PERGAMON

Computers & Geosciences 27 (2001) 1135–1145

COMPUTERS & GEOSCIENCES

A cell-based point-in-polygon algorithm suitable for large sets of points☆

Borut Žalik[a,*], Ivana Kolingerova[b]

[a] Department of Computer Science, Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia

[b] Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia, Czech Republic

Received 1 August 2000; received in revised form 11 January 2001; accepted 15 January 2001

Abstract

The paper describes a new algorithm for solving the point-in-polygon problem. It is especially suitable when it is necessary to check whether many points are placed inside or outside a polygon. The algorithm works in two steps. First, a grid of cells equal in size is generated, and the polygon is laid on that grid. A heuristic approach is proposed for cell dimensioning. The cells of the grid are marked as being inside, outside, or on the polygon border. A modified flood-fill algorithm is applied for cell classification. In the second step, points are tested individually. If the tested point falls into an inner or an outer cell, the result is returned without any additional calculations. If the cell contains the polygon border, it is possible to determine the local point position. The analysis of time complexity shows that the initialization is finished in $O(n\sqrt{n})$ time, while the expected time complexity for checking an individual point is $O(\sqrt{n})$, where n represents the number of polygon edges. The algorithm works with $O(n)$ space complexity. The paper also gives practical results using artificial and real polygons from a GIS environment.

Keywords: Computational geometry; GIS; Point-in-polygon; Uniform subdivision; Polygon

1. Introduction

A point-in-polygon test is one of the most elementary tests in computational geometry and its applications (Preparata and Shamos, 1985). The problem is easy to state: having a polygon P and an arbitrary point $\boldsymbol{q}$ we would like to know whether point $\boldsymbol{q}$ is inside polygon P or not.

In practice, the point-in-polygon test (frequently also called as a containment test) is applied many times to the same polygon, e.g., if a polygon represents a rectangular window and a clipping process is applied (Skala, 1994a). In some applications, polygons are defined by a large number of vertices. For example, in geographic information systems (GIS), we can easily find such polygons representing roads, banks of rivers, borders of states, etc. It is not unusual to handle polygons with a few thousand or tens of thousands of vertices, and to match many points against such a polygon for containment. Let us imagine a polygon representing the area close to the highway inside which the level of noise exceeds the recommended level. Then, we would like to identify all the houses inside that polygon. In such a situation, a fast algorithm is needed to ensure a rapid feedback of the system.

In this paper, an efficient algorithm designed for repetitive point-in-polygon questions is considered. It works in two steps. In the first step, the edges of a polygon are inserted into a grid of cells in $O(n\sqrt{n})$ time where n is the total number of the polygon's vertices. The positions of the tested points, which are considered in the second step, are then decided locally within the

☆Code available from server at http://www.iamg.org/CGEditor/index.htm.

*Corresponding author. Tel.: +368-62-220-7471; fax: +368-62-211-178.

E-mail address: zalik@uni-mb.si (B. Žalik).

0098-3004/01/$ - see front matter
PII: S0098-3004(01)00037-1

individual cells into which the tested points fall. If the cell is not crossed by any polygon edge, the position of the tested point is determined in the constant time $O(1)$.

The paper is divided into five sections. Section 2 introduces the vocabulary and notations, and gives a brief overview of previous work done on this subject. The algorithm is explained in Section 3. In Section 4, the time and space complexity of the algorithm is estimated and practical results are given. Finally, in Section 5, conclusions are drawn.

2. Background

Let us have n points $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1}$ in the plane and n line segments $l_0 = \boldsymbol{p}_0\boldsymbol{p}_1,\ l_1 = \boldsymbol{p}_1\boldsymbol{p}_2, \ldots, l_{n-1} = \boldsymbol{p}_{n-1}\boldsymbol{p}_0$ connecting these points. They condition a simple polygon if (O'Rourke, 1994)

- the neighbouring line segments meet at only one common point,
- non-neighbouring line segments do not have any point in common (they neither intersect nor touch).

The line segments are called polygon edges and the points where these line segments meet are called the vertices of the polygon. Any polygon not satisfying these two conditions is designated as a non-simple polygon. In computational geometry and computer graphics, simple polygons are preferred.

If a polygon does not contain holes, it splits the plane exactly into two parts: the bounded interior and unbounded exterior. If the polygon contains a finite number of holes, it splits the plane into more than two regions, but only one is unbounded. Using terminology from the theory of geometric modelling (Mortenson, 1985), we define the following:

- A loop is the sequence of edges separating the bounded and unbounded parts of the plane. Each polygon has exactly one loop.
- A ring is the sequence of edges denoting the inside border of the polygon. The rings may be nested forming a ring hierarchy.

It is necessary that the loop and the rings are oriented unambiguously. The loop and the rings at even levels of the hierarchy are oriented in the same way (e.g. in a clockwise direction), whereas the rings at odd levels are oriented in the opposite direction (e.g. anti-clockwise direction). In this way, the material occupied by the polygon is always on the same side (right side) of the polygon's edges (see Fig. 1A).

Beside dividing polygons into simple and non-simple polygons, other classifications also exist (Preparata and Shamos, 1985; O'Rourke, 1994; Mortenson, 1985; Feito et al., 1995). Many algorithms run faster if the considered polygons are convex (Kolingerova, 1995). A polygon is convex if all of its vertices are convex. A polygon vertex is convex if less than a half of the small circle, centred at the considered vertex, is occupied by the polygon, otherwise it is concave (Mortenson, 1985). Vertex $\boldsymbol{a}$ in Fig. 1B is convex whereas vertex $\boldsymbol{b}$ is concave. If the orientation of a polygon is known, the convexity of a vertex is easily determined by calculating the cross product of the oriented edges (vectors) at the considered vertex. Polygon orientation has been formalized by Feito et al. (1995).

The following different algorithms have been suggested to solve the point-in-polygon problem:

- *Ray crossing* (ray intersection) *method* (Manber, 1989; Foley et al., 1990). This is the most popular method. It successfully solves all kinds of polygons. It works in $O(n)$ time (n is the number of the polygon edges). Boundary cases are solved relatively easily and therefore, the algorithm is numerically stable.
- *Sum of angles* (Preparata and Shamos, 1985; Foley et al., 1990). This method also works in $O(n)$ time. However, it is sensitive to the problems of finite arithmetic (Hoffman, 1989) (rounding errors, accumulation of errors) and therefore, less appropriate

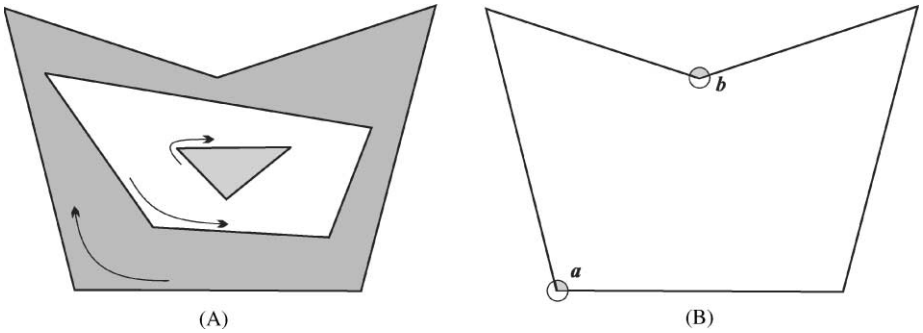

Fig. 1. Loop and nested rings (A), convex and concave polygon vertex (B).

for polygons defined by a large number of vertices. In comparison with the ray crossing algorithm, it is also slower.

- *Swath method* (Salomon, 1978). This requires pre-processing. The polygon is first divided into swaths (trapezoids). That can be done in time $O(n \log n)$ (Seidel, 1991; Žalik, and Clapworthy, 1999). By binary search, the swath is located for the considered point in time $O(\log n)$ and then the ray crossing method is applied to the polygon edges that are located in the swath. The algorithm is useful when more points must be matched against the same polygon.
- *Sign of offset* (Taylor, 1994). This algorithm requires that the polygon is oriented and works only for convex polygons. Let $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$ be the vertices of a polygon and $\boldsymbol{q}$ the tested point. Let us define two vectors $\boldsymbol{a} = (\boldsymbol{v}_j - \boldsymbol{v}_i),\ j = i + 1,\ 0 < i < n\ j = 0, i = n$, and $\boldsymbol{b} = (\boldsymbol{q} - \boldsymbol{v}_i)$. The side of the polygon edge on which the tested point lies can be determined by calculating the cross product between vectors $\boldsymbol{a} \times \boldsymbol{b}$. If the sign of the cross product does not change for all polygon edges, the tested point is inside the polygon. It is clear that the algorithm works in $O(n)$ time.
- *Wedge method* (Preparata and Shamos, 1985). This method works with convex polygons. At first, a polygon is divided into n wedges in linear time using an arbitrary point inside the polygon (e.g. centre of gravity). By means of a binary search, the wedge inside which the point falls is determined in $O(\log n)$ time. Then, by using the cross product, the position of the tested point is determined in a constant time.
- *A method with thin regular slices* (Skala, 1994b). The algorithm is a modification of the wedge method, but the polygon is divided into a large number of slabs/slices. To achieve the desired time complexity, at most two polygon edges should be located at each side of the slab. Since polygon edges in a convex polygon form two monotone chains, the preprocessing requires $O(n)$ time. The slab into which the point falls is then found in constant time $O(1)$ using a simple formula.
- Skala (1996) developed an algorithm using dual representation. It requires pre-processing which is finished in $O(n)$ time, whereas the containment test works in constant time $O(1)$. However, the algorithm works only with convex polygons which do not contain holes.
- *Grid method* (Huang and Shih, 1997). The idea is suitable for a raster-based situation. The polygon is represented by a group of cells—pixels. The colour of the pixel at a given location can be used to determine if the tested point is inside the polygon or not. Clearly, this can be performed always in linear time. However, as Hung and Shih (1997), stated this method is not suitable for vector-based representations of polygons.
- *Triangle-based method* (Feito et al., 1995). The polygon is decomposed into a set of triangles in linear time. Each triangle is defined by a vertex located at the origin, whereas the other two vertices are determined by a polygon edge. The triangles thus obtained are classified as being positively or negatively oriented. A line starting at the origin and containing the tested point is then matched with all the triangles. The algorithm counts the number of intersections with the positively and negatively oriented triangles. If the result is positive, the point is inside the polygon. As reported by the authors (Feito et al., 1995), the algorithm, if carefully implemented, is about twice as fast as the ray crossing method, but it still has $O(n)$ time complexity.
- *A coded coordinate system method* (Chen and Townsend, 1987). For a tested point a coordinate system is introduced, and all polygon vertices are processed against it. It is clear that these vertices are located in one of the quadrants or on the coordinate axes. The quadrants and the coordinate axes are coded by a two-bit code. The position of the consecutive vertices is checked and five different possibilities are identified (i.e. from the movement between the top two quadrants to no movements between quadrants). Then the line-crossing method is applied. A horizontal test line from the tested point to negative infinity is applied. Now, only the edge which has vertices belonging to the second and third quadrants must be tested. The authors reported that the algorithm is faster by about 25 per cent, when compared with the classical ray crossing method.

3. The algorithm

In computational geometry, some kind of pre-processing is often required to speed up repetitive geometric tasks (Preparata and Shamos, 1985). In general, during pre-processing, geometric data are organized according to their position in the space being considered. This permits the examination of a smaller number of geometric elements in the main part of the algorithm, which then runs faster. There are two possible ways of organizing geometric data: a uniform and a non-uniform subdivision of the space of interest.

(a) For a non-uniform subdivision, a large number of approaches exist (e.g., quadtrees, K-D trees, binary trees, octrees) (Samet, 1989, 1990). The main advantage of these approaches is that they adjust themselves to the actual distribution of the input data. However, the data structures and algorithms themselves can be complicated.

(b) Maintaining the uniform plane subdivision is straightforward but the results are not optimal if the input data distribution is irregular. Despite this, the uniform subdivision is one of the most popular acceleration methods in computational geometry and computer graphics. It is applied to many algorithms [e.g. ray-tracing (Fujimoto et al., 1986), searching for the closest neighbour (Preparata and Shamos, 1985), constructing the triangulation on a given set of points (Fang and Piegl, 1993), determining topology from a set of line segments (Žalik, 1999)] and remarkable increases in the speed of solution have been reported. The simplicity of this method has led to the suggestion that it could be used in the development of the spatial databases suitable for GIS (Brobst et al., 1999).

In our situation, the uniform space subdivision method is used. The considered polygon is covered by a grid of cells equal in size. The cells, which are completely outside or inside the polygon, are marked appropriately. If the point to be tested for the inclusion falls within such a cell, its location with respect to the polygon is given immediately. However, some cells contain one or more polygon edges but if the tested point does fall into such a cell, we only need to check only the edges placed in that cell to determine exactly the point position. In real situations, only a small subset of all polygon edges must be tested. We name the described algorithm *CBCA* (Cell-Based Containment Algorithm). It is explained in detail in the following sections.

3.1. Initialization

The initialization of the *CBCA* algorithm consists of three steps:

- forming the grid of cells of the same size,
- rasterizing the edges of the polygon,
- classifying the cells not crossed by any polygon edge as being either inside or outside the polygon.

3.1.1. Forming the uniform subdivision

Given a polygon, the first reasonable question is how many cells are needed to cover it adequately. It is possible to determine the size and number of cells in the uniform subdivision in two ways:

- some heuristics are applied (Fang and Piegl, 1993; Žalik, 1999), or
- the number of cells is fixed (Fujimoto et al., 1986).

We could simply state that more cells produce better results. This consideration is valid if the pre-processing time is not important—namely, the more cells we have, the more time is needed for the rasterizing process and determining the status of the non-crossed cells. This means that we must test more points to eliminate the influence of the initialization. Therefore, a simple heuristic approach is proposed taking into account features of the input polygon.

First, the bounding rectangle of the polygon is determined. To avoid the problems connected with finite arithmetic, the bounding rectangle is stretched a little (by a few per cent of what is considered as ε in Eq. (1)) as suggested by Fang and Piegl (1993).

$$x_{\min} = x_{\min} - \varepsilon, \quad y_{\min} = y_{\min} - \varepsilon,$$
$$x_{\max} = x_{\max} + \varepsilon, \quad y_{\max} = y_{\max} + \varepsilon. \tag{1}$$

The number of cells is then determined in the following way:

$$NoOfCells_x = 2\left\lfloor ratio\sqrt{n}\right\rfloor,$$
$$NoOfCells_y = 2\left\lfloor \frac{\sqrt{n}}{ratio}\right\rfloor, \tag{2}$$

where n is the number of polygon vertices and *ratio* is determined as

$$ratio = \frac{x_{\max} - x_{\min}}{y_{\max} - y_{\min}}. \tag{3}$$

The size of the cells is then trivially determined by

$$Size_x = \frac{x_{\max} - x_{\min}}{NoOfCells_x},$$
$$Size_y = \frac{y_{\max} - y_{\min}}{NoOfCells_y}. \tag{4}$$

All cells are marked as *Undefined* at the start.

3.1.2. Determination of cells containing polygon edges

In the second initialization step, the cells containing parts of the polygon border are determined. If an edge is located in more than one cell, all crossed cells must be determined. These cells are marked as being *Grey*. From Fig. 2, it can be seen that the well-known Bresenham (1965) algorithm does not find all the crossed cells, and therefore, a different type of rasterizing algorithm is

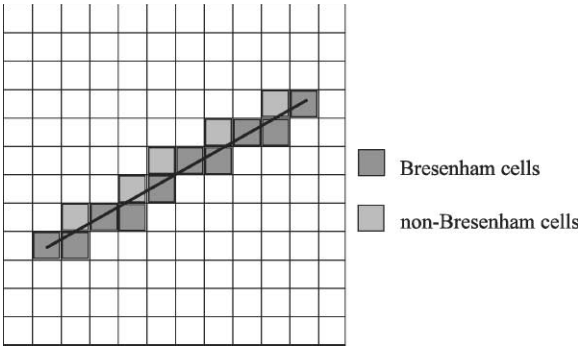

Fig. 2. Code-based algorithm for determining cells crossed by an edge.

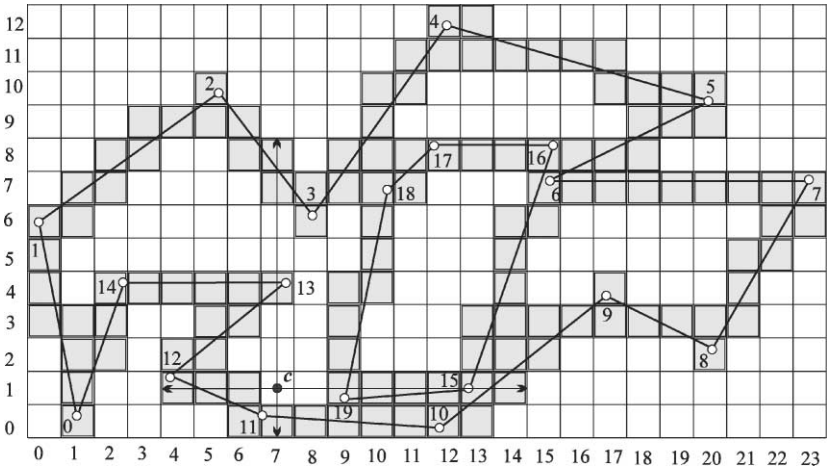

Fig. 3. Determination of *Grey* cells containing polygon edges.

needed. In particular, all cells being crossed by an edge must be determined. The most frequently used algorithm in such situations is undoubtedly Cleary and Wyvill's (1988) algorithm. However, we have used the Code-based algorithm, which finds the so-called non-Bresenham cells by comparing two neighbouring Bresenham cells. These obtain the codes regarding their connectivity (in 2D we distinguish only between vertex and edge connectivity, in 3D, there is also face connectivity). Having these codes, it is easy to determine which non-Bresenham cell has been crossed by the line segment. The main advantages of the Code-based algorithm are:

- It is less sensitive to the accumulation of rounding errors than other algorithms.
- It easily discovers the situations in which the line segments pass through the vertex of the cell, when non-Bresenham cells are not needed.
- The algorithm can be easily adapted to other (faster) algorithms for rasterizing line segments.

Details of the Code-based algorithm developed for the traversing 3D cells (voxels) are given by Žalik et al. (1997).

In this method, each cell may contain more than one polygon edge and all of them are stored in a one-way connected list associated with each cell.

Fig. 3 shows an example of a polygon defined by a loop and a ring after the rasterization process has been finished. Cell (15, 7) contains for example three edges: $e_{5,6}$, $e_{6.7}$, and $e_{15,16}$. Before the polygon edges are inserted into the grid, they must be oriented as described in Section 2.

3.1.3. Determination of inner and outer cells

In the last pre-processing step, the cells still having the status *Undefined* are classified as being completely inside or completely outside the polygon. Inner cells should get the status *Black* and the outer ones *White*. For this task, using one of the classical raster-based algorithms for filling polygons seems to be good solution (Foley et al., 1990). However, if we observe Fig. 3 more carefully, we can see that *Grey* cells defining loop and rings can touch or even coincide. Due to this, it is possible that there are no inner cells between the border cells (the cells defining holes and loops). In this way, the classical algorithms for polygon filling would not always return a correct result. Fortunately, the solution is not difficult and it works in the following two steps:

- First, the majority of the outer cells of the polygon's loop are marked as *White*. It is easy to recognize them. We start by moving from the borders of the grid in left, right, up, and down directions until we encounter a *Grey* cell. However, if *White* cells are "hidden" behind *Grey* cells, they cannot be reached. An example is seen in Fig. 3, where cell (4, 3) still has the status *Undefined*. Such cells are classified in the second step.
- The second step applies the well-known flood-fill algorithm (Foley et al., 1990), which operates while *Unclassified* cells exist. However, at first, we have to determine which "colour" (*White* or *Black*) is used for flooding. We apply the following algorithm:
  The search for *Unclassified* cells proceeds from the bottom to the top. When the first *Unclassified* cell is detected, the algorithm determines its colour in the following way:
  - A point in the middle of the *Unclassified* cell is determined. Let us denote it by $\boldsymbol{c}$.
  - From $\boldsymbol{c}$, four rays $\boldsymbol{r}_i$, $0 \leqslant i \leqslant 3$ are sent in four directions; left, right, bottom and top. The rays occupy as many cells as are needed to reach the border of the grid, or the cell with the status

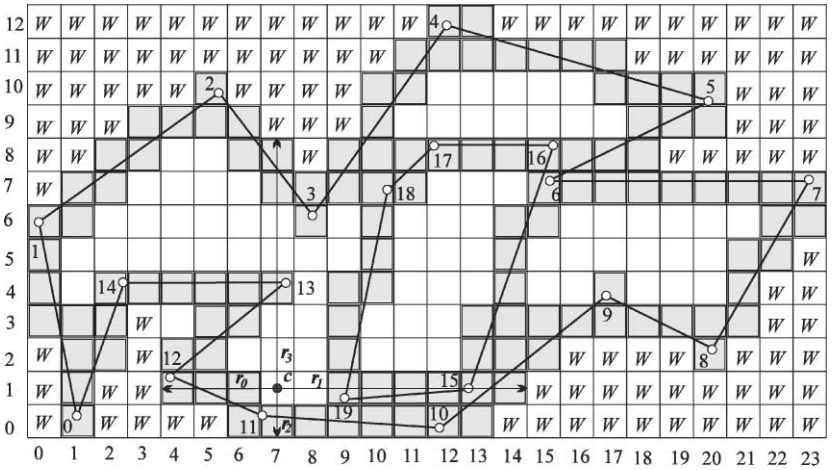

Fig. 4. Determination of colour of seed of flood-fill algorithm.

White or *Black*. The cells crossed by a ray are considered to be the active cells of that ray.

- The ray, which crosses the smallest number of cells, is accepted for the final estimation.
- The number of intersections between the chosen ray and the edges stored in *Grey* cells is counted. If the number of intersections is even, the estimated cell has the same colour as the cell at which the ray stops (if it shows outside the grid, the colour is *White*), otherwise the colour is inverse.
- The flood-fill algorithm now floods all the cells which have the status *Unclassified* starting with the cell whose position (and colour) has just been determined.

Let us illustrate the algorithm using Fig. 4, where the majority of outer cells have already been marked as *White* (*W* in Fig. 4). By visiting cells from bottom to top and from left to right, we find the first detected *Unclassified* cell is at the position $(7,1)$. The centre point $\boldsymbol{c}$ in that cell is calculated. As already described, the rays $\boldsymbol{r}_0, \boldsymbol{r}_1, \boldsymbol{r}_2$, and $\boldsymbol{r}_3$ are calculated. The length of vector $\boldsymbol{r}_2$ is the smallest. It hits the border of the grid through cell $(7,0)$ and therefore, its active length is 1. Now, all edges existing in the cell $(7,0)$ are tested to determine whether or not they intersect ray $\boldsymbol{r}_2$. At the ray–edge intersection, a special case appears when the ray passes exactly through the edge vertex. The approach from the classical ray intersection method is applied (Manber, 1989; Foley et al., 1990). In our case, only edge $e_{10,11}$ is crossed by ray $\boldsymbol{r}_2$. Since the number of intersections is odd, the seed for the flood-fill algorithm is set to *Black*. The flood-fill algorithm is run after that. In the example of Fig. 4, the described procedure is activated 4 times more to mark correctly all cells.

3.2. Testing a point for containment

After the initialization, we are ready to determine the position of an arbitrary point $\boldsymbol{q}$. At first, position of the cell into which point $\boldsymbol{q}$ falls is calculated by

$$Pos_x = \left\lfloor \frac{q_x - x_{\min}}{Size_x} \right\rfloor,$$

$$Pos_y = \left\lfloor \frac{q_y - y_{\min}}{Size_y} \right\rfloor. \tag{5}$$

There are two possibilities regarding the colour of the cell (Pos_x, Pos_y):

- The cell is either *Black* or *White*: the position of the point $\boldsymbol{q}$ is easily determined as being on the inside or outside of the polygon.
- The cell is *Grey*: the position of the point is calculated using the edges stored in that cell. The details are explained next.

Let us suppose, point $\boldsymbol{q}$ falls into a *Grey* cell. To determine the position of the point, the algorithm performs the following actions:

- First, the nearest polygon edge within all the edges being stored in the considered cell is found. Let us denote it as $e_{ij} = (v_i, v_j)$, where v_i and v_j are the vertices determining that edge. They may or may not be located inside cell (Pos_x, Pos_y).
- To determine the position of the point $\boldsymbol{q}$, it is necessary to determine on which side of edge e_{ij} the point is located. Let us remember that the polygon edges were oriented correctly before they were inserted into the uniform grid. Two vectors $\boldsymbol{u}_1 = (\boldsymbol{v}_j - \boldsymbol{v}_i)$ and $\boldsymbol{u}_2 = (\boldsymbol{v}_i - \boldsymbol{q})$ are formed and the cross product $\boldsymbol{u}_1 \times \boldsymbol{u}_2$ is calculated. The sign of the z

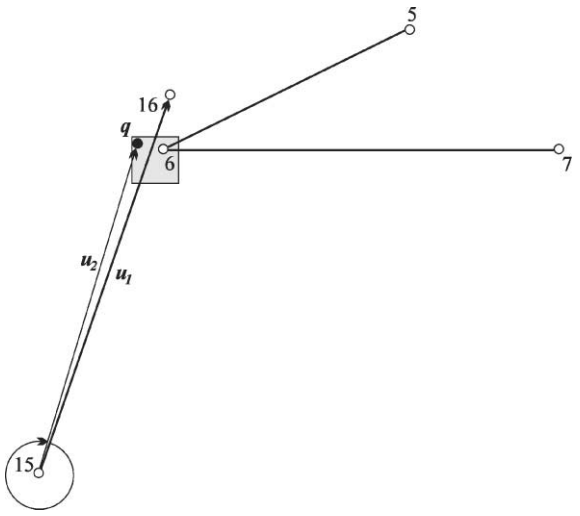

Fig. 5. Local inclusion test.

components determines the position of the point. Three possibilities exist:

- $z = 0$: point $\boldsymbol{q}$ is located on the border of the polygon. Normally, the equality test has to be performed using a small tolerance ε (Hoffman, 1989).
- $z > 0$: point $\boldsymbol{q}$ is inside the polygon,
- $z < 0$: point $\boldsymbol{q}$ is outside the polygon.

In Fig. 5, an example is given. Cell (15, 7) from Fig. 4 is enlarged and the tested point $\boldsymbol{q}$ is marked by the filled circle. In this cell, three edges exist. Edge $e_{15,16}$ belonging to the ring (located at the odd level of the ring hierarchy) is the closest edge. Point $\boldsymbol{q}$ is located inside the ring, and it is therefore outside the polygon. The vector product between vectors $\boldsymbol{u}_1 = (\boldsymbol{v}_{15} - \boldsymbol{v}_{16})$ and $\boldsymbol{u}_2 = (\boldsymbol{v}_{15} - q)$ is negative and the point $\boldsymbol{q}$ is classified as being outside the polygon.

Determination of the closest edge must be carried out carefully because more edges could be at the same distance from the considered point. In fact, only one situation causes a real problem and this appears when the following conditions are fulfilled:

- the edges with the same distances are neighbouring edges,
- the shortest distance is the distance to the common vertex,
- the vertex is concave.

When these conditions are met, the tested point is declared as being inside the polygon.

In Fig. 6, the previously mentioned special case is highlighted. Tested point $\boldsymbol{q}$ is located inside the polygon. It is positioned inside a region shaded in grey in Fig. 6. Within this region, the search for an edge with the minimal distance from $\boldsymbol{q}$ returns two edges: $e_{1,2}$ and $e_{2,3}$ which share the vertex $\boldsymbol{v}_2$. As seen from Fig. 6, this

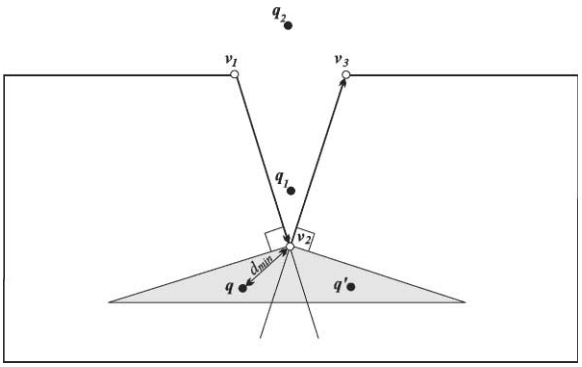

Fig. 6. Solving special case.

vertex is concave. If the cross-product with edge $e_{1,2}$ could be calculated, a positive sign would be obtained (point $\boldsymbol{q}$ is on the right side of edge $e_{1,2}$) indicating that the point is inside the polygon. On the contrary, using edge $e_{2,3}$, a negative sign of the cross-product would be obtained (the point is on the left side of the line defined by the vertices $\boldsymbol{v}_2$ and $\boldsymbol{v}_3$) indicating that the point is outside the polygon. The situation is different for the point $\boldsymbol{q}'$ in Fig. 6. As already mentioned, when the described special case is identified, the tested point is declared as being inside the polygon.

Let us consider two more positions of points in Fig. 6. Point $\boldsymbol{q}_1$ has two edges at the same distance. However, this situation is not a special case, because the shortest distance is not the distance to the common vertex. Indeed, calculating the cross product using any of these edges gives the same (negative) sign. Point $\boldsymbol{q}_2$ has the same distances to four edges—actually, to two common vertices. However, none of these vertices is concave and therefore, we may use any of the edges for calculating the cross product which show that the point is outside the polygon. At that point, it is worth noticing that all the calculations are made within the floating point arithmetic. There are three main weaknesses of the floating-point calculations that may cause undesirable effects and failure of the algorithm: conversion errors, round-off errors and digit-cancellation errors (Hoffman, 1989). Due the nature of digital computers, these errors cannot be avoided but careful implementation can reduce their negative effects. For this reason, an acceptably small tolerance ε is introduced and within this tolerance, two real numbers are considered as equal. It is a question, however, of how small (or large) ε should be. Setting ε for some small value (10^{-6} for example) does not always give good results (the coordinates may be set in nm in microelectronics and up to 10 Mm in GIS applications). It proved to be better if determination of ε is linked with the bounding box of the polygon being considered (Žalik et al., 2000). In this way, we preserve the comparable ratio between ε and the coordinates of the polygon. It is also important where in

the programming code the checking for this tolerance is applied. The most dangerous situation appears when the difference between two, nearly equal numbers, is calculated, and the result is immediately used as a denominator. The resulting number has fewer significant digits and division with such a number can give a result, which differs from the result obtained by "real arithmetic" by more than the introduced tolerance ε. Such situations must be predicted before the difference and division can be applied. In the described algorithm, we avoid such situations. For example, after calculating the cross product we do not apply the division without checking if the z component of the cross product is close enough to 0, as explained in the beginning of this subsection.

The following pseudo code summarizes the whole algorithm in a more comprehensive form:

```
CBCA(Polygon, NoOfVertices, SetOfPoints,
      NoOfPoints)
begin
   Bbox=DetermineBoundingBox(SetOfPoints,
         NoOfPoints)
   Grid = ConstructGrid(Bbox, NoOfVertices)
   MarkCells(Grid, Polygon, NoOfVertices)
   for i = 1 to NoOfPoints do
   begin
      (i,j) = GetPointPosition(Grid, SetOfPoints[i])
      CellColour = WhatColour(Grid, i, j)
      case CellColour of
      WHITE: SetOfPoints[i].position = Outside
      BLACK: SetOfPoints[i].position = Inside
      GREY:
         begin
           SetOfNearestEdges=FindTheNearestEdges
           (Grid, i, j, SetOfPoints[i])
           if NoOfNearest(SetOfNearestEdges) = 1 then
           begin
              Edge = FirstEdgeOf(SetOfNearestEdges)
              SetOfPoints[i].position = ByCrossProduct
              (Edge, SetOfPoints[i])
           end
           else /* there is more edges at the same
           distance */
           begin
              if BoundaryCase(SetOfNearestEdges) then
                 SetOfPoints[i].position = Inside
              else
                Edge = FirstEdgeOf(SetOfNearestEdges)
              SetOfPoints[i].position = ByCrossProduct
                (Edge, SetOfPoints[i])
              endif
           endif
         end
      endcase
   endfor
end
```

4. Time and space complexity estimation

Let us consider a time complexity of the proposed *CBCA* algorithm designed to support repetitive questions about the containment of different tested points. As explained, the algorithm works in two stages: an initialization step and a point-in-polygon test. Consider the time complexity of the initialization stage first. As described, the initialization consists of three steps:

(a) Forming the uniform subdivision: in this step the following tasks are performed:
- If the bounding box is not known, it can be determined in linear time $O(n)$, where n is the number of polygon vertices.
- A grid of equal sized cells is generated using Eqs. (2) and (3). There are $NoOfCells_x.NoOfCells_y$ cells in the grid. Using Eq. (2), we can determine the greatest number of cells m, which is

$$m = 2(1+\sqrt{n})2(1+\sqrt{n}) = 4(1+2\sqrt{n}+n) = O(n). \quad (6)$$

All m cells are then visited and their status is set to value *Undefined*. This operation is finished in linear time $O(n)$.

(b) In the second step, all n edges are inserted in the uniform grid. The cells containing at least one edge are coloured as *Grey* cells. The time needed for the rasterization of an edge depends on the number of cells containing the considered edge. Let us denote by nc the number of grey cells. In general, $nc < n$ and therefore, $O(n)$ could be regarded as an acceptable estimate.

(c) In the last initialization step, the remaining *Undefined* cells are marked as being either *White* or *Black*. There are $k < n$ non-visited cells. The flood-filling algorithm visits each of the k cells exactly once. However, before flooding can be started, the flood colour is determined. As explained in Section 3, at most $NoOfCells_x + NoOfCells_y = 2\sqrt{n}$ additional cells are visited and among them, the shortest sequence is chosen. We can suppose that less than $\sqrt{n}$ edges exist in the checked sequence of cells. Therefore, the time complexity of that step can be estimated as $O(n\sqrt{n})$.

The whole initialization is finished in time

$$O(n) + O(n) + O(n\sqrt{n}) = O(n\sqrt{n}). \quad (7)$$

Testing a point for containment is performed in the following steps:

(a) At first, the cell into which the point falls is determined in constant time using Eq. (5).

(b) Regarding the colour of the cell, there are two possibilities:

- If the status of the cell is not *Grey*, the result is returned immediately in constant time $O(1)$.
- If the point falls into a *Grey* cell, the closest edge must be found, and a vector product must be calculated. Here, in general, there are $\sqrt{n}$ edges in a *Grey* cell, and in this instance, the answer is obtained in $O(\sqrt{n})$ time.

Let us summarize: the expected time complexity of the *CBCA* algorithm consists of an initialization terminated in $O(n\sqrt{n})$ time, and point-in-polygon test finished either in constant time $O(1)$ or in $O(\sqrt{n})$ time.

However, theoretical time complexity does not always indicate clearly, the speed of an algorithm. For this, measurements of CPU time often give better information. We have compared our *CBCA* algorithm with the most popular ray crossing method. Both algorithms have been implemented in C++ and measured on a PC Pentium 133 MHz. Table 1 shows the CPU time spent by using the *CBCA* algorithm while Table 2 shows the CPU time spent by the ray crossing method. The last two polygons have been obtained from a GIS database. As expected, the benefit of the proposed algorithm increases with the number of tested points and the number of polygon vertices. The shape of the polygon naturally also plays an important role. For example, in Table 1 it can be seen that a polygon with 100 vertices has been processed faster than a polygon with only 10 vertices. Comparing the results from Tables 1 and 2 it can be seen that this significant speed-up is achieved when more than 1000 tested points are expected even when the number of polygon vertices is small. However, appropriate selection of the heuristics ensures that the algorithm is not too slow even if just a few points are tested for containment. Fig. 7A shows a diagram of CPU time for a polygon with 10 vertices, which is the average number of vertices in GIS applications. Fig. 7B shows CPU time for a large polygon obtained from a GIS database with 28150 vertices.

Finally, let us estimate the space complexity of the algorithm. For a polygon with n edges we need at most $2\sqrt{n} \times 2\sqrt{n} = 4n = O(n)$ cells where each cell contains on average one edge. For testing a point, constant space is needed, and therefore, the common space complexity of the algorithm is $O(n)$.

5. Conclusions

This paper presents a new algorithm for solving a point-in-polygon problem. The algorithm is optimized for those situations where repetitive questions are expected (such examples are not rare in geographic information systems). The algorithm employs a uniform

Table 1
CPU time in seconds spent using *CBCA* algorithm

No. of polygon points	No. of tested points														
	1			10			1000			10,000			50,000		
	Init.	Inclusion	Total	Init.	Inclusion	Total	Init.	Inclusion	Total	Init.	Inclusion	Total	Init.	Inclusion	Total
3	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	0.01	<0.01	0.20	0.20	<0.01	1.00	1.00
10	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	0.01	<0.01	0.20	0.20	<0.01	1.00	1.00
100	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.01	0.01	<0.01	0.14	0.14	<0.01	0.73	0.73
1249	0.04	<0.01	0.04	0.04	<0.01	0.04	0.04	0.02	0.06	0.04	0.13	0.17	0.04	0.71	0.75
28 105	0.37	<0.01	0.37	0.37	<0.01	0.37	0.37	0.05	0.42	0.37	0.84	1.21	0.37	3.80	4.17

Table 2
CPU time in seconds spent by ray crossing method

No. of polygon points	No. of tested points				
	1	10	1000	10,000	50,000
3	<0.01	<0.01	0.10	1.43	5.56
10	<0.01	<0.01	0.18	2.18	8.50
100	<0.01	<0.01	0.68	7.23	33.50
1249	<0.01	0.02	3.87	32.90	171.00
28 105	0.06	0.80	76.59	758.00	—

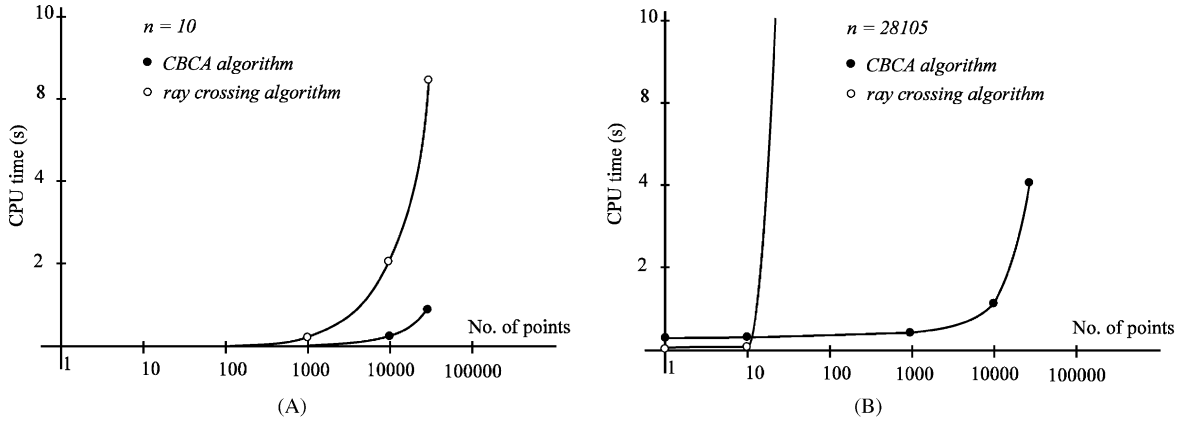

Fig. 7. CPU time spent related to number of tested points: polygon with 10 vertices (A); and polygon with 28 105 vertices (B).

space subdivision and a heuristics approach is proposed for the organization of that subdivision. The algorithm works in two steps: at first, the polygon edges are inserted into the uniform grid, and then, cells which do not contain any edges are classified as being either on the inside or outside of the polygon. The expected time complexity of this step is $O(n\sqrt{n})$. In the second step, arbitrary points are tested regarding containment. If a tested point falls into a cell being classified as inner or outer, the result is returned in constant time $O(1)$. If the point falls in a cell containing at least one polygon edge, the answer is obtained using just the edges existing in that cell. The time complexity analysis shows that the expected time complexity of the second step is $O(\sqrt{n})$. The algorithm works with $O(n)$ space complexity. Only one special boundary case has been identified for the point-in-polygon test using the *CBCA* algorithm. The solution of that boundary case, explained in the paper, does not slow down the algorithm, and it is easy to implement. The *CBCA* algorithm is numerically stable because it eliminates the problems caused by the accumulation of rounding errors. In addition, the situation when the tested point is on the border of the polygon is easily detected (the third component of the cross product is zero which can be detected by introducing a small tolerance). The algorithm requires a little more programming code than the majority of other approaches, but we are rewarded with rapid answers to point-in-polygon questions. The algorithm has been in use for a while within a GIS application and the results are very positive. A demonstration can be downloaded from http://www.uni-mb.is/ ~GeMMA.

Acknowledgements

This work has been carried out as part of a bilateral Czech–Slovene project AlgVIS (project ME 259) and project VS 97155. The authors would like to thank to Ms. J. Rees for language corrections and improvements.

References

Bresenham, J.E., 1965. Algorithm for computer control of a digital plotter. IBM System Journal 4 (1), 25–30.

Brobst, S., Gant, S., Thompson, F., 1999. Partitioning very large database tables with Oracle8. Oracle Magazine 13 (2), 123–126.

Chen, M., Townsend, P., 1987. Efficient and consistent algorithms for determining the containment of points in polygons and polyhedra. In: Marechal, G. (Ed.), Proceedings of Eurographics'87. Elsevier Science, Amsterdam, pp. 423–437.

Cleary, J.C., Wyvill, G., 1988. Analysis of an algorithm for fast ray tracing using uniform space subdivision. The Visual Computer 4 (2), 65–83.

Fang, T.-P., Piegl, L., 1993. Delaunay triangulation using a uniform grid. IEEE Computer Graphics & Applications 13 (3), 36–47.

Feito, F., Torres, J.C., Urena, A., 1995. Orientation, simplicity, and inclusion test for planar polygons. Computers & Graphics 19 (4), 595–600.

Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., 1990. Computer Graphics—Principles and Practice, 2nd edn. Addison-Wesley, Reading, MA, 1174pp.

Fujimoto, A., Tanaka, T, Iwata, K., 1986. ARTS: accelerated ray-tracing system. IEEE Computer & Graphics Applications 6 (1), 65–83.

Hoffman, C.M., 1989. Geometric & Solid Modeling, an Introduction, Morgan Kaufman, San Mateo, 338pp.

Huang, C-.W., Shih, T-.Y., 1997. On the complexity of point-in-polygon algorithms. Computers & Geosciences 23 (1), 109–118.

Kolingerova, I., 1995. Detection of mutual position of a point and a convex polygon in E^2. Machine Graphics & Vision 4 (3), 151–160.

Manber, U., 1989. Introduction to algorithms—a creative approach, Addison-Wesley, Reading, MA, 478pp.

Mortenson, M.E., 1985. Geometric Modeling, Wiley, New York, 763pp.

O'Rourke, J., 1994. Computational Geometry in C. Cambridge University Press, Cambridge, 346pp.

Preparata, F.P., Shamos, M.I., 1985. Computational Geometry: an Introduction, 2nd edn. Springer, New York, 398pp.

Salomon, K.B., 1978. An efficient point-in-polygon algorithm. Computers & Geosciences 4 (2), 173–175.

Samet, H., 1989. The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, MA, 493pp.

Samet, H., 1990. Applications of Spatial Data Structures. Addison-Wesley, Reading, MA, 507pp.

Seidel, R., 1991. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Computational Geometry: Theory and Applications 1 (1), 51–64.

Skala, V., 1994a. $O(\ln N)$ line clipping algorithm in E^2. Computers & Graphics 18 (4), 517–524.

Skala, V., 1994b. Point-in-polygon with $O(1)$ complexity. Technical Report No. 68/94, University of West Bohemia, Pilsen, Czech Republic, 1994.

Skala, V., 1996. Line clipping in E_2 with suboptimal complexity $O(1)$. Computers & Graphics 20 (4), 523–530.

Taylor, G., 1994. Point in polygon test. Survey Review 32, 479–484.

Žalik, B., 1999. A topology construction from line drawings using a uniform plane subdivision technique. Computer-Aided Design 31 (5), 335–348.

Žalik, B., Clapworthy, G.J., 1999. A universal trapezoidation algorithm for planar polygons. Computers & Graphics 23 (3), 353–363.

Žalik, B., Clapworthy, G., Oblonšek, Č., 1997. An efficient code-based voxel-traversing algorithm. Computer Graphics Forum 16 (2), 119–128.

Žalik, B., Podgorelec, D., Starodub, R., 2000. Trapezoidation of concave polygons with nested holes. Technical Report for AutoDESK, GmbH Munchen, 22pp.