ELSEVIER

Computer-Aided Design 37 (2005) 1027–1038

COMPUTER-AIDED DESIGN

www.elsevier.com/locate/cad

An efficient sweep-line Delaunay triangulation algorithm

Borut Žalik*

Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, SI-2000 Maribor, Slovenia

Received 15 April 2004; received in revised form 12 October 2004; accepted 14 October 2004

Abstract

This paper introduces a new algorithm for constructing a 2D Delaunay triangulation. It is based on a sweep-line paradigm, which is combined with a local optimization criterion—a characteristic of incremental insertion algorithms. The sweep-line status is represented by a so-called advancing front, which is implemented as a hash-table. Heuristics have been introduced to prevent the construction of tiny triangles, which would probably be legalized. This algorithm has been compared with other popular Delaunay algorithms and it is the fastest algorithm among them. In addition, this algorithm does not use a lot of memory for supporting data structure, it is easy to understand and simple to implement.

Keywords: Computational geometry; Delaunay triangulation; Sweep-line paradigm

1. Introduction

The construction of a maximally connected planar graph on a given set of points is an old engineering problem. It is usually considered as a triangulation. The most investigated triangulation is so-called Delaunay triangulation. It optimises the minimal interior angle of constructed triangles, which makes it convenient for different engineering applications. Several approaches have been proposed for its construction. A survey with empirical comparison can be found in [1], where the algorithms are classified into five groups:

- incremental insertion algorithms [2–6],
- gift-wrapping algorithms [7],
- divide and conquer algorithms [8,9],
- convex hull based algorithms [10], and
- sweep-line algorithms [11].

This paper presents a new algorithm for constructing Delaunay triangulation. It is based on the sweep-line paradigm combined with the Lawson's recursive local optimization procedure. As confirmed by experiments, this proposed algorithm is fast, it is easy to understand, practically independent on the distribution of input points, the supporting data structure is simple and easy to implement, all of which makes it superior to the algorithms normally available.

This paper is organised as follows. Section 2 introduces a necessary vocabulary and gives an overview of the existing techniques. In Section 3, the algorithm is explained in detail. Section 4 gives results and comparisons. The algorithm is summarized in Section 5.

2. Background

2.1. Delaunay triangulation

Let $\mathcal{S}$ be a set of non-collinear points in the plane. Triangulation $\mathcal{T}(\mathcal{S})$ is the maximal division of a plane into a set of triangles with the restriction that each triangle edge, except those defining the convex hull of $\mathcal{S}$, is shared by two adjacent triangles. A Delaunay triangulation $\mathcal{DT}(\mathcal{S})$ is a unique triangulation constructed on $\mathcal{S}$ such that a circumcircle of any triangle does not contain any other point from $\mathcal{S}$. This condition, frequently also considered as an *empty circle property*, optimizes triangulation according to the minimal inner angle of the triangles. Triangles from $\mathcal{DT}(\mathcal{S})$ are

* Tel.: +386 2 220 7471; fax: +386 2 251 1178
E-mail address: zalik@uni-mb.si.

0010-4485//$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cad.2004.10.004

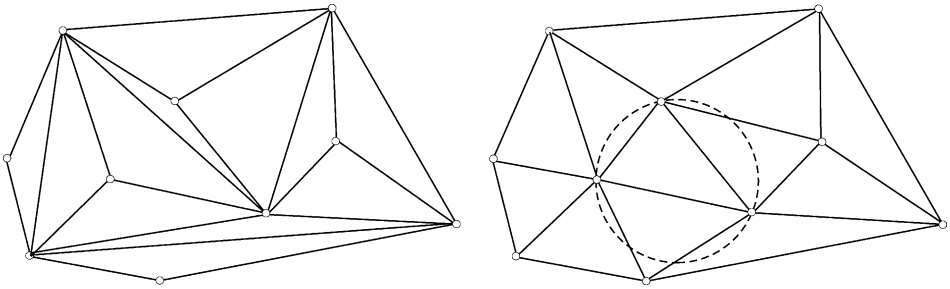


Fig. 1. Non-Delaunay (a) and Delaunay (b) triangulation.

considered as Delaunay triangles (or legal triangles) and their circumcircles as Delaunay circles. Fig. 1 shows an example of non-Delaunay and Delaunay triangulation.

In 1977, Lawson [12] showed that any triangulation $\mathcal{T}(\mathcal{S})$ can be transformed into $\mathcal{DT}(\mathcal{S})$ by applying the empty circle test on all pairs of triangles. If the empty circle property is violated, the common edge of the two triangles are swapped (see Fig. 2). The Delaunay triangulation is obtained when this procedure is recursively applied on all inner edges of the triangulation. Let us observe the situation in Fig. 2a. Triangle $\triangle_{i,k,j}$ contains vertex v_l of neighbouring triangle $\triangle_{j,l,i}$. Two new triangles $\triangle_{l,i,k}$ and $\triangle_{l,k,j}$ are obtained after swapping the common edge. The swapped edge $\overline{v_k v_l}$ becomes legal (Fig. 2b) and the remaining edges $\overline{v_i v_k}$, $\overline{v_k v_j}$, $\overline{v_j v_l}$, and $\overline{v_l v_i}$ determine the pairs of triangles, which are recursively checked for the empty circle property. This procedure is known as a Lawson's local optimization shortly named also a *legalization* [13].

The Lawson's local optimization is applied by the incremental insertion Delaunay triangulation algorithms. These algorithms consecutively insert points into existing $\mathcal{DT}(\mathcal{S})$ and work in two steps:

1. Firstly, a triangle containing the inserted point is determined. This triangle is split into three new triangles (four, if the point falls on an edge).
2. These new triangles are recursively legalised using Lawson's local optimization.

Surprisingly, the most time-consuming part of the incremental insertion algorithms is the search for a triangle containing the inserted point (for that different approaches have been suggested [2–6]), while the legalization is terminated in expected logarithmic time [13].

2.2. Fortune's sweep-line technique

The sweep-line (or plane sweep) is one of the most popular acceleration techniques used to solve 2D geometric problems [14]. The idea of the sweep-line technique is very simple: firstly, the geometric elements are sorted. Then, it is imagined that the sweep-line glides over the plane and stops at so-called event points (usually, the sweep-line stops at the geometric elements being considered). A portion of the problem is solved and the data structures are updated. The part of the problem being swept is already completely solved, while the remaining part is unsolved. Unfortunately, this simple and clear principle cannot be applied directly for constructing Delaunay triangulation. Namely, the point where the sweep-line stops is also connected to the points that have not been passed, yet. Up to now, only Fortune found a clever way for applying the sweep-line approach for constructing $\mathcal{DT}(\mathcal{P})$ [11] (actually, Fortune designed his algorithm to construct the Voronoi diagram—a dual graph of Delaunay triangulation [15]). His algorithm maintains information about those parts of the diagram already swept, which cannot be changed. The border between the solved and the unsolved parts is divided by a so called *beach line*, which consists of a connected chain of parabolic arcs (see Fig. 3).

As the sweep-line moves through the plane, the beach line takes different forms controlled by two kinds of events.

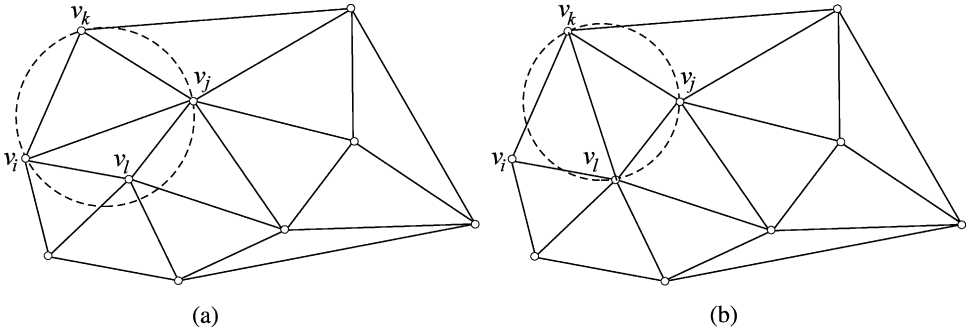


Fig. 2. Empty circle property and the legalization.

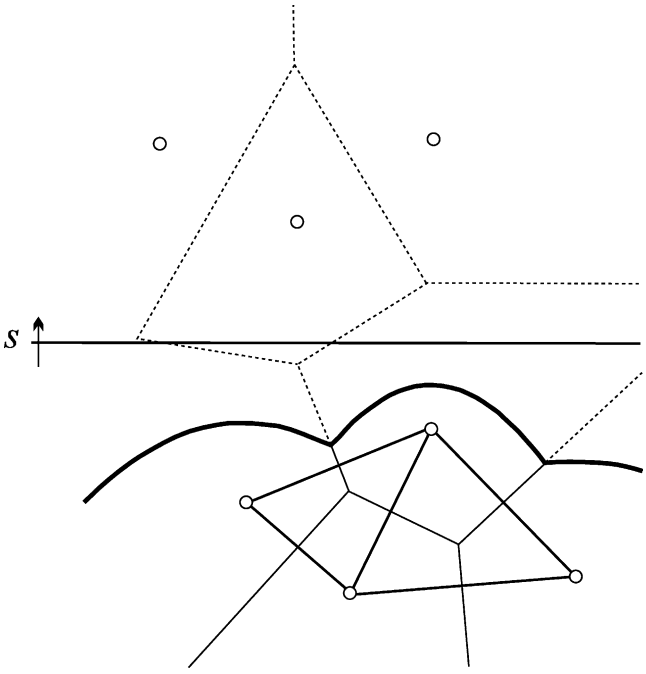


Fig. 3. Beach line is behind the sweep-line.

The first event, named *point even* (or side event), appears when the sweep-line hits the point from $\mathcal{S}$. In this case, a vertical projection is done on the chain of parabolic arcs, and a new parabolic arc is created (see Fig. 4).

The second event (so called *circle event*) appears when an existing arc of the beach line shrinks to a point and disappears (point q in Fig. 5). The event point q is the centre point of vertices v_i, v_j, and v_k, with the highest point going through the sweep-line. In this way, there cannot be any point form $\mathcal{S}$ inside this circle, which ensures the empty circle property is not violated. When observing the Voronoi diagram, the circle event appears at the Voronoi vertices.

There are n point events and at most $2n-5$ circle events (there, is at most, $2n-5$ Voronoi vertices [13]) giving us in total, at most, $3n-5$ events.

Section 3 presents a new sweep-line algorithm combined with the Lawson's local optimization.

3. The algorithm

Let us consider Fig. 6 and suppose that sweep-line s has already passed the first 15 points. The algorithm surrounds them by two bordering polylines:

- The lower border (denoted by dot-dashed lines in Fig. 6) forms part of the convex hull. We name it a *lower convex hull*.
- The upper border polyline (plotted by dashed lines) is considered as an *advancing front*. The advancing front separates the swept vertices from the non-swept. The shape of the advancing front depends on the arrangement of the points already passed and, in general, does not coincide with the upper convex hull.

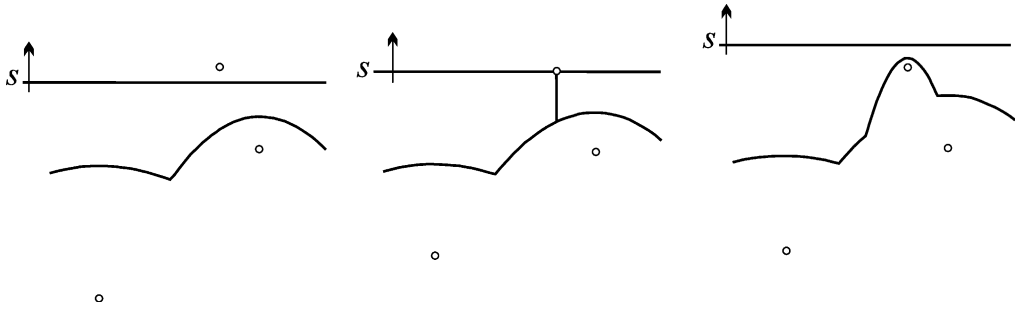


Fig. 4. Sweep-line hits the point.

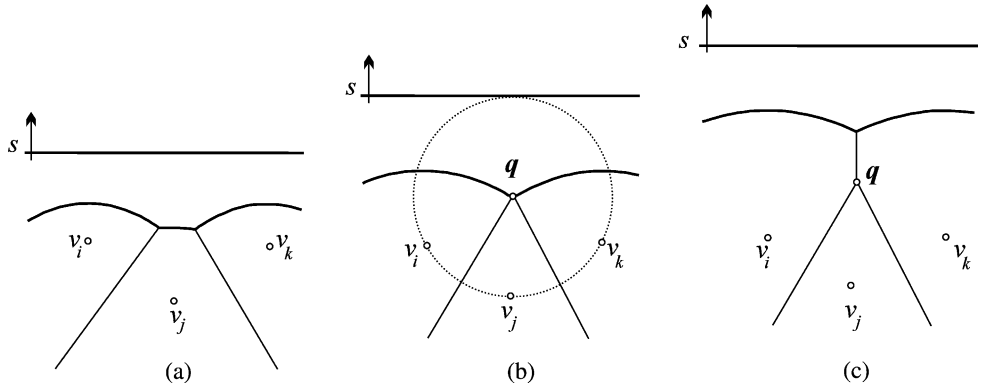


Fig. 5. The circle event.

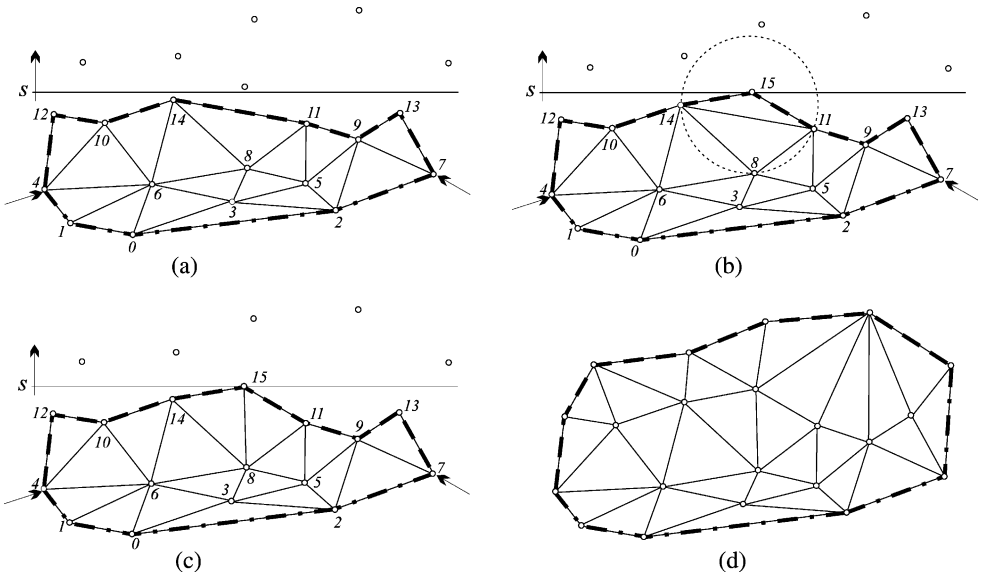


Fig. 6. The main idea of sweep-line triangulation.

The lower convex hull and the advancing front have exactly two vertices in common—the vertices with the minimal and maximal x coordinates among the swept points (vertices v_4 and v_7 from Fig. 6a). All the vertices between the lower convex hull and the advancing front are triangulated according to the empty circle property. When the sweep-line meets the next point (vertex v_{15} in our example), the reaction of the algorithm is as follows: at first, a vertical projection of vertex v_{15} is done on the advancing front. It hits edge $\overline{v_{14}v_{11}}$ (let us omit temporarily the case where the projection misses the advancing front). It is easy to construct a new triangle $\triangle_{15,11,14}$ and change the advancing front to correspond to the new situation (see Fig. 6b). Unfortunately, there is no guarantee that the new triangle $\triangle_{15,11,14}$ is legal and must be checked with its neighboring triangle ($\triangle_{8,14,11}$) according to the empty circle property. As seen from Fig. 6b, the empty circle property is violated, and diagonal swapping is employed recursively. As a result, Delaunay triangulation is obtained between the modified advancing front and the lower convex hull (see Fig. 6c). Fig. 6d shows the situation after all points have been swept. As seen, in this case, the advancing front is not convex and does not correspond to the upper convex hull. Additional triangles have to be added for this, and legalised, before the advancing front becomes convex, which finishes the triangulation process.

3.1. Initialization

The initialization of the algorithm (besides sorting the input points) includes:

- construction of the first triangle (or more of them),
- determination of the initial advancing front, and the initial lower convex hull.

The first three points are taken after sorting the input points regarding the y coordinate. Let us assume they are non-collinear. The points are oriented in a counter-clockwise direction and the advancing front and the lower convex hull are initialised. As shown in Fig. 7, six possible configurations may appear. For example, if x coordinate of vertex v_0 is between x coordinates of vertices v_1 and v_2 (Fig. 7a and b), the advancing front contains two vertices ($\{v_1, v_2\}$ or $\{v_2, v_1\}$), while the lower convex hull contains three ($\{v_1, v_0, v_2\}$ or $\{v_2, v_0, v_1\}$). The vertices in the advancing front and the lower convex hull are always sorted regarding the x coordinate. Each vertex of the advancing front (or the lower convex hull) stores the pointer to the farthest right (the farthest left) triangle being defined by that vertex. In this way, triangles touching the advancing front or the lower convex hull with one or two edges are accessed directly.

If the first m, ($m<n$) points are collinear, the first non-collinear point is used to construct the $m-1$ triangles. According to this, the advancing front and the lower convex hull have to be initialised adequately (see examples in Fig. 8).

3.2. Triangulation

The next unused vertex v_i is taken from the sorted array of input vertices and it is vertically projected on the advancing front. One of four cases may occur:

- the projection hits the advancing front,
- the projection misses the advancing front,
- vertex v_i is located on the advancing front, or
- vertex v_i coincides with a vertex of the advancing front.

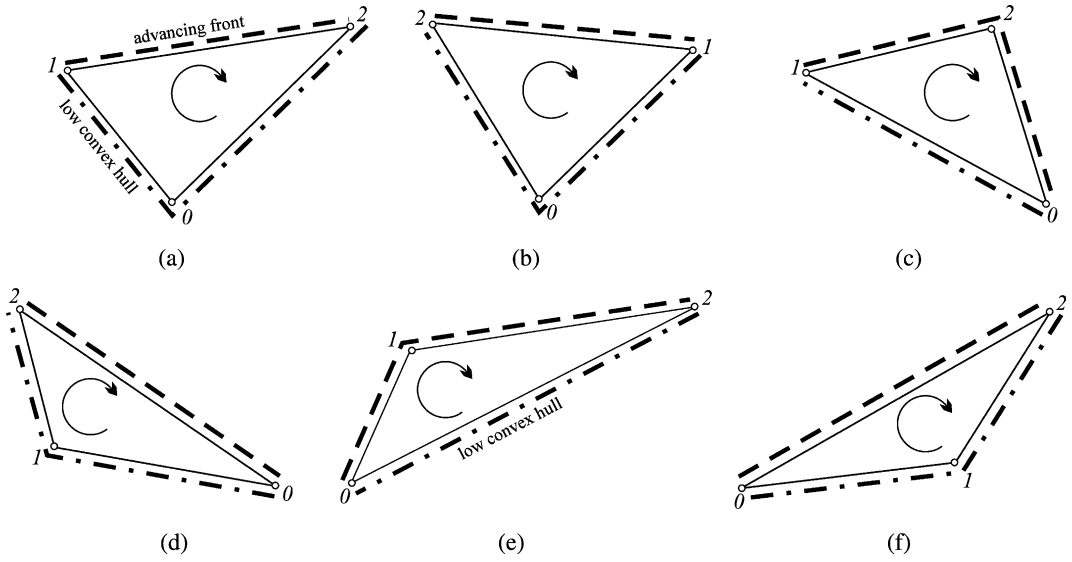


Fig. 7. Initial configurations at non-collinear points.

3.2.1. The projection hits the advancing front

This case is the most common and has been used to highlight the main idea of the approach at the beginning of this section. Now, all necessary details are considered. An edge of the advancing front, which is hit by the vertical projection of vertex v_i is defined by two vertices denoted as v_L (left vertex) and v_R (right vertex) (Fig. 9a). A new triangle $\triangle_{i,R,L}$ is easily constructed. Through vertex v_L, the neighboring triangle $\triangle_{k,L,R}$ is directly accessible, and the triangles are checked recursively for Delaunay criteria. The advancing front is updated by inserting the vertex v_i between vertices v_L and v_R (Fig. 9b). Usually, vertex v_i also has to be connected with other vertices from the advancing front. The most straightforward idea would be to construct the new triangles to the left (and to the right) starting at v_L (and v_R) using v_i as a peak, until the first edge of the advancing front is pierced. In this way, the advancing front would always represent the upper part of the convex hull. Unfortunately, this idea leads to relatively inefficient implementation as a lot of tiny triangles are obtained, which are legalised in the next iterations of the algorithm (see Table 8 in Section 4.4 for practical results). Another approach is proposed to prevent the construction of triangles, which would very probably be legalized. Let us consider the procedure while examining the right side of the advancing front regarding vertex v_R; the solution for the left side being symmetrical.

The angle between the vectors determined by vertices v_i, v_R, v_{R+}, is observed (Fig. 9b). If the angle is smaller than $\pi/2$, triangle $\triangle_{i,R+,R}$ is generated (Fig. 9c), otherwise the walk in the right-side direction is stopped. The new triangle is faced recursively against Delaunay criteria with the two neighboring triangles ($\triangle_{i,R,L}$ and $\triangle_{R+1,l,R}$). The vertex v_R is removed from the advancing front (Fig. 9c) and the process is repeated for vertices v_i, v_{R++}, v_{R+} until the angle between them is greater than $\pi/2$ (Fig. 9d). Triangles obtained in this way do not tend to violate Delaunay criterion too frequently, reducing the spent CPU time (see the analysis in Section 4.4).

Unfortunately, in this way the advancing front may become considerably wavy leading, in some cases, to what we call *basins* (Fig. 10a). A basin appears when no point

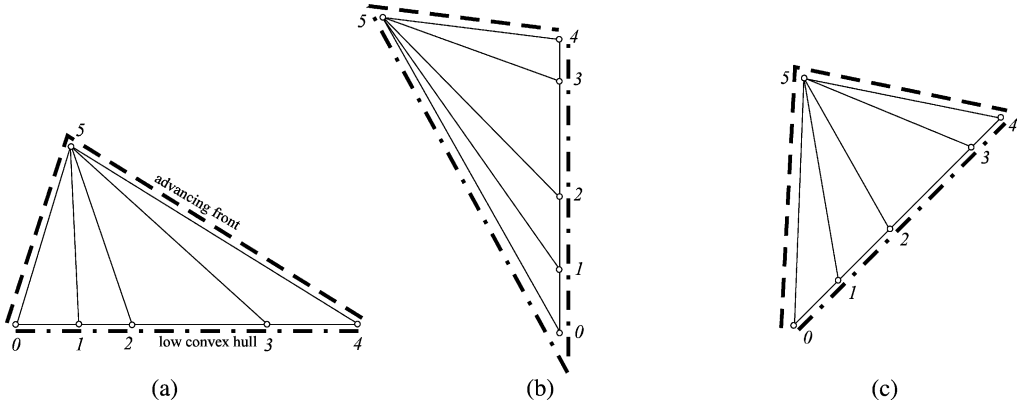


Fig. 8. The first m input points are collinear.

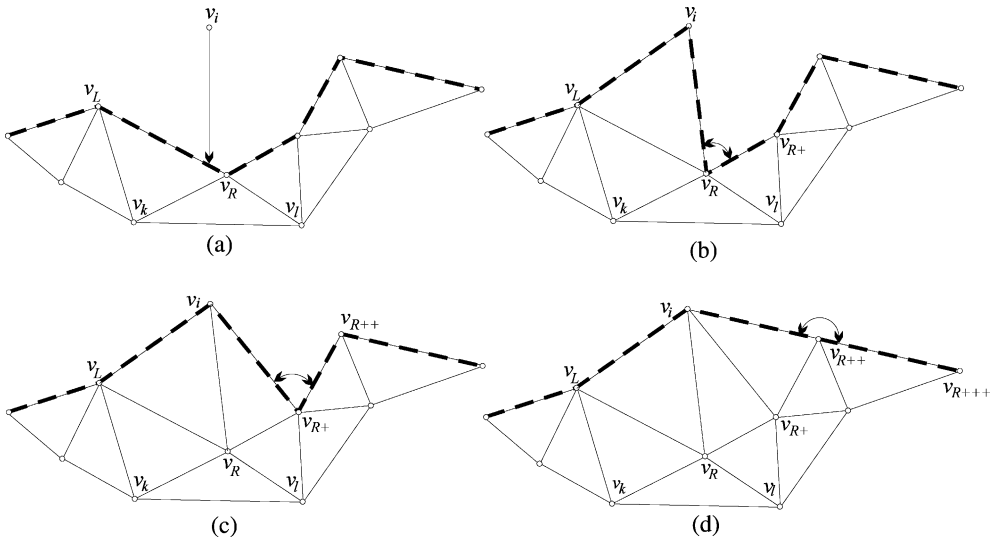


Fig. 9. Advancing front is hit.

appears for a while upon the successive line segments of the advancing front. When the point finally appears, tiny triangles are generated and then legalized by several diagonal swaps. This, of course, slows down the algorithm and makes it sensitive to the distribution of the input points. In order to remove such an unpleasant situation, another heuristic is introduced to detect the basins early enough as follows: let vertex v_{R} be the right-side neighbour of vertex v_i in the advancing front, and let $v_{\mathrm{R}+}$ be its right-side follower (Fig. 10b). The slope of the line connecting v_i and $v_{\mathrm{R}+}$ is determined as to whether it is smaller than $3\pi/4$. In this case, a basin is found. By calculating the signed area of triangle $\triangle_{i,\mathrm{R}+,\mathrm{R}}$, either v_{R} or $v_{\mathrm{R}+}$ is considered as the left-side border of the basin (in Fig. 10b the left-side border is vertex $v_{\mathrm{R}+}$). The bottom vertex of the basin is marked as v_b in Fig. 10c. The right-side border vertex (denoted by v_a in Fig. 10c) of the basin is the first vertex of the advancing front walking from vertex v_b to the right-side whose right neighbouring vertex has smaller or the same y coordinate than the considered vertex. Two monotone chains of vertices are constructed and their triangulation is now very simple [13]. Of course, to ensure the resulting triangulation is optimal, each generated triangle is faced against the empty circle property with their neighbours (Fig. 10d). Finally, the advancing front connects the borders of the basin, and in this way it is smoothed considerably.

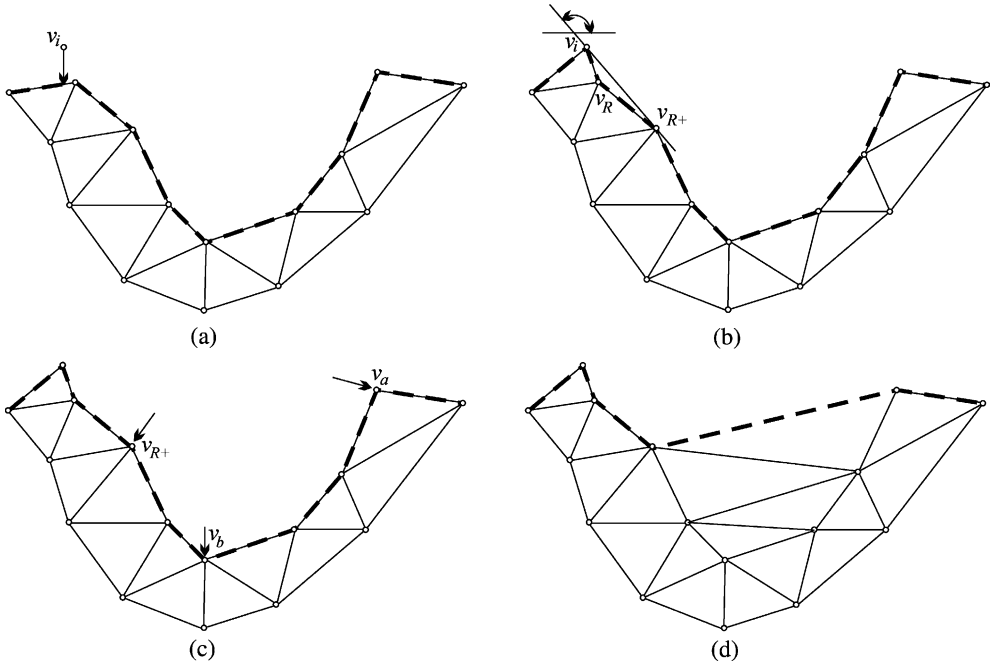


Fig. 10. The basin.

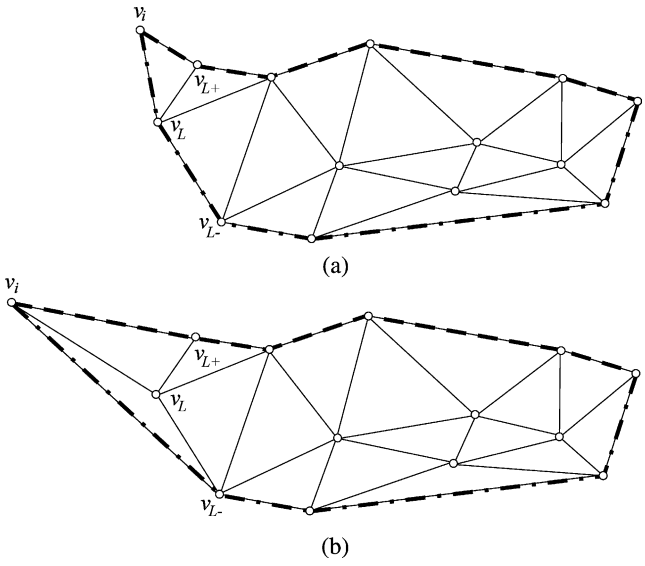


Fig. 11. The advancing front is missed.

3.2.2. The projection misses the advancing front

The vertical projection of vertex v_i may miss the advancing front either from the left or from the right side. As the solutions are symmetrical, only the left case is explained. In this case, a new triangle $\triangle_{i,\mathrm{L}+,\mathrm{L}}$ is constructed and checked for Delaunay criteria. The new triangles in the right-side direction of the advancing front are generated using the same procedure as in Section 3.2.1 (Fig. 11a). Then, the new triangles are also added along the lower convex hull. All possible triangles are generated in this case because we have to keep the complete convex hull at the bottom of the triangulation (triangle $\triangle_{i,\mathrm{L},\mathrm{L}-}$ in Fig. 11b). The new vertex is appended to the left side of the lower convex hull.

3.2.3. Vertex is located on the advancing front

All sweep-line algorithms have a characteristic case, when more than one geometric element of interest is located on the sweep-line. Usually, this special case has to be treated very carefully to assure the numeric stability of the algorithm.

In our case, the special case is also solved separately but the solution is stable and does not slow down the algorithm. Actually, its implementation is very short and simple. Fig. 12a shows the special case. It is detected very easily: vertices v_i, v_{L}, and v_{R} should have the same y coordinate within tolerance ϵ. Triangle $\triangle_{\mathrm{L},\mathrm{R},k}$ accessible from vertex v_{L}, is split and two new triangles $\triangle_{\mathrm{L},i,k}$ and $\triangle_{i,\mathrm{R},k}$ are obtained (Fig. 12b). Both triangles are legalized with the neighbouring triangle and the advancing front is updated at the end (vertex v_i is inserted between v_{L} and v_{R}).

3.2.4. Vertex coincides with a vertex of the advancing front

In this case, vertex v_i is neglected and the next vertex is taken immediately.

3.3. Finalization

When all points are swept, some vertices of the advancing front remain concave and some triangles are still missed in triangulation because of this (see Fig. 13a).

The missed triangles are added by performing a walk through the edges of the advancing front. The algorithm is a simple extension of the Graham scan—a well-known algorithm for determining the convex hull [14]. In short: three consequent vertices of the advancing front are taken. The signed area of the triangle obtained is calculated. If positive, the vertices determine the missed triangle (three of such vertices are, for example, vertices v_1, v_2, and v_3), which is generated and checked for Delaunay criteria. The middle vertex (vertex v_2 in Fig. 13a) is erased from the advancing front. The next triple (v_1, v_3, v_4) is checked. As the signed area is negative, the algorithm moves a step further and considers vertices (v_3, v_4, v_5). The complete Delaunay triangulation is shown in Fig. 13b.

3.4. Implementation of the advancing front

The advancing front can be realised by any data structure supporting the range search as, for example, AVL tree. In our case, a simple hash-table on a double connected list is used for representation of the advancing front (see Fig. 14). The number of entries into the hash-table was determined using the following heuristic:

$$\text{No. of Entries} = 1 + \left\lfloor \frac{n}{k} \right\rfloor; \tag{1}$$

where k was set experimentally to the value 100. However, as shown in the results of the experiments given in Section 4.3, considerably larger values can be taken without fear of spoiling the characteristics of the algorithm significantly. Each record of the advancing front stores the key (x coordinate of the vertex), vertex index v_i, and index of the triangle t_i sharing its edge with the advancing front.

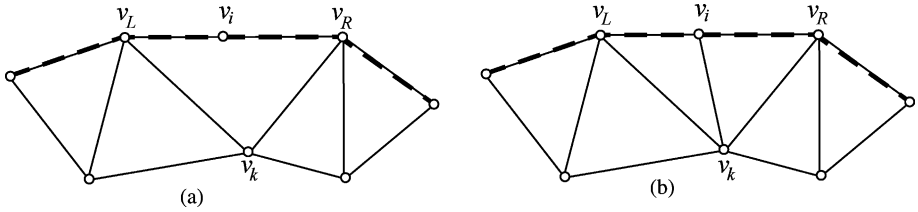


Fig. 12. Solving the special case.

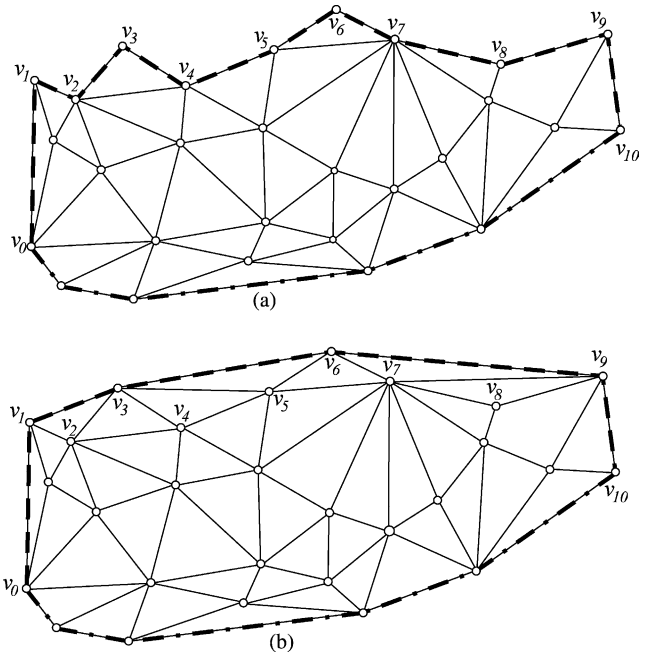


Fig. 13. Conversion of the advancing front to the upper convex hull.

The lower convex hull is represented by a double connected list. No hash-table is needed because the range search is never applied on the lower convex hull.

4. Analysis of the algorithm

The algorithm has been analyzed from the following aspects:

- time and space complexity analysis,
- spent CPU time using artificial and engineering data sets,
- analyizing the influence the hash-table's size, and
- CPU time-independent comparisons.

4.1. Time and space complexity

Firstly, all input points are sorted regarding the y coordinate. Using Quicksort, this is done in

$$T_1(n) = \mathrm{O}(n \log n), \tag{2}$$

where n is the number of points being triangulated. The second part of the algorithm, i.e. the construction of the triangulation, is estimated as follows. The entry into the hash table is calculated firstly for each inserted point. This is done in constant time. Let us assume, there are *NoOfEntries*, *NoOfEntries* >0, entries and there are m vertices in the advancing front. To locate the pierced edge in the advancing front, (m/*NoOfEntries*) tests are done on average. Generation of the new triangles, and checking for the introduced heuristics are done in constant time. What remains is the legalization of the new triangles. According to [13] this task is done in logarithmic time. The total time complexity of the algorithm's second part is therefore

$$T_2(n) = n\left(\frac{m}{NoOfEntries} + \log n\right). \tag{3}$$

Usually, $m \ll n$, and No. of Entries $\gg 1$. In this case, (m/*NoOfEntries*) $\ll n$ and can be neglected giving us an expected time complexity

$$T_2(n) = \mathrm{O}(n \log n). \tag{4}$$

The common expected time complexity of the proposed algorithm is therefore:

$$T_{\text{common}}(n) = T_1(n) + T_2(n)$$
$$= \mathrm{O}(n \log n) + \mathrm{O}(n \log n) = \mathrm{O}(n \log n). \tag{5}$$

Let us estimate the space complexity of the algorithm. $2n$ memory locations are reserved for triangles, (n/No Of Entries), (0 $<$ No Of Entries $<n$) memory locations are reserved for hash-table, and m, $m<n$, locations are needed for the advancing front, leading to the linear space complexity $S=\mathrm{O}(n)$.

4.2. Comparing spent CPU

This section gives an analysis of the proposed algorithm's spent CPU time. The main purpose of these tests is to estimate:

- how efficient the algorithm is when compared with other popular Delaunay triangulation algorithms,
- how the algorithm behaves on different point distributions, and
- how it copes with the engineering data.

The actual run time of an algorithm depends on many factors as, for example: programming language, operating system, compiler and compiling options, and the programming experience of the programmer. However, the idea of the algorithm still plays the key role. Thus, the measurements of Delaunay algorithms' spent CPU time are very common

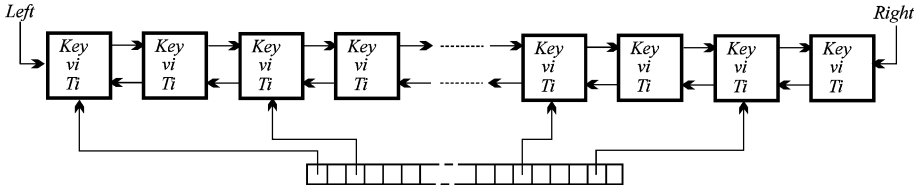


Fig. 14. Storing the advancing front.

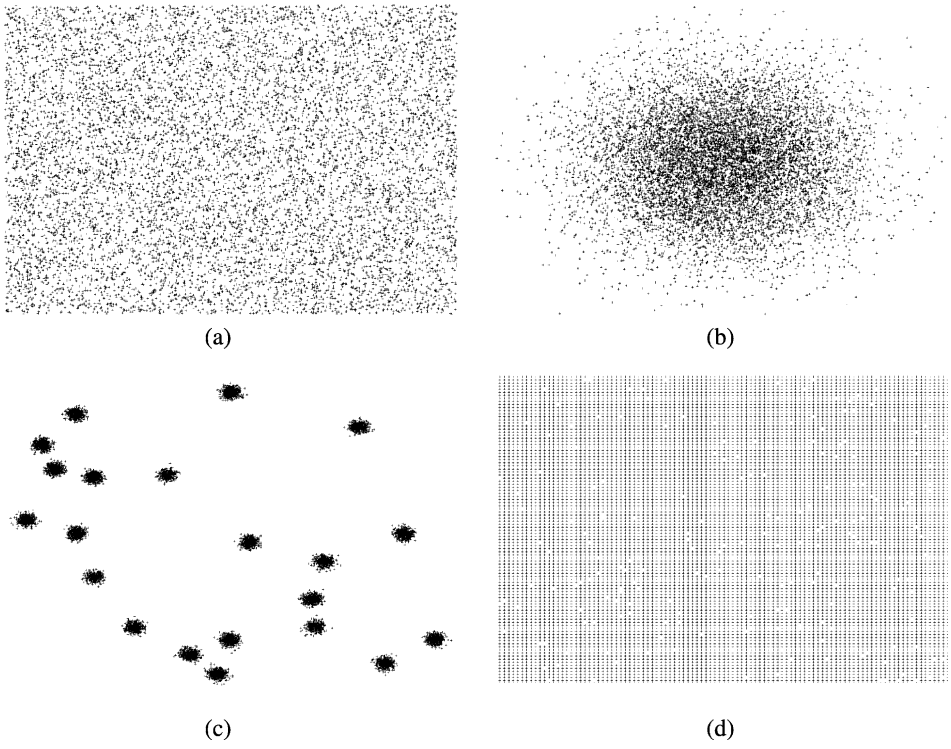


Fig. 15. 10,000 points at various distributions: uniform (a), Gaussian (b), points arranged in clusters (c), and points arranged in grid (d).

[1,16]. In our case, all tested algorithms were executed on a PC with an AMD Athlon XP 2000 + 1.666 GHz processor and 512 MB RAM running under the Windows 2000 operating system. A code written solely by the author of this paper, and Shewchuk [17,18] were applied for testing. Shewchuk's set of triangulation algorithms is, because of excellent characteristics and numerical stability, often used as the reference for evaluation of triangulation algorithms [16,19]. His code is available to the public at the WEB (see [17]). For comparison, we used inexact (floating-point) implementation of his algorithms. The following algorithms were used for testing with the proposed sweep-line Delaunay triangulation algorithm (SL):

- The improved version [5] of the Guibas, Knuth, and Sharir's [3] randomised incremental algorithm using the directed acyclic graph (denoted as GKS-KZ).
- The randomised incremental algorithm based on the nearest point paradigm using two-level uniform space subdivision [6] (denoted as US2l).
- Lee and Schachter's [8] divide and conquer algorithm implemented by Shewchuk within a triangulation package *Triangle* [17] (denoted as D&C-S).
- Fortune's [11] sweep-line algorithm implemented by Shewchuk in *Triangle* (denoted as F-S).

At first, artificial data sets are used with uniform and Gaussian distributions and points arranged in a grid and clusters (see Fig. 15). The results are shown in Tables 1 and 2. The proposed algorithm is the fastest for all distributions. The second fastest is Lee and Schachter's divide and conquer algorithm. For our algorithm, points arranged in regular grid represent the best case. The reason for this is explained later in Section 4.4.

Table 3 shows standard deviation σ for all tested algorithms when 1,000,000 points are triangulated. As can be seen, D&C-S and both sweep-line algorithms are less sensitive on different distributions of the input points.

Table 4 shows how much time in our algorithm is spent on sorting the input points and how much on the Delaunay

Table 1
Spent CPU time (s) for uniform and Gaussian point distributions

	Uniform distribution			Gaussian distribution		
	100,000	500,000	1,000,000	100,000	500,000	1,000,000
GKS-KZ	2.63	16.40	78.60	2.63	15.80	34.52
US2l	1.40	8.71	23.28	1.45	13.31	42.20
D&C-S	0.90	5.92	13.47	0.91	5.67	13.00
F-S	1.40	10.72	26.41	1.40	10.51	27.12
SL	0.25	1.38	3.00	0.27	1.39	3.05

Table 2
Spent CPU time (s) for points arranged in clusters and grids

	Cluster data distribution			Grid data distribution		
	100,000	500,000	1,000,000	100,000	500,000	1,000,000
GKS-KZ	2.10	11.87	25.10	2.40	15.42	35.12
US2l	2.30	30.41	80.25	1.30	8.89	24.05
D&C-S	0.89	5.08	11.40	1.59	9.18	20.30
F-S	1.30	8.00	18.58	1.41	8.62	23.50
SL	0.25	1.25	3.08	0.20	1.10	2.50

triangulation. On average 1/4 of the time is required for Quicksort, and the rest for the triangulation.

At the end, engineering data sets were used. The points were obtained from the mesh for FEM, and from a GIS database (boundary stones of the parcels). Again, all referenced algorithms were used. Results are shown in Table 5. Two examples of the data sets are displayed in Fig. 16. The proposed algorithm is again the most efficient.

4.3. The influence of the resolution of the hash-table

In this section we present the relation between spent CPU time and the number of entries in the hash-table. Table 6 shows spent CPU time while triangulating 500,000 points at different point distributions, setting different values for k.

The minimum is not sharp, and any value between 10 and 2000 gives very acceptable results. Of course, if k is larger, the number of entries into the hash-table is smaller and, therefore, the hash-table occupies less memory. Worse results are obtained only if k is close to 1 (or too large, of course). This is surprising at first glance, but can easily be explained. If there are too many entries into the hash-table, many entries are empty during sweeping. The wasting of time when searching for the first non-NULL entry significantly influences the total CPU time spent. Although the hash-table does not occupy a lot of memory on today's computers, even if k is small, this finding enables memory saving without fear of the algorithm becoming considerably slower. For example, if 1,000,000 points need to be triangulated, and if $k=2000$, the hash-table consists of only 500 entries. Each entry contains a 4 bytes large pointer into the advancing front. This requires only 2000 extra bytes for the hash-table. In our practical experience, $k=100$ is used.

Table 3
Average spent CPU time (s) and standard deviation σ for 1,000,000 points

	GKS-KZ	US2l	D&C-S	F-S	SL
Average CPU time (s)	43.336	42.445	14.543	23.903	2.908
Standard deviation σ	20.744	23.103	3.412	3.358	0.237

Table 4
Spent CPU time within sweep-line algorithm SL

	100,000	500,000	1,000,000
Sorting	0.06	0.35	0.78
Triangulation	0.19	1.03	2.22
Total	0.25	1.38	3.00

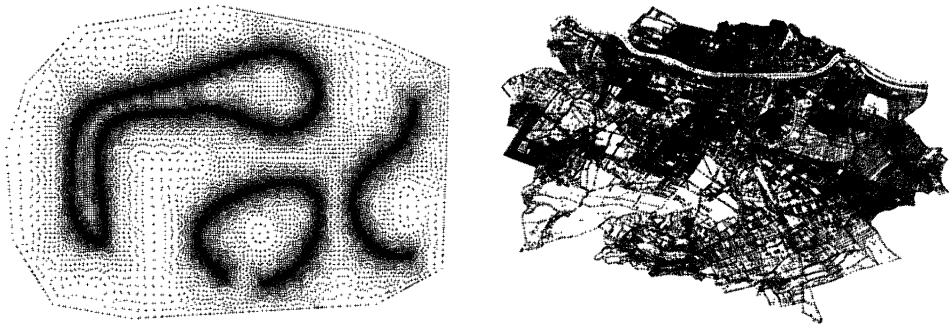
4.4. CPU time-independent comparisons

The suitability of the introduced heuristics in Section 3 is estimated at first. The main aim of the heuristics is to reduce the creation of incorrect triangles and, as a consequence, to minimize the number of calls to the legalization procedure. The comparison is done using incremental insertion algorithms, which are based on the legalization procedure as introduced by Lawson [12], other algorithms do not use it. Table 7 gives the number of legalised triangles done by the proposed algorithm and by the incremental insertion algorithm using different point distributions. Sets of points containing 500,000 points were used. Four different sets were investigated for each distribution, and the average number of successful calls of the legalization procedure is given.

A noticeable deviation was only noticed at points arranged in grids, the remaining distributions practically give the same results. The proposed algorithm requires only 26% of diagonal swaps regarding the incremental insertion algorithm. As seen in Table 7, every 2.5th generated triangle is legalised in our algorithm. The situation is much worse for the incremental insertion algorithm where each triangle is legalised 1.5 times on average. Both algorithms behave better at points arranged in grids. In this case, our algorithm only legalises every 5th triangle. Here lies the answer as to why the algorithm's run time is shorter using this point distribution (see Section 4.3). Our algorithm does 83% less diagonal swaps than the incremental insertion algorithms when the points arranged in grids are triangulated.

Table 5
Spent CPU time (s) for engineering datasets

No. of points	GKS-KZ	US2l	D&C-S	F-S	SL
59,041	1.31	0.99	0.47	0.72	0.16
127,318	3.51	1.78	1.05	1.58	0.32
193,360	4.90	3.23	1.58	2.61	0.53
219,163	5.94	4.34	1.91	3.34	0.63



(a) (b)

Fig. 16. 59,041 points for FEM (a) and 193,360 boundary stones (b).

Table 6
The influence of the parameter k on spent CPU time (s)

Distribution/k	1	10	50	100	200	500	1000	2000	5000
Uniform	4.16	1.63	1.39	1.38	1.37	1.36	1.36	1.38	1.41
Gauss	3.34	1.56	1.41	1.39	1.38	1.38	1.39	1.41	1.46
Grid	1.76	1.15	1.11	1.10	1.09	1.09	1.11	1.11	1.13
Cluster	1.73	1.27	1.25	1.25	1.26	1.26	1.27	1.33	1.39

The most time-consuming part for the incremental insertion algorithms is point location, but legalization also consumes time, if executed too frequently. What this means in our case is shown in Table 8 (the same sets of points as in Table 7 have been used), where the algorithm is executed with and without the heuristics. The number of legalised triangles and the spent CPU time increased dramatically in the latest case. The algorithm becomes very sensible on the point distribution. As can be seen, triangulation of the points arranged in the grid is more than 5 times quicker than triangulation of points grouped in clusters. Surprisingly, Gaussian distributed points are triangulated much faster than points distributed uniformly, although the number of diagonal swaps is almost the same. The reason for this is that more of the very tiny triangles are generated at uniform distribution. In this way, the deepness of the legalization recursion seems to be larger, requiring more calculation before entering the part where the diagonal swap is executed.

Finally, let us compare our sweep-line approach with Fortune's sweep-line algorithm. Both algorithms sort the input points at first. Fortune's algorithm introduces the beach-line—the connected chain of parabolic arcs, which lags behind the sweep-line. In this way, legalization is not needed, but efficient maintenance of the beach-line is demanding. The sweep-line stops in, at most, $3n-5$ event points. Only n event points is known in advance, the remaining ones have to be calculated. At the proposed algorithm, the sweep-line has only n known stops at the points. The algorithm applies the advancing front represented by the line segments, which is much easier to implement. The advancing front is stored in a simple data structure. Lawson's recursive local optimization is needed in our case to assure the Delaunay criteria. Both algorithms are insensitive on different distributions of input points.

5. Conclusion

Brute computer power does not remove the need for the fast Delaunay triangulators. Quite the opposite, using cheap on-board memory the number of points needed to be triangulated increases dramatically.

This paper introduces a new algorithm for constructing Delaunay triangulation in the plane. The algorithm efficiently combines the sweep-line paradigm with

Table 7
Number of legalised triangles

distribution	No. of triangles	Sweep-line	Incremental insertion	Ratio
Uniform	998,897	389,874	1,495,322	0.26
Gaussian	991,196	384,472	1,468,752	0.26
Cluster	999,969	392,625	1,498,816	0.26
Grid	997,177	190,858	1,093,703	0.17

Table 8
Comparison of the sweep-line algorithm without using heuristics

Point distribution	No. of triangles	Heuristic used		Heuristic not used	
		Legalised triangles	CPU(s)	Legalised triangles	CPU(s)
Uniform	998,897	389,874	1.38	4,747,383	27.13
Gaussian	991,196	384,472	1.39	4,395,203	15.43
Grid	997,177	190,858	1.10	2,958,296	6.53
Cluster	999,969	392,625	1.25	5,233,570	35.22

the legalization—the characteristic of incremental insertion Delaunay triangulation algorithms. By introduced heuristics, the number of triangles needed to be legalised, is reduced efficiently, which is also reflected in spent CPU time. This algorithm has many attractive features:

- it is easy to understand,
- it uses simple data structure,
- the data structure does not consume a lot of additional space,
- it is not sensible to the distribution of input points, and
- it is very fast.

All of this rates this algorithm among the best algorithms for constructing Delaunay triangulation. A demo if this algorithm can be downloaded from http://gemma.uni-mb.si/~gemma/projects/Triangulation.htm.

Acknowledgements

The author is grateful to the Ministry of Education, Science and Sport of Republic Slovenia for supporting this research under the project P2-0041—Computer systems, methodologies and intelligent services. The author would like to thank to dr. Gorazd Hren from Faculty of Mechanical Engineering at University of Maribor for providing the engineering data sets.

References

[1] Su P, Drysdale RLS. A comparison of sequential Delaunay triangulation algorithms. Proceedings of SCG'95 (ACM Symposium on Computational Geometry) 1995 p. 61–70.
[2] Guibas L, Stolfi J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Trans Graph 1985 4(1):75–123.
[3] Guibas L, Knuth D, Sharir M. Randomised incremental construction of Delaunay and Voronoi diagrams. Algorithmica 1992;(7):381–413.
[4] Mücke EP, Saias I, Zhu B. Fast randomized point location without preprocessing and two- and three-dimensional Delaunay triangulations. Proceedings of SCG'96 (ACM Symposium on Computational Geometry) 1996 p. 274–83.
[5] Kolingerová I, Žalik B. Improvements to randomized incremental delaunay insertion. Comput Graph 2001;(26):477–90.
[6] Žalik B, Kolingerová I. An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. Int J Geograph Inf Sci 2003;(17):119–38.
[7] Fang T-P, Piegl L. Delaunay triangulation using a uniform grid. IEEE Comput Graph Appl 1986 13(3):36–47.
[8] Lee DT, Schachter BJ. Two algorithms for constructing a delaunay triangulation. Int J Comput Inf Sci 1980;(9):219–42.
[9] Dwyer RA. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. Algorithmica 1987;(2):137–51.
[10] Edelsbrunner H, Seidel R. Voronoi diagrams in linear expected time. Discrete Comput Geom 1986;(1):25–44.
[11] Fortune S. A sweep-line algorithm for Voronoi diagrams. Algorithmica 1987;(2):153–74.
[12] Lawson CL. In: Rice JR, editor. Software for C1 surface interpolation. Mathematical software III. New York: Academic Press; 1977. p. 161–94.
[13] de Berg M, van Kreveld M, Overmars M, Schwarzkopf O. Computational geometry, algorithms and applications. Berlin: Springer; 1997.
[14] Preparata FP, Shamos MI. Computational geometry: an introduction. Berlin: Springer; 1985.
[15] Aurenhammer F. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Comput Surveys 1991;(3):345–405.
[16] Boissonnat JD, Devillers O, Teillaud M, Yvinec M. Triangulations in CGAL. In: Computational Geometry 2000. Hong Kong: ACM Press; 2000, p. 11–8.
[17] Shewchuk JR. A two-dimensional quality mesh generator, Delaunay triangulator. http://www.cs.berkeley.edu/jrs/index.html; 2004.
[18] Shewchuk JR. Triangle: engineering a 2D quality mesh generator and delaunay triangulator. Proceedings of SCG'96 (ACM Symposium on Computational Geometry) 1996 p. 124–133.
[19] Devillers O. Improved incremental randomized delaunay triangulation. Proceedings of SCG'98 (ACM Symposium on Computational Geometry) 1998 p. 106–15.

Borut Žalik is a full professor at the Department of Computer Science, Faculty of EE and CS, University of Maribor, Slovenia. He received his BSc in Electrical engineering in 1985, MSc and PhD in computer science, both from the University of Maribor in 1989 and 1993, respectively. For two years, he had a position of a visiting senior research fellow at De Montfort University, UK. Currently, he is the vice-dean for the research at the Faculty of EE and CS, University of Maribor. Žalik leads a Laboratory for geometric modelling and multimedia algorithms. His research interests include computational geometry, geometric modeling, CAD, GIS, and multimedia applications.