The Angular bisector network Implementation and the CGAL library

Štěpán Obdržálek
xobdrzal@fel.cvut.cz

Statement

Abstract

The CGAL (Computational Geometry Algorithms library) [1] is a C++ library, which tries to help the users with an implementation of algorithms of computational geometry. The library is well-designed, but being still in early stage of development and due to different compilers' limitations, the usage is not yet as straightforward, as one wishes.

Straight skeleton (Angular Bisector Network, ABN) of a planar polygon, which can be grasped as a modification of a planar Voronoi diagram without parabolic arcs, has been successfully used by Oliva et al. [8] as a part of a system for three dimensional reconstruction of objects from a given set of 2D contours in parallel cross sections. The Oliva's algorithm itself is used for the construction of intermediate contour layers during the reconstruction process, in order to neither create self intersecting surface nor a surface with holes.

This work first discusses the CGAL library as it was originally planned tool for a Straight Skeleton algorithm developing. Then, the idea and the implementation of the algorithm is described. The algorithm is implemented without the support of the CGAL, because the CGAL revealed to be unsuitable.

Chapter 1

Introduction

Aichholzer [3] resp. Oliva et al. [8] introduced a new internal structure for simple planar polygon called *Straight Skeleton* resp. *Angular Bisector Network, ABN*, which is made of straight line segments, which are pieces of angular bisectors of polygon edges. The *Straight Skeleton* is in structure similar to the Generalized Voronoi Diagram, but instead of parabolic arcs, whose are contained in GVD, it contains only straight line segments.

There is a lot of literature about construction of the Voronoi Diagram, eg. [9, 4] and publicly available algorithms for its computation, but the idea of *straight skeleton* is mentioned very rarely and there is no known implementation of the algorithm available in public.

The first part of this work is dedicated to the CGAL library [1]. Characteristics of the CGAL are discussed regarding the Straight Skeleton construction, and finally a conclusion is made, that there is no advantage of the CGAL usage. That is because the library does not offer any more advanced data structure nor any algorithm, which could be used to simplify the algorithm implementation.

The second part of this work describes the algorithm itself. The text was presented as a stand-alone article on Spring Conference on Computer Graphic 1998 [6].

In praxis, the algorithm is exploited in segmentation of the capillary bed of the human placenta during a 3D reconstruction from contours in parallel cross-sections.

The reconstructed surface, constructed from easily segmentable slices, is then used for prediction of the contour shape in the slices in between, where the boundaries of the object of interest are not sharp enough [5].

The reconstruction itself is based on a method, that was described by Oliva et al. [8]. This method ensures correct results for arbitrary complex 3D structures, it can properly handle all different variations between neighbouring cross-sections, such as multi-branching, holes, disconnected areas etc., and always ensures a valid topology of the resulting reconstructed shape.

Chapter 2

The CGAL

2.1 Overview

Geometric algorithms are used in many application domains. However, implementing these algorithms isn't always easy. As a result, many useful geometric algorithms haven't found their way into practice yet.

The most common problems are caused by the dissimilarity between fast floating-point arithmetic normally used in practice, and exact arithmetic over the real numbers assumed in theoretical works, the lack of explicit handling of degenerate cases in these works, and the inherent complexity of many efficient solutions. Therefore, development of a new library CGAL, the Computational Geometry Algorithms Library [1], has been started.

The CGAL library consists of a number of different parts. The elementary part of the library (called kernel) consists of primitive geometric objects (points, lines, spheres, etc.) and predicates on them (orientation, test for points, intersection tests, etc.). The other part of the library contains a number of standard geometric algorithms and data structures such as convex hull, smallest enclosing circle, and triangulation. The last part of the library consists of a support library, for example for I/O, visualization, and random generators.

The CGAL library itself is based on two other libraries, LEDA, the Library of Efficient Data types and Algorithms [2], and STL, the Standard Template Library [10]. The LEDA is primarily used to provide data types for rational numbers with exact arithmetic support, while the STL provides data containers.

2.2 Characteristics of the CGAL regarding the Straight Skeleton construction

Advantages

Robustness – Using templates for parametrisation of numeric data types achieves the possibility to compute with exact arithmetic. In such a case, algorithms implemented in CGAL are guarantied to produce correct results. On the other hand, using inexact data types, such as double or float, produces far quicker code, but there is a possibility of incorrect result given by an algorithm.

Generality – Using templates allows the user to choose data types the algorithms will work with. It's not only the selection between exact/inexact number types, but also geometric elements or data containers can be freely chosen. For example the same algorithm is provided for computation in both homogeneous and cartesian coordinates, or also in user defined.

Efficiency – Using templates allows the authors of the library to provide more versions of an algorithm. When, for some special or degenerated case of input data, a more efficient version of the algorithm exists, it can be supplied as a C++ template specialization. Also generic data structures and algorithms implemented by templates are more efficient than those implemented by generic pointers or any other common method, for the exact type is known at the compile time. Therefore no run time overhead is needed and also compiler optimizations for individual data types could take place.

Disadvantages

Portability – The CGAL library is based on templates. All algorithms and data structures are parametrised by template arguments, whose are often again types parametrised by templates. Although most of today's C++ compilers support templates, when it comes to portability issues, most of them fail to fulfil pretensions set by such a complex library as the CGAL is. Authors of the CGAL were aware of this and ensured correct compilation for some common UNIX compilers. It is performed using specific workarounds for each compiler; these are implemented by means of conditional compilation. Nevertheless compilation with an unsupported compiler is either impossible or requires too big effort.

Compilation times – Regarding the portability, the choice of HW platform for development of applications is limited. On such available platform (SGI INDY), simple sample application, like a computation of a polygon difference, took nearly 20 minutes to compile.

2.3 Conclusion over the CGAL

The primary idea was to use the CGAL to help with an implementation of an algorithm, which constructs the Straight Skeleton. But it revealed, that the CGAL is of no use for several reasons. First, the CGAL doesn't contain any advanced data structure nor any algorithm suitable for the algorithm implementation. Next, the implemented algorithm heavily uses goniometric functions, therefore the usage of exact arithmetic is also impossible, for it supports only rational numbers. Containers used within the CGAL are taken over the STL [10], so it's enough to use only the STL (which is a cross-platform standard). As a result, the only part of CGAL useful for the algorithm implementation is the kernel (elementary primitives), but the overhead caused by the CGAL usage would be too high. Furthermore, the present implementation of the algorithm is compileable at least under Watcom C++, Microsoft Visual C++, Irix CC and GNU C++. Such a portability would be with the CGAL impossible.

As a consequence, the decision was made not to use CGAL, and to implement the algorithm from scratch. The supervisor of the work agreed to this change.

Chapter 3

Straight skeleton

Skeleton like structure is often used for the description of basic topological characteristics of a 2D object. In the image processing and the computer vision fields, *skeleton*, informally defined as a set of points located in the centers of such circles included in the object, that they touch the object's boundary in at least two distinct points, is used. Such a structure is the well-known Generalized Voronoi Diagram.

The *Straight Skeleton* differs, in general, from the GVD. If the given polygon is convex, then both structures are identical. Otherwise, the GVD contains parabolically curved segments in the neighborhood of reflex vertices. Parabolically curved segments are avoided in the *Straight skeleton*. Both structures reflect the shape of a polygon in a similar manner, however the *Straight Skeleton* is more sensible to local changes of the given shape, for adding a reflex vertex with very small external angle may change the skeleton structure completely.

3.1 Straight Skeleton Computation

The principle of the algorithm can be imagined as a construction of a roof with constant slope from given shape of the walls [3, 7]. It can be done by a sweep algorithm, which simulates cutting of the roof by parallel planes and checks local changes of the polygonal base in the cross sections. In a 2D view, it appears as shrinking of the polygon. The polygon edges are moving in constant speed inward the polygon and they are changing their lengths. The polygon vertices move along the angular bisectors as long as the polygon does not change its topology. Aichholzer [3] described two possible types of changes:

- *Edge event*: An edge shrinks to zero, making its neighboring edges adjacent.

- *Split event*: A reflex vertex runs to this edge and splits it, thus split the whole polygon. New adjacencies occur between the split edge and each of the two edges incident to the reflex vertex.

The straight skeleton $S(P)$ of the polygon P is defined as a union of pieces of angular bisectors traced out by polygon *vertices* during the shrinking process. Each edge (a straight line segment) e sweeps out certain area which we call *face* of e. Bisector pieces are called *arcs*, and their endpoints which are not vertices of P are called *nodes* of $S(P)$. An example is in fig. 3.1, where the straight skeleton arcs are drawn by a thick lines and shape of intermediate levels by a thin line. For more details refer to [3]. The algorithm handles vertices and nodes in the same way.

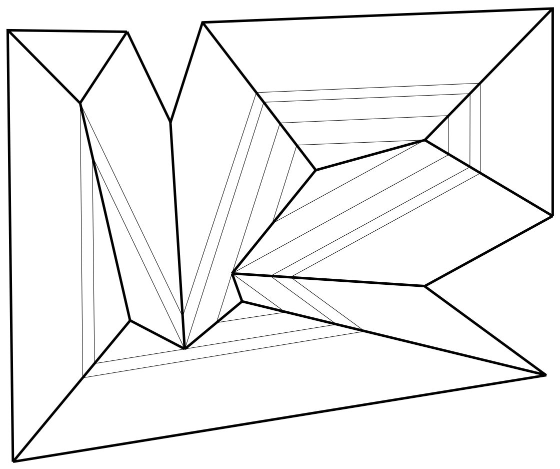


Figure 3.1: Polygon hierarchy (thin line) and straight skeleton (thick line)

The method for the straight skeleton computation will be described in two steps. First for a convex polygon, then for a non-convex one. The algorithm applies the principle of the roof construction by sweeping, but instead of constructing the polygonal base in the cross sections, it manages only the pointers to the edges of the original polygon.

The basic data structure used by the algorithm is a *set of circular lists of active vertices (SLAV).* This structure stores a loop of vertices for outer boundary and for all holes and sub-polygons created during the straight skeleton computation. In the case of convex polygon, it always contains only one list (LAV). In the case of a simple non-convex polygon, it stores a list for every sub-polygon (as described later) and in the case of polygons with holes also a list for each hole.

All vertices in the SLAV have references to both neighbors (vertices of the polygon) in the circular lists (LAV) stored in the SLAV.

3.1.1 Convex Polygon Skeleton Computation

Given a simple convex polygon P, only the *edge events* occur and the straight skeleton $S(P)$ is computed in the following steps (We suppose the polygon vertices and edges are oriented counter-clockwise and the polygon interior is on the left-hand side of its boundary):

1. Initialization:
    (a) Organize given vertices $V_1, V_2 \ldots, V_n$ into one *double connected circular list of active vertices* (LAV) and store it in the SLAV. The vertices in LAV are all active (not processed) at this moment.

    (b) for each vertex V_i in LAV add the pointers to two incident edges $e_{i-1} = V_{i-1}V_i$ and $e_i = V_iV_{i+1}$, and compute the vertex angle bisector (ray) b_i,

    (c) for each vertex V_i compute the nearer intersection of the bisector b_i with adjacent vertex bisectors b_{i-1} and b_{i+1} starting at the neighboring vertices V_{i-1}, V_{i+1} and (if the intersection exists) store it into a priority queue according to the distance to the line $L(e_i)$ supporting the edge e_i. For each intersection point I_i store also two pointers to the vertices V_a, V_b, these vertices are two the origins of bisectors which have created the intersection point I_i. They are necessary for the identification of appropriate edges (e_a and e_b in fig. 3.2) during the bisector computation in later steps of the algorithm.

2. While the priority queue with the intersection points is not empty do:
    (a) Pop the intersection point I from the front of the priority queue,

    (b) if the vertices/nodes V_a and V_b, pointed by I, are marked as processed then continue on the step 2,
    else the edge e between the vertices/nodes V_a, V_b shrinks to zero length (*edge event* - this edge is in fig. 3.2 marked by a cross),

    (c) if the predecessor of the predecessor of V_a (according to the LAV) is equal to V_b (peak of the roof)
    then output three straight skeleton arcs V_aI, V_bI and V_cI, where V_c is both the predecessor of V_a and the successor of V_b in the LAV, mark V_a, V_b and V_c as processed, and continue on the step 2,

    (d) output two skeleton arcs of the straight skeleton V_aI and V_bI,

    (e) modify the list of active vertices/nodes (See figure 3.2 for details):

- Mark the vertices/nodes V_a, V_b (pointed to by I) as processed (marked by a cross in fig. 3.2),
- create a new node V with the coordinates of the intersection I (a square mark in fig. 3.2),
- insert this new node V into the LAV. That means connect it with the predecessor of V_a and the successor of V_b in the LAV (thick arrows in the fig. 3.2),
- link the new node V with appropriate edges e_a and e_b (pointed to by the vertices V_a and V_b),

(f) for the new node V, created from I:

- compute a new angle bisector b between the line segments e_a and e_b,
- compute the intersections of this bisector with the bisectors starting from the neighboring vertices in the LAV in the same way as in the step 1c,
- store the nearer intersection (if it exists) to the priority queue.

As can be seen in the steps 1c and 2f, there are duplicities among the intersection points in the priority queue. The algorithm always computes one intersection for one vertex. Step 2b removes these duplicities.

3.1.2 Non-convex Polygon Skeleton Computation

The principle of the method for the straight skeleton computation is in the case of non-convex polygons similar. New intersection points (one for each vertex/node in all LAVs in the SLAV) are after their computation stored into the priority queue as in the section 3.1.1, but they have a new attribute indicating their event type: *edge event* or *split event*.

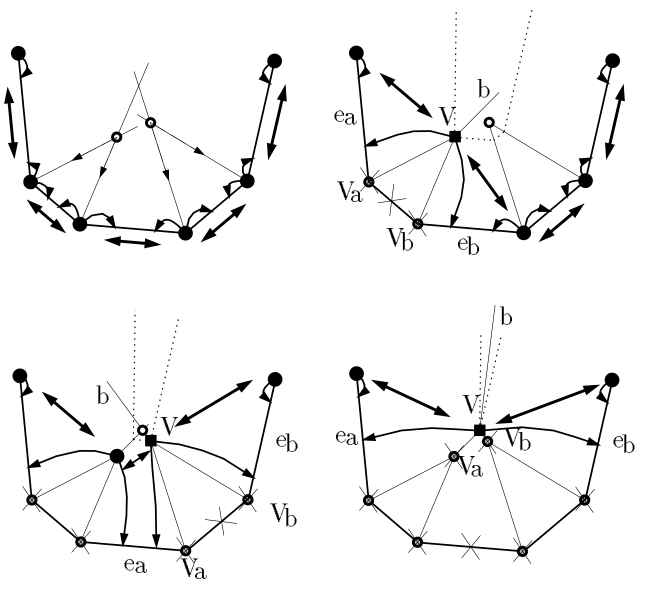
At first, let's discuss the circumstances which we have to take into account.

Presence of a reflex vertex may (but may not) lead into a polygon splitting (see fig. 3.3). In case of the point A a standard *edge event* occurs which is handled the same way as the edge event of non-reflex vertex (as described in the section 3.1.1). Let's concentrate on the case of the point B, where a *split event* occurs.

The first task is to determine the coordinates of the point B. Then we will discuss the insertion of B into the appropriate LAV in the SLAV and a special case of a multiple split edge. The name B represents the coordinates of the tested intersection point and also the coordinates of the new vertex later inserted into LAV. They will distinguished in the algorithm.

Determination of the coordinates of the point B

Point B can be characterized as having the same perpendicular distance to the straight line supporting the edge "opposite" to the vertex V and from both



● = active vertices/nodes in the list (LAV)

⊗ = vertices/nodes marked as processed

○ = intersection points in the priority queue

■ = the current node V

↔ = pointers in the list of active vertices (LAV)

↷ = pointer to the appropriate edge for bisector computation

Figure 3.2: Initialization and the first three steps of the skeleton algorithm for convex vertices

straight lines supporting the edges starting at the vertex V. We have to find such an "opposite" edge.

All edges e of the original polygon are traversed and tested whether they can be the "opposite" edges. Unfortunately a simple test of the intersection between a bisector starting at V and the currently tested edge cannot be used (see fig. 3.4b). It's necessary to test the intersection with the whole line supporting the edge e_i and to test whether the candidate point B_i lays in the area limited by the currently tested edge e_i and by the bisectors b_i and b_{i+1} leading from the vertices at both ends of this edge (see fig. 3.4a,b).

Simple intersection test between the bisector starting at V and the (whole) line supporting the currently tested edge e_i rejects the edges laying "behind"

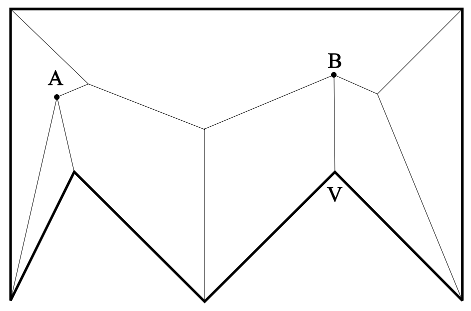


Figure 3.3: Reflex vertex can yield to both cases: an edge event (point A) or a split event (point B)

the vertex V. Then the coordinates of the candidate point B_i are computed as the intersection between the bisector at V and the axis of the angle between one of the edges starting at V and the line supporting the tested edge e_i (see fig. 3.4). Simple check should be performed to properly handle the case when one of the edges starting at V is parallel to e_i.

The resulting point B is selected from all the candidates B_i as the one nearest to the vertex V.

Managing the LAV in the case of a split event

When the intersection B (called I on Fig. 3.6) of the *split event* type is processed, it is necessary to split the appropriate polygon into two parts. Splitting of the polygon also implies splitting of the appropriate LAV into two parts and implies insertion of two new nodes V_1 and V_2 with the coordinates taken from I into them – each copy into one LAV (see details in Fig. 3.6 and in the second algorithm).

Both vertices V_1 and V_2 points to the same split edge e_i and share it.

A special case – an edge split more than once

As mentioned before, the algorithm works with the original edges and doesn't construct the intermediate roof shapes in the cross-sections. This yields the situation when one original edge is shared by more than one intermediate polygon after the previous polygon split. In a case when one edge is already split and the next edge event for this edge occurs, we have to choose the opposite edge end-points correctly (resp. the vertices/nodes which are active in the current roof construction level as the points X and Y in fig. 3.5 when processing the vertex

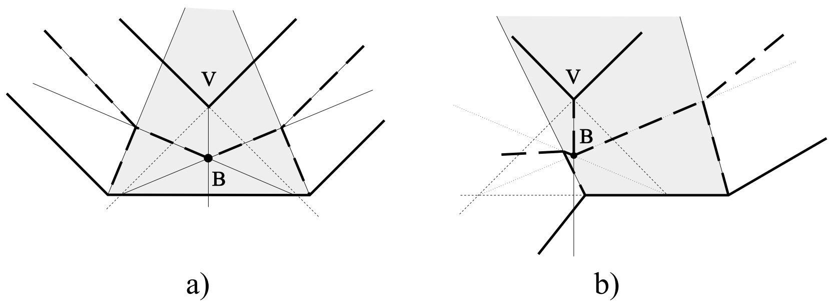


Figure 3.4: Intersection point computation for a reflex vertex V which yields a split event (point B)

In the case of the sub-edge SY (a part of the edge $e_i = ZY$), the real end point of the original edge e_i is the point Z, but we search for the point X, which is necessary for the correct interconnection of pointers when the tow new nodes with the coordinates of B are inserted into the SLAV. We mention two methods how to determine X.

One possible solution is to really split the border edge and to create a new auxiliary vertex S. It would remove the cause of the problem, because no edge will be split more than once and the split edge end-points determination is trivial. It implies that the insertion of the two new nodes into the SLAV is also trivial, as we know exactly in which of the sub-polygons (LAVs) it has to be interconnected. It solves the problem with the shared edge e_i but such a vertex S has to be handled in a special way, for it is not included in the skeleton.

Our solution is based on the idea to store only the nodes which are present in the straight skeleton and on the fact, that the references to the split edges are stored in all of the sub-polygons, which share these edges. It reveals during the traversal of all sub-polygon LAVs in multiple hits of the split edge, each time for each sub-polygon sharing it.

For instance the two subpolygons in fig. 3.5 share a reference to the edge e_i. During the processing of the intersection B associated with the vertex V, the polygon $XMVNY$ is split into two parts XMB and BNY and the vertex V is marked as processed.

All this is done for the reason to correctly connect V_1 and V_2 on created on B's coordinates between Y and X and not between original endpoints Y and Z. During the SLAV traversal the correct part of the split edge e_i is selected by means of the same criterion as described before, i.e. the candidate point B have to lay rightwards to the traversed vertex's (Y) bisector and leftwards to the bisector of the traversed vertex's successor (X) in the LAV.

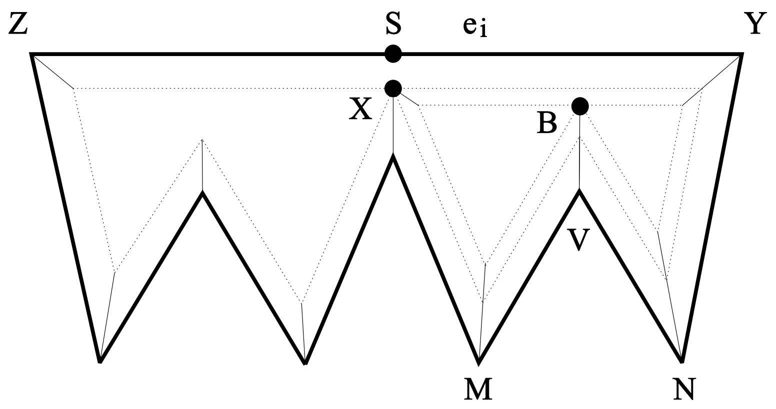


Figure 3.5: Correct selection of the active vertex/node X for already split edge

The algorithm for simple non-convex polygons

Algorithm for non-convex polygons is in the principle similar to the algorithm described in the section 3.1.1. As an extension, it handles the intersections that may generate polygon splits (split events):

1. Initialization
   (a) Generate one LAV as in the convex case, store it in SLAV,
   (b) compute the vertex bisectors as in the convex case,
   (c) compute intersections with the bisectors from the previous and the following vertices as in the convex case and for reflex vertices compute also the intersections appropriate to the opposite edges (point B in figs. 3.3 and 3.4). Store the nearest intersection point I of these three into the priority queue. In addition store also the type of the intersection point (edge event or split event).

2. While the priority queue is not empty do:
   (a) Pop the lowest intersection I from queue as in the convex case. If the type of I is the *edge event*,
   then process steps 2b to 2g of the algorithm for convex polygons,
   else (*split event*) continue within this algorithm,
   (b) if the intersection point points to already processed vertex/node continue on step 2 as in the convex case,
   (c) output one arc VI of the straight skeleton, where vertex/node V is the one pointed to by the intersection point I. Intersections of the

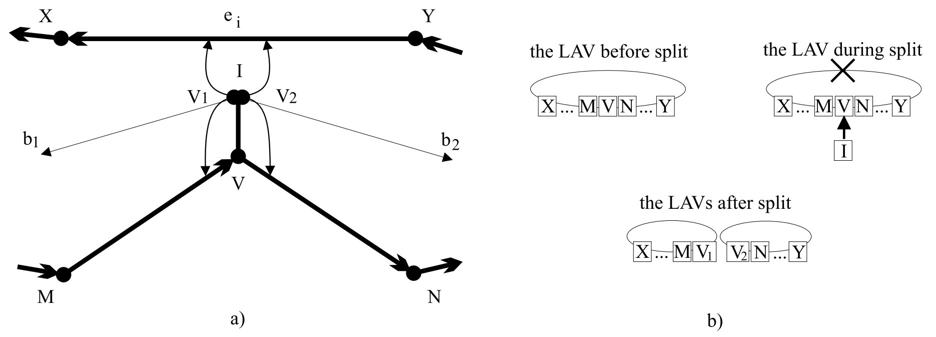


Figure 3.6: Split event at the reflex vertex: a) managing the edge pointers, b) LAV split caused by the new nodes V_1 and V_2

split event type always point to exactly one vertex in LAV/SLAV (compare with the convex version of the algorithm, where the *edge event* type intersections point to two vertices/nodes V_a and V_B),

(d) modify the set of lists of active vertices/nodes (SLAV):
   - Mark the vertex/node V (pointed to by I) as processed,
   - create two new nodes V_1 and V_2 with the same coordinates as the intersection point I,
   - search for the opposite edge in SLAV (sequentially),
   - insert both new nodes into the SLAV (break one LAV into two parts — see fig. 3.6b for an example). Vertex V_1 will be interconnected between the predecessor of V (M on fig. 3.6) and the vertex/node which is an end point of the opposite edge (X on fig. 3.6). V_2 will be connected between the successor of V (N on fig. 3.6) and the vertex/node which is a starting point of the opposite edge (Y on fig. 3.6). This step actually splits the polygon into two parts (as discussed before in this section).
   - if one of the new LAVs consists of only two vertices, don't store this LAV into SLAV, but output one skeleton arc instead (see the discussion earlier in this section),
   - link the new nodes V_1 and V_2 with the appropriate edges (see fig. 3.6a).

(e) for both nodes V_1 and V_2:
   - compute new angle bisectors between the edges linked to them in step 2f,
   - compute the intersections of these bisectors with bisectors starting at their neighboring vertices according to the LAVs (e.g. at

points N and Y for node V_2 and at M and X for node V_1 in fig. 3.6a), the same way as in step 1c. New intersection points of both types may occur,

- store the nearest intersections (one for V_1, one for V_2) into the priority queue.

3.1.3 Straight Skeleton Computation for Polygons with Holes

The algorithm can also handle the polygons with holes as long as they have appropriate orientation. That means the polygon interior lays leftwards to all of the edges, vertices on the outer boundary are counter-clockwise ordered and the vertices on the holes are clockwise ordered. It results in more cycles of vertices (LAVs) in the SLAV in the step 1a of the algorithm.

An example of the straight skeleton of a polygon with one hole is in fig. 3.7.

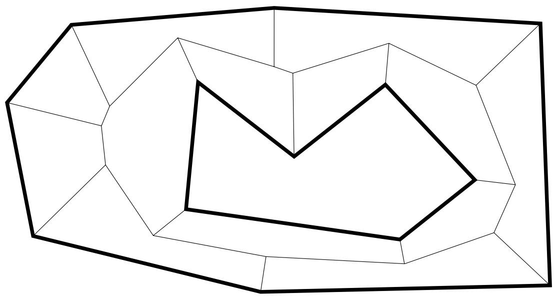


Figure 3.7: An example of the straight skeleton of a polygon with a hole

3.2 Implementation

The algorithm is implemented in C++ and is platform independent. In adition to standard libraries, it utilizes only the STL [10], which is quite multi-platform standard now. The implementation was successfully tested with some common compilers, Watcom C++, Microsoft Visual C++, Irix CC and GNU C++.

The source code is divided into two files. First of them, PRIM.CPP, provides some basic primitives, such as Point and Ray classes, and also some basic operations on them, e.g. distance or intersection computation.

The main burden lays on the SKELET.CPP source code. At first, it declares some datatypes the algorithm uses, then execution routines follows. Below is described the interface required to use the algorithm:

```
struct Point
{
  Number x, y;
};
```

This is how the point is represented. The type `Number` is currently based on the `double` type and exact arithmetic is simulated by inexact comparison:

```
struct Number
{
  Number (const double x = 0.0) : n (x) { };
  operator const double& (void) const { return n; }
  operator       double& (void)       { return n; }
  operator == (const Number &x) const;       // comparison with tolerance
  operator != (const Number &x) const { return !(*this == x); }
  operator <= (const Number &x) const { return n < x.n || *this == x; }
  operator >= (const Number &x) const { return n > x.n || *this == x; }
  operator <  (const Number &x) const { return n < x.n && *this != x; }
  operator >  (const Number &x) const { return n > x.n && *this != x; }

  operator == (const double x) const { return *this == Number (x); }
  operator != (const double x) const { return *this != Number (x); }
  operator <= (const double x) const { return *this <= Number (x); }
  operator >= (const double x) const { return *this >= Number (x); }
  operator <  (const double x) const { return *this <  Number (x); }
  operator >  (const double x) const { return *this >  Number (x); }

  double n;
};
```

However the type `Number` could be of any type as long as it has well-defined standard mathematic operations and also that basic goniometric functions are defined for it.

```
struct Ray
{
  Point origin;
  Number angle;          // in radians
};
```

This is the representation of a ray (semi-finite line). Representation with angle explicitly expressed makes the algorithm simpler, but requires the usage of goniometric functions.

```
typedef vector <Point>   Contour;
typedef vector <Contour> ContourVector;
```

Each contour (outer boundary or a hole) is represented as a STL vector of points. The input to the algorithm is a vector of such contours (`ContourVector`). It's presumed, that the outer boundary is oriented counter-clockwise, and that holes are oriented clockwise. However, no check is performed to ensure the orientation is correct.

```
struct Vertex
{
  Point point;         // vertex's coordinates
  Ray axis;            // vertex's angle bisector

  Ray leftLine;        // Rays supporting appropriate original
  Ray rightLine;       // polygon edges

  Vertex *leftVertex;  // 2 processed vertices, (*this) one
  Vertex *rightVertex; // was created during their processing

  Vertex *nextVertex,  // links within the LAV
  Vertex *prevVertex;

  bool done;           // processed (non-active) flag
  int ID;              // unique identification number
};
```

`Vertex` is the structure that represents both vertices on the boundary and nodes created by the algorithm during the skeleton construction. Internally, vertices/nodes are stored in a STL list, and the SLAV is managed by pointers within this list.

```
struct SkeletonLine
{
  struct SkeletonPoint
  {
    const Vertex *vertex;        // contains point's coordinates
    SkeletonLine *left, *right;  // two wings for each end-point
  } lower, higher;               // 2 end-points in each segment
  int ID;                        // unique identification number
};
```

The result of the algorithm is given in *winged edge* representation. The `SkeletonLine` represents such an edge.

```
class Skeleton : public list <SkeletonLine>
{
  ...
};
```

The resulting skeleton is nothing more than a STL list of individual winged edges.

```
Skeleton &makeSkeleton (ContourVector &contours);
```

Finally, this is the prototype of the function, which creates the skeleton.

Not all data structures was presented, as the resting are used only internally and are not exported by the algorithm implementation. These include segments, intersections, intersection queue and vertex list representations.

3.3 The complexity of the algorithm

In this section, both the storage and the time complexity will be discussed. Let n be the total number of vertices of the input polygon, m the number of reflex ones, let p be a number of local peaks of the roof shape (always equal to one for convex polygons), and finally let t be a total number of both vertices and nodes the algorithm will handle.

First step to state the complexities is to determine the total number of vertices and nodes (t), that will ever be processed.

At the beginning of the algorithm, the number of active vertices is equal to n. Processing one intersection of the *Edge event* type decreases the number of active vertices by one, for two vertices/nodes are marked as processed (removed from SLAV, made inactive), and only one new node is created. On the other hand, processing intersection of the *Split event* type increases the number of active vertices by one, for only one vertex/node is marked as processed and two new nodes are created. These two nodes are guarantied to be non-reflex (see nodes V_1 and V_2 on fig. 3.6). Finally, if a LAV contains only three vertices/nodes (peak of the roof), these three are marked as processed and no new node is created.

For a convex polygon, t can be expressed as

$$t = n + n - 3 = 2n - 3,$$

because for the n original vertices, $(n-3)$ times the *Edge event* occurs and once a peak of roof is encountered.

For non-convex polygon, there is only $(n-m)$ non-reflex vertices, and there is p peaks of the roof. As an addition to convex case, m reflex vertices may (in worst case) produce $2m$ new nodes during *Split events*, and these new nodes would generate additional $2m$ nodes when they will be processed. So the equation for t in non-convex case is:

$$t = (n-m) + (n-m) - 3p + m + 2m + 2m = 2n + 3m - 3p.$$

This equation respects the worst case, when every of the m reflex vertices causes the *Split Event*. But, if we let s be the number of reflex vertices, that really causes the *Split Event*, then the expression

$$t = 2n + 3s - 3p$$

will be valid in all cases. Finally, let's consider the correspondence between number of *Split Events* (s) and the number of local peaks (p). It's clear, that each *Split event* splits one polygonal shape into two parts, thus increases the total number of local peaks by one. If no *Split Event* occurs, only one peak exist. So the correspondence is simple:

$$p = s\ + 1.$$

As a result, the total number of vertices and nodes is expressed as:

$$t = 2n - 3,$$

so it is linear to the number of input vertices n.

Memory requirements of the algorithm include two data structures. The first is the list of all vertices and nodes the algorithm will ever use. There is exactly t of them. The second structure is the priority queue of intersections. It stores one intersection per one active vertex/node, so the total number of intersections that may be in the queue at once is at most t. It makes the storage complexity linear to the number of input vertices.

Now, let's consider the time complexity. It should be stated here, that the `push` and `pop` operations over the priority queue are performed in logarithmic time.

The time complexity of the initialization part of the algorithm is $n \log(n)$. That's because for each input vertex only constant-time operations are performed with the only exception – pushing the computed intersection point into the priority queue, it takes the logarithmic time.

The second part of the algorithm runs t times. Each time the `pop` operation is performed (logarithmic time) and then, for the peaks of the roof or for the vertices processed earlier, the constant-time code follows. For the *Edge Events* logarithmic-time code follows, because there is a new intersection pushed into the queue. The worst case is the *Split Event*, when the sequential search for opposite edge is required; it has linear complexity.

So the final time complexity is $n \log(n) + t \log(t) + t.s$, the asymptotical time complexity is then $O(n \log n + n.m)$ or simply $O(n^2)$.

3.4 The Results

The implemented algorithm for the construction of a straight skeleton of non-convex polygons with holes runs in $O(nm + n \log n)$ time, where n denotes the total number of polygon vertices and m the number of reflex ones. The algorithms handles convex, non-convex polygons and polygons with holes correctly.

Handling the split events takes time $O(n \log n)$ with $O(n)$ storage in the priority queue, but it is not the significant part of the whole complexity. The most time consuming task is the computation of the split events — that means handling m ($m < n$) reflex vertices. That results in the global time complexity of the presented algorithm $O(n^2)$ with $O(n)$ storage. That is a similar result as in [8].

Measurements of the time complexity of the algorithm for polygons with increasing number of vertices has been performed. The execution times for different non-convex polygons (as shown in fig. 3.8 and table 3.1) confirm the theoretical presumptions of the quadratic time complexity.

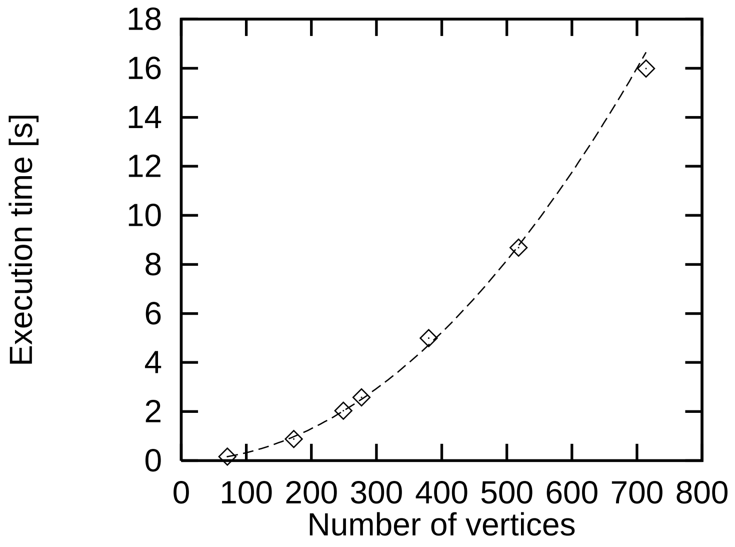


Figure 3.8: Results of the tests for polygons with different number of vertices

No. of vertices	**Execution time**
71	0.16 s
173	0.88 s
249	2.03 s
277	2.58 s
380	5.00 s
518	8.68 s
714	15.99 s

Table 3.1: Results of the tests for polygons with different number of vertices

Chapter 4

Conclusion

The task was to implement an algorithm for the Straight Skeleton construction using the CGAL library. But, as discussed in the first part of this text, the CGAL revealed to be unsuitable for the algorithm implementation. The most important reason is that the data structures and the algorithms the CGAL provides do not fit the needs the Straight Skeleton construction has. Thus the algorithm was implemented without the use of the CGAL, only standard C++ libraries (including STL [10]) was utilised.

The task of this work was fulfiled, the implemented algorithm works properly for arbitrary polygons, and runs with $O(n^2)$ time complexity and $O(n)$ storage complexity.

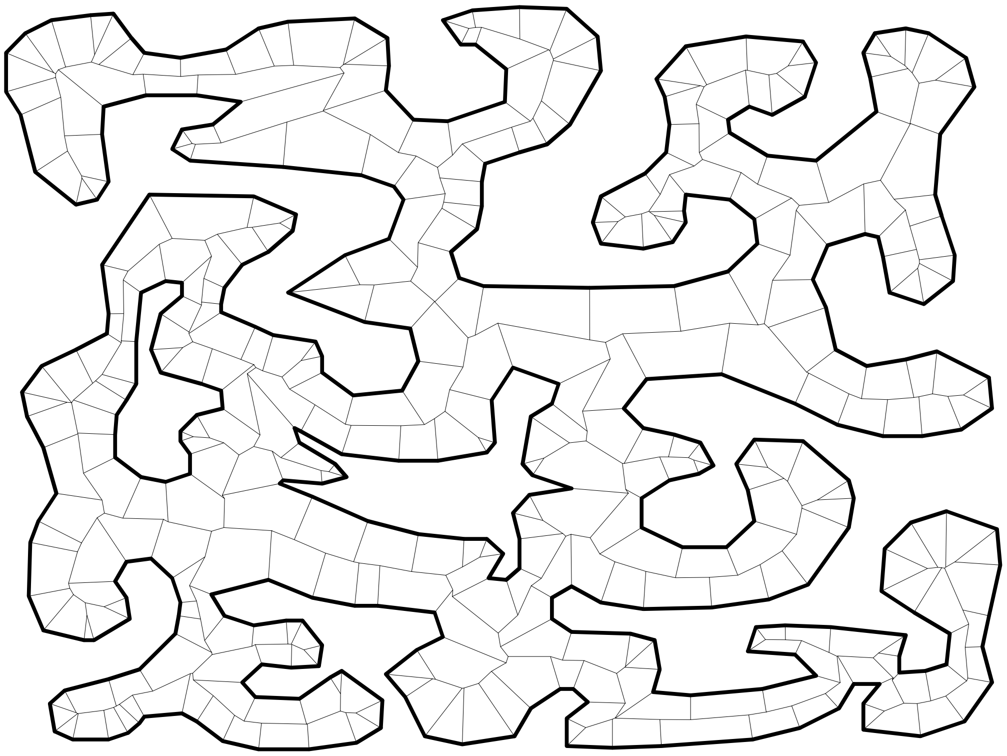


Figure 4.1: A more complex example of a Straight skeleton

Bibliography

[1] Computational geometry algorithms library. http://www.cs.uu.nl/CGAL/.

[2] Leda - library of efficient data types and algorithms. http://www.mpi–sb.mpg.de/LEDA/leda.html.

[3] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science, http://www.iicm.edu/jucs_1_12, Institute for Image Processing and Computer Supported New Media*, 1(12):752–761, 1995.

[4] M. de Berg, M. vam Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications.* Springer Verlag Berlin Heidelberg New York, 1997.

[5] P. Felkel. 3D Reconstruction from Cross Sections by means of Contour Tiling. In A. Strejc, editor, *Workshop '98*, volume I, pages 241–242. Czech Technical University Publishing House, Prague, Czech Republic, Feb. 3.–5. 1998.

[6] P. Felkel and Š. Obdržálek. Straight skeleton computation. In L. Szirmay-Kalos, editor, *Spring Conference on Computer Graphics*, pages 210–218, Budmerice, Slovakia, Apr. 23–25 1998.

[7] P. Felkel and J. Žára. The Roof Construction Problem. In A. Strejc, editor, *Proceedings of CTU Workshop '93*, volume I - Informatics & Cybernetics, pages 85–86. CTU-Publishing House, Prague, Jan. 18–21, 1993.

[8] J.-M. Oliva, M. Perrin, and S. Coquillart. 3D Reconstruction of Complex Polyhedral Shapes from Contours using a Simplified Generalized Voronoï Diagram. *Computer Graphics Forum*, 15(3):C–397–C–408, 1996.

[9] F. P. Preparata and M. I. Shamos. *Computational Geometry – An Introduction.* Springer-Verlag, New york, 1985.

[10] A. Stephanov. STL - The Standard Template Library.

List of Figures

Contents