Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Disertační práce

2015 **Ing. Roman Soukal**

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

ALGORITMY VYHLEDÁVÁNÍ BODŮ PROCHÁZKOU

Ing. Roman Soukal

disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitel: Prof. Dr. Ing. Ivana Kolingerová

Katedra: Katedra informatiky a výpočetní techniky

Plzeň 2015

University of West Bohemia in Pilsen
Faculty of Applied Sciences

WALKING LOCATION ALGORITHMS

Ing. Roman Soukal

Doctoral Thesis
submitted in partial fulfillment of the requirements
for a degree of Doctor of Philosophy
in Computer Science and Engineering

Supervisor: Prof. Dr. Ing. Ivana Kolingerová

Department: Department of Computer Science and Engineering

Pilsen 2015

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni. Prohlašuji, že tuto práci jsem zpracoval samostatně s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této disertační práce.

Abstrakt

Vyhledávání takového trojúhelníku trojúhelníkové sítě, který obsahuje požadovaný bod (tzv. problém lokace bodu), je jedním z nejčastěji řešených problémů výpočetní geometrie. Obvykle je potřeba provést velké množství vyhledávacích operací, a proto jsou kladeny velké nároky na rychlost použitých algoritmů. Dalšími důležitými aspekty výběru vhodného algoritmu jsou i odolnost algoritmu vůči změnám v trojúhelníkové síti, minimální paměťové nároky anebo přijatelná implementační náročnost. Tzv. algoritmy procházky patří mezi nejoblíbenější řešení problému lokace bodu, protože nabízejí odolnost vůči změnám v trojúhelníkové síti při zanedbatelných paměťových požadavcích a obvykle jednoduché implementaci za stále přijatelně nízké očekávané výpočetní složitosti. Proto jsou také často vhodným řešením pro konkrétní aplikace.

Práce představuje soubor sedmi komentovaných odborných článků napsaných autorem práce (spolu se spoluautory) během autorova doktorského studia. Články se zaměřují především na výzkum procházkových algoritmů pro konkrétní aplikace. Bylo vyvinuto několik procházkových algoritmů aplikovatelných na rovinné trojuhelníkové sítě, případně na povrchové trojúhelníkové modely 3D objektů. Nově navržené algoritmy mají uplatnění v řadě oblastí, jako například v počítačové grafice, geografických informačních systémech, haptice, virtuální realitě atd. Dva z prezentovaných článků byly publikovány v impaktovaných časopisech, jeden článek je v recenzním řízení impaktovaného časopisu a čtyři další články byly otisknuty ve sbornících mezinárodních konferencí. I proto představuje důležitou součást této práce příloha, která nabízí jednotlivé články v jejich otisknuté podobě.

Abstract

Finding which triangle in a triangle mesh contains a query point (so-called point location problem) is one of the most frequent tasks in computational geometry. Usually, a large number of point locations has to be performed, and so there is a need for fast algorithms. Moreover, the resistance to changes in the triangle mesh is frequently required as well as minimal additional memory demands or acceptable implementation effort. The so-called walking algorithms rank among the most popular solutions for point location problem since they are offering low complexity, resistance to the changes in the mesh, an easy implementation, and negligible additional memory requirements, which makes them often suitable for particular applications.

The thesis provides a survey for the collection of seven commented research papers which were written by the author of this thesis with co-authors during the author's doctoral study. The papers focus on the research into walking location algorithms during which several walking algorithms offering a number of contributions were developed. Their applications cover variety of areas (e.g., computer graphics, geographic information systems, haptics and virtual reality, etc.) in two different domains: planar triangle meshes and triangulated meshes of 3D model objects. Two of the presented research papers were published in the JSR international journals, one paper has been submitted to journal publication and four other papers were published in proceedings of international conferences. Therefore, substantial part forming the thesis is an appendix where the articles are attached.

"All truly great thoughts are conceived by walking."

- Friedrich Nietzsche

Acknowledgments

First of all I would like to express my gratitude to my collaborators who contributed to the work presented in the thesis. In particular I want to thank to Martina Málková, Václav Purchart and Tomáš Vomáčka. Great thanks belong to my collaborator and also supervisor Prof. Ivana Kolingerová not only for the significant scientific contribution but also for her pleasant human approach, plenty of time and patience. I would like to thank also to my colleagues Oldřich Petřík, Libor Váša, Jan Rus and others for their numerous practical advises, hours of discussion and plenty of inspiration. I also want to express my thanks to people from the Department of Computer Science and Engineering from the Faculty of Applied Sciences, University of West Bohemia in Pilsen, for great support and pleasant working environment. Last but certainly not least, I would like to thank to all beloved people around me who supported me during the doctoral study.

Contents

Chapter 1

Introduction

The point location problem is a very frequent task of computational geometry. There are plenty of applications in a variety of areas, including computer graphics, virtual reality simulations, geographic information systems, computer aided design, haptics, etc. In general, points are searched in meshes consisting of arbitrary (also non-convex) polygons, however, in this thesis we focus on point locations in the following two types of meshes: planar triangle meshes and surface triangle meshes of 3D model objects.

The definition for point location problem in planar triangle meshes is straightforward. Given a query point $\mathbf{q}$ and a planar triangle mesh T, the task is to find such a triangle of the mesh T in which $\mathbf{q}$ geometrically lies. For surface triangle meshes, the problem definition is usually further specified using additional conditions, since the query point rarely lies exactly on the surface of one of the triangles. Therefore the problem definition varies depending on the particular application.

In many applications, a large number of point locations has to be performed, and so there is a need for fast algorithms. The algorithms solving point location problems can be divided into two groups: algorithms using preprocessing (usually for constructing an additional data structures) and algorithms working without preprocessing. The former algorithms concentrate on achieving the lowest expected computational complexity possible ($O(1)$ for suitable data but generally $O(\log n)$ per point query, where n is a number of vertices in the mesh) which is achieved by using sophisticated data structures. The latter group includes so called walking algorithms. Despite their low computational complexity, the former algorithms have some disadvantages. Optimal complexity is achieved at the cost of required data preprocessing, additional memory demands and more complicated maintenance, especially in the case when the triangle mesh is frequently changed.

The name of walking algorithms has arisen from their operating principle. They use triangle neighborhood relations to navigate through the triangle mesh between an initial triangle and the triangle which contains the query point. The initial triangle for such a walk may be arbitrary, however, its clever selection may radically shorten the length of the walk. Walking algorithms do not need

any additional data structures and usually do not need any data preprocessing. Only neighborhood relations are required, however, they are usually utilized in applications for other purposes. Walking algorithms also offer easy implementation and still acceptable sublinear expected time complexity. Therefore, they are often more favored possibility than the optimal time complexity solutions.

This chapter introduces the problematics of walking algorithms: provides basic geometrical background which is used in the walking algorithms and shows necessary prerequisites for the mesh (Section 1.1), explains a selection of the initial triangle (Section 1.2) and appoints a classification of walking algorithms (Section 1.3). Other chapters are organized as follows. Chapter 2 presents an overview of contributions of the papers attached with the thesis. Chapter 3 provides a summary of the research and presents a future work and Chapter 4 contains the list of authors publications, talks and related projects. Finally, the reprints of the research papers, forming the most important part of the thesis, are provided in appendices.

1.1 Background Mathematics

To determine the position of a point $\mathbf{v}$ with respect to an oriented edge (or oriented line) $\overrightarrow{\mathbf{tu}}$, the sign of the determinant in a so-called 2D orientation test [2] is used:

$$orientation2D(\mathbf{t},\mathbf{u},\mathbf{v}) = \begin{vmatrix} u_x - t_x & v_x - t_x \\ u_y - t_y & v_y - t_y \end{vmatrix} \tag{1.1}$$

where a positive value is returned for $\mathbf{v}$ on the left of $\overrightarrow{\mathbf{tu}}$ and a negative for $\mathbf{v}$ on the right of $\overrightarrow{\mathbf{tu}}$.

Figure 1.1 shows the resulting signs of orientation tests from Equation 1.1 for a triangle $\tau_{\mathbf{t_0t_1t_2}}$ with CCW (counterclockwise) order of vertices.

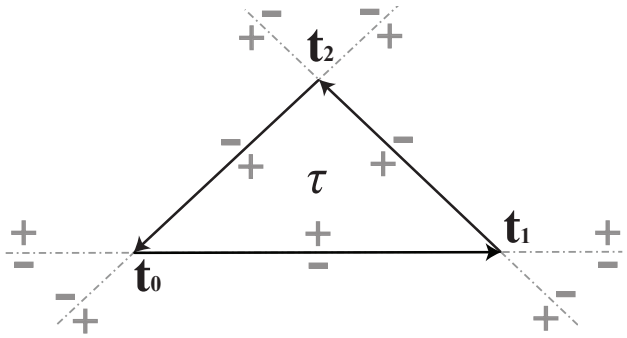

Figure 1.1: Example of orientation tests for triangle edges

Sometimes, it is more suitable to use the implicit line equation of the oriented line instead of the equation above. To determine the position of a point $\mathbf{v}$ with respect to an oriented line $\lambda = \overrightarrow{\mathbf{tu}}$, we compute its implicit equation -

Equations 1.2, 1.3. The position of $\mathbf{v}$ is given by the sign of Equation 1.4 as in the orientation test.

$$\lambda : a \ \cdot x + b \cdot y + c = 0 \tag{1.2}$$

$$(a, b, c) = (t_x, t_y, 1) \times (u_x, u_y, 1) \tag{1.3}$$

$$position(\lambda, \mathbf{v}) = a \cdot v_x + b \cdot v_y + c \tag{1.4}$$

A keystone for the walking-based approach in the third dimension is the so-called 3D orientation test which determines position of a point against a plane: let us have a plane given by three points $\mathbf{t}, \mathbf{u}, \mathbf{v}$ and a tested point $\mathbf{w}$. Equation 1.5 computes whether $\mathbf{w}$ lies above, on or below the given plane when seen from the side where $\mathbf{t}, \mathbf{u}, \mathbf{v}$ points are CCW oriented. In other words, the test decides whether the orientation of these points is positive, neutral or negative.

$$orientation3D(\mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w}) = \begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix} \tag{1.5}$$

Triangle mesh has to fulfill some necessary prerequisites to make the utilization of walking algorithms possible. First, the neighborhood relations are required (information about neighbors is stored for each triangle). Second, the mesh must not contain errors. Third, all the triangle vertices are usually requested to be ordered in a uniform orientation. And fourth, the planar mesh should be usually convex-shaped and without holes, however, the thesis also provides a solution for non-convex shaped planar triangle meshes with holes. The convexity is not required for the surface triangle meshes, but all the triangles have to hold connectivity for all their three neighbors.

1.2 Selection of the initial triangle

Process of the point location by walking algorithms usually works in two steps. The first step is a selection of the initial triangle for the walk and the second step is a using of the neighborhood relationships between the triangles (walking) to find the target triangle, containing the query point.

A proper selection of the initial triangle is very important and may radically shorten the walk and thus speed up the whole location process. Therefore, walking algorithms can reach the lowest expected computational complexity possible $O(\log n)$ (or even $O(1)$ for appropriate data) if sophisticated data structures (such as grids, trees, hash tables, hierarchical structures) are used for the choice of the initial triangle. Walking algorithms are then used only for the final location which is usually short. However, as outlined above, the use of additional data structures provides several disadvantages and therefore it is not suitable for a variety of applications. The most important disadvantage is the need of maintaining the structures valid - especially if the triangle mesh is frequently

changed. Sometimes additional memory requirements may also be a problem similarly as an inefficient parallelization due to the bottleneck problem of the data structure.

Thus the initial triangle is frequently chosen in a clever way without additional structures. For example, the initial triangle is taken as the closest triangle from a set A of randomly chosen triangles from the triangle mesh T, where $||A|| \ll ||T||$. Note that the triangles from A are generated on the fly and are not stored except the closest one. Sometimes some additional information about the data is known and can be used in the selection of the initial triangle for the speed-up of the process. Such information can be the knowledge of the range of the mesh vertices coordinates, in which case all the locations start in the triangle containing the point lying in the middle of the range. Or, in some applications (e.g., the construction of Delaunay triangulation (DT) by incremental insertion), the located points are known at the beginning; thus, the points can be sorted in such an order that the next located query point will be close to the last one. Then, the use of the target triangle from the last location as the initial one for the next location provides a significant speed-up. Sometimes, the located points are also ordered properly without additional sorting.

1.3 Classification of walking algorithms

Given an initial triangle, the walk may proceed. There exist several algorithms solving this step, and according to the style of determining the walking way, they can be classified into three groups: visibility, straight, and orthogonal walks. Definitions of groups are various with respect to the dimension in which the query points are searched and for the planar triangle meshes it is simple and can be described as follows:

Visibility algorithms perform local tests (usually orientation tests) in each triangle they walk through. These tests look for such an edge that defines a line separating the query point from the third vertex of the triangle. The walk then moves across this edge to the neighborhood triangle. For convex-shaped triangle meshes they never cross the border of the mesh. However, deterministic versions of visibility walk algorithms may loop for non-Delaunay triangulations. Unlike deterministic versions, randomized (stochastic) versions of visibility walk algorithms do not loop but they are slower because a randomization step is done in each triangle.

Straight walk algorithms use not only local comparisons to determine the way of the walk but also use a line connecting one point of the starting triangle with the query point and traverse triangles crossed by this line. This way, their walk is short and does not loop. For convex-shaped triangle meshes, they never cross the border of the mesh and they do not loop.

Orthogonal walks first navigate along one coordinate axis and then along the other, which makes the local tests cheaper, since only coordinate components can be compared during the walk. However, more triangles are visited during

the walk. The border of a triangle mesh may be crossed during the walk, in which case a special modification is needed, resulting in a slower location process and additional implementation effort.

The thesis presents the following research regarding planar point location algorithms. Appendix A shows a straight walk algorithm which uses a combination of faster tests and of one visibility walk algorithm and provides a speed-up of the original straight walk principle. Appendix B describes an algorithm which combines directness of the straight walk algorithms and cheap test of the orthogonal walk algorithms. Appendix B uses the visibility walk principle, too. A new visibility walk algorithm is presented in Appendix C. This algorithm is useful mainly for Delaunay triangulation especially where the appropriate hierarchical structure is used. A special modification of the straight walk algorithm which can search in non-convex triangle meshes and, moreover, which can determine whether the point lies outside the mesh, is given in Appendix D. Finally, Appendix E describes the best known representatives from all three groups of walking algorithms for the planar triangle meshes, compares them and provides advice for the choice of the proper algorithm for a particular application.

Problem definition of the point location on the surface triangle meshes is not so straightforward as for the planar triangle meshes and usually is applied for a specific particular application. Appendix F could be an example. It describes an effective walking algorithm for point location on the surface of a general star-shaped polyhedron. Also, a point location algorithm can be used as a way to solve other problems of computational geometry. Appendix G could serve as an example where a surface walking algorithm is successfully used as a tool for solving of the collision detection problem.

Chapter 2

Overview of contributions

This chapter summarizes main contribution of the work collected in this thesis. The overview is structured into two main sections addressing contribution in the areas of planar walking algorithms and surface walking algorithms.

2.1 Planar walking algorithms

The thesis presents four planar walking algorithms while three of them bring significant speed-up against previous algorithms (up to tens of percents - Appendix A, B, C) and one of these algorithms shows possibility to use a walking algorithm also for non-convex meshes with holes (Appendix D). The thesis also contains a survey of planar walking algorithms for the use in TINs (triangulated irregular networks - Appendix E) but most of its conclusions are applicable in general, not only for TIN meshes.

2.1.1 Straight walk algorithm modification for point location in a triangulation

Appendix A presents an algorithm improving the original straight walk algorithm [3]. The proposed algorithm is based on the standard straight walk principles and copies the path but 2D orientation tests (see Equation 1.1) are substituted by the implicit line equation tests (Equation 1.4) which are cheaper. Implicit line equations (Equations 1.2, 1.3) are pre-processed in the initialization phase of each location query. It results in faster location process. The proposed algorithm allows substitution by cheaper tests at the cost that target triangle may not be found exactly. Therefore, the stochastic visibility walk (which is using 2D orientation tests) is used for the final (and usually short) determination of the target triangle.

2.1.2 Hybrid walking point location algorithm

Appendix B describes an interesting possibility how to carry out the location query with path similar to the straight walk algorithm while using only the comparison of each coordinate value instead of some orientation tests (likewise in orthogonal walk algorithms). The main idea of the 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 (a scale-loss rotation) is then used for the tested points throughout the walk to enable a cheap comparison of their position with respect to this line. Despite the use of transformation, the walk is still rather fast since only the query point and the tested points (one per visited triangle) are transformed. However, the improvement is rather minor and orthogonal walks still usually achieve better results.

2.1.3 A new visibility walk algorithm for point location in planar triangulation

Appendix C provides a simple visibility walk algorithm which is able to search for the query point with only one 2D orientation test for each visited triangle (except the first triangle where two orientation tests are needed). The algorithm utilizes properties of the barycentric coordinates. It remembers the result of 2D orientation test from the last visited triangle and computes a 2D orientation test for another triangle edge. Then the result of 2D orientation test for the last third edge can be computed without the test itself. It is achieved by the use of additional memory, where for each triangle vertices the value of 2D orientation test is stored.

The store of the auxiliary values makes the algorithm inappropriate for applications where the triangle mesh is often changed or where there is a lack of memory. On the other hand, the stored values can be useful for other purposes since they inform about triangles area (the needed value is a double of the triangle area). For longer location paths, the algorithm is slower than orthogonal walk algorithms. But it is highly relevant for shorter walks since its initialization step is very cheap and there is no need of final locations. Therefore, the use is expected mainly in applications with searching in popular hierarchical triangle meshes [5, 1] since the location is usually very short in the individual layers. Note that the algorithm is not appropriate for non-Delaunay triangulations since it can theoretically cause an infinitive loop [7].

2.1.4 Walking algorithm for point location in a triangulated non-convex domain with holes

The disadvantage of the walking algorithms may be a fact that they could handle only triangulated convex domains without holes and sometimes it is not

possible or suitable to triangulate the remaining empty space. Therefore, the algorithm presented in Appendix D can search in both, in a triangulated non-convex domains and in triangulated domains with holes. The algorithm uses the straight walk principle with 2D orientation tests. If it reaches the border of the triangle mesh, it walks along the border until it finds a triangle proper for continuation of the walk (the triangle intersected by the line defining the straight walk). If such a triangle does not exist, the walk returns the result that the query point is outside the mesh. Thus, the algorithm is also able to detect if the query point lies inside or outside of the triangulated polygon. The algorithm can be used also for point-inside-polygon testing, although we do not expect using of the algorithm for such queries.

2.1.5 Walking algorithms for point location in TIN models

It is obvious that several planar walking algorithms were published, each offers different advantages, and the appropriate algorithm should be chosen according to the particular application. However, these algorithms have never been summarized, tested and compared to provide a tool for such a selection. This is provided by Appendix E, with focus on the use in geosciences where walking algorithms are very useful especially for point location in TIN models. Note that planar walking algorithms are easily applicable for 2.5D terrain models where the height information is omitted for the location purposes.

The article describes all substantial walking algorithms in detail to provide all information needed for their implementation, discuss their behavior and test them on different datasets including random data, data from a cadastre and LIDAR data. All the algorithms are tested in two forms: with double precision floating point arithmetic and also with adaptive floating point arithmetic [6]. The article also presents a simple comparison of the speed, implementation effort and stability of the algorithm, and providing a tool for decision which algorithm is suitable to implement in a particular solution. The scope of the article is not limited only to geosciences or to TIN models. The most of conclusions has a general validity and can be used in a variety of areas and applications.

2.2 Surface walking algorithms

The thesis presents two surface walking algorithms while the former of them partly uses a transformation to 2D and then it is searching on the surface of a star-shaped tetrahedron and the latter is an algorithm which works exclusively in the third dimension and applies a walking location algorithm to the collision detection problem.

2.2.1 Star-shaped polyhedron point location with orthogonal walk algorithm

The star-shaped polyhedron point location problem, solved in Appendix F, is defined as follows. For the given oriented half-line segment, defined by the center point of the star-shaped polyhedron (generally an arbitrary point which lies in the kernel of the polyhedron) and a query point, the goal is to find a triangle from the polyhedron surface which is intersected by that half-line segment. The main use of the algorithm is expected especially for spherical point location queries which are important mainly for spherical remeshing methods.

The article presents a combination of two walking algorithms for point location on the surface of the star-shaped polyhedron. The former is a modification of the planar orthogonal walk algorithm and works in the simplified spherical coordinate system. The main idea of this algorithm is based on a planar orthogonal walk algorithm which locates points in planar triangle meshes. But unlike the original orthogonal walk presented in [2] it uses only a comparison of each coordinate value which is significantly faster and no orientation tests are needed. Moreover, it uses only two tests per triangle (instead of three). Naturally, as with the proposed straight walk algorithm (Section 2.1.1), the target triangle may not be found exactly and thus the stochastic visibility walk is used for the final (and also usually short) determination of the target triangle.

The latter algorithm is a spatial generalization of the stochastic visibility walk algorithm and uses the 3D orientation test. Connection of these two algorithms combined with the proposed solution for an effective choice of the initial triangle using a non-uniform grid makes the algorithm very efficient.

2.2.2 Surface point location by walking algorithm for haptic visualization of triangulated 3D models

Appendix G focuses on the collision detection problem of a haptic device with the surface of 3D model which is defined by a triangle mesh. The goal is to find a triangle (if such a triangle exists) which is in the collision trajectory of the haptic probe to provide an appropriate feedback to the user. If such a triangle does not exist, no feedback is provided.

The algorithm is based on the straight walk principle. However, unlike the planar straight walk, where the walk follows the triangles intersected by the line defined by the query point and a point inside the initial triangle, for the surface straight walk algorithm, the path is defined by the plane generated by the query point, by a point inside the initial triangle and by the motion vector of the haptic probe. Since meshes can be generally non-convex-shaped, the algorithm may not find the final triangle although it exists (it depends on a selection of the initial triangle because it is the only part generating the plane which is variable). Therefore, the problem of the initial triangle choice is one of the key assumptions and it is solved by a spatial generalization of the algorithm

which was originally proposed by [4] and which chooses the initial triangle as the closest triangle from a randomly chosen set of triangles from the triangle mesh. Moreover, it is necessary to repeat the whole location process if the result is negative to ensure the validity of such a result.

Although the algorithm was developed for the haptic visualization, it is not limited to the haptic collision detection only. It can be used for all point location problems, where the input contains both, a point close to the surface of a triangulated 3D model and a vector directing towards the model. For example, for a parametric description of the model, we can get a point on the surface as well as a vector directing towards the model (it may be the opposite surface normal at this point). Results show that the proposed algorithm can handle queries on rather complex-shaped models with hundreds of thousands of triangles in a good time and thus it can be successfully used in haptic visualization. The algorithm is suitable also for models changing in time. Although it is not a primary task of the algorithm, it can also handle queries when the model is composed of multiple components. Moreover, the algorithm is easily and effectively parallelizable which can significantly speed up the search process.

Chapter 3

Summary and Future work

This thesis presents issues of the point location problem and aims mainly at the so-called walking algorithms. The most important part of the thesis is the collection of author's papers. The thesis addresses a variety of different aspects of the walking algorithms: their advantages and disadvantages, reliability and performance, suitability for particular applications, etc. As shown in the thesis, walking algorithms provide solutions with acceptable computational as well as implementation complexity, low memory requirements and flexibility in the case of the mesh modification. Therefore, they are very popular, especially for engineering applications where the flexible changes of the mesh are requested.

The thesis presents several new walking algorithms where five of them are for the planar point location and two of them search in the surface triangle meshes. Some of them present a novel approach which has not been used yet for similar problem solutions (see Appendix D, Appendix F, Appendix G), some provide improvements of the original existing algorithm with a significant speed-up against previous versions (up to tens of percents - see Appendix A, Appendix B, Appendix F) or are more effective for some particular solutions (see Appendix C). And finally, the most significant planar walking algorithms are summarized, tested and compared to provide a tool for a proper choice of the relevant walking algorithm for a particular application (with the focus on geosciences but the validity is more general - see Appendix E).

Still, there is a lot of space for future work, mainly in particular applications. For the planar triangle meshes, most of the development of the walking algorithms is expected for unusual problems or for specific data where a specified solution will be necessary. For the surface triangle meshes there is a space for development in particular applications where the mesh is changing in time, e.g., the haptic visualization. A specific solution is then dependent on the particular problem definition. The area of tetrahedral meshes is also very perspective, especially if the mesh is changing in time, as is in the area of the dynamic protein research where some algorithms have been tested but yet without a relevant publication.

Chapter 4

Activities

4.1 List of authors publications

- Soukal, R., Kolingerová, I.
  Straight walk algorithm modification for point location in a triangulation.
  Proceedings of the 25th European Workshop on Computational Geometry, pp. 219–222, (2009)

- Soukal, R., Holub, P.
  A note on packing chromatic number of the square lattics.
  Electronic Journal of Combinatorics, Volume 17, Issue 1, pp. 1–8, (2010), ISSN 1077–8926, IF 0.568 (2013)

- Soukal, R., Kolingerová, I.
  Star-shaped polyhedron point location with orthogonal walk algorithm.
  Procedia Computer Science, Volume 1, Issue 1, pp. 219-–228, Elsevier (2010), ISSN 1877–0509

- Soukal, R., Málková, M., Vomáčka, T., Kolingerová, I.
  Hybrid walking point location algorithm.
  Proceedings of the 5th International Conference on Advanced Engineering Computing and Applications in Sciences, pp. 7–12, IARIA XPS Press (2011), ISSN 2308–4499, ISBN 978–1–61208–172–4

- Soukal, R., Málková, M., Kolingerová, I.
  Walking algorithms for point location in TIN models.
  Computational Geosciences, Volume 16, Issue 4, pp. 853–869, Springer Verlag (2012), ISSN 1420–0597, IF 1.612 (2013)

- Soukal, R., Málková, M., Kolingerová, I.
  A new visibility walk algorithm for point location in planar triangulation.
  Lecture Notes in Computer Science (Advances in Visual Computing), Volume 7432, pp. 736–745, Springer Verlag (2012), ISBN 978–3–642–33190–9

- Soukal, R., Purchart, V., Kolingerová, I.
  Surface point location by walking algorithm for haptic visualization of triangulated 3D models.
  Advances in Engineering Software, Volume 75, pp. 58–67, Elsevier (2014), ISSN 0965–9978, IF 1.422 (2013)

4.2 Articles in the review process

- Soukal, R., Kolingerová, I.
  Walking algorithm for point location in a triangulated non-convex domain with holes.
  Submitted to (currently in the first revision): International Journal of Geographical Information Science, Taylor & Francis, IF 1.479 (2013)

4.3 Talks

- Walking algorithms for point location.
  University of Maribor, 2008
  Maribor, Slovenia

- Straight walk algorithm modification for point location in a triangulation.
  25th European Workshop on Computational Geometry, 2009
  Brussels, Belgium

- Star-shaped polyhedron point location with orthogonal walk algorithm.
  10th International Conference on Computational Science, 2010
  Amsterdam, Netherlands

- Hybrid walking point location algorithm.
  5th International Conference on Advanced Engineering Computing and Applications in Sciences, 2011
  Lisbon, Portugal

- A new visibility walk algorithm for point location in planar triangulation.
  8th International Symposium on Visual Computing, 2012
  Rethymnon, Greece

4.4 Participation on projects

- Triangulated Models for Haptic and Virtual Reality.
  Project 201/09/0097, Czech Science Foundation (GACR)
  2009–2011

- Advanced Computing and Information Systems.
  Project SGS–2010–028, UWB grant
  2010–2012

- Analysis and Visualization of Protein Structures.
  Project 202/10/1435, Czech Science Foundation (GACR)
  2011–2012

- INGEM – Interactive Gemetric Models for Simulation of Natural Phenomena and Crowds.
  Project Kontakt no. LH11006, Ministry of Education, Youth and Sports of the Czech Republic
  2013–2014

- Advanced Computing and Information Systems.
  Project SGS–2013–029, UWB grant
  2013–2014

Bibliography

[1] Devillers, O. The Delaunay hierarchy. *International Journal of Foundations of Computer Science 13* (2002), 163–180.

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

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

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

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

[6] Shewchuk, Jonathan, R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry 18*, 3 (1997), 305–363.

[7] Weller, F. On the total correctness of Lawson's oriented walk. In *Proceedings of the 10th International Canadian Conference on Computational Geometry* (1998), pp. 10–12.

Appendices - Paper Reprints

Appendix A

Straight walk algorithm modification for point location in a triangulation

Soukal, R., Kolingerová, I.

Proceedings of the 25th European Workshop on Computational Geometry, pp 219–222, (2009)

Straight Walk Algorithm Modification for Point Location in a Triangulation *

Roman Soukal † Ivana Kolingerová ‡

Abstract

Finding an element in a mesh which contains a query point is a very frequent task in computational geometry. This paper shows a modification of one planar technique for point location in triangulation. This algorithm is based on a modification of the straight walk location algorithm combined with the visibility walk algorithm where straight walk brings speedup and visibility walk reliably finds a target. This modification does not improve the worst-case running time although empirical data shows that its performance is better than previous *Straight walk* algorithm.

1 Introduction

For a query point q and a given triangulation T of n vertices in the plane the point location problem usually means how to find a triangle ω from T which contains the query point. One of the possibilities how to solve this problem is to use one of the walking location techniques. The name of these algorithms has arisen from the way of locating the triangle ω which contains q. For a starting triangle α chosen as one of the triangles of T and the query point q the walking strategy makes use of connectivity in the triangle mesh and it goes through triangles between α and ω. There are three main types of walking strategies. First, the visibility walk makes use of an orientation edge test to determinate which triangle is the next. Second, the straight walk passes all triangles in the mesh between α and ω which are intersected by a line pq where p is a point inside α. Finally, the orthogonal walk passes all triangles in the mesh between α and ω in the directions of coordinate axes. The triangle α may be chosen randomly or as the closest triangle to q from the set A of randomly chosen triangles from T, $\|A\| \ll \|T\|$ [7]. This choice improves speed of the algorithm. Walking algorithms are frequently used despite suboptimal complexity ($O(\sqrt[3]{n})$ up to $O(\sqrt{n})$ [3] - depending on how the α was chosen).

Sophisticated data structures such as DAG [1], [6], skip list [11], quad tree, buckets [10], uniform grid [9], [12] and data structures based on random sampling [8], [2] are used in the point location algorithms with the best known complexity $O(\log n)$ per point query. However, these algorithms have some disadvantages. First, these data structures consume generally $O(n)$ amount of memory which may be a problem for huge datasets. Second, implementation effort for most of these structures may be nontrivial (especially for modifications of these structures). Finally, most of these structures are hierarchical and the top level of the hierarchy may be a bottleneck in case of parallelization. Walking algorithms do not need any extra memory, their implementation is rather simple and their usability for parallelization is good, thus often they are a better choice than optimal time complexity solutions.

This paper shows a straight walk algorithm modification which is faster than the standard straight walk algorithm [3]. The paper is organized as follows. Section 2 and 3 present already existing algorithms *Remembering Stochastic walk* and *Straight walk*. Section 4 shows our modification of this algorithm, problem resulting from the character of this modification and its solution. Empirical results are presented in Section 5.

2 Remembering Stochastic Walk

Visibility walk algorithms use orientation test for a triple of points t, u, v (Equation 1).

$$orientation(t,u,v) = sgn\left(\begin{vmatrix} u_x - t_x & v_x - t_x \\ u_y - t_y & v_y - t_y \end{vmatrix}\right) \quad (1)$$

Decision, whether or not the edge rl of the current triangle τ ($\tau = srl$) should be crossed to continue the walk into the next triangle depends on the result after substitution q, r, l to the Equation 1 where q is the query point. The walk continues to the next triangle over the edge rl if $orientation(q,r,l) = -orientation(s,r,l)$ where s is vertex of τ, $r, l \neq s$ and q is the query point. Simple visibility walk algorithms use edges of τ for tests in the given order, but these visibility walks may loop for a non-Delaunay triangulation [4]. For non-Delaunay triangulation it is necessary to choose the tested edges of τ in a random order. This modification is called *Stochastic*. Naturally it is not necessary to test the edge incident with

*This work is supported by the Grant Agency of the Czech Republic - the project GA 201/09/0097

†Department of Computer Science, Faculty of Applied Sciences, University of West Bohemia, soukal@kiv.zcu.cz

‡Department of Computer Science, Faculty of Applied Sciences, University of West Bohemia, kolinger@kiv.zcu.cz

the previous triangle in the walk. This improvement is called *Remembering* and may save up to one orientation test for each triangle. Therefore, one or two orientation tests are needed for each triangle (except the triangle α).

3 Straight Walk

Generally, the straight walk passes all triangles in the triangulation T intersected by the line pq where q is the query point and p is a point in the given starting triangle α. The standard straight walk algorithm [3] is composed of two steps and point p is chosen as a vertex of α. In the initialization step (grey color in Figure 1) a triangle γ incident to p and intersected by pq must be found. During this step one orientation test is needed for each visited triangle and the number of visited triangles is at most the degree of p, thus at most n.

After the initialization is completed, the straight walk (black color in Figure 1) may start. For each triangle τ, $\tau = srl$ in the straight walk, the line pq goes into $\tau, \tau \neq \alpha$ through edge $e = rl$ (see τ in Figure 1). Depending on the $orientation(s, p, q)$, new r (or new l) is selected as s. The singular case $orientation(s, p, q) = 0$ is added to one of these situations. Now the straight walk goes out of τ through the new edge $e = rl$. By testing on which side of e the q lies it is decided whether the τ contains the q or whether the walk must go on. In the latter case the walk goes to the neighbor of τ through e. Therefore the *Straight walk* evidently needs two orientation tests per triangle and some orientation tests in the initialization step. Pseudo code of the algorithm is given as Algorithm 1 [3].

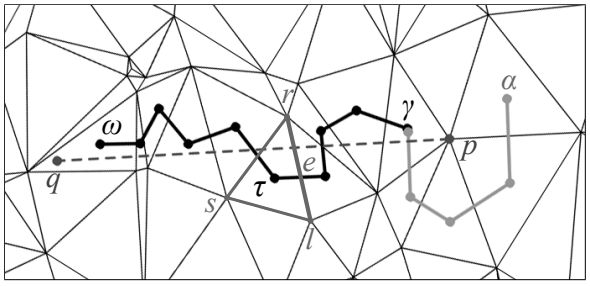

Figure 1: Straight walk example

4 The Proposed Modification of Straight Walk

The first idea is to simplify the initialization step of the algorithm in Section 3 to a constant number of operations. The second idea is to use a cheaper operation than the orientation test used in *Straight walk* algorithm in Section 3.

```
// traverses the triangulation T
Input:
  • the query point q
  • the chosen starting triangle α, α ∈ T
Output:
  • the triangle ω which contains q

// following the line segment from p to q
// initialization step
p = vertex of α;
if orientation(r, p, q) < 0 then
    while orientation(l, p, q) < 0 do
        r = l;
        τ = neighbor of τ trough pl;
        l = vertex of τ, l ≠ p, l ≠ r;
    end
else
    repeat
        l = r;
        τ = neighbor of τ trough pr;
        r = vertex of τ, r ≠ p, r ≠ l;
    until orientation(r, p, q) < 0 ;
end
// end of initialization and start of straight walk
// now pq has r on the right and l on the left side
while orientation(q, r, l) < 0 do
    τ = neighbor of τ trough rl;
    s = vertex of τ, s ≠ r, s ≠ l;
    if orientation(s, p, q) < 0 then
        r = s;
    else
        l = s;
    end
end
// end of Straight walk step
// now τ contains q
return τ;
```

Algorithm 1: Straight Walk

A fundamental prerequisite for the initialization step is to suitably choose the point p. In Section 3, p is chosen as one vertex of the starting triangle α but it is not necessary. The main idea is to choose p in a way that no other operations in the initialization step are needed. First, the point s is chosen as the closest vertex from α to q. The edge $e = rl$ is the edge of α and $\alpha = srl$. Next, p is chosen on e where $r, l \neq p$ and $\|pq\| > 0$. Now the straight walk may start. For each triangle τ in the straight walk (except α) the edge e of τ is edge used to cross to τ and s is the vertex of τ facing e. The line pq goes out of τ through the edge e that is determined by the means of using the orientation test. If s lies on the left side of pq, e $(s \in e)$ is the edge of τ on the right of s, else e $(s \in e)$ is the edge of τ on the left of s. By testing on which side of e the q lies it is decided whether τ contains q or whether the walk must go on.

For each triangle τ the position of s is tested against the line pq and only s is changing during the walk, it is therefore possible to use an implicit line equation test in the place of the orientation test to speed up

the process. The implicit line equation of pq is computed in the initialization step (Equations 2, 3, 4) and the position of s is found by a substitution into Equation 5.

$$\lambda : a \cdot x + b \cdot y + c = 0 \tag{2}$$

$$[a, b, c] = [p_x, p_y, 1] \times [q_x, q_y, 1] \tag{3}$$

$$a = p_y - q_y, b = q_x - p_x, c = p_x \cdot q_y - p_y \cdot q_x \tag{4}$$

$$position(\lambda, s) = sgn(a \cdot s_x + b \cdot s_y + c) \tag{5}$$

The implicit line equation test is subject to higher numerical imprecision than the orientation test but the straight walk algorithm is robust enough to resist it. Now there is one orientation test and one position test per triangle. The orientation test is used to detect whether the triangle ω was found. A direct replacement of this orientation test is problematic because two points are changing during the walk, therefore the original algorithm must be modified. For this modification a new line $\lambda_n, \lambda_n \perp pq, q \in \lambda_n$ must be computed in the initialization step (Equation 6, 7).

$$\lambda_n : a \cdot x + b \cdot y + c = 0 \tag{6}$$

$$a = \lambda_y, b = \lambda_x, c = \lambda_x \cdot q_y - \lambda_y \cdot q_x \tag{7}$$

The orientation test for detection, whether the triangle ω was found, is replaced with the position test of λ_n and s. If s lays on the other side of λ_n than p, the straight walk ends. A situation is possible where $\tau \neq \omega$ at the end of the straight walk (see Figure 2). Because of it, *Remembering Stochastic walk* algorithm is used for final location. As a rule, this final location is very short, but extreme cases exist where almost the whole walk is performed by *Remembering Stochastic walk*. However, this situation is not probable and degradation of *Straight walk* to *Remembering Stochastic walk* is not a problem, because *Remembering Stochastic walk* is not dramatically worse (see Section 5). It is possible to a find more detailed description of this modified variant of *Straight walk* in the pseudo code Algorithm 2 and in the example Figure 2. The straight walk step is coloured black and the final location by *Remembering Stochastic walk* is grey. The triangle γ is the triangle where the straight walk step ends and *Remembering Stochastic walk* starts, the edge e_γ is the edge to γ and the point s_γ is the point s for γ, whose position at the other side of λ_n than p causes the end of the straight walk.

5 Empirical Results

The following algorithms were tested: *Remembering walk*, *Remembering Stochastic walk*, *Straight walk* (Algorithm 1) and *Modified Straight walk* (Algorithm 2). Tests were performed on Delaunay triangulations of different sizes of datasets (especially 100000 and

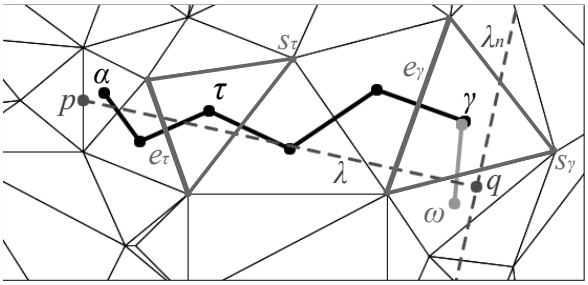

Figure 2: Modified Straight walk example

```
// traverses the triangulation T
Input:
  • the query point q
  • the chosen start triangle α, α ∈ T
Output:
  • the triangle ω which contains q
// initialization step
τ = α;
s = the vertex of τ closest to q;
e = rl = edge of τ facing s;
choose p:  p ∈ e, r, l ≠ p, ||pq|| > 0;
compute the line λ (pq);
compute the line λ_n;
found = false;
// end of initialization and start of straight walk
repeat
    if position(λ_n, s) > 0 then
        found = true;
    else
        if position(λ, s) > 0 then
            e = edge of τ on the left from s, s ∈ e;
        else
            e = edge of τ on the right from s, s ∈ e;
        end
        t = neighbor of τ trough e;
        s = vertex of τ facing e;
    end
until found ;
// end of the straight walk step
// final location
τ = Remembering Stochastic Walk (τ, q);
// now τ contains q
return τ;
```

Algorithm 2: Modified Straight Walk

1000000 points) and on different types of point distributions (randomly in square, grid, gauss distribution, clusters, arcs, real data). First triangle δ was randomly chosen for each location. 1000000 of randomly generated points were located by each algorithm on each dataset. Selected results are in Table 1. The following properties were examined for each algorithm: the average length of the walk (#Δ), the average number of the orientation tests (#ori), the average number of position tests (#pos) and the average time (t[μs])per one location (tested on Intel Q6600 2,40GHz). The algorithms are coded in Java with the double precision floating point arithmetic.

Algorithm	#Δ	#ori	#pos	t[μs]
	ϕ per located point			
Delaunay triangulation of 100.000 points distributed in a square				
Rem. walk	377.80	508.55	0	43.97
Rem. Stoch. walk	360.12	465.11	0	58.04
Straight walk [3]	338.98	677.21	0	57.14
Mod. Straight walk	340.83	3.77	677.23	41.07
Delaunay triangulation of 1.000.000 points distributed in a square				
Rem. walk	1185.93	1590.30	0	214.97
Rem. Stoch. walk	1122.87	1443.80	0	255.57
Straight walk [3]	1069.09	2137.46	0	234.16
Mod. Straight walk	1071.11	3.76	2138.37	188.61
Delaunay triangulation of 100.000 points distributed in a clusters				
Rem. walk	214.25	290.09	0	16.73
Rem. Stoch. walk	215.06	279.95	0	24.33
Straight walk [3]	173.66	346.68	0	20.87
Mod. Straight walk	174.40	5.21	341.42	12.06
Delaunay triangulation of 70.433 points from real data				
Rem. walk	324.00	437.51	0	34.47
Rem. Stoch. walk	324.42	422.14	0	47.76
Straight walk [3]	297.08	593.46	0	46.81
Mod. Straight walk	298.02	5.22	588.59	32.6

Table 1: Comparison of algorithms

With regard to the number of tests, *Remembering Stochastic walk* with about 1.30 orientation tests per triangle is the best. Non-Stochastic *Remembering walk* is a little worse with about 1.35 orientation tests per triangle. Both straight walks have about two tests per triangle (a small difference is due to the initialization step (the standard *Straight walk*) or due to the final location (*Modified Straight walk*)) but *Modified Straight walk* mainly uses faster tests. With regards to the number of visited triangles, the best are the *Straight walk* algorithms. In comparison of time per one location, *Modified Straight walk* is the fastest way because of faster tests. *Remembering walk* has good results too but it is only usable for Delaunay triangulations. *Remembering Stochastic walk* has very good results regarding the number of tests per triangle but the cost for the randomization is too high, therefore the standard *Straight walk* with two orientation tests per triangle is a little faster.

Dataset	ϕ #Δ	max #Δ
	per located point	
DT of 100.000 random pts	1.77	10
DT of 1.000.000 random pts	1.76	9
DT of 100.000 pts in clusters	3.23	341
DT of 70.433 pts from real data	3.23	186

Table 2: Length of final location by Remembering Stochastic walk in Modified Straight walk algorithm

Experiments show that the length of the final location with *Remembering Stochastic walk* in *Modified Straight walk* is generally short (see Table 2). There are fluctuations on some types of datasets (e. g. clusters) but they are rare and may be ignored on average.

6 Conclusion

The proposed algorithm was verified to find a point in a Delaunay triangulations in a better time than the popular *Remembering Stochastic walk*. Our algorithm can be used also for non-Delaunay triangulations. Testing on such data is our future work.

Acknowledgments

We would like to thank Mr. Brož and Mr. Hlaváček for their feedback and inspiring discusion.

References

[1] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. *Computational Geometry, Algorithms and Applications.* Berlin Heidelberg: Springer, 1997.

[2] O. Devillers. *Improved Incremental Randomized Delaunay Triangulation* Proceedings of the 14th Annual Symposium on Computational Geometry 1998, 106-115, 2001.

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

[4] L. De Floriani, B. Falcidieno, G. Nagy, C. Pienovi. *On Sorting Triangles in a Delaunay Tesselation.* Algorithmica 6, 522-532, 1991.

[5] I. Kolingerová. *A Small Improvement in the Walking Algorithm for Point Location in a Triangulation.* 22nd European Workshop on Computational Geometry, 221-224, 2006.

[6] I. Kolingerová, B. Žalik. *Improvements to Randomized Incremental Delaunay Insertion.* Computers & Graphics, 26, 477-490, 2002.

[7] E. P. Mücke, I. Saias, B. Zhu. *Fast Randomized Point Location without Preprocessing in Two- and Three-dimensional Delaunay Triangulations.* Proceedings of the 12th Annual Symposium on Computational Geometry 1996, 274-283, 1996.

[8] K. Mulmuley. *Randomized Multidimensional Search Trees: Dynamic Sampling.* Proceedings of the 7th Annual Symposium on Computational Geometry 1991, 121-131, 1991.

[9] S. W. Sloan. *A Fast Algorithm for Constructing Delaunay Triangulations in the Plane.* Advanced Engineering Software, 9(1):34-55, 1987.

[10] P. Su, R. L. S. Drysdale. *A Comparison of Sequential Delaunay Triangulation Algorithms.* Proceedings of the 11th Annual Symposium on Computational Geometry 1995, 61-70, 1995.

[11] M. Zadravec, B. Žalik. *An Almost Distribution Independent Incremental Delaunay Triangulation Algorithm.* The Visual Computer, 21(6):384-396, 2005.

[12] B. Žalik, 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.

Appendix B

Hybrid walking point location algorithm

Soukal, R., Málková, M., Vomáčka, T., Kolingerová, I.

Proceedings of the 5th International Conference on Advanced Engineering Computing and Applications in Sciences, pp. 7–12, IARIA XPS Press (2011), ISSN 2308–4499, ISBN 978–1–61208–172–4

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.

 ISBN: 978-1-61208-172-4

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:

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

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

 ISBN: 978-1-61208-172-4

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

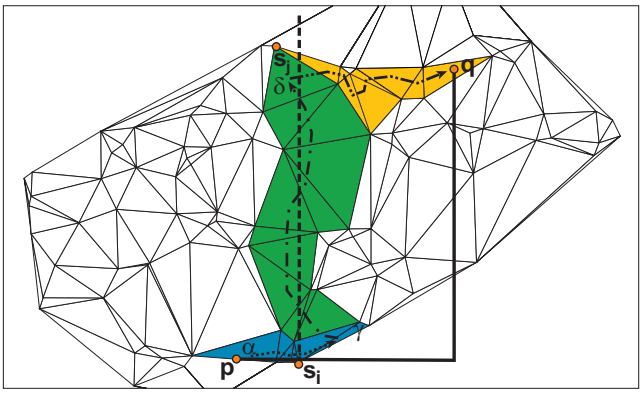

Figure 1. A case when the orthogonal walk crosses the border of the triangular mesh (a dotted line denotes the horizontal walk, a chain line denotes the vertical walk, a chain line with double dots denotes the final location by the RSW algorithm; the dashed and solid line denote the lines controlling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

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

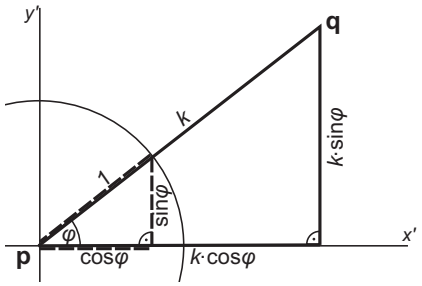

Figure 2. Components of the transformation matrix (x' is parallel with the x-axis, y' with the y-axis).

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 $\overrightarrow{\mathbf{p'q'}}$, 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 ω which contains $\mathbf{q}$

```
// initialization step
triangle τ = α = lrs;
point p = s;
if p = q then return τ;
vector a = q − p;
// k = ||a||
double kcos = a_x;
double ksin = a_y;
q'_x = q_x · kcos − q_y · ksin;
q'_y = q_x · ksin + q_y · kcos;
double r'_y = r_x · ksin + r_y · kcos;
if r'_y > q'_y then
    // r is above pq→
    double l'_y = l_x · ksin + l_y · kcos;
    while l'_y > q'_y do
        τ = neighbor of τ trough ε_pl;
        r = l;
        l = vertex of τ, where l ≠ p, l ≠ r;
        l'_y = l_x · ksin + l_y · kcos;
    end
    s = l; l = r; r = p;
else
    // r is below pq→
    repeat
        τ = neighbor of τ trough ε_pr;
        l = r;
        r = vertex of τ, where r ≠ p, r ≠ l;
        r'_y = r_x · ksin + r_y · kcos;
    until r'_y > q'_y;
    s = r; r = l; l = p;
end
// now pq→ intersects τ
// walk step - following the line segment pq→
s'_x = s_x · kcos − s_y · ksin;
while s'_x < q'_x do
    s'_y = s_x · ksin + s_y · kcos;
    if s'_y < q'_y then r = s; else l = s;
    τ = neighbor of τ trough ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
    s'_x = s_x · kcos − s_y · ksin;
end
return remembering_stochastic_walk(q, τ);
```

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),

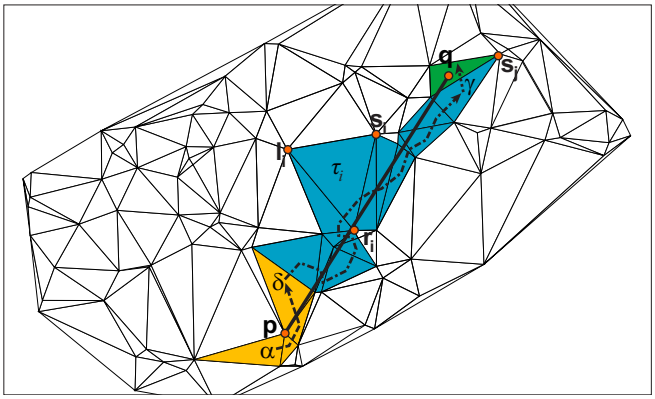

Figure 3. A hybrid walk example (a dashed line denotes the initialization step, a chain line denotes the walk and the final location by RSW algorithm is marked by a dotted line).

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.

 ISBN: 978-1-61208-172-4

Appendix C

A new visibility walk algorithm for point location in planar triangulation

Soukal, R., Málková, M., Kolingerová, I.

Lecture Notes in Computer Science (Advances in Visual Computing), Volume 7432, pp. 736–745, Springer Verlag (2012), ISBN 978–3–642–33190–9

A New Visibility Walk Algorithm for Point Location in Planar Triangulation

Roman Soukal, Martina Malková, and Ivana Kolingerová

University of West Bohemia, Plzeň, Czech Republic

Abstract. Finding which triangle in a planar triangle mesh contains a query point is one of the most frequent tasks in computational geometry. Usually, a large number of point locations has to be performed, and so there is a need for fast algorithms resistant to changes in triangulation and having minimal additional memory requirements. The so-called walking algorithms offer low complexity, easy implementation and negligible additional memory requirements, which makes them suitable for such applications. In this paper, we propose a walking algorithm which significantly improves the current barycentric approach and propose how to effectively combine this algorithm with a suitable hierarchical structure in order to improve its computational complexity. The hierarchical data structure used in our solution is easy to implement and requires low additional memory while providing a significant acceleration thanks to the logarithmic computational complexity of the search process.

1 Introduction

The point location problem is often solved in computational geometry tasks, such as triangulation construction and deformation, morphing and terrain editing. In this text, we focus on point location algorithms for triangle meshes, which are the most common geometry representation.These algorithms can also be used for terrain models represented by triangle meshes without any preprocessing, only by omitting the height information during the location. The input mesh is expected to be convex and without holes, other types of data should be triangulated first.

The point location problem is defined as follows. For a given planar triangle 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 computational complexity possible, in this case $O(\log n)$ per query point (n is the number of vertices in the mesh). However, these algorithms have additional memory demands, they are more difficult to implement, and their modification to cover adding or removing vertices is often problematic. The latter group tries to avoid these disadvantages, but has a slightly higher, but still sublinear complexity. The most important representatives of this group are walking algorithms.

Walking algorithms use the triangle neighborhood relations to go (*walk*) via the triangles between the starting triangle and the one containing the query

G. Bebis et al. (Eds.): ISVC 2012, Part II, LNCS 7432, pp. 736–745, 2012.

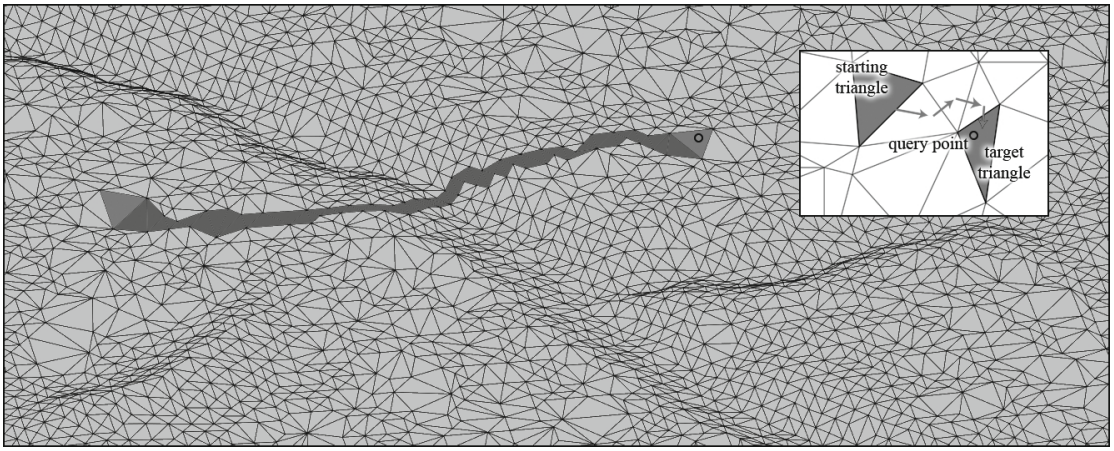

Fig. 1. Example of the point location with a walking algorithm on the part of a terrain model

point (see Figure 1). The starting triangle may be arbitrary, however, its clever selection may radically shorten the length of the walk.

The so-called barycentric walk [1] is an easily implementable algorithm which is independent of the triangles orientation. Although, in its original form, it is also one of the slowest walking algorithms, but has a high potential for speed-up, because its walk is the shortest among the existing walking algorithms. Our approach significantly improves the speed of this algorithm, making it more interesting for further use in complex algorithms and applications. We also propose its combination with a suitable additional data structure in order to improve its computational complexity. The hierarchical data structure used in our solution is easy to implement and requires low additional memory, but it provides a significant acceleration thanks to the logarithmic computational complexity of the search process.

The rest of the paper is organised as follows. Section 2 presents the existing walking algorithms and several approaches for a sophisticated selection of the starting triangle for the walk. Section 3 describes our new proposed algorithm, Section 4 shows experiments comparing our solution with the existing algorithms. Section 5 summarizes the paper.

2 State of the Art

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 to find the triangle containing the query point *(walking)*.

Clever selection of the initial triangle may radically improve the speed of the process. Some approaches provide solutions using additional information about the data, such as the range of the mesh vertices [2], take advantage of sorted vertices [3] or sort them properly prior to the location ([4],[5], [6], [7]).

Without any additional memory, we can speed up the process by selecting the initial triangle as the closest one from a randomly chosen subset of triangles [8]. An ideal size of such a set is $O(\sqrt[3]{n})$ [9] for a random input.

More efficient solutions lead to some additional memory consumption. [10] proposed a method simplifying the mesh and locating the point in the simplified version first. From the triangulation T with n vertices, only $m = k \cdot n$ vertices (where $k \in (0, 1)$) are randomly selected, triangulated and so a higher layer for the location is created. The number of triangles is much smaller in the new layer and it radically improves the speed of a walk in it. If m is still bigger than a chosen size, other layers are constructed in the same way. The point location then runs in several steps. First, the triangle containing the query point is found on the highest layer. The closest vertex of this triangle defines the starting point for the walk in the lower layer, until the triangle in the lowest layer is found. In each layer, the walk is short and therefore fast. [11] analyses this algorithm, specifies the computational complexity as $O(\log n)$ for any input and proposes the optimal value of $k = 0.025$ which is valid for random input and leads to the best rate between speed and memory usage.

[12] introduce a bucketing method, which uses a uniform grid to quickly find a proper initial triangle. Empty cells slow down the algorithm, therefore it is suitable mainly for uniformly distributed vertices. Some algorithms (e.g., [13], [14]) try to avoid the sensitivity of the original bucketing method to data uniformity by using adaptive structures instead of a uniform grid. However, on highly non-uniform data, the dynamic hierarchy algorithm mentioned above [10], [11] still provides better results with lower additional memory.

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

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 [15]. The algorithm starts in the initial triangle and uses this 2D orientation test to move to its neighbors until it reaches the query point:

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

where points $\mathbf{t}, \mathbf{u}$ define an oriented line and $\mathbf{v}$ is the tested point.

The algorithm tests the edges of the current triangle in a deterministic order, leading to the fact that the walk may loop for non-Delaunay triangulations (see Figure 3a). [16] proposed an algorithm avoiding the loops 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 speeded up by remembering this edge and skipping the

test. The stochastic walk has been shown in [17] to need $O(\sqrt{n \cdot \log n})$ expected time for uniform data.

[1] proposed a visibility walk algorithm which uses barycentric coordinates instead of the 2D orientation test. Barycentric coordinates b_0, b_1, b_2 describe the position of a point $\mathbf{q}$ with respect to a triangle $\tau_{t_0 t_1 t_2}$ (see Figure 2). The point $\mathbf{q}$ is an affine combination of $\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}$:

$$\mathbf{q} = b_0 \cdot \mathbf{t_0} + b_1 \cdot \mathbf{t_1} + b_2 \cdot \mathbf{t_2} \tag{2}$$

where $b_i \in R$, $i = 0, 1, 2$, $b_0 + b_1 + b_2 = 1$.

b_i can by computed using the 2D orientation tests:

$$b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{q}) = \frac{orientation2D(\mathbf{t}_{[(i+\mathbf{1}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{2}) \mod \mathbf{3}]}, \mathbf{q})}{orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})} \tag{3}$$

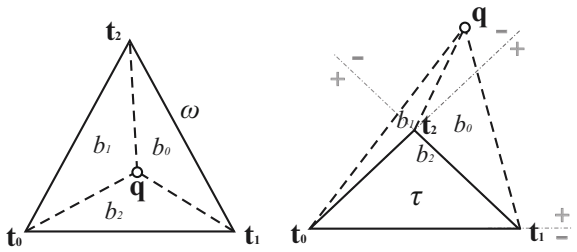

Fig. 2. Barycentric coordinates of $\mathbf{q}$ inside a triangle ω, $\mathbf{b}(\omega) = (0.25, 0.35, 0.4)$, and outside a triangle τ, $\mathbf{b}(\tau) = (-0.75, -0.25, 2)$

Each barycentric component b_i corresponds to one edge of the triangle, defining on which side of this edge the point $\mathbf{q}$ lies: either inner (positive value), or outer (negative value) side of the triangle, disregarding the orientation of the triangle. The higher the value of the component is, the bigger the triangle defined by the edge and the query point is. The walk via bigger triangles is usually shorter, therefore, the algorithm takes advantage of it by crossing such an edge of the triangle that corresponds to the component with the highest negative value.

The barycentric components of $\mathbf{q}$ are computed for each visited triangle. Since the third component can be computed from the other two components, we need three 2D orientation tests for each visited triangle. Although not stated by [1], the algorithm may loop in some rare cases - see Figure 3b. In the triangle τ, the area s_2 is greater than s_1, so the walk does not cross the edge leading to the triangle containing $\mathbf{q}$. A similar case happens in other thin triangles.

Straight walk algorithms walk along 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. This way, the walk is short.

The standard straight walk algorithm [16,19] chooses $\mathbf{p}$ as any of the starting triangle vertices and moves around it until it finds the triangle intersected by

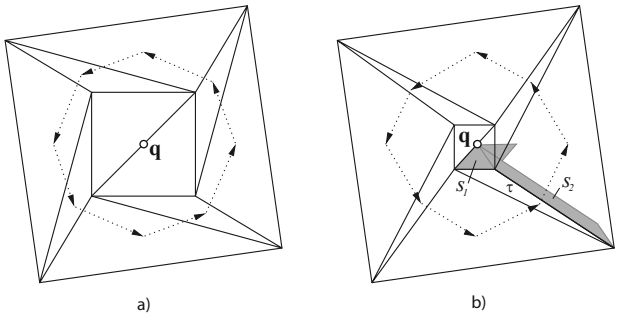

Fig. 3. a) Loop of Lawson's oriented walk [18] and b) Loop of the Barycentric walk

the line segment $\overrightarrow{\mathbf{pq}}$. Then it follows this line segment, using one orientation test to choose the proper edge, and another for checking if $\mathbf{q}$ is still on the outer side of this edge.

Orthogonal walks first navigate along one coordinate axis and then along the other, which makes the walk longer, but the local tests much cheaper, since only components of the coordinates are compared during the walk.

The original orthogonal walk [16] chooses one vertex of the starting triangle and then follows the horizontal line defined by this vertex, until it finds the triangle intersected by the vertical line defined by the query point. Then it continues along this line to the target triangle. This approach has a significant drawback: the border of a triangulation may be crossed during the walk, in which case a special modification is needed, resulting in a slower location process and additional implementation effort.

A speed-up of this walk was proposed in [2], where fewer tests are done during the location for a price that the walk may not find the correct target triangle, but only a triangle in its neighborhood, and a visibility walk algorithm is used for the final short location. The algorithm does not need any modification for dealing with the crossing of the border, but the final location by a visibility walk in this case may be much longer.

3 Proposed Algorithm

Our algorithm is based on the original barycentric walk [1]. However, unlike [1] it does not use the barycentric coordinates, but only the orientation tests of the edges, with respect to their relation to barycentric coordinates.

To decide which edge to cross to the next triangle, we compute the same orientation test as the Lawson's oriented walk [15], but we use not only its sign, but also the values. Let us call the proposed algorithm *direct walk* because the purpose of this change is to maintain the shortest walk of [1] while speeding up the algorithm by performing less orientation tests in each triangle.

If we denote the value of the orientation test for the point $\mathbf{q}$ and an edge opposite to vertex $\mathbf{v_i}$ as c_i (let us call it *orientation coordinate*), we can derive interesting properties from its relation to the barycentric coordinate b_i:

$$c_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{q}) = b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{q}) \cdot orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}) \quad (4)$$

where $c_i \in R$, $i = 0, 1, 2$ and $c_0 + c_1 + c_2 = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})$.

Our algorithm uses the properties of the orientation coordinates in the following way. In each triangle, it uses the orientation coordinate c_{prev} corresponding to the edge crossed to this triangle. The new coordinate c_i for this edge can be obtained as $c_i = -c_{prev}$. The coordinate $c_{[(i+1) \mod 3]}$ is computed using the orientation test (Eq. 1), and the last coordinate $c_{[(i+2) \mod 3]}$ is derived from the knowledge of the area A of the current triangle as $c_{[(i+2) \mod 3]} = 2A - c_i - c_{[(i+1) \mod 3]}$ (the barycentric property), where $2A = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})$.

As a result, for each visited triangle we have to compute two orientation tests: one for an edge and one to get the doubled area of the triangle (three for the first triangle, where we have to compute c_{prev} as well). In applications working with areas of the triangles of the input mesh, one may prefer to store them. The use of these values during the location then results in the need of only one orientation test per visited triangle (two for the first triangle).

Just as the barycentric walk, our algorithm is resistant to variable orientation of the triangles. If there is a possibility of such a case, we need to test the sign of the triangle area and in case of the negative sign, we reverse the signs of all the orientation coordinates. The orientation information can be easily obtained from the $2A$ value which produces a signed value. For more details, see the pseudo-code in Algorithm 1.

Input: the query point $\mathbf{q}$, the chosen starting triangle $\alpha \in T$
Output: the triangle ω which contains $\mathbf{q}$
integer i;
edge ϵ;
triangle $\tau = \alpha = \mathbf{t_0 t_1 t_2}$;
double min, c_0, c_1, c_2, c_a;
$c_0 = orientation2D(\mathbf{t_1}, \mathbf{t_2}, \mathbf{q})$;
$c_1 = orientation2D(\mathbf{t_2}, \mathbf{t_0}, \mathbf{q})$;
c_a = stored double area of τ; // or $c_a = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})$
$c_2 = c_a - c_0 - c_1$;
// the following condition is necessary for non-uniform triangle orientation
// if $c_a < 0$ then begin $c_0 = -c_0$; $c_1 = -c_1$; $c_2 = -c_2$; end
min = minimal c_j where $j \in \{0, 1, 2\}$;
ϵ = edge corresponding to minimal c_j;
while $min < 0$ **do**
    τ = neighbour of τ over ϵ;
    i = index of ϵ in τ;
    $c_i = -min$;
    $c_{[(i+1) \mod 3]} = orientation2D(\mathbf{t}_{[(i+\mathbf{2}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{3}) \mod \mathbf{3}]}, \mathbf{q})$;
    c_a = stored double area of τ; // or $c_a = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})$
    $c_{[(i+2) \mod 3]} = c_a - c_i - c_{i+1}$;
    // the following condition is necessary for non-uniform triangle orientation
    // if $c_a < 0$ then begin $c_0 = -c_0$; $c_1 = -c_1$; $c_2 = -c_2$; end
    min = minimal c_j where $j \in \{0, 1, 2\}$;
    ϵ = edge corresponding to minimal c_j;
end
return τ;

Algorithm 1. Direct walk

Often, there is a need for a faster algorithm for a price of some memory consumption, in which case, we offer to use our walking algorithm in a combination with a hierarchical structure proposed by [11] favoured for its low memory requirements, an easy modification (in terms of adding or deleting vertices from the input triangle mesh) and excellent results for non-uniform data. For the detailed description of the hierarchical structure, see Section 2. Here we will describe its combination with our method.

Our algorithm is suitable for such a structure thanks to its fast initialization step, which is performed in each layer. Also, when moving to the lower layer, we do not have to compute which vertex of the target triangle is closest to the query point - we already know its orientation coordinates and can therefore easily select the vertex with the highest orientation coordinate c_i and use it as the starting for the walk in the lower layer. This way, we save three distance computations in each descend. Also, since there is only a fragment of the original number of the triangles in the higher layers, we can store the area information there and thus speed up the algorithm even more for a low additional memory consumption.

4 Experimental Results

We selected the most popular and also the fastest of the mentioned algorithms to compare with our method: the remembering walk (RW) [15], the remembering stochastic walk (RSW) [16], the barycentric walk (BW) [1], the straight walk (SW) [16], and the orthogonal walk (OW) [2].

For the test purposes, we implemented the specified algorithms in C++ and tested on Intel Q6600 2,40GHz. SSE2 random generator was used for RSW algorithm since is declared as up to five times faster then the standard C random generator [21].

The tests were performed on the triangulations on three types of datasets: randomly distributed points in a unit square, LIDAR data and data from a cadastre. On each dataset, we constructed four types of triangulations: Delaunay (DT), Greedy, MWT and Min-max angle. The results were similar, so we present them on the most popular triangulation type, DT.

In each case, we performed 10^7 location processes and computed the average number of the tested quantities. The following qualities were examined for each algorithm: the average number of visited triangles ($\#\Delta$), the average number of tests ($\#tests$) and the average time per one location ($t[\mu s]$). Note that the number of tests done by each algorithm is presented only to measure how many tests per triangle the particular algorithm does on average, not to compare the performance of the algorithms - for that, we should use the time values, since the speed of their tests differ among the algorithms. The properties $\#tests$ and $\#\Delta$ consists of two values for OW: the former value concerns the walk, the latter concerns the final location performed by RW.

Table 1 shows the results of the tests for a random initial triangle and a random target point. Table 2 presents the performance of the algorithms when they are used in the hierarchical approach by [11]. The hierarchical approach was

tested on randomly generated rectangular datasets, the datasets for testing the original algorithms were bounded by a rectangle and retriangularized to obtain a fair measurement of the orthogonal walk, which is slower for non-rectangular data, where it often crosses the border of the triangulation.

Two versions of our algorithm were tested, DW1 uses the precomputed area information, DW2 does not. For the hierarchical approach, DW2 stores the area information only for the higher layers, not for the original mesh.

Algorithms that performed remarkably worse with the selected hierarchical structure, such as the original barycentric walk and the straight walk, were not included in the final tests.

The results in Table 1 confirm that the shortest length of the barycentric walk was maintained and its speed was improved. Although the barycentric walk without the precomputed areas (DW2) is still slightly slower than the RW algorithm, it can be preferred when the triangle orientation varies throughout the triangulation. If the intended application uses the triangle areas for other purposes, DW1 can be used to obtain the best performance.

When combined with the hierarchical structure (see Table 2), our algorithm becomes the fastest, thanks to the short walk and avoiding the computation of

Table 1. Comparison of the walking algorithms with randomly chosen α ($\#\Delta$ represents the number of visited triangles, $\#tests$ represents the number of performed tests, $t[\mu s]$ represents the time per one location)

	$\#\Delta$	$\#tests$	$t[\mu s]$	$\#\Delta$	$\#tests$	$t[\mu s]$	$\#\Delta$	$\#tests$	$t[\mu s]$
	ϕ **per located point**			ϕ **per located point**			ϕ **per located point**		
Cadastre data									
	4897 vertices (9774 Δ)			15824 vertices (31642 Δ)			70437 vertices (140868 Δ)		
RW	94.4	129.5	3.48	160.5	213.2	6.52	321.3	417.9	14.90
RSW	92.1	122.2	5.89	158.1	208.5	10.71	312.9	414.8	23.04
BW	87.4	262.2	5.92	149.6	448.8	10.72	283.1	849.2	21.87
SW	2.8+88.2	3.3+175.4	4.75	2.7+149.6	3.2+298.3	8.63	2.8+294.8	3.3+588.9	18.72
OW	94.2+4.3	192.6+7.0	2.01	176.5+2.7	357.3+5.2	4.32	319.0+3.5	642.0+6.4	9.50
DW2	87.4	175.8	4.34	149.6	300.2	8.00	283.1	567.2	17.43
DW1	87.4	88.4	2.87	149.6	150.6	5.48	283.1	284.1	12.63
LIDAR									
	34932 vertices (69858 Δ)			313348 vertices (626690 Δ)			3722068 vertices (7444130 Δ)		
RW	205.5	275.4	8.98	613.5	823.3	38.18	2569.1	3444.2	181.51
RSW	202.6	265.6	14.26	608.5	792.6	54.09	2543.3	3305.4	248.08
BW	179.3	537.9	13.34	509.1	1527.3	47.24	2203.3	6609.8	222.76
SW	2.7+180.8	3.2+360.6	10.94	2.7+542.8	3.2+1084.6	42.98	2.8+2316.4	3.3+4631.8	202.65
OW	240.7+1.7	485.8+3.8	6.60	695.0+1.7	1394.3+3.8	31.88	2741.8+2.2	5487.8+4.5	148.45
DW2	179.3	359.6	10.20	509.1	1019.2	38.41	2203.3	4407.6	184.06
DW1	179.3	180.3	7.23	509.1	510.1	29.90	2203.3	2204.3	147.73
Randomly distributed points in the unit square									
	10^4 vertices (19994 Δ)			10^5 vertices (199994 Δ)			10^6 vertices (1999994 Δ)		
RW	118.1	159.2	4.34	366.1	491.5	19.43	1144.7	1533.0	81.39
RSW	115.8	152.5	7.40	362.5	474.5	29.02	1130.5	1476.5	110.94
BW	103.2	309.6	7.08	326.8	980.1	27.60	1028.2	3084.5	105.12
SW	2.8+105.5	3.3+210.0	5.71	2.7+335.6	3.2+670.2	23.56	2.7+1065.4	3.2+2129.8	93.96
OW	137.8+1.8	279.9+3.9	2.84	433.3+1.7	870.8+3.8	15.71	1336.3+1.8	2676.9+3.9	72.42
DW2	103.2	207.4	5.22	326.8	654.6	21.71	1028.2	2057.4	86.89
DW1	103.2	104.2	3.46	326.8	327.8	16.16	1028.2	1029.2	69.37

Table 2. Comparison of the walking algorithms with hierarchical structure ($\#\Delta$ represents the number of visited triangles, $\#tests$ represents the number of performed tests, $t[\mu s]$ represents the time per one location)

	$\#\Delta$	$\#tests$	$t[\mu s]$	$\#\Delta$	$\#tests$	$t[\mu s]$	$\#\Delta$	$\#tests$	$t[\mu s]$
	ϕ **per located point**			ϕ **per located point**			ϕ **per located point**		
	10^2 vertices (2 layers)			10^3 vertices (2 layers)			10^4 vertices (3 layers)		
RW	7.8	10.8	0.40	13.4	18.7	0.60	24.0	32.0	1.06
RSW	7.6	11.2	0.60	13.2	18.4	0.96	22.6	29.0	1.63
OW	10.8 + 2.3	25.9 + 3.3	0.63	13.8 + 3.5	31.7 + 7.4	0.74	19.5 + 4.8	43.1 + 10.6	1.08
DW2	6.3	13.5	0.35	11.8	19.8	0.55	20.7	30.1	0.87
DW1	6.3	8.3	0.22	11.8	13.8	0.40	20.7	23.7	0.71
	10^5 vertices (3 layers)			10^6 vertices (4 layers)			10^7 vertices (4 layers)		
RW	30.9	41.2	1.36	24.5	30.8	1.21	35.5	47.9	1.84
RSW	27.0	36.7	1.95	23.3	32.1	1.81	36.3	48.8	2.88
OW	25.8 + 5.8	58.2 + 12.4	1.36	23.8 + 5.9	51.9 + 14.2	1.43	30.5 + 9.27	65.3 + 18.25	1.87
DW2	22.7	29.3	0.90	22.0	30.4	0.94	31.3	45.0	1.58
DW1	22.7	25.7	0.81	22.0	26.0	0.84	31.3	35.3	1.34

distances in each descend. Note that between the location time for 10^5 and 10^6 vertices, there is not much difference. The explanation is simple - 10^5 vertices is just below the limit for the creation of a new layer, while for 10^6 vertices the new layer is created.

Although we identified a situation where both the barycentric walk and our improvement can loop, during our thorough tests on the types of triangulations listed above, neither of the algorithms looped.

5 Conclusion

We presented a modification of the barycentric approach for the point location and proposed how to combine it with a popular hierarchical structure to gain more speed-up than in combination with other walking algorithms.

We compared the performance of our algorithm with the most popular and also the fastest of the existing walking algorithms. We compared both the original algorithms and their combination with the selected hierarchical data structure. Our approach proved to be faster than the original, while maintaining its advantages. When combined with the hierarchical structure, our algorithm becomes the fastest, thanks to its suitability for the structure.

Acknowledgement. This work has been supported by the Czech Science Foundation under the project P202/10/1435 and by University of West Bohemia under the project SGS-2010-02.

References

1. Sundareswara, R., Schrater, P.: Extensible point location algorithm. In: International Conference on Geometric Modeling and Graphics, pp. 84–89 (2003)

2. Soukal, R., Kolingerová, I.: Star-shaped polyhedron point location with orthogonal walk algorithm. Procedia Computer Science 1, 219–228 (2010)
3. Purchart, V., Kolingerová, I., Beneš, B.: Interactive sand-covered terrain surface model with haptic feedback. In: GIS Ostrava 2012 - Surface Models for Geosciences (2012)
4. Sloan, S.W.: A fast algorithm for constructing Delaunay triangulations in the plane. Advanced Engineering Software 9, 34–55 (1987)
5. Zhou, S., Jones, C.B.: HCPO: an efficient insertion order for incremental Delaunay triangulation. Information Processing Letters 93, 37–42 (2005)
6. Amenta, N., Choi, S., Rote, G.: Incremental constructions con brio. In: SCG 2003: Proceedings of the 19th Annual Symposium on Computational Geometry, pp. 211–219. ACM, New York (2003)
7. Buchin, K.: Incremental construction along space-filling curves. In: EuroCG 2005: Proceedings of the 21th European Workshop on Computational Geometry, pp. 17–20 (2005)
8. Mücke, E.P., Saias, I., Zhu, B.: Fast randomized point location without preprocessing in two and three-dimensional Delaunay triangulations. In: Proceedings of the 12th Annual Symposium on Computational Geometry, vol. 26, pp. 274–283 (1996)
9. Devroye, L., Mucke, E.P., Zhu, B.: A note on point location in Delaunay triangulations of random points (1998)
10. Mulmuley, K.: Randomized multidimensional search trees: Dynamic sampling. In: Proceedings of the 7th Annual Symposium on Computational Geometry, pp. 121–131 (1991)
11. Devillers, O.: The Delaunay hierarchy. International Journal of Foundations of Computer Science 13, 163–180 (2002)
12. Su, P., Drysdale, R.L.S.: A comparison of sequential Delaunay triangulation algorithms. In: Proceedings of the 11th Annual Symposium on Computational Geometry, pp. 61–70 (1995)
13. Zadravec, M., Žalik, B.: An almost distribution independent incremental Delaunay triangulation algorithm. The Visual Computer 21, 384–396 (2005)
14. Žalik, B., Kolingerová, I.: An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. International Journal of Geographical Information Science 17, 119–138 (2003)
15. Lawson, C.L.: In: Mathematical Software III; Software for C1 Surface Interpolation, pp. 161–194. Academic Press, New York (1977)
16. Devillers, O., Pion, S., Teillaud, M.: Walking in a triangulation. In: Proceedings of the 17th Annual Symposium on Computational Geometry, pp. 106–114 (2001)
17. Zhu, B.: On Lawson's Oriented Walk in Random Delaunay Triangulations. In: Lingas, A., Nilsson, B.J. (eds.) FCT 2003. LNCS, vol. 2751, pp. 222–233. Springer, Heidelberg (2003)
18. Weller, F.: On the total correctness of Lawson's oriented walk. In: Proceedings of the 10th International Canadian Conference on Computational Geometry, pp. 10–12 (1998)
19. Mehlhorn, K., Näher, S.: Leda: A platform for combinatorial and geometric computing. Communications of the ACM 38, 96–102 (1995)
20. Soukal, R., Kolingerová, I.: Straight walk algorithm modification for point location in a triangulation. In: EuroCG 2009: Proceedings of the 25th European Workshop on Computational Geometry, Brussels, Belgium, pp. 219–222 (2009)
21. Owens, K., Parikh, R.: Fast random number generator on the intel pentium 4 processor. Intel Software Network (2009)

Appendix D

Walking algorithm for point location in a triangulated non-convex domain with holes

Soukal, R., Kolingerová, I.

Submitted to (currently in the first revision):

International Journal of Geographical Information Science, Taylor & Francis, IF 1.479 (2013)

International Journal of Geographical Information Science
Vol. 00, No. 00, Month 200x, 1–21

RESEARCH ARTICLE

Walking algorithm for point location in a triangulated non-convex domain with holes

Roman Soukal[a*] and Ivana Kolingerová[ab]

[a]*Department of Computer Science and Engineering, Faculty of Applied Sciences, Univesity of West Bohemia, Univerzitní 8, 306 14 Pilsen, Czech Republic*
[b]*NTIS - New Technologies for Information Society, Univesity of West Bohemia, Univerzitní 8, 306 14 Pilsen, Czech Republic*
(*Received 00 Month 200x; final version received 00 Month 200x*)

Retrieval of the triangle in a triangulated terrain model which contains a given point (the so-called point location problem) is a frequent task in geosciences and GIS software. The algorithm solving this problem has to be efficient as a plethora of such searches has to be performed when constructing, analyzing or modifying a terrain model. So-called walking algorithms provide solutions with acceptable computational as well as implementation complexity, low memory requirements and flexibility in the case of the terrain model modification. However, existing walking algorithms concentrate on convex triangulated domains only and do not take into account the needs of some geography and cartography applications where non-convex shapes of the triangulated domain are often used. In this paper, we propose a new algorithm for location of a point in a triangulated non-convex domain with holes, based on the walking principle. As far as we know, it is the only walking algorithm for non-convex shapes of the triangulated domain. The algorithm was tested and verified on real cadaster data, on CAD data sets and on artificially generated examples with positive results.

Keywords: Terrain model; Non-convex triangulated domain; Triangle mesh; Location algorithm

*Corresponding author. Email: soukal@kiv.zcu.cz

ISSN: 1365-8816 print/ISSN 1362-3087 online

DOI: 10.1080/1365881YYxxxxxxxx
http://www.informaworld.com

1. Introduction

For a given query point, the point location problem in a triangulated terrain model means to find the triangle from the model, in which the given point lies. It is a frequent operation in the GIS algorithms and software: it is needed in the construction and the modification of the triangulated terrain model (Amenta et al. 2003, Boissonnat and Teillaud 1986, Buchin 2005, Dæhlen et al. 2001, Devillers 2002, Green and Sibson 1978, Lawson 1977, Purchart et al. 2012, Sloan 1987, Su and Drysdale 1995, Zadravec and Žalik 2005, Žalik, B. and Kolingerová, I. 2003), in the interactive inspection of the model (Purchart et al. 2012), in the identification of the place where some shapes are to be inserted (Dæhlen et al. 2001, Koch 2005, Schilling et al. 2007, 2009), in the accuracy verification of the model (Höhle et al. 2010), etc. Usually, many point location queries have to be performed, so the location algorithm should be efficient - with low computational complexity, and it should be easily programmed and maintained. Moreover, triangulated model can be frequently changed and vertices can move over time, therefore location algorithm should be able to handle these possibilities.

Due to the importance of the point location problem, many algorithms have been developed, but not all of them are suitable for a non-convex shape of the triangulated domain. Optimal computational complexity $O(\log n)$ for a model with $O(n)$ triangles is achieved at a price of additional memory for location data structures and more complicated manipulation, especially in the case when the triangulated model is frequently changed or vertices are moving over time. Therefore, the so-called walking algorithms are often used, thanks to their simplicity, still sublinear expected complexity and ability to handle triangulated models which are changing over time without significant changes.

The name of the walking algorithms describes their principle: the search goes from a triangle to its neighbor in the direction of the given query point until the target triangle, containing the given point, is found. No additional data structures are needed; just neighborhood relations between triangles have to be available. The starting triangle for such a walk may be arbitrary; however, an appropriately selected starting triangle may radically shorten the length of the walk.

For terrain models, the point location is usually formulated as a planar problem: for a given planar triangle mesh of a set of points, bounded by its convex hull, find the triangle which contains the given planar point. The omission of the third coordinate giving the height of points does not make a problem as this simplification is generally accepted and widely used in the terrain representation. However, as far as we know, all existing walking algorithms concentrate on convex triangulated areas only and the limitation to a convex shape of the triangulated domain might be too restricting. Terrain data can have any non-convex shape, even with holes. For example, it is ordinary for common LiDAR data, where water surfaces usually absorb laser rays (Antonarakis et al. 2008).

It seems that the solution can be an additional triangulization of the non-triangulated areas. But in practice, since vertices are missing in these areas, it causes a variety of complications. For example, the triangles near the border of the convex hull are skinny and cause troubles in modeling and visualization, such as incorrect shading (Shewchuk 2002) and erroneous shape of contour lines computed on such triangles (Kolingerová et al. 2009). Moreover, additionally triangulated areas often contain lots of long narrow triangles which make standard walking algorithms less effective and may cause numerical problems. Therefore, we have developed a new walking algorithm, suitable for non-convex triangulated domains with holes without any preprocessing (see the illustrative example in Fig. 1) and present it in this paper. Note that an auxiliary data structure for bound-

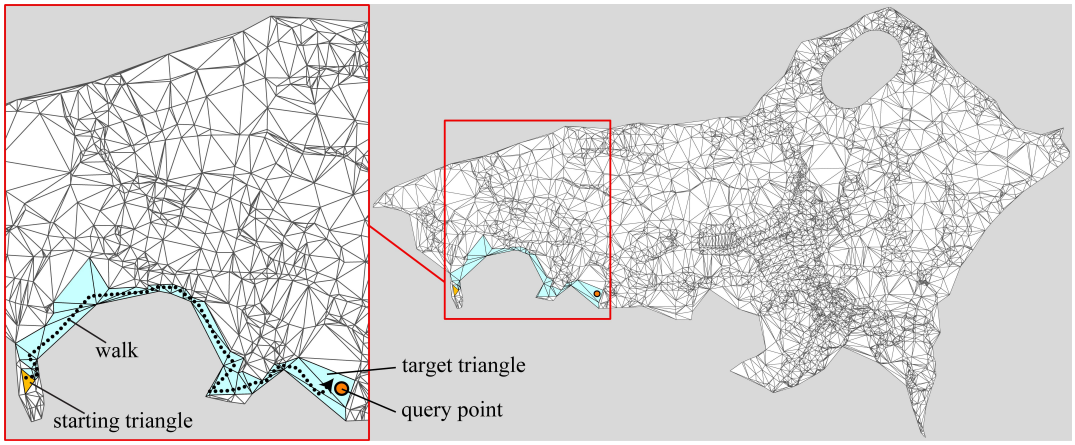

Figure 1. Illustrative example of a walking algorithm in a triangulated non-convex domain with holes (cadaster data)

ary representation is not expected since its construction and maintenance have similar drawbacks as other additional data structures.

The paper is organized as follows. Section 2 describes the notation and the mathematical background used throughout the text. Section 3 provides an overview in the task of a point location in a triangulated non-convex domain with holes and presents some modifications of existing methods to solve the problem. Section 4 describes the proposed walking algorithms for point location in a triangulated non-convex domain with holes. Section 5 presents our experiments performed on real geodetic and CAD datasets and shows detailed results of the algorithm, Section 6 concludes the paper.

2. Basic Notation and Mathematical Backgorund

The term *border* (of the triangle mesh) denotes both the outer border of the triangulated domain and the border of an inner hole.

Vectors, points and vertices are denoted by bold lower case characters (e.g. $\mathbf{a}, \mathbf{b}$). Scalar variables are denoted by lower case characters in italic (e.g., k, l). For an oriented line segment between two points (e.g., points $\mathbf{a}, \mathbf{b}$ where $\mathbf{a} \neq \mathbf{b}$), $\overrightarrow{\mathbf{ab}}$ is used.

The letter ϵ denotes an edge of a triangle, subscripts are used to specify vertices that belong to this edge. Other lower case Greek letters are used for triangles. A query point is usually denoted as $\mathbf{q}$, T is the triangle mesh in which we want to locate $\mathbf{q}$. The starting triangle of the walking is always marked as α ($\alpha \in T$), the triangle which contains $\mathbf{q}$ is denoted as ω ($\omega \in T$), the i-th triangle visited by the walking algorithm is denoted as τ_i.

To determine the position of a point $\mathbf{v}$ with respect to an oriented edge (or oriented line) $\overrightarrow{\mathbf{tu}}$, we use the sign of the determinant in the so-called 2D orientation test (Devillers et al. 2001):

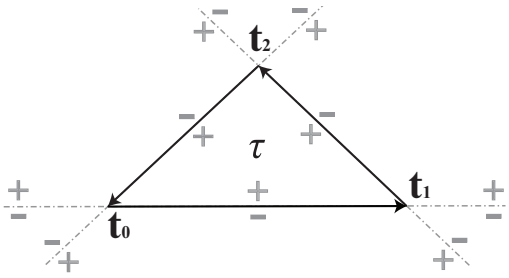

Figure 2. Example of orientation tests for triangle edges

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

where the positive value is returned for $\mathbf{v}$ on the left of $\overrightarrow{\mathbf{tu}}$ and negative for $\mathbf{v}$ on the right of $\overrightarrow{\mathbf{tu}}$.

Figure 2 shows the resulting signs of orientation tests from Equation 1 for a triangle $\tau_{\mathbf{t_0 t_1 t_2}}$ with the CCW order of vertices.

3. State of the Art

Let us describe two main categories of point location algorithms: the walking algorithms and the algorithms using additional location data structures.

3.1. *Walking algorithms*

There are several walking algorithms solving point location problem, and according to the style how they determine the way of the walk, they can be divided into three groups: visibility, straight and orthogonal walks. The survey of these algorithms is provided in (Soukal et al. 2012b, Devillers et al. 2001). If the triangulated domain is convex, most of walking algorithms never cross the border of the triangle mesh, however, as far as we know, there is no walking algorithm solving complex cases such as triangle meshes with holes or triangle meshes with non-convex boundaries.

Visibility algorithms perform local tests (usually 2D orientation tests - see Equation 1) in each triangle they walk through. These tests look for such an edge which defines a line separating the query point from the third vertex of the triangle. The walk then moves across this edge to the neighboring triangle. For a triangulated convex domain, it never crosses the border of the triangle mesh, however, deterministic versions of visibility walk algorithms (Lawson 1977) may loop for non-Delaunay triangulations (Devillers et al. 2001, Weller 1998). It has been shown that for a planar Delaunay triangulation, the deterministic versions cannot loop (Floriani et al. 1991, Weller 1998). For non-Delaunay

triangulations, a randomized (stochastic) version exists (Devillers et al. 2001), it is slower, because the randomization step is done in each triangle, but it does not loop. The stochastic walk has been shown in Zhu (2003) to need $O(\sqrt{n \cdot \log n})$ expected time for the uniform data. Kolingerová (2006), Soukal et al. (2012a) present improvements which save some tests and bring speed-up, but may also loop for non-Delaunay triangulations. Sundareswara and Schrater (2003) presents an algorithm which uses barycentric coordinates and as well as Soukal et al. (2012a) does not require a consistent orientation of triangle vertices. However, this algorithm is slow and may also loop for non-Delaunay triangulations.

Straight walk algorithms use not only local comparisons to determine where to walk, but also use a line connecting one point of the starting triangle with the query point and traverse triangles crossed by this line. This way, their walk is short and does not loop. For convex triangulations, the algorithms never cross the border of the triangulation and they do not loop. The straight walk has been proved to visit $O(\sqrt{n})$ triangles in the expected case and uniform distribution of vertices (Green and Sibson 1978, Mücke et al. 1996). The standard straight walk algorithm (Mehlhorn and Näher 1995, Devillers et al. 2001), which uses 2D orientation tests (see Equation 1), was sped up by Soukal and Kolingerová (2009) using faster tests.

Orthogonal walks first navigate along one coordinate axis and then along the other, which makes the local tests cheaper, since only coordinate components are compared during the walk. However, more triangles are visited during the walk. The domain boundary may be crossed during the walk, in which case a special modification is needed, resulting in a slower location process and additional implementation effort. Bounds based on Boissonnat and Teillaud (1993) show that the orthogonal walk has similar complexity as the straight walk (Devroye et al. 1998, Mücke et al. 1996). The original orthogonal walk (Devillers et al. 2001) does not handle boundary crossing during the location. Such a situation usually causes location failure (the target triangle is not found). It is solved in the modification (Soukal and Kolingerová 2010) which also brings the speedup of the original algorithm.

A clever selection of the starting triangle for walking may radically improve the speed of the algorithm, since it reduces the number of visited triangles during the walk. If any additional information about the data is known, it can be used in the selection which speeds up the process without any additional memory and additional time requirements (Soukal and Kolingerová 2010, Sloan 1987, Zhou and Jones 2005, Amenta et al. 2003, Buchin 2005, Purchart et al. 2012). Without any knowledge of the data, the initial triangle can be chosen randomly. A better, yet still fast and simple alternative without any additional memory use was proposed in Mücke et al. (1996), where the initial triangle is selected as the nearest triangle from a set A of randomly chosen triangles from T, where $\|A\| \ll \|T\|$. Note that the subset A is generated randomly on the fly for each location process, thus no additional data structure is needed. For Delaunay triangulation of random points, an analysis of the ideal size of such a random subset has been proposed by Devroye et al. (1998), leading to the size of $O(\sqrt[3]{n})$.

3.2. *Point location using additional data structures*

Efficient point location solutions usually use hierarchical structures, but it leads to some additional memory consumption. Most of these solutions are used to find such a starting triangle for the walking algorithm, which is close to the triangle containing the query point. The final location is then performed by the walking algorithm and is usually

short. But as we mentioned in the previous subsection, walking algorithms do not handle triangle meshes with holes or triangle meshes in non-convex domains, therefore, these algorithms should be modified to make the location process reliable.

Mulmuley (1991) proposed a method simplifying the Delaunay triangulation to a smaller one and locating the point in the simplified version first. From the triangulation T with n vertices, only $m = k \cdot n$ vertices (where $k \in (0, 1)$) are randomly selected, triangulated and in this way a higher layer for the location is created. The number of triangles is much smaller in the new layer and it reduces the walk length and speeds up the location process. If m is still bigger than a chosen size, more layers are computed in the same way. A point location then runs in several steps. First, the triangle containing the query point is found in the highest layer. The nearest vertex of this triangle defines a starting point for the walk in the lower layer, until the triangle in the lowest layer is found. In each layer, the walk is short and therefore fast. Devillers (2002) analyses this algorithm, precises the time complexity to $O(\log n)$ for any input and proposes an optimal value of $k = 0.025$ which is valid for a random input and leads to the best rate between speed and memory use. The possible solution how to make this algorithm working for non-convex triangulated domains or domains with holes is to additionally triangulate the holes and non-convexities in the lowest layer and save a flag for these new triangles to easily decide, if the query point is outside the triangle mesh (the query point is inside the triangle with this flag).

Su and Drysdale (1995) introduce a bucketing method, which uses a uniform grid to quickly find a proper initial triangle. Each cell of the grid (a bucket) contains at most one vertex of the triangulation. During the point location, the cell containing the query point is found. If it contains also a triangulation vertex, the walk starts from this vertex (from any triangle formed by this vertex), if it does not, the nearest cell containing the vertex is found using a spiral search. As a bigger number of empty cells causes a longer spiral search, the algorithm is useful mainly for triangle meshes with uniformly distributed vertices where most of the cells contain a vertex. For triangle meshes in non-convex triangulated domains or in domains with holes, each bucket can remember all the triangles incident to the bucket and then those triangles are tested. But this solution may consume a big additional amount of memory, especially for non-uniformly distributed vertices.

Some algorithms (e.g. Žalik, B. and Kolingerová, I. (2003), Zadravec and Žalik (2005)) try to avoid the sensitivity of the original bucketing method on data uniformity by using adaptive structures instead of a uniform grid and it would work for triangulated non-convex domains with holes without significant changes, but it has high memory requirements. However, on highly non-uniform data, the dynamic hierarchy algorithm (Mulmuley 1991, Devillers 2002) with additional triangualization of holes and non-convexities still provides better results with lower additional memory.

4. Proposed Algorithm

Let us have a query point and a triangulated domain which may have a non-convex shape and may contain holes. We suppose that the triangulated domain is edge-connected, i.e., each triangle is connected with the rest of triangles by at least one edge. This condition ensures that there exists a path for the walking between any two triangles. If the given triangulated domain does not fulfill this condition, e.g., the domain consists of several separated triangulated polygons - the algorithm still could be used, however, more

restarts for particular polygons would be necessary. The algorithm uses the 2D orientation tests during the location and assumes the triangle mesh to have all its triangles oriented in the same way - either all clockwise (CW) or all counterclockwise (CCW). If this assumption is not met, one more 2D orientation test determining the triangle orientation is needed for each triangle visited during the walk. In the following text, we assume the CCW order of the vertices.

The proposed algorithm consists of three phases: an *initialization*, a *walk* and a *border walk*. The initialization is performed only once at the start of the location process and then other two phases are used as many times as needed. The walk phase is used preferably and it switches to the border walk phase only if the border of the triangle mesh is reached and the walk phase cannot continue. The border walk phase tries to find a suitable triangle for the continuation of the walk phase. When such a triangle is found, the walk phase is invoked again, otherwise the algorithm ends with the result that $\mathbf{q}$ is outside the tested triangle mesh. Although the main idea is straightforward, particular realization should be complex and effective, therefore, especially in the border walk phase, we focus on performing the minimum number of mathematical tests.

The border walk phase gets its name because it walks around the border of the triangle mesh or around the border of the hole in the triangle mesh. The switching between the walk phase and the border walk phase is repeated until the triangle ω containing the query point $\mathbf{q}$ is found during the walk phase or a suitable triangle for the continuation of the walk phase is not found during the border walk phase ($\mathbf{q}$ is outside the triangle mesh).

We chose the straight walk algorithm (see Section 3.1) as a base of our algorithm, since its path is deterministic – under the same initial conditions, the resulting path will be always identical. Generally, the straight walk algorithm goes through all the triangles between $\mathbf{p}$ and $\mathbf{q}$ intersected by the oriented line $\overrightarrow{\mathbf{pq}}$, where the point $\mathbf{p}$ is chosen inside a starting triangle α, and $\mathbf{q}$ is the query point.

Although the straight walk algorithm presented by Soukal and Kolingerová (2009) is faster than the standard straight walk algorithm (Mehlhorn and Näher 1995, Devillers et al. 2001) due to cheaper tests, we chose the latter one (Mehlhorn and Näher 1995, Devillers et al. 2001), since the former algorithm utilizes a visibility walk algorithm for final location and it may reach the border during this final location.

We need to define some more terminology. Since the line $\overrightarrow{\mathbf{pq}}$ is oriented, for each triangle intersected by $\overrightarrow{\mathbf{pq}}$, let us denote its edge where $\overrightarrow{\mathbf{pq}}$ enters this triangle as the *input edge* and the edge where $\overrightarrow{\mathbf{pq}}$ leaves the triangle as the *output edge*. Thus, during the straight walk, for each visited triangle τ_i, the input edge of τ_i is the edge which was used to go to τ_i. The second edge of τ_i which is crossed by $\overrightarrow{\mathbf{pq}}$ is the output edge of τ_i. The output edge of τ_i is also the input edge of τ_{i+1}.

Furthermore, let us denote each edge, which is on the border of the triangle mesh, a *border edge*. If the border edge is also an input edge or an output edge, respectively, let us denote it a *border input edge* or *border output edge*, respectively (see Fig. 3, where ϵ_{b-in} are border input edges and ϵ_{b-out} are border output edges). A *border input triangle* is a triangle which contains a border input edge in its set of edges and a *border output triangle* is a triangle which contains a border output edge in its set of edges.

4.1. *Initialization Phase*

At the beginning of the initialization phase, we suppose that the first triangle α has been already chosen - randomly or in some clever way (see Section 3).

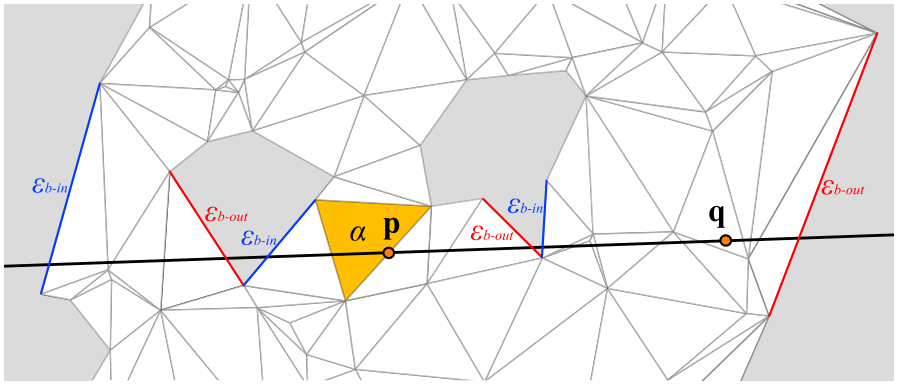

Figure 3. Example of border input edges ($\epsilon_{b\text{-}in}$) and border output edges ($\epsilon_{b\text{-}out}$) on a non-convex mesh with holes

There are two goals in the initialization phase: to choose suitably the point $\mathbf{p}$ ($\mathbf{p} \in \alpha$) defining $\overrightarrow{\mathbf{pq}}$ and to determine the edge of α intersected by $\overrightarrow{\mathbf{pq}}$ which will be used to go to the next triangle.

Before the point $\mathbf{p}$ is chosen, 2D orientation test (see Equation 1) is performed for all the edges of α similarly as in the visibility walk algorithms. Thus, the position of $\mathbf{q}$ with respect to the starting triangle edges is tested. If all the results are non-negative, the starting triangle contains the query point and the algorithm ends. Otherwise one edge with a negative value is selected and the point $\mathbf{p}$ defining $\overrightarrow{\mathbf{pq}}$ is chosen in the middle of that edge, which is used to determine the next triangle. The initialization phase is now finished. Note that three orientation tests are done during the initialization phase.

4.2. *Walk Phase*

There are two goals for each visited triangle τ_i in the straight walk during the walk phase: to find the output edge of τ_i intersected by $\overrightarrow{\mathbf{pq}}$ which will be used to go to the next triangle and to decide whether the current triangle τ_i contains $\mathbf{q}$ or whether the walk will continue.

For each triangle $\tau_i = (\mathbf{l_i}, \mathbf{r_i}, \mathbf{s_i})$, where the vertices $\mathbf{l_i}, \mathbf{r_i}$ determine the input edge $\epsilon_{l_i r_i}$ and the vertex $\mathbf{s_i}$ is opposite to $\epsilon_{l_i r_i}$, the algorithm determines the output edge by comparing the vertex $\mathbf{s_i}$ to $\overrightarrow{\mathbf{pq}}$ using the 2D orientation test. If $\mathbf{s_i}$ is on the right side of $\overrightarrow{\mathbf{pq}}$, the output edge is $\epsilon_{s_i l_i}$, otherwise, the output edge is $\epsilon_{r_i s_i}$.

Before the algorithm continues through the output edge to the next triangle, it computes the orientation test for the point $\mathbf{q}$ with respect to the output edge. If the point $\mathbf{q}$ is on the left side of the edge (the orientation test returns a positive sign), the final triangle containing $\mathbf{q}$ has been found. Otherwise, the walk continues through the output edge to the next triangle (see example of the walk phase in Figure 4). Note that two orientation tests per triangle are done during the walk phase.

If the output edge is also a border output edge, the walk phase cannot continue and switches to the border walk phase, which tries to find a triangle suitable for the continuation of the walk phase (a border input triangle intersected by $\overrightarrow{\mathbf{pq}}$). If such a triangle is found during the border walk phase, then the walk phase continues from this triangle,

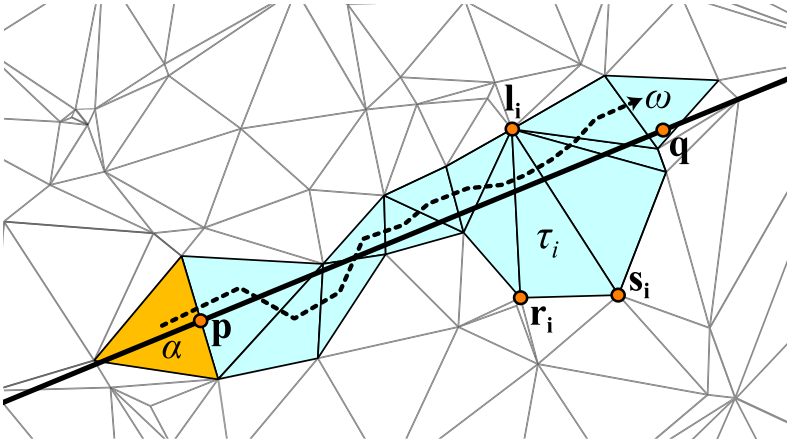

Figure 4. Initialization (triangle α) and walk phase of the straight walk algorithm (dashed line)

otherwise **q** is outside the tested triangle mesh.

4.3. *Border Walk Phase*

The border walk phase follows the border of the triangle mesh always in the same direction (CW direction was chosen for the following text). A fixed direction is necessary for three main reasons. First, it is impossible to identify reliably in advance which direction is shorter since the algorithm works only with local information. Second, the algorithm is simpler and more straightforward than it would be if some heuristic to determinate the appropriate direction was incorporated. Third, if the border walk phase is invoked repeatedly during one location process, the use of the same direction is necessary for proper functionality.

Let us suppose that in the walk phase the search came to a border output edge. Let us denote this border output edge as ϵ'_{rl} and the intersection point of the $\overrightarrow{\mathbf{pq}}$ with ϵ'_{rl} as **g**, see Fig. 5. Then the border walk goes around the border of the triangle mesh until a triangle with a border input edge intersected by the line $\overrightarrow{\mathbf{pq}}$ is met, see ϵ_{rl1} in Fig. 5. If **g** and **q** are on the opposite sides of this border input edge, the continuation triangle has been successfully found and the search may continue by the walk phase. Confirmation, whether **g** and **q** are on the opposite sides of the border input edge ϵ_{rl}, can be done by two orientation tests in which we substitute vertices of the border input edge (**r**, **l**) and consecutively points **g** and **q**.

If **g** and **q** are not on the opposite sides (see ϵ_{rl1} in Fig. 6), then the search continues around the border of the triangle mesh to find another border input edge, see ϵ_{rl2} in Fig. 6. The points **g** and **q** are again checked and if they are on the opposite sides of the border input edge, the border walk phase is interrupted and the search continues by the walk phase.

When the border walk phase is walking along the border of the triangle mesh, it visits alternatively border input edges and border output edges. If the query point is outside the tested triangle mesh, like **q** in Fig. 7, the border walk phase comes back to the triangle

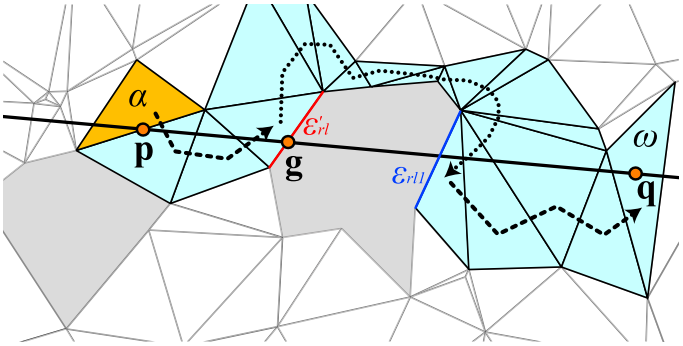

Figure 5. Simple example of proposed walking algorithm on the mesh with holes (walk phase is shown by dashed polyline and border walk phase by doted polyline)

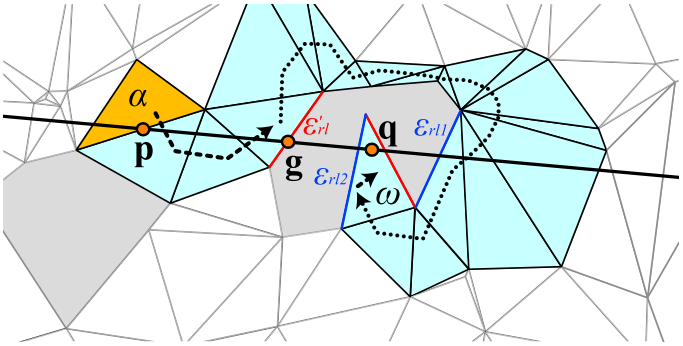

Figure 6. Example of proposed walking algorithm on the mesh with holes (walk phase is shown by dashed polyline and border walk phase by doted polyline)

where it started. This case has to be distinguished from the previous case. It is done by the check whether the border output edge is identical with the border output edge ϵ'_{rl}. If so, it means that the search has inspected all the triangles around the border of the tested triangle mesh and has come to the conclusion that the query point is outside.

Note that orientation tests are performed only for the visited triangles where the border edge is detected (one orientation test is done per each tested border edge and two more orientation tests are performed per each border input edge). It distinctly speeds up the border walk phase due to savings in the orientation tests but, on the other hand, the border walk phase cannot be interrupted elsewhere than in the border triangles.

Let us demonstrate that the algorithm is capable of handling also more complicated situations, such as in Fig. 8. The first run of the border walk phase (see *1st border walk phase* in Fig. 8) demonstrates the case where the first visited border input edge ϵ_{rl1} is before the point $\mathbf{g}$ ($\mathbf{g_1}$ in Fig. 8). In such a case the relevant triangle is not suitable for the continuation of the walk phase and the continuation of the border walk phase is

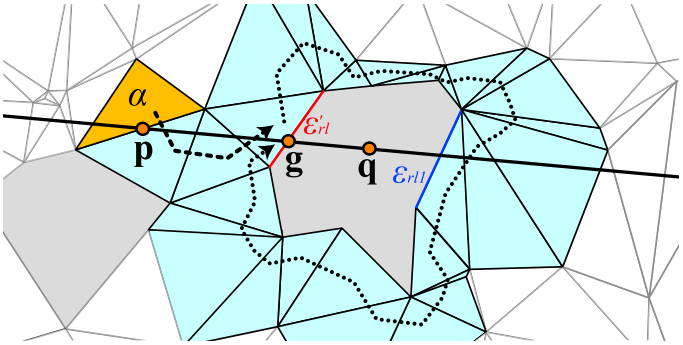

Figure 7. Example of proposed walking algorithm on the mesh with holes, where the query point is outside triangulated area (walk phase is shown by dashed polyline and border walk phase by doted polyline)

necessary. Then the walk phase continues from the border input edge ϵ_{rl2} (see 2^{nd} *walk phase* in Fig. 8). The second run of the border walk phase (see 2^{nd} *border walk phase* in Fig. 8) shows why the relevant point $\mathbf{g}$ ($\mathbf{g_2}$ for 2^{nd} border walk phase in Fig. 8) is useful: it avoids continuation of the walk phase from the border input edge ϵ_{rl4} and the walk phase (see 3^{rd} *walk phase*) correctly continues from the border input edge ϵ_{rl5}.

Fig. 9 shows a complicated shape of the triangulated domain, where almost all the triangles are visited during the border walk phase. First the walk phase walks from $\mathbf{p}$ to $\mathbf{g}$ (the dashed line), then the border walk phase walks around the mesh boundary to the triangle with border input edge ϵ_{lr3} (the dotted line), since other input edges ϵ_{lr1} and ϵ_{lr2} are not between $\mathbf{g}$ and $\mathbf{q}$. Finally, the walk phase walks to the goal triangle ω (the query point $\mathbf{q}$) or the border walk phase continues to the triangle with the border output edge ϵ'_{lr} and decides that the query point is outside the mesh (the query point $\mathbf{q}'$).

For further description and implementation details of the algorithm see pseudo-code in Algorithm 1.

5. Experimental results

For the test purposes, we implemented our algorithm in C++ in two versions: with the double precision floating point arithmetic (standard solution) and also with adaptive floating point arithmetic (Shewchuk 1997) (numerically robust solution) to avoid numerical problems. The solutions were tested on Intel Q6600 2.40GHz in the single thread mode. The SSE2 random generator was used for the choice of the first triangle, since it is declared as up to five times faster then the standard C random generator (Owens and Parikh 2009).

The tests are presented on two types of datasets: data from the cadaster of cities (7 different datasets, where vertices define the boundaries of building sites - see shape illustrations in Figure 10a-g) and CAD data (2 different datasets - see shape illustrations in Figure 10h-i). Thanks to its popularity and triangulation quality, we chose Delaunay triangulation for experimental results with preserved non-convex domain boundaries by the technique (Kolingerová and Žalik 2006). For each dataset, we generated two testing

Input: the query point $\mathbf{q}$, the chosen starting triangle $\alpha \in T$

Output: the triangle which contains $\mathbf{q}$

```
/* initialization phase                                                                          */
triangle τ = α;
point r, l, s;
foreach edge ε ∈ τ do
    r = first vertex of ε;
    l = second vertex of ε;
    if orientation2D(r, l, q) < 0 then break ;
end
/* checks, if τ contains q and returns τ, if so                                                  */
if orientation2D(r, l, q) ≥ 0 then return τ;
point p = point on the edge ε_rl where p ≠ l, p ≠ r;
point p' = p;
/* now pq→ has r on the right and l on the left side                                             */
repeat
    /* walk phase - follows the line segment pq→                                                 */
    if neighbor of τ over ε_rl is not null then
        τ = neighbor of τ over ε_rl;
        s = vertex of τ where s ∉ ε_rl;
        if orientation2D(p, q, s) < 0 then r = s;
        else l = s;
    end
    /* border walk phase - walks around non-triangulated space                                   */
    else
        g = intersection point of ε_rl with pq→;
        edge ε'_rl = ε_rl;
        repeat
            /* looking for border input edge                                                     */
            repeat
                s = r; r = l;
                l = vertex of τ where l ∉ ε_rs;
                while neighbor of τ over ε_rl is not null do
                    τ = neighbor of τ over ε_rl;
                    l = vertex of τ where new l ∉ ε_rl;
                end
            until orientation2D(p, q, l) ≤ 0;
            /* now ε_rl is the border input edge                                                 */
            /* checks, if τ is suitable as a new starting triangle for the walk phase and breaks the border walk phase loop, if
               τ is suitable                                                                     */
            if orientation2D(r, l, g) < 0 and orientation2D(r, l, q) ≥ 0 then break;
            /* looks for the border output edge                                                  */
            repeat
                s = r; r = l;
                l = vertex of τ where l ∉ ε_rs;
                while neighbor of τ over ε_rl is not null do
                    τ = neighbor of τ over ε_rl;
                    l = vertex of τ where new l ∉ ε_rl;
                end
            until orientation2D(p, q, l) ≥ 0;
            /* now ε_rl is the border output edge                                                */
            /* checks, if the border output edge ε_rl is the same as edge where the border walk has started */
        until ε_rl = ε'_lr;
        /* checks, if q is outside the triangle mesh                                             */
        if ε_rl is border output edge then return null;
        else
            /* ε_rl is border input edge                                                         */
            /* initialization of the walk phase                                                  */
            s = vertex of τ where s ∉ ε_rl;
            if orientation2D(p, q, s) < 0 then
                l = r; r = s;
            end
            else
                r = l; l = s;
            end
            /* now pq→ has r on the right and l on the left side                                 */
        end
    end
until orientation2D(r, l, q) ≥ 0;
return τ;
```

Algorithm 1: Straight walk algorithm for point location in a triangulated non-convex domain with holes

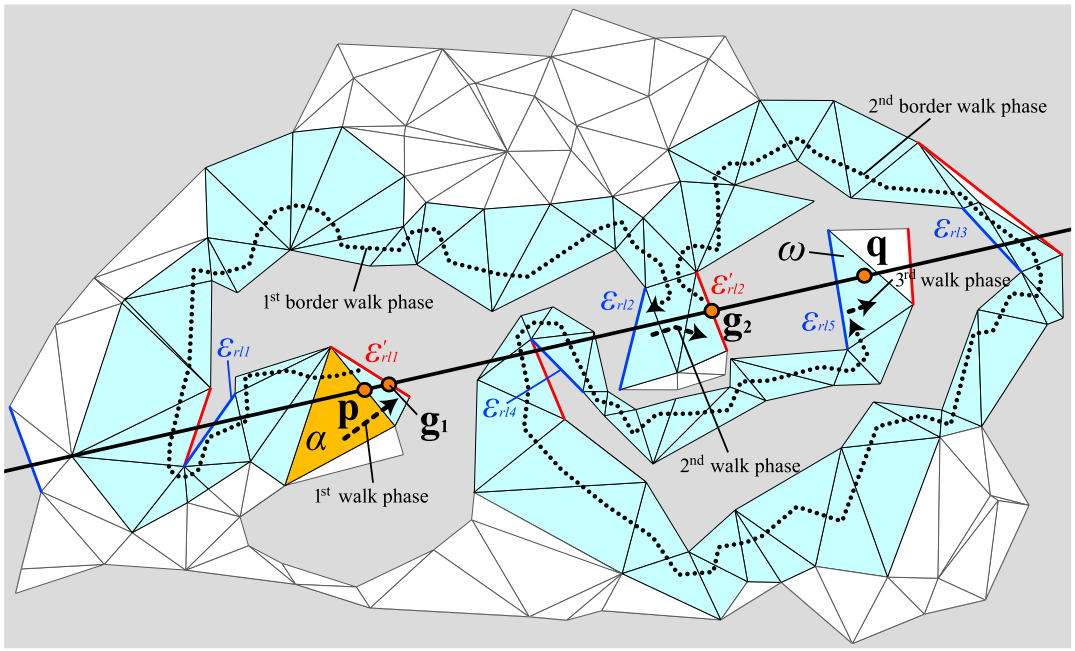

Figure 8. Complex example of the proposed walking algorithm on a triangulated non-convex domain with holes, the walk phase is marked by dashed polyline and the border walk phase is marked by dotted polyline

sets: the set of 10^6 query points distributed randomly inside the tested triangle mesh and the set of 10^6 query points distributed randomly inside the convex hull of the tested triangle mesh. Tests were performed for both sets of query points twice: with the first triangle randomly chosen and using selection of the first triangle without any additional memory proposed by (Mücke et al. 1996), where the first triangle is selected as the closest triangle from a set A of randomly chosen triangles from T, where $\|A\| \ll \|T\|$. We used $\|A\| = \sqrt[3]{n}$ as proposed in (Devroye et al. 1998) for Delaunay triangulations of random points, where n is the number of vertices in the mesh. Only one vertex of each triangle was used for distance computation purposes.

Selected results are in Table 1. The following quantities were examined for each dataset: the size ($\|A\|$) of the set of triangles A from which the nearest triangle to the target point is chosen as the first triangle for the walk (the value 1 is used if the first triangle was chosen randomly); the percentage of the query points located inside the triangle mesh (*Q. pts in* [%]); the average total number of visited triangles during one location process (during the choice of the first triangle, the walk phase and the border walk phase - *Total* ϕ Δ); the average time per one location with double precision floating point arithmetic ($t[\mu s]$) and the average time per one location with numerically robust arithmetic ($t_e[\mu s]$). Moreover, the walk phase and the border walk phase were examined in detail. For each of these phases, we tested the following qualities: the average number of runs (ϕ #); the average number of visited triangles (ϕ Δ); the average number of orientation edge tests (ϕ *tests*) and the maximal number of the visited triangles during the longest of 10^6 location queries (*max* Δ). Table 1 also shows for each dataset its type, the numbers

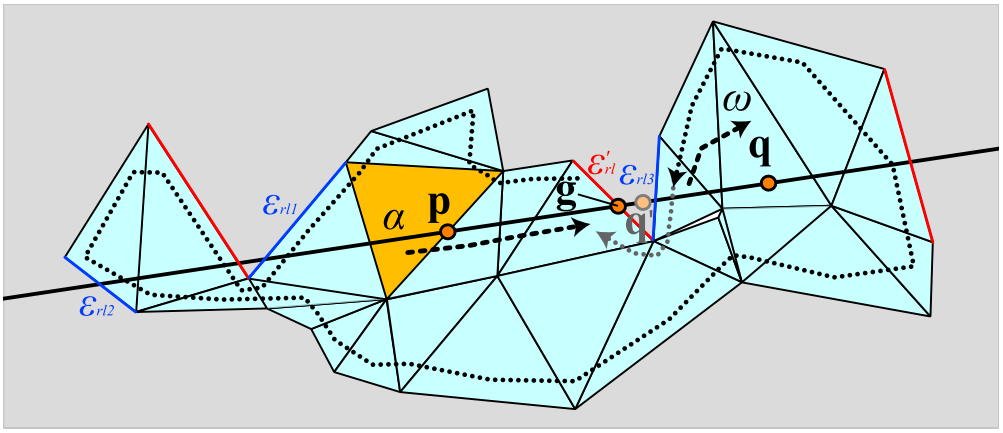

Figure 9. Specific example of the proposed walking algorithm on a triangulated non-convex domain, where almost all triangle are visited during the location, and where $\mathbf{q}$ is inside triangle mesh and $\mathbf{q}'$ is outside triangle mesh, the walk phase is by dashed polyline and the border walk phase is by dotted polyline

of vertices, triangles and border edges, and also the percentage of the convex hull (in Table 1 as CH) area which was triangulated.

As is evident in Table 1, the location process is faster and the path is shorter if we use the choice of the first triangle proposed by (Mücke et al. 1996), thus Fig. 11 and Fig 12 show the dependency of the number of visited triangles on the number of dataset vertices, where the choice of the first triangle by (Mücke et al. 1996) was used. Although the tested data are very diverse, the selected size of $\|A\| = \sqrt[3]{n}$ seems as a good compromise for all the tested data. Fig. 11 shows the number of visited triangles in each phase during the location for the query points randomly distributed in the mesh, Fig. 12 shows the same relation for query points randomly distributed in the convex hull of the mesh.

CAD datasets are characteristic by specific vertices distribution, where vertices are sampled in very high detail near the borders. Therefore, these data sets are very useful to provide more complicated configurations to test our algorithm. The special character of these data sets results in fluctuations in the border walk length (see $\phi\ \Delta$ of the border walk phase in Table 1), thus we have not included corresponding results to Fig. 11 and Fig 12. We can see in Table 1 that both datasets have high percentage of vertices on the border, for example the CAD dataset with 39721 vertices has 11708 from its 39721 vertices on the border, which results in a long walk - especially, if the query point is outside the tested mesh, since the border walk has to walk around all the outer border.

For a non-convex triangulated domain, the number of visited triangles by the proposed algorithm during the walk phase is usually equal or lower than the number of visited triangles with the original straight walk algorithm for the same input in the fully triangulated domain. It is caused by the border walk phase, which sometimes performs a part of the walk, which is normally performed by the straight walk algorithm in the fully triangulated domain. Therefore, the expected complexity of the walk phase is at least equal to the expected complexity of the straight walk algorithm, where $O(\sqrt{n})$ triangles are visited in the expected case and uniform distribution of vertices is expected (Green and Sibson 1978, Mücke et al. 1996).

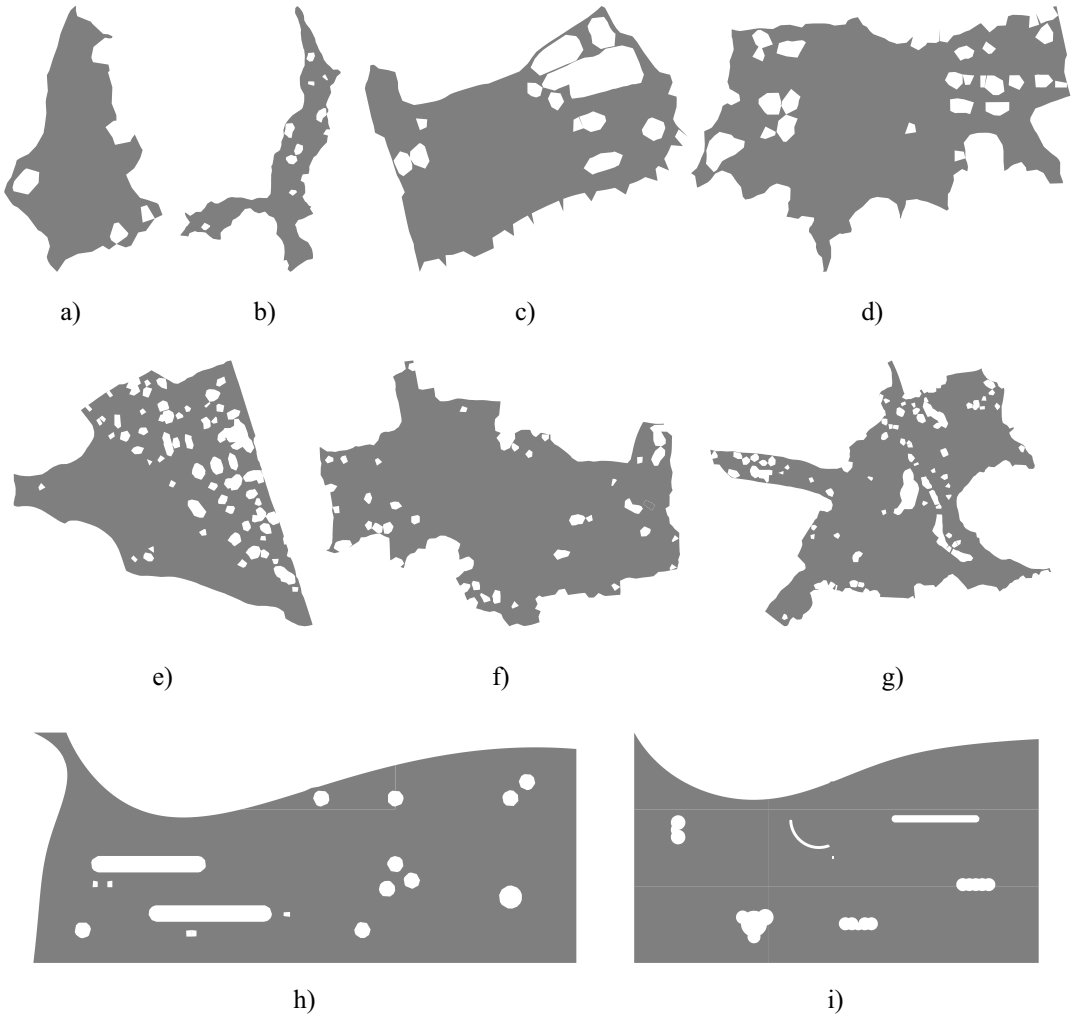

Figure 10. Shape illustrations of used datasets (cadaster data (a-g) and CAD data (h-i)), triangulated area is grey

As we can see in Table 1, Fig. 11 and Fig 12, the number of visited triangles during the border walk phase is dependent not only on the number of vertices, but also on the character of data. Assuming m border edges in a triangle mesh, the border walk phase visits maximally m border triangles, if the query point is inside the triangle mesh, and maximally $2m$ border triangles, if the query point is outside the triangle mesh. A degenerate case for the query point inside the triangle mesh, where almost all the border triangles are visited during the border walk phase, can be seen in Fig. 9 (query point $\mathbf{q}$). A similar example, where almost all the border triangles are visited twice and where the query point is outside the triangle mesh, can be derived from the example in Fig.9 (let us imagine the query point outside the mesh, on the right side from $\mathbf{q}$, lying on $\overrightarrow{\mathbf{pq}}$). The border triangles are visited for the first time, when the triangle for the continuation of the walk phase is searched, and the second time, when the outside position of the query

Query points distribution	$\|A\|$	Q. pts in [%]	Walk phase				Border walk phase				Total	Query time	
			ϕ #	ϕ Δ	ϕ *tests*	*max* Δ	ϕ #	ϕ Δ	ϕ *tests*	*max* Δ	ϕ Δ	$t[\mu s]$	$t_e[\mu s]$
cadaster data, 4897 vertices (9635Δ), 163 border edges, triangulated 75.53% of CH area (see Fig. 10a)													
tested mesh	1	100	1.08	64.9	129.5	197	0.08	13.1	4.1	553	78.0	4.42	6.07
convex hull	1	75.56	1.11	69.1	137.6	227	0.35	142.2	42.9	1022	211.3	7.23	9.42
tested mesh	17	100	1.05	21.6	42.9	179	0.05	11.9	3.7	551	49.5	2.85	3.52
convex hull	17	75.56	1.06	21.1	41.9	158	0.31	121.8	36.4	973	158.9	4.88	5.96
cadaster data, 13829 vertices (27297Δ), 377 border edges, triangulated 36.25% of CH area (see Fig. 10b)													
tested mesh	1	100	1.65	140.8	280.4	494	0.65	67.8	17.0	1471	208.6	10.62	14.77
convex hull	1	36.26	1.52	90.8	180.5	450	1.16	1010.7	229.2	3123	1101.5	26.61	31.85
tested mesh	24	100	1.10	32.2	63.9	411	0.10	21.4	5.1	1406	76.5	4.31	5.36
convex hull	24	36.26	1.08	25.8	51.3	313	0.72	862.1	194.1	2788	910.9	21.92	24.74
cadaster data, 15819 vertices (31267Δ), 391 border edges, triangulated 65.16% of CH area (see Fig. 10c)													
tested mesh	1	100	1.32	109.2	217.8	382	0.32	21.8	7.0	850	131.0	7.90	11.12
convex hull	1	65.16	1.63	89.4	178.0	383	0.98	209.1	62.6	1615	298.5	11.12	13.78
tested mesh	25	100	1.13	31.5	62.5	254	0.13	7.7	2.6	876	63.1	4.07	5.06
convex hull	25	65.16	1.15	28.3	56.3	191	0.50	185.7	54.4	1624	238.0	7.76	9.57
cadaster data, 19993 vertices (39554Δ), 472 border edges, triangulated 65.19% of CH area (see Fig. 10d)													
tested mesh	1	100	1.60	128.3	255.8	487	0.60	82.0	26.1	1138	210.3	10.78	14.54
convex hull	1	65.19	1.66	108.8	216.6	472	1.00	319.7	99.3	2051	428.4	15.44	19.85
tested mesh	27	100	1.17	32.7	64.9	216	0.17	18.9	6.1	1045	77.6	4.55	5.67
convex hull	27	65.19	1.17	32.4	64.3	219	0.52	304.2	93.1	2023	362.6	11.50	14.02
cadaster data, 41851 vertices (82950Δ), 862 border edges, triangulated 74.91% of CH area (see Fig. 10e)													
tested mesh	1	100	1.84	192.1	383.0	796	0.84	39.4	13.0	1575	231.5	14.94	19.50
convex hull	1	74.88	1.98	180.8	360.2	923	1.23	263.9	77.2	2885	444.7	20.48	26.41
tested mesh	35	100	1.31	40.4	80.1	320	0.31	23.0	7.2	1503	97.4	6.01	7.15
convex hull	35	74.88	1.30	39.6	78.5	325	0.55	235.2	67.5	2811	308.7	12.01	13.42
cadaster data, 60234 vertices (119824Δ), 702 border edges, triangulated 68.71% of CH area (see Fig. 10f)													
tested mesh	1	100	1.36	198.9	397.0	641	0.36	59.4	19.7	1856	258.3	16.04	21.29
convex hull	1	68.74	1.40	210.3	419.8	852	0.71	568.0	184.3	3570	778.3	32.70	41.48
tested mesh	39	100	1.10	46.4	92.3	318	0.10	24.3	8.1	1721	108.6	6.92	8.25
convex hull	39	68.74	1.13	44.8	89.1	297	0.44	523.3	169.4	3372	606.1	21.25	25.25
cadaster data, 70428 vertices (139706Δ), 1270 border edges, triangulated 50.53% of CH area (see Fig. 10g)													
tested mesh	1	100	2.21	203.3	405.1	879	1.21	203.5	65.4	2600	406.8	20.94	27.31
convex hull	1	50.47	2.02	228.6	455.7	764	1.52	1248.5	389.0	4867	1477.1	53.57	65.14
tested mesh	41	100	1.25	49.5	98.5	443	0.25	48.7	15.6	2413	138.2	8.17	9.47
convex hull	41	50.47	1.25	44.9	89.2	443	0.74	1072.0	332.5	4723	1156.9	38.43	45.71
CAD data, 39721 vertices (67766Δ), 11708 border edges, triangulated 75.82% of CH area (see Fig. 10h)													
tested mesh	1	100	1.66	111.5	222.2	2021	0.66	482.2	309.4	18027	593.7	24.91	33.21
convex hull	1	75.84	1.49	141.0	281.2	2007	0.74	3211.1	1614.1	25577	3352.1	111.66	136.34
tested mesh	34	100	1.17	41.0	81.6	1114	0.17	111.8	76.6	12432	185.8	9.7	11.4
convex hull	34	75.84	1.14	35.5	70.6	1074	0.38	2598.1	1228.2	24862	2666.6	88.17	102.1
CAD data, 78090 vertices (143289Δ), 12903 border edges, triangulated 82.59% of CH area (see Fig. 10i)													
tested mesh	1	100	1.65	199.5	398.1	1585	0.65	477.9	295.1	16657	677.4	33.36	43.84
convex hull	1	82.64	1.32	182.0	363.4	1350	0.49	2396.0	1391.9	28785	2578.0	106.48	131.14
tested mesh	43	100	1.08	46.4	92.4	824	0.08	78.5	47.8	14402	166.9	9.83	11.43
convex hull	43	82.64	1.07	42.6	84.8	1099	0.25	2168.9	1233.1	28705	2253.4	86.35	104.4

Table 1. The comparison of the algorithm parameters on different datasets; *Query points distribution* shows whether targets are randomly generated in the mesh or in the convex hull of the mesh, $\|A\|$ shows the number of triangles from which the suitable first triangle is selected, *Q. in* [%] shows percentage of query points inside the mesh, *Total* ϕ Δ shows the average total number of visited triangles during the one location process, $t[\mu s]$ and $t_e[\mu s]$ shows the average time per one location query with ($t_e[\mu s]$) and without numerically robust tests ($t[\mu s]$) and ϕ #, ϕ Δ, ϕ *tests* and *max* Δ shows parameters of location phases, where ϕ # is average number of phase executions during one location, ϕ Δ is average total number of visited triangles by the phase during the one location, ϕ Δ is average total number of orientation tests performed by the phase during the one location and *max* Δ is maximal number of visited triangles by the phase during the longest of 10^6 locations

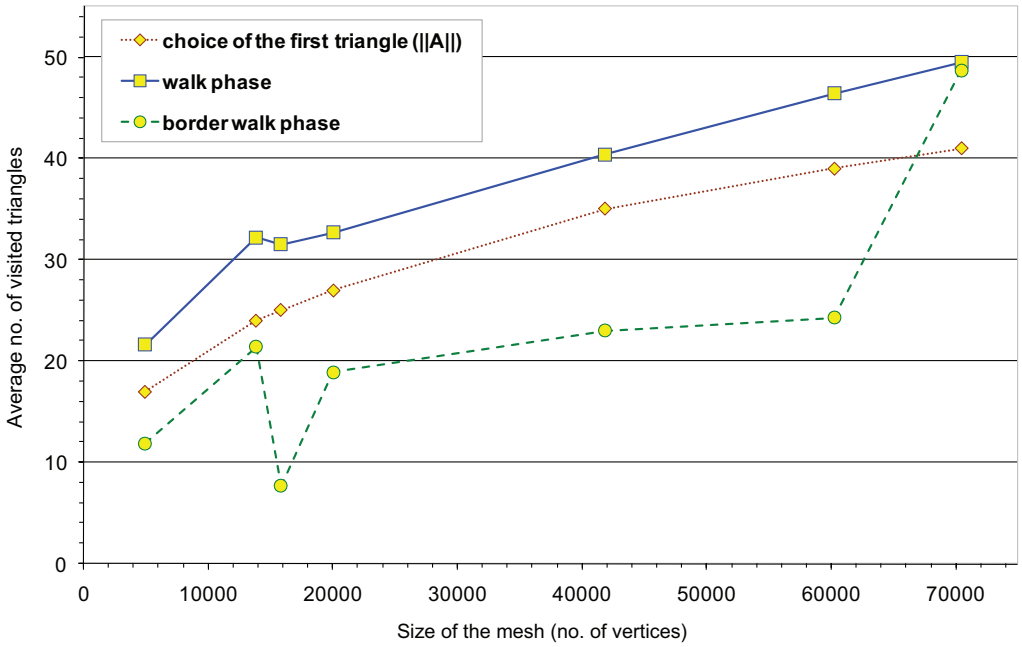

Figure 11. Dependency of average number of visited triangles in each phase (y-axis) on the number of vertices in the mesh (x-axis) during one point location for query points randomly distributed in the mesh (cadaster data)

point is verified.

Note that the average number of the visited border triangles during the border walk phase does not correspond to the average number of visited triangles (ϕ Δ) during the same phase, which is higher. But as follows from the algorithm, the average number of visited border edges during the border walk phase corresponds to the average number of orientation edge tests (see ϕ *tests* in Table 1) during the same phase, since one orientation test is performed for each visited border edge. This number is very similar to the average number of visited border triangles, but not necessarily equal, since one border triangle may contain two border edges in its set of edges. According to the tests (see Table 1), the border walk phase visits in average from 1.5 times up to 5 times more triangles (including border triangles) than border triangles. It can be assumed that the border walk phase visits $O(m)$ triangles in the worst case, but we are aware that singular cases may exist where $m \approx n$. However, we do not suppose that it is a common case for an ordinary terrain model data.

Let us see ϕ Δ of the border walk phase, cadaster data and query points distributed inside the mesh in Table 1 and Fig. 11. As we can see, ϕ Δ of the border walk phase is usually lower or comparable to ϕ Δ of the walk phase. Therefore, we assume the same expected complexity for the border walk phase as is for the straight walk ($O(\sqrt{n})$ visited triangles), if the query points are inside the mesh, the mesh shape is not complicated and the mesh has uniformly distributed vertices, since the border walk phase is performed only if the border is crossed during the walk phase. For more complicated shapes, the

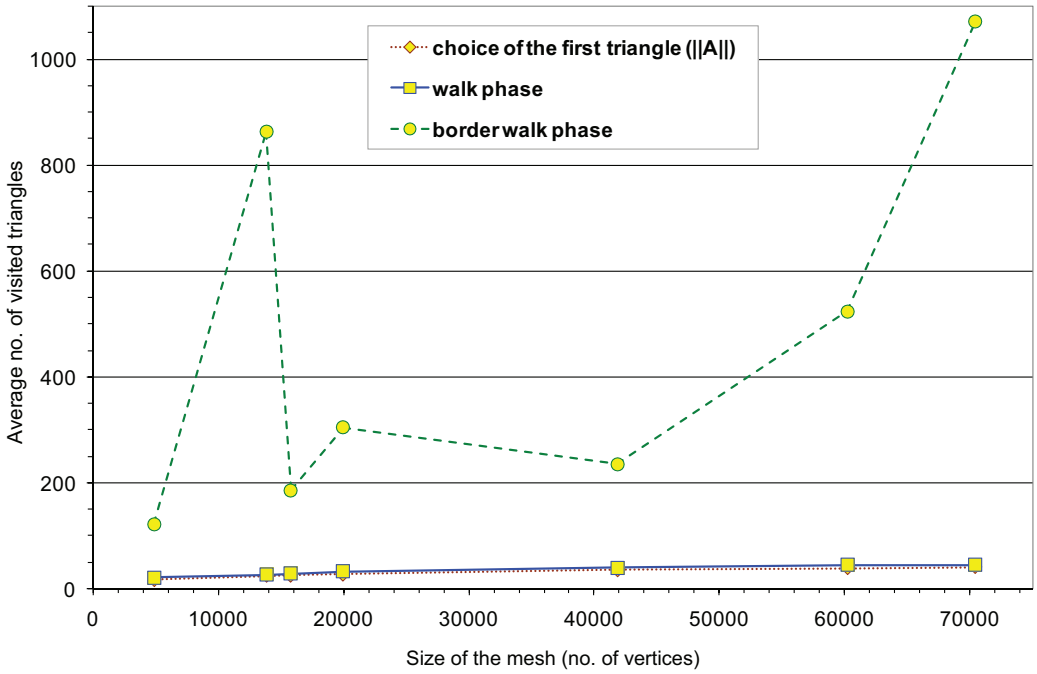

Figure 12. Dependency of average number of visited triangles in each phase (y-axis) on the number of vertices in the mesh (x-axis) during one point location for query points randomly distributed in the convex hull of the mesh (cadaster data)

expected complexity may tend to $O(m)$. We can see some fluctuations also in our cadaster data (see Fig. 11), for example in the datasets with 13829 or 70428 vertices, where the fluctuations are caused by a higher 'level of non-convexity'.

Expected complexity of the border walk phase for general query points is $O(m)$, and appropriate ϕ Δ depends not only on the shape of the triangle mesh, but especially on the percentage of query points lying outside the tested mesh (see the percentage of query points inside the mesh in Table 1 and the relevant fluctuations in Fig. 12).

Note that the average number of triangles visited by each phase cannot be compared directly, since two orientation tests are done for each triangle during the walk phase, but much less than one orientation test is performed for each visited triangle during the border walk phase (since the orientation tests are performed only for the border triangles). Therefore the border walk phase is faster than the walk phase with the same number of visited triangles.

Let us sum up this reasoning and our tests to compare average times per point query. Naturally, location of the query points lying inside the mesh is distinctly faster than for the general query points lying inside the convex hull, but we have shown that our algorithm can be successfully utilized also for the general query points, which may lay outside the mesh. We have not mentioned the time differences between the double precision floating point arithmetic solution and the robust adaptive floating point solution. As can be seen from Table 1, they are not significant, therefore, we recommend the use of the numerically robust version with the adaptive floating point arithmetic.

Apart from the datasets above, the algorithm was tested also on numerous generated non-convex datasets with different 'levels of non-convexity'. During all the tests, we did not register any cases of instability or infinite looping. Algorithm works robustly and reliably and may be used for any continuous non-convex triangulated domain. The accuracy of the results from our algorithm (the numerically robust version) was verified and verification confirmed the correctness of the results.

6. Conclusion

We presented a new algorithm for the point location in a triangulated domain with holes, based on the walking principle. As far as we know, it is the only walking algorithm for non-convex shapes of the triangulated domain. The properties of the algorithm correspond to the class of walking algorithms: no extra location data structures with the exception of the neighborhood relations between triangles (which are usually needed for other purposes, anyway), suboptimal (but sublinear in average) time complexity.

The algorithm was tested and verified on real cadaster data, on CAD datasets and on artificially generated examples. The tests showed a strong dependence on the complexity of shape of the boundary which is unavoidable for the walking principle but behavior was reasonable namely for the cadaster data.

The algorithm could be used also for the related problem to find whether a given point lies inside a triangulated polygon domain. However, as we expect the main use in the point location problem and as many efficient algorithms exist for the point-inside-polygon problem, we did not research into this direction in detail.

Acknowledgements

The authors would like to thank their colleagues Oldřich Petřík, Václav Purchart and Jan Rus for their feedback and inspiring discussions. This work has been supported by the UWB grant SGS-2013-029 – Advanced Computer and Information Systems, by Ministry of Education, Youth, and Sport of Czech Republic – University Spec. Research – 1311, by the Ministry of Education of the Czech Republic – project Kontakt II LH11006, and by the European Regional Development Fund (ERDF) – project NTIS (New Technologies for Information Society), European Centre of Excellence, CZ.1.05/1.1.00/0.2.0090.

References

Amenta, Nina, Sunghee Choi, and Günter Rote (2003), Incremental constructions con brio. In *SCG '03: Proceedings of the 19th annual symposium on Computational geometry*, 211–219, ACM, New York, NY, USA.

Antonarakis, S, A, S Richards, K, and J Brasington (2008), Object-based land cover classification using airborne lidar. *Remote Sensing of Environment*, 112, 2988 – 2998.

Boissonnat, Jean-Daniel and Monique Teillaud (1986), The hierarchical representation of objects: the Delaunay tree. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, 260–268, ACM, New York, NY, USA.

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

Buchin, Kevin (2005), Incremental construction along space-filling curves. In *EuroCG'05: Proceedings of the 21th European Workshop on Computational Geometry*, 17–20.

Dæhlen, M, M Fimland, and Ø Hjelle (2001), *A Triangle-based Carrier for Geographical Data*, 105–120. Taylor and Francis Books Ltd.

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

Devillers, Olivier (2002), The Delaunay hierarchy. *International Journal of Foundations of Computer Science*, 13, 163–180.

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

Floriani, Leila De, Bianca Falcidieno, George Nagy, and Caterina Pienovi (1991), On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6, 522–532.

Green, P. J. and R. Sibson (1978), Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21, 168–173.

Höhle, Joachim, Christian Oster Pedersen, Tomas Bayer, and Poul Frederiksen (2010), The photogrammetric derivation of digital terrain models in built-up areas. *Photogrammetric Journal of Finland*, 22, 33–45.

Koch, A. (2005), An integrated semantically correct 2.5 dimensional object oriented TIN. In *Proceedings of the 1st international Workshop on Next Generation 3D City Models*.

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

Kolingerová, I. and B. Žalik (2006), Reconstructing domain boundaries within a given set of points, using Delaunay triangulation. *Computers & Geosciences*, 32, 1310–1319.

Kolingerová, Ivana, Matyáš Dolák, and Václav Strych (2009), Eliminating contour line artefacts by using constrained edges. *Computers & Graphics*, 35, 1975–1987.

Lawson, C. L. (1977), *Mathematical Software III; Software for C1 Surface Interpolation*, 161–194. Academic Press, New York.

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

Mücke, E. P., I. Saias, and B. Zhu (1996), 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, 274–283.

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

Owens, Kipp and Rajiv Parikh (2009), Fast random number generator on the intel pentium 4 processor. Intel Software Network.

Purchart, Václav, Ivana Kolingerová, and Bedřich Beneš (2012), Interactive sand-covered terrain surface model with haptic feedback. In *GIS Ostrava 2012 - Surface models for geosciences*, 215–223.

Schilling, Arne, Jens Basanow, and Alexander Zipf (2007), Vector based mapping of polygons on irregular terrain meshes for web 3d map services. In *3rd International Conference on Web Information Systems and Technologies (WEBIST), Barcelona*.

Schilling, Arne, Sandra Lanig, Pascal Neis, and Alexander Zipf (2009), Integrating terrain surface and street network for 3d routing. In *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, 109–126, Springer Berlin Heidelberg.

Shewchuk, Jonathan Richard (2002), What is a good linear element? - interpolation, conditioning, and quality measures. In *In 11th International Meshing Roundtable*, 115–126.

Shewchuk, Richard, Jonathan (1997), Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Computational Geometry*, 18, 305–363.

Sloan, S. W. (1987), A fast algorithm for constructing Delaunay triangulations in the plane. *Advanced Engineering Software*, 9, 34–55.

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

Soukal, Roman and I. Kolingerová (2010), Star-shaped polyhedron point location with orthogonal walk algorithm. *Procedia Computer Science*, 1, 219–228.

Soukal, Roman, Martina Málková, and Ivana Kolingerová (2012a), A new visibility walk algorithm for point location in planar triangulation. In *Advances in Visual Computing*, volume 7432 of *Lecture Notes in Computer Science*, 736–745, Springer Berlin Heidelberg.

Soukal, Roman, Martina Málková, and Ivana Kolingerová (2012b), Walking algorithms for point location in tin models. *Computational Geosciences*, 16, 853–869.

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

Sundareswara, Rashmi and Paul Schrater (2003), Extensible point location algorithm. In *International Conference on Geometric Modeling and Graphics*, 84–89.

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

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

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

Zhu, Binhai (2003), On Lawsons oriented walk in random delaunay triangulations. In *Fundamentals of Computation Theory* (Andrzej Lingas and Bengt Nilsson, eds.), volume 2751 of *Lecture Notes in Computer Science*, 222–233, Springer Berlin / Heidelberg.

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

Appendix E

Walking algorithms for point location in TIN models

Soukal, R., Málková, M., Kolingerová, I.

Computational Geosciences, Volume 16, Issue 4, pp. 853–869, Springer Verlag (2012), ISSN 1420–0597, IF 1.612 (2013)

Comput Geosci (2012) 16:853–869
DOI 10.1007/s10596-012-9305-3

REVIEW PAPER

Walking algorithms for point location in TIN models

Roman Soukal · Martina Málková · Ivana Kolingerová

Received: 10 April 2012 / Accepted: 25 June 2012 / Published online: 28 July 2012

Abstract Finding which triangle in a planar or 2.5D triangle mesh contains a query point (so-called point location problem) is a frequent task in geosciences, especially when working with triangulated irregular network models. Usually, a large number of point locations has to be performed, and so there is a need for fast algorithms having minimal additional memory requirements and resistant to changes in the triangulation. So-called walking algorithms offer low complexity, easy implementation, and negligible additional memory requirements, which makes them suitable for such applications. In this article, we focus on these algorithms, summarize, and compare them with regard to their use in geosciences. Since such a summary has not been done yet, our article should serve those who are dealing with this problem in their application to decide which algorithm would be the best for their solution. Moreover, the influence of the triangulation type on the number of the visited triangles is discussed.

Keywords Point searching · Searching algorithms · Planar triangulation · TIN models · Terrain models

R. Soukal (✉) · M. Málková · I. Kolingerová
Faculty of Applied Sciences, Department of Computer Science and Engineering, University of West Bohemia, Univerzitní 22, 306 14 Pilsen, Czech Republic
e-mail: soukal@kiv.zcu.cz

I. Kolingerová
e-mail: kolinger@kiv.zcu.cz

1 Introduction

In this text, we focus on the planar point location problem in triangle meshes which means that for a given planar triangle mesh and a query point, a triangle geometrically containing the query point has to be found.

Point location finds applications in a variety of areas, including computer graphics, geographic information systems, computer aided design, etc. In geosciences, its main use lies in applications working with a triangulated irregular network (TIN)-based digital terrain model (DTM), since TIN meshes are one of the most frequent representations of terrain models and since point location in TIN meshes is a very important task in construction, manipulation, or analysis of these models, e.g., inserting new objects into an existing DTM [5, 14, 23, 24], verification of accuracy of the DTM model [13], or interactive dynamic erosion surface modeling [22].

Naturally, algorithms presented here can be used for 2.5D terrain models without any preprocessing only by omitting the height information during the location. All the algorithms expect the mesh to be convex shaped and without holes. However, the data used in geosciences are usually triangulated point clouds fulfilling this expectation. Other types of data should be triangulated first.

In many applications, a large number of point locations have to be performed, and so there is a need of fast algorithms. For a fast search, the location algorithms use additional data structures. Optimal computational complexity $O(\log n)$ is achieved at the price of additional memory demands and more complicated manipulation, especially in the case when the triangle mesh is frequently changed. Therefore, the so-called walking algorithms are often used, thanks to their

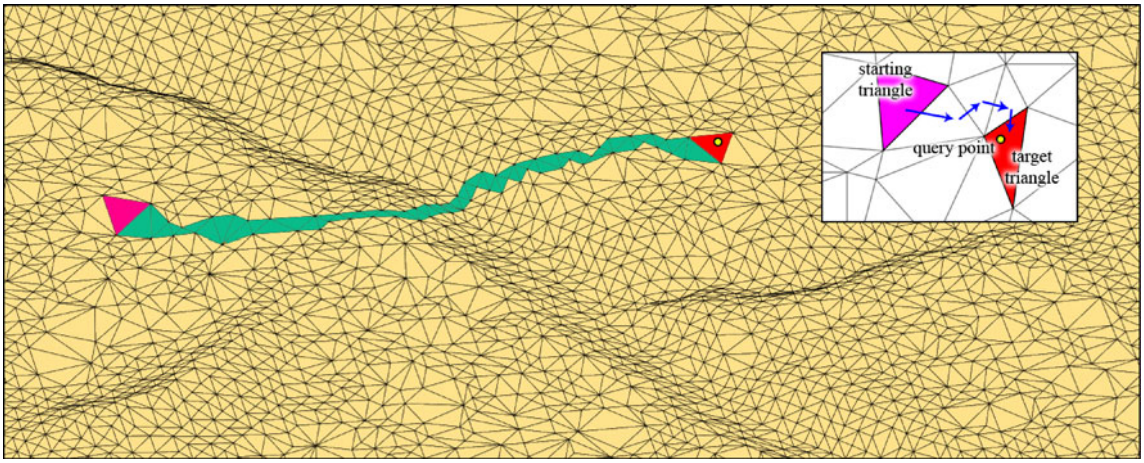

Fig. 1 Example of a point location with a walking algorithm on a part of a TIN model

easy implementation and still expected sublinear time complexity.

The name of walking algorithms has arisen from their operating principle. They use the triangle neighborhood relations to navigate through the triangle mesh and locate the triangle containing the query point (see Fig. 1). The starting triangle for such a walk may be arbitrary; however, its clever selection may radically shorten the length of the walk. Walking algorithms do not need any additional data structures, only the neighborhood relations, which are usually utilized in the application for other purposes.

There are several walking algorithms published, each offers different advantages, and the appropriate algorithm should be chosen according to the particular application. However, these algorithms have never been summarized, tested, and compared to provide a tool for such a selection. The aim of this article is to do so, especially for use in geosciences. We describe all substantial walking algorithms in detail to provide all information needed for their implementation, discuss their behavior, and test them on different datasets including random data, data from a cadastre, and light detection and ranging (LIDAR) data.

Section 2 describes the notation used throughout the text. The mathematic background for the algorithms is presented in Section 3. Section 4 covers the task of a clever selection of the starting triangle and provides an overview of the walking algorithms. Sections 5, 6, and 7 describe the types of walking algorithms along with their extensions: visibility walks, straight walks, and orthogonal walks. Section 8 presents our experiments performed on random data and real geodetic datasets. The influence of the triangulation type on the number of visited triangles is discussed, and performance tests of all the presented algorithms are proposed, followed by a summarizing comparison based on both measurable properties of the algorithms and our own experience. Appendix contains pseudocodes of the presented walking algorithms.

2 Basic notation

Vectors, points, and vertices are denoted by bold lowercase characters (e.g., $\mathbf{a}$, $\mathbf{b}$), and a query point is usually denoted as $\mathbf{q}$. Scalar variables are denoted by lowercase characters in italic (e.g., k, l). For a line segment between two points (e.g., points $\mathbf{a}, \mathbf{b}$ where $\mathbf{a} \neq \mathbf{b}$), we use $\overrightarrow{\mathbf{ab}}$.

The letter λ is used to denote a line, which will be always considered as oriented; subscripts are used for lines having a relation to λ (e.g., λ_n is a line orthogonal to λ). An edge of a triangle is denoted by letter ϵ, and subscripts are used to specify vertices that belong to this edge. Other lowercase Greek letters are used for triangles.

A letter T is used for the triangle mesh in which we want to locate the query point $\mathbf{q}$. The starting triangle is always marked as α ($\alpha \in T$), the triangle which contains $\mathbf{q}$ is denoted as ω ($\omega \in T$), and the ith triangle visited by the walking algorithm is denoted as τ_i. Capital letters X and Y are usually used for random variables in statistics.

3 Background mathematics

This section presents basic geometric tasks appearing in the walking algorithms. To determine the position of a

point $\mathbf{v}$ with respect to an oriented edge (or oriented line) $\overrightarrow{\mathbf{tu}}$, we use the sign of the determinant in a so-called 2D orientation test [7]:

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

where a positive value is returned for $\mathbf{v}$ on the left of $\overrightarrow{\mathbf{tu}}$ and negative for $\mathbf{v}$ on the right of $\overrightarrow{\mathbf{tu}}$.

Sometimes, it is more suitable to use the implicit line equation of the oriented line instead of the equation above. To determine the position of a point $\mathbf{v}$ with respect to an oriented line $\overrightarrow{\mathbf{tu}}$, we compute its implicit equation—Eqs. 2 and 3. The position of $\mathbf{v}$ is given by the sign of Eq. 4 as in the orientation test.

$$\lambda : a \cdot x + b \cdot y + c = 0 \quad (2)$$

$$(a, b, c) = (t_x, t_y, 1) \times (u_x, u_y, 1) \quad (3)$$

$$\text{position}(\lambda, \mathbf{v}) = a \cdot v_x + b \cdot v_y + c \quad (4)$$

Algorithms using the 2D orientation tests during the location need the triangulation to have all its triangles' vertices oriented in the same way—either all clockwise or all counterclockwise (CCW). In the presented pseudocodes, we assume CCW order of the vertices. Figure 2 shows the resulting signs of orientation tests from Eq. 1 for a triangle $\tau_{\mathbf{t_0 t_1 t_2}}$ with CCW order of vertices.

Some walking algorithms do not use the tests above but use the so-called barycentric coordinates to navigate through the mesh. Barycentric coordinates b_0, b_1, b_2 describe the position of a point $\mathbf{v}$ with respect to a triangle $\tau_{t_0 t_1 t_2}$. The point $\mathbf{v}$ is an affine combination of $\mathbf{t_0}$, $\mathbf{t_1}$, and $\mathbf{t_2}$:

$$\mathbf{v} = b_0 \cdot \mathbf{t_0} + b_1 \cdot \mathbf{t_1} + b_2 \cdot \mathbf{t_2} \quad (5)$$

where $b_i \in R$, $i = 0, 1, 2$, and $b_0 + b_1 + b_2 = 1$.

$b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v})$ can by computed using the 2D orientation tests:

$$b_i = \frac{\text{orientation2D}(\mathbf{t}_{[(i+1) \mod 3]}, \mathbf{t}_{[(i+2) \mod 3]}, \mathbf{v})}{\text{orientation2D}(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})}. \quad (6)$$

We can use the 2D orientation tests to compute the barycentric coordinates, since such a test computes the double of the area of a triangle formed by the three tested vertices, signed according to the orientation of the triangle. Each barycentric component b_i corresponds to one vertex of the examined triangle and expresses the area of a triangle given by the edge opposite to this vertex and the point $\mathbf{v}$, divided by the area of the examined triangle. A positive value of the component means that the point is located on the same side as the third vertex of the triangle, a zero value means that the point lies on this edge, and a negative value describes a point lying on the opposite side of this edge than the third vertex of the triangle. Therefore, we can easily decide if the point is inside the triangle by checking whether all the barycentric components are nonnegative. Figure 3 shows an example of barycentric coordinate components of a vertex inside and outside the tested triangle.

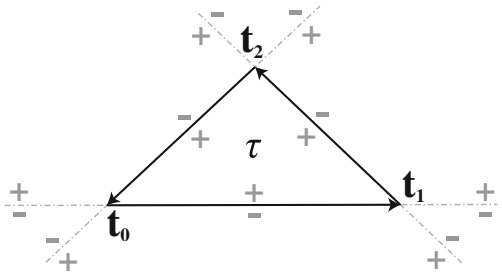

Fig. 2 Example of orientation tests for triangle edges

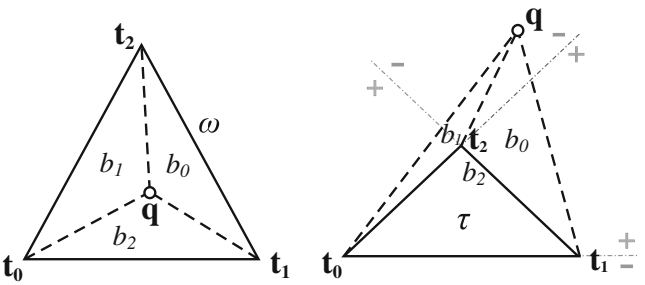

Fig. 3 Barycentric coordinates of $\mathbf{q}$ inside a triangle ω, $\mathbf{b}(\omega) = (0.25, 0.35, 0.4)$ and outside a triangle τ, $\mathbf{b}(\tau) = (-0.75, -0.25, 2)$

4 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.

4.1 Selection of the relevant initial triangle

Clever selection of the initial triangle may radically improve the speed of the process. If we know any additional information about the data, we can use it in our selection and speed up the process without any additional memory or time requirements. Such information can be the knowledge of the range of the mesh vertices coordinates, in which case we start all the locations in a triangle containing the point lying in the middle of the range, used for instance by [28]. In some applications (e.g., construction of Delaunay triangulation (DT) by incremental insertion), we know the located points at the beginning; thus, we can sort the points in such an order that the next query point will be close to the last one. Then, using the target triangle from the last location as the initial one for the next location provides a significant speedup [1, 4, 26, 36]. Sometimes, the located points are ordered properly without additional sorting (i.e., [22]).

Without any knowledge of the data, we may simply choose the initial triangle randomly. A better yet still fast and simple alternative without any additional memory use was proposed by [18], who select the initial triangle as the closest triangle from a set A of randomly chosen triangles from T, where $\|A\| \ll \|T\|$. An analysis of the ideal size of such a random subset of a Delaunay triangulation has been proposed by [8], leading to a size of $O(\sqrt[3]{n})$.

More efficient solutions lead to some additional memory consumption. Mulmuley [19] proposed a method simplifying the triangulation and locating the point in the simplified version first. From the triangulation T with n vertices, only $m = k \cdot n$ vertices (where $k \in (0, 1)$) are randomly selected and triangulated and so a higher layer for the location is created. The number of triangles is much smaller in the new layer, and it radically improves the speed of a walk in it. If m is still bigger than a chosen size, other layers are computed in the same way. The point location then runs in several steps. First, the triangle containing the query point is found on the highest layer. The closest vertex of this triangle defines the starting point for the walk in the lower layer, until the triangle in the lowest layer is found. In each layer, the walk is short and therefore fast. Devillers [6] analyzes this algorithm, precises the time complexity to $O(\log n)$ for any input, and proposes an optimal value of $k = 0.025$ which is valid for random input and leading to the best rate between speed and memory use.

Su and Drysdale [29] introduce a bucketing method, which uses a uniform grid to quickly find a proper initial triangle. Each cell of the grid (a bucket) contains at most one vertex of the triangulation. During the point location, the cell containing the query point is found. If it contains also a triangulation vertex, the walk starts from this vertex (from any triangle formed by this vertex); if it does not, the nearest cell containing a vertex is found using a spiral search. Because the bigger number of empty cells causes longer spiral search, the algorithm is useful mainly for triangle meshes with uniformly distributed vertices where most cells contain a vertex.

Some algorithms (e.g., [34, 38]) try to avoid the sensitivity of the original bucketing method on data uniformity by using adaptive structures instead of a uniform grid. However, on highly nonuniform data, the dynamic hierarchy algorithm mentioned above [6, 19] still provides better results with lower additional memory.

4.2 Walking algorithms

Given an initial triangle, the walk may proceed. There exist several algorithms solving this step, and according to the style on how they determine the way of the walk, they can be divided into three groups: visibility, straight, and orthogonal walks.

Visibility algorithms perform local tests in each triangle they walk through. These tests look for such an edge that defines a line separating the query point from the third vertex of the triangle. The walk then moves across this edge to the neighborhood triangle. For convex triangulations, they never cross the border of the triangulation; however, deterministic versions of visibility walk algorithms may loop for non-Delaunay triangulations. Randomized (stochastic) versions are slower because a randomization step is done in each triangle. The stochastic walk has been shown in [37] to need $O(\sqrt{n \cdot \log n})$ expected time for uniform data.

Straight walk algorithms use not only the local comparisons to determine the way of the walk but also use a line connecting one point of the starting triangle with the query point and traverse triangles crossed by this line. This way, their walk is short and does not loop. For convex triangulations, they never cross the border of the triangulation and they do not loop. The straight walk has been proved to visit $O(\sqrt{n})$ triangles in the expected case and uniform distribution of vertices [11, 18].

Orthogonal walks first navigate along one coordinate axis and then along the other, which makes the local tests cheaper, since only coordinate components are compared during the walk. However, more triangles are visited during the walk. The border of a trian-

gulation may be crossed during the walk, in which case a special modification is needed, resulting in a slower location process and additional implementation effort. Bounds based on [3] show that the orthogonal walk has similar complexity as the straight walk [8, 18].

5 Visibility walk algorithms

The name of this group of algorithms comes from the fact that they use local "visibility" tests to determine the way of their walk. Let us recall that these tests look for such an edge that defines a line separating the query point from the third vertex of the triangle. The walk then moves across this edge to the neighborhood triangle. The algorithms differ in the particular way of testing this visibility; the following subsections describe each algorithm in detail.

5.1 Lawson's oriented walk

This fundamental visibility walk algorithm was first published by [16]. The algorithm uses the 2D orientation test (Eq. 1) to move until it reaches the target point **q**. In each triangle τ, the algorithm tests the triangle edges until it finds such an edge ϵ_{ab}, where the third vertex of the triangle, denoted as **c**, lies on the opposite side of the edge than **q** (in our case, this means that the test produces a negative sign, see Fig. 2). Then, it crosses such an edge to the next triangle. If such an edge does not exist, the triangle containing **q** has been found. The pseudocode of the Lawson's oriented walk can be found in Algorithm 1.

The simple Lawson's oriented walk algorithm tests edges of τ in a deterministic order, which depends on the order of edges in triangles, generated during the construction of the triangulation. This leads to the fact that the walk may loop in some specific configurations of the triangle mesh [7, 33], see Fig. 4. It has been shown that for a planar Delaunay triangulation, the Lawson's oriented walk cannot loop [10, 33]. However, on a constrained Delaunay triangulation and other triangulation types, the walk may loop, and so the following extension should be used.

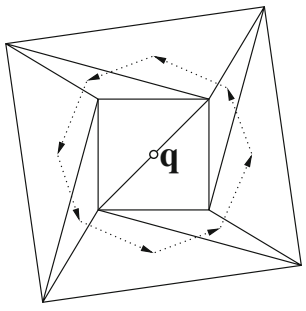

Fig. 4 Loop of Lawson's oriented walk [33]

5.2 Remembering stochastic walk

To prevent from the loop of the Lawson's oriented walk in a non-Delaunay triangulation, [7] proposed an algorithm that chooses 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 with the previous triangle, the process was speed up by remembering this edge and skipping its test. This improved stochastic walk is called *remembering* and brings a significant speedup, since only one or two orientation tests are needed instead of one, two, or three (except of course the first triangle, where all the three edges may be tested). For a Delaunay triangulation, this improvement should be used without the stochastic modification, which slows down the process. We call such an algorithm *remembering walk* (RW), and the algorithm with the stochastic step is called *remembering stochastic walk* (RSW). The pseudocode of the RSW algorithm can be found in Algorithm 2, and an example output of this algorithm is shown in Fig. 5.

5.3 Fast remembering stochastic walk

In each step of the RSW algorithm, one or two orientation tests are performed because the orientation test for the edge to the previous triangle is skipped. Kolingerová [15] suggested to choose the next triangle on the basis of only one orientation test. If the query point is on the opposite side of the tested edge than the third vertex of the triangle, the walk moves to the neighbor triangle over this edge. Otherwise, it moves to the neighbor triangle over the nontested edge.

The walk goes correctly, but there is no condition to determine the target triangle, containing the query point, which leads to an infinite walk, circling among the triangles around the query point. Kolingerová suggested to solve this problem by using this walk only for k steps and continue with the RSW algorithm (or for a Delaunay triangulation, RW algorithm). She also proposed experiments showing how the optimal value of k can be computed based on the number of mesh vertices. However, the value of optimal k varies for different vertex distributions and also for different implementations, so a comparison based on a specific value of k would be useless. Therefore, in further tests, we will not include this algorithm to the comparison.

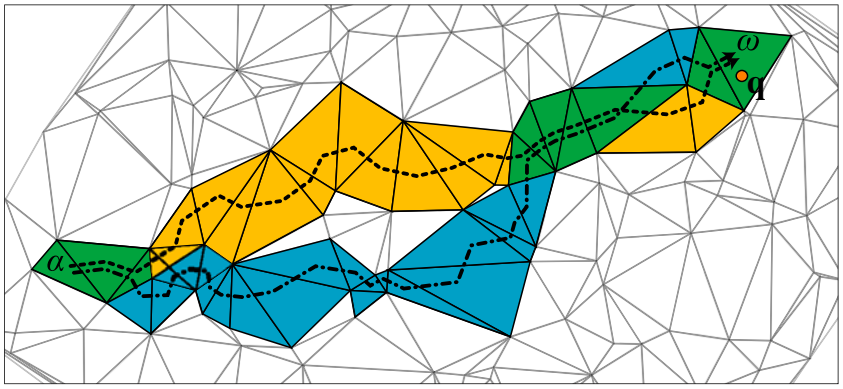

Fig. 5 Example of visibility walk outputs (the remembering stochastic walk is denoted by a *chain line*, the barycentric walk by a *dashed line*)

5.4 Barycentric walk

This algorithm differs from other visibility walk algorithms in the way that it tests the position of the query point with respect to the current triangle. Instead of the 2D orientation test, it uses the barycentric coordinates (see Eq. 6). As already said, the barycentric coordinates have a significant property—with respect to the triangle containing the query point, its barycentric coordinates are all nonnegative. Otherwise, the components with negative values determine beyond which side(s) of the triangle the point lies. This property was utilized by [30] in their algorithm called *Barycentric Walk*.

The main idea of this algorithm (for pseudocode, see Algorithm 3) is as follows: In each step, the algorithm computes barycentric coordinates of **q** with respect to the current triangle. If they are nonnegative, the target triangle is found. Otherwise, we choose the next triangle as a neighbor over the edge given by two vertices with maximum values of the corresponding barycentric coordinate components. We can invert this choice and look for a vertex with a minimal value of its corresponding barycentric coordinate, and the edge to cross will be the one opposite to this vertex. Since the third barycentric component can be computed from the other two components (see Eq. 5), we need three 2D orientation tests for each visited triangle. An example output of this algorithm is shown in Fig. 5.

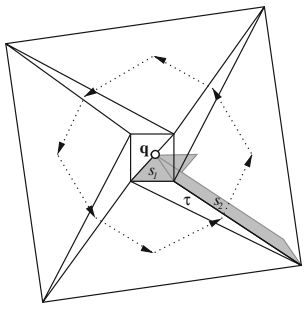

Fig. 6 Loop of the barycentric walk

The main advantage of this algorithm is that it does not depend on the triangles' orientation, so it is very useful when the orientation of the input triangles varies throughout the mesh. Also, Sundareswara claims that the number of triangles visited by the barycentric walk is lower on average than by other visibility walk algorithms, and our experiments confirm this claim (see Section 8.2). Although not stated by [30], the algorithm may loop in some rare cases—see Fig. 6. In the triangle τ, the area s_2 is greater than s_1, so the walk does not cross the edge leading to the triangle containing **q**. A similar case happens in other thin triangles.

6 Straight walk algorithms

Straight walk algorithms do not use only the local comparisons to determine the way of the walk, but they also use a line $\overrightarrow{\mathbf{pq}}$, connecting one point **p** from the starting triangle with the query point **q**, and pass all triangles intersected by this line.

6.1 Original straight walk

The standard straight walk (SW) algorithm [7, 17] works in two steps: an initialization step and a straight walk step. In the initialization step, a point **p** is chosen as one of the vertices of the starting triangle, and by turning around **p**, a triangle δ intersected by the line segment $\overrightarrow{\mathbf{pq}}$ is found. The walk step starts from δ, and in each step, it determines the edge to cross by comparing one of the triangle's vertices to $\overrightarrow{\mathbf{pq}}$ (using the 2D orientation test). Before crossing, it computes the orientation test for the point **q** with respect to the edge it is to cross. If the point **q** is on the inner side of the edge (in our case, the orientation test returns a positive sign), the final triangle has been found. Otherwise, the walk crosses the edge to the next triangle and continues.

Now, let us describe the walking step in detail (the pseudocode of the whole algorithm can be found in Algorithm 4, and an example of the walk is in Fig. 7). For each triangle τ_i with vertices $l_i, r_i,$ and s_i, the line $\overrightarrow{\mathbf{pq}}$ goes into $\tau_i (\tau_i \neq \alpha)$ over its edge $\epsilon_{l_i r_i}$. We compute the orientation of the point $\mathbf{s_i}$ ($\mathbf{s_i}$ is opposite to $\epsilon_{l_i r_i}$) with respect to the line $\overrightarrow{\mathbf{pq}}$. If $\mathbf{s_i}$ is on the right side of $\overrightarrow{\mathbf{pq}}$, we cross the edge $\epsilon_{s_i l_i}$; otherwise, we cross the edge $\epsilon_{r_i s_i}$. As already said, another test is then done: the orientation test of the point $\mathbf{q}$ with respect to the edge chosen to cross, deciding if the final triangle has been found.

During the initialization step, one orientation test is needed for each visited triangle. During the walk, two orientation tests per triangle are done.

6.2 Normal-line straight walk

During the initialization step, the number of performed orientation tests to find the triangle δ is at most equal to the degree of $\mathbf{p}$, which may reach up to n. The modification of [27] makes the number of operations in the initialization step constant and also uses cheaper operations for the location process to speed up the algorithm. For this purpose, the algorithm uses not only the line segment $\overrightarrow{\mathbf{pq}}$ to guide the walk but also its normal line in the point $\mathbf{q}$. Therefore, it is named *normal-line straight walk* (NSW).

The idea of improving the initialization step is as follows (see also Fig. 8). Instead of using an arbitrary point $\mathbf{p}$ and turning around it to find the triangle crossed by the line $\overrightarrow{\mathbf{pq}}$, a point $\mathbf{p}$ is chosen such that the current triangle is crossed by $\overrightarrow{\mathbf{pq}}$. First, from the vertices of the initial triangle, the vertex $\mathbf{v}$ closest to $\mathbf{q}$ is chosen.

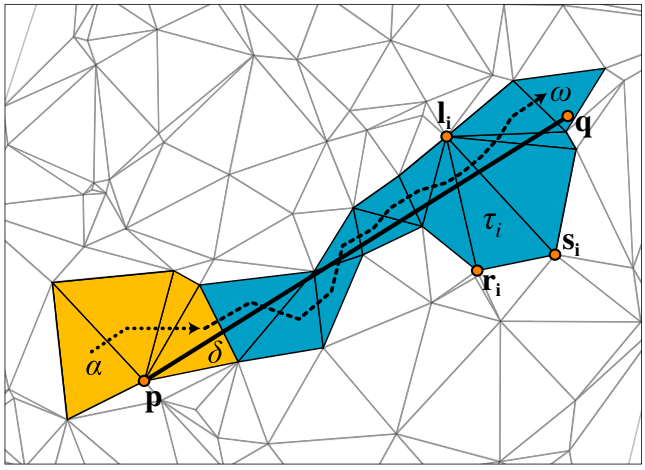

Fig. 7 Straight walk algorithm (a *dotted line* denotes the initialization step, the walk is marked by a *dashed line*)

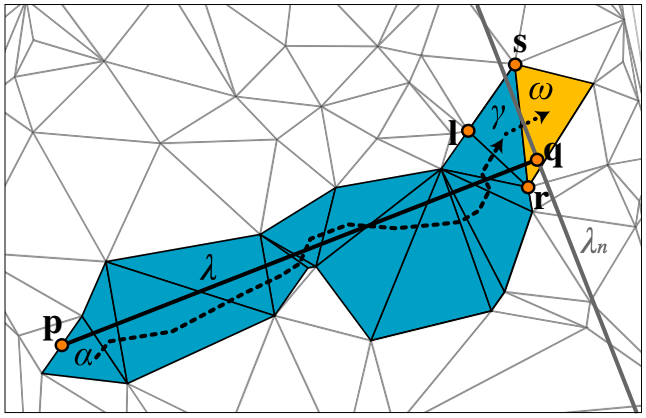

Fig. 8 Normal-line straight walk algorithm (a *dashed line* denotes the walk, a *dotted line* denotes the final short walk by the RSW algorithm)

Then, $\mathbf{p}$ can be chosen anywhere on the edge opposite $\mathbf{v}$ except its endpoints (which could cause loop).

We tested this algorithm also with the original straight walk initialization step, but since the results of this combination were worse, we did not include these tests in further comparisons.

After selecting the proper point $\mathbf{p}$, the walk may begin. The walk runs nearly in the same way as the original straight walk, with two significant differences. To compute the position of $\mathbf{s_i}$ with respect to the line $\overrightarrow{\mathbf{pq}}$, an implicit line equation of $\lambda = \overrightarrow{\mathbf{pq}}$ (Eq. 4) is used instead of the 2D orientation test. The implicit line equation is precomputed in the initialization step. A test using the implicit line equation is faster; however, in the original form of the algorithm, it cannot be used for the decision whether $\mathbf{q}$ is inside the current triangle. This comparison is done with respect to an edge, which is different in each step, so the implicit line equation of this edge cannot be precomputed. Therefore, the orientation test for this decision is replaced by the position test of $\mathbf{s_i}$ with respect to the normal line λ_n, $\lambda_n \perp \lambda$, $\mathbf{q} \in \lambda_n$, which is precomputed in the initialization step and does not change during the walk. If $\mathbf{s_i}$ is on the other side of λ_n than $\mathbf{p}$, the straight walk ends. However, this does not necessarily mean that the current triangle contains $\mathbf{q}$ (see Fig. 8). Therefore, the RSW algorithm (Section 5.2) is always used for the final location, which is very short (usually about two triangles according to our experiments in Section 8.2). See Algorithm 5 for pseudocode of this modified straight walk algorithm.

7 Orthogonal walk algorithms

The main idea of this group of algorithms, proposed by [7], is to make the position tests cheaper by walk-

ing along the coordinate axis and comparing only one coordinate at a time.

7.1 Original orthogonal walk

The original orthogonal walk [7] consists of three steps: an initialization step, a walk along the x-axis, and a walk along the y-axis (see Fig. 9). The walk is usually longer than other walks, but its tests are much cheaper, which result in a faster location.

Now, let us describe the algorithm in detail (its pseudocode is in Algorithm 6). In the initialization step, a point $\mathbf{p}$ is selected as one of the vertices of the starting triangle and a point $\mathbf{a}$ is set as $\mathbf{a} = (q_x, p_y)$. We rotate around $\mathbf{p}$ to find a triangle δ that is intersected by the line $\lambda_x = \overrightarrow{\mathbf{pa}}$, which is parallel to the x-axis. In the tests, only the y-coordinates of the vertices are compared to find δ and only one comparison is needed per triangle.

As the next step, the directions of the horizontal and vertical walks need to be determined according to the position of $\mathbf{q}$ with respect to $\mathbf{p}$. In the following text, we will describe only the case, when $\mathbf{q}$ is above and to the right of $\mathbf{p}$ ($q_x > p_x, q_y > p_y$); other cases are analogous.

The horizontal walk follows the line λ_x, until the current triangle contains $\mathbf{a}$. For each triangle τ_i with vertices $\mathbf{l_i}$, $\mathbf{r_i}$, and $\mathbf{s_i}$, let the edge $\epsilon_{l_i r_i}$ be the edge used to cross to this triangle, $\mathbf{l_i}$ be above λ_x, and $\mathbf{r_i}$ below. The next edge to cross is determined by comparing s_{iy} and a_y. If $s_{iy} > a_y$ ($\mathbf{s_i}$ is above λ_x), the edge $\epsilon_{r_i s_i}$ is crossed to move to the next triangle; otherwise, the edge $\epsilon_{l_i s_i}$ is crossed (see Fig. 9). The horizontal walk ends if $\mathbf{a}$ is inside τ_i: if the x-coordinate of any of the vertices $\mathbf{l}$ and $\mathbf{r}$ is greater than a_x, the orientation test (orientation2D($\mathbf{l_i}$, $\mathbf{r_i}$, $\mathbf{a}$) ≥ 0) is performed to verify if the triangle containing $\mathbf{a}$ was found.

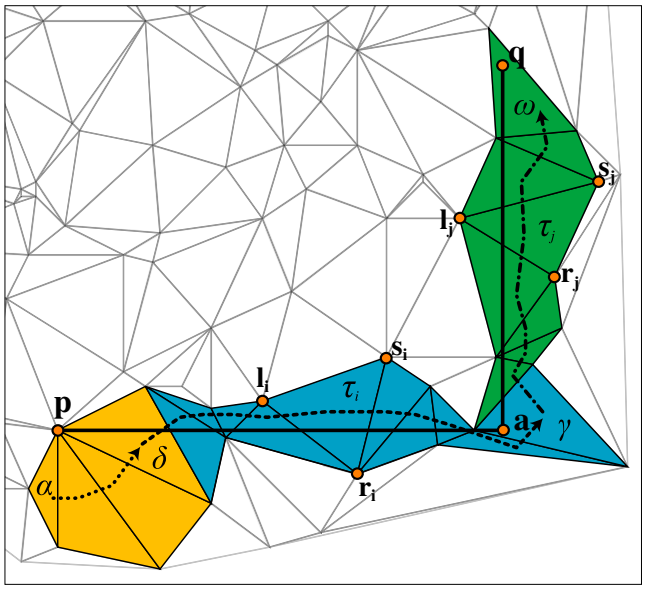

Fig. 9 Orthogonal walk algorithm (a *dotted line* denotes the initialization step, the walk along the x-axis is marked by a *dashed line*, and the walk along the y-axis is marked by a *chain line)*

The vertical walk follows the line $\lambda_y = \overrightarrow{\mathbf{aq}}$, which is parallel to the y-axis. At the beginning of the vertical walk, such vertices $\mathbf{l_j}, \mathbf{r_j}, \mathbf{s_j}$ are chosen so that $\mathbf{l_j}$ is to the left from λ_y and $\mathbf{r_j}$ is to the right from λ_y. For each triangle τ_j with vertices $\mathbf{l_j}$, $\mathbf{r_j}$, *and* $\mathbf{s_j}$ in the vertical walk, the edge $\epsilon_{l_j r_j}$ is the edge used to cross to this triangle. This time, we compare s_{jx} and q_x to determine which edge to cross. If $s_{jx} < q_x$ ($\mathbf{s_j}$ is to the left of λ_y), we cross the edge $\epsilon_{s_j r_j}$; otherwise, we cross the edge $\epsilon_{l_j s_j}$. The vertical walk ends if $\mathbf{q}$ is inside τ_j (again, using the test orientation2D($\mathbf{l_j}$, $\mathbf{r_j}$, $\mathbf{q}$) ≥ 0).

Three comparisons are needed for each triangle during the walk. Note that the orientation tests are also used in the orthogonal walk, but only for a few last triangles in each direction, to determine the exact location of the points $\mathbf{a}$ and $\mathbf{q}$.

The orthogonal walk has one significant drawback. If it crosses the border of a triangle mesh during horizontal or vertical walk, it usually does not find the target triangle. In its original form proposed by [7], this case is not handled, but the algorithm can be modified to return the correct result—if the border is crossed during the horizontal walk, it switches to the vertical walk, but with a new vertical line, set by selecting a new reference point $\mathbf{a}'$ as an arbitrary inner point of the current triangle and $\mathbf{q}' = (a'_x, q_y)$. If the border is crossed during the vertical walk, the walk stops. If the border was crossed during the horizontal or vertical search, the algorithm uses its last visited triangle as the new starting triangle and runs again.

Another possibility to avoid crossing the border is to enclose our data in a rectangle and triangulate the additional area during preprocessing. However, this is not always desired.

7.2 Improved orthogonal walk

Improvements to orthogonal walk proposed by [28] are analogous to those for the straight walk algorithm (Section 6.2). The first idea is to simplify the initialization step of the orthogonal walk algorithm to a constant number of operations. Second idea is to use a smaller number of coordinate comparisons per each visited triangle and avoid the orientation tests during the orthogonal walk, resulting in a need to use RSW for a short final location of ω. The pseudocode of the modified orthogonal walk can be found in Algorithm 7, and an example of the walk is in Fig. 10.

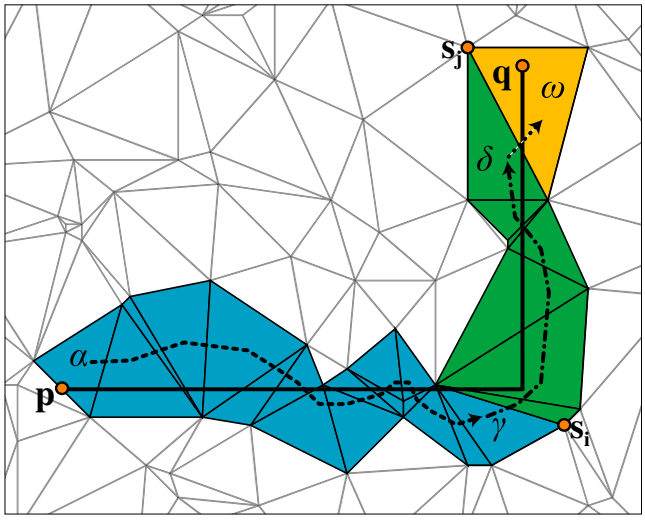

Fig. 10 Improved orthogonal walk algorithm (a *dashed line* denotes the walk along the x-axis, the walk along the y-axis is marked by a *chain line*, and the final short walk by the RSW algorithm is marked by a *chain line*)

The idea of improving the initialization step is as follows: First, the position of **q** is determined with respect to an arbitrary vertex of the starting triangle α. If **q** is above and to the right of **p** ($q_x > p_x, q_y > p_y$) (other cases are analogous), a point $\mathbf{s}_\alpha$ is chosen as a vertex of α with the maximal x-coordinate value.

During the walk, the original algorithm tested whether the current triangle contains the query point by comparing the target point with both endpoints of the edge which should have been crossed. If at least one of the endpoints had higher x-coordinate than **q**, it performed the 2D orientation test on this edge, and in the case of a negative result, the walk continued. However, since it is not possible to determine the target triangle only by testing the given coordinates of the two points, the tests can be simplified by comparing only **s** with **q**. This way, the walk not necessarily finishes in the target triangle, but it does in a close neighborhood. For the case of a walk in the x-axis direction, the target triangle needs not be determined exactly, and for the y-axis direction, a very short walk locating **q** is done by the RSW algorithm (usually about three triangles for rectangular input data according to our experiments in Section 8.2). However, a simple initialization test has to be added before the walk in the y-direction starts: a point $\mathbf{s}_\gamma$ is chosen as a vertex of γ with the maximal y-coordinate value.

The improved orthogonal walk has the same problems with border crossing as the original one. The horizontal crossing is solved in the same way. In a case of a border crossing, the last triangle of the walk is usually not in the close neighborhood of the one containing the query point, but somewhere further. However, as the RSW is used for the final location, it finds the correct triangle, but it may perform more than only a few steps in this case, which significantly slows down process.

8 Experimental results

We tested the following algorithms: the RW and the RSW (see Section 5.2), the barycentric walk (BW, Section 5.4), SW (Section 6.1), NSW (Section 6.2), orthogonal walk (OW, Section 7.1), and improved orthogonal walk (IOW, Section 7.2).

The tests were performed on many different datasets of three types: randomly distributed points in a unit square, data from cadastre of the cities, where the vertices defining the boundaries of built-up areas (see Fig. 11) and almost regular LIDAR data scanned over the landscape where occasionally some points missing (i.e., on the water surfaces).

In the tests, we took into account also numerical precision. If the located point is very close to the edge of the final triangle, the algorithms may return its neighbor as a result due to numerical problems. Furthermore, if the query point is very close to a vertex of a triangle mesh, it may cause looping of the visibility walk algorithms (or algorithms using RSW for the final location). We tested the possibility of such a loop on a Delaunay triangulation of 10^5 randomly generated points in a unit square. During 10^{10} location processes, where the located points were selected at a maximal distance of 10^{-15} around the mesh vertices, the RSW algorithm looped three times. During location processes performed for 10^{12} randomly generated query points, the algorithm did not loop.

For the test purposes, we implemented the specified algorithms in C++ with both double precision floating point arithmetic and also with adaptive floating point arithmetic [25] to avoid numerical problems. Adaptive robust geometric predicates were used only where it was needed, i.e., either for the final location or also during the whole process of the visibility walks to prevent the loop. The algorithms were tested on Intel Q6600 2.40 GHz. The SSE2 random generator was used for the RSW algorithm because it is declared as up to five times faster then the standard C random generator [20].

The sophisticated selection of the initial triangle (Section 4.1) was not included in the tests, since it speeds up all the algorithms in the same way. Therefore, a random initial triangle and a random target point were generated during the testing. Each time, we performed 10^7 location processes and computed the average number of the tested quantities.

Fig. 11 Data from cadastre: dataset with 4,897 vertices (*left*) and with 15,824 vertices (*right*)

First, we tested the influence of the type and size of the triangulation on the walk length (Section 8.1). Then, we tested the specified algorithms on different dataset types and sizes (Section 8.2).

8.1 Influence of the triangulation type on the walk length

In the first part of our experiments, we tested the influence of the triangulation type on the length of the walk (number of triangles visited by the walk). The tests were performed on two types of datasets: randomly distributed points in the unit square and data from cadastre. On each dataset, we constructed four types of triangulations: a DT [25], a Greedy triangulation (GT) [9], a minimum weight triangulation (MWT) [2], and a min–max angle triangulation (min–max) [12]. We also considered a regular triangulation [32], but we decided not to include it in the final comparison, since on the same dataset, it eliminates redundant vertices [35]. Therefore, the number of vertices in the final triangulation is smaller then in the other types of triangulations, leading to unequal conditions for the tests of the walk length. If lower weights are used for the construction of the regular triangulation, the number of vertices remains about the same; however, such a triangulation is similar to DT.

The test results indicated that the number of visited triangles depends on the geometric properties of the triangulation, especially on the total sum of all edge lengths Σe_i. To confirm this assumption, we calculated the correlation between Σe_i and the number of visited triangles during the walk. As a measure of correlation, we used the Pearson correlation coefficient [21] in the form presented by [31]. This coefficient $r(X, Y)$ is defined for two random variables X and Y:

$$r(X, Y) = \frac{\operatorname{cov}(X, Y)}{\operatorname{stdev}(X) \cdot \operatorname{stdev}(Y)} \tag{7}$$

where $\operatorname{cov}(X, Y)$ is the covariance between the variables X and Y and $\operatorname{stdev}(X) \cdot \operatorname{stdev}(Y)$ is a product of their standard deviations. The coefficient cannot exceed 1 in absolute value. If $r(X, Y) = 1$, then there is a perfect increasing linear dependence (correlation) between the variables, -1 shows a perfect decreasing linear dependence (anticorrelation). The closer the value is to zero, the less dependent the tested variables are.

The correlation coefficient can be estimated from a limited sample of the values:

$$r(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n} (x_i - \hat{x}) \cdot (y_i - \hat{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \hat{x})^2} \cdot \sqrt{\sum_{i=1}^{n} (y_i - \hat{y})^2}} \tag{8}$$

where $\hat{x}$ denotes the average value and the correlation is evaluated for the number of visited triangles during the walk ($\mathbf{x}$) and Σe_i ($\mathbf{y}$) of the tested triangulations and n is the number of tested cases. Each case contained one of the four triangulations constructed on one dataset.

Table 1 shows the behavior of RSW algorithm on four selected datasets; other algorithms had similar results in terms of correlation coefficients. The correla-

Table 1 The relation between the average number of visited triangles (ϕ #Δ) by the RSW algorithm and the sum of the edge lengths of the triangulations ($\Sigma\epsilon_i$), $r(\mathbf{x}, \mathbf{y})$ denotes the correlation coefficient of ϕ #Δ and $\Sigma\epsilon_i$

	DT	GT	MWT	Min–max	$r(\mathbf{x}, \mathbf{y})$	DT	GT	MWT	Min–max	$r(\mathbf{x}, \mathbf{y})$
Dataset	Uniform distribution, 10^4 vertices (19,994 Δ)					Uniform distribution, 10^5 vertices (199,994 Δ)				
Σe_i	723.09	707.12	706.43	770.15	0.998	2,207.64	2,158.76	2,156.69	2,355.01	0.997
ϕ #Δ	115.513	114.41	114.259	120.187		363.938	360.938	360.539	378.648	
Dataset	Real geodetic data, 15,824 vertices (31,642 Δ)					Real geodetic data, 70,437 vertices (140,868 Δ)				
$\Sigma\epsilon_i$	12,091.4	10,447.3	10,403.7	15,408.6	0.999	20,398.0	18,492.4	18,449.9	28,390.4	0.997
ϕ #Δ	158.098	151.911	150.839	174.102		321.27	308.134	307.646	357.233	

tion coefficient is nearly 1, indicating a perfect increasing linear correlation. Our tests performed for all the presented algorithms on many other datasets produced similar results and so verified that the length of the walk is linearly dependent on the sum of the length of the triangulation edges.

8.2 Performance comparison of the algorithms

As we found that the walk is linearly dependent on the sum of the length of the triangulation edges, there is no need to present the test results for all the triangulation types. Thanks to its popularity, we chose Delaunay triangulation to compare the specified algorithms. Recall that the RW and BW algorithms may loop for other types of triangulations, so despite their good performance, they can be used safely only in algorithms working with a Delaunay triangulation. The tests were performed on many different datasets of three types: randomly distributed points in a unit square, data from cadastre, and LIDAR data (see details in the introduction of Section 8).

Selected results are in Table 2. The following qualities were examined for each algorithm: the average number of visited triangles (#Δ), the average number of tests (#tests), the average time per one location with double precision floating point arithmetic (t (μs)), and the average time per one location with a numerically robust solution (t_e (μs), the adaptive robust geometric predicates are used wherever needed). The average time per one location using numerically robust solution (t_e (μs)) is also compared for randomly distributed points in a unit square in Fig. 12, where the horizontal axis shows the number of vertices in the dataset and the vertical axis shows t_e (μs).

Note that the average number of tests performed by each algorithm has only information value which is not relevant as a measure of the performance of the algorithms—for that, we should use the time values, since the speed of the tests vary among the algorithms. The properties #tests and #Δ consist of two values for some algorithms. Mostly, the former value concerns the walk and the latter concerns the final location performed by RSW (or eventually by RW for the nonstochastic versions of algorithms). For the SW algorithm (Section 6.1), the former value concerns the initialization step and the latter concerns the walk step. For the OW algorithm (Section 7.1), the former value of #tests denotes the number of coordinate comparisons, and the latter one, the number of 2D orientation tests. Results are summarized in Section 8.3.

8.3 Comparison of the implementation issues

To provide more information to those willing to implement a walking algorithm, we consider in this section not only measurable properties (stability and speed) but also implementation effort, which we evaluated according to our experience. The readers may choose according to their priorities and requirements of the intended application.

Table 3 shows the comparison of the proposed algorithms. The specified properties are evaluated by integers from 1 to 5, where 1 denotes the best performance and 5 the worst. Now, let us discuss the performance of each algorithm.

In stability, we address four issues: possible looping caused by numerical problems, looping in non-Delaunay triangulations, the possibility of crossing the edge of the triangle mesh during the location, and significant slowdown under specific circumstances. Numerical problems can be avoided by using algorithms with adaptive robust geometric predicates [25]; evaluation of algorithms using these tests is listed in parentheses in Table 3.

RW (from Section 5.2) is very simple to implement and also quite fast; however, it may loop on other than Delaunay triangulations. On our datasets with other types of triangulations than DT, it never looped; however, it is not guaranteed. Therefore, for a stable solution with a non-Delaunay triangulation, another algorithm should be used. As other visibility walks, it can

Table 2 Comparison of the walking algorithms with randomly chosen α (#Δ represents the number of visited triangles, #tests represents the number of performed tests, t (μs) represents the time per one location with double precision floating point arithmetics, and t_e (μs) represents the time per one location with adaptive robust arithmetics)

	#Δ	#tests	t (μs)	t_e (μs)	#Δ	#tests	t (μs)	t_e (μs)	#Δ	#tests	t (μs)	t_e (μs)
Algorithm	ϕ per located point				ϕ per located point				ϕ per located point			
Cadastre data												
	4,897 vertices (9,774 Δ)				15,824 vertices (31,642 Δ)				70,437 vertices (140,868 Δ)			
RW	94.4	129.5	3.48	5.17	160.5	213.2	6.52	9.42	321.3	417.9	14.90	20.70
RSW	92.1	122.2	5.89	8.81	158.1	208.5	10.71	15.72	312.9	414.8	23.04	32.83
BW	87.4	262.2	5.92	8.85	149.6	448.8	10.72	15.69	283.1	849.2	21.87	31.33
SW	2.8 + 88.2	3.3 + 175.4	4.75	7.09	2.7 + 149.6	3.2 + 298.3	8.63	12.58	2.8 + 294.8	3.3 + 588.9	18.72	26.47
NSW	87.5 + 2.3	174.0 + 4.3	4.79	4.87	149.2 + 2.4	297.3 + 4.4	8.57	8.65	293.4 + 2.9	585.8 + 4.9	18.30	18.35
OW	96.7	297.6 + 4.1	2.16	2.34	179.7	546.6 + 5.0	4.68	4.84	324.4	980.9 + 6.2	10.25	10.43
IOW	94.2 + 4.3	192.6 + 7.0	2.01	2.21	176.5 + 2.7	357.3 + 5.2	4.32	4.53	319.0 + 3.5	642.0 + 6.4	9.50	9.72
LIDAR												
	34,932 vertices (69,858 Δ)				313,348 vertices (626,690 Δ)				3,722,068 vertices (7,444,130 Δ)			
RW	205.5	275.4	8.98	12.61	613.5	823.3	38.18	49.07	2,569.1	3,444.2	181.51	227.28
RSW	202.6	265.6	14.26	20.48	608.5	792.6	54.09	73.02	2,543.3	3,305.4	248.08	326.66
BW	179.3	537.9	13.34	19.3	509.1	1,527.3	47.24	63.96	2,203.3	6,609.8	222.76	295.26
SW	2.7 + 180.8	3.2 + 360.6	10.94	15.63	2.7 + 542.8	3.2 + 1,084.6	42.98	57.14	2.8 + 2,316.4	3.3 + 4,631.8	202.65	262.40
NSW	182.8 + 1.9	364.7 + 4.0	10.86	10.93	539.4 + 1.9	1,077.8 + 4.0	41.76	41.85	2,333.0 + 2.3	4,665.0 + 4.6	199.69	200.17
OW	239.7	719.9 + 3.5	6.91	7.05	681.8	2,046.0 + 3.5	32.48	32.65	2,756.9	8,270.9 + 4.0	154.53	155.03
IOW	240.7 + 1.7	485.8 + 3.8	6.60	6.72	695.0 + 1.7	1,394.3 + 3.8	31.88	31.97	2,741.8 + 2.2	5,487.8 + 4.5	148.45	148.75
Randomly distributed points in the unit square												
	10^4 vertices (19,994 Δ)				10^5 vertices (199,994 Δ)				10^6 vertices (1,999,994 Δ)			
RW	118.1	159.2	4.34	6.47	366.1	491.5	19.43	25.98	1,144.7	1,533.0	81.39	101.94
RSW	115.8	152.5	7.40	11.07	362.5	474.5	29.02	40.52	1,130.5	1,476.5	110.94	146.78
BW	103.2	309.6	7.08	10.55	326.8	980.1	27.60	38.56	1,028.2	3,084.5	105.12	139.44
SW	2.8 + 105.5	3.3 + 210.0	5.71	8.51	2.7 + 335.6	3.2 + 670.2	23.56	32.44	2.7 + 1,065.4	3.2 + 2,129.8	93.96	122.23
NSW	107.1 + 1.7	213.2 + 3.8	5.76	5.82	337.0 + 1.8	673.0 + 3.9	23.14	23.21	1,057.6 + 1.8	2,114.3 + 3.9	91.32	91.38
OW	136.7	410.8 + 3.5	2.96	3.09	432.2	1,297.1 + 3.5	16.46	16.54	1,364.6	4,094.4 + 3.5	76.56	76.71
IOW	137.8 + 1.8	279.9 + 3.9	2.84	2.97	433.3 + 1.7	870.8 + 3.8	15.71	15.87	1,336.3 + 1.8	2,676.9 + 3.9	72.42	72.71

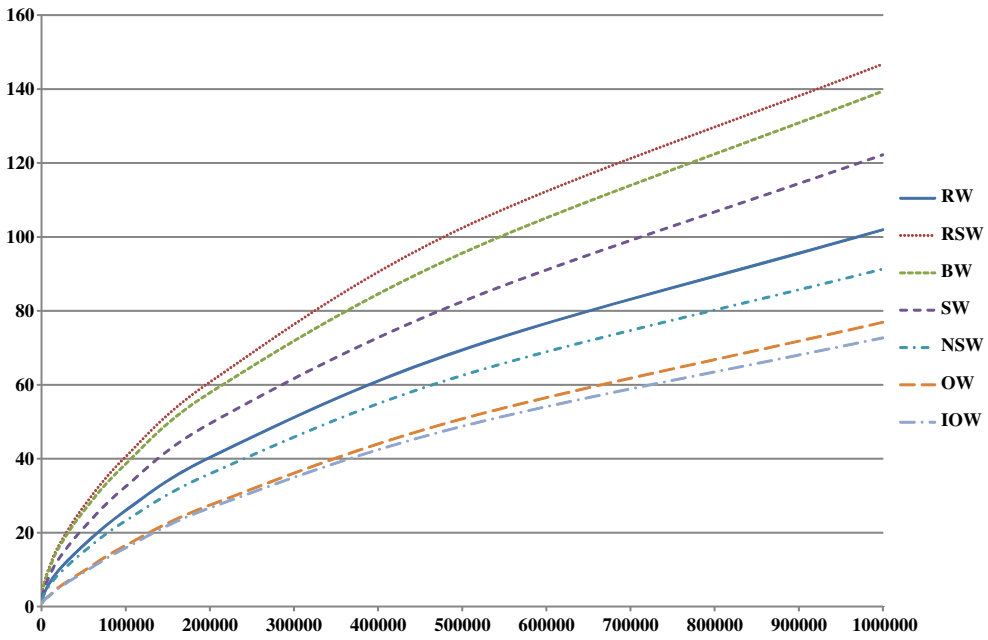

Fig. 12 The average time of one robust location for randomly distributed points in a unit square

also loop without adaptive robust geometric predicates, but if the query point is not very close to any vertex of the triangle mesh, this loop is very improbable. If the input triangulation is guaranteed to be Delaunay, RW should be preferred to RSW.

RSW (from Section 5.2) is a stable algorithm, but like other visibility walks, it can loop without adaptive robust geometric predicates. However, this algorithm is also the slowest because of the randomization step, which provides space to improvements by a new effective pseudorandomizer. Thanks to the stability and simplicity of the algorithm, it is the most popular walking algorithm in the scientific community.

BW (from Section 5.4) is the second slowest algorithm and can also loop for non-Delaunay triangulations, but such a configuration is less probable than for RW (see Fig. 6). As well as other visibility walks, it can also loop without adaptive robust geometric predicates. This algorithm is useful especially when the orientation of the input triangles varies throughout the mesh, since its tests do not depend on the triangle orientation. Also, an interesting result from the tests is that the algorithm visits the least number of triangles (even less than straight walk).

SW (from Section 6.1) is the most stable algorithm, and it cannot loop, even without adaptive robust geometric predicates. It is faster than RSW; however, it is more complicated to implement.

NSW (from Section 6.2) is easier to implement than SW, but its implementation effort is similar, since apart from the algorithm itself, RSW has to be implemented for a final short location. It may seem that the implicit line equation test (see Eq. 4) is less numerically accurate; however, an incorrect decision causes only a visit of another triangle and does not affect the stability of the algorithm. Since the final location by RSW is usually short (see Table 2), we can get exact results without noticeable slowdown by using adaptive robust geometric predicates only for the final location.

OW (from Section 7.1) is very fast thanks to its cheap tests, however, not usable in its original form if there is a possibility that the walk crosses the border of a triangulation, so it has the worst stability mark. If we implement an additional solution for such a situation, the algorithm becomes slower. Note that to be able to test this algorithm in our tests, we bounded our input in a rectangle by adding four vertices to the mesh and retriangulating it.

IOW (from Section 7.2) is the fastest walking algorithm, but also complex to implement, because RSW

Table 3 Overall comparison of the algorithms (evaluation of precise solutions using adaptive robust geometric predicates is listed in parentheses)

Algorithm	Implementation	Stability	Speed
RW	1	4 (3)	2 (3)
RSW	2	2 (1)	4 (5)
BW	2	3 (2)	4 (5)
SW	3	1 (1)	3 (4)
NSW	3	2 (1)	2 (2)
OW	5	5 (5)	1 (1)
IOW	5	3 (2)	1 (1)

has to be implemented for the final, usually short location. Again, we need the adaptive robust geometric predicates only for the short final location by RSW. If the walk crosses the border of a triangulation, its final triangle will be further from the target, resulting in a longer walk with RSW. Therefore, it is slower for such cases (and has lower stability mark), but it is always correct. Note that for our tests, we bounded the input data by a rectangle to make the testing of OW possible; thus, the final location by RSW was short.

For the most types of input data, we recommend the NSW or the IOW algorithm with adaptive robust geometric predicates. IOW for such cases where crossing the border of a triangulation is less probable, NSW in other cases. For the final location, it is desirable to use RW instead of RSW when we are walking in DT. For a quick solution, we recommend RSW (or RW for DT) which is easy to implement. BW is useful especially when the orientation of the input triangles varies throughout the mesh, since tests do not depend on the triangle orientation. For a stable algorithm which cannot loop without adaptive robust geometric predicates, we recommend SW.

9 Conclusion

We presented a complete overview of the existing walking algorithms and their extensions for planar point location in triangulations. We proposed experiments comparing the time performance of these algorithms on random as well as real datasets, on which we constructed the common triangulations. We discovered an interesting linear dependence of the length of the walk on the sum of edges of the triangulation. We also made a simple comparison of the speed, implementation effort, and stability of the algorithm, providing a tool for decision which algorithm is suitable to implement in a particular solution.

Acknowledgements The authors would like to thank their colleagues Libor Váša, Tomáš Bayer, Tomáš Koutný, and Radek Fiala for their feedback and inspiring discussions. This work has been supported by the Grant Agency of the Czech Republic under the research projects 201/09/0097 and P202/10/1435 and by the project SGS-2010-02.

Appendix: Pseudocodes

For easier implementation of the described walking algorithms, this section provides their pseudocodes. Note that to achieve better readability, we assume that the query point lies inside the given triangle mesh. Otherwise, a test if the neighboring triangle exists should be included.

For the sake of simplicity, all the following algorithms are considered to have the input and the output in the following form:

Input

- The query point $\mathbf{q}$
- the chosen starting triangle $\alpha \in T$

Output

- The triangle ω which contains $\mathbf{q}$

Algorithm 1 Lawson's oriented walk

```
triangle τ = α;
boolean found = false;
while not found do
    found = true;
    foreach edge ϵ ∈ τ do
        point l = first vertex of ϵ;
        point r = second vertex of ϵ;
        if orientation2D(l, r, q) < 0 then
            τ = neighbor of τ over ϵ;
            found = false;
            break; // terminates the foreach cycle
        end
    end
end
return τ;
```

Algorithm 2 Remembering stochastic walk

```
triangle τ = α;
triangle ψ = τ; // the previous triangle is initialized as τ
boolean found = false;
while not found do
    found = true;
    int k = random_int(3); // k ∈ {0, 1, 2}
    for i = k to k + 2 do
        point l = t_(i mod 3);
        point r = t_[(i+1) mod 3];
        // "remembering" condition
        if ψ is not neighbor of τ over ϵ_lr then
            if orientation2D(l, r, q) < 0 then
                ψ = τ;
                τ = neighbor of τ over ϵ_lr;
                found = false;
                break; // terminates the for cycle
            end
        end
    end
end
return τ;
```

Algorithm 3 Barycentric walk

triangle $\tau = \alpha = \mathbf{t_0 t_1 t_2}$;
boolean $found = false$;
double min;
edge ϵ;
while *not found* **do**
  // compute barycentric coordinates of q with respect to the triangle (t_0, t_1, t_2)
  double $b_0 = b_0(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{q})$; // computes the first component
  double $b_1 = b_1(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{q})$; // computes the second component
  double $b_2 = 1 - b_0 - b_1$;
  ϵ = edge $\mathbf{t_1 t_2}$;
  $min = b_0$;
  if $b_1 < min$ **then**
    ϵ = edge $\mathbf{t_2 t_0}$;
    $min = b_1$;
  end
  if $b_2 < min$ **then**
    ϵ = edge $\mathbf{t_0 t_1}$;
    $min = b_2$;
  end
  if $min < 0$ **then**
    τ = neighbor of τ over ϵ;
  else
    $found = true$;
  end
end
return τ;

Algorithm 4 Straight walk

triangle $\tau = \alpha = \mathbf{lrs}$;
point $\mathbf{p} = \mathbf{s}$;
if $orientation2D(\mathbf{p}, \mathbf{q}, \mathbf{r}) > 0$ **then**
  while $orientation2D(\mathbf{p}, \mathbf{q}, \mathbf{l}) > 0$ **do**
    $\mathbf{r} = \mathbf{l}$;
    τ = neighbor of τ over ϵ_{pl};
    $\mathbf{l}$ = vertex of τ where $\mathbf{l} \neq \mathbf{p}, \mathbf{l} \neq \mathbf{r}$;
  end
else
  repeat
    $\mathbf{l} = \mathbf{r}$;
    τ = neighbor of τ over ϵ_{pr};
    $\mathbf{r}$ = vertex of τ where $\mathbf{r} \neq \mathbf{p}, \mathbf{r} \neq \mathbf{l}$;
  until $orientation2D(\mathbf{p}, \mathbf{q}, \mathbf{r}) > 0$;
end
$switch(\mathbf{l}, \mathbf{r})$;
// now $\overrightarrow{\mathbf{pq}}$ has $\mathbf{r}$ on the right and $\mathbf{l}$ on the left side
// straight walk - following the line segment $\overrightarrow{\mathbf{pq}}$
while $orientation2D(\mathbf{l}, \mathbf{r}, \mathbf{q}) < 0$ **do**
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ where $\mathbf{s} \neq \mathbf{r}, \mathbf{s} \neq \mathbf{l}$;
  if $orientation2D(\mathbf{p}, \mathbf{q}, \mathbf{s}) < 0$ **then** $\mathbf{r} = \mathbf{s}$ **else** $\mathbf{l} = \mathbf{s}$;
end
return τ;

Algorithm 5 Normal-line straight walk

$\tau = \alpha = \mathbf{lrs}$ where $\mathbf{s}$ is the nearest vertex to $\mathbf{q}$;
$\mathbf{p}$ = point on edge ϵ_{lr} where $\mathbf{l}, \mathbf{r} \neq \mathbf{p}$ (i.e., $\mathbf{p}$ is midpoint of ϵ_{lr});
line λ = line segment $\overrightarrow{\mathbf{pq}}$;
line λ_n = line segment orthogonal to $\overrightarrow{\mathbf{pq}}$ where $\mathbf{q} \in \lambda_n$;
// following the line segment λ from $\mathbf{p}$ to $\mathbf{q}$
while $position(\lambda_n, \mathbf{s}) < 0$ **do**
  if $position(\lambda, \mathbf{s}) < 0$ **then** $\mathbf{r} = \mathbf{s}$ **else** $\mathbf{l} = \mathbf{s}$;
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ where $\mathbf{s} \neq \mathbf{r}, \mathbf{s} \neq \mathbf{l}$;
end
return $remembering\_stochastic\_walk(\mathbf{q}, \tau)$;

Algorithm 6 Orthogonal walk

triangle $\tau = \alpha = \mathbf{lrs}$; point $\mathbf{p} = \mathbf{s}$; point $\mathbf{a} = (q_x, p_y)$;
// $\mathbf{q}$ is above and to the right of $\mathbf{p}$ $(q_x > p_x, q_y > p_y)$, other cases are analogous
if $r_y > p_y$ **then**
  while $l_y > p_y$ **do**
    $\mathbf{r} = \mathbf{l}$;
    τ = neighbor of τ over ϵ_{pr};
    $\mathbf{l}$ = vertex of τ where $\mathbf{l} \neq \mathbf{p}, \mathbf{l} \neq \mathbf{r}$;
  end
else
  repeat
    $\mathbf{l} = \mathbf{r}$;
    τ = neighbor of τ over ϵ_{pl};
    $\mathbf{r}$ = vertex of τ where $\mathbf{r} \neq \mathbf{p}, \mathbf{r} \neq \mathbf{l}$;
  until $r_y > p_y$;
end
$switch(\mathbf{l}, \mathbf{r})$;
// now τ is intersected by $\overrightarrow{\mathbf{pa}}$, $p_y \leq l_y$ and $p_y \geq r_y$; traverse T in the dir. of x-axis
while *($l_x < a_x$ and $r_x < a_x$) or ($orientation2D(\mathbf{l}, \mathbf{r}, \mathbf{a}) < 0$)* **do**
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ where $\mathbf{s} \neq \mathbf{l}, \mathbf{s} \neq \mathbf{r}$;
  if $s_y > a_y$ **then** $\mathbf{l} = \mathbf{s}$ **else** $\mathbf{r} = \mathbf{s}$
end
if $r_x > q_x$ **then**
  $\mathbf{s} = \mathbf{l}$;
  $\mathbf{l}$ = vertex of τ where $\mathbf{l} \neq \mathbf{r}, \mathbf{l} \neq \mathbf{s}$;
end
// now τ is intersected by $\overrightarrow{\mathbf{aq}}$, $l_x \leq q_x$ and $r_x \geq q_x$); traverse T in the dir. of y-axis
while *($l_y < q_y$ and $r_y < q_y$) or ($orientation2D(\mathbf{l}, \mathbf{r}, \mathbf{q}) < 0$)* **do**
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ where $\mathbf{s} \neq \mathbf{l}, \mathbf{s} \neq \mathbf{r}$;
  if $s_x > q_x$ **then** $\mathbf{r} = \mathbf{s}$ **else** $\mathbf{l} = \mathbf{s}$;
end
return τ;

Algorithm 7 Improved orthogonal walk

$\mathbf{p}$ = a point generated anywhere inside α;
// we describe the case where $\mathbf{q}$ is above and to the right of $\mathbf{p}$ $(q_x > p_x, q_y > p_y)$, other cases are analogous
$\tau = \alpha = \mathbf{lrs}$, where $\mathbf{s}$ is the vertex with maximal x coordinate;
// traverses T in the direction of x-axis
while $s_x < q_x$ **do**
  if $s_y < p_y$ **then** $\mathbf{r} = \mathbf{s}$ **else** $\mathbf{l} = \mathbf{s}$;
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ where $\mathbf{s} \neq \mathbf{r}, \mathbf{s} \neq \mathbf{l}$;
end
$\tau = \mathbf{lrs}$, where $\mathbf{s}$ is the vertex with maximal y coordinate;
// traverses T in the direction of y-axis
while $s_y < q_y$ **do**
  if $s_x < q_x$ **then** $\mathbf{l} = \mathbf{s}$ **else** $\mathbf{r} = \mathbf{s}$;
  τ = neighbor of τ over ϵ_{lr};
  $\mathbf{s}$ = vertex of τ, where $\mathbf{s} \neq \mathbf{r}, \mathbf{s} \neq \mathbf{l}$;
end
return $remembering\_stochastic\_walk(\mathbf{q}, \tau)$;

References

1. Amenta, N., Choi, S., Rote, G.: Incremental constructions con BRIO. In: SCG '03: Proceedings of the 19th Annual Symposium on Computational Geometry, pp. 211–219. ACM, New York, NY, USA (2003)
2. Beirouti, R.: A fast heuristic for finding the minimum weight triangulation. Tech. Rep., Vancouver, BC, Canada (1997)
3. Boissonnat, J.D., Teillaud, M.: On the randomized construction of the Delaunay tree. Theor. Comp. Sci. **112**(2), 339–354 (1993)
4. Buchin, K.: Incremental construction along space-filling curves. In: EuroCG'05: Proceedings of the 21th European Workshop on Computational Geometry, pp. 17–20 (2005)
5. Dæhlen, M., Fimland, M., Hjelle, Ø.: A Triangle-based Carrier for Geographical Data, pp. 105–120. Taylor and Francis, New York (2001)
6. Devillers, O.: The Delaunay hierarchy. Int. J. Found. Comput. Sci. **13**, 163–180 (2002)
7. Devillers, O., Pion, S., Teillaud, M.: Walking in a triangulation. In: Proceedings of the 17th Annual Symposium on Computational Geometry, pp. 106–114 (2001)
8. Devroye, L., Mucke, E.P., Zhu, B.: A note on point location in Delaunay triangulations of random points. Algorithmica **22**(4), 477–482 (1998)
9. Drysdale, R.L.S., Rote, G., Aichholzer, O.: A simple linear time greedy triangulation algorithm for uniformly distributed points. Report IIG-408, Institutes for Information Processing. Technische Universit at Graz (1995)
10. Floriani, L.D., Falcidieno, B., Nagy, G., Pienovi, C.: On sorting triangles in a Delaunay tessellation. Algorithmica **6**(4), 522–532 (1991)
11. Green, P.J., Sibson, R.: Computing Dirichlet tessellations in the plane. Comput. J. **21**(2), 168–173 (1978)
12. Hansford, D.: The neutral case for the min-max triangulation. Comput. Aided Geom. Des. **7**(5), 431–438 (1990)
13. Höhle, J., Oster Pedersen, C., Bayer, T., Frederiksen, P.: The photogrammetric derivation of digital terrain models in built-up areas. Photogramm. J. Finl. **22**(1), 33–45 (2010)
14. Koch, A.: An integrated semantically correct 2.5 dimensional object oriented TIN. In: Proceedings of the 1st International Workshop on Next Generation 3D City Models (2005)
15. Kolingerová, I.: A small improvement in the walking algorithm for point location in a triangulation. In: Proceedings of the 22nd European Workshop on Computational Geometry, pp. 221–224 (2006)
16. Lawson, C.L.: Mathematical Software III; Software for C1 Surface Interpolation, pp. 161–194. Academic, New York (1977)
17. Mehlhorn, K., Näher, S.: Leda: a platform for combinatorial and geometric computing. Commun. ACM **38**(1), 96–102 (1995)
18. Mücke, E.P., Saias, I., Zhu, B.: Fast randomized point location without preprocessing in two and three-dimensional Delaunay triangulations. In: Proceedings of the 12th Annual Symposium on Computational Geometry, vol. 26, pp. 274–283 (1996)
19. Mulmuley, K.: Randomized multidimensional search trees: dynamic sampling. In: Proceedings of the 7th Annual Symposium on Computational Geometry, pp. 121–131 (1991)
20. Owens, K., Parikh, R.: Fast random number generator on the Intel Pentium 4 processor. Intel Software Network (2009)
21. Pearson, K.: Notes on the history of correlation. In: Royal Society Proceedings, pp. 25–45 (1920)
22. Purchart, V., Kolingerová, I., Beneš, B.: Interactive sand-covered terrain surface model with haptic feedback. In: GIS Ostrava 2012 - Surface Models for Geosciences, pp. 215–223 (2012)
23. Schilling, A., Basanow, J., Zipf, A.: Vector based mapping of polygons on irregular terrain meshes for web 3D map services. In: 3rd International Conference on Web Information Systems and Technologies (WEBIST), Barcelona (2007)
24. Schilling, A., Lanig, S., Neis, P., Zipf, A.: Integrating terrain surface and street network for 3d routing. In: 3D Geo-Information Sciences, Lecture Notes in Geoinformation and Cartography, pp. 109–126. Springer, Berlin Heidelberg (2009)
25. Shewchuk Jonathan, R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. Discrete Comput. Geom. **18**, 305–363 (1996)
26. Sloan, S.W.: A fast algorithm for constructing Delaunay triangulations in the plane. Adv. Eng. Softw. **9**(1), 34–55 (1987)
27. Soukal, R., Kolingerová, I.: Straight walk algorithm modification for point location in a triangulation. In: EuroCG'09: Proceedings of the 25th European Workshop on Computational Geometry, pp. 219–222. Brussels, Belgium (2009)
28. Soukal, R., Kolingerová, I.: Star-shaped polyhedron point location with orthogonal walk algorithm. Procedia CS **1**(1), 219–228 (2010)
29. Su, P., Drysdale, R.L.S.: A comparison of sequential Delaunay triangulation algorithms. In: Proceedings of the 11th Annual Symposium on Computational Geometry, pp. 61–70 (1995)
30. Sundareswara, R., Schrater, P.: Extensible point location algorithm. In: International Conference on Geometric Modeling and Graphics, pp. 84–89 (2003)
31. Vasa, L., Skala, V.: A perception correlated comparison method for dynamic meshes. IEEE Trans. Vis. Comput. Graph. **17**(2), 220–230 (2011)
32. Vigo, M., Pla, N., Cotrina, J.: Regular triangulations of dynamic sets of points. Comput. Aided Geom. Des. **19**(2), 127–149 (2002)
33. Weller, F.: On the total correctness of Lawson's oriented walk. In: Proceedings of the 10th International Canadian Conference on Computational Geometry, pp. 10–12 (1998)
34. Zadravec, M., Žalik, B.: An almost distribution independent incremental Delaunay triangulation algorithm. Vis. Comput. **21**(6), 384–396 (2005)

35. Zemek, M., Kolingerová, I.: Hybrid algorithm for deletion of a point in regular and delaunay triangulation. In: Proceedings of the Spring Conference on Computer Graphics, pp. 137–144. Budmerice, Slovakia (2009)
36. Zhou, S., Jones, C.B.: HCPO: an efficient insertion order for incremental Delaunay triangulation. Inf. Process. Lett. **93**(1), 37–42 (2005)
37. Zhu, B.: On lawsons oriented walk in random delaunay triangulations. In: Lingas, A., Nilsson, B. (eds.) Fundamentals of Computation Theory. Lecture Notes in Computer Science, vol. 2751, pp. 222–233. Springer, Berlin (2003)
38. Žalik, B., Kolingerová, I.: An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. Int. J. Geogr. Inf. Sci. **17**(2), 119–138 (2003)

Appendix F

Star-shaped polyhedron point location with orthogonal walk algorithm

Soukal, R., Kolingerová, I.

Procedia Computer Science, Volume 1, Issue 1, pp. 219-–228, Elsevier (2010), ISSN 1877–0509

Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 1 (2012) 219–228
Procedia Computer Science
www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

Star-shaped polyhedron point location with orthogonal walk algorithm

Roman Soukal[a], Ivana Kolingerová[a]

[a] *Faculty of Applied Sciences, University of West Bohemia, Univerzitni 8, Plzen 306 14, Czech Republic*

Abstract

The point location problem is one of the most frequent tasks in computational geometry. The walking algorithms are one of the most popular solutions for finding an element in a mesh which contains a query point. Despite their suboptimal complexity, the walking algorithms are very popular because they do not require any additional memory and their implementation is simple. This paper describes the modifications of two walking algorithms for point location on a surface of a star-shaped polyhedron, a generalization of the Remembering Stochastic walk algorithm for a star-shaped polyhedron and a modification of the planar Orthogonal walk algorithm. The latter uses spherical coordinates to transfer the spatial point location problem to the planar point location problem. This way, the problem can be solved by the traditional planar algorithms. Along with the modifications, the paper proposes new methods for finding a proper starting triangle for the walking process with or without preprocessing.

Keywords: walking algorithms, spherical point location, star-shaped polyhedron, orthogonal walk, spherical coordinate system

1. Introduction

Finding which polygon in a mesh contains a query point is a frequent task in computational geometry. For a query point q and a given triangulation T of n vertices in the plane the planar point location problem usually means how to find a triangle ω from T which contains q.

The algorithms solving this problem can be divided into two groups: algorithms using sophisticated data structures and so called *walking algorithms*. The former concentrates on achieving the lowest complexity possible, in this case $O(\log n)$ per point query which is achieved by using sophisticated data structures such as DAG [1, 2], skip list [3], quad tree, buckets [4], uniform grid [5, 6] and data structures based on random sampling [7, 8]. Despite their low complexity, these algorithms have some disadvantages. First, the data structures listed above consume generally $O(n)$ amount of memory which may be a problem for huge datasets. Second, the implementation effort for most of these structures is nontrivial (especially for modifications of these structures). Finally, most of these structures are hierarchical and the top level of the hierarchy may become a bottleneck in case of parallelization.

Walking algorithms do not need any extra memory, their implementation is rather simple and their usability for parallelization is good, thus often they are a better choice than the optimal time complexity solutions. The name of

Email address: soukal@kiv.zcu.cz (Roman Soukal)

1877-0509 © 2012 Published by Elsevier Ltd.
doi:10.1016/j.procs.2010.04.025

these algorithms has arisen from the way of locating the triangle ω which contains q. From a starting triangle α chosen as one of the triangles of T and the query point q the walking strategy makes use of connectivity of the triangle mesh to go through triangles between α and ω.

In a higher dimension, the definition of a general spatial point location problem is not so straightforward. We concentrated on the point location on a star-shaped polyhedron surface. For a star-shaped polyhedron P, its surface triangulation T of n vertices and a center point c, the spatial point location problem of a query point q usually means how to find a triangle ω from T which is intersected by the ray cq. Note that the center point c of P with surface triangulation T is such a point inside P that each vertex of T is directly visible from c. In the following text, we assume c is the part of input.

The star-shaped polyhedron point location is often used for a spherical point location but it is not limited to this use. Its main application is in spherical remeshing methods [9, 10, 11]. Here, the surface triangulation T is an original irregular mesh parametrized onto the unit sphere using a spherical parametrization [12, 13, 14, 15] and T' is a regular spherical mesh. During the sampling process, for each vertex q of T', it is necessary to find the triangle ω from T which contains q. Apparently, it is a star-shaped polyhedron point location problem.

In this paper we present two walking algorithms for point location on the surface of star-shaped polyhedron. The former is a simple modification of the planar Remembering Stochastic walk algorithm [16]. The latter is a modification of the Orthogonal walk algorithm [16]. This modification uses planar point location using the simplification of spherical coordinate system.

The paper is organized as follows. Section 2 presents the existing planar walking algorithms and the only published walking algorithm (as far as we know) for point location on a spherical surface. Section 3 describes our Remembering Stochastic walk algorithm for point location on the surface of a star-shaped polyhedron. Section 4 presents our idea of using the spherical coordinate system to enable the point location with the existing planar walking techniques. Planar point location in spherical coordinates brings some difficulties which are discussed in this section. In Section 5, a modification of the planar Orthogonal walk algorithm for point location in spherical coordinates is presented. The possible preprocessing methods for the algorithm in Section 5 are proposed in Section 6. The empirical results of all our modifications are presented in Section 7.

2. Walking algorithms

There are three main types of the walking strategies. The visibility walk makes use of point-inside-triangle tests to determine which triangle is the next [17, 18, 19, 20]. The straight walk passes all such triangles in the mesh between α and ω that are intersected by a line pq where p is a point inside α [21, 16]. The orthogonal walk passes all the triangles in the mesh between α and ω in the directions of coordinate axes [16]. The complexity of the walking algorithms is suboptimal from $O(\sqrt[3]{n})$ up to $O(\sqrt{n})$ [16]. The starting triangle α may be chosen randomly, by the use of hierarchical structures or as the closest triangle to q from a set A of randomly chosen triangles from T, $\|A\| \ll \|T\|$ [18]. A proper choice rapidly improves the speed of the algorithm.

Wu at al.[22] proposed a spherical modification of a planar location algorithm by Sundareswara at al.[19] which uses barycentric coordinates to find the triangle ω. The main idea of this variant of visibility walk algorithm is to compute the barycentric coordinates of q in the current triangle τ to determine which neighbor triangle is closer to q and will be the next to visit. Wu at al. also proposed the choice of the first triangle using the subdivision of the regular octahedron. The disadvantage of Wu's algorithm is its limitation to a spherical surface.

3. Star-shaped Polyhedron Point Location Using Remembering Stochastic Walk

Lawson's Oriented walk [17] is a very popular planar visibility walk algorithm which uses planar orientation edge test (Equation 1) to determine which triangle is the next in its walk. Simple Lawson's Oriented walk algorithm uses edges of τ for tests in a given order, but this method may loop for a non-Delaunay triangulation [23, 24]. For such a triangulation it is necessary to choose the tested edges of τ in a random order. This modification is called *Stochastic*[16]. Furthermore, it is not necessary to test the edge incident with the previous triangle in the walk. This improvement is called *Remembering*[16] and it may save up to one orientation test for each triangle. Therefore, one or two orientation tests are needed for each triangle (except α).

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

$$orientation3D(t,u,v,w) = sgn\left(\begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix}\right) \quad (2)$$

We propose a modification of planar Remembering Stochastic walk algorithm as another possibility for point location on a star-shaped polyhedron. Our modification uses the center point c of the polyhedron. Instead of the classical edge test (Equation 1) which is used in the planar point location, we use spatial orientation facet test (Equation 2) which is used for walking in tetrahedral meshes. The decision whether or not the edge rl of the current triangle τ should be crossed to continue the walk into the next triangle depends on the result after substitution q, c, r, l to the Equation 2 where q is the query point, c is the center point and r, l are vertices of τ. Assuming that the vertices of the triangle are in the CCW order in left-handed coordinate system, the walk continues to the next triangle over the edge rl if the $orientation3D(q,c,r,l) = 1$. If the $orientation3D(q,c,v,w) \leq 0$ for all edges of τ, the triangle τ contains the query point q. In the proposed algorithm, we use simple idea which allows walking on the triangulated surface of a general star-shaped polyhedron, in contrast to Wu's barycentric walk algorithm [22] which allows walking only on a spherical surface. In the following text, we use the term *spatial algorithms* for this algorithm and Wu's barycentric walk algorithm for spherical location [22].

4. Point Location in Spherical Coordinates

In this section, we present a technique which uses spherical parametrization and allows planar point location on the surface of a star-shaped polyhedron. This technique is faster then spatial algorithms but brings some difficulties, where planar algorithm does not return correct output, so it cannot be used separately. However, it serves well to find a triangle close to the correct triangle. The search is then finished by one of spatial algorithms.

The star-shaped polyhedron triangulation mesh T consists of an array of vertices V and an array of faces (triangles) F. Each element $f = \{v_i, v_j, v_k\} \in F$ contains indexes of its three vertices $v_i, v_j, v_k \in V$ and of its neighbor triangles $f_m, f_n, f_o \in F$ where $v_j v_k$ is the edge shared with (its neighbor) triangle f_m, $v_k v_i$ is the edge shared with f_n and $v_i v_j$ is the edge shared with f_o. Each vertex v_i of V can be denoted as a pair $v_i = (p_i, h_i)$, where $p_i = \{x_i, y_i, z_i\}$ is a triple of Cartesian coordinates and $h_i = \{\varphi_i, \theta_i\}$ is a pair of spherical radian coordinates. The spherical coordinates $\{\varphi_i, \theta_i\}$ of v_i are computed from Cartesian coordinates using Equation 3. Note that $c = \{x_c, y_c, z_c\}$ is the center point of a star-shaped polyhedron and the range of $arctg2$ function is defined as $(-\pi, \pi\rangle$.

$$\varphi_i = arctg2(y_i - y_c, x_i - x_c), \theta_i = arccos\left(\frac{z_i - z_c}{\sqrt{(x_i - x_c)^2 + (y_i - y_c)^2 + (z_i - z_c)^2}}\right) \quad (3)$$

The query point q is given by either Cartesian or spherical coordinates. Assuming that the spherical coordinates $\{\varphi, \theta\}$ are planar coordinates of points and vertices, we can use normal planar walking algorithms. Note that in the following text, the third spherical coordinate r (radius) is ignored and we use the term *planar walk* for walking in spherical coordinates, where we use φ instead of x and θ instead of y part of planar coordinates. However, to make use of standard planar algorithms possible, we represented the edges of the model in spherical coordinates as line segments, projecting only their endpoints. This way we obtain a standard planar triangulation, usable for planar walking algorithms without any need of changes. This simplification brings some difficulties as follows.

The problem lies in the fact that spherical coordinate system is a curvilinear coordinate system [25] and the line segment between two points in Cartesian coordinates is an arc in spherical coordinates. It contradicts to our simplification where we represented the edges in spherical coordinates as line segments, projecting only their endpoints. Figure 1 shows three examples, where the orientation test in spherical coordinates produced incorrect results regarding the original position in Cartesian coordinates. This problem is shown on three levels of subdivision of an icosahedron. Figure 1a shows the original icosahedron in spherical coordinates, Figures 1b, 1c its first and second level of subdivision. A point d lies originally on an edge e (Figure 1b) of the icosahedron, but in spherical coordinates, it may lie outside the edge (Figure 1a). The edge e and its subdivisions are bold and colored red. In Cartesian coordinates, a

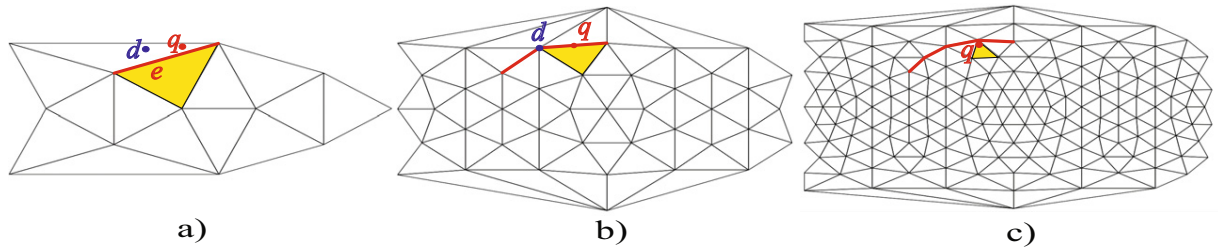

Figure 1: Problem of the planar orientation edge test in our simplification of spherical coordinates (the icosahedron (a) and its first (b) and second (c) level of subdivision, red color represents the surface subdivisions of the edge e)

point q is located in a triangle which is colored yellow in Figures 1a, b, c, but in spherical coordinates it may lie on an edge (Figure 1b) or even in a different triangle (Figure 1a).

Hence our simplification is not geometrically correct and the planar orientation edge test (Equation 1) in the spherical coordinates occasionally returns incorrect results. The probability of incorrect results goes down with higher density of mesh, but not to zero. However, the triangle returned from the planar point location is always very close to the correct one, thus planar walking algorithms in spherical coordinates are a good choice for fast location of a proper starting triangle for slower, but precise spatial algorithms. In most cases the final location with a spatial algorithm will be very short (see Section 7).

For better readability, the *border* triangles (triangles whose vertices lie on the opposite sides in our simplification of spherical coordinates) are not displayed in all planar figures, except Figure 2 (see Figure 2a where these triangles are colored red and one chosen border triangle is highlighted by green). All types of the planar walking strategies sometimes fail on such triangles and may loop. For bigger datasets, cases where the walk goes over these border triangles are very rare and they appear only if α is chosen as one of the border triangles or one such triangle contains q (see Figure 2b) or q is near to it. Hence if the planar walk detects that the current triangle τ is border, the planar location ends and ω is located by one of spatial algorithms.

Border triangles are recognized and flagged during the computing of spherical coordinates. The detection is rather simple. We substitute the spherical coordinates h_i, h_j, h_k of the vertices v_i, v_j, v_k of the triangle $\tau = v_i, v_j, v_k$ to the planar orientation test from Equation 1. The result of the $orientation2D(h_i, h_j, h_k)$ is opposite for the border triangle than for the other triangles. Assuming that the vertices of the triangle are in CCW order on the surface of a polyhedron in left-handed coordinate system, the triangle τ is the border triangle if the $orientation2D(h_i, h_j, h_k) = 1$. Note that we use φ coordinate instead of x and θ instead of y in test from Equation 1.

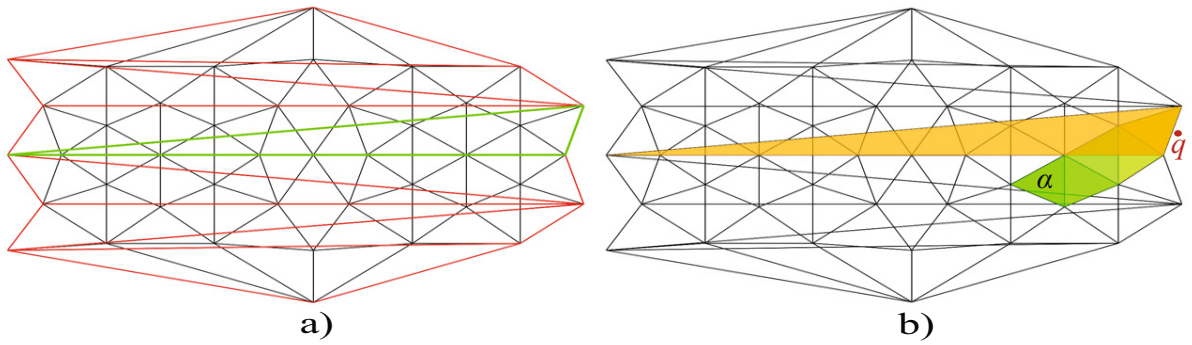

Figure 2: Planar triangulation of the icosahedron with shown border triangles colored by red (a) and with location of point in the border triangle (b)

5. Orthogonal Walk Algorithm in Spherical Coordinates

In this section, we present a modification of an orthogonal walk algorithm for our simplification of spherical coordinates. The planar Orthogonal walk was proposed by Devillers at al. [16] with an idea of cheaper tests. The algorithm goes first in the direction of one coordinate axis and then in the other coordinate axis. Planar orientation tests (Equation 1) were substituted by comparisons of coordinate values. An orthogonal walk is usually longer than other walks (see Figure 3a) but the cost of its tests is significantly lower which results in a faster location.

Devillers's Orthogonal walk reliably finds a proper location of q but from Section 4 we know that we need the orthogonal walk to find only the approximate location of q, thus we can optimize the algorithm for time. If the returned triangle is close to the proper triangle ω which contains q, it is enough. The original Orthogonal walk [16] uses three comparisons for each visited triangle, computes few planar orientation tests (Equation 1) and contains quite a difficult initialization step. Our modification (see Algorithm 1) simplifies the initialization step, uses only two comparisons for each visited triangle and does not use the planar orientation tests.

Now let us explain our modification. The orthogonal walk algorithm starts in a triangle α (the proper choice is explained in Section 6). First, the algorithm must choose a starting point $a = \{a_\varphi, a_\theta\}$ anywhere inside α. Let us assume $q = \{q_\varphi, q_\theta\}$ is above and to the right of a ($q_\theta > a_\theta, q_\varphi > a_\varphi$), other three possibilities would be analogous. In the initialization step we choose $s = \{s_\varphi, s_\theta\}$ as a vertex of α with the maximal horizontal (φ) value. The vertices s, r, l of α are in CCW order.

Now the walk in φ direction may start. For each triangle τ in the orthogonal walk (except α) the edge e of τ is the edge used to cross to τ and s is the vertex of τ facing $e = rl$. The walk leaves τ over the edge f which is found by comparing s_θ with a_θ. If s_θ is lower than a_θ, the walk continues over the edge $f = rs$, τ is the neighbor over f and $l = s$, else the walk continues over the edge $f = ls$, τ is neighbor over f and $r = s$. The new s is chosen from the vertices of τ, $r, l \neq s$. If $s_\varphi > q_\varphi$ or τ is border, the walk in φ direction ends, else it continues for the current τ.

Now we choose s as a vertex of τ with the maximal vertical (θ) value and the walk in θ direction may start. The edge f which is used to go to the next triangle is found by comparing s_φ with q_φ. If s_φ is lower than q_φ, the walk continues over the edge $f = rs$, τ is the neighbor over f and $l = s$, else the walk continues over the edge $f = ls$, τ is the neighbor over f and $r = s$. The new s is chosen from the vertices of τ, $r, l \neq s$. If $s_\theta > q_\theta$ or τ is a border triangle, the orthogonal walk ends. Now τ is close to the triangle ω which contains the query point q and the final location is made by the Remembering Stochastic walk from Section 3.

Figure 3 shows an example of our orthogonal walk on the surface of a icosahedron in the first level of subdivision. The triangle β is a triangle where the horizontal walk stops and vertical walk begins. The triangle γ is the final triangle of our orthogonal walk and the first triangle of the final spatial location. Figure 3a shows the walk in our simplification of spherical coordinates and Figure 3b shows the walk in the Cartesian coordinates.

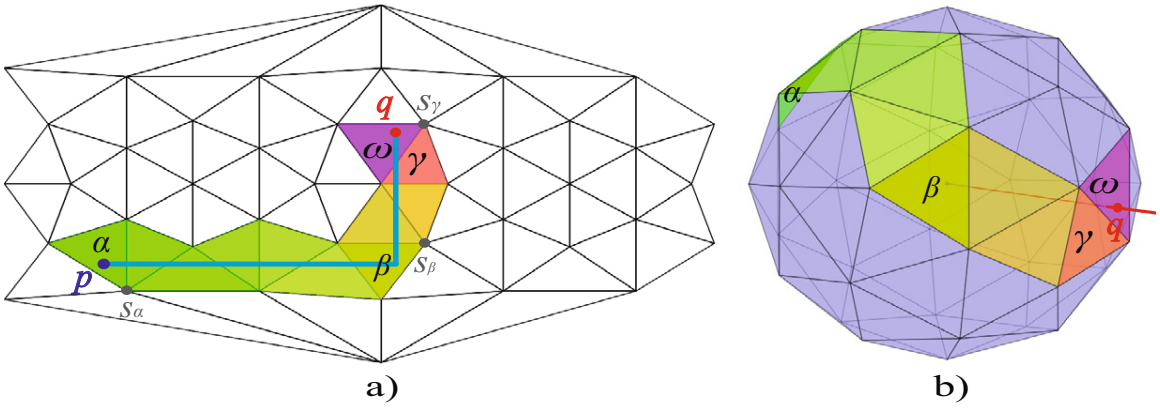

Figure 3: Our orthogonal walk algorithm in the spherical (a) and the Cartesian (b) coordinates

6. Preprocessing

A very important part of a walking algorithm is the choice of the starting triangle which may improve the speed of the algorithm. The easiest way is to choose the starting triangle α randomly or as the triangle of a planar mesh which contains a point in the middle of this mesh. Mücke et al. proposes the way how to choose a good starting triangle without preprocessing[18] where α is chosen as the closest triangle to q from a set A of randomly chosen triangles from the mesh T, $||A|| \ll ||T||$. For the best performance, Mücke recommends $||A|| = 2\sqrt[4]{n}$ and our results confirmed that. But if we choose the starting triangle α as the triangle containing the point b which is in the middle of the spherical domain ($b_\varphi = 0, b_\theta = 0.5\pi$), the performance is very similar to [18].

Input:

- the query point q
- the chosen starting triangle $\alpha, \alpha \in T$

Output:

- the triangle ω which contains the query point q

a = a point generated anywhere inside α;
// we describe the case where q is above and to the right of a ($q_\theta > a_\theta, q_\phi > a_\phi$), other cases are analogous
$\tau = \alpha =$ srl where s is the vertex with maximal φ coordinate;
// note that the vertices of τ (s, r, l) are always in CCW order

// traverses the triangulation T in the direction of the horizontal axis of φ
while $s_\varphi < q_\varphi$ *and notBorder*(τ) **do**
  if $s_\theta < a_\theta$ **then**
    τ = neighbor of τ over rs;
    $l = s$;
  else
    τ = neighbor of τ over ls;
    $r = s$;
  end
  s = vertex of τ where $s \neq r, s \neq l$;
end

$\tau =$ srl where s is the vertex with maximal θ coordinate;

// traverses the triangulation T in the direction of the vertical axis of θ
while $s_\theta < q_\theta$ *and notBorder*(τ) **do**
  if $s_\varphi < q_\varphi$ **then**
    τ = neighbor of τ over rs;
    $l = s$;
  else
    τ = neighbor of τ over ls;
    $r = s$;
  end
  s = vertex of τ where $s \neq r, s \neq l$;
end

// the final location is done by another walking algorithm (e. g. Remembering Stochastic walk in Section 3)
ω = SpatialRememberingStochasticWalk(q, τ);

return ω;

Algorithm 1: Our modification of the Orthogonal walk

At the cost of additional memory, we can improve performance of our algorithm in the following way. The advantage of spherical coordinates is the known range of φ and θ values and it can be used to find a suitable starting triangle for our orthogonal walk algorithm using a grid. For each cell g_{ij} of the grid, the suitable starting triangle α_{ij} is triangle which contains the center point q_{ij} of g_{ij}. For the polyhedron whose triangles are very similar, each cell g_{ij} of the uniform grid (see Figure 4a) contains a different number of these similar triangles, especially near poles, the triangles are very wide and the number of triangles in these cells is much lower.

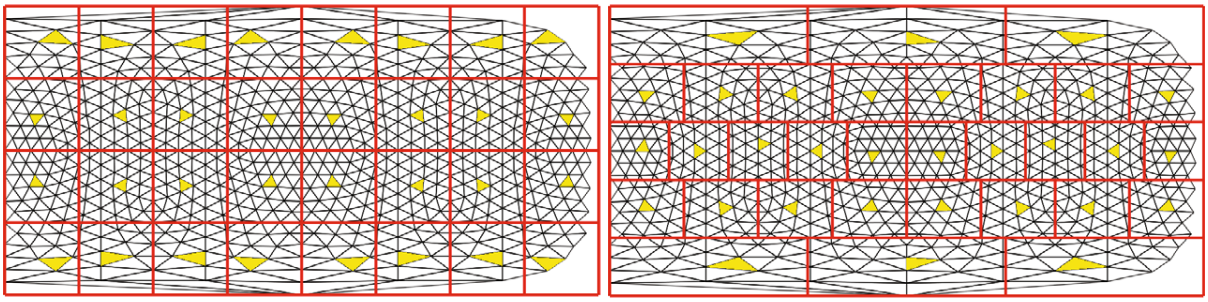

Figure 4: The uniform and the nonuniform grid of 32 cells for finding a suitable starting triangle (the starting triangle for each cell of grid is yellow colored)

This consideration leads us to use of a nonuniform grid preserving the character of the spherical projection. The grid is nonuniformly subdivided only in the φ direction (the planar triangulation is divided uniformly to k longitudinal strips and each strip G_i is divided vertically to l_i cells - see Figure 4b). Each cell g_{ij} contains such triangles of T that their area of these triangles on the spherical surface is similar for all g_{ij}. Assuming that the surface of a unit sphere ($r = 1$) can be approximated by the function $sin(\theta)$ in the plane, the spherical surface S_i equivalent to a planar longitudinal strip G_i for $\theta \in \langle a, b\rangle$ in the spherical coordinates can be computed as the area of the planar strip bounded by the functions $f(\varphi) = 2\pi sin(\theta), \theta \in \langle a, b\rangle$ (see Equation 4 and Figure 5 - the equations are derived from the definition of determinant of Jacobian matrix for spherical coordinate parametrization and from the range of φ coordinate). The surface S of the unit sphere can be computed as $S = 2\pi\,(cos(0) - cos(\pi)) = 4\pi$. Given m is a number of cells of the nonuniform grid, k is the number of longitudinal strips and l_i is the number of cells in each longitudinal strip G_i, Equations 5, 6 describe the computation of k and l_i. Figure 4 shows grid structures for the choice of the first triangle where the first triangle for each cell of the grid is colored yellow. Figure 4a shows a uniform grid of 32 cells and Figure 4b shows the nonuniform grid with the same number of cells. The matching cell g_{ij} of a query point $q = \{\varphi_q, \theta_q\}$ is computed identically (see Equations 7) for uniform and nonuniform grid (in the uniform grid, l_i is the same for all i). Note that if $\theta_q = \pi$ then $i = k - 1$ or if $\varphi_q = \pi$ then $j = l_i - 1$.

$$S_i = 2\pi \int_a^b f(\theta)d\theta = 2\pi \int_a^b sin(\theta)d\theta = 2\pi\,[-cos(\theta)]_a^b = 2\pi\,(cos(a) - cos(b)) \tag{4}$$

$$k = \left\lfloor \sqrt{\frac{\pi m}{4}} + 0.5 \right\rfloor \tag{5}$$

$$l_i = \left\lfloor \frac{mS_i}{S} + 0.5 \right\rfloor = \left\lfloor \frac{1}{2} m \int_{\frac{i\pi}{k}}^{\frac{(i+1)\pi}{k}} sin(\theta)d\theta + 0.5 \right\rfloor, i = 0, 1, ..., k - 1 \tag{6}$$

$$i = \left\lfloor k\frac{\theta_q}{\pi} \right\rfloor, j = \left\lfloor l_i \frac{\varphi_q + \pi}{2\pi} \right\rfloor \tag{7}$$

7. Experimental Results

We tested the following algorithms: Remembering Stochastic walk presented in Section 3 (RSW), Wu's Barycentric walk [22] (WBW) and our orthogonal walk (OW). Tests were performed on sixteen different datasets (a real parametrized models, subdivisions of regular polyhedra (tetrahedron, octahedron, icosahedron), a randomly generated star-shape polyhedra). The results correspond to the sizes of datasets and do not differ too much for different datasets of the same size, therefore the results will be illustrated on the following datasets: Headus Skull, Stanford Bunny, an icosahedron in the 7th level of subdivision and a randomly generated star-shape polyhedra with 10^5 vertices. 10^6 randomly generated points were located by each algorithm on each dataset.

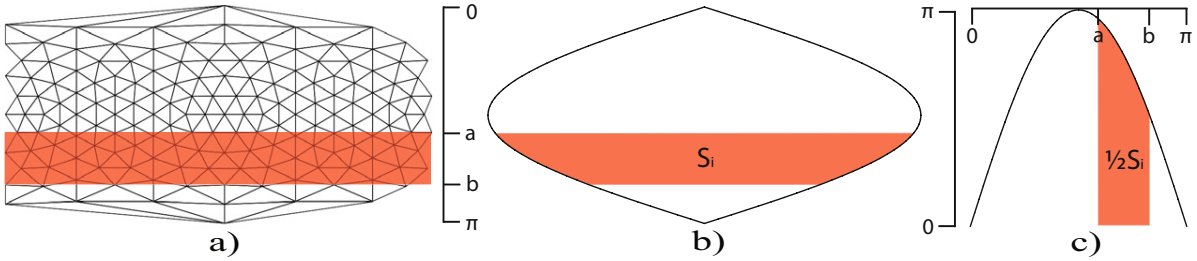

Figure 5: An illustrative example to computing an area of the strip (by red) in our simplification of spherical coordinates (a), its real surface area on the unit sphere (b) adjusted for easier computation (c)

Selected results are in Table 1 (without preprocessing) and Table 2 (with preprocessing - see Section 6). The following properties were examined for each algorithm: the average length of the walk (#Δ), the average number of the tests (#test) and the average time (t[μs]) per one location (tested on Intel Q6600 2,40GHz). The properties #test and #Δ for OW consists of two values, the former value concerns the orthogonal walk and the latter one concerns the final location by RSW. In Table 2, we tested RSW with the uniform and the nonuniform grid and WBW, where we choose a starting triangle using n levels of subdivision of a regular octahedron [22]. The properties #test and #Δ for WBW consist of two values, where the former value concerns the choice of a starting triangle and the latter concerns the final location. In Table 2, we compare such algorithms that have the same number of elements (i. e., for WBW, the number of triangles in the lowest level of subdivision of octahedron is the same as the number of grid cells used by RSW). The algorithms were coded in Java with double precision floating point arithmetic.

Algorithm	**#Δ**	**#test**	**t[μs]**	**#Δ**	**#test**	**t[μs]**
	ϕ **per located point**			ϕ **per located point**		
	Headus Skull (40000 Δ, 20002 vertices)			Stanford Bunny (71882 Δ, 35943 vertices)		
WBW	117.5	352.6	20.46	167.0	501.1	34.11
RSW	135.2	233.1	26.49	188.1	323.4	43.40
OW	159.1+2.1	318.1+5.1	11.23	240.9+2.1	481.8+5.2	21.04
	Icosahedron (7th level, 327680 Δ, 163842 vertices)			Star-shaped polyhedron (199996 Δ, 10^5 vertices)		
WBW	361.2	1083.5	79.11	N/A	N/A	N/A
RSW	392.3	680.7	110.48	324.4	559.3	97.22
OW	515.1+1.93	1030.2+4.8	44.87	409.0+2.64	818.0+6.0	42.66

Table 1: Comparison of algorithms without preprocessing

To sum up the results without preprocessing (Table 1), the RSW is about 30% slower than WBW, but it can be used for a general star-shaped polyhedron, therefore we use it for the final location in OW. The OW is almost twice as fast as WBW and the time of the final location by RSW is not significant because its walk is usually very short in average (see Table 1). To sum up the results with preprocessing, for the same number of elements (see above), the OW is evidently faster than WBW and the grid is more memory-economical than the octahedron hierarchy used in [22]. The nonuniform grid is faster than the uniform grid but the difference is not great.

8. Conclusion and Future work

We presented two new walking algorithms for the point location on triangulated surface of general star-shaped polyhedron. The best performance is achieved by using both these algorithms together and the proposed solution is faster than other walking solutions. We also presented how to find a suitable starting triangle by using a uniform and a nonuniform grid in $O(1)$ time. This choice further improves the performance of our algorithms at the cost of additional memory.

Algorithm	#Δ	#test	t[μs]	#Δ	#test	t[μs]
	ϕ per located point			ϕ per located point		
128 elements	Stanford Bunny (71882 Δ, 35943 vertices)			Icosahedron (7th level, 327680 Δ, 163842 vertices)		
WBW (2nd level)	6.2+14.3	18.7+43.0	4.58	6.2+31.6	18.7+94.9	10.47
OW (uniform grid)	17.5+1.8	35.0+4.5	3.55	39.8+1.85	79.7+4.65	8.86
OW (nonuniform grid)	14.9+1.7	29.8+4.4	3.02	35.6+1.6	71.2+4.2	7.81
512 elements	Stanford Bunny (71882 Δ, 35943 vertices)			Icosahedron (7th level, 327680 Δ, 163842 vertices)		
WBW (3rd level)	8.0+7.8	23.9+23.3	3.50	8.0+16.4	23.9+49.1	6.49
OW (uniform grid)	8.2+1.7	16.5+4.4	2.28	19.4+1.6	38.8+4.1	4.96
OW (nonuniform grid)	7.1+1.6	14.3+4.3	2.10	17.4+1.6	34.8+4.1	4.61
2048 elements	Stanford Bunny (71882 Δ, 35943 vertices)			Icosahedron (7th level, 327680 Δ, 163842 vertices)		
WBW (4th level)	9.7+4.4	29.2+13.2	2.97	9.7+8.7	29.2+26.2	4.60
OW (uniform grid)	3.8+1.7	7.7+4.5	1.63	9.3+1.6	18.6+4.1	3.05
OW (nonuniform grid)	3.3+1.7	6.6+4.3	1.49	8.2+1.7	16.5+4.4	2.90

Table 2: Comparison of algorithms with preprocessing

As a future work, we would like to focus on point location in tetrahedral meshes, where we expect a practical application in dynamic proteins research.

Acknowledgments

We would like to thank Miss Málková, Mr. Váša and Mr. Hlaváček for their feedback and inspiring discussions. This work is supported by the Grant Agency of the Czech Republic - the project 201/09/0097.

References

[1] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, Computational Geometry, Algorithms and Applications, Berlin Heidelberg: Springer, 1997.
[2] I. Kolingerová, B. Žalik, Improvements to randomized incremental Delaunay insertion, Computers & Graphics 26 (2002) 477–490.
[3] M. Zadravec, B. Žalik, An almost distribution independent incremental Delaunay triangulation algorithm, The Visual Computer 21 (6) (2005) 384–396.
[4] P. Su, R. L. S. Drysdale, A comparison of sequential Delaunay triangulation algorithms, in: Proceedings of the 11th Annual Symposium on Computational Geometry, 1995, pp. 61–70.
[5] S. W. Sloan, A fast algorithm for constructing Delaunay triangulations in the plane, Advanced Engineering Software 9 (1) (1987) 34–55.
[6] B. Žalik, I. Kolingerová, An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm, International Journal of Geographical Information Science 17 (2) (2003) 119–138.
[7] K. Mulmuley, Randomized multidimensional search trees: Dynamic sampling, in: Proceedings of the 7th Annual Symposium on Computational Geometry, 1991, pp. 121–131.
[8] O. Devillers, Improved incremental randomized Delaunay triangulation, in: Proceedings of the 14th Annual Symposium on Computational Geometry, 1998, pp. 106–115.
[9] L. Kobbelt, J. Vorsatz, U. Labsik, H. P. Seidel, A shrink wrapping approach to remeshing polygonal surfaces, Computer Graphics Forum, Eurographics '99 (18) (1999) 119–130.
[10] Hormann, U. Labsik, G. Greiner, Remeshing triangulated surfaces with optimal parameterizations, Computer-Aided Design 33 (11) (2001) 779–788.
[11] E. Praun, H. Hoppe, Spherical parametrization and remeshing, in: Proceedings of the 30th Annual Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH, 2003, pp. 340–349.
[12] A. Certain, J. Popovic, T. DeRose, D. S. T. Duchamp, W. Stuetzle, Interactive multiresolution surface viewing, in: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH, 1996, pp. 91–98.
[13] U. Labsik, L. Kobbelt, R. Schneider, H. P. Seidel, Progressive transmission of subdivision surfaces, Computational Geometry 15 (2000) 25–39.
[14] S. Saba, I. Yavneh, C. Gotsman, A. Sheffer, Practical spherical embedding of manifold triangle meshes, in: International Conference on Shape Modeling and Applications, 2005, pp. 340–349.
[15] K. Hormann, B. Lévy, A. Sheffer, Mesh parameterization: Theory and practice, in: ACM SIGGRAPH Course Notes, 2007.
[16] O. Devillers, S. Pion, M. Teillaud, Walking in a triangulation, in: Proceedings of the 17th Annual Symposium on Computational Geometry, 2001, pp. 106–114.
[17] C. L. Lawson, Mathematical Software III; Software for C1 Surface Interpolation, Academic Press, New York, 1977, pp. 161–194.

[18] E. P. Mücke, I. Saias, 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, Vol. 26, 1996, pp. 274–283.
[19] R. Sundareswara, P. Schrater, Extensible point location algorithm, in: International Conference on Geometric Modeling and Graphics, 2003, pp. 84–89.
[20] 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, 2006, pp. 221–224.
[21] K. Mehlhorn, S. Näher, Leda: A platform for combinatorial and geometric computing, Communications of the ACM 38 (1) (1995) 96–102.
[22] Y. Wu, Y. He, H. Tian, A spherical point location algorithm based on barycentric coordinates, in: Proceedings of the 5th Computational Science and Its Applications, 2005, pp. 1099–1108.
[23] L. D. Floriani, B. Falcidieno, G. Nagy, C. Pienovi, On sorting triangles in a Delaunay tesselation, Algorithmica 6 (1991) 522–532.
[24] F. Weller, On the total correctness of Lawson's oriented walk, in: Proceedings of the 10th International Canadian Conference on Computational Geometry, 1998.
[25] P. Moon, D. E. Spencer, Field Theory Handbook, Springer-Verlag, 1988.

Appendix G

Surface point location by walking algorithm for haptic visualization of triangulated 3D models

Soukal, R., Purchart, V., Kolingerová, I.

Advances in Engineering Software, Volume 75, pp. 58-–67, Elsevier (2014), ISSN 0965–9978, IF 1.422 (2013)

Advances in Engineering Software 75 (2014) 58–67

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Surface point location by walking algorithm for haptic visualization of triangulated 3D models

Roman Soukal *, Václav Purchart, Ivana Kolingerová

University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering, Plzen, Czech Republic

ARTICLE INFO

Article history:
Received 16 July 2013
Received in revised form 11 May 2014
Accepted 12 May 2014
Available online 17 June 2014

Keywords:
Triangle lookup
Searching algorithm
Walking algorithm
Triangulated surface model
Haptic visualization
Collision detection

ABSTRACT

Haptic devices are nowadays gaining popularity because of their increasing availability. These special input/output devices provide, unlike mouse or keyboard, a native 3D manipulation, especially a more precise control and a force interaction. With more accurate description of the model, haptics can achieve more realistic force feedback. Therefore, triangulated surface models are often used for an authentic interpretation of 3D models. A common task in haptic visualization using triangulated surface models is to find a triangle which is in the collision trajectory of the haptic probe. Since the render rate of the haptic visualization is relatively high (usually about 1 kHz), the task becomes highly non-trivial for complex mesh models, especially for the meshes which are changing over time. The paper presents a fast and novel location algorithm able to find the triangle which is close to the haptic probe and in the direction of the probe motion vector. The algorithm has negligible additional memory requirements, since it does not need additional searching data structures and uses only the information usually available for triangulated models. Therefore, the algorithm could handle even triangular meshes changing over time. Results show that the proposed algorithm is fast enough to be used in haptic visualization of complex-shaped models with hundreds of thousands of triangles.

Introduction

In this paper we focus on the collision detection problem of the haptic device with the surface of 3D model which is defined by a triangular mesh. The goal is to find a triangle (if such a triangle exists) which is in the collision trajectory of the haptic probe to provide appropriate feedback to the user.

Haptic visualization is a tactile feedback method which provides a sense of touch to the user via a haptic device by applying forces, vibrations, or motions while visual perception is usually mediated by a display device. See an example of the haptic device Phantom Omni® used in our experiments in Fig. 1. The haptic visualization finds applications in a variety of areas including haptic surgery simulations [1–3], industry design-based manufacturing [4], or the virtual reality for blind computer users [5]. The majority of the haptic rendering techniques [6,7] require to detect collision (and intersection) of a haptic cursor with the visualized model. However, unlike graphics visualization where a sufficient render rate is about 25 Hz, the render rate required by the haptics is about 1000 Hz to provide an authentic feedback (as it is mentioned by Colgate and Brown [8] a human skin is sensitive to force change of a frequency higher than 500 Hz). Therefore, collisions need to be detected and computed as fast as possible.

Since a higher precision of model representation results in a more realistic perception, many haptic applications [2,6,9,10] use triangulated surface meshes for the representation of models. Existing approaches for the triangulated surface meshes usually deal with static scenes or local changes of the model, where the location methods with search data structures (especially a spatial subdivision techniques) provide adequate results [9]. As mentioned in [10], most of the existing algorithms address collision detection and intersection computation for small models which consist of a few thousands of polygons or they use some kind of down-sampled finite element model [2].

However, in the haptic visualization and interaction used for geometric modeling the model and its topology is often changing, which results in updates of the search data structures. These updates may not be trivial and if the changes in the triangular mesh are frequent, updates of data structures may significantly affect the performance. Moreover, search data structures consume additional memory.

Therefore, the goal is to develop an algorithm which does not utilize additional search data structures and still achieves the

* Corresponding author. Address: University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering, Plzen, Czech Republic. Tel.: +420 377632408, +420 777812230.
E-mail address: soukal@kiv.zcu.cz (R. Soukal).

http://dx.doi.org/10.1016/j.advengsoft.2014.05.004
0965-9978/

Fig. 1. Example of haptic device – Phantom Omni®.

performance necessary for the application in the haptic visualization. When the haptic probe is moving on the surface of the model, the render rate required by the haptics is about 1000 Hz, thus the longest detection/location should take less than 1 ms. When the haptic probe is moving in a free space, the collisions may be detected in a lower rate, since a low delay of the first touch feedback is not noticeable. The frame rate about 100 Hz is sufficient, thus the longest detection/location should take less than 10 ms.

The problem is more precisely defined as follows. For a given position $\mathbf{q}$ of the haptic probe and its motion vector $\mathbf{m}$ (the current direction of movement of the haptic probe), the goal is to find and return such a triangle from the given surface triangular mesh, which is intersected by the line $\lambda = [\mathbf{q}, \mathbf{m}]$ in the distance from $\mathbf{q}$ lower than or equal to a defined maximal allowable distance $dist_{max}$. Moreover, $\mathbf{m}$ should direct towards the front face of the found target triangle. If such a triangle does not exist, *null* should be returned. Note that $dist_{max}$ should be small enough to ensure unique result. We suppose that the model does not contain errors or holes, all the triangles have uniform orientation and have information about their neighbors.

The proposed solution ranks among walking algorithms, which are popular especially for point location in planar triangular meshes or in tetrahedral meshes. The name of walking algorithms describes their principle: generally, the search goes from a triangle to its neighbor in the direction of the given query point, until the target triangle (which contains the query point) is found. In our case, the walking algorithm is searching in the direction of a triangle, which is intersected by the line defined by the position of the haptic probe and the haptic motion vector (see Fig. 2). Since the next triangle is chosen with respect to local tests, the utilization of a walking algorithm for a point location on the triangulated surface model is not simple and, to the best of our knowledge, no complex walking algorithm for point location on a triangulated surface model has been published.

The proposed algorithm has negligible additional memory requirements since it does not need additional data structures. It only needs the information about neighboring triangles, which is usually required for other purposes as well. Therefore, the algorithm can handle even triangular meshes which are changing over time. Although the algorithm was developed especially for haptic visualization, it is not limited to the haptic collision detection only. It can be used for all point location problems, where the input contains both: a point close to the surface of the triangulated 3D model and a vector directing towards the model. For example, for a parametric description of the model, we can get a point on the surface as well as a vector directing towards the model (it may be the opposite surface normal at this point).

Results show that the proposed algorithm can handle queries on rather complex-shaped models with hundreds of thousands of triangles in a good time and thus it can be successfully used in haptic visualization. The algorithm is suitable also for models changing in time. Although it is not a primary task of the algorithm, it can also handle queries when the model is composed of multiple components. Moreover, the algorithm is easily and effectively parallelizable which can significantly speed up the search process.

The paper is organized as follows. 'State of the art' provides an overview in the task of a collision detection regarding to haptic visualization. 'Proposed method' describes the proposed walking

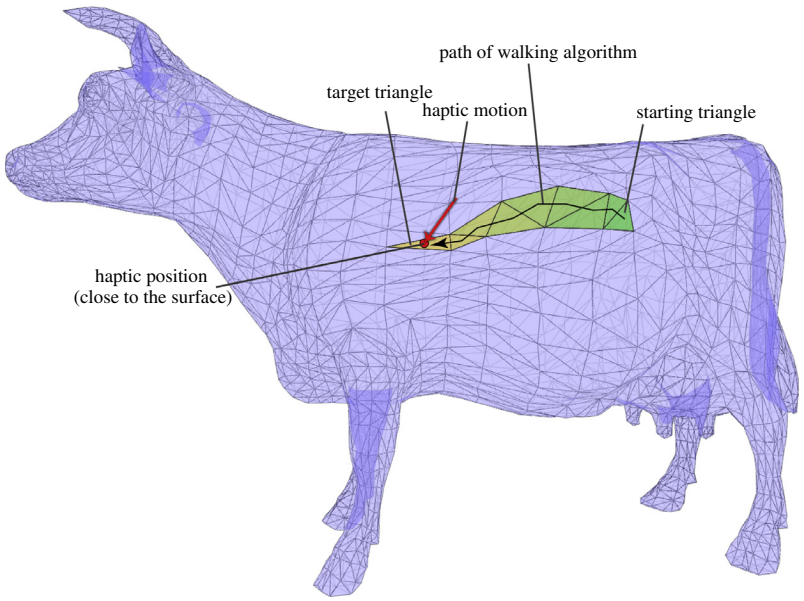

Fig. 2. Illustration of the collision detection on the surface of triangulated 3D model using the walking algorithm. The target triangle is intersected by a line, which is defined by the position of the haptic probe and the haptic motion vector. For a better depiction the model is partially transparent which makes overlapping parts of the mesh a little darker.

algorithms for collision detection in haptic visualization. 'Experimental results' presents our experiments performed on well-known triangulated surface models and 'Conclusion' concludes the paper.

State of the art

Collision detection and contact determination problems has been extensively studied in [9,11–17].

The simplest algorithms for collision detection are based on using bounding volumes and spatial, hierarchical decomposition techniques. For such a decomposition, e.g., k–d trees and octrees [11], cone trees, sphere trees [12,13], R-trees and their variants, trees based on S-bounds [15], and oriented bounding boxes [9] are used. Other spatial representations are based on BSP's [16] and its extensions to multi-space partitions [17].

As it is mentioned, e.g., in [2], it is very difficult to handle interactive rendering framerates for complex models due to a computation of collision detection. Sela et al. [2] propose a haptic surgical simulator where the finite element method (FEM) is used for off-line preprocessing of physical properties around the cut. Then the discontinuous free-form deformation (DFFM) is created. Note that only DFFM is used during a real-time cutting stage since a fast solution is needed. A surface model of a skin or flash is represented as a polygonal surface. To handle collision queries of a haptic cursor in a sufficient rate, authors use a preprocessed uniform voxel grid around the model.

Gregory et al. [10] present a hybrid collision detection framework for haptic interaction which uses a hierarchical representation of uniform grids and trees of oriented bounding boxes. Their framework utilizes the frame-to-frame coherence of a haptic probe. However, like other techniques using search data structures, modification for surface triangular meshes which are changing in time is complicated and continuous updating of data structures provides a significant decrease of the speed.

The approach proposed in this paper is based on the walking principle which is usually used for point location in planar triangular meshes or in tetrahedral meshes. The surveys of walking algorithms are presented in [18,19]. [20,21] used a walking algorithm in spherical re-meshing problem for point location on the surface of a triangulated sphere. Some algorithms (e.g. [22]) utilize the walking algorithms for point location on the surface triangular meshes in the local context (following the prior utilization of a search data structure, such as the octree). However, as far as we know, complex walking algorithms have not been applied to the problem of point location or collision detection on the surface triangular meshes.

There are several walking algorithms solving point location problem, and according to the style how they determine the way of the walk they can be divided into three groups: visibility, straight and orthogonal walks. Visibility walks [18,23–25] use local "visibility" tests to determine the way of their walk. These tests look for such an edge of a triangle 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. Straight walk algorithms [18,26,27] use not only the local comparisons to determine the way of the walk, but also use a line connecting one point of the starting triangle with the query point and traverse triangles crossed by this line. Orthogonal walks [18,21] first navigate along one coordinate axis and then along the other.

A clever selection of the starting triangle for walking may radically improve the speed of the algorithm, since it reduces the number of visited triangles during the walk. If any additional information about the data is known, it can be used in the selection, e.g., the target triangle from the last location query can be used as a starting triangle for the next query, if there is a coherency between queries. Without any knowledge of the data, the initial triangle can be chosen randomly. A better, yet still fast and simple alternative without any additional memory use was proposed in [28], where the initial triangle is selected as the nearest triangle from a set A of randomly chosen triangles from the scanned triangular mesh T. The total number of randomly selected triangles is significantly lower than the total number of triangles in T ($\|A\| \ll \|T\|$). For planar Delaunay triangulation of random points, an analysis of the ideal size of such a random subset has been proposed by [29], leading to the size of $O(\sqrt[3]{n})$.

Proposed method

A keystone for the walking-based approach is the orientation test of a point against a plane: let us have a plane given by three points **t**, **u**, **v** and a tested point **w**. Eq. (1) computes whether **w** lies above, on or below the given plane when seen from the side where **t**, **u**, **v** points are CCW oriented. In other words, the test decides whether the orientation of these points is positive, neutral or negative.

$$orientation3D(\mathbf{t},\mathbf{u},\mathbf{v},\mathbf{w}) = \begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix} \quad (1)$$

The proposed walking algorithm supposes that the model does not contain errors or holes, all the triangles have uniform orientation and have information about their neighbors. Note that CCW orientation of all the model triangles from the outside of the model is expected in the following text. Let us denote the i-th triangle visited by the proposed walking algorithm as τ_i.

Recall that the goal is to find and return a triangle ω from the surface triangular mesh T ($\omega \in T$), which is intersected by the line $\lambda = [\mathbf{q},\mathbf{m}]$ defined by the haptic probe position **q** and the haptic motion vector **m**. Generally, the line may intersect the model T in more intersection points, therefore, the intersection point should be in the distance from **q** lower than or equal to a defined maximal allowable distance $dist_{max}$. Moreover, the haptic motion vector **m** should direct towards the front face of ω. If such a triangle does not exist, *null* should be returned.

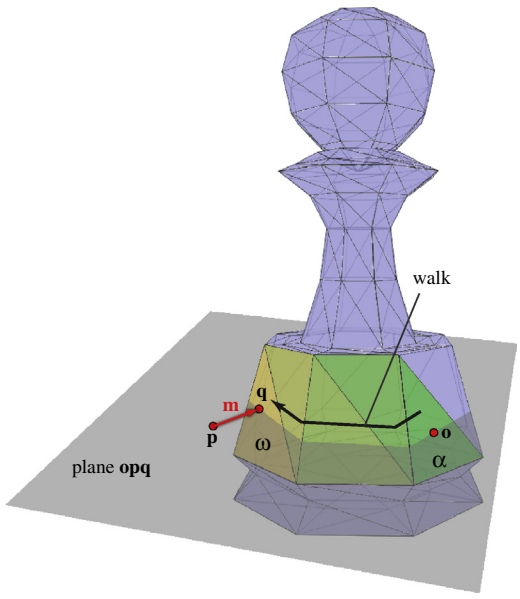

Fig. 3. Example of the walk on the surface model of the pawn.

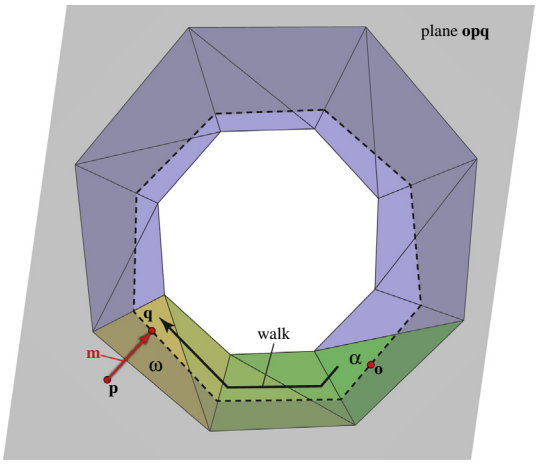

Fig. 4. Situation from Fig. 3 projected into the plane **opq** (only triangles intersected by the plane are shown). Dashed line is intersection of T and the plane **opq**.

Let us define *properly oriented triangle* as a triangle that **m** direct towards its front face. In the other words, if a triangle defined by vertices $\mathbf{t_0t_1t_2}$ is a properly oriented triangle, the orientation test $orientation3D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_0} - \mathbf{m})$ provides a positive value (an auxiliary point $\mathbf{t_0} - \mathbf{m}$ lies above that triangle). Thus, the target triangle ω is always a properly oriented triangle.

Let us present now the main idea of the proposed location algorithm. It is based on the straight walk principle, however, it uses a plane to guide the search instead a line. First, a proper starting triangle $\alpha, \alpha \in T$ is chosen to start the search; a proper choice will be described later. Let us have a plane **opq** given by the points $\mathbf{q}, \mathbf{p} = \mathbf{q} - \mathbf{m}$ and the center of gravity **o** of the triangle α. The plane **opq** cuts the model. The algorithm then walks from the starting triangle through the neighboring triangles cut by **opq** and attempts to find the target triangle ω. Fig. 3 shows an example of the walk, Fig. 4 shows the same situation in the projection to **opq** (only triangles intersected by the plane **opq** are projected into the plane).

On one hand, the auxiliary plane **opq** guides the walk so that the walking path is given; on the other hand, in the given path starting in the triangle α, the triangle ω may not exist. This behavior complicates the location process since the walk following this plane comes back to the starting triangle α without finding ω. See an example of an unsuccessful walk in Fig. 5, where the path from the triangle α to the triangle ω does not exist and the walk comes back to α (it may also happen when α lies in another component than ω – not the case in Fig. 5). Then the walk is restarted with a new choice of the starting triangle and with another cutting plane **opq**. The more complicated shape of the model is, the more such iterations may be needed, however, as will be shown in 'Experimental results', the number of required iterations can be greatly reduced by a clever choice of the first triangle.

Walking iterations may be unsuccessful due to non-existence of the target triangle ω. If there is a possibility that ω does not exist (as it is in the collision detection), an upper number of the allowed iterations should be set. Then the walk is repeated until the target triangle ω is found or the upper number of allowed iterations is reached. As will be shown in 'Experimental results', even a low upper number of allowed iterations ensures a negligible number of false negative results (results where ω exists but is not found). Moreover, since a lot of location queries is performed in a very short time, the distance between two positions **q** in the consecutive location queries is very low, usually several times lower than $dist_{max}$. Therefore, the algorithm can afford to return several consecutive false negative results without consequences before the haptic probe enters the model.

As we mentioned, the number of performed iterations can be significantly lowered by a clever choice of the first triangle α. Generally, the closer the α to ω, the lower the average number of performed iterations is. When the haptic probe is moving, the search problem is solved repeatedly, with two following **q** positions being mutually close since the probe movement is continuous. Therefore, it is useful to employ coherence in the choice of the starting triangle – the triangle ω from the last location query is used as the starting triangle α for the first iteration of the next location query.

Now let us explain the selection of the starting triangle α where coherence cannot be employed or if it has not been successful in the first iteration, see Algorithm 1 for a detailed description. The goal of this part is to find a triangle which is 'close enough' to **q** and properly oriented. The check of the proper orientation helps to minimize the probability of a choice of a triangle which is close to the target triangle ω in the Euclidean distance but far in the topological distance (measured on the model surface). The algorithm utilizes a random sampling [28] – a subset of the triangles from T is checked on the Euclidean distance from the point **q** and the properly oriented triangle nearest to **q** is chosen as α. The quadratic distance from the first of triangle vertices is measured for simplicity and efficiency. The size of the random sample is given by a user parameter k. A bigger k causes more computation but a

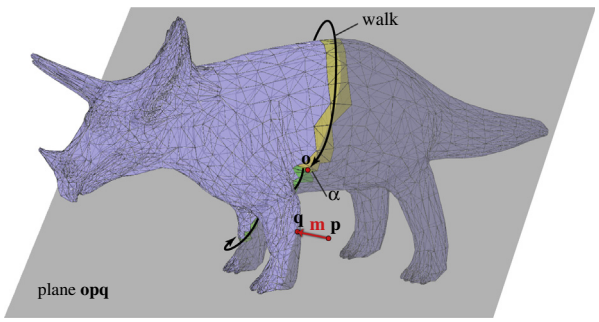

Fig. 5. Example of an unsuccessful iteration of the walk on the surface model of triceratops.

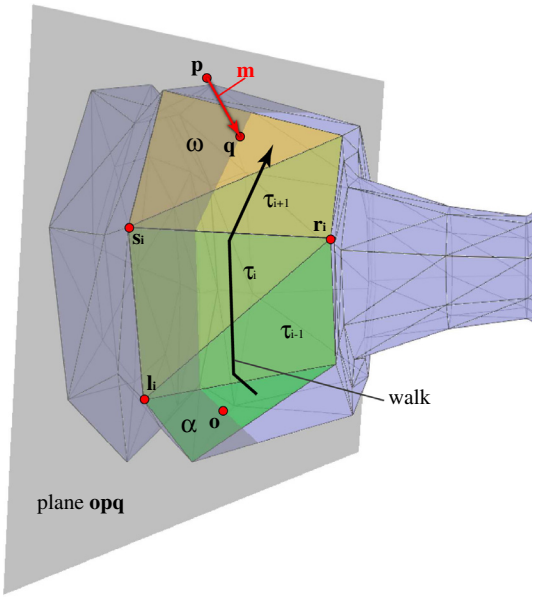

Fig. 6. Example of the step of the straight walk algorithm on the surface model of the pawn; τ_{i-1} is the previous triangle, τ_i is the current, and τ_{i+1} is the next triangle on the walk.

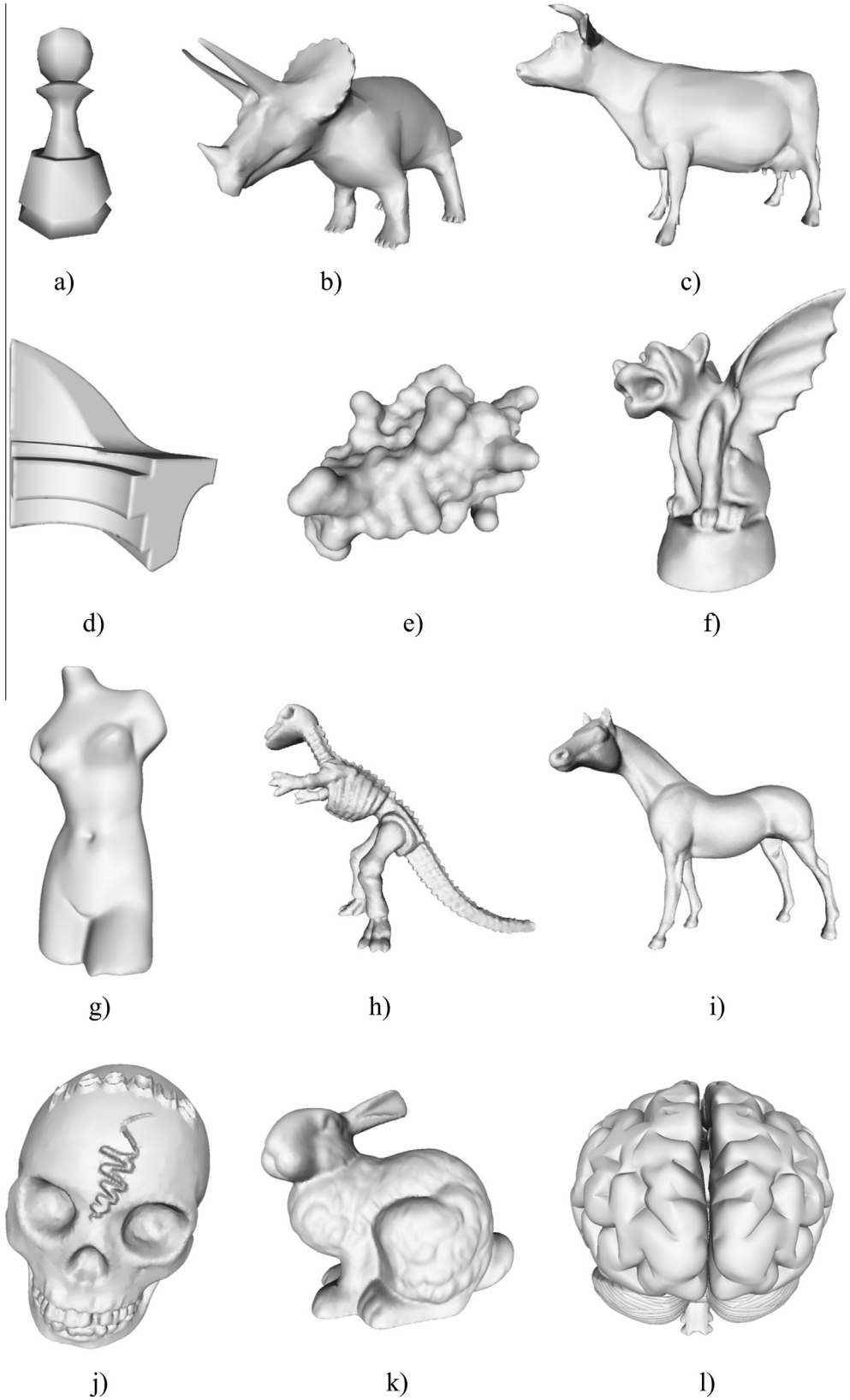

Fig. 7. Example of datasets visualizations, (a) pawn, (b) triceratops, (c) cow, (d) fandisk, (e) blob, (f) gargoyle, (g) venusbody, (h) dinosaur, (i) horse, (j) Headus skull, (k) Stanford bunny and (l) brain.

better choice of the starting triangle and vice versa. If no properly oriented triangle is found in k steps, the search continues until a properly oriented triangle is found. A proper value of k is highly dependent on the specific model, however, $k = 2 \cdot \sqrt[3]{n}$ (where n is the number of vertices in T) worked as a good compromise for all the tested models in our experiments.

Algorithm 1. First triangle selection

Input:
the triangular mesh T, the haptic probe information (the query point $\mathbf{q}$, the motion vector $\mathbf{m}$),
the number of choices of the first triangle k

Output:
the chosen starting triangle $\alpha \in T$

```
/* initialization                                                    */
double dist_min = maximal double value;
triangle α = null;
int i = 0;
/* looking for a proper nearby triangle                              */
repeat
    i = i + 1;
    triangle τ = random triangle from T;
    /* triangle τ = t0t1t2 (τ is defined by vertices t0, t1, t2)     */
    double dist = ||t0 − q||;
    /* check if τ is closer to q than the current α                  */
    if dist < dist_min then
        point t = t0 − m;
        /* check if τ is properly oriented (m direct towards the front face of the τ) */
        if orientation3D(t0, t1, t2, t) > 0 then
            α = τ;
            dist_min = dist;
        end
    end
until i ≥ k and α is not null;
return α;
```

Now the whole of the location algorithm will be described. First, a proper starting triangle α is chosen. Then its center of gravity $\mathbf{o}$ and $\mathbf{p} = \mathbf{q} - \mathbf{m}$ are computed and the cutting plane $\mathbf{opq}$ is defined. Note that the edge intersected by the plane is such an edge which is crossed by the plane between both vertices of the edge (vertices included). Thus, if the plane passes through the triangle vertex, both edges of the triangle defined by this vertex are considered as the edges intersected by the plane. Although it is not commented in the following text, if there are two possibilities how to choose the intersected output edge, an arbitrary one of them can be chosen for the proper continuation of the walk.

There are two tasks on each visited triangle τ_i in the walk: to find the output edge of τ_i intersected by the plane $\mathbf{opq}$ which will be used to go to the next triangle and to decide whether the current triangle τ_i is intersected by $\vec{\mathbf{pq}}$ in the distance from $\mathbf{q}$ lower than or equal to $dist_{max}$ or whether the walk will continue.

For the first triangle α, there are usually two edges intersected by the plane $\mathbf{opq}$, but due to the proper orientation of α, we can choose the output edge using orientation tests so that the path over this edge will be probably shorter (points $\mathbf{opq}$ are viewed in the CW order from the first vertex of the output edge and in the CCW order from the second vertex of the output edge, where the order of the edge vertices is given by the orientation of the triangle). For each successive triangle $\tau_i = (\mathbf{l_i}, \mathbf{r_i}, \mathbf{s_i})$, the vertices $\mathbf{l_i}, \mathbf{r_i}$ are determined by the input edge $\epsilon_{l_i r_i}$ (the edge used to enter the triangle τ_i) and the vertex $\mathbf{s_i}$ is opposite to $\epsilon_{l_i r_i}$. The algorithm determines the output edge by comparing the vertex $\mathbf{s_i}$ to the plane $\mathbf{opq}$ using the 3D orientation test. See Fig. 6 for an example of the walk. Note that $\mathbf{l_i}$ is always above the plane $\mathbf{opq}$ (points $\mathbf{o}, \mathbf{p}, \mathbf{q}$ are in the CCW order when seen from $\mathbf{l_i}$) and $\mathbf{r_i}$ is always bellow or on the plane. If $\mathbf{s_i}$ is above $\mathbf{opq}$ (on the left side of $\mathbf{opq}$ in Fig. 6), the output edge is $\epsilon_{r_i s_i}$, otherwise, the output edge is $\epsilon_{s_i l_i}$. Note that the back side of $\mathbf{opq}$ is seen in Fig. 6 (points $\mathbf{o}, \mathbf{p}, \mathbf{q}$ are in the CW order).

Before the algorithm continues through the output edge to the next triangle, it computes the orientation test for the point $\mathbf{q}$ with respect to the plane defined by the output edge and the point $\mathbf{p}$. If the point $\mathbf{q}$ lies on the same side as the third vertex of τ_i (vertex not defining output edge), a triangle intersected by $\vec{\mathbf{pq}}$ has been found. Then it is checked, if the triangle is properly oriented. If so, the intersection point of the triangle with $\vec{\mathbf{pq}}$ is computed. If that intersection point is in a distance from $\mathbf{q}$ lower than or equal to $dist_{max}$, the target triangle ω was found. In all other cases, the algorithm continues through the output edge to the next triangle. See pseudo-code of the algorithm in Algorithm 2.

Note that two orientation tests per each visited triangle τ_i are needed. One test to determine which edge is the output edge and one test to determine whether the current triangle τ_i is intersected by $\vec{\mathbf{pq}}$ (by control on which side of the plane defined by the output edge and by the point $\mathbf{p}$ the query point $\mathbf{q}$ lies). For each found intersected triangle, the controlled side of the plane is changed to the opposite of the previous one, since the algorithm is approaching the next intersected triangle from the opposite side. In order to determine on which side of the plane the query point $\mathbf{q}$ should be, an auxiliary variable is used (the variable *mark* in Algorithm 2). Additionally, one orientation test and sometimes also one intersection point computation are performed, if this triangle is intersected by $\vec{\mathbf{pq}}$.

Algorithm 2. Walking algorithm for surface location

Input:
the triangular mesh T, the information about haptic probe (query point $\mathbf{q}$, motion vector $\mathbf{m}$), the triangle β which was found in the last location, the number of choices of the first triangle k, the maximal allowable distance $dist_{max}$, the maximal number of iterations $iter_{max}$

Output:
the triangle ω intersected by the line $\lambda = [\mathbf{q}, \mathbf{m}]$ in distance from $\mathbf{q}$ lower than or equal $dist_{max}$

```
/* initialization                                                              */
triangle α, τ;
point o, r, l, s;
point p = q − m;
boolean found = false;
int iter = 0;
if β is null then α = first_triangle selection (q, m, T, k); else α = β;
repeat
    iter = iter + 1;
    int mark = 1;
    τ = α;
    /* triangle τ = t0 t1 t2 (τ is defined by vertices t0, t1, t2)            */
    o = ( t0 + t1 + t2 )/ 3;
    /* now plane opq, which is cutting T, is defined by points o, p, q        */
    foreach edge ϵ ∈ τ do
        r = first vertex of ϵ;
        l = second vertex of ϵ;
        if orientation3D (o, p, q, r) ≤ 0 and orientation3D (o, p, q, l) ≥ 0 then break;
    end
    /* now r and l are on the opposite sides of the plane opq                 */
    /* from the r side points o, p, q are in the CW order and from the l side are in the CCW order */
    repeat
        if mark · orientation3D (r, l, p, q) ≤ 0 then
            /* now τ = t0 t1 t2 is intersected by pq→                          */
            mark = − mark;
            s = t0 − m;
            /* check if τ is properly oriented (m direct towards the front face of the τ) */
            if orientation3D (t0, t1, t2, s) > 0 then
                s = intersection point of pq→ with triangle τ;
                /* check if q is close enough to τ                            */
                if ‖q − s‖ ≤ dist_max then
                    found = true;
                    break;
                end
            end
        end
        τ = neighbor of τ over ϵ_rl;
        s = vertex of τ where s ∉ ϵ_rl;
        if orientation3D (o, p, q, s) < 0 then r = s; else l = s;
    until τ equals α;
    if not found then α = first_triangle_selection(q, m, T, k);
until found or iter ≥ iter_max;
if found then return τ; else return null;
```

The described algorithm may need more iterations to find the target triangle (it happens especially when ω does not exist, then the upper number of iterations is always performed). As these particular iterations are independent, location process can be parallelized, i.e., more threads may run separated walks with different starting triangles at the same time. The thread which first finds the target triangle stops the whole location.

Experimental results

For the test purposes, we have implemented our algorithm in C++ with adaptive floating point arithmetic [30] to avoid numerical problems. The solutions have been tested on Intel Q6600 2.40 GHz in the single thread mode. The SSE2 random generator was used for randomization, since it is declared as up to five times faster then the standard C random generator [31].

Table 1
The test results measured with the first variant of input data (points generated on the surface, where the corresponding inverted surface normals are used as input vectors). The upper number of allowed iterations is not restricted.

Dataset			Visited Δ		Tests		Intersect.	Iterations		Time
Name	# of Δ	k	ϕ	*Max*	ϕ	*Max*	Dist. ϕ	ϕ	*Max*	t (μs)
Pawn	304	10	13.51	429	34.09	904	1.12	1.20	14	2.3
Triceratops	5660	28	29.77	8674	68.19	17463	1.03	1.20	77	7.0
Cow	5804	28	48.42	3671	107.28	7419	1.10	1.44	47	9.4
Fandisk	12946	36	19.77	1819	47.37	3647	1.00	1.00	7	7.4
Blob	16068	40	27.28	2395	62.46	4803	1.02	1.05	9	9.0
Gargoyle	20000	42	31.41	2625	71.14	5262	1.02	1.05	16	9.9
Venusbody	22720	44	18.98	1611	45.18	3250	1.00	1.00	6	8.9
Dinosaur	28136	48	42.79	11146	94.98	22576	1.03	1.14	95	13.1
Horse	39698	54	37.88	5980	84.56	12027	1.01	1.06	16	13.6
Skull	40000	54	26.09	4231	59.58	8495	1.01	1.01	22	11.8
Bunny	71888	66	35.61	2894	79.29	5805	1.01	1.02	6	15.9
Brain	588032	132	435.91	136768	885.72	273939	1.34	1.36	125	91.2

Table 2
The comparison of the algorithm parameters on the second variant of input (points generated near the surface in the distance lower than or equal to the maximal allowable distance $dist_{max}$, where input vectors are randomly deviated from the corresponding inverted surface normals, i.e. the maximal deviation is lower than 90°). The upper number of allowed iterations is restricted to 10.

Dataset			Found	Visited Δ		Tests		Intersect. dist. ϕ	Iterations		Time
Name	# of Δ	k	# of Δ	ϕ	*Max*	ϕ	*Max*		ϕ	*Max*	t (μs)
Pawn	304	10	999782	16.91	517	41.49	1063	1.21	1.30	10	2.4
Triceratops	5660	28	999002	44.74	2476	99.44	4962	1.14	1.32	10	9.2
Cow	5804	28	996583	62.34	2771	136.39	5595	1.26	1.55	10	11.2
Fandisk	12946	36	999996	33.66	3777	75.51	7585	1.07	1.02	10	8.9
Blob	16068	40	999934	80.70	4022	171.51	8072	1.28	1.20	10	15.0
Gargoyle	20000	42	999941	69.45	4557	148.64	9145	1.21	1.15	10	14.1
Venusbody	22720	44	999994	34.76	4797	77.75	9621	1.05	1.04	10	10.8
Dinosaur	28136	48	999601	80.82	6605	172.65	13239	1.24	1.27	10	17.6
Horse	39698	54	999930	62.24	5187	134.29	10412	1.08	1.13	10	16.5
Skull	40000	54	999735	83.57	6093	176.87	12219	1.21	1.13	10	18.9
Bunny	71888	66	999993	73.48	6084	156.43	12200	1.11	1.08	10	20.2
Brain	588032	132	992373	766.14	24591	1549.98	49235	1.74	1.63	10	139.8

The tests are presented on the following well-known 3D models: pawn (304 triangles), triceratops (5660 triangles), cow (5804 triangles), fandisk (12,946 triangles), blob (16,068 triangles), gargoyle (20,000 triangles), venusbody (22,720 triangles), dinosaur (28,136 triangles), horse (39,698 triangles), Headus skull (40,000 triangles), Stanford bunny (71,888 triangles), and brain (588,032 triangles) – see models visualization example in Fig. 7.

For each dataset, we have tested three different variants of input data with the following positions to the surface model:

1. points generated on a surface – motion vectors of these points correspond to the opposite normal vectors of triangles of the model,
2. points generated near a surface – oriented lines λ defined by these points and their motion vectors intersect triangles on the surface of 3D model in distance lower than or equal to the defined maximal allowable distance $dist_{max}$, input motion vectors are randomly deviated from the opposite surface normals which correspond to the intersected triangles of the model; the maximal deviation is lower than 90°,
3. points generated further from a surface – oriented lines λ intersect triangles on the surface of 3D model in a little greater distance than $dist_{max}$, input motion vectors are generated in the same way as in the second variant.

For each tested variant and each dataset, we performed 10^6 location queries on a randomly generated input corresponding to the above variants (input data coherence has not been utilized in tests).

As it is described in 'Proposed method', the first triangle is selected as the closest properly oriented triangle to **q** from a set of k randomly chosen triangles from T. The determination of proper k depends on the specific dataset, however, we set $k = 2 \cdot \sqrt[3]{n}$ as a good compromise through the datasets, where n is the number of vertices in the mesh.

For each dataset, the following parameters were measured:

- k is the size of the set of randomly chosen triangles from T which was used to choose the starting triangle,
- *Visited* Δ is the number of visited triangles, where ϕ shows the average and *max* shows the maximum number of visited triangles,
- *Tests* is the number of performed orientation tests, where ϕ shows the average and *max* shows the maximum number of orientation tests,
- *Intersect. dist.* ϕ shows the average number of computations for the intersection point of λ with a surface triangle and of the corresponding distance computation,
- *Iterations* is the number of performed iterations, where ϕ shows the average and *max* shows the maximum number of iterations,
- t (μs) shows the average location time.

All the quantities reflect values for one location query while *max* values show specific location queries where the appropriate value was maximal.

Table 1 corresponds to the point location on the surface. Such a situation is useful especially for applications where the target triangle ω always exists and we need to find it. Thus the algorithm

Table 3
The comparison of the algorithm parameters on the set of input points generated farther from the surface in the distance larger than the defined maximal allowable distance $dist_{max}$. The number of iterations was restricted to 10.

Dataset			Visited Δ		Tests		Intersect. dist. ϕ	Iterations ϕ/*Max*	Time t (μs)
Name	# of Δ	k	ϕ	*Max*	ϕ	*Max*			
Pawn	304	10	320.42	898	701.30	1861	10.05	10	27.5
Triceratops	5660	28	1132.21	2906	2341.48	5894	9.96	10	130.4
Cow	5804	28	1102.53	2943	2282.64	5932	9.58	10	128.1
Fandisk	12946	36	2531.65	5296	5145.98	10624	11.85	10	277.0
Blob	16068	40	2703.99	5015	5495.06	10088	12.74	10	305.3
Gargoyle	20000	42	2635.90	6650	5358.98	13384	12.10	10	309.3
Venusbody	22720	44	2802.73	5960	5684.90	11974	10.57	10	331.1
Dinosaur	28136	48	2133.49	9184	4356.47	18426	11.34	10	281.7
Horse	39698	54	2966.65	9812	6021.38	19695	10.56	10	382.5
Skull	40000	54	3875.68	7075	7838.21	14282	12.02	10	472.6
Bunny	71888	66	4905.23	10372	9899.73	20770	11.43	10	626.8
Brain	588032	132	11017.44	28211	22147.02	56472	13.42	10	1645.3

is searching until the triangle is found. The upper number of allowed iterations is irrelevant in this case and it is not restricted. The search behaves well since the opposite surface normals were used as input motion vectors which led to a suitable choice of the starting triangle.

On the other hand, if there is a possibility that the searched triangle ω does not exist (as it is in the collision detection), the upper number of allowed iterations should be defined. For the second and the third variant of input data we limit the upper number of allowed iterations to 10 since it is a good compromise between performance and accuracy. See Table 2 for the tests of the second variant of input data (points generated near the surface). Since the number of iterations is restricted, triangle ω may not be always found, despite the fact that it exists. Therefore, the number of located triangles ω is also shown in Table 2 as *Found # of* Δ and shows the efficiency of the collision detection.

Although the upper number of allowed iterations is restricted to 10, the average results are worse than in Table 1. It is caused by the input motion vectors which are diverted from the opposite surface normals and it results in the worse choice of the first triangle. Let us denote that the haptic visualization is highly dependent on the specific behavior of the user, thus it is very difficult to measure it. Therefore, input data are generated randomly to provide more objective results. However, in haptic visualization, during the motion on the surface of the object, coherency can be used for the choice of the first triangle, as described in 'Proposed method'. Moreover, the real motion vectors of the haptic probe are usually less diverted from the opposite surface normals. Both factors cause significantly better results in practical haptic visualization than is shown in Table 2.

As is shown in the table, efficiency is rather high in spite of a low number of iterations such as 10 (always more than 99% of ω triangles were found). Since the distance traveled by the haptic probe between two consecutive location queries is usually several times lower than $dist_{max}$, we have several attempts to detect the collision before the haptic probe enters the model. The probability that the haptic probe enters the model is therefore very low and we did not register it in practice.

The worst test results are on the *brain* model. It is caused mainly by its size (588 k faces), complex shape (a lot of folds) and by the fact that the model consists of more components (the path sometimes does not exist). Although the coherence was not used in the tests and thus location queries are several times slower than in practice, algorithm is still fast enough for haptic visualization, even for big shaped-complicated models as is the brain model. Average time per one location query for the brain model is less than 0.14 ms which corresponds to the average number of visited triangles (about 766) for this model and which is better than the requirements. As we can see, the longest walk visits almost 25 k triangles (location time less than 3 ms), but such cases are very rare and in practice where the coherence is often used in the choice of α, we have not registered them. Moreover, an occasional execution of a little slower location queries may not be a noticeable problem in haptic visualization.

The tests of the third type of input data (points generated further from the surface in a little larger distance than $dist_{max}$) are in Table 3. The location is slower in comparison with other scenarios, but we must realize that there is no contact of the haptic probe with the surface in such a case. So there is no need of feedback computation. Moreover, since the distance is only a little greater than $dist_{max}$ from the surface, the collision detection is justified. However, in practice, we can skip collision detection in the cases where it is obviously useless, e.g., using several simple bounding objects.

Experiments presented above show that the proposed algorithm behaves well for haptic-related tasks as well as for more general problems. Moreover, it can handle mesh models consisting of several components (continuous path between components does not exist). It is provided by a repetitive random choice of the starting triangle.

The proposed location algorithm is very fast for cases where the haptic cursor is close to the surface. On the other hand, for cases where there is no collision, it is several times slower. But we must realize that in such a case there is no force acting against the user, thus the location query can be slower. Once the haptic probe is in the proximity of the surface, detection becomes very fast and an appropriate force feedback could be computed. This perfectly meets usual requirements for the haptic visualization.

Note that our algorithm can be scaled to comply with various criteria – at the first extreme case the location is very fast at a cost of increasing number of location errors. The second extreme case is perfect correctness of the location, but at a cost of computing time.

In our experiments we set the location parameters to meet haptics requirements (location time less than 1 ms and negligible wrong location rate where the target triangle exists and location time less than 10 ms where the target triangle does not exist).

We have also performed a haptic visualization test. In cases, where the target triangle does not exist, results were similar to results in Table 3. In cases, where the target triangle exists, the frame-to-frame coherence was utilized for the choice of the first triangle in the most location queries. Therefore, the performance indicators (No. of visited triangles, No. of performed tests, and No. of performed iterations) were significantly better than in Table 2. Average time for one location query was not measured,

since it may be inaccurate for individual queries. However, according to other performance indicators, corresponding improvement of average time can be also expected. As it was mentioned above, all the practical haptic visualization tests may differ significantly, since they depend on the used model and mainly on the behavior of the user. Thus the specific measured values are not presented in the text.

Conclusion

We have presented the walking algorithm for a triangle location or collision detection problem on the surface (possibly changing) of triangulated 3D models. Although we focus mainly on its application in haptic visualization, the algorithm is also useful for general use. As it is shown in experiments, even for the biggest tested model which is complex-shaped and contains 588 k triangles, the average location time is deeply under the required 1 ms which is far better than other published methods. The main advantages of the presented algorithm are its capability of handling time-changing models or models consisting of several components, easiness of implementation and negligible memory requirements. The algorithm is also suitable for parallelization.

Acknowledgment

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic, Project Kontakt No. LH11006, by University spec. research - 311, and by UWB grant SGS-2013-029 – Advanced Computing and Information Systems.

References

[1] Zhou J, Shen X, Georganas ND. Haptic tele-surgery simulation. In: Proceedings. The 3rd IEEE international workshop on haptic, audio and visual environments and their applications, 2004. HAVE 2004; 2004. p. 99–104.
[2] Sela G, Subag J, Lindblad A, Albocher D, Schein S, Elber G. Real-time haptic incision simulation using FEM-based discontinuous free-form deformation. Comput-Aid Des 2007;39(8):685–93.
[3] Liu A, Tendick F, Cleary K, Kaufmann C. A survey of surgical simulation: applications, technology, and education. Presence: Teleop Virt Environ 2003;12(6):599–614.
[4] Pocheville A, Kheddar A, Yokoi K. I-touch: a generic multimodal framework for industry virtual prototyping. In: TExCRA'04. First IEEE technical exhibition based conference on robotics and automation, 2004; 2004. p. 65–6.
[5] Colwell C, Petrie H, Kornbrot D, Hardwick A, Furner S. Haptic virtual reality for blind computer users. In: Proceedings of the third international ACM conference on assistive technologies, Assets'98. New York (NY, USA): ACM; 1998. p. 92–9.
[6] Zilles CB, Salisbury JK. A constraint-based god-object method for haptic display. Proceedings. 1995 IEEE/RSJ international conference on intelligent robots and systems 95. 'Human robot interaction and cooperative robots', vol. 3. IEEE; 1995. p. 146–51.
[7] Ruspini DC, Kolarov K, Khatib O. The haptic display of complex graphical environments. In: Proceedings of the 24th annual conference on computer graphics and interactive techniques, SIGGRAPH '97. New York (NY, USA): ACM Press/Addison-Wesley Publishing Co; 1997. p. 345–52.
[8] Colgate J, Brown J. Factors affecting the z-width of a haptic display. In: Proceedings. 1994 IEEE international conference on robotics and automation, 1994, vol. 4; 1994. p. 3205–10.
[9] Gottschalk S, Lin MC, Manocha D. Obbtree: a hierarchical structure for rapid interference detection. In: Proceedings of the 23rd annual conference on computer graphics and interactive techniques, SIGGRAPH '96. New York (NY, USA): ACM; 1996. p. 171–80.
[10] Gregory A, Lin MC, Gottschalk S, Taylor R. A framework for fast and accurate collision detection for haptic interaction. In: ACM SIGGRAPH 2005 courses, SIGGRAPH '05. New York (NY, USA): ACM; 2005. p. 38–45.
[11] Samet H. An Overview of Quadtrees, Octrees, and Related Hierarchical Data Structures. Theoretical Foundations of Computer Graphics and CAD, in: NATO ASI Series, vol. 40. Berlin, Heidelberg: Springer; 1988. p. 51–68.
[12] Hubbard P. Interactive collision detection. In: Proceedings. IEEE 1993 symposium on research frontiers in virtual reality, 1993; 1993. p. 24–31.
[13] Quinlan S. Efficient distance computation between non-convex objects. In: Proceedings of international conference on robotics and automation; 1994. p. 3324–9.
[14] Beckmann N, Kriegel H-P, Schneider R, Seeger B. The R*-tree: an efficient and robust access method for points and rectangles. In: International conference on management of data. ACM; 1990. p. 322–31.
[15] Cameron S. Approximation hierarchies and S-bounds. In: Proceedings of the first ACM symposium on solid modeling foundations and CAD/CAM applications, SMA '91. New York (NY, USA): ACM; 1991. p. 129–37.
[16] Naylor B, Amanatides J, Thibault W. Merging BSP trees yield polyhedral modeling results. In: Proc of ACM Siggraph; 1990. p. 115–124.
[17] Bouma WJ, GV, Jr. Collision detection and analysis in a physically based simulation. In: Eurographics workshop on animation and simulation; 1991. p. 191–203.
[18] Devillers O, Pion S, Teillaud M. Walking in a triangulation. In: Proceedings of the 17th annual symposium on computational geometry; 2001. p. 106–14.
[19] Soukal R, Málková M, Kolingerová I. Walking algorithms for point location in TIN models. Comput Geosci 2012;16:853–69.
[20] Wu Y, He Y, Tian H. A spherical point location algorithm based on barycentric coordinates. In: Proceedings of the 5th computational science and its applications; 2005. p. 1099–108.
[21] Soukal R, Kolingerová I. Star-shaped polyhedron point location with orthogonal walk algorithm. Proc Comput Sci 2010;1:219–28.
[22] Rypl D, Bittnar Z. Triangulation of 3D surfaces reconstructed by interpolating subdivision. Comput Struct 2004;82(23–26):2093–103.
[23] Kolingerová I, Žalik B. Reconstructing domain boundaries within a given set of points, using Delaunay triangulation. Comput Geosci 2006;32(9):1310–9.
[24] Sundareswara R, Schrater P. Extensible point location algorithm. In: International conference on geometric modeling and graphics; 2003. p. 84–9.
[25] Soukal R, Málková M, Kolingerová I. A new visibility walk algorithm for point location in planar triangulation. Advances in visual computing, of lecture notes in computer science, vol. 7432. Berlin Heidelberg: Springer; 2012. p. 736–45.
[26] Mehlhorn K, Näher S. Leda: a platform for combinatorial and geometric computing. Commun ACM 1995;38(1):96–102.
[27] Soukal R, Kolingerová I. Straight walk algorithm modification for point location in a triangulation. In: EuroCG'09: Proceedings of the 25th European workshop on computational geometry, Brussels, Belgium; 2009. p. 219–22.
[28] Mücke EP, Saias I, Zhu B. Fast randomized point location without preprocessing in two and three-dimensional delaunay triangulations. In: Proceedings of the 12th annual symposium on computational geometry, vol. 26; 1996. p. 274–83.
[29] Devroye L, Mucke EP, Zhu B. A note on point location in delaunay triangulations of random points; 1998.
[30] Shewchuk Jonathan R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Dis Comput Geom 1997;18(3):305–63.
[31] Owens K, Parikh R. Fast random number generator on the intel pentium 4 processor. Intel Softw Network 2009.