University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Walking Location Algorithms

State of the Art and Concept of PhD. Thesis

Roman Soukal

Technical Report No. DCSE/TR-2010-03
April, 2010

Distribution: public/confidental

Technical Report No. DCSE/TR-2010-03
April 2010

Walking Location Algorithms

Roman Soukal

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

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Copyright ©2010 University of West Bohemia in Pilsen, Czech Republic

Acknowledgments

I would like to thank my colleagues for their numerous practical advices. Great thank belongs to my supervisor Prof. Ivana Kolingerová for her time, patience and valuable comments.

Contents

1 Introduction **4**

1.1 Basic Notation 5

1.2 Problem Definition 6

1.2.1 Point Location in 2D Triangulations 6

1.2.2 Point Location in 3D Triangulations 6

1.2.3 Point Location in Surface Triangulations 6

1.3 Framework 7

1.3.1 Planar Walking Algorithms 7

1.3.2 Star-shaped polyhedron point location 8

2 Planar Point Location **11**

2.1 Visibility Walk Algorithms 14

2.1.1 Lawson's Oriented Walk 14

2.1.2 Remembering Stochastic Walk 15

2.1.3 Fast Remembering Stochastic Walk 17

2.1.4 Barycentric Walk 18

2.2 Straight Walk Algorithm 20

2.3 Orthogonal Walk Algorithm 22

2.4 Choice of the Starting Triangle 25

2.5 Our Modifications of Planar Walking Algorithms 26

2.5.1 Distance Fast RSW 26

2.5.2 Flag Fast RSW 27

2.5.3 Progressive Visibility Walk 27

2.5.4 Normal-line Straight Walk 29

2.5.5 Improved Orthogonal Walk 31

2.6 Experimental Results 34

3 Point Location on Star-shaped Polyhedron Surface **36**
3.1 Wu's Barycentric Walk 37
3.2 Our 3D Walking Algorithms 40
3.2.1 Remembering Walk 40
3.2.2 Plücker Line Coordinates Visibility Walk 40
3.2.3 Barycentric Walk 40
3.3 Point Location in Spherical Coordinate System 42
3.3.1 Orthogonal Walk Algorithm in Spherical Coordinates . . 44
3.4 Preprocessing and using of hierarchical structures 46
3.5 Experimental Results 48

4 Future Work **51**
4.1 Planar Point Location 51
4.2 3D Point Location 52
4.3 Point Location on Surface Triangulation 53

Bibliography **57**

Activities **58**

Chapter 1

Introduction

Finding which element in a mesh contains a query point is a very frequent task in computational geometry. In general, elements may be arbitrary (also non convex) polygons or polyhedra. In this text we focus on point locations in the following two types of meshes: 2D triangulations and surface triangulations of 3D models of objects.

The algorithms solving point location problems can be divided into two groups: algorithms using additional data structures and algorithms without using additional data structures. The latter group includes mainly so called *walking algorithms*. The former algorithms concentrate on achieving the lowest complexity possible, in this case $O(\log n)$ per point query which is achieved by using sophisticated data structures (n is a number of vertices in the mesh). Despite their low complexity, these algorithms have some disadvantages which will be later outlined.

The name of walking algorithms has arisen from the way of point location. The walking strategy makes use of connectivity of the mesh to go through elements between the starting element and the element which contains query point. The walking algorithms do not need any additional data structures, they use only connectivity in the mesh, thus often they are more favored possibility than the optimal time complexity solutions. Hence we focus on walking algorithms in the text of this thesis.

This work is organized as follows. Chapter 1 includes introduction, basic notations, problem definitions and details about used framework. Chapter 2 presents the planar point location, especially the walking algorithms. Our modifications of the planar walking algorithms are presented in Section 2.5. Chapter 3 shows problematic of the point location on the surface of a star-shaped polyhedron and walking algorithms for it. Future work is presented in Chapter 4.

1.1 Basic Notation

In this subsection, we explain the basic notation. This notation is used in the text of this thesis if it is not specified otherwise in the respective subsection.

Vectors including points and vertices are denoted by bold Roman characters (e.g. $\mathbf{a}, \mathbf{b}$). Scalar values are denoted by Roman characters in italic (e.g. k, l), especially components of vectors are denoted by Roman characters in italic with lower indices (e.g. $\mathbf{a} = (a_x, a_y)$).

We use lower case Greek letters for denotation of elements as triangles or tetrahedra (e.g. β, γ). Letters ϵ, ξ are used specially for denotation of edges of triangle or faces of tetrahedron. Having in mind a particular edge or face, we use lower indices of the vertices which belong to this edge or face (e.g. ϵ_{ab}, ξ_{abc}). In addition, the letter λ is used specially for denotation of a line. Having in mind a particular line, we use lower indices (e.g., λ_n is a line orthogonal to λ). For a line segment between two points (e.g., points $\mathbf{a}, \mathbf{b}$ where $\mathbf{a} \neq \mathbf{b}$), we can use $\overrightarrow{\mathbf{ab}}$. The letter ρ is usually used for denotation of a plane.

Upper case Roman characters are used to denotation of sets and arrays of vectors or elements (e.g. $A = \{\mathbf{a}, \mathbf{b}\}$, $T = \{\alpha, \tau, \omega\}$). Bold upper case Roman characters are used to denotation of matrices (e.g. $\mathbf{A}$).

We usually use letter T for triangle mesh or tetrahedral mesh in which we want to locate query point. We usually use letter $\mathbf{q}$ for such a point. We assume that a part of input of each walking algorithm is also the element (triangle or tetrahedron) in which the walk of the algorithm start. When we talk about this starting element, we use the Greek letter α ($\alpha \in T$) for its denotation. We denote ω ($\omega \in T$) the triangle or tetrahedron which really contains $\mathbf{q}$. The Greek letter τ is usually used for each element on the walk of any walking algorithm.

1.2 Problem Definition

1.2.1 Point Location in 2D Triangulations

For a query point $\mathbf{q}$ and a given planar 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 $\mathbf{q}$. In the following text we use term *planar point location* for such a problem.

1.2.2 Point Location in 3D Triangulations

For a query point $\mathbf{q}$ and a given tetrahedral mesh T of n vertices, the point location problem usually means how to find a tetrahedron ω from T which contains $\mathbf{q}$. In the following text we use the term *spatial point location* for such a problem. This problem is described in [36] and is not included in this text.

1.2.3 Point Location in Surface Triangulations

For a query point $\mathbf{q}$ and a given surface triangulation T of an object model of n vertices, the definition of a surface point location problem, in general, is not so straightforward. Hence, in this text, we concentrate on the point location on a star-shaped polyhedron surface which is used especially as point location on a spherical surface. For a star-shaped polyhedron ζ, its surface triangulation T of n vertices and a center point $\mathbf{c}$, the spatial point location problem of a query point $\mathbf{q}$ usually means how to find a triangle ω from T which is intersected by the ray $\overrightarrow{\mathbf{cq}}$. Note that the center point $\mathbf{c}$ of ζ with the surface triangulation T is such a point inside ζ that each vertex of T is directly visible from $\mathbf{c}$. In the following text, we assume $\mathbf{c}$ is the part of input.

1.3 Framework

Very important part of the walking algorithms for triangle and tetrahedral meshes is a behavior of the algorithms in case $\mathbf{q}$ lays outside of the mesh, current triangle τ is on the border of this mesh and the walk wants to go outside of the mesh. In our algorithms, we return $\omega = null$ for such a situation. Tests to *null* are not mentioned in all presented algorithms and respected modification is very simple. In some particular applications, the output is demanded as last valid triangle on the walk. The corresponding modification is very simple too. In all algorithms, we assume that whole the triangulation is saved in the memory of a computer.

Another important part of all walking algorithms is the behavior of the algorithms if $\mathbf{q}$ lays on an edge ϵ. We extend the solution to the set Q that for each element ω from Q, ϵ is an edge of ω. The situation, where $\mathbf{q} = \mathbf{t}$, $\mathbf{t}$ is a vertex of T, is analogous.

1.3.1 Planar Walking Algorithms

In this subsection, we present framework used in planar walking algorithms.

2D orientation test [8] (see Equation 1.1, where $\mathbf{t}, \mathbf{u}$ are edge vertices or line points and $\mathbf{v}$ is the examined point) is very important for most of the walking algorithms because it returns position of one point against an edge (or a line) given by two vertices (by two points) (see Section 2.1.1). The side of $\overrightarrow{\mathbf{tu}}$ on which the point $\mathbf{v}$ lies is given by the sign of the determinant.

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

Sometimes it may be more useful to compute the implicit line equation (Equation 1.2) of a line λ from its two points (Equations 1.3, 1.4, where $\mathbf{g}, \mathbf{h} \in \lambda, \mathbf{g} \neq \mathbf{h}$) [34]. Then the position of an examined point $\mathbf{v}$ against this line is computed by the substitution of a, b, c coefficients of λ and coordinates of determining point $\mathbf{v}$ into Equation 1.5. We often use normal-line λ_n which is orthogonal to λ and goes through $\mathbf{h}, \mathbf{h} \in \lambda$ (see Equations 1.6, 1.7). For λ_n, we use the same test to determine the position of the point $\mathbf{v}$ as for λ (we substitute λ_n and $\mathbf{v}$ into Equation 1.5).

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

$$(a, b, c) = (g_x, g_y, 1) \times (h_x, h_y, 1) \tag{1.3}$$

$$a = g_y - h_y, b = h_x - g_x, c = g_x \cdot h_y - g_y \cdot h_x \tag{1.4}$$

$$position(\lambda, \mathbf{v}) = \operatorname{sgn}(a \cdot v_x + b \cdot v_y + c) \tag{1.5}$$

$$\lambda_n : a_n \cdot x + b_n \cdot y + c_n = 0 \tag{1.6}$$

$$a_n = b, b_n = a, c_n = a \cdot h_y - b \cdot h_x \tag{1.7}$$

Some planar walking algorithms (see Section 2.1.4) need barycentric coordinates of a triangle. Let us denote $\mathbf{b} = (b_0, b_1, b_2)$ as barycentric coordinates vector of a point $\mathbf{v}$ in the triangle $\sigma = \mathbf{t_0 t_1 t_2}$. Equations 1.8, 1.9, 1.10, 1.11, 1.12 show computing and relationships of components of the barycentric vector $\mathbf{b}$ [33, 43].

$$\mathbf{A}(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}) = \begin{bmatrix} t_{1x} - t_{0x} & t_{2x} - t_{0x} \\ t_{1y} - t_{0y} & t_{2y} - t_{0y} \end{bmatrix} \tag{1.8}$$

$$b_0(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = \frac{(t_{1y} - t_{2y}) \cdot (v_x - t_{2x}) - (t_{1x} - t_{2x}) \cdot (v_y - t_{2y})}{\det\left[\mathbf{A}(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})\right]} \tag{1.9}$$

$$b_1(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = \frac{(t_{2y} - t_{0y}) \cdot (v_x - t_{2x}) - (t_{2x} - t_{0x}) \cdot (v_y - t_{2y})}{\det\left[\mathbf{A}(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})\right]} \tag{1.10}$$

$$\sum_{i=0}^{2} b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = 1 \tag{1.11}$$

$$b_2(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = 1 - b_0(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) - b_1(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) \tag{1.12}$$

1.3.2 Star-shaped polyhedron point location

In this subsection, we present framework used for point location in tetrahedral meshes and also for point location on triangulation surface, for both of them by the walking algorithms.

The implicit line equation of a plane is used in some surface algorithms (see Section 3.1). The computation of this implicit line coefficients (Equation 1.13) of the plane ρ given by a triangle $\sigma = \mathbf{t_0 t_1 t_2}$ is described in Equation 1.14.

$$\rho : a \cdot x + b \cdot y + c \cdot z + d = 0 \tag{1.13}$$

$$(a, b, c, d) = (t_{0x}, t_{0y}, t_{0z}, 1) \times (t_{1x}, t_{1y}, t_{1z}, 1) \times (t_{2x}, t_{2y}, t_{2z}, 1) \tag{1.14}$$

3D orientation test (see Equation 1.15, where $\mathbf{t}, \mathbf{u}, \mathbf{v}$ are vertices of a face and $\mathbf{w}$ is examined point) is very important for most of the spatial walking algorithms because it returns the position of one point against a face given by three vertices (see Section 3.2.1). The side of $\sigma = \mathbf{tuv}$ on which the point $\mathbf{w}$ lies is given by the sign of the determinant.

$$orientation3D(\mathbf{t}, \mathbf{u}, \mathbf{v}, \mathbf{w}) = \begin{vmatrix} u_x - t_x & v_x - t_x & w_x - t_x \\ u_y - t_y & v_y - t_y & w_y - t_y \\ u_z - t_z & v_z - t_z & w_z - t_z \end{vmatrix} \tag{1.15}$$

Some spatial walking algorithms (see Section 3.2.3) need barycentric coordinates of a tetrahedron. Let us denote $\mathbf{b} = (b_0, b_1, b_2, b_3)$ as barycentric coordinates vector of a point $\mathbf{v}$ against tetrahedron $\sigma = \mathbf{t_0 t_1 t_2 t_3}$. For barycentric coordinates computation of tetrahedron we use 3D orientation test (see Equations 1.15, 1.16). Equation 1.17, 1.18 show relationships of components of barycentric vector $\mathbf{b}$ [33, 43].

$$b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{w}) = \frac{orientation3D(\mathbf{t}_{[(i+\mathbf{1}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{2}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{3}) \mod \mathbf{3}]}, \mathbf{w})}{orientation3D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3})} \tag{1.16}$$

$$\sum_{i=0}^{3} b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{w}) = 1 \tag{1.17}$$

$$b_3(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{w}) = 1 - \sum_{i=0}^{2} b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{t_3}, \mathbf{w}) \tag{1.18}$$

Lines in 3D can easily be given as coordinates of two distinct points (six numbers) or they can be given as coordinates of two distinct planes (eight numbers). The utilization of Plücker line coordinates is a compromise between these ways which can easily generate both. Furthermore, Plücker line coordinates are much more efficient for computations. For the points $\mathbf{g}, \mathbf{h}$ determining a line where $\mathbf{g} \neq \mathbf{h}$, Plücker line coordinates $\mathbf{u}, \mathbf{v}$ are given in Equations [31, 11, 13].

$$\mathbf{u} = \mathbf{g} - \mathbf{h} = (g_x - h_x, g_y - h_y, g_z - h_z) \tag{1.19}$$

$$\mathbf{v} = \mathbf{g} \times \mathbf{h} = (g_y \cdot h_z - h_y \cdot g_z, h_x \cdot g_z - g_x \cdot h_z, g_x \cdot h_y - h_x \cdot g_y) \tag{1.20}$$

The main value of Plücker line coordinates is in the 3D line orientation test which is use especially in ray tracing. Let us denote $\mathbf{u_1}, \mathbf{v_1}$ vectors of Plücker line coordinates of the first line and $\mathbf{u_2}, \mathbf{v_2}$ vectors of Plücker line coordinates of the second line. The orientation of given lines can be computed by the sum of two dot products of these vectors (see Equation 1.21). When $rayOrientation(\mathbf{u_1}, \mathbf{v_1}, \mathbf{u_2}, \mathbf{v_2}) = 0$ then lines have an intersection. If $rayOrientation(\mathbf{u_1}, \mathbf{v_1}, \mathbf{u_2}, \mathbf{v_2}) < 0$ then the orientation of the tested lines is clockwise. If $rayOrientation(\mathbf{u_1}, \mathbf{v_1}, \mathbf{u_2}, \mathbf{v_2}) > 0$ then the orientation of the tested lines is counterclockwise. Figure 1.1 shows the line pair signature meaning (where the orientation of two lines λ_1 and λ_2 is CW, CCW or lines intersect). If the orientation of a ray against each edge of a triangle is the same then the triangle is intersected by this ray (see Figure 1.2).

$$rayOrientation(\mathbf{u_1}, \mathbf{v_1}, \mathbf{u_2}, \mathbf{v_2}) = \mathbf{u_1} \cdot \mathbf{v_2} + \mathbf{u_2} \cdot \mathbf{v_1} \tag{1.21}$$

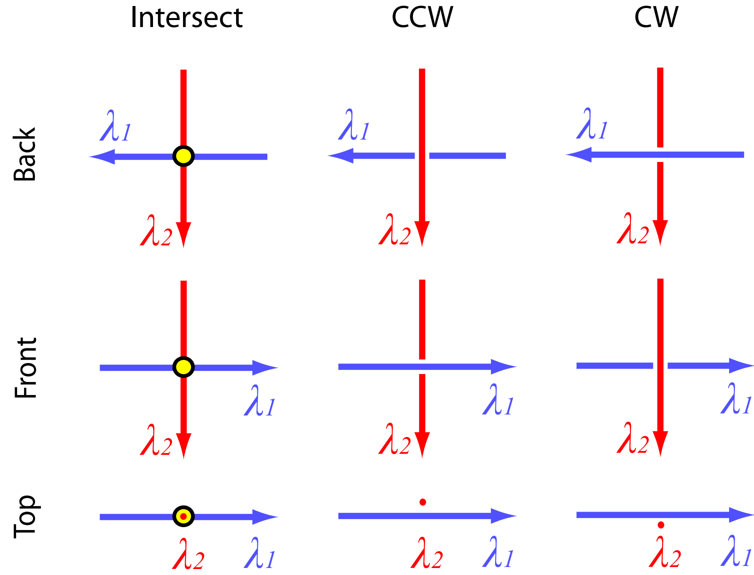

Figure 1.1: Line pair signature meaning

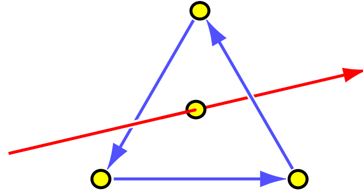

Figure 1.2: Triangle intersected by a ray

Especially in the star-shaped polyhedron point location, we use the computation of spherical coordinate values φ and θ (see Equations 1.22, 1.23, where $\mathbf{g}$ is an examined point and $\mathbf{c}$ is the center point of star-shaped polyhedron.

$$g_\varphi(\mathbf{g}, \mathbf{c}) = \mathrm{arctg}_2(g_y - c_y, g_x - c_x) \tag{1.22}$$

$$g_\theta(\mathbf{g}, \mathbf{c}) = \arccos\left(\frac{g_z - c_z}{\sqrt{(g_x - c_x)^2 + (g_y - c_y)^2 + (g_z - c_z)^2}}\right) \tag{1.23}$$

Chapter 2

Planar Point Location

The algorithms solving planar point location problem (see Figure 2.1) can be divided into two groups: algorithms using sophisticated data structures and algorithms without using additional data structures. The latter group includes mainly so called *walking algorithms.* The former concentrates on achieving the lowest complexity possible, in this case $O(\log n)$ per point query which is performed by using sophisticated data structures such as slabs [10], trapezoidal maps (usually with binary search tree [27, 25, 5]), directed acyclic graph (DAG) [2, 3, 7, 5, 18], skip list [44], quad tree, uniform grid [35, 47] (with bucketing in [39]) and data structures based on random sampling [24, 6].

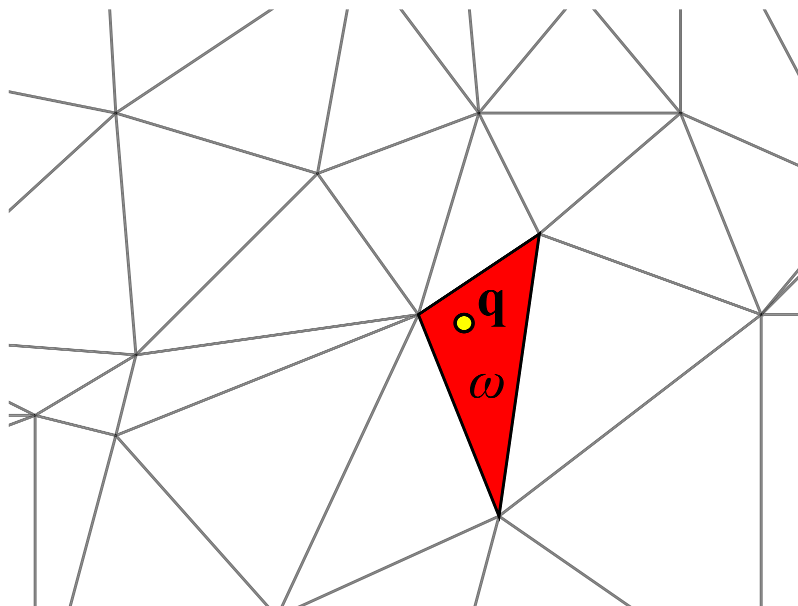

Figure 2.1: Planar point location problem

Despite their low time complexity, these algorithms have some disadvantages. First, the data structures listed above consume generally $O(n)$ amount of memory in the worst case (except slabs with $O(n^2)$ amount of memory in the worst

case) which may be a problem for huge datasets. Second, the implementation effort for most of these structures is nontrivial (especially the modifications of these structures which is possible where the triangulation is modified). Finally, most of these structures are hierarchical and the top level of the hierarchy may become a bottleneck in case of parallelization.

Walking algorithms have not this disadvantages. They do not need any extra memory (they need only connectivity information in a mesh), 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 these algorithms has arisen from the way of locating the triangle ω which contains $\mathbf{q}$. From a starting triangle α chosen as one of the triangles of T and the query point $\mathbf{q}$ the walking strategy makes use of connectivity of the triangle mesh to go through triangles between α and ω (see Figure 2.2). It should be mentioned that many optimal complexity solutions use sophisticated data structures for location of proper α and final location is performed by one of the walking algorithms.

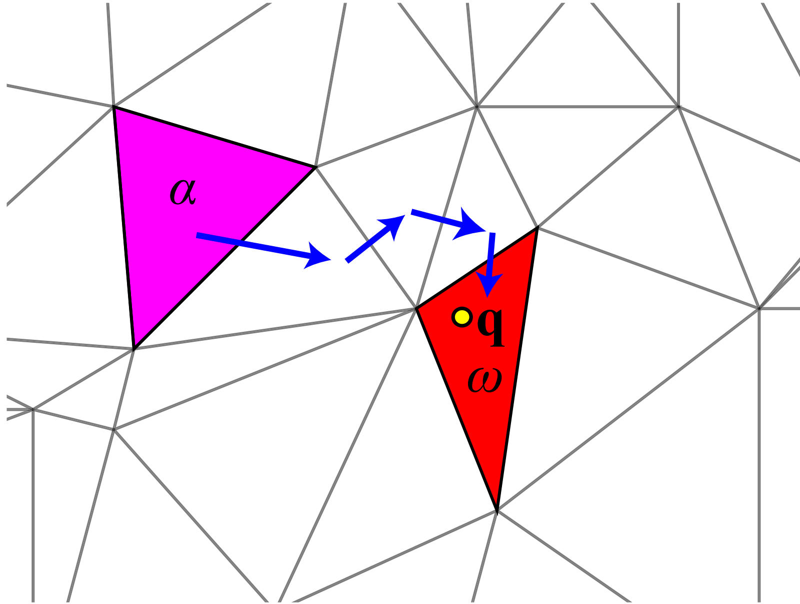

Figure 2.2: Walking strategy

Devillers et al. [8] divided walking strategies to three main types: visibility walk, straight walk and orthogonal walk. The visibility walk (see the whole Section 2.1) makes use of point-inside-triangle tests to determine which triangle is the next in the walk. The most of the published visibility walk algorithms [23, 8, 17] (see Sections 2.1.2, 2.1.3) come out from the first oriented walk published by Lawson in 1977 [20] (see Section 2.1.1). A partial exception may be barycentric walk algorithm published by Sundareswara et al. in 2003 [39] (see Section 2.1.4).

The straight walk algorithms pass all such triangles in the mesh between α and ω that are intersected by a line $\overrightarrow{\mathbf{pq}}$ where $\mathbf{p}$ is a point inside α [21, 8]. The

orthogonal walk algorithms pass all the triangles in the mesh between α and ω which are intersected by the lines λ_x, λ_y collinear with coordinate axes which go through $\mathbf{p}$ and $\mathbf{q}$ where $\mathbf{p}$ is a point inside α [8] ($\mathbf{p} \in \lambda_x$ and $\mathbf{q} \in \lambda_y$).

The starting triangle α may be chosen randomly, by the use of hierarchical structures, by the help of additional information or by a special way, e.g. as the closest triangle to $\mathbf{q}$ from a set A of randomly chosen triangles from T, $\|A\| \ll \|T\|$ [23] (see Section 2.4). A proper choice rapidly improves the speed and expected computational complexity of the algorithm (e.g. from $O(\sqrt{n})$ to $O(\sqrt[3]{n})$ for the algorithm without additional data storage published by Mücke et. al. [23], later analyzed in [9]).

2.1 Visibility Walk Algorithms

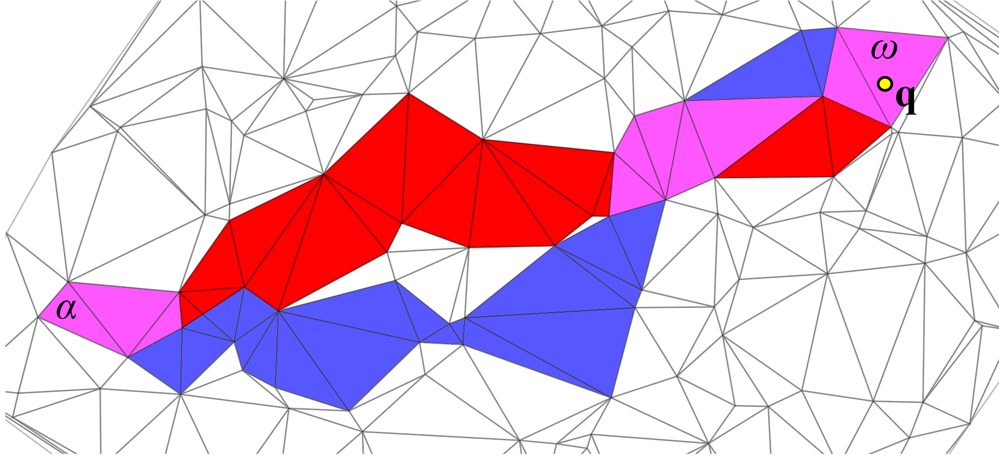

Figure 2.3: Example of visibility walk algorithms (blue path of the remembering stochastic walk in Section 2.1.2 and red path of the barycentric walk in Section 2.1.4, shared triangles are pink)

2.1.1 Lawson's Oriented Walk

This very simple visibility walk algorithm was first published by C. L. Lawson [20]. The algorithm uses the planar orientation edge test (see Equation 1.1) to determine which triangle is the next one in its walk. Assuming that vertices of γ are in the same orientation (CW or CCW) for each triangle $\gamma \in T, \gamma = \mathbf{lrs}$, the result from Equation 1.1 (after substitution of $\mathbf{l}, \mathbf{r}, \mathbf{p}$ coordinates) indicates the position of $\mathbf{p}$ against the oriented edge ϵ_{lr}, where $\mathbf{p}$ is the tested point. Figure 2.4 shows the resulted signs of orientation tests for the triangle $\tau = \mathbf{t_0 t_1 t_2}$ with CCW vertices orientation.

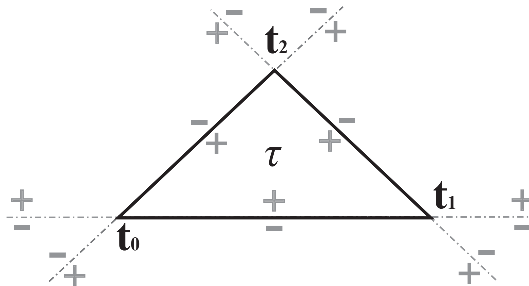

Figure 2.4: The orientation test for triangle with CCW orientation of vertices

Algorithm 2.1 describes the simple Lawson's oriented walk. We assume CCW orientation of triangle vertices in all following pseudo codes. The input and output is the same for almost all further walking algorithms, so it is in further

pseudo codes only if it is different. The walk starts in a triangle α. In each step of the algorithm we test the position of the query point $\mathbf{q}$ against edges of the current triangle τ until we find the edge ϵ_{ab} for which the third vertex $\mathbf{c}$ of τ lies on the opposite side of ϵ_{ab} than $\mathbf{q}$. Then the walk continues to the next triangle over ϵ_{ab}. If a matching edge does not exist in τ then $\tau = \omega$ and τ contains $\mathbf{q}$.

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

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 trough ε;
  |   |   | found = false;
  |   |   | break; // terminates the foreach cycle
  |   | end
  | end
end
// now τ contains q
return τ;
```

Algorithm 2.1: Lawson's oriented walk

The simple Lawson's oriented walk algorithm uses edges of τ for tests in a deterministic order which depends on the arrangement of edges in triangles. The arrangement of edges in triangles is generated during construction of triangulation and it causes that walk may loop in some specific configurations of the triangle mesh [8, 41] (see Figure 2.5). It is shown that for planar Delaunay triangulation is Lawson's oriented walk totally correct [12, 41].

2.1.2 Remembering Stochastic Walk

We know from Section 2.1.1 that the simple Lawson's oriented walk algorithm uses edges of τ for tests in a given order and this method may loop for a non-Delaunay triangulation. For such a triangulation it is necessary to choose the tested edges of τ in a random order. This modification is called *stochastic* [8]. Furthermore, it is not necessary to test the edge incident with the previous triangle in the walk (let us denote previous triangle as ψ), hence we remember the previous triangle ψ and skip the orientation test for the relevant edge ϵ ($\epsilon \in \tau \wedge \epsilon \in \psi$). This improvement is called *remembering* [8] and it may save up to one orientation test for each triangle. It brings significant speedup. Therefore, one

or two orientation tests are needed for each triangle during the walk (except α).

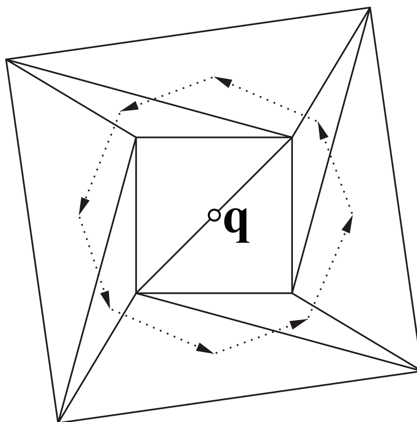

Figure 2.5: Loop of Lawson's oriented walk [41]

Let us denote vertices of τ as $\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}$, ($\tau = \mathbf{t_0 t_1 t_2}$). The algorithm 2.2 describes the remembering stochastic walk algorithm (in the following text we use a shortcut *RSW* for this algorithm). We can see an example of RSW in Figure 2.3 (blue color). Note that we can use the remembering improvement also for the non-stochastic Lawson's oriented walk. We call such an algorithm *remembering walk* (RW). The application of remembering walk is advisable for Delaunay triangulation because the stochasticity is not necessary and consumes a significant amount of algorithm speed.

```
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" improvement condition
        if ψ is not neighbor of τ trough ε_lr then
            if orientation2D(l,r,q) < 0 then
                ψ = τ;
                τ = neighbor of τ trough ε_lr;
                found = false;
                break; // terminates the for cycle
            end
        end
    end
end
return τ;
```

Algorithm 2.2: Remembering stochastic walk

2.1.3 Fast Remembering Stochastic Walk

In each step of the RSW algorithm, we performed one or two orientation tests because the orientation test for the edge to previous triangle ψ is skipped (except the triangle α where up to three tests may be performed). For each triangle $\tau, \tau \neq \omega$ on the particular walk we can choose the next triangle on the basis of one orientation test [17]. If the test for the edge $\epsilon, \epsilon \in \tau$, where the neighbor over ϵ is not ψ, returns that ϵ is not suitable to go through it, we continue the walk through another edge $\xi \in \tau, \xi \neq \epsilon$, where the neighbor through ξ is not ψ. We obviously performed exactly one orientation test for each triangle.

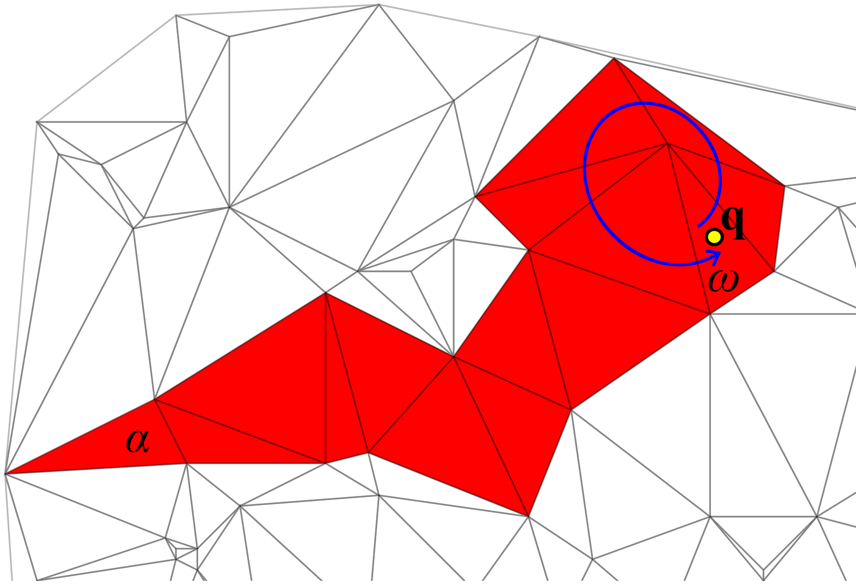

Figure 2.6: Loop of fast RSW

The walk goes correctly but the problem occurs when we need to determine the triangle ω and terminate the walk (see Figure 2.6). The solution presented in [17] uses this fast walk in k steps and after ω is located by the RSW algorithm. Naturally the value of k should depends on n, where n is the number of vertices in the triangle mesh. The question is which value of k is optimal. Kolingerová presents this dependency as a multiple of the average walk length. According to her special choice of the triangle α (see Section 2.4), the expected average walk length is presented as $O(\sqrt[3]{n})$, more precisely between $2\sqrt[3]{n}$ and $2.15\sqrt[3]{n}$. The experiments show that the optimal value of k can be computed as $1.15\sqrt[3]{n}$ and the improvement is much lower than expected (only 8% saved edge tests).

The stochastic version of [17] is described by the pseudo code in the Algorithm 2.3. Naturally, for Delaunay triangulation, the algorithm may be also in non-stochastic version (let us call *fast remembering walk* this modification). Note that the value of k corresponds to the stochastic version and experiments with the non-stochastic version may return different results.

```
Input:
    • the query point q
    • the chosen starting triangle α, α ∈ T
    • the chosen number of fast walk steps k
Output:
    • the triangle ω which contains q

triangle τ = α;
triangle ψ = τ; // previous triangle is initialized as τ
for i = 1 to k do
    edge ϵ = random edge of τ, ψ is not neighbor of τ trough ϵ;
    point l = first vertex of ϵ;
    point r = second vertex of ϵ;
    ψ = τ;
    if orientation2D(l, r, q) < 0 then
        τ = neighbor of τ trough ϵ;
    else
        edge ξ = second edge of τ, ξ ≠ ϵ, ψ is not neighbor of τ trough ξ;
        τ = neighbor of τ trough ξ;
    end
end
return remembering_stochastic_walk(q, τ);
```

Algorithm 2.3: Fast remembering stochastic walk

2.1.4 Barycentric Walk

This algorithm differs from visibility walk algorithms presented before that it uses the barycentric coordinates of the triangle (see Equations 1.8 - 1.12 in Section 1.3.1) instead of the orientation test (see Equation 1.1). Barycentric coordinates have a very significant property - the final triangle ω has all barycentric coordinates of the query point $\mathbf{q}$ non-negative. This property was utilized by Sundareswara et al. in very specific visibility walk algorithm [40].

The main idea of this algorithm (see Algorithm 2.4) is really simple. In each step, the algorithm computes barycentric coordinates of $\mathbf{q}$ for the current triangle $\tau = \mathbf{lrs}$. If the coordinates are non-negative then $\omega = \tau$. Otherwise we choose the next triangle as a neighbor over the edge given by two vertices with a maximum value of the correspondent barycentric coordinate components. We can transform the problem to the inverse problem (we are looking for a vertex $\mathbf{s}$ with minimal barycentric value) and the searched edge is the edge ϵ_{lr} which is opposite to $\mathbf{s}$. Both cases ($\mathbf{q}$ inside the triangle ω and outside of the triangle τ) are illustrated in Figure 2.7, where barycentric coordinates are $\mathbf{b}(\omega) = (0.25, 0.35, 0.4)$ and $\mathbf{b}(\tau) = (-0.75, -0.25, 2)$.

The main advantages of this algorithm are that it works in triangulations where the triangles may have a random orientation and it does not loop for non-Delaunay triangulations [40]. Sundareswara claims that the number of visited triangles by the barycentric walk is a bit lower in average than by other visibility walk algorithms (see Figure 2.3, where the barycentric walk is compared with the RSW algorithm). Our experiments really confirm that the barycentric walk

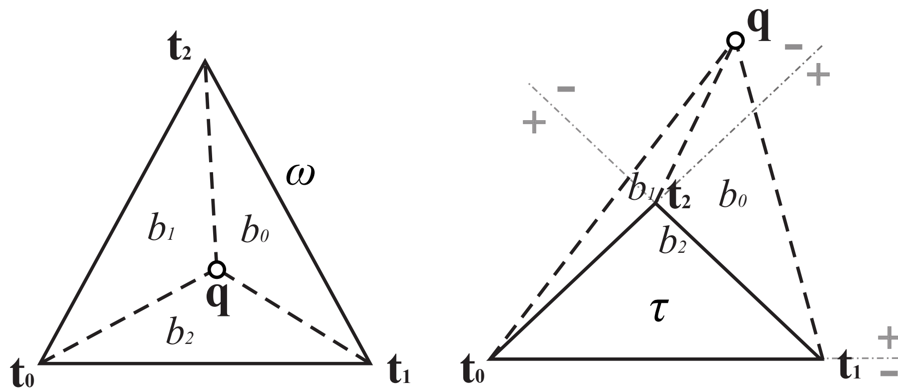

Figure 2.7: Barycentric coordinates of **q** inside (ω) and outside (τ) of triangle

is shorter in average than other visibility walk algorithms (see Section 2.6).

```
triangle τ = α = t0t1t2;
boolean found = false;
double min;
edge ε;

while not found do
    double b0 = b0(t0, t1, t2, q);
    double b1 = b1(t0, t1, t2, q);
    double b2 = 1 − b0 − b1;
    ε = edge t1t2;
    min = b0;
    if  b1 < min then
        ε = edge t2t0;
        min = b1;
    end
    if  b2 < min then
        ε = edge t0t1;
        min = b2;
    end
    if  min < 0 then
        τ = neighbor of τ trough ε;
    else
        found = true;
    end
end

return τ;
```

Algorithm 2.4: Barycentric walk

2.2 Straight Walk Algorithm

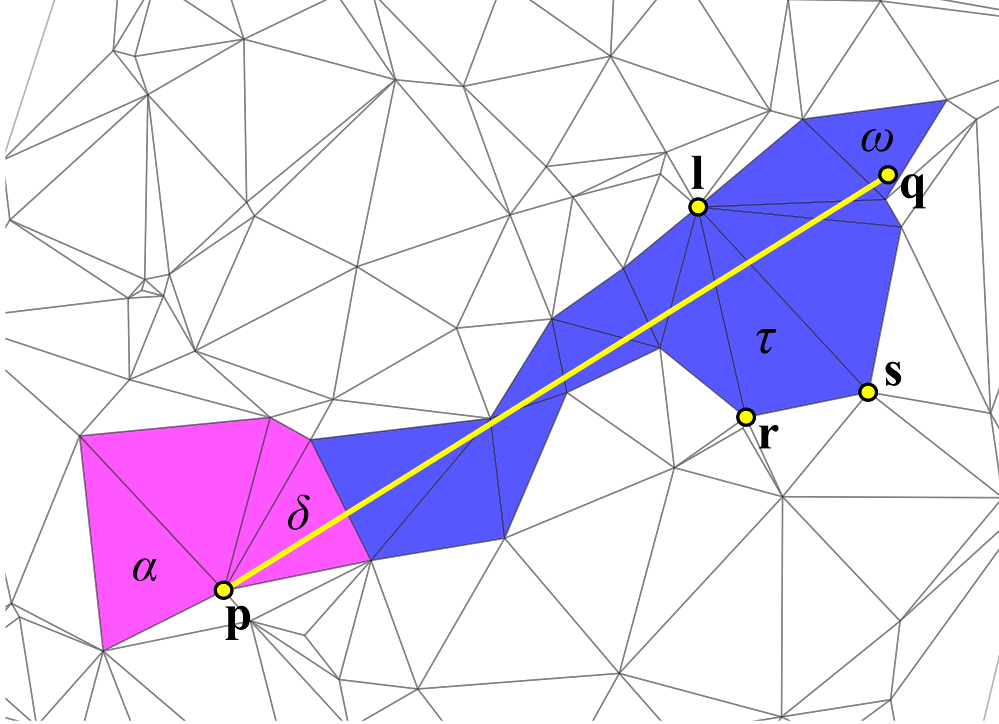

Figure 2.8: Straight walk algorithm

Generally, the straight walk passes all triangles in the triangulation T intersected by the line $\overrightarrow{\mathbf{pq}}$ where $\mathbf{p}$ is a point, $\mathbf{p} \in \alpha = \mathbf{lrp}$. The standard straight walk algorithm [8] is composed of two steps: initialization step and straight walk step.

Pseudo code of the algorithm is given as Algorithm 2.5 [8] and an example of the walk is in Figure 2.8. The initialization step is colored pink and the straight walk is light blue. The triangle δ is such a triangle where the initialization step ends and straight walk begins.

In the initialization step (pink color in Figure 2.8), $\mathbf{p}$ is chosen as one of the triangle α vertices and a triangle γ incident to $\mathbf{p}$ which is intersected by the line segment $\overrightarrow{\mathbf{pq}}$ must be found. Then $\mathbf{r}$ and $\mathbf{l}$ are switched. Now the line segment $\overrightarrow{\mathbf{pq}}$ has $\mathbf{r}$ on the right and $\mathbf{l}$ on the left side. During the initialization step, one orientation test is needed for each visited triangle and the number of visited triangles is at most the degree of $\mathbf{p}$.

After the initialization is completed, the straight walk (blue color in Figure 2.8) may start. For each triangle τ, $\tau = \mathbf{lrs}$ in the straight walk, the line $\overrightarrow{\mathbf{pq}}$ goes into $\tau, \tau \neq \alpha$ through the edge ϵ_{lr} . Depending on the $orientation(\mathbf{p}, \mathbf{q}, \mathbf{s})$, new $\mathbf{r}$ (or new $\mathbf{l}$) is set to $\mathbf{s}$. If $orientation(\mathbf{p}, \mathbf{q}, \mathbf{s}) < 0$ then $\mathbf{r} = \mathbf{s}$, otherwise $\mathbf{l} = \mathbf{s}$. The singular case $orientation(\mathbf{p}, \mathbf{q}, \mathbf{s}) = 0$ is added to one of these situations. Now the straight walk goes out of τ through the new edge ϵ_{lr}. By testing on

which side of ϵ_{lr} 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 ϵ_{lr}. Therefore the straight walk evidently needs two orientation tests per triangle and some orientation tests in the initialization step.

```
// initialization step
triangle τ = α = lrs;
point p = s;
if orientation2D(p, q, r) > 0 then
    while orientation2D(p, q, l) > 0 do
        r = l;
        τ = neighbor of τ trough ε_pl;
        l = vertex of τ where l ≠ p, l ≠ r;
    end
else
    repeat
        l = r;
        τ = neighbor of τ trough ε_pr;
        r = vertex of τ where r ≠ p, r ≠ l;
    until orientation2D(p, q, r) > 0 ;
end
switch(l, r);
// now pq→ has r on the right and l on the left side
// straight walk - following the line segment pq→
while orientation2D(l, r, q) < 0 do
    τ = neighbor of τ trough ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
    if orientation2D(p, q, s) < 0 then
        r = s;
    else
        l = s;
    end
end
return τ;
```

Algorithm 2.5: Straight Walk

2.3 Orthogonal Walk Algorithm

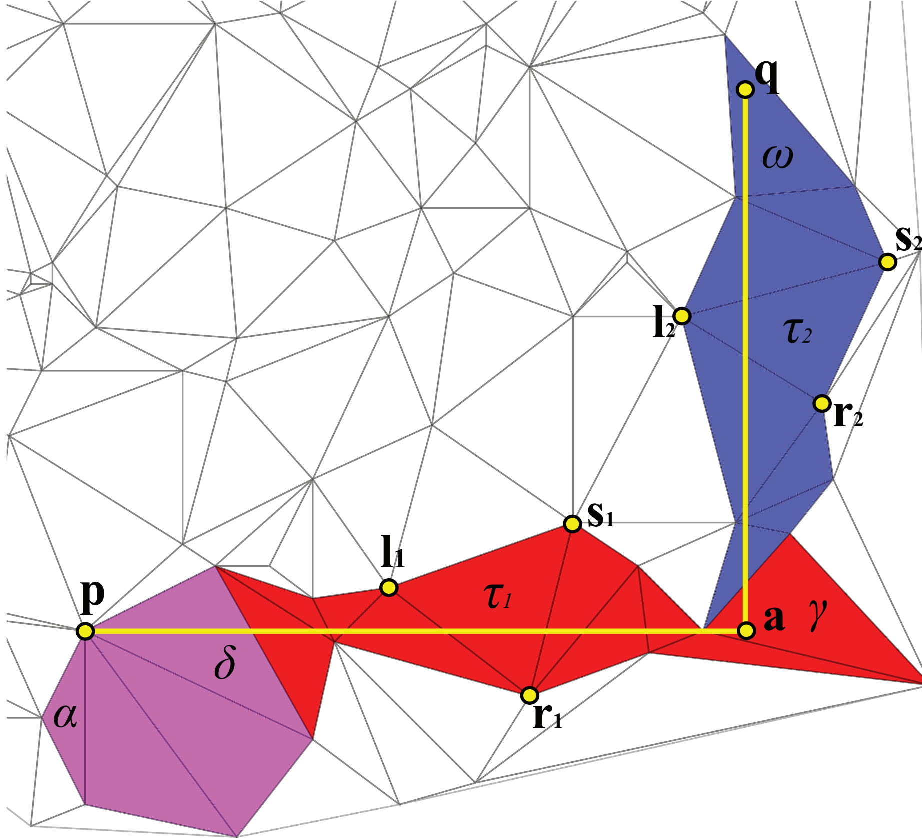

Figure 2.9: Orthogonal walk algorithm

The planar orthogonal walk was proposed by Devillers at al. [8] 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 1.1) were substituted by comparisons of coordinate values. An orthogonal walk is usually longer than other walks (see Figure 2.9) but the cost of its tests is significantly lower which results in a faster location.

Denote $\mathbf{p}$ as one of the triangle α vertices. A new point $\mathbf{a}$ is set as $\mathbf{a} = (q_x, p_y)$. In the following text, we describe one case of orthogonal walk where $\mathbf{q}$ is above and to the right of $\mathbf{p}$ ($q_x > p_x, q_y > p_y$). Other cases are simply analogous. The algorithm is described in Algorithm 2.6 and consists of three steps. In the initialization step (pink color in Figure 2.9), the algorithm looks for a triangle $\delta = \mathbf{lrp}$ incident to $\mathbf{p}$ which is intersected by the line segment $\lambda_x = \overrightarrow{\mathbf{pa}}$ which is collinear with the horizontal axis x. Then $\mathbf{r}$ and $\mathbf{l}$ are switched. Now the line segment λ_x has $\mathbf{r}$ bellow and $\mathbf{l}$ above. Note that vertices of all triangles are still in CCW order.

If δ is found then the horizontal walk step (red color in Figure 2.9) may begin. This step is following λ_x until the current triangle contains $\mathbf{a}$. For each triangle

τ in the horizontal walk the edge ϵ_{lr} of τ is the edge used to cross to this triangle. In such a triangle, we denote the vertex against the edge ϵ_{lr} as $\mathbf{s}$. The edge ϵ_{lr} to the next triangle is found by comparing s_y with a_y. If s_y is higher than a_y then the walk continues over ϵ_{lr} where $\mathbf{l} = \mathbf{s}$, otherwise the walk continues over ϵ_{lr} where $\mathbf{r} = \mathbf{s}$. The horizontal walk ends if $\mathbf{a}$ is inside τ ($orientation2D(\mathbf{l}, \mathbf{r}, \mathbf{a}) \geq 0$).

The vertical walk step is following the line segment $\lambda_y = \overrightarrow{\mathbf{aq}}$ which is collinear with the vertical axis y. At the beginning of a vertical walk step, $\mathbf{l}, \mathbf{r}, \mathbf{s}$ are chosen such that $\mathbf{l}$ is to left from λ_y and $\mathbf{r}$ is to right from λ_y ($l_x \leq q_x$ and $r_x \geq q_x$). Then the vertical walk in the y direction (blue color in Figure 2.9) may start. For each triangle τ in the vertical walk the edge ϵ_{lr} of τ is the edge used to cross to this triangle. In such a triangle, we denote the vertex against the edge ϵ_{lr} as $\mathbf{s}$. The edge ϵ_{lr} to the next triangle is found by comparing s_x with q_x. If s_x is lower than q_x then the walk continues over ϵ_{lr} where $\mathbf{l} = \mathbf{s}$, otherwise the walk continues over ϵ_{lr} where $\mathbf{r} = \mathbf{s}$. The vertical walk ends if $\mathbf{q}$ is inside τ ($orientation2D(\mathbf{l}, \mathbf{r}, \mathbf{q}) \geq 0$).

In the initialization step, one comparison is needed for each visited triangle. Three comparisons are needed for each triangle during the walk. Orientation tests are usually used very rarely because the short-circuit boolean operators are used and test is performed only if the condition before is false. However, minimally two orientation tests are needed for each location (one in horizontal and one in vertical direction). An example of orthogonal walk is in Figure 2.9. The triangle, where the initialization step ends, is denoted as δ. The triangle, where the horizontal walk step ends and the vertical walk step begins, is denoted as γ.

```
// initialization step
triangle τ = α = lrs;
point p = s;
point a = (q_x, p_y);
// we describe the case where q is above and to the right of 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
        r = l;
        τ = neighbor of τ trough ε_pr;
        l = vertex of τ where l ≠ p, l ≠ r;
    end
else
    repeat
        l = r;
        τ = neighbor of τ trough ε_pl;
        r = vertex of τ where r ≠ p, r ≠ l;
    until r_y > p_y ;
end
switch(l, r);
// now τ is intersected by pa and l is above p and r is below p (p_y ≤ l_y and p_y ≥ r_y)
// note that short-circuit boolean operators are used
// traverses the triangulation T in the direction of the horizontal axis x
while (l_x < a_x and r_x < a_x) or orientation2D(l, r, a) < 0 do
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ l, s ≠ r;
    if s_y > a_y then
        l = s;
    else
        r = s;
    end
end
// now τ contains a
if r_x > q_x then
    s = l;
    l = vertex of τ where l ≠ r, l ≠ s;
end
// now τ is intersected by aq, l is to left and r is to right from q (l_x ≤ q_x and r_x ≥ q_x)
// traverses the triangulation T in the direction of the vertical axis y
while (l_y < q_y and r_y < q_y) or orientation2D(l, r, q) < 0 do
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ l, s ≠ r;
    if s_x > q_x then
        r = s;
    else
        l = s;
    end
end
// now τ contains q
return τ;
```

Algorithm 2.6: Orthogonal walk

2.4 Choice of the Starting Triangle

They are two main types of these strategies: the strategies using additional data structures and the strategies which do not consume additional memory. We aim at letter strategies in this subsection.

The simplest way how to choose the starting triangle is to choose α as a random triangle of the triangulation T. If we have an additional information about the triangulation we can choose α in some clever way. For example, if we know the range of the triangulation T, we can choose α for all locations in T as a triangle which contains the point in the middle of T.

Mücke [23] proposed the choice of α without preprocessing and additional structures, where the starting triangle is chosen as the closest triangle to $\mathbf{q}$ from a set A of randomly chosen triangles from T, $\|A\| \ll \|T\|$. This simple modification improves performance of all walking algorithms. Because the distances are computed only for one-to-one comparison, it is sufficient to compute the quadratic distance. The point-triangle distance problem is substituted by point-point distance problem where the triangle is substituted by one of triangle's edge midpoints (midpoint is the middle point of the edge). What is the optimal size of A? Mücke presented that $\lceil \sqrt[3]{n} \rceil$ is the optimal size of A for RSW algorithm where n is a number of vertices in a mesh. The choice of the starting triangle is described in Algorithm 2.7. Our test shows that the optimal size of A is $O(\sqrt[3.5]{n})$ dependent [36].

```
Input:
    • the query point q
    • the triangulation T of n vertices
Output:
    • the starting triangle α

triangle α;
double dist = max double value;
for i = 1 to ⌈∛n⌉ do
    triangle σ = random triangle from T;
    point p = midpoint of the first edge of σ;
    double dist_new = (p_x − q_x)^2 + (p_y − q_y)^2;
    if dist_new < dist then
        α = σ;
        dist = dist_new;
    end
end
return α;
```

Algorithm 2.7: Choice of the starting triangle

2.5 Our Modifications of Planar Walking Algorithms

2.5.1 Distance Fast RSW

As we know from Section 2.1.3, the problem of the fast RSW [17] is how to recognize the walk is actually in ω. Our modification published in [36] computes the distance of each triangle τ from the query point $\mathbf{q}$. As in Section 2.4, the point-triangle distance problem is substituted by the point-point distance problem where a triangle is substituted by one of its vertices and only quadratic distances are computed. For distance computation, we use the vertex of τ which is facing the edge we used to go to τ. If the distance is not lower than the last computed distance, the fast RSW ends and ω is found by the standard RSW algorithm.

Naturally, the fast RSW may end before the walk reaches ω. In the worst case, almost all the walk is performed by the standard RSW. The cost for distance computation is not low, therefore the distance is not computed for each triangle but for each k-th triangle. This modification also decreases the probability of fast RSW termination before the algorithm reaches ω. Our tests prove that the optimal value of k is n-dependent. With the optimal choice of the starting triangle [36], $k = \lceil 0.34 \cdot \sqrt[4.58]{n} + 0.06 \rceil$ [36]). Note that this modification may be used also with the non-stochastic version of algorithm for Delaunay triangulation. For such an algorithm, with the optimal choice of α, $k = \lceil 0.23 \cdot \sqrt[4.90]{n} + 0.18 \rceil$. The stochastic version is described in Algorithm 2.8.

```
triangle τ = α;
triangle ψ = τ; // previous triangle is initialized as τ
double dist_new = max double value;
double dist;

repeat
    dist = dist_new;
    for i = 1 to k do
        edge ε = random edge of τ, ψ is not neighbor of τ trough ε;
        point l = first vertex of ε;
        point r = second vertex of ε;
        ψ = τ;
        if orientation2D(l, r, q) < 0 then
            τ = neighbor of τ trough ε;
        else
            edge ξ = second edge of τ, ξ ≠ ε, ψ ≠ neighbor of τ trough ξ;
            τ = neighbor of τ trough ξ;
        end
    end
    point p = vertex of τ facing edge to previous triangle ψ;
    dist_new = (p_x − q_x)^2 + (p_y − q_y)^2;
until dist_new > dist ;

return remembering_stochastic_walk(q, τ);
```

Algorithm 2.8: Distance fast RSW

2.5.2 Flag Fast RSW

Another way how to solve the problem of the fast remembering stochastic walk algorithm from Section 2.1.3 is to use an additional integer flag for each triangle. This solution consumes an additional linear amount of memory and works very simply. Each search query has a unique identification number (id) and each visited triangle is flagged with this id. If we visit a triangle which has been already flagged with id of the current location, then the fast RSW algorithm is looping and it is terminated. The final location is performed again by the standard RSW algorithm (see Algorithm 2.9).

```
triangle τ = α;
triangle ψ = τ; // previous triangle is initialized as τ
int k = unique identification number of this location;
repeat
    τ_loc = k;
    edge ϵ = random edge of τ, ψ is not neighbor of τ trough ϵ;
    point l = first vertex of ϵ;
    point r = second vertex of ϵ;
    ψ = τ;
    if  orientation2D(l, r, q) < 0 then
        τ = neighbor of τ trough ϵ;
    else
        edge ξ = second edge of τ, ξ ≠ ϵ, ψ ≠ neighbor of τ trough ξ;
        τ = neighbor of τ trough ξ;
    end
until τ_loc = k ;
// now τ has been visited twice during this walk
return remembering_stochastic_walk(q, τ);
```

Algorithm 2.9: Flag fast RSW

This modification has two cardinal disadvantages which make algorithm unpractical. The former, algorithm consume additional amount of memory required for flag. The latter and more significant, algorithm is useless for parallel locations because two parallel location may change flags so fiddly that algorithm loops or returns results in a long time. Another disadvantage may be necessity of data structure modification (adding a flag) which is not always possible.

2.5.3 Progressive Visibility Walk

The computation of barycentric coordinates from Equations 1.8 - 1.12 can be rewritten as Equations 2.1, 2.2 and this is the main principle of our algorithm. Analogically to barycentric coordinates b_i, we define coordinates c_i (*orientation coordinates* in the following text - see Equations 2.3 - 2.6 - we use only numerator of barycentric coordinates which is one 2D orientation test). Apparently if the vertices of tested triangle $\sigma = \mathbf{t_0 t_1 t_2}$ are in CCW order and $b_i \leq b_j \leq b_k$ then $c_i \leq c_j \leq c_k$ for $i, j, k \in \{0, 1, 2\}$ where $i \neq j, i \neq k, j \neq k$. Note that if the vertices of the tested triangle $\sigma = \mathbf{t_0 t_1 t_2}$ are in CW order and $b_i \leq b_j \leq b_k$ then $c_i \geq c_j \geq c_k$. This piece of knowledge allows us to use Algorithm 2.4 with

orientation coordinates instead of barycentric coordinates. It improves speed performance of the algorithm.

$$b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = \frac{orientation2D(\mathbf{t}_{[(i+\mathbf{1}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{2}) \mod \mathbf{3}]}, \mathbf{v})}{orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})} \tag{2.1}$$

$$\sum_{i=0}^{2} b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = 1 \tag{2.2}$$

$$c_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = b_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) \cdot orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}) \tag{2.3}$$

$$c_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = orientation2D(\mathbf{t}_{[(i+\mathbf{1}) \mod \mathbf{3}]}, \mathbf{t}_{[(i+\mathbf{2}) \mod \mathbf{3}]}, \mathbf{v}) \tag{2.4}$$

$$\sum_{i=0}^{2} c_i(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}) \tag{2.5}$$

$$c_2(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}) - c_0(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) - c_1(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}, \mathbf{v}) \tag{2.6}$$

The following improvement allows us to use only one orientation test per visited triangle at the cost of additional memory. We remember $\sigma_{ori} = orientation2D(\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2})$ (see 2d orientation test in Equation 1.1) for each triangle $\sigma \in T$ where $\sigma = \mathbf{t_0}\mathbf{t_1}\mathbf{t_2}$. The additional memory demands are linear but still significantly lower than additional memory demands of hierarchical structures. Note that this additional information may be useful for other applications, too, because σ_{ori} is a double of the triangle surface and gives information about the triangle size.

Let us describe the algorithm which is given by pseudo code in Algorithm 2.10. We modify the non-stochastic version of remembering walk algorithm. As we know, in the remembering walk, we remember which triangle was the previous one. In this algorithm, we also remember the orientation coordinate for the edge we used to go to the current triangle τ (denote it c_{prev}). The inverted value of c_{prev} is an orientation coordinate for the edge to the previous triangle in the current triangle τ (denote it c_i where $c_i = -c_{prev}$). Then the orientation coordinate for another edge of τ is computed (let us denote it c_j). For the computation of the orientation coordinate c_k of the last remaining edge of τ, we utilize remembered τ_{ori} and Equation 2.6 ($c_k = \tau_{ori} - c_i - c_j$). Apparently only one orientation test is needed for each visited triangle during the walk (except the first triangle where c_{prev} does not exist and where two orientation tests are needed). Note that a disadvantage may be also a necessity of the data structure modification which is not always allowed.

```
// initialization step
triangle τ = α = t0t1t2;
triangle ψ;
double c0 = orientation2D(t1, t2, q);
double c1 = orientation2D(t2, t0, q);
double c2 = τori − c0 − c1;
edge ϵ = edge t1t2;
double min = c0;
if  c1 < min then
|  ϵ = edge t2t0;
|  min = c1;
end
if  c2 < min then
|  ϵ = edge t0t1;
|  min = c2;
end
while min < 0 do
|  ψ = τ;
|  τ = neighbor of τ trough ϵ;
|  double ci = −min;
|  edge ϵ = first edge of τ, ψ is not neighbor of τ trough ϵ;
|  point l = first vertex of ϵ;
|  point r = second vertex of ϵ;
|  double cj = orientation2D(l, r, q);
|  double ck = τori − ci − cj;
|  if cj < ck then
|  |  min = cj;
|  else
|  |  min = ck;
|  |  edge ξ = second edge of τ, ξ ≠ ϵ, ψ is not neighbor of τ trough ξ;
|  |  ϵ = ξ;
|  end
end
return τ;
```

Algorithm 2.10: Progressive visibility walk

2.5.4 Normal-line Straight Walk

The first idea of this modification published in [37] is to simplify the initialization step of the straight walk algorithm from Section 2.2 to a constant number of operations. The second idea is to use a cheaper operation than the orientation test used in Section 2.2.

The description of this modified straight walk algorithm is possible to find in the pseudo code Algorithm 2.11. A fundamental prerequisite for the initialization step is to suitably choose the point $\mathbf{p}$. In Section 2.2, $\mathbf{p}$ is chosen as one vertex of the starting triangle α but it is not necessary. The main idea is to choose $\mathbf{p}$ in the way that no other operations in the initialization step are needed.

First, the point $\mathbf{s}$ is chosen as the closest vertex from α to $\mathbf{q}$. The edge ϵ_{lr} is the edge of $\alpha = \mathbf{lrs}$. Next, $\mathbf{p}$ is chosen on ϵ_{lr} where $\mathbf{p} \neq \mathbf{l}, \mathbf{r}$. Now the straight walk step may start.

In other words, the straight walk algorithm from Section 2.2 works as follows.

For each triangle $\tau, \tau = \mathbf{lrs}$ in the straight walk (except α) the edge ϵ_{lr} of τ is the edge that has been used to cross to τ and $\mathbf{s}$ is the vertex of τ facing ϵ_{lr}. The line segment $\overrightarrow{\mathbf{pq}}$ (let us denote it λ) goes out of τ through the edge ξ which is determined by the means of using the orientation test. If $orientation(\mathbf{p}, \mathbf{q}, \mathbf{s}) < 0$ then $\xi = \epsilon_{ls}, \xi \in \tau$, otherwise $\xi = \epsilon_{rs}, \xi \in \tau$. By testing on which side of ξ the point $\mathbf{q}$ lies, it is decided whether τ contains $\mathbf{q}$ or whether the walk must go on.

For each triangle τ the position of $\mathbf{s}$ is tested against the line segment $\lambda = \overrightarrow{\mathbf{pq}}$ and only $\mathbf{s}$ is changing during the walk, therefore it is possible to use an implicit line equation test instead of the orientation test to speed up the process. The implicit line equation of λ is computed in the initialization step (Equations 1.2, 1.3, 1.4) and the position of $\mathbf{s}$ is found by a substitution into Equation 1.5.

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 each triangle during the walk. 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 normal-line $\lambda_n, \lambda_n \perp \lambda, q \in \lambda_n$ is computed in the initialization step (Equation 1.6, 1.7).

The orientation test for detection, whether the triangle ω was found, is replaced with the position test of λ_n and $\mathbf{s}$. If $\mathbf{s}$ lays on the other side of λ_n than $\mathbf{p}$ then the straight walk ends. A situation is possible where $\tau \neq \omega$ at the end of the straight walk (see Figure 2.10). Thus the RSW algorithm is always 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 RSW. However, this situation is not probable and degradation of this modified straight walk to RSW is not a significant problem because RSW is not dramatically worse (see Section 2.6).

```
// initialization step
τ = α = lrs where s is the closest vertex to q;
p = point on edge ε_lr where l,r ≠ p (for example p is midpoint of ε_lr);
line λ = line segment pq;
line λ_n = line segment orthogonal to pq where q ∈ λ_n;
// following the line segment λ from p to q
while position(λ_n, s) < 0 do
    if position(λ, s) < 0 then
        r = s;
    else
        l = s;
    end
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
end
return remembering_stochastic_walk(q, τ);
```

Algorithm 2.11: Normal-line straight walk

The example of the normal straight walk algorithm is given in Figure 2.10. The straight walk step is colored red and the final location by RSW is light

blue. The triangle γ is the triangle where the straight walk step ends and RSW starts, the edge ϵ_{lr} is the edge used to go to γ. The point $\mathbf{s}$ is the point of γ, whose position at the other side of normal-line λ_n than $\mathbf{p}$ causes the end of the straight walk.

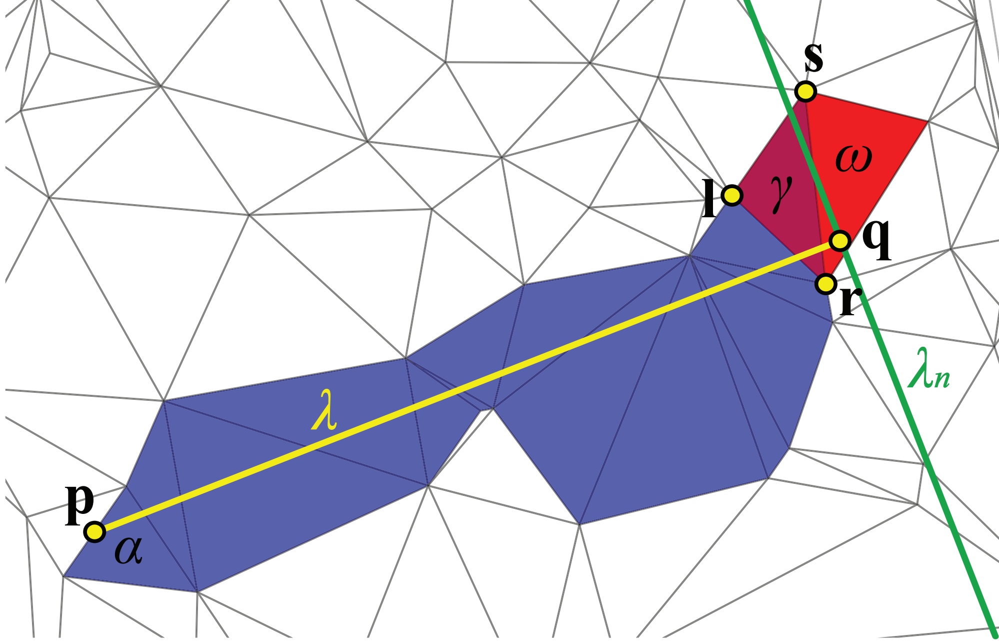

Figure 2.10: Normal-line straight walk algorithm

2.5.5 Improved Orthogonal Walk

Analogous to Section 2.5.4, the first idea of this modification is to simplify the initialization step of the orthogonal walk algorithm from Section 2.3 to a constant number of operations. The second idea is to use a smaller number of coordinate comparisons per each visited triangle and not to use orientation tests at the cost that we use RSW for final location of ω analogously to Section 2.5.4. The description of this modified orthogonal walk algorithm is possible to find in the pseudo code Algorithm 2.12.

Now let us explain our modification. First, the algorithm must choose a starting point $\mathbf{p}$ anywhere inside α. Let us assume $\mathbf{q}$ is above and to the right of $\mathbf{p}$ $(q_x > p_x, q_y > p_y)$, other three possibilities would be analogous. In the initialization step we choose $\mathbf{s} = (s_x, s_y)$ as a vertex of α with the maximal horizontal value x. The vertices $\mathbf{l}, \mathbf{r}, \mathbf{s}$ of α are again in CCW order.

Now the walk in the direction of horizontal axis x may start. This particular walk is following the line segment $\lambda_x = \overrightarrow{\mathbf{pa}}$ collinear with horizontal axis x where $\mathbf{a} = (q_x, p_y)$. For each triangle $\tau, \tau = \mathbf{lrs}$ in the horizontal walk (except α) the edge ϵ_{lr} of τ is the edge used to cross to τ and $\mathbf{s}$ is the vertex of τ facing ϵ_{lr}. The line segment λ_x goes out of τ through the edge $\xi, \xi \in \tau$. The edge ξ is determined by the means of using the vertical coordinate comparison test. If

$s_y < p_y$ then $\xi = \epsilon_{ls}$, otherwise $\xi = \epsilon_{rs}$. The end of horizontal walk is detected by the means of using horizontal coordinate comparison. If **s** is to right of **q** ($s_x > q_x$) then the horizontal walk ends, otherwise the walk continues through the edge ξ to the next triangle.

The vertical walk in y axis direction needs a simple initialization step, too. The point **s** is chosen as a vertex of τ with the maximal vertical value y. Now the walk in the direction of the horizontal axis y may start. This particular walk is following the line segment $\lambda_y = \overrightarrow{\mathbf{aq}}$ collinear with the vertical axis y. The edge ϵ_{lr} of τ is the edge used to cross to τ and **s** is the vertex of τ facing ϵ_{lr}. The line segment λ_y goes out of τ through the edge $\xi, \xi \in \tau$. The edge ξ is determined by the means of using the horizontal coordinate comparison test. If $s_x > q_x$ then $\xi = \epsilon_{ls}$, otherwise $\xi = \epsilon_{rs}$. The end of the vertical walk is detected by the means of using the vertical coordinate comparison. If **s** is above **q** ($s_y > q_y$) then the vertical walk ends, otherwise the walk continues through edge ξ to the next triangle.

After the vertical walk ends, τ is close to the triangle ω which contains the query point **q**. We use only two coordinate comparisons and no orientation tests per each visited triangle at the cost of the estimated result. The final location is performed by the RSW algorithm from Section 2.1.2 and usually is very short (see experimental results in Section 2.6).

```
// initialization step
p = a point generated anywhere inside α;
// we describe the case where q is above and to the right of p (q_x > p_x, q_y > p_y),
   other cases are analogous
τ = α = lrs where s is the vertex with maximal x coordinate;
// traverses the triangulation T in the direction of the horizontal axis x
while s_x < q_x do
    if s_y < p_y then
        r = s;
    else
        l = s;
    end
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
end
τ = lrs where s is the vertex with maximal y coordinate;
// traverses the triangulation T in the direction of the vertical axis y
while s_y < q_y do
    if s_x < q_x then
        l = s;
    else
        r = s;
    end
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
end
return remembering_stochastic_walk(q, τ);
```

Algorithm 2.12: Modification of orthogonal walk

Figure 2.11 shows an example of our modification of orthogonal walk. The triangle γ is a triangle where the horizontal walk stops and the vertical walk begins. The triangle δ is the final triangle of our orthogonal walk where the vertical walk ends. Figure contains points **s** which cause end of the horizontal walk ($\mathbf{s}_\gamma$) and of the vertical walk ($\mathbf{s}_\delta$).

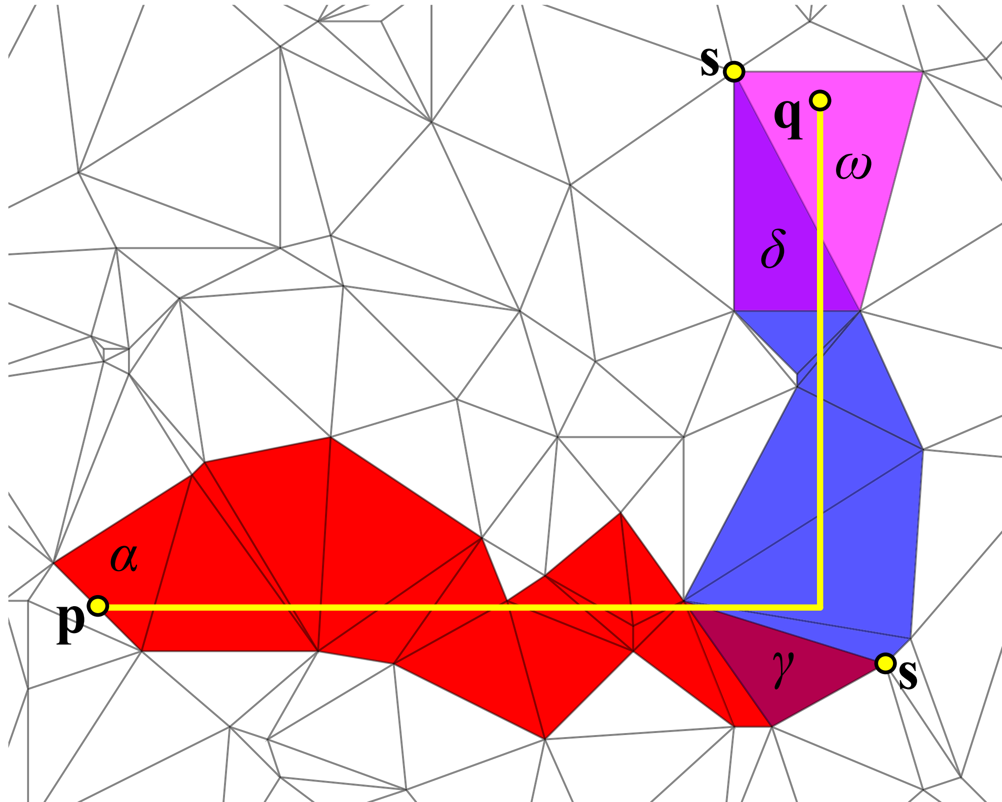

Figure 2.11: Improved orthogonal walk algorithm

2.6 Experimental Results

We tested the following algorithms: Lawson's oriented walk (Section 2.1.1), remembering walk (RW), remembering stochastic walk (RSW - see Section 2.1.2), fast RW, fast RSW (Section 2.1.3), barycentric walk (Section 2.1.4), straight walk (Section 2.2), orthogonal walk (Section 2.3) and our modifications of these algorithms (see subsections of Section 2.5): distance fast RW, distance fast RSW, flag fast RW, flag fast RWS, progressive visibility walk, normal straight walk and our improved orthogonal walk.

Tests were performed on three Delaunay triangulations of different number of points ($n = 10^4, 10^5, 10^6$) randomly distributed on the square. We use algorithms without a special choice of the first triangle (α is chosen randomly). Note that for the algorithms fast RW, fast RSW, distance fast RW and distance fast RSW, the way of choosing k is not described in relevant sources [17, 36], if α is chosen randomly. Therefore the values of k were estimated and are not probably optimal. 10^6 randomly generated points were located by each algorithm on each triangulation.

Selected results are in Table 2.1. 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 some algorithms consists of two values, the former value concerns the walk and the latter one concerns usually the final location by RSW (eventually by RW for non-stochastic versions of algorithms). Only one exception is the orthogonal walk algorithm from Section 2.3, where the former value of #test is the number of coordinate comparisons and the latter is the number of orientation tests. The algorithms were coded in Java with double precision floating point arithmetic.

Note that presented results are rather partial and serve especially for a better orientation of a reader. More detailed tests are planed in the future work (see Section 4.1). Furthermore, despite of relatively good results, some of the presented algorithms (i.e. deterministic variants of Lawson's oriented walk, e.g. RW) are not available for non-Delaunay triangulations.

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**		
	10^4 vertices (19952 Δ)			10^5 vertices (199909 Δ)			10^6 vertices (1999810 Δ)		
Lawson's oriented walk	118.01	208.15	6.33	371.39	647.47	39.20	1172.34	2033.38	214.38
Remembering walk	118.02	160.46	5.49	368.99	496.75	34.42	1174.44	1574.06	185.27
RSW	115.28	153.08	12.48	367.75	482.68	68.35	1148.23	1500.84	281.37
Fast RW [a]	57.0 + 63.82	57.0 + 87.21	4.58	181.0 + 196.83	181.0 + 265.67	32.78	575.0 + 627.1	575.0 + 841.28	181.74
Fast RSW [a]	57.0 + 61.7	57.0 + 82.41	11.99	181.0 + 196.65	181.0 + 258.72	66.63	575.0 + 623.86	575.0 + 816.17	283.30
Barycentric walk	105.4	316.19	5.73	333.22	999.65	43.97	1027.71	3083.13	232.74
Straight walk	107.75	214.86	4.44	338.13	675.61	39.98	1071.96	2143.31	204.1
Orthogonal walk	133.78	408.69 + 3.46	5.05	433.93	1309.29 + 3.52	34.74	1381.47	4151.87 + 3.52	159.88
Distance fast RW [b]	27.37 + 90.27	27.37 + 122.51	4.95	239.08 + 134.09	239.08 + 181.13	31.55	1137.47 + 30.5	1137.47 + 42.06	168.97
Distance fast RSW [b]	27.9 + 88.25	27.9 + 117.29	12.38	256.69 + 116.08	256.69 + 153.18	67.78	1133.51 + 27.88	1133.51 + 37.49	285.01
Flag fast RW	122.34 + 2.6	122.34 + 4.59	4.56	370.98 + 3.96	370.98 + 6.42	29.67	1169.96 + 9.08	1169.96 + 13.27	171.03
Flag fast RSW	118.75 + 3.85	118.75 + 6.08	11.46	367.6 + 7.2	367.6 + 10.45	65.02	1138.67 + 18.42	1138.67 + 25.1	275.21
Progressive visibility walk	105.78	106.78	3.86	333.3	334.3	27.94	1037.32	1038.32	146.85
Normal straight walk	107.1 + 1.76	215.2 + 3.84	3.61	339.74 + 1.76	680.48 + 3.85	31.98	1072.71 + 1.77	2146.41 + 3.85	165.70
Improved orthogonal walk	134.61 + 1.81	269.23 + 3.91	3.23	432.57 + 1.76	865.14 + 3.84	27.89	1368.95 + 1.76	2737.89 + 3.84	142.97

Table 2.1: Comparison of algorithms with randomly chosen α

[a] $k = \lfloor 0.6 \cdot \sqrt{n} + 0.5 \rfloor$
[b] $k = \lceil \frac{1}{2} \cdot \sqrt[5]{n} \rceil$

Chapter 3

Point Location on Star-shaped Polyhedron Surface

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 [16, 14, 26]. Here, the surface triangulation T is an original irregular mesh parametrized onto the unit sphere using a spherical parametrization [4, 19, 30, 15] and T' is a regular spherical mesh. During the sampling process, for each vertex $\mathbf{q}$ of T', it is necessary to find the triangle ω from T which contains $\mathbf{q}$. Apparently, it is a star-shaped polyhedron point location problem (see Figure 3.1).

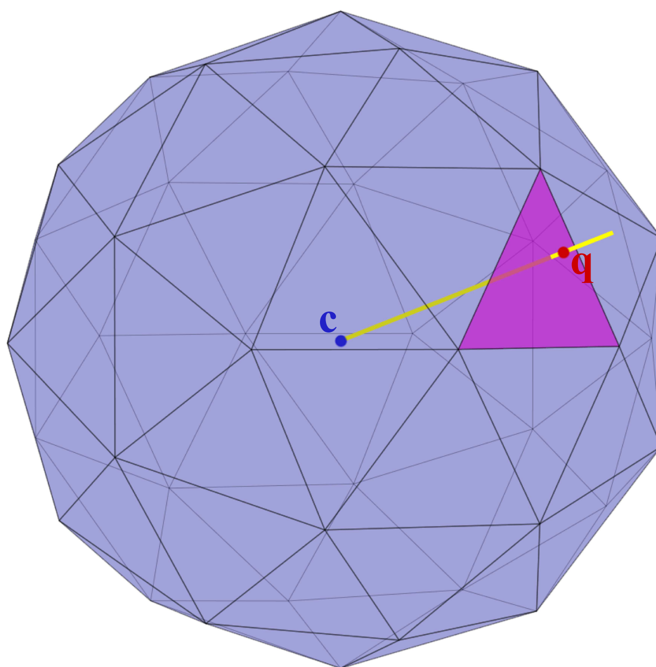

Figure 3.1: Star-shaped polyhedron point location

3.1 Wu's Barycentric Walk

Wu et al.[42] proposed a spherical modification of a planar location algorithm by Sundareswara at al.[40] (see Section 2.1.4) 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 $\mathbf{q}$ in the current triangle τ to determine which neighbor triangle is closer to $\mathbf{q}$ and will be the next one to visit. Wu at al. also proposed the choice of the first triangle using subdivision of the regular octahedron. The disadvantage of Wu's algorithm is its limitation to a spherical surface.

The algorithm is described in Algorithm 3.1. Wu et al. projects $\mathbf{q}$ to each triangle $\tau = \mathbf{t_0 t_1 t_2}$ on the walk as $\mathbf{q}'$ ($\mathbf{q}'$ is the intersection point of the line $\overrightarrow{\mathbf{cq}}$ and the plane given by τ). The implicit line equation of the plane ρ given by τ is used (see Equations 1.13, 1.14). Wu et al. suggest $\mathbf{c} = (0, 0, 0)$. Therefore $\mathbf{q}'$ can be computed using Equations 3.1, 3.2 by the substitution of coefficients a, b, c, d of plane ρ and of query point $\mathbf{q}$.

$$\mathbf{q}' = k \cdot \mathbf{q} \tag{3.1}$$

$$k(\rho, \mathbf{q}) = -\frac{d}{a \cdot q_x + b \cdot q_y + c \cdot q_z} \tag{3.2}$$

Then the barycentric coordinates of $\mathbf{q}'$ in τ are computed. Wu et al. do not present a particular way of computing of the barycentric coordinates but they are likely to use planar computation of barycentric coordinates from Euqations 1.8 - 1.12 where the least changing Cartesian coordinate of the plane ρ is ignored [10]. There are three possibilities how $\mathbf{q}'$ is computed. If the half line segment $\overrightarrow{\mathbf{cq}}$ is intersecting the plane given by τ ($k > 0$) then the walk continues as in Section 2.1.4 (through the edge opposite to the vertex of τ with the minimal barycentric coordinate). If the half line segment $\overrightarrow{\mathbf{qc}}$ is intersecting the plane given by τ ($k < 0$) then the walk continues other way round than in Section 2.1.4 (through the edge opposite to the vertex of τ with the maximal barycentric coordinate). In the singular case, where the line $\overrightarrow{\mathbf{cq}}$ is parallel with the plane given by τ ($k \to \pm\infty$), the next triangle is chosen randomly as a neighbor of τ which is not the previous triangle.

This algorithm also uses a special data structure to choose a good starting triangle α. It uses l levels of detail of the regular octahedron subdivision. Each triangle σ in the m-th level of the regular octahedron subdivision (where $m < l$) contains the pointer to the triangle in the $(m+1)$-th level of the regular octahedron subdivision which contains the centroid of σ. Each triangle σ in the lowest level of the regular octahedron subdivision contains the pointer to the triangle from T which contains the centroid of σ. Wu et al. also present that the best subdivision level l depends on the number of vertices n in T and on the number of the query points n_q. The way how l is computed is presented in Equation 3.3 [42].

$$l(n, n_q) = \left\lfloor \log_2 \left(\frac{n_q}{2 \cdot \ln\left(2 \cdot \sqrt{n}\right)} + \frac{1}{2} \cdot \sqrt{\left(\frac{n_q}{\ln\left(2 \cdot \sqrt{n}\right)} \right)^2 + 4 \cdot n_q} \right) \right\rfloor \tag{3.3}$$

```
triangle τ = α = t0t1t2;
triangle ψ;
boolean found = false;
double extremum; // may be minimum or maximum (it depends on the current k)
edge ϵ;
while not found do
    plane ρ = plane given by τ = t0t1t2;
    double k = k(ρ, q);
    if k → ±∞ then
        ϵ = random edge of τ, ψ is not neighbor of τ trough ϵ;
        extremum = −1;
    else
        point q′ = k · q;
        // computing barycentric coordinates
        double b0 = b0(t0, t1, t2, q′);
        double b1 = b1(t0, t1, t2, q′);
        double b2 = 1 − b0 − b1;
        ϵ = edge t1t2;
        extremum = b0;
        // if k > 0 then we are finding minimum, otherwise we are finding maximum
        if k > 0 then
            if b1 < extremum then
                ϵ = edge t2t0;
                extremum = b1;
            end
            if b2 < extremum then
                ϵ = edge t0t1;
                extremum = b2;
            end
        else
            if b1 > extremum then
                ϵ = edge t2t0;
                extremum = b1;
            end
            if b2 > extremum then
                ϵ = edge t0t1;
                extremum = b2;
            end
            extremum = −extremum;
        end
    end
    if extremum < 0 then
        ψ = τ;
        τ = neighbor of τ trough ϵ;
    else
        found = true;
    end
end
return τ;
```

Algorithm 3.1: Wu's barycentric walk

The algorithm (see Algorithm 3.2) which searches a proper triangle α is really simple. It starts in a random triangle of the first level of the regular octahedron subdivision. Then the triangle of the current level of the subdivision which contains $\mathbf{q}$ is found by Algorithm 3.1. The walk (performed again by Algorithm 3.1) continues from the triangle from the next level of the subdivision which contains centroid of the current triangle. This procedure is repeated until the walk is in the lowest level of the subdivision (in the l-th level). The walk in the l-th level of the subdivision ends in the triangle which contains $\mathbf{q}$ (as well as at higher levels of subdivision - let us denote γ such a triangle in the l-th level of the subdivision). Algorithm returns the starting triangle α as a triangle pointed from γ (it is the triangle from original mesh T which contains centroid of γ). Such a triangle is usually close to the triangle $\omega, \omega \in T$ which really contains $\mathbf{q}$.

Input:

- the query point $\mathbf{q}$
- the regular octahedron O with l levels of subdivision

Output:

- the starting triangle α

```
int level = 1;
triangle τ = random triangle from the first level of O subdivision;
while level < l do
    τ = Wu's_barycentric_walk(q, τ); // see Algorithm 3.1
    τ = pointed triangle of τ containing its centroid;
    level = level + 1;
end
// now τ ∈ T
return τ;
```

Algorithm 3.2: The choice of α using regular octahedron subdivision

3.2 Our 3D Walking Algorithms

3.2.1 Remembering Walk

We propose a modification of planar remembering stochastic walk algorithm (see Section 2.1.2) published in [38] as another possibility for point location on a star-shaped polyhedron. Our modification uses the center point $\mathbf{c}$ of the polyhedron. Instead of the classical edge test (Equation 1.1) which is used in the planar point location, we use spatial orientation facet test (Equation 1.15) which is used for walking in tetrahedral meshes. The decision whether or not the edge ϵ_{lr} of the current triangle τ should be crossed to continue the walk into the next triangle depends on the result after substitution $\mathbf{l}, \mathbf{r}, \mathbf{c}, \mathbf{q}$ to the Equation 1.15 where $\mathbf{q}$ is the query point, $\mathbf{c}$ is the center point and $\mathbf{l}, \mathbf{r}$ are vertices of τ.

Assuming that the vertices of the triangle are in the CCW order from outside the polyhedron in the left-handed coordinate system, the walk continues to the next triangle over the edge ϵ_{lr} if the $orientation3D(\mathbf{l}, \mathbf{r}, \mathbf{c}, \mathbf{q}) > 0$. If the $orientation3D(\mathbf{v}, \mathbf{w}, \mathbf{c}, \mathbf{q}) \leq 0$ for all edges ϵ_{vw} of τ, the triangle τ contains the query point $\mathbf{q}$. In the proposed algorithm (see Algorithm 3.3), we use this simple idea which allows walking on the triangulated surface of a general star-shaped polyhedron, in contrast to Wu's barycentric walk algorithm [42] which allows walking only on a spherical surface.

3.2.2 Plücker Line Coordinates Visibility Walk

This very simple modification of the algorithm from Section 3.2.1 uses Plücker line coordinates (see Equations 1.19, 1.20, 1.21) to determinate which triangle is the next on the walk instead of the standard 3D orientation test (Equation 1.15). Plücker line coordinates of the ray $\overrightarrow{\mathbf{cq}}$ are computed in the initialization step and during the walk only the Plücker line coordinates of edges are computed. Analogous to Algorithm 3.3, the walk continues through the edge ϵ_{lr} if $\overrightarrow{\mathbf{lr}}$ and $\overrightarrow{\mathbf{cq}}$ are in the clockwise orientation (the orientation of two line segments is computed using Equation 1.21). If the orientation of all edges of τ against $\overrightarrow{\mathbf{cq}}$ is CCW then $\tau = \omega$ and it is intersected by the ray $\overrightarrow{\mathbf{cq}}$ (see Figure 1.2 and Section 1.3.2 which describes the orientation test of two lines).

3.2.3 Barycentric Walk

Wu et al. [42] use barycentric coordinates of a triangle and have to solve several singular situations in the algorithm from Section 3.1. Our idea is to use barycentric coordinates of a tetrahedron similar to barycentric coordinates of a triangle in the planar point location (see Section 2.1.4). Tetrahedron is made from the surface triangle by adding of the center point $\mathbf{c}$. The barycentric coordinates of the tetrahedron are computed by Equations 1.16, 1.17, 1.18 and the algorithm is described in Algorithm 3.4 (assuming that the vertices of the

triangle are in the CCW order from outside the polyhedron in the left-handed coordinate system).

```
triangle τ = α;
triangle ψ = τ; // 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];
        if ψ is not neighbor of τ trough ε_lr then
            if orientation3D(l, r, c, q) > 0 then
                ψ = τ;
                τ = neighbor of τ trough ε_lr;
                found = false;
                break; // terminates the for cycle
            end
        end
    end
end

return τ;
```

Algorithm 3.3: Remembering stochastic walk for star-shaped polyhedron

```
triangle τ = α = t0t1t2;
boolean found = false;
double max;
edge ε;

while not found do
    // computing three barycentric coordinates of tetrahedra
    double b0 = b0(t0, t1, t2, c, q);
    double b1 = b1(t0, t1, t2, c, q);
    double b2 = b2(t0, t1, t2, c, q);
    ε = edge t1t2;
    max = b0;
    if b1 > max then
        ε = edge t2t0;
        max = b1;
    end
    if b2 > max then
        ε = edge t0t1;
        max = b2;
    end
    if max > 0 then
        τ = neighbor of τ trough ε;
    else
        found = true;
    end
end

return τ;
```

Algorithm 3.4: Barycentric walk for star-shaped polyhedron

3.3 Point Location in Spherical Coordinate System

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 the spatial algorithms but brings some difficulties, where the planar algorithm does not return a 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 the spatial algorithms from Sections 3.1, 3.2.

The star-shaped polyhedron triangulation mesh T consists of an array of vertices V and an array of faces (triangles) F. Each triangle $\sigma = \mathbf{v_i v_j v_k} \in F$ contains indices of its three vertices $\mathbf{v_i}, \mathbf{v_j}, \mathbf{v_k} \in V$ and of its neighbor triangles $\psi_m, \psi_n, \psi_o \in F$ where ϵ_{jk} is the edge shared with (its neighbor) triangle ψ_m, ϵ_{ki} is the edge shared with ψ_m and ϵ_{ij} is the edge shared with ψ_o. Each vertex $\mathbf{v_i}$ of V can be denoted as a pair $\mathbf{v_i} = (\mathbf{p_i}, \mathbf{h_i})$, where $\mathbf{p_i} = (p_{ix}, p_{iy}, p_{iz})$ is a triple of Cartesian coordinates and $\mathbf{h_i} = (h_{i\varphi}, h_{i\theta})$ is a pair of spherical radian coordinates. The spherical coordinates $(h_{i\varphi}, h_{i\theta})$ of $\mathbf{v_i}$ are computed from Cartesian coordinates using Equations 1.22, 1.23. Note that $\mathbf{c} = (c_x, c_y, c_z)$ is the center point of a star-shaped polyhedron and the range of arctg_2 function is defined as $(-\pi, \pi\rangle$.

The query point $\mathbf{q}$ is given by either Cartesian or spherical coordinates. Assuming that the spherical coordinates (φ, θ) 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.

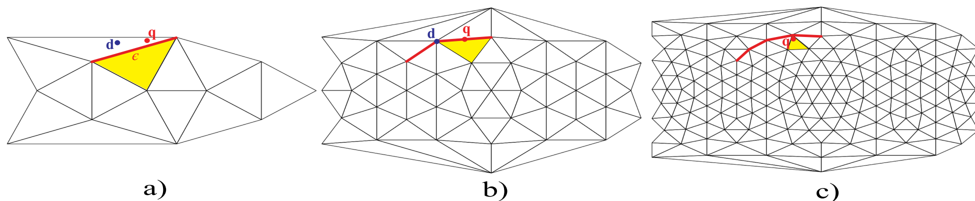

Figure 3.2: Problem of the planar orientation edge test in our simplification of spherical coordinates (the icosahedron (a) and its first (b) and second (c) level of the subdivision, red color represents the surface subdivisions of the edge ϵ)

The problem lies in the fact that spherical coordinate system is a curvilinear coordinate system [22] and the line segment between two points in Cartesian coordinates is an arc in spherical coordinates. It contradicts to our simplifica-

tion where we represented the edges in spherical coordinates as line segments, projecting only their endpoints. Figure 3.2 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 3.2a shows the original icosahedron in spherical coordinates, Figures 3.2b, 3.2c its first and second level of subdivision. A point $\mathbf{d}$ lies originally on an edge ϵ (Figure 3.2b) of the icosahedron, but in spherical coordinates, it may lie outside the edge (Figure 3.2a). The edge ϵ and its subdivisions are bold and colored red. In Cartesian coordinates, a point $\mathbf{q}$ is located in a triangle which is colored yellow in Figures 3.2a, b, c, but in spherical coordinates it may lie on an edge (Figure 3.2b) or even in a different triangle (Figure 3.2a).

Hence our simplification is not geometrically correct and the planar orientation edge test (Equation 1.1) 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 3.5).

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 3.3 (see Figure 3.3a 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 $\mathbf{q}$ (see Figure 3.3b) or $\mathbf{q}$ is near to it. Hence if the planar walk detects that the current triangle τ is a border one, the planar location ends and ω is located by one of spatial algorithms.

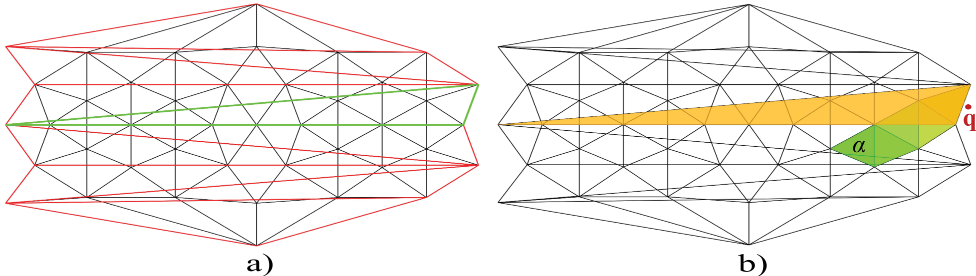

Figure 3.3: Planar triangulation of the icosahedron with shown border triangles colored by red (a) and with location of point in the border triangle (b)

Border triangles are recognized and flagged during computing the spherical coordinates. The detection is rather simple. We substitute the spherical coordinates $\mathbf{h_i}, \mathbf{h_j}, \mathbf{h_k}$ of the vertices $\mathbf{v_i}, \mathbf{v_j}, \mathbf{v_k}$ of the triangle $\tau = \mathbf{v_i}\mathbf{v_j}\mathbf{v_k}$ to the

planar orientation test from Equation 1.1. The result of the $\mathbf{h_i}, \mathbf{h_j}, \mathbf{h_k}$ 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 $orientation2D(\mathbf{h_i}, \mathbf{h_j}, \mathbf{h_k}) > 0$. Note that we use φ coordinate instead of x and θ instead of y in the test from Equation 1.1.

When we are walking on a surface of polyhedron using some spatial point location algorithm (Sections 3.1, 3.2), the resulting path is usually straighter and shorter than the planar walk, especially in case when α is close to $\mathbf{q}$ on the surface but it is on the other side of the mesh in spherical coordinates. Figure 3.4 shows the path of one location process of a planar RSW algorithm in our simplification of spherical coordinates where the passed triangles are filled by gradient from green to yellow color. On the surface (Figure 3.4a), the starting triangle α is quite close to the target triangle ω but in spherical coordinates it is not (Figure 3.4b), so the resulting walk is longer than it is necessary. Despite longer paths this property is not significant because it can be minimized by a clever choose of α (see Section 3.4) and because the planar walking algorithms are faster in average thanks to the cheaper cost of the tests.

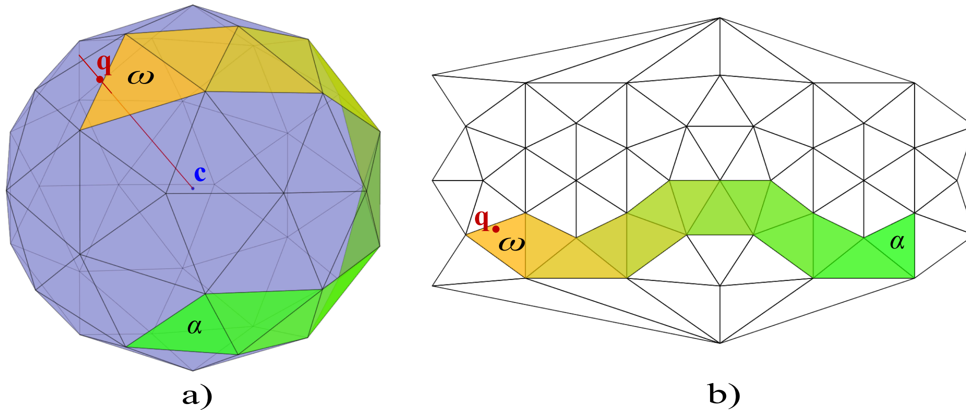

Figure 3.4: Path of a planar walk (the passed triangles are filled by gradient from green to yellow)

3.3.1 Orthogonal Walk Algorithm in Spherical Coordinates

In this section, we present a modification of an orthogonal walk algorithm for our simplification of spherical coordinates. This modification is established on algorithm from Section 2.3. The Cartesian coordinates x and y are substituted by spherical coordinates φ and θ. The orthogonal walk step ends if the current triangle τ is border or in the same case as Algorithm 2.12. The final location is performed by one of the algorithms from Sections 3.1, 3.2.

Figure 3.5 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 3.5a shows the walk in our simplification of spherical coordinates and Figure 3.5b shows the walk in the Cartesian coordinates. The algorithm description is given in Algorithm 3.5.

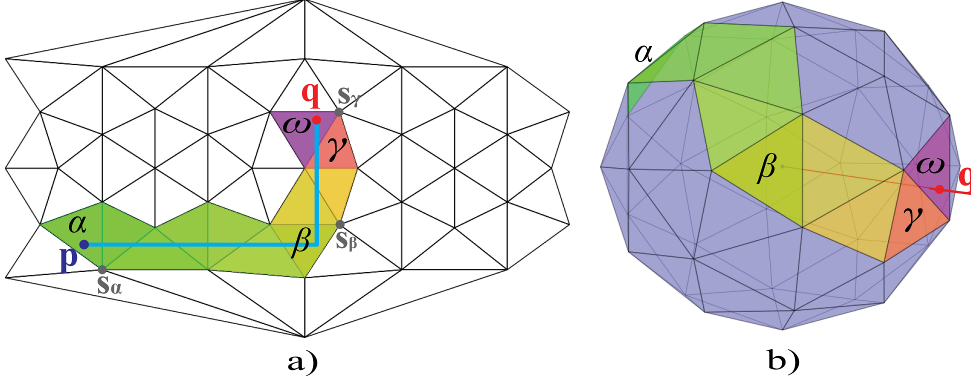

Figure 3.5: The path of our spatial orthogonal walk algorithm in the spherical (a) and the Cartesian (b) coordinates

```
// initialization step
p = a point generated anywhere inside α;
// we describe the case where q is above and to the right of p (q_φ > p_φ, q_θ > p_θ),
   other cases are analogous
τ = α = lrs where s is the vertex with maximal φ coordinate;
// traverses the triangulation T in the direction of the horizontal axis x
while s_φ < q_φ and notBorder(τ) do
    if s_θ < p_θ then
        r = s;
    else
        l = s;
    end
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
end
τ = lrs where s is the vertex with maximal θ coordinate;
// traverses the triangulation T in the direction of the vertical axis y
while s_θ < q_θ and notBorder(τ) do
    if s_φ < q_φ then
        l = s;
    else
        r = s;
    end
    τ = neighbor of τ over ε_lr;
    s = vertex of τ where s ≠ r, s ≠ l;
end
// the final location is done by another walking algorithm (e. g. spatial remembering
   stochastic walk in Section 3.2.1)
return remembering_stochastic_walk(q, τ);
```

Algorithm 3.5: Modification of orthogonal walk for spherical coordinates

3.4 Preprocessing and using of hierarchical structures

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[23] where α is chosen as the closest triangle to $\mathbf{q}$ from a set A of randomly chosen triangles from the mesh T, $\|A\| \ll \|T\|$. For the best performance, Mücke recommends $\|A\| = 2 \cdot \sqrt[4]{n}$ for spatial data and our results confirmed that. But if we choose the starting triangle α as the triangle containing the point $\mathbf{b}$ which is in the middle of the spherical domain ($b_\varphi = 0, b_\theta = 0.5 \cdot \pi$), the performance is very similar to [23].

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 η_{ij} of the grid, the suitable starting triangle α_{ij} is the triangle which contains the center point $\mathbf{q_{ij}}$ of η_{ij}. For the polyhedron whose triangles are very similar, each cell η_{ij} of the uniform grid (see Figure 3.6a) 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.

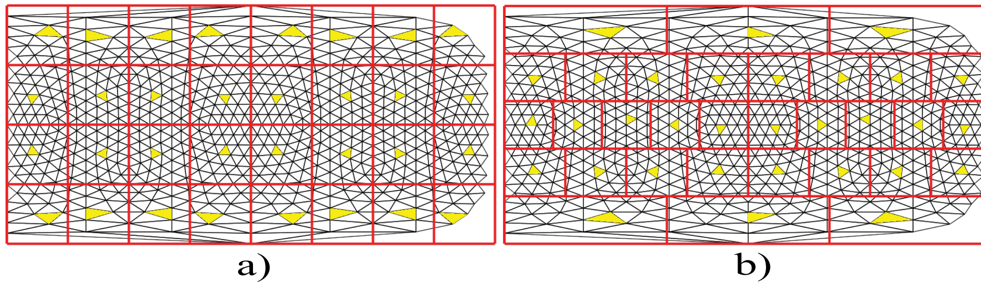

Figure 3.6: The uniform and the nonuniform grid of 32 cells for finding a suitable starting triangle (the starting triangle for each cell of grid is yellow colored)

This consideration leads us to the 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 3.6b). Each cell η_{ij} contains such triangles of T that their area of these triangles on the spherical surface is similar for all η_{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 3.4 and Figure 3.7 - the equations are derived from the definition of determinant of Jacobian matrix for spherical coordinate parametrization [1, 28] 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 3.5, 3.6 describe the computation of k and l_i. Figure 3.6 shows grid structures for the choice of the first triangle where the first triangle for each cell of the grid is colored yellow. Figure 3.6a shows a uniform grid of 32 cells and Figure 3.6b shows the nonuniform grid with the same number of cells. The matching cell η_{ij} of a query point $\mathbf{q} = (\varphi_q, \theta_q)$ is computed identically (see Equations 3.7) for uniform and nonuniform grid (in the uniform grid, l_i is the same for each i). Note that if $q_\theta = \pi$ then $i = k - 1$ or if $q_\varphi = \pi$ then $j = l_i - 1$.

$$S_i = 2\pi \int_a^b f(\theta)d\theta = 2\pi \int_a^b \sin(\theta)d\theta = 2\pi\left[-\cos(\theta)\right]_a^b = 2\pi\left(\cos(a) - \cos(b)\right) \tag{3.4}$$

$$k = \left\lfloor \sqrt{\frac{\pi m}{4}} + 0.5 \right\rfloor \tag{3.5}$$

$$l_i = \left\lfloor \frac{mS_i}{S} + 0.5 \right\rfloor = \left\lfloor \frac{1}{2} m \int_{\frac{i\pi}{k}}^{\frac{(i+1)\pi}{k}} \sin(\theta)d\theta + 0.5 \right\rfloor, i = 0, 1, ..., k-1 \tag{3.6}$$

$$i = \left\lfloor k\frac{\theta_q}{\pi} \right\rfloor, j = \left\lfloor l_i \frac{\varphi_q + \pi}{2\pi} \right\rfloor \tag{3.7}$$

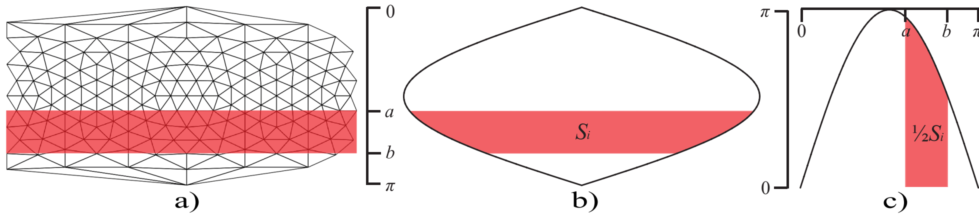

Figure 3.7: An illustrative example to computing an area of the strip (by red) in our simplification of spherical coordinates (a), its real surface area on the unit sphere (b) adjusted for easier computation (c)

3.5 Experimental Results

We tested the following algorithms: Remembering stochastic walk presented in Section 3.2.1 (RSW), Wu's barycentric walk [42] (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.

Selected results are in Table 3.1 (without preprocessing) and Table 3.2 (with preprocessing - see Section 3.4). 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 3.2, 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 [42]. 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 3.2, 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.

To sum up the results without preprocessing (Table 3.1), 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 3.1). 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 [42]. The nonuniform grid is faster than the uniform grid but the difference is not great.

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 3.1: Comparison of algorithms without preprocessing

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 3.2: Comparison of algorithms with preprocessing

Chapter 4

Future Work

4.1 Planar Point Location

As a future work in planar point location, we want to perform more thorough tests of our proposed algorithms. Several number of types of triangulations exists and it is possible that each walking algorithm will return different results for each type of triangulation. Thus we want to compare all walking algorithms over different types of the walking algorithms. For example, we have a suspicion that the barycentric walk algorithm (see Section 2.1.4) may not be safe for non-Delaunay triangulations (as is proclaimed in [40]) and the algorithm may loop for such triangulations. If the algorithm loops for non-Delaunay triangulations then our progressive visibility walk algorithm (see Section 2.5.3) may loop too.

Usually, triangulations are made from datasets which may have specific characters and properties given by assortment of points. Therefore we want to test algorithms on as many datasets as it will be possible. In these tests, we will aim at particular applications, especially utilization of geodesic datasets for geographical applications.

The quality of the walking algorithms is not determined only by their speed. Therefore we want to test several other properties of each algorithm (including efficiency aspects, stability, finality, resistance against singularities and practical applicability).

In other work, we will aim at modification of the walking algorithms for the application in hierarchical clustered data. For a huge datasets, hierarchical cluster structure allows to keep in memory only the highest level triangulation where each vertex represents the spacious parts of the lower level triangulation [32]. The whole lower level triangulations cannot be stored in memory and certain parts are read only if it is needed. Our future modification will be looking for the query point $\mathbf{q}$ on different levels of the cluster subdivision and change the level of detail as needed.

4.2 3D Point Location

As a future work in the point location in tetrahedral meshes we want to test existed algorithms in the same way as is described in Section 4.1. We also have some preliminary ideas of new algorithms which we want also to test. For point location in tetrahedral meshes, we expect a practical application in dynamic proteins research [45] where the point location is performed especially in the construction of Delaunay and regular triangulations of the dynamic proteins models [46].

Analogously to Section 4.1, for a huge tetrahedral meshes, it is impossible to store the whole tetrahedronization in memory. Therefore, we may use a hierarchical cluster structure which allows to keep in memory only the highest level tetrahedral mesh where each vertex represents a spacious part of the lower level tetrahedral mesh [32]. The whole lower level meshes cannot be stored in memory and the certain parts are read only if it is needed. Our future modification will be looking for the query point $\mathbf{q}$ on the different levels of cluster subdivision and change the level of detail as needed.

4.3 Point Location on Surface Triangulation

We presented new walking algorithms for the point location on triangulated surface of general star-shaped polyhedron, where we are looking for a triangle ω which is intersected by the ray $\overrightarrow{\mathbf{cq}}$ ($\mathbf{q}$ is the query point and $\mathbf{c}$ is the center point of star-shaped polyhedron and it is the part of input). The definition of a surface point location problem, in general, is not so straightforward. Therefore, for our possible future work, we define three different particular problem definitions which depend on the position of the query point $\mathbf{q}$.

First, we assume that $\mathbf{q}$ lies on one of the surface triangles. Our goal is to locate this triangle. Second, we assume that $\mathbf{q}$ is near the surface triangulation but does not lie on it. Our goal is to find a proper way how to locate the triangle ω which is optimal for the given criteria. The criteria of optimality may be various with respect to the practical application. Finally, we have a triangulated parametrized surface (e.g. as in [29]) and the query point $\mathbf{q}$ lies on this parametrized surface (generally not necessarily on one of the surface triangles). Our goal is to locate the triangle which "covers" the part of the surface where $\mathbf{q}$ lies. Apparently, in the parametrized surface, we can use a similar idea as in Section 3.3.

Bibliography

[1] Bartsch, H.-J. *Matematické vzorce*, 4 ed. Academia, 2006. 47

[2] Boissonnat, J.-D., and Teillaud, M. The hierarchical representation of objects: the Delaunay tree. In *SCG '86: Proceedings of the second annual symposium on Computational geometry* (New York, NY, USA, 1986), ACM, pp. 260–268. 11

[3] Boissonnat, J.-D., and Teillaud, M. On the randomized construction of the Delaunay tree. *Theoretical Computer Science 112*, 2 (1993), 339 – 354. 11

[4] Certain, A., Popovic, J., DeRose, T., T. Duchamp, D. S., and Stuetzle, W. Interactive multiresolution surface viewing. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH* (1996), pp. 91–98. 36

[5] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry, Algorithms and Applications.* Berlin Heidelberg: Springer, 1997. 11

[6] Devillers, O. Improved incremental randomized Delaunay triangulation. In *Proceedings of the 14th Annual Symposium on Computational Geometry* (1998), pp. 106–115. 11

[7] Devillers, O. The Delaunay hierarchy. *International Journal of Foundations of Computer Science 13* (2002), 163–180. 11

[8] 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. 7, 12, 13, 15, 20, 22

[9] Devroye, L., Mucke, E. P., and Zhu, B. A note on point location in Delaunay triangulations of random points, 1998. 13

[10] Dobkin, D., and Lipton, R. J. Multidimensional searching problems. *SIAM Journal on Computing 5*, 2 (1976), 181–186. 11, 37

[11] Erickson, J. Plücker coordinates. Ray Tracing News 10(3), 1997. 9

[12] Floriani, L. D., Falcidieno, B., Nagy, G., and Pienovi, C. On sorting triangles in a Delaunay tessellation. *Algorithmica 6*, 4 (1991), 522–532. 15

[13] Hodge, W. V. D., and Pedoe, D. *Methods of algebraic geometry*, vol. 1. Cambridge University Press, 1994. 9

[14] Hormann, Labsik, U., and Greiner, G. Remeshing triangulated surfaces with optimal parameterizations. *Computer-Aided Design 33*, 11 (2001), 779–788. 36

[15] Hormann, K., Lévy, B., and Sheffer, A. Mesh parameterization: Theory and practice. In *ACM SIGGRAPH Course Notes* (2007). 36

[16] Kobbelt, L., Vorsatz, J., Labsik, U., and Seidel, H. P. A shrink wrapping approach to remeshing polygonal surfaces. *Computer Graphics Forum, Eurographics '99*, 18 (1999), 119–130. 36

[17] 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* (2006), pp. 221–224. 12, 17, 26, 34

[18] Kolingerová, I., and Žalik, B. Improvements to randomized incremental Delaunay insertion. *Computers & Graphics 26* (2002), 477–490. 11

[19] Labsik, U., Kobbelt, L., Schneider, R., and Seidel, H. P. Progressive transmission of subdivision surfaces. *Computational Geometry 15* (2000), 25–39. 36

[20] Lawson, C. L. *Mathematical Software III; Software for C1 Surface Interpolation.* Academic Press, New York, 1977, pp. 161–194. 12, 14

[21] Mehlhorn, K., and Näher, S. Leda: A platform for combinatorial and geometric computing. *Communications of the ACM 38*, 1 (1995), 96–102. 12

[22] Moon, P., and Spencer, D. E. *Field Theory Handbook.* Springer-Verlag, 1988. 42

[23] 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. 12, 13, 25, 46

[24] Mulmuley, K. Randomized multidimensional search trees: Dynamic sampling. In *Proceedings of the 7th Annual Symposium on Computational Geometry* (1991), pp. 121–131. 11

[25] O'Rourke, J. *Computational Geometry in C.* Cambridge University Press, 1994. 11

[26] Praun, E., and Hoppe, H. Spherical parametrization and remeshing. In *Proceedings of the 30th Annual Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH* (2003), pp. 340–349. 36

[27] Preparata, F. P. Planar point location revisited. In *Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, 1988), Springer-Verlag, pp. 1–17. 11

[28] Rektorys, K. *Přehled užité matematiky*, 7 ed., vol. 1. Prometheus, 2009. 47

[29] Rypl, D. Approaches to discretization of 3D surfaces. In *CTU Reports*, vol. 7 (2). CTU Publishing House, Prague, Czech Republic, 2003. 53

[30] Saba, S., Yavneh, I., Gotsman, C., and Sheffer, A. Practical spherical embedding of manifold triangle meshes. In *International Conference on Shape Modeling and Applications* (2005), pp. 340–349. 36

[31] Shoemake, K. Plücker coordinate tutorial. Ray Tracing News 11(1), 1998. 9

[32] Skála, J., and Kolingerová, I. Clustering geometric data streams. In *SIGRAD 2007: Proceedings of the Annual SIGRAD Conference, Special Theme: Computer Graphics in Healthcare* (Uppsala, Sweden, 2007), pp. 17–23. 51, 52

[33] Skala, V. Barycentric coordinates computation in homogeneous coordinates. *Computers & Graphics 32*, 1 (2008), 120 – 127. 8, 9

[34] Skala, V. Computation in projective space. In *MAMECTIS'09: Proceedings of the 11th WSEAS International Conference on Mathematical Methods, Computational Techniques and Iintelligent Systems* (Stevens Point, Wisconsin, USA, 2009), World Scientific and Engineering Academy and Society (WSEAS), pp. 152–157. 7

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

[36] Soukal, R. Walking algorithms for point location. Diploma thesis, 2008. 6, 25, 26, 34

[37] Soukal, R., and 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), pp. 219–222. 29

[38] Soukal, R., and Kolingerová, I. Star-shaped polyhedron point location with orthogonal walk algorithm. In *ICCS 2010: Proceedings of the 10th International Conference on Computational Science* (Amsterdam, Netherlands, 2010). 40

[39] Su, P., and Drysdale, R. L. S. A comparison of sequential Delaunay triangulation algorithms. In *Proceedings of the 11th Annual Symposium on Computational Geometry* (1995), pp. 61–70. 11, 12

[40] Sundareswara, R., and Schrater, P. Extensible point location algorithm. In *International Conference on Geometric Modeling and Graphics* (2003), pp. 84–89. 18, 37, 51

[41] Weller, F. On the total correctness of Lawson's oriented walk. In *Proceedings of the 10th International Canadian Conference on Computational Geometry* (1998). 15, 16

[42] Wu, Y., He, Y., and Tian, H. A spherical point location algorithm based on barycentric coordinates. In *Proceedings of the 5th Computational Science and Its Applications* (2005), pp. 1099–1108. 37, 40, 48

[43] Yiu, P. The uses of homogeneous barycentric coordinates in plane Euclidean geometry. *International Journal of Mathematical Education in Science and Technology 31*, 4 (2000), 569 – 578. 8, 9

[44] Zadravec, M., and Žalik, B. An almost distribution independent incremental Delaunay triangulation algorithm. *The Visual Computer 21*, 6 (2005), 384–396. 11

[45] Zemek, M., and Kolingerová, I. Hybrid algorithm for deletion of a point in regular and delaunay triangulation. In *Proceedings of the Spring Conference on Computer Graphics* (Budmerice, Slovakia, 2009). 52

[46] Zemek, M., Skála, J., Ivana Kolingerová, I., Medek, P., and Sochor, J. Fast method for computation of channels in dynamic proteins. In *Proceedings of the Vision, Modeling and Visualization* (Konstanz, Germany, 2008), pp. 333–342. 52

[47] Žalik, B., and Kolingerová, I. An incremental construction algorithm for Delaunay triangulation using the nearest-point paradigm. *International Journal of Geographical Information Science 17*, 2 (2003), 119–138. 11

Activities

Reviewed Publications

- Roman Soukal and I. Kolingerová. *Star-shaped Polyhedron Point Location with Orthogonal Walk Algorithm.* Accepted for ICCS 2010: The 10th International Conference on Computational Science, Amsterdam, Netherlands, 2010.
- Roman Soukal and Přemysl Holub. *A Note on Packing Chromatic Number of the Square Lattics.* Electronic Journal of Combinatorics, vol. 17, 2010.
- Roman Soukal and I. Kolingerová. *Straight Walk Algorithm Modification for Point Location in a Triangulation.* EuroCG'09: Proceedings of the 25th European Workshop on Computational Geometry, pp 219-222, Brussels, Belgium, 2009.

Student Publications

- Roman Soukal. *Walking Algorithms for Point Location.* Diploma thesis (supervised by Ivana Kolingerová), University of West Bohemia, Pilsen, Czech Republic, 2008.
- Roman Soukal and Ivana Kolingerová. *Procházkové lokační algoritmy.* SVK: Studenstká vědecká konference, Plzeň, 2008.

Related Talks

- *Efektivní lokace bodu pomocí hierarchických datových struktur.* Center of Computer Graphics and Data Visualization, University of West Bohemia, Czech Republic, May 2010.
- *Walking in a triangulation.* Center of Computer Graphics and Data Visualization, University of West Bohemia, Czech Republic, March 2009.
- *Walking algorithms for point location.* University of Maribor, Slovenia, October 2008.

Stays Abroad

- University of Maribor, Slovenia. October 2008, 1 week.

Participations in Projects

- *Triangulated Models for Haptic and Virtual Reality.* The Grant Agency of the Czech Republic, Project Code GAČR 201/09/0097.
- *Algorithms for Terrain Modeling, Bilateral Cooperation Czech Republic - Slovenia.* The Ministry of Education, Youth and Sports, Project Code KONTAKT 9-06-26/2007-08.