International Journal of Engineering, Mathematical and Physical Sciences
ISSN: 2517-9934
Vol:6, No:5, 2012

Optimal algorithm for constructing the Delaunay Triangulation in E^d

V. Tereshchenko and D. Taran

Abstract—In this paper we propose a new approach to constructing the Delaunay Triangulation and the optimum algorithm for the case of multidimensional spaces $(d \geq 2)$. Analysing the modern state, it is possible to draw a conclusion, that the ideas for the existing effective algorithms developed for the case of $d \geq 2$ are not simple to generalize on a multidimensional case, without the loss of efficiency. We offer for the solving this problem an effective algorithm that satisfies all the given requirements. But theoretical complexity of the problem it is impossible to improve as the *Worst - Case Optimality* for algorithms of solving such a problem is proved.

Keywords—Delaunay Triangulation, multidimensional space, Voronoi Diagram, optimal algorithm.

I. Introduction

THE problem of the effective algorithm construction for the Delaunay triangulation has been actual for a long time already. Therefore, many approaches were offered for her solving [1-10]. In the case of a plane the task is completely done as there is the *Worst - Case Optimal* algorithm (the Fortune algorithm [11]). However, a theoretical optimal algorithm for a general multidimensional case has not been found yet. Therefore, except theoretical complication there is need in practical estimations of time of algorithms'work. For solving the given task there are several general methods and all known algorithms work in accordance with one of them, in particular: divide-and-conquer, based on a sweep plane (or space), successive points addition (random incremental method) and the direct algorithms.

One alternative method should also be noted. It lies in the existence of the direct connection between d-dimensional Delaunay triangulation and $(d+1)$ - dimensional convex hull. Every point of set is given in accordance the point $(d+1)$ of dimensional paraboloid. Every $(x_1, x_2, \ldots, x_d)$ is given in accordance the point $(x_1, x_2, \ldots, x_d, x_1^2 + x_2^2 + \ldots + x_n^2)$. A convex hull is built on the gotten points. Projecting then every face of a convex hull back into d -dimensional space (by rejecting the last coordinate) we will get triangulation simplexes of initial set of points [1].

But the situation with the algorithms of construction the multidimensional convex hulls is not much better than the situation with the Delaunay triangulation. Therefore this approach does not give an optimal algorithm in theory. But it is widely used in practice (the most rapid algorithm of construction the convex hull is considered **Quick Hull** [12]).

Divide-and-Conquer Method. Divide-and-conquer is a general and universal approach to the solution of wide class tasks. As for the task of the Delaunay triangulation this approach did not find a direct application in general. It is connected with the complication to divide a task into independent subtasks. Although for the case of the plane this approach gives an optimal theoretical algorithm [13]. Thus a basic problem consists in that, how to divide a task into two subtasks. There are two methods for its doing:

1. To divide a set into two subsets with a median on first coordinate. Thus the equal amount of points in both subsets is got, but much fixing is to be done on the point of confluence, as these subtasks are dependent.

2. To build a dividing chain of triangles and get two independent subtasks.

The first method is advantageous only for the case of a plane. In multidimensional spaces, fixing of structure can touch almost all built simplexes. The second approach gives quite good results in general case, one of its realizations is descedgeed under the name of **DeWall** [2]. It should be noted that doing a task by the method Divide and Conqueror is hardly to succeed to get *Output - Sensitive algorithm*.

Sweep Method. The idea of sweep line method consists in a moving imaginary line on a plane, stopped on certain points. Thus, passing the way on the points of events, a sweep line solves the set problem. In multidimensional spaces this algorithm has the type of a motion of a certain surface (more often of a hyperplane). This approach gives the best results of the Delaunay triangulation construction on a plane and is widely used in practice (Fortune's Algorithm[11]). However for obvious reasons this algorithm can‘t be adapted for a spatial case. To support the front hyperplane with the sweep line it is necessary to deal with $(d-1)$ -dimensional structures. That is why all operations for such structures acquire high calculable complexity and have a high complexity of implication.

Random Incremental Method. The basis of this approach consists in adding the set points one by one and fixing of triangulation [4,5]. The operation of fixing consists in consideration of ”suspicious” pairs of nearby simplexes and so-called ”flipping” operation for improper pairs.

The problem of this method is that in multidimensional spaces the flip operation is very complex and consists of many different cases. Another problem is that it is difficult to assess the algorithm complexity, because in different cases the amount of flips may be starting with several ones and ending with a complete change of previously constructed simplexes. Also the number of simplexes formed at a certain step can far exceed the number of resulting simplexes, that will increase radically the working time of the algorithm. This algorithm ”ungovernability” can somehow be compensated choosing the

V. Tereshchenko and D. Taran are with the Faculty of Cybernetics, National Taras Shevchenko University of Kyiv, Kyiv, Ukraine, 02095 e-mail: vtereshch@gmail.com, eqis.mail@gmail.com

Manuscript received February 15, 2012; revised February 29, 2012.

order of points bypassing. The order would be ideal when only the resulting simplexes of triangulation are added each time. It would turn this algorithm into the direct one. But this procedure is complicated and it is unclear how it can be found.

Direct algorithms. Direct algorithms build the final result step by step . In this case, the direct algorithm discovers another simplex of resulting triangulation at each step. In general, the algorithm consists of the following steps:

1. To find the original edge and add it to the line.
2. To get the next edge out of the turn.
3. To find a simplex of triangulation based on this edge.
4. If there is a simplex, add all other edges of the simplex to the turn.
5. To go to Step 2.

Unobvious here there are only steps 1 and 3. But step 1 is done once for the entire algorithm, that‘s why its optimization does not cause much concern. As for step 3, there is an obvious approach to its interpretation. It consists in reviewing the entire set of points and selecting the necessary one (this point forms the simplex of a minimum radius). This approach to doing this task gives us the slowest algorithm. There is a time estimate $O(mn)$ where m is number of resulting simplexes. So even on a plane this algorithm is quadratic. But unlike all other approaches the direct algorithm is exactly Output-Sensitive. There are several optimizations of finding the next simplex. The main is the grid method (the hash space points) and the method of bubble inflating. But these optimizations are unstable and in some cases may work for an indefinite time.

II. The research aim

As it is seen from the previous review of the existing algorithms for doing the sums, all the algorithms have some drawbacks and in certain situations their work time can be very high. Let‘s formulate what exactly we would like to receive from the new algorithm:

1) the optimal work time;
2) stability of work on any sets of points;
3) and the universality of multidimensional spaces.

As for these demands, a direct algorithm has high expectations, because it meets quite simply the 2) and 3) requirements. Although the first requirement is the most important, but the direct algorithm leaves much space for optimization. The proposed algorithm in this paper is the achievement of the optimum working speed of direct algorithms. An appropriate implementation of the algorithm and the analysis of its working time were done.

III. Algorithm

A. The general idea

As it was mentioned earlier, the offered algorithm is a direct one. This means that its general steps correspond to the steps of the direct algorithm. But each step is optimized to get the optimal working speed. The step of searching the next simplex underwent the major changes , as this step is the weakest point of the direct algorithm. There is an open edge at this point for which the top is to be found, forming together with the edge the resulting simplex of triangulation. The idea is to reject the review of failed points. Let‘s have at some stage a nominee for a successful point and consider the location of points that fit better, figure 1.

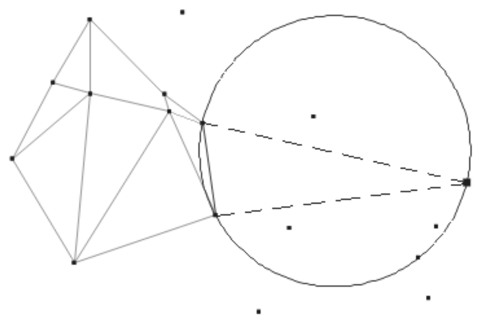
Fig. 1. A successful point and the bettered location of points.

The more optimal points are to be found inside the circle (sphere) circumscedgeing the considered simplex. Then having found a point inside we’ll narrow the area of further search. Continuing in this way we will find the point which has no other points inside the descedgeed circle. By the Delaunay triangulation definition this is the point that forms a triangle of triangulation. It is easy to notice that this method of search is not limited by the case of a plane. It is also suitable for multidimensional spaces.

B. Searching for the initial edge

The first step is to find the initial edge, where the first simplex will be built. The edge is called a face of simplex. In d-dimensional space simplex seems to be $d + 1$ point. The edges of this simplex are all subsets of points that are of d size. Obviously, all faces of the convex hull of the set of points belong to certain simplexes of triangulation of these points. Therefore, the initial edge can be any face of a convex hull. So to find the initial face we can use the same algorithm search Gift Wrapping (**Gift Wrapping** [13, 14]. The idea is to find consistent hyperplanes, each of which has one point of the convex hull more than the previous one. First find the point of the least first coordinate. This point clearly belongs to the convex hull. Fix the hyperplane perpendicular to the first axis coordinate. The normal of the plane $n = (1, 0, \ldots, 0)$. This will be the initial hyperplane F (figure 3).

Then one after another the next points are searched by the resulting face. Such a point is chosen which forms the largest angle built by the hyperplane on this step. Let‘s consider how to find the angle between the point and the hyperplane, figure 2:

Vector $\mathbf{a}$ is a perpendicular to the edge (p_1p_2), which lies in the hyperplane. Vector $\mathbf{n}$ is a normal to the hyperplane. $\mathbf{v_k}$ - is a vector from the new vertex to one of the edge’s vertexes. The cotangent of an angle can be computed using the following formula:

$$ctg(\varphi) = pr_{\mathbf{a}}\mathbf{v_k}/pr_{\mathbf{n}}\mathbf{v_k}$$

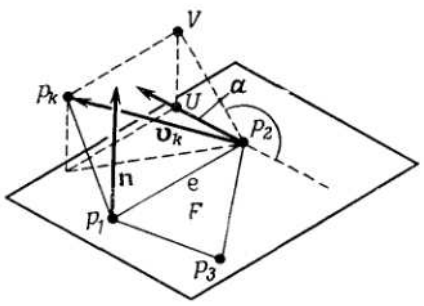

Fig. 2. Finding the angle between the point and the hyperplane.

The projection of vector $\mathbf{a}$ on vector $\mathbf{b}$ is computed as follows:

$$pr_{\mathbf{b}}\mathbf{a} = (\mathbf{a}, \mathbf{b})/|\mathbf{b}|$$

With cotangents of angles, the largest angle can be found , minimizing the value of the cotangent. Now let‘s consider in detail how to build vectors $\mathbf{a}$ and $\mathbf{n}$. At some stage we have hyperplane $F_j - 1$ built on the $j - 1$ points. There is also its normal n. Vector $\mathbf{a}$ must be a perpendicular to vector $\mathbf{n}$, vectors $\mathbf{v_1 v_i}$ ($j - 2$ vectors) and the axes of coordinates $x_{j+1}, x_{j+2}, \ldots, x_d$. These $d - 1$ correlations set the system of linear equations, which having been done we obtain vector $\mathbf{a}$. With vectors n and a the cotangents of all angles can be found, then find the next point v_j. Now, with the hyperplane F_j, it is necessary to recalculate normal $\mathbf{n}$. The normal is to be perpendicular to vectors $\mathbf{v_1 v_i}$ ($j - 1$ vector) and the axes of coordinates $x_{j+1}, x_{j+2}, , x_d$. Having done the corresponding system of lineare quations we get normal $\mathbf{n}$ to the new hyperplane p_j. Thus for $d - 1$ approaching we get the face of the convex hull.

The Gauss method is used for solving systems of linear equations. It is comfortable to implement and this is its main advantage. Not really the optimal time of its work in our case is not critical. So let‘s have a system of linear equations:

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ + a_{nn}x_n = b_n \end{array}$$

The first part of the algorithm results matrix $\mathbf{A} = (a_{ij})$ into a triangular form. For this the line with a nonzero coefficient is selected first for this. The place of this line is to be changed with the first one, and then divide all the coefficients on a_{11}. Thus we obtain a unit as the first coefficient of the first line. Reset then all the coefficients of the first column with the help of the first line. For this subtract the first row multiplied at ai1 from each line with number i ($i \geq 2$). Then similarly with the second line ”destroy” all the other lines. As a result we get the following system:

$$\begin{array}{l} x_1 + 0 + \ldots + 0 = b'_1 \\ 0 + x_2 + \ldots + 0 = b'_2 \\ \ldots \\ 0 + 0 + \ldots + x_n = b'_n \end{array}$$

Solving this system is found in an understandable way:

$$x* = (b'_1, b'_2, \ldots, b'_n)$$

It should be noted that this algorithm is general and we considered the square case for convenience. If there are less equations than variable ones (let them $m < n$), we will get half triangled matrix as result transformations. In this case all the variables $x_{m+1}, \ldots, x_n$ are independent. Assign them all the value 1, and then calculate all other variables considering this. Also, sometimes during the transformation a zero column can be obtained. This also means that the variable that corresponds to this column is independent. We can also choose 1 for its meaning.

C. Hashing edges

At each step of finding a new simplex, it is necessary to perform certain operations on the edges: add edges to the list of ”open”, to check the presence of edges in this list. As these operations are repeated many times, it is necessary to ensure their optimality. All marks on the open edges must be stored in hashes. Thus we get the speed of doing all the operations for time $O(1)$ we need. A edge is a list of d points that are stored by their numbers. Let us solve the task for N points. Then you can choose the hash function of the following form:

$$hash(e) = (N^0 e.v_1 + N^1 e.v_2 + N^2 e.v_3 + \ldots N^{d-1} e.v_d) mod P$$

Here P is some large prime number, which is caused by memory size, which we are ready to make a hash. With hashing function, edges can be recorded in the corresponding cells array size P. Searching for a particular edge one should simply check the corresponding cell on the presence of this edge. It should be noted that for avoiding differences it is necessary to sort the list of the edge’s vertexes. Otherwise, for different permutations of the same vertexes different hashes will be received.

D. Search for the next simplex

The subtasks is to find the vertex for an open edge, which together with that edge forms simplex of resulting triangulation. Modification is that, instead of searching all vertexes , find the desired vertex using modified *KD-tree*.

The search starts from an infinite search region $[-\infty; +\infty]^d$. So first let‘s find any point that can form a simplex with this edge. Now we can narrow the search region, since all the ”better” points for the given one are inside the hypersphere fixed around the current simplex (formed by the edge and point-challenger). The hypersphere (which is the region of the next search) is approximated to the hypercube.The larger dimensionality of space d is, the rougher this approximation is. Therefore, with increasing of dimension space the efficiency of the algorithm is lost. The search ends when no single point is found within the gotten region. This means that the chosen point is the vertex of simplex triangulation. The situation is possible when no point was found in the search process. This means that the edge is the face of the convex hull. In this case, this edge closes without adding the simplex. If the point was found, then the corresponding simplex is constructed and recorded into the resulting list of triangulation simplexes. In addition, d new edges simplex are attached to all open edges. When adding a new edge it is possible that this edge is in the queue already. In this case, it must be removed from the queue and marked as ”complete”. Hash is used for quick checking of the edge‘s availability as descedgeed in the previous section.

KD-tree [13, 15] is a structure used for fast search the points belonging to the query range. Query ranges are rectangles (parallelepipeds) with the sides parallel to axes of coordinates. It uses $O(n)$ memory. It is built for the time $O(nlogn)$. Each query is done on average for $O(logn)$, in the worst case, for $O(n^{1-1/d})$. In our case the search region should be narrowed in finding the point. The new region is considered as approximation of circumscedgeed sphere around the simplex-applicant to hypercube. Supposing, for finding the next point, we have a hypersphere centered at the point $c = (c_1, c_2, \dots, c_d)$ and radius R. The query range will be parallelepiped, parallel to the axes of coordinates. For each coordinate $k = 1, 2, ..., d$ limited by numbers $c_k - R$, $c_k + R$. Using KD-tree without changes does not provide with a pure logarithmic search time. In many cases, the time is linear, which makes the algorithm a simple direct algorithm (which is the slowest of all). But the next two optimizations will return the algorithm the desired speed.

The first optimization. Sometimes the search procedure does not result scienter. An example is the convex hull faces, for which there is no simplex, that closes the open edge. In this case, the search should review all the points and this will increase the working time of the algorithm significantly. But in some cases we can cut off the large search region without a detailed verification. It should be noted that there is 1 or 2 simplexes for every edge of triangulation. Open edges are the edges one of the simplexes of which has already been found, and the other one is to be found. Therefore every open edge actually has a direction - the points search is only in one half-space. That‘s why the search on regions that lie completely in the other semispace can be immediately discarded. Checking whether a parallelepiped belongs completely to a certain half-space is simple. To do this, check each of the vertexes belonging to this half-space. If they all belong to it - then the whole parallelepiped belongs to this half-space.

The second optimization. In the original KD-tree it does not matter from which subset to start the search if it should be continued on both. This is due to the fact that the query range is fixed, that‘s why the search must check both subsets. In our case, checking one of the subsets, the region may be narrowed and there will be no need in checking the other one. So you need to start the search with a more reliable subset. If the edge is wholly belongs to one of the subsets, it is advantageous to start the search from it. This is due to the fact that the best points in general are closer to the edge than others.

The update of the query range. When finding a new vertex-pretender the query range must be updated. The hypersphere circumscedgeing the simplex must be found. The simplex is given with $d + 1$ point. Let‘s write the system of equations for finding the center of the circumscedgeed hypersphere. This is the point, the distance from which to all points of simplex is the same:

$$(c_1 - p_{11})^2 + (c_2 - p_{12})^2 + \ldots + (c_d - p_{1d})^2 =$$
$$=(c_1 - p_{i1})^2 + (c_2 - p_{i2})^2 + \ldots + (c_d - p_{id})^2,$$
$$(2 \le i \le d + 1)$$

Reordering the variables we obtain the following equality:

$$c_1(2p_{11} - 2p_{i1}) + \ldots + c_d(2p_{1d} - 2p_{id}) =$$
$$=(p_{11}^2 - p_{i1}^2) + \ldots + (p_{1d}^2 - p_{id}^2)$$

Thus, we have a system with d $(2 \le i \le d + 1)$ linear equations for variables $c = (c_1, c_2, \dots, c_d)$. Solving it by the Gauss method, we obtain the solution - the center of circumscedgeed hypersphere. With the center the distance to one of the points of simplex can be calculated , the radius is obtained.

E. *The final processing*

Each time, getting a new simplex, it is necessary to write it into the resulting list simplex. It is also necessary to maintain the structure of the neighborhood. For this each of the open edges is put an according simplex, for which this edge is recorded in the line and hash. After finding the simplex, a closing edge, a mark on the neighbouring resulting simplex and the simplex responding to this edge is made . Also, adding a edge there may be a situation when it is already in the queue. It is necessary to make a mark on the neighbouring simplexes, that generated these edges, and the very edges should be removed from the line. Parallelly the Voronoi diagram of these points may be built. The vertexes of the diagram are the centers of the hyperspheres circumscedgeed around simplexes. They were computed in the search simplex process. The edges are the segments between the centers of neighbouring simplexes, figure 3.

IV. Estimation of the algorithm complexity

First of all, it should be considered the amount of memory used by the algorithm. Let‘s consider all the structures necessary for the work:

1. The set of all points is stored in the KD-Tree - $O(n)$.
2. Hash operations with the edges - $O(P)$.
3.The list the resulting simplex - $O(m)$.

So the memory evaluation required for the algorithm is $O(n + m + P)$.

Let‘s consider a sequence of basic operations needed to perform each step of the algorithm:

1. Searching for the initial edge (d repetitions).
1.1. The Gauss method for finding vector $\mathbf{a}$ - $O(d^3)$.
1.2. Finding the point with a maximum angle - $O(dn)$.
1.3. The Gauss method for finding a normal- $O(d^3)$.
2. Finding the next simplex (m repetitions):

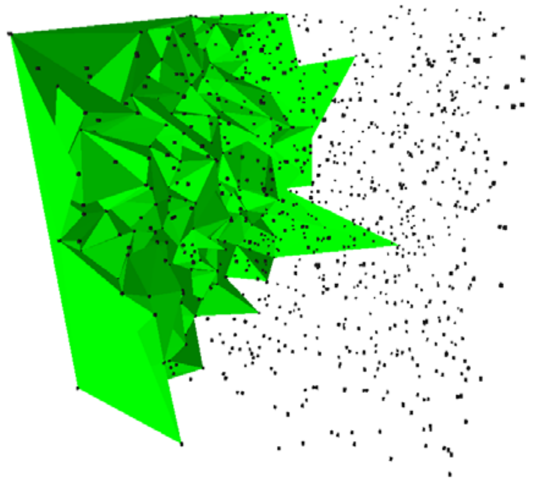

Fig. 3. Constructing the Delaunay Triangulation.

TABLE I
COMPARING WORK TIME OF ALGORITHMS

number of points	D-Range	Fortune	Calls	K
100000	2788	1413	27,52	1,973107
200000	5782	3110	29,04	1,859164
300000	8803	4708	29,4	1,869796
400000	12248	6777	30,91	1,807289
500000	14988	8472	30,35	1,769122
600000	18539	10831	32,46	1,711661
700000	21882	12545	32,4	1,744281
800000	24739	15466	31,91	1,599573
900000	28388	17602	32,69	1,612771
1000000	32512	20428	33,92	1,591541

2.1. Get the next edge out of the line - $O(1)$.

2.2. Finding the desired point - $O(n^{1-1/d})$ (on average $O(logn)$).

2.3. Adding the simplex into resulting list - $O(1)$.

2.4. The hash of new edges - $O(1)$.

Hence there is an estimate of the algorithm complexity in the worst case $O(2d^4+d^2n+mn^{1-1/d})$. But the complicated case is practically unattainable, and such assessment is due only to the complexity to estimate it less roughly. In practice, the expected complexity of the algorithm is more important. It leads to the following estimation of the algorithm complexity $O(2d^4+d^2n+mlogn)$. Assuming a constant space dimension, we obtain a more concise evaluation of the expected complexity $O(mlogn)$.

Comparing work time is given in Table 1. For reference let‘s take the fastest algorithm in the plane. For this the implementation of the Fortune algorithm [11]. Both algorithms are to be tested on one computer (usually a laptop with usual characteristics) under the same conditions.

In the first column the number of points for which the tests were conducted is specified. In the second and the third ones the specific time of both algorithms (in milliseconds) is given. The fourth column shows the average number of calls searching for the optimal point of building a new simplex. Coefficient

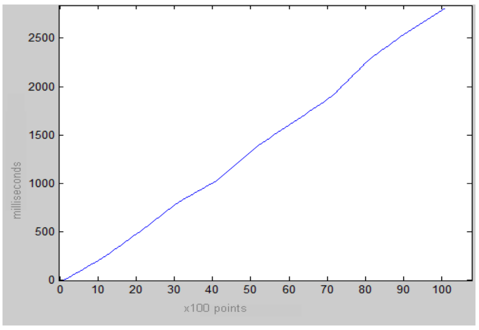

Fig. 4. The time work of D-Range algorithm in three dimensions space.

K indicates how many times the proposed algorithm is slower than the Fortune algorithm.

Despite expectations, both algorithms are close to linear time work. Paying attention to coefficient K it is evident that it monotonously decreases with increasing number of points. Some of the D-Range algorithm slowness is connected with the cost of time on the preprocessing (KD-tree and the search of the initial edge). But, unlike offered algorithm (D-Range algorithm) the Fortune algorithm is to work on the plane and is not adapted in a multidimensional space. Figure 4 shows schedule the time work of D-Range algorithm in three dimensions space:

V. Conclusions

The result of this work is a new efficient algorithm that is applicable to the cases of multidimensional problem solving of Delaunay triangulation or the Voronoi diagram construction. Its practical implementation is confirmed. Also, paying attention to set requirements, you can specify the stability of the algorithm work and a low dependence on any specific sampling points. The algorithm behaves identically on any sets. Although the algorithm is not theoretically optimal, it is optimal in terms of the expected speed.

References

[1] J.E. Goodman and J. O'Rourke, eds. *Handbook of Discrete and Computational Geometry.* Second Edition, Chapman and Hall/CRC Press, 2004.

[2] P. Cignoni, C. Montani, R. Scopigno. *DeWall: A Fast Divide and Conquer Delaunay Triangulation Algorithm in E^d.* Computer-Aided Design Vol. 30 (5):333-341, 1998.

[3] M. de Berg, O. Cheong, M.van Kreveld, M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 2008.

[4] L. Guibas, D. Knuth, M. Sharir. *Randomized incremental construction of Delaunay and Voronoi diagrams.* Algorithmica 7:381-413,1992.

[5] H. Edelsbrunner, S. Nimish. *Incremental Topological Flipping Works for Regular Triangulations.* Algorithmica 15 (3): 223-241,1996.

[6] M. Caroli, M. Teillaud. *Delaunay triangulations of point sets in closed euclidean d-manifolds.* In Proc. of the 27th annual ACM symposium on Computational geometry, pages 274-282, 2011.

[7] M. Hoffmann., Y. Okamoto. *The minimum weight triangulation problem with few inner points.* Computational Geometry. 34 (3):149-158, 2006.

[8] J. Gudmundsson, H. Haverkort, and M. van Kreveld. *Constrained higher order Delaunay triangulations.* Comput.Geom. Theory Appl., 30:271-277, 2005.

[9] R.I. Silveira, M. van Kreveld. *Towards a Definition of Higher Order Constrained Delaunay Triangulations.* In proc. 19th Canadian Conference on Computational Geometry, pages 322-337, 2007.

[10] T. de Kok, M. van Kreveld and M. Löffler. *Generating realistic terrains with higher-order Delaunay triangulations.* Comput. Geom. Theory Appl., 36:52-65,2007.

[11] S. Fortune. *A sweepline algorithm for Voronoi diagrams.* Algorithmica, 2:153-174,1987.

[12] C.B. Barber, D.P. Dobkin, and H.T. Huhdanpaa. *The Quickhull algorithm for convex hulls.* ACM Trans. on Mathematical Software, 22(4):469-483, 1996.

[13] F. Preparata and M.I. Shamos. *Computational Geometry: An introduction.* Springer-Verlag, Berlin, 1985.

[14] D. R. Chand and S. S. Kapur. *An algorithm for convex polytopes.* JASM 17(1): 78-86, 1970.

[15] J. L. Bentley. *Muldimensional binary search trees used for associative searching.* Communications of the ACM 18: 509-517, 1975.

Vasyl Tereshchenko In 1986 graduated from the Mathematics and Mechanics Faculty of Kyiv National Taras Shevchenko University. In 1992 graduated from graduate school and in 1993 defended PhD dissertation on the degree C.Sci. (Phys-Math.). In 2011 I defended a dissertation for the degree a doctor of Phys.-Math. sciences (theoretical bases of computer science and cybernetics).

Since 1994 - Associate Professor Faculty of Cybernetics KNTSU. Lecturer in computer graphics and in computational geometry, and also in databases and in the theory of algorithms.