipping non-Delaunay edges, until all edges become Delaunay [12].

After inserting p, the union of all triangles incident to p forms a star-shaped polygon P. The edges on the border of P constitute the current edges being considered for flipping, and are noted here by the set $F(p)$ (Figure 2). Note that just after p insertion, P is either a triangle or a quadrilateral, and during the whole flipping process $F(p)$ continues to be a star-shaped polygon.

The insertion of one point may require $O(n)$ edge flips, however for DTs with a random input it is known that the expected number of edge flips is constant, no matter how they are distributed [11].

Note that if p is inserted in a constrained edge e, the `crep` list of indices of e is copied into the `crep` lists of the two new sub-edges created after the division of e at point p. The final pseudo codes of the insertion routines are as follows:

```
vertex* insert_point_in_edge ( p, e )
   if p is not exactly in e, project p to edge e;
   orig_crep = e->crep;
   v = new vertex created by inserting p in e according to Figure 2;
   set the crep list of the two created sub edges of e to be orig_crep;
   push the four edges of F(p) on stack;
   flip_edges ( p, stack );
   return v;

vertex* insert_point_in_face ( p, f )
   v = new vertex created by inserting p in f according to Figure 2;
   push the three edges of F(p) on stack;
   flip_edges ( p, stack );
   return v;

flip_edges ( p, stack )
   while stack not empty
      edge* e = stack.pop();
      if ( e is not constrained AND e is not Delaunay )
         f = face incident to e, which does not contain p;
         push on stack the two edges of f that are different from e;
         flip ( e );
```

The `flip_edges` routine tries to flip each edge in the stack, while the stack is not empty. The decision of actually flipping an edge e relies on two tests. The first one simply checks if e is constrained. This test is not subjected to numerical errors. The second test checks whether e is Delaunay or not. This is equivalent to determine if the circle passing through p and the endpoints of e does not contain the opposite point of p in relation to e.

Point-in-circle tests computed with a 4x4 determinant are subject to numerical errors. Different approaches have been used to achieve a robust behavior, such as arbitrary-precision arithmetic [19], or exact computation based

on floating-point numbers [17]. Our choice follows our main approach of using epsilons and floating-point arithmetic: we consider p inside the circle c only if: $distance(center(c), p) < radius(c) - epsilon$. It is worth to mention that we have faced never-ending loops during the flipping process when using only a simple determinant evaluation.

Segment insertion. The final task of the `insert_constraint` routine is to insert segment s defined by a pair of already inserted vertices. Our approach takes three steps: all edges of the CDT which are crossed by s are deleted, s is inserted in the CDT creating a new edge e, and finally the two introduced non-triangular faces on each side of e are retriangulated (see Figure 3).

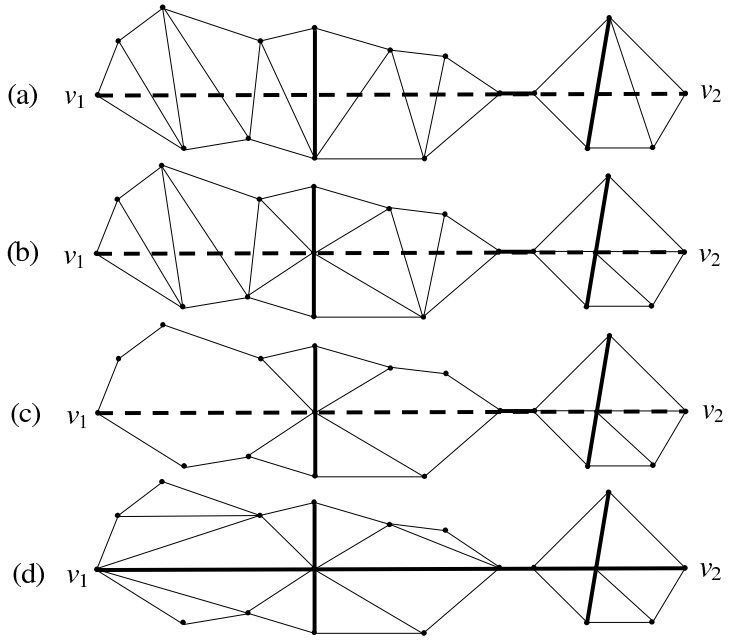

Fig. 3. a) Example of the insertion of a segment $s = \{v1, v2\}$. b) Crossing constrained edges (in bold) are subdivided and the intersection points are inserted. c) Remaining crossing edges are removed. d) Finally, s sub-segments are inserted, and the non-triangular faces are retriangulated.

Our implementation follows the approach taken by Shewchuk [17] and Anglada [1]. However, instead of deleting edges and retriangulating faces, the alternate approach presented by Bernal [2] would also be applicable: only edge flips are performed in order to make the missing segments appear. This seems to be an interesting approach and we intend, in a future work, to compare implementations of both approaches.

Let v_1 and v_2 be the two vertices which are the endpoints of the segment s to insert. The pseudo code of routine `insert_segment` can be summarized as follows :

```
insert_segment ( v1, v2, i )
   // step 1
   edge_list = all constrained edges crossed by segment {v1,v2};
   for all edges e in edge_list
       p = intersection point between e and segment {v1,v2};
       insert_point_in_edge ( p, e );
   // step 2
   edge_list = all edges crossed by segment {v1,v2};
   for all edges e in edge_list
       remove edge e from the CDT;
   // step 3
   vertex_list = all vertices crossed by segment {v1,v2};
   for all vertices v in vertex_list
       if ( v is not the last element in vertex_list )
          vs = successor vertex of v in vertex_list;
          if ( if v and vs are connected by an edge )
             e = edge connecting v and vs;
             add index i to e->crep;
          else
             e = add new edge in the CDT connecting v and vs;
             add index i to e->crep;
             retriangulate the two faces adjacent to e;
```

To decide if a vertex is crossed by s we take into account the epsilon distance. The determination of lists of elements (edges or vertices) crossed by s can be efficiently done due to the adjacency information stored in the `SymEdge` data structure. In the actual implementation, a list containing all crossed elements is determined once , and used during the entire routine `insert_segment`.

The process of retriangulating the two faces on each side of e is based on incrementally inserting interior edges that are Delaunay. We refer the reader to the work of Anglada [1], where a detailed description is given.

5 Constraint Removal

The removal process of a constraint i is based on two main steps. The first step searches for all edges representing constraint i, and setting that they no longer represent the constraint i. At this point edges may become unconstrained. The second step consists in removing those vertices that are possibly no longer used by any constraint, and removing intersection vertices that are no longer representing intersections (see Figure 4).

One vertex v of the constraint i is required in order to start the (local) search for the edges representing constraint i. Vertex v is saved when the constraint is inserted, and therefore a list is maintained associating one vertex with each constraint index. Note that v needs to be a "corner vertex" of C and not a vertex between two collinear edges representing C. This is to guarantee that v is not removed from the CDT due to the removal of another constraint sharing v. The pseudo-code of the removal process is given below:

```
remove_constraint ( i )
   // step 1
   v = one vertex of the constraint i;
   put on stack all incident edges to v representing the constraint i;
   mark all edges in stack;
   while ( stack is not empty )
      edge* e = stack.pop();
      add e to edge_list;
      push to stack all incident edges to e representing constraint i
      and which are not marked; ensure all edges in stack are marked;
   for all edges e in edge_list
      remove index i from e->crep;
   // step 2
   vertex_list = all vertices which are endpoints of edges in edge_list;
   for all vertices v in vertex_list
      ref = number of different indices referenced by the
      remaining constrained edges adjacent to v;
      n = number of remaining constrained edges adjacent to v;
      if ( n==0 )
         remove_vertex ( v );
      else if ( n==2 )
         let e1 and e2 be the remaining constrained edges adjacent to v;
         if ( e1->crep == e2->crep AND e1 is collinear to e2 )
            let v1 and v2 be the two vertices incident to e1 and e2, and
            which are different than v;
            crep = e1->crep;
            remove_vertex ( v );
            edge* e = insert_segment ( v1, v2, -1 );
            e->crep = crep;
```

Routine `remove_vertex` removes a vertex v from the CDT as well as all edges (constrained or not) which are incident to v. After removal, a non-triangular face f appears, which needs to be retriangulated. Currently we follow the approach presented by Anglada [1]. However, the alternate approach based on an *ear algorithm* [7] could also be used with possibly best performance.

When a vertex is removed to simplify two collinear edges, a call to `insert_segment` is done right after a call to `remove_vertex`. Certainly more optimized processes can be devised, however they may not be worth to imple-

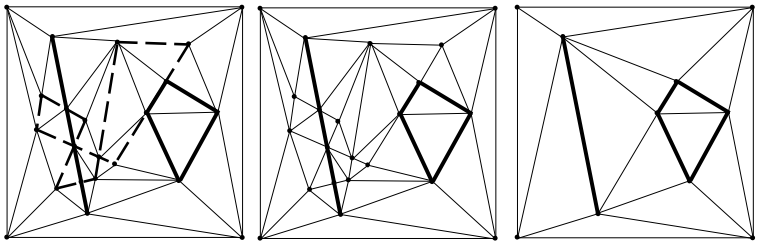

Fig. 4. The constraint C to be removed is drawn as a dashed polygonal line, and the other two intersecting constraints are drawn in bold (left). As soon as all edges of C are identified, they are set to no longer represent constraint C (middle). Finally, points which are no longer required are removed (right).

ment as collinear edges are not likely to happen very often. Figure 5 illustrates the importance of removing redundant vertices in collinear edges.

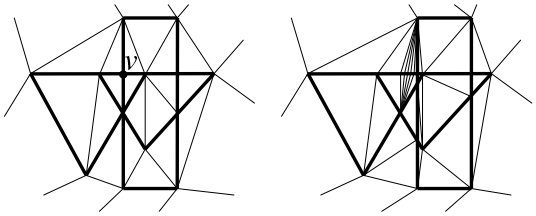

Fig. 5. The left image shows three constraints intersecting at vertex v: two triangles with one collinear edge and a rectangle R. If redundant intersection vertices were not removed, several new intersection vertices would be accumulated during a displacement of R. Such an undesirable situation (right image) is detected and fixed in step 2 of the routine `remove_constraint`.

6 Results

Using the insertion and removal routines it is possible to reinsert constraints to new positions, allowing them to move while the CDT is dynamically updated. Depending on the size of the CDT and on the size and number of constraints being manipulated, constraints can be interactively displaced in applications. This is the case with the examples showed in Figure 6.

Performance. So far, interactive graphical applications maintaining dynamic CDTs with over 10K triangles in a Pentium III 600Mhz computer have been implemented. The speed of these applications greatly depends on the size of constraints being manipulated and on the overhead to update display lists for rendering. Table 1 gives some performance data for the current version of our code.

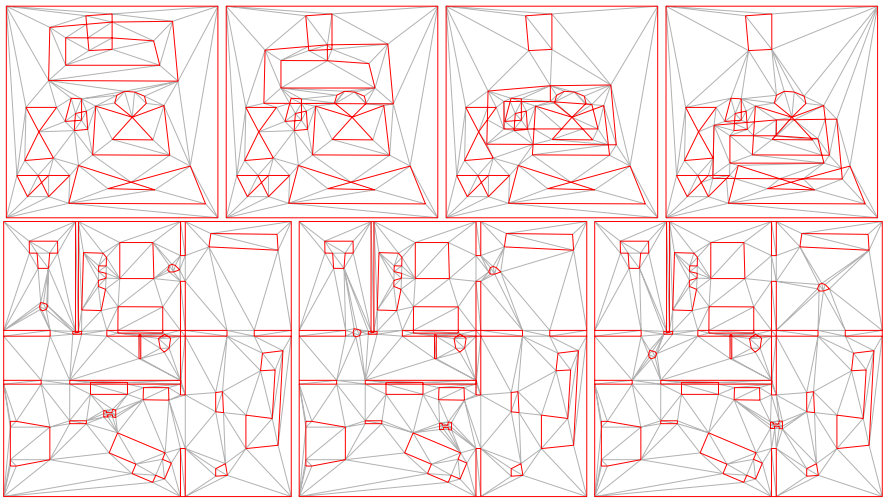

Fig. 6. The top row shows one constraint being displaced in a CDT constructed from a data set containing several intersection and overlapping cases. The bottom row shows the dynamic update of small polygons representing autonomous agents navigating in a planar environment.

Table 1. Performance values. C is a constraint selected to be displaced in each data set. For instance, in the world map data set, C is the Sicilia island. The units are: v=vertices, s=seconds, ms=milliseconds.

data set	CDT size	CDT construction	C size	C removal	C insertion
head (fig.7b)	1433 v	2.1 s	8 v	3 ms	4 ms
hexagons (fig.7a)	8332 v	3.7 s	6 v	3 ms	4 ms
world map (fig.7d)	80652 v	38.1 s	60 v	50 ms	20 ms

7 Applications

We have tested the usefulness of our triangulator with applications in several fields. Having at our disposal a CDT handling overlapping and dynamic constraints, new possibilities are open to many applications.

Visualization. In planar data visualization, data acquisition often provides a set of values associated with unorganized points in the plane. The triangulation of the set of points is often used as a way to create connections between these points [29] [21]. Then, given any point p in the plane, its value can be computed by interpolating the values associated with the vertices of the triangle containing p. In this way, values for all points in the plane can be determined, and a smooth colored visualization of the data can be obtained. This is the case, for instance, when analyzing a temperature or elevation data distribution in a planar domain.

In many cases, to enforce correct interpolation between certain values, it is important to insert constraints representing specific features on the terrain being represented. We have identified three classes of problems in planar visualization that can benefit from the capability to dynamically update constraints:

- Dynamic changes in the data set. Such cases appear whenever features being represented are subject to dynamically change. Examples are the accurate representation of borders of lakes and rivers, continent boundaries, or for data coming from floating maritime devices equipped with GPS.
- Data analysis along paths. For the design and analysis of locations where railway paths, oil tube paths, ski runs, etc, will be installed. Designed paths can be placed as constraints over the triangulation of data points in the target terrain, obtaining data interpolation exactly along the paths. The possibility to update paths dynamically allows an animated visualization of the associated data, opening new methodologies for planning such paths.
- Classification of statistical data according to several binary criteria and visualization of the resulting classes [28]. Let each criterion be defined by a piecewise linear separator which can be modified interactively. A concrete example are habitants of a city represented as points in the plane with "age" and "education level" as their coordinates. One separator could be "qualification for a certain task", another "interest in cultural activities". In order to visualize this data, we construct the corresponding CDT with the two separators as constraints. The basic color of each point is e.g. a monotonous function of the sum of its coordinates. Then the whole triangulation is colored by interpolation. Now, we associate with each separator two additional colors, indicating "criterion fulfilled/not fulfilled", and modify the basic color of each point in the triangulation by the correct one. The result is a visualization of the valid state of the dynamic classification.

Geometric modeling. Our dynamic CDT can be efficiently used to compute Boolean operation on polygons. When inserted in a CDT, all polygons (intersecting or not) are correctly triangulated. Therefore, their union is determined by the union of the triangles which are inside any of the polygons. And their intersection is determined by the union of the triangles which are contained in all polygons. Note that the boundary of such unions can be efficiently determined by simply traversing adjacent constrained edges in the triangulation.

The ability to dynamically update the position of inserted polygons allows the interactive design of planar models using the Constructive Solid Geometry (CSG) approach. The triangulation itself can be hidden from the designer, who works mainly manipulating a tree of Boolean operations among polygons.

Reconstruction. In the field of shape reconstruction from contours (see Figure 7b,c), a common approach is to consider adjacent contours as constraints in a CDT in order to specify the best strategy to link vertices [3].

The dynamic update of constraints allows, for instance, to reconstruct deformable models at the same time the data set is being captured by some measurement equipment. This is an important kind of application in the medical domain, in the implementation of training and computer-assisted surgery systems.

GIS. Constrained triangulations are important tools in the domain of geographic information systems [23]. They are used for the representation of different kind of data, which can be spatially retrieved using different kind of queries.

The possibility to dynamically update constraints is certainly an important characteristic of systems targeting the design and update of geographical databases. Figure 7d shows the CDT of a world map. In this example, our algorithm greatly simplifies the construction of the CDT as no requirements concerning the constraints are needed. For instance, we could easily handle overlapping borders between countries, intersections of rivers or lakes traversing country borders, etc.

CDTs like the one shown in Figure 7d can also serve as a base mesh for other applications, such as the analysis of ocean circulation models [20].

Path planning. The primary application that motivated this work is the determination of collision-free paths among obstacles in the plane. Practical implementations for the determination of shortest paths usually rely on visibility graphs and take quadratic time and space on the number n of obstacle vertices [25]. Using CDTs, approximate shortest paths can be derived in *O(n log n)* time (after the CDT construction). This time results from a graph search over *O(n)* triangles.

Let p_1 and p_2 be two given points in the triangulated domain. First, point location is performed in order to determine the triangles t_1 and t_2 containing points p_1 and p_2, respectively. Then, a graph search is performed over the triangulation adjacency graph, in order to determine the shortest sequence of adjacent triangles (a *channel*) connecting t_1 and t_2, and not traversing any constrained edge. Channels are determined by an A* search rooted at t_1, and switching through adjacent triangles without crossing constrained edges. This process continues until t_2 is reached. If no triangles are available to search before reaching t_2, a collision-free path joining p_1 and p_2 does not exist.

Let P be the boundary of a channel successfully determined with the procedure described above. The problem is now reduced to find the shortest path between p_1 and p_2 inside the simple polygon P. This can be done with the known *funnel algorithm* [25], which runs in linear time.

Our approach becomes particularly interesting because we are able to efficiently update the environment description when constraints move. Moreover, allowing intersecting obstacles greatly simplifies the definition of grown obstacles being used to take into account paths generated for discs and not only for points. Figures 7e and 7f illustrate some applications of our collision-free path determination approach.

8 Final Remarks

We have given extensions to known CDT algorithms in order to cope with dynamic and overlapping constraints. Constraints are identified by ids for later removal, and the required management of ids is presented for the insertion and removal procedures.

We exemplified the use of our CDT algorithms by means of several applications, and the performed tests indicate that the algorithm is very robust[1].

As future work, we intend to study the inclusion of an optimized routine to move a constraint, without the overhead of two consecutive remove and insert calls. Extensions to 3D are feasible, but hard to be robustly implemented.

Acknowledgments. The authors thank the anonymous reviewers for their helpful comments.

References

1. Anglada, M.V. (1997): An Improved Incremental Algorithm for Constructing Restricted Delaunay Triangulations. Computer & Graphics, **21(2)**, 215–223
2. Bernal, J. (1995): Inserting Line Segments into Triangulations and Tetrahedralizations. Actas de los VI Encuentros de Geometria Computacional, Universitat Politecnica de Catalunya, Barcelona, Spain
3. Boissonnat, J.-D. (1988): Shape Reconstruction from Planar Cross Sections. Computer Vision, Graphics, and Image Processing, **44**, 1–29
4. Boissonat, J.-D., Teillaud, M. (1993): On the Randomized Construction of the Delaunay Tree. Theoretical Computer Science, **112**, 339–354
5. Chew, L.P. (1987): Constrained Delaunay Triangulations. Proceedings of the Annual Symposium on Computational Geometry ACM, 215–222
6. Devillers, O. (1998): Improved incremental randomized Delaunay triangulation. Proc. of the 14th ACM Symposium on Computational Geometry, 106–115
7. Devillers, O. (1999): On Deletion in Delaunay Triangulations. Proceedings of the 15th Annual ACM Symposium on Computational Geometry, June
8. Devillers, O., Pion, S., Teillaud, M. (2001): Walking in a Triangulation. ACM Symposium on Computational Geometry
9. Fortune, S. (1987): A Sweepline Algorithm for Voronoi Diagrams. Algorithmica, **2**, 153–174
10. De Floriani, L., Puppo, A. (1992): An On-Line Algorithm for Constrained Delaunay Triangulation. Computer Vision, Graphics and Image Processing, **54**, 290–300
11. Guibas, L.J., Knuth, D.E., Sharir, M. (1992): Randomized Incremental Construction of Delaunay and Voronoi Diagrams. Algorithmica, **7**, 381–413
12. Guibas, L., Stolfi, J. (1985): Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. ACM Transactions on Graphics, **4(2)**, 75–123

[1] For information concerning the source code please contact the first author

13. Held, M. (2001): VRONI: An Engineering Approach to the Reliable and Efficient Computation of Voronoi Diagrams of Points and Line Segments. Computational Geometry Theory and Applications, **18**, 95–123
14. Mücke, E.P., Saias, I., Zhu, B. (1996): Fast Randomized Point Location Without Preprocessing in Two and Three-dimensional Delaunay Triangulations. Proc. of the Twelfth ACM Symposium on Computational Geometry, May.
15. Preparata, F.P., Shamos, M.I. (1985): Computational Geometry. Springer-Verlag, New York
16. Su, P., Drysdale, R.L.S. (1995): A Comparison of Sequential Delaunay Triangulation Algorithms. Proceedings of the ACM Eleventh Annual Symposium on Computational Geometry, June, 61–70
17. Shewchuk, J.R. (1996): Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. First ACM Workshop on Applied Computational Geometry, Philadelphia, Pennsylvania, May, 124–133
18. Sugihara, K., Iri, M., Inagaki, H., Imai, T. (2000): Topology-Oriented Implementation - An Approach to Robust Geometric Algorithms. Algorithmica **27(1)**, 5–20
19. Schirra, S., Veltkamp, R., Yvinec, M. (1999): The CGAL Reference Manual. Release 2.0 (www.cgal.org)
20. Legrand, S., Legat, V., Dellersnijder, E. (2000): Delaunay Mesh Generation for an Unstructured-Grid Ocean General Circulation Model. Ocean Modelling **2**, 17–28
21. Treinish, L. A. (1995): Visualization of Scattered Meteorological Data. IEEE Computer Graphics and Applications, **15(4)**, July, 20–26
22. Weller, F. (1998): On the Total Correctness of Lawson's Oriented Walk Algorithm. The 10th International Canadian Conference on Computational Geometry, Montral, Qubec, Canada, August, 10–12
23. De Floriani, L., Puppo, E., Magillo, P. (1999): Applications of Computational Geometry to Geographic Information Systems, Chapter 7 in Handbook of Computational Geometry, J.R. Sack, J. Urrutia (Eds), Elsevier Science, 333–388
24. Lawson, C. L. (1977): Software for C1 Surface Interpolation. In J. R. Rice (ed), Mathematical Software III, Academic Press, New York, 161–194
25. Mitchell, J.S.B. (1997): Shortest Paths and Networks. Handbook of Discrete and Computational Geometry, Discrete Mathematics & its Applications, Jacob E. Goodman and Joseph O'Rourke, ed., CRC Press, 445–466
26. Kallmann, M., Thalmann, D. (2001): Star Vertices: A Compact Representation for Planar Meshes with Adjacency Information. Journal of Graphics Tools, **6(1)**, 7–18
27. Mäntylä, M. (1988): An Introduction to Solid Modeling, Computer Science Press, Maryland
28. Bennett, K.P., Mangasarian, O.L. (1994): Multicategory discrimination via linear programming. Optimization Methods and Software, **3**, 29–39
29. Nielson, G.M. (1997): Tools for Triangulations and Tetrahedrizations and Constructing Functions Defined over Them. In G.M. Nielson, H. Hagen, H. Mueller (Eds.): Scientific Visualization. IEEE Computer Society Press, 429–525

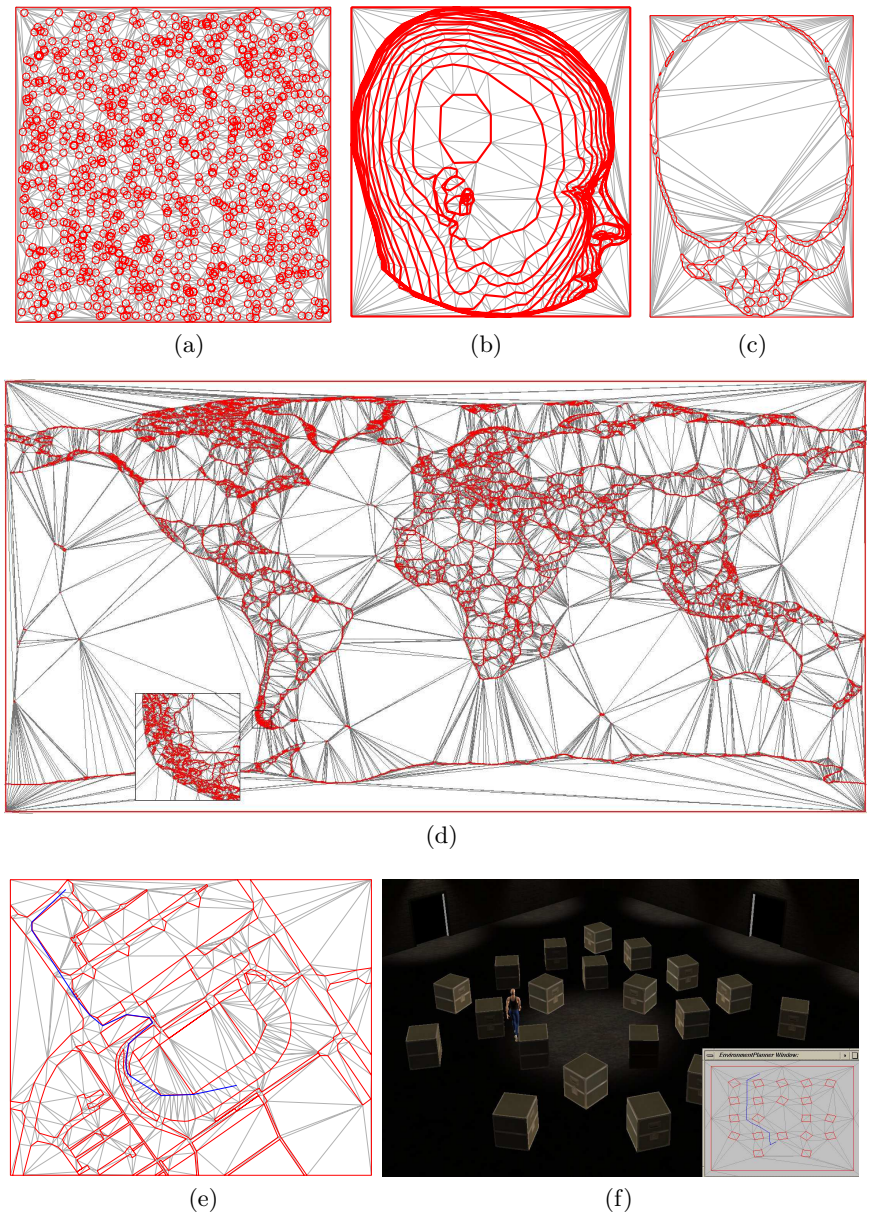

Fig. 7. a) 960 hexagonal constraints inserted randomly in a square. b) Contours scanned from a 5-month old boy's skull with premature ossification. c) One slice of the boy's skull. d) A world map with 2800 inserted constraints delimiting countries, islands and lakes. e) A path joining two points in the reconstructed town of Grangemouth, Scotland. f) Screenshot of an interactive application where a virtual human is able to walk to any selected location without colliding with boxes inside the room. Boxes are also subject to change position. Data sources: Gill Barequet web page at Tel Aviv University (b,c), DEWA/GRID center for data and information management of the United Nations Environment Programme - UNEP (d), and CROSSES IST European Project (e).