Hybrid Walking Point Location Algorithm

Roman Soukal
Martina Málková
Tomáš Vomáčka
Ivana Kolingerová
Department of Computer Science and Engineering
University of West Bohemia
Plzen, Czech Republic
soukal@kiv.zcu.cz, mmalkov@kiv.zcu.cz, tvomacka@kiv.zcu.cz, kolinger@kiv.zcu.cz

***Abstract*—Finding which triangle in a planar triangular mesh contains a query point (so-called point location problem) is one of the most frequent tasks in computational geometry. Therefore, using an algorithm with the lowest possible complexity is appropriate. However, such complexity may be achieved only by using additional data structures, leading into algorithms that are more difficult to implement and have additional memory demands. A possible solution is to use walking algorithms, which are easier to implement. They either do not require any additional memory, or require only a small portion of it in order to achieve the lowest possible complexity as well. In this paper, we propose a new walking algorithm combining two existing approaches to provide speed, robustness and easy implementation, and compare it with the fastest representatives of walking algorithms. Experiments proved that our algorithm is faster than the fastest existing visibility and straight walk algorithms, and depending on the character of input data, either as fast as the orthogonal walk algorithms or faster.**

Keywords*-*Algorithm design and analysis; Computational geometry; Computer graphics.

I. INTRODUCTION

Point location problem is a very frequent task in computational geometry problems, such as triangulation construction, morphing and terrain editing. In this text, we focus on point location algorithms for triangular meshes, since triangular meshes are the most common way of data representation and its manipulation. Other representations, such as convex or non-convex polygonal meshes, can be triangulated first to use these algorithms. The algorithms can also be used for terrain models represented by triangular meshes without any preprocessing only by not using the height information during the location.

The point location problem is defined as follows. For a given planar triangular mesh and a query point, the task is to find which triangle from the mesh geometrically contains the query point. Algorithms solving this problem can be divided into two groups: algorithms with and without additional data structures. The former concentrate on having the lowest time complexity possible, in this case $O(\log n)$ per query point (n is the number of vertices in the mesh). Despite their low complexity, these algorithms have some disadvantages: they have additional memory demands, they are more difficult to implement, and they are often problematically modified to cover adding or deleting vertices. The latter group tries to avoid these disadvantages, but has a slightly higher, but still sublinear, complexity.

The name of walking algorithms has arisen from their operating principle. They use the triangle neighborhood relations to go via the triangles between the starting triangle and the one containing the query point. Such point location process is called a *walk*. The starting triangle may be arbitrary, however, its clever selection may radically shorten the length of the walk, therefore we will cover this topic as well. Walking algorithms do not need any additional data structures, they use only the neighborhood relations in the mesh, thus often they are more often chosen than the optimal time complexity solutions.

There exist several walking algorithms solving the location process, some are robust, others faster. In this paper, we propose a new walking algorithm combining two existing solutions in order to gain speed from the faster and still remain robust. The main idea of our approach is to compute such a transformation that the line connecting a selected vertex of the starting triangle and the query point is parallel with x-axis. This transformation is then used for the tested points throughout the walk to enable a cheap comparison of their position with respect to this line. Surprisingly, despite the use of transformations, the walk is still fast, because only the query point and the tested points (one per visited triangle) are transformed. Since the walk goes straight between the starting triangle and the query point, it cannot cross the border of a convex triangular mesh, which contributes to its robustness.

Section II presents the existing walking algorithms and a sophisticated selection of the starting triangle for the walk. Section III describes our new proposed algorithm, Section IV shows experiments comparing our solution with the existing algorithms. Section V summarizes the characteristics of our algorithm.

II. Overview

Point location by walking algorithms usually works in two steps: (1) selection of the initial triangle for the walk, and (2) using the neighborhood relationships between the triangles *(walking)* to find the target triangle, containing the query point.

A sophisticated selection of the initial triangle may radically improve the speed of the process. One way is to use some additional knowledge about the data, i.e., if we know the range of the mesh vertices coordinates, we can start all the locations in a triangle containing the point lying in the middle of the triangular mesh, used for instance by [1]. Or, we may know that the next query point will be close to the last one, in which case the best solution is to use the target triangle from the last location as the initial one for the next [11], [17].

Without any knowledge about the data, we may select the initial triangle as the nearest triangle from a set A of randomly chosen triangles from T, where $\|A\| \ll \|T\|$ [9]. Devroye et al. in [4] showed that such an improved straight walk achieves $O(\sqrt[3]{n})$ time complexity per one search for uniformly distributed points, Zhu in [18] came to the same complexity for the remembering stochastic walk.

Other solutions use some additional memory: [10] simplifies the triangular mesh and locate the points in the simplified version first, [13] introduces a bucketing method, which uses a uniform grid to quickly find a proper initial triangle. Some algorithms [14], [16] try to avoid the sensitivity of the original bucketing method on data uniformity by using adaptive structures instead of a uniform grid.

When we know the initial triangle, the walk may proceed. There are several algorithms solving this step. They can be divided into three groups: visibility, straight and orthogonal walks, according to the style how they determine the way of the walk.

Visibility walks use local "visibility" tests to determine the way of their walk. These tests look for such an edge that defines a line separating the query point and the third vertex of the triangle. The walk then moves across this edge to the neighborhood triangle.

The first visibility walk algorithm is called Lawson's oriented walk [7]. The algorithm starts in the initial triangle and uses the 2D orientation test to move to its neighbors until it reaches the query point:

$$orientation2D(\mathbf{t}, \mathbf{u}, \mathbf{v}) = \begin{vmatrix} u_x - t_x & v_x - t_x \\ u_y - t_y & v_y - t_y \end{vmatrix} \qquad (1)$$

where points $\mathbf{t}, \mathbf{u}$ define an oriented line and $\mathbf{v}$ is the tested point. In each triangle, the algorithm tests the triangle edges until it finds an edge, where the third vertex of the triangle lies on the opposite side of the edge than the query point. Then, it crosses such an edge to the next triangle. If such an edge does not exist, the triangle containing the query point has been found.

The Lawson's oriented walk algorithm tests edges of the current triangle in a deterministic order, depending on the arrangement of edges in triangles, generated during the construction of the triangulation. This leads to the fact that the walk may loop for non-Delaunay triangulations [3], [15]. [3] proposed an algorithm avoiding the loop by choosing the edges of the current triangle in a random order. This modification is called *stochastic*. Furthermore, since it is not necessary to test the edge incident to the previous triangle, the process was sped up by remembering this edge and skipping the test. This modification is called *remembering* and brings a significant speedup, since only one or two orientation tests are needed instead of up to three (except of course the first triangle, where all the three edges may be tested). As the second test is done only to find out whether we are in the target triangle, [6] suggested to speed up the process even more by testing only one edge for the first k steps. If the triangle is found within this k steps, we circle around it, so it is necessary to determine a proper k based on the input.

Straight walk algorithms do not use only the local comparisons to determine the way of the walk, but they use an oriented line $\overrightarrow{\mathbf{pq}}$, connecting one point $\mathbf{p}$ (its choice depends on the particular solution) of the starting triangle with the query point $\mathbf{q}$ and then pass all triangles intersected by this line.

The standard straight walk algorithm [3], [8] works in two steps: an initialization step and a straight walk step. In the initialization step, a point $\mathbf{p}$ is chosen as any of the starting triangle vertices and a triangle intersected by the line segment $\overrightarrow{\mathbf{pq}}$ is found. The walk starts from this triangle, and in each step, it uses such a vertex of the current triangle that is opposite to the edge used to enter this triangle and finds out its position with respect to $\overrightarrow{\mathbf{pq}}$ (using the 2D orientation test). Based on its position, it selects which edge it should cross to the next triangle. Before crossing, it computes the orientation test for the point $\mathbf{q}$ with respect to this edge. If the point $\mathbf{q}$ is on the inner side of the edge, the final triangle has been found. Otherwise, the walk crosses the edge to the next triangle and continues.

[12] proposed a modification of this method simplifying the initialization step and speeding up the algorithm. Instead of the 2D orientation tests, an implicit line equation of $\overrightarrow{\mathbf{pq}}$ is used. The equation is precomputed in the initialization step along with an implicit equation of a line normal to $\overrightarrow{\mathbf{pq}}$ in $\mathbf{q}$, which is used to determine if there is a possibility that the target triangle has been found. However, the location of the target triangle is not precise, so the remembering stochastic walk algorithm is used for the short, final location (usually about 2 triangles, for more detail see Section IV).

Orthogonal walks first navigate along one coordinate axis and then along the other, which makes the local tests cheaper, since only components of the coordinates are compared during the walk. The walk is usually longer than other walks, but its tests are much cheaper, which results in a faster location.

The original orthogonal walk [3] consists of three steps: an initialization step, a walk along the x-axis, and a walk along the y-axis. During the initialization step, any vertex $\mathbf{p}$ of the starting triangle is chosen and two lines are defined: a horizontal line, containing this vertex and parallel to the x-axis, and a vertical line, containing the query point and parallel to the y-axis. A triangle intersected by the horizontal line and containing $\mathbf{p}$ is found. The walk then follows this line in a similar manner as the straight walk, but with component comparisons only: y-values are used to determine the next triangle, x-values are used to determine if the triangle intersected by the vertical line is found. When it is done, the walk continues by following the vertical line, where y-values are used to determine the next triangles and x-values to determine the target triangle containing the query point. For a few final triangles in each walk direction, it uses 2D orientation tests for a precise location.

The original orthogonal walk has a significant drawback: it does not solve the case, when the walk crosses the border of the triangular mesh. If the horizontal walk crosses the border, the vertical walk starts from the last triangle and usually does not find the correct triangle.

[1] proposes a modification, where the initialization step is simplified, and the walk is sped up by using fewer comparisons. Instead of two comparisons determining when the target triangle may be found, in which case the original walk uses the 2D orientation test to be sure, it uses only one comparison. This way the walk may stop too early, but since the previous tests were not precise anyway, slightly less precision is not so important, and the Remembering stochastic walk algorithm (RSW - details see in [3]) is used for the last few steps. The use of RSW also solves the problem with the possibility of crossing borders of the triangular mesh, because in such a case, the algorithm does not end at the correct triangle, but the final location with RSW does. However, the final walk is then longer and slows down the whole location.

Figure 1 shows such a situation: the horizontal walk reaches the border at the triangle γ. Here, the algorithm switches to the vertical walk, where the vertical line is moved to the last tested point. The vertical walk stops at the triangle δ, because the y component of $\mathbf{s_j}$ is higher then the one of $\mathbf{q}$. This should mean that the triangle contains $\mathbf{q}$ or is close to the target triangle, but as the horizontal walk had to stop early, the triangle is still quite far. The original algorithm would stop here and would not locate the right triangle. Its modification uses the RSW algorithm at this step and therefore locates the right triangle, but for a higher time cost, because the RSW algorithm has more expensive tests.

The complexity of the presented algorithms has been proved only for their basic representatives, and also either

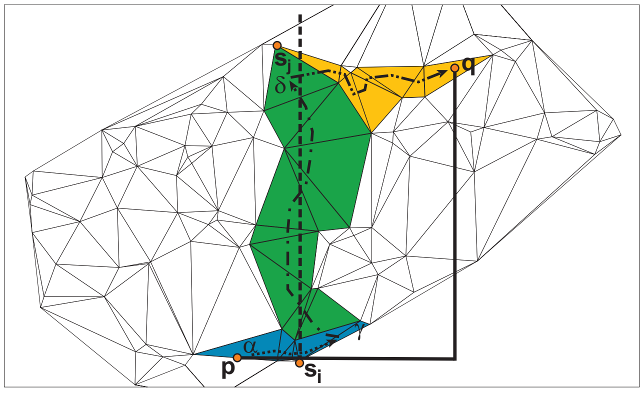

Figure 1. A case when the orthogonal walk crosses the border of the triangular mesh (a dotted line denotes the horizontal walk, a chain line denotes the vertical walk, a chain line with double dots denotes the final location by the RSW algorithm; the dashed and solid line denote the lines controlling the walk).

the time complexity or the number of visited triangles has been derived (note that these two values do not necessarily correspond). The stochastic walk has been shown to need $O(\sqrt{n \cdot \log n})$ expected time for uniform data [18]. The straight walk has been proved to visit $O(\sqrt{n})$ triangles in the expected case and uniform distribution [5], [10], a bound based on [2] shows that the orthogonal walk has similar complexity as the straight walk [4], [10].

III. The Proposed Algorithm

The algorithm described in this paper is called a *Hybrid walk*, because it combines the basic idea of two walking strategies: straight and orthogonal walk, to keep the advantages of both. The algorithm works in four steps. In the initialization step, a point $\mathbf{p}$ is chosen as any of the vertices of the starting triangle, and a transformation matrix M is set to transform the tested points in a way as if the line $\overrightarrow{\mathbf{pq}}$ was parallel with the x-axis and $p'_x < q'_x$ (prime symbol denotes the transformed vertices, i.e., $\mathbf{p}' = \mathbf{p} \cdot M$). From this point, each tested vertex is first transformed, and then only its coordinate components are compared in the tests. In the next step, a triangle intersected by $\overrightarrow{\mathbf{pq}}$ and containing $\mathbf{p}$ is found. The third step is the walk itself, following the line $\overrightarrow{\mathbf{pq}}$. The final, short location (about 2 triangles) is done by the RSW algorithm.

There exist many transformations meeting the previous requirements, to achieve the fastest computation possible, we chose the transformation matrix combining rotation by angle φ and scaling by k. The variables φ and k are determined by the mutual position of the points $\mathbf{p}, \mathbf{q}$. The equation used for transforming a vertex v is as follows:

$$\mathbf{v}' = (v_x, v_y) \cdot \begin{pmatrix} k \cdot \cos\varphi & k \cdot \sin\varphi \\ -k \cdot \sin\varphi & k \cdot \cos\varphi \end{pmatrix} \qquad (2)$$

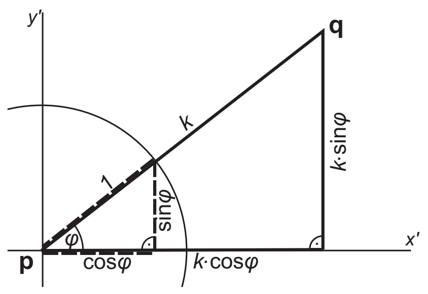

Figure 2. Components of the transformation matrix (x' is parallel with the x-axis, y' with the y-axis).

Computing sine and cosine of φ would be slow, but it can be avoided by using triangle similarity (see Figure 2), so $q_y - p_y$ and $q_x - p_x$ are computed instead of $k \cdot \sin\varphi$ and $k \cdot \cos\varphi$. Note that this fast computation of sine and cosine is the reason why k was introduced, otherwise $k = 1$ would be used.

Now, let us describe the algorithm in detail (for pseudocode, see Algorithm 1). In further text, we assume that the vertices of the triangles are in a CCW order. During the initialization step, the point $\mathbf{p}$ needs to be selected as simply and fast as possible, because the tests done after the transformation are cheaper. Therefore, we choose any of the vertices of the starting triangle as $\mathbf{p}$ (as is done in [3]) and compute the transformation matrix.

When the transformation matrix is set, the triangle δ intersected by $\overrightarrow{\mathbf{pq}}$ and containing $\mathbf{p}$ needs to be located. This is done in a similar manner as in the straight walk, but with cheaper tests thanks to the use of transformations. We select a different vertex than $\mathbf{p}$ from the starting triangle, let us denote it $\mathbf{r}$, and compute the y-coordinate of its transformed version $\mathbf{r}'$. Then we turn around $\mathbf{p}$ until we find the desired triangle. In each triangle, we determine which edge we should cross to the neighborhood triangle by comparing r'_y with q'_y. Therefore, during this step, only one transformed coordinate has to be computed and one coordinate component comparison per triangle is performed.

The walk starts from the triangle δ and follows the line $\overrightarrow{\mathbf{pq}}$. In each triangle τ_i with vertices $\mathbf{l_i}, \mathbf{r_i}, \mathbf{s_i}$, the edge $\epsilon_{l_i r_i}$ is used to cross to this triangle, $\mathbf{l_i}$ is to the left of $\overrightarrow{\mathbf{pq}}$ and $\mathbf{r_i}$ to the right. The edge to cross is determined by comparing the y components of $\mathbf{s_i'}$ and $\mathbf{q}'$. If $\mathbf{s_i'}$ is above $\overrightarrow{\mathbf{p}'\mathbf{q}'}$, we cross the edge $\epsilon_{r_i s_i}$, otherwise, we cross the edge $\epsilon_{l_i s_i}$. Note that if the line leaves the triangle through its vertex, the walk may continue by both $\epsilon_{r_i s_i}$ and $\epsilon_{l_i s_i}$, in the pseudocode we choose the latter one. Also the x-components of $\mathbf{s_i'}$ and $\mathbf{q}'$ are compared to end the walk if there is the possibility that the target triangle has been found. Therefore, during this step, both transformed components of s_i have to be computed and two component comparisons per triangle are performed.

The triangle in which the walk ends does not necessarily contain the query point, but it is usually very close to the one that does. The final location is performed by the RSW algorithm (as can be seen in Section IV, it visits about 2 triangles in average). For its implementation, we used the pseudocode from [3].

```
Input: the query point $\mathbf{q}$, the chosen starting triangle $\alpha \in T$
Output: the triangle $\omega$ which contains $\mathbf{q}$
// initialization step
triangle $\tau = \alpha = \mathbf{lrs}$;
point $\mathbf{p} = \mathbf{s}$;
if $\mathbf{p} = \mathbf{q}$ then return $\tau$;
vector $\mathbf{a} = \mathbf{q} - \mathbf{p}$;
// $k = \|\mathbf{a}\|$
double $kcos = a_x$;
double $ksin = a_y$;
$q'_x = q_x \cdot kcos - q_y \cdot ksin$;
$q'_y = q_x \cdot ksin + q_y \cdot kcos$;
double $r'_y = r_x \cdot ksin + r_y \cdot kcos$;
if $r'_y > q'_y$ then
    // $\mathbf{r}$ is above $\overrightarrow{\mathbf{pq}}$
    double $l'_y = l_x \cdot ksin + l_y \cdot kcos$;
    while $l'_y > q'_y$ do
        $\tau$ = neighbor of $\tau$ trough $\epsilon_{pl}$;
        $\mathbf{r} = \mathbf{l}$;
        $\mathbf{l}$ = vertex of $\tau$, where $\mathbf{l} \neq \mathbf{p}, \mathbf{l} \neq \mathbf{r}$;
        $l'_y = l_x \cdot ksin + l_y \cdot kcos$;
    end
    $\mathbf{s} = \mathbf{l}$; $\mathbf{l} = \mathbf{r}$; $\mathbf{r} = \mathbf{p}$;
else
    // $\mathbf{r}$ is below $\overrightarrow{\mathbf{pq}}$
    repeat
        $\tau$ = neighbor of $\tau$ trough $\epsilon_{pr}$;
        $\mathbf{l} = \mathbf{r}$;
        $\mathbf{r}$ = vertex of $\tau$, where $\mathbf{r} \neq \mathbf{p}, \mathbf{r} \neq \mathbf{l}$;
        $r'_y = r_x \cdot ksin + r_y \cdot kcos$;
    until $r'_y > q'_y$;
    $\mathbf{s} = \mathbf{r}$; $\mathbf{r} = \mathbf{l}$; $\mathbf{l} = \mathbf{p}$;
end
// now $\overrightarrow{\mathbf{pq}}$ intersects $\tau$
// walk step - following the line segment $\overrightarrow{\mathbf{pq}}$
$s'_x = s_x \cdot kcos - s_y \cdot ksin$;
while $s'_x < q'_x$ do
    $s'_y = s_x \cdot ksin + s_y \cdot kcos$;
    if $s'_y < q'_y$ then $\mathbf{r} = \mathbf{s}$; else $\mathbf{l} = \mathbf{s}$;
    $\tau$ = neighbor of $\tau$ trough $\epsilon_{lr}$;
    $\mathbf{s}$ = vertex of $\tau$ where $\mathbf{s} \neq \mathbf{r}, \mathbf{s} \neq \mathbf{l}$;
    $s'_x = s_x \cdot kcos - s_y \cdot ksin$;
end
return $remembering\_stochastic\_walk(\mathbf{q}, \tau)$;
```

Algorithm 1: Hybrid Walk

IV. Experimental results

For the testing purposes, we implemented the proposed algorithm and the previous algorithms in Java with double precision floating point arithmetic. The algorithms were tested on Intel Q6600 2,40GHz. Based on the tests performed on different types of triangulations, we chose Delaunay triangulation as a sufficient representative. The tests were performed on triangulations of many different datasets, which were of three different types: randomly distributed points in the unit square, the real geodetic data from land registers and LIDAR data.

We selected the fastest of the existing algorithms and compared them with our proposed solution. The selected algorithms were: Remembering stochastic walk (RSW),

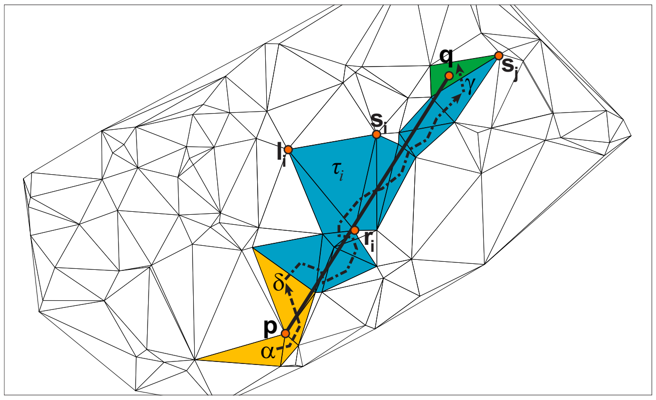

Figure 3. A hybrid walk example (a dashed line denotes the initialization step, a chain line denotes the walk and the final location by RSW algorithm is marked by a dotted line).

Normal Straight Walk (NSW), Improved Orthogonal Walk (IOW) and our Hybrid Walk (HW). Remembering walk is faster than its modification RSW, however, it may loop for other than Delaunay triangulations, thus to be objective, we did not include it in our test results.

In each test, 10^7 pairs of the initial triangle and the target point were generated randomly, and the average number of the tested properties were computed. Such properties were: the length of the walk (#Δ), the number of the tests (#tests) and the time per one location (t[μs]). The properties #tests and #Δ consist of two values for some algorithms. The former value concerns the walk, the latter concerns the final location performed by RSW.

Table I contains selected results of the tests for a triangular mesh enclosed in a rectangle preventing the orthogonal walk from crossing the border of the triangular mesh. Note that the number of triangles visited by our algorithm is about the same as in NSW, because both algorithms visit triangles intersected by the line connecting a point from the starting triangle with the query point, the only difference is in the particular selected point. However, tests done in each triangle by NSW are slower than by our algorithm, therefore the time per one location is higher. RSW algorithm is the slowest because of the 2D orientation tests and randomization done in each step. We included it in the tests as a representative of visibility walk group, but especially because it is used for the final location in all NSW, IOW and HW algorithms.

Table II compares our algorithm with IOW for randomly distributed points in a rectangle 2:1, rotated by $\pi/6$, which aims to resemble a more realistic situation, where some particular walks cross the border of the triangular mesh. It can be seen that our algorithm is the most suitable for such data that are not enclosed in a shape preventing the walk from crossing the border of the triangular mesh. Even a small percentage of walks leaving the triangular mesh slows down the whole location process in a way that our algorithm is faster. With a growing percentage of such walks, our algorithm becomes significantly faster. Recall Figure 1, providing an explanation for this behavior. Each walk leaving a triangular mesh in OW leads into a longer final walk done by RSW algorithm, which is slower (see Table I).

Even for the cases, when the walk does not cross the triangular mesh border (Table I), our algorithm has comparable results to OW, particularly for a uniform distribution its speed is similar or better. Moreover, its implementation is simpler than the one of OW, because our algorithm does not have four different cases which need to be solved separately.

V. Conclusion

We presented a new walking algorithm, combining the basic idea of two walking strategies (straight and orthogonal walk). Experiments proved that our algorithm is faster than the fastest existing visibility and straight walk algorithms, and comparable with orthogonal walk algorithms. If there is even a small percentage of walks that cross the boundary of the triangular mesh, our algorithm becomes faster than the orthogonal walk. Furthermore, its implementation is simpler, because the problem does not split into cases which has to be solved separately, as it is for the orthogonal walk.

Acknowledgment

The authors would like to thank their colleague Martin Maňák for his feedback and inspiring discussions. This work has been supported by the Grant Agency of the Czech Republic under the research project 201/09/0097 and by the project SGS-2010-028.

References

[1] Star-shaped polyhedron point location with orthogonal walk algorithm. *Procedia Computer Science*, 1(1):219–228, 2010.

[2] Jean-Daniel Boissonnat and Monique Teillaud. On the randomized construction of the Delaunay tree. *Theoretical Computer Science*, 112(2):339 – 354, 1993.

[3] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. In *Proceedings of the 17th Annual Symposium on Computational Geometry*, pages 106–114, 2001.

[4] L. Devroye, E. P. Mucke, and Binhai Zhu. A note on point location in Delaunay triangulations of random points, 1998.

[5] P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21:168–173, 1978.

[6] I. Kolingerová. A small improvement in the walking algorithm for point location in a triangulation. In *Proceedings of the 22nd European Workshop on Computational Geometry*, pages 221–224, 2006.

[7] C. L. Lawson. *Mathematical Software III; Software for C1 Surface Interpolation*, pages 161–194. Academic Press, New York, 1977.

Algorithm	#Δ	#test	t[μs]	#Δ	#test	t[μs]	#Δ	#test	t[μs]
	ϕ per located point			ϕ per located point			ϕ per located point		
Real geodetic data from land registers									
	4897 vertices (9774 Δ)			15824 vertices (31642 Δ)			70437 vertices (140868 Δ)		
RSW	92.09	122.18	6.24	158.1	208.45	15.19	321.27	417.88	64.29
NSW	87.46 + 2.29	174.02 + 4.28	3.92	149.16 + 2.43	297.31 + 4.42	10.75	293.4 + 2.93	585.8 + 4.9	49.32
IOW	94.16 + 4.33	188.32 + 7.03	3.30	176.52 + 2.69	353.04 + 5.19	8.40	319.04 + 3.48	638.08 + 6.35	28.72
HW	87.46 + 2.29	174.02 + 4.28	3.52	149.21 + 2.44	297.83 + 4.81	9.31	293.6 + 2.94	594.6 + 5.57	39.69
LIDAR									
	34932 vertices (69858 Δ)			313348 vertices (626690 Δ)			3722068 vertices (7444130 Δ)		
RSW	205.3	266.18	25.59	647.23	830.46	137.78	2563.63	3277.01	880.98
NSW	189.99 + 1.76	378.98 + 3.76	15.98	598.62 + 1.92	1196.24 + 3.92	105.57	2385.51 + 2.03	4770.03 + 4.02	659.48
IOW	246.61 + 1.75	493.22 + 3.82	13.74	801.88 + 1.75	1603.76 + 3.83	82.29	2886.95 + 2.0	5773.9 + 4.21	448.46
HW	187.61 + 1.76	374.7 + 3.83	14.09	596.38 + 1.92	1192.16 + 4.07	89.42	2385.69 + 2.03	4770.78 + 4.23	530.01
Randomly distributed points in the unit square									
	10^4 vertices (19994 Δ)			10^5 vertices (199994 Δ)			10^6 vertices (1999994 Δ)		
RSW	115.19	150.58	7.44	362.82	468.62	71.31	1144.86	1472.05	287.63
NSW	107.23 + 1.76	213.45 + 3.76	4.87	339.75 + 1.76	678.51 + 3.76	53.74	1073.59 + 1.76	2146.17 + 3.76	220.92
IOW	135.35 + 1.76	270.71 + 3.84	4.61	432.08 + 1.76	864.15 + 3.84	49.11	1371.06 + 1.76	2742.12 + 3.84	188.51
HW	108.5 + 1.76	216.4 + 3.84	4.29	342.9 + 1.76	685.23 + 3.84	49.09	1065.25 + 1.76	2129.88 + 3.85	191.13

Table I
COMPARISON OF THE SELECTED ALGORITHMS WITH RANDOMLY CHOSEN α

Algorithm	#Δ	#test	t[μs]	#Δ	#test	t[μs]	#Δ	#test	t[μs]
	ϕ per located point			ϕ per located point			ϕ per located point		
Randomly distributed points in rotated rectangle 2x1									
	10^4 vertices (19994 Δ)			10^5 vertices (199994 Δ)			10^6 vertices (1999994 Δ)		
IOW	142.17 + 17.45	284.34 + 24.49	5.93	451.16 + 52.23	902.33 + 69.96	61.35	1419.15 + 150.07	2838.3 + 197.61	239.87
HW	123.07 + 1.77	245.46 + 3.85	4.88	385.05 + 1.76	769.42 + 3.84	55.13	1218.09 + 1.76	2435.55 + 3.84	217.44

Table II
COMPARISON OF THE SELECTED ALGORITHMS ON A RECTANGLE ROTATED BY $\pi/6$ WITH RANDOMLY CHOSEN α

[8] Kurt Mehlhorn and Stefan Näher. Leda: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[9] E. P. Mücke, I. Saias, and B. Zhu. Fast randomized point location without preprocessing in two and three-dimensional Delaunay triangulations. In *Proceedings of the 12th Annual Symposium on Computational Geometry*, volume 26, pages 274–283, 1996.

[10] K. Mulmuley. Randomized multidimensional search trees: Dynamic sampling. In *Proceedings of the 7th Annual Symposium on Computational Geometry*, pages 121–131, 1991.

[11] S. W. Sloan. A fast algorithm for constructing Delaunay triangulations in the plane. *Advances in Engineering Software*, 9(1):34–55, 1987.

[12] Roman Soukal and I. Kolingerová. Straight walk algorithm modification for point location in a triangulation. In *EuroCG'09: Proceedings of the 25th European Workshop on Computational Geometry*, pages 219–222, Brussels, Belgium, 2009.

[13] P. Su and R. L. S. Drysdale. A comparison of sequential Delaunay triangulation algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 61–70, 1995.

[14] B. Žalik and I. Kolingerová. An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. *International Journal of Geographical Information Science*, 17(2):119–138, 2003.

[15] Frank Weller. On the total correctness of Lawson's oriented walk. In *Proceedings of the 10th International Canadian Conference on Computational Geometry*, pages 10–12, 1998.

[16] M. Zadravec and B. Žalik. An almost distribution independent incremental Delaunay triangulation algorithm. *The Visual Computer*, 21(6):384–396, 2005.

[17] Sheng Zhou and Christopher B. Jones. HCPO: an efficient insertion order for incremental Delaunay triangulation. *Information Processing Letters*, 93(1):37–42, 2005.

[18] Binhai Zhu. On lawsons oriented walk in random delaunay triangulations. In Andrzej Lingas and Bengt Nilsson, editors, *Fundamentals of Computation Theory*, volume 2751 of *Lecture Notes in Computer Science*, pages 222–233. Springer Berlin / Heidelberg, 2003.