ELSEVIER

Computational Geometry 20 (2001) 131–144

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

The point in polygon problem for arbitrary polygons

Kai Hormann [a,*], Alexander Agathos [b]

[a] *University of Erlangen, Computer Graphics Group, Am Weichselgarten 9, 91058 Erlangen, Germany*
[b] *University of Athens, Department of Informatics, Athens, Greece*

Communicated by K. Mehlhorn; received 18 January 2000; received in revised form 9 January 2001; accepted 20 March 2001

Abstract

A detailed discussion of the point in polygon problem for arbitrary polygons is given. Two concepts for solving this problem are known in literature: the *even–odd rule* and the *winding number*, the former leading to *ray-crossing*, the latter to *angle summation* algorithms. First we show by mathematical means that both concepts are very closely related, thereby developing a first version of an algorithm for determining the winding number. Then we examine how to accelerate this algorithm and how to handle special cases. Furthermore we compare these algorithms with those found in literature and discuss the results. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Polygons; Point containment; Winding number; Integer algorithms; Computational geometry

1. Introduction

A very natural problem in the field of computational geometry is the point in polygon test: given a point R and an arbitrary closed polygon P represented as an array of n points $P_0, P_1, \ldots, P_{n-1}, P_n = P_0$, determine whether R is inside or outside the polygon P. While the definition of the interior of standard geometric primitives such as circles and rectangles is clear, the interior of self-intersecting closed polygons is less obvious. In literature [1,4,5,7,8,10,13,14], two main definitions can be found.

The first one is the *even–odd* or *parity rule*, in which a line is drawn from R to some other point S that is guaranteed to lie outside the polygon. If this line $\overline{RS}$ crosses the edges $e_i = \overline{P_i P_{i+1}}$ of the polygon an odd number of times, the point is inside P, otherwise it is outside (see Fig. 1(a)). This rule can easily be turned into an algorithm that loops over the edges of P, decides for each edge whether it crosses the line or not, and counts the crossings. Various implementations of this strategy exist [2–4,6,8,10–12] which differ in the way how to compute the intersection between the line and an edge and how this rather costly procedure can be avoided for edges that can be guaranteed not to cross the line. We discuss these issues in detail in Section 3.

[*] Corresponding author.
E-mail addresses: hormann@informatik.uni-erlangen.de (K. Hormann), agalex@yahoo.com (A. Agathos).

PII: S0925-7721(01)00012-8

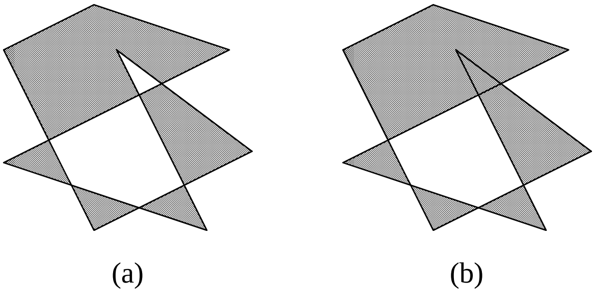

Fig. 1. The interior of a self-intersecting polygon based on (a) the even–odd rule and (b) the nonzero winding number.

The second one is based on the *winding number* of R with respect to P, which is the number of revolutions made around that point while traveling once along P. By definition, R will be inside the polygon, if the winding number is nonzero, as shown in Fig. 1(b). We show that the same result as with the even–odd rule can be obtained by letting the interior consist of those points whose winding number is odd. Therefore, both definitions of the interior can be based on the winding number, making this concept the more general one.

In Section 2 we explain in detail how the *incremental angle* algorithm [13] for determining the winding number can be derived mathematically. Further analysis of this algorithm leads to a modification that turns it into a ray-crossing algorithm, revealing that both concepts are the same in principle. The resulting algorithm is capable of handling any special cases that might occur, e.g., R may coincide with one of the vertices P_i of P or may lie on one of P's edges e_i.

Several methods for accelerating this basic algorithm are discussed in Section 3. Of course the problem always is of complexity $\mathrm{O}(n)$ for arbitrary polygons, hence "acceleration" refers to reducing a constant time factor. The complexity can only be reduced for special polygons, e.g., if the polygon is convex, an $\mathrm{O}(\log n)$ algorithm can be found [7,8,10]. The performance of the different algorithms is analyzed in Section 4 and a comparison to those found in literature is made. Section 5 summarizes the proposed ideas.

2. Winding numbers

As stated in Section 1, the answer to the point in polygon problem can be derived from the winding number. Starting with the mathematical definition of the winding number, we simplify the general formula step by step until we obtain the pseudo-code of a very intelligible algorithm that determines the winding number of a point with respect to an arbitrary polygon.

The winding number $\omega(R, C)$ of a point R with respect to a closed curve $C(t) = (x(t), y(t))^{\mathrm{T}}$, $t \in [a, b]$, $C(a) = C(b)$, is the number of revolutions made around R while traveling once along C, provided that R is not visited in doing so. Whenever there exists $\tilde{t} \in [a, b]$ such that $C(\tilde{t}) = R$, the winding number $\omega(R, C)$ is undefined. Otherwise it can be calculated by integrating the differential of the angle $\varphi(t)$ between the edge $\overline{RC(t)}$ and the positive horizontal axis through R (cf. Fig. 2(a)). As $C(t)$ is a closed curve, this always yields $\omega \cdot 2\pi$ with $\omega \in \mathbb{Z}$ denoting the winding number.

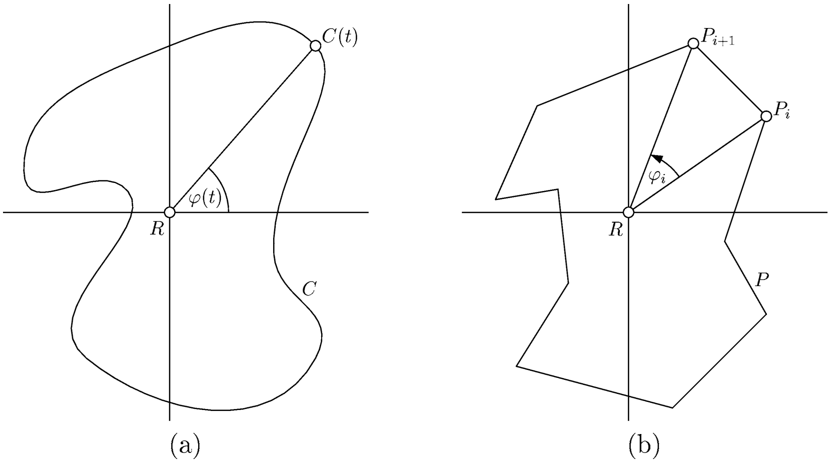

Fig. 2. (a) The continuous angle $\varphi(t)$ for curves. (b) The discrete signed angle φ_i for polygons.

Without loss of generality we assume $R = (0, 0)$ so that $\varphi(t) = \arctan(y(t)/x(t))$ and

$$\omega(R, C) = \frac{1}{2\pi}\int_a^b \mathrm{d}\varphi(t) = \frac{1}{2\pi}\int_a^b \frac{\mathrm{d}\varphi}{\mathrm{d}t}(t)\,\mathrm{d}t = \frac{1}{2\pi}\int_a^b \frac{\dot{y}(t)x(t) - y(t)\dot{x}(t)}{x(t)^2 + y(t)^2}\,\mathrm{d}t. \tag{1}$$

A closed polygon P represented as an array of n points $P_0, P_1, \ldots, P_{n-1}, P_n = P_0$ can be seen as a piecewise linear curve $t \mapsto (x_i(t-i), y_i(t-i))^{\mathrm{T}}$, $t \in [i, i+1]$, with $(x_i(t), y_i(t))^{\mathrm{T}} = tP_{i+1} + (1-t)P_i$. Using Eq. (1) and Appendix A we obtain

$$\begin{aligned}\omega(R, P) &= \frac{1}{2\pi}\sum_{i=0}^{n-1}\int_0^1 \frac{\dot{y}_i(t)x_i(t) - y_i(t)\dot{x}_i(t)}{x_i(t)^2 + y_i(t)^2}\,\mathrm{d}t \\ &= \frac{1}{2\pi}\sum_{i=0}^{n-1}\arccos\frac{\langle P_i | P_{i+1}\rangle}{\|P_i\|\,\|P_{i+1}\|}\cdot \operatorname{sign}\begin{vmatrix} P_i^x & P_{i+1}^x \\ P_i^y & P_{i+1}^y \end{vmatrix} && (2) \\ &= \frac{1}{2\pi}\sum_{i=0}^{n-1}\varphi_i, && (3)\end{aligned}$$

where φ_i is the signed angle between the edges $\overline{RP_i}$ and $\overline{RP_{i+1}}$ (cf. Fig. 2(b)).

Eq. (2) can be used for creating an algorithm for computing the winding number but it involves expensive calls to the *arccos* and *sqrt* routines. Although these can be accelerated by using lookup-tables and nearest-neighbor interpolation, as we can eliminate rounding errors by rounding the final result to the nearest integer value, this still remains a comparatively slow algorithm.

Further simplification of Eq. (3) can be achieved by considering the rounded partial sums $\widehat{s}_j = \frac{1}{4}\lfloor\sum_{i=0}^{j}\frac{\varphi_i}{\pi/2}\rfloor$ with $\omega(R, P) = \widehat{s}_{n-1}$. This is equivalent to counting only quarter-revolutions and can be realized as follows. Based on an algorithm that is explained on p. 251 of Rogers' book [9] for testing

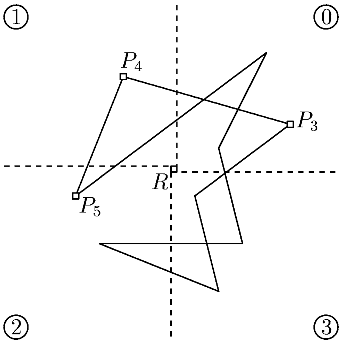

Fig. 3. Classification of vertices P_i by quadrants, e.g., $q_3 = 0$, $q_4 = 1$ and $q_5 = 2$.

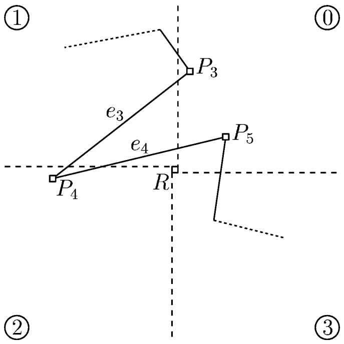

Fig. 4. Example of half ccw- or cw-revolution: edge e_3 with $\delta_3 = 2$ is ccw, e_4 with $\delta_4 = -2$ is cw.

whether a polygon surrounds a rectangular window or is disjoint to it, we classify each vertex P_i of the polygon P by the number q_i of the quadrant in which it is located with respect to R (cf. Fig. 3), i.e.,

$$q_i = \left\{ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} \right\}, \quad \text{if } \arctan \frac{P_i^y}{P_i^x} \in \left\{ \begin{matrix} [0,\ \pi/2) \\ [\pi/2,\ \pi) \\ [\pi,\ 3\pi/2) \\ [3\pi/2,\ 2\pi) \end{matrix} \right\}, \quad \text{resp. } P_i^x \left\{ \begin{matrix} > \\ \leqslant \\ < \\ \geqslant \end{matrix} \right\} R^x,\ P_i^y \left\{ \begin{matrix} \geqslant \\ > \\ \leqslant \\ < \end{matrix} \right\} R^y.$$

Now we can define the *quarter* angle[1] $\delta_i = q_{i+1} - q_i$, $i = 0, \ldots, n-1$, for each of the polygon's edges e_i. If $\delta_i = 0$, then the corresponding edge is located wholly in one quadrant and nothing happens. If $\delta_i \in \{1, -3\}$, the edge crosses one of the quadrant boundaries in counter-clockwise (ccw) direction and a quarter ccw-revolution around R is made, while the reverse holds for $\delta_i \in \{-1, 3\}$. If $\delta_i \in \{2, -2\}$, a further check is required to decide whether a half ccw- or cw-revolution around R occurs by moving along the corresponding edge e_i (cf. Fig. 4). This can be done by checking the orientation of the triangle $\triangle(R, P_i, P_{i+1})$, i.e., by finding the sign of the determinant,

$$\operatorname{sign} \begin{vmatrix} P_i^x - R^x & P_{i+1}^x - R^x \\ P_i^y - R^y & P_{i+1}^y - R^y \end{vmatrix} = \left\{ \begin{matrix} + \\ - \end{matrix} \right\} \quad \Longleftrightarrow \quad \left\{ \begin{matrix} \text{ccw} \\ \text{cw} \end{matrix} \right\}.$$

By further introducing the *adjusted* quarter angles $\widehat{\delta}_i$ via the table

δ_i	$\widehat{\delta}_i$
0	0
1, −3	1
−1, 3	−1
2 ccw, −2 ccw	2
2 cw, −2 cw	−2

[1] Because of $0 \leqslant q_i \leqslant 3$ we know that $-3 \leqslant \delta_i \leqslant 3$.

```
⋆ evaluation of the determinant ⋆
function det($i$)
  return $(P_i^x - R^x) * (P_{i+1}^y - R^y) - (P_{i+1}^x - R^x) * (P_i^y - R^y)$
⋆ quadrant classification ⋆
for $i = 0$ to $n - 1$
  if $P_i^x > R^x$ and $P_i^y \geqslant R^y$:  $q_i = 0$
  if $P_i^x \leqslant R^x$ and $P_i^y > R^y$:  $q_i = 1$
  if $P_i^x < R^x$ and $P_i^y \leqslant R^y$:  $q_i = 2$
  if $P_i^x \geqslant R^x$ and $P_i^y < R^y$:  $q_i = 3$
$q_n = q_0$
⋆ determination of winding number ⋆
$\omega = 0$
for $i = 0$ to $n - 1$
  switch $q_{i+1} - q_i$:
    $1, -3$:  $\omega = \omega + 1$
    $-1, 3$:  $\omega = \omega - 1$
    $2, -2$:  $\omega = \omega + 2 *$ sign of det($i$)
return $\omega/4$
```

Algorithm 1. First version of a winding number algorithm.

we can sum up these $\widehat{\delta}_i$ to count the number of quarter ccw-revolutions around R and get

$$\widehat{s}_j = \frac{1}{4}\sum_{i=0}^{j}\widehat{\delta}_i \quad \Longrightarrow \quad \omega(R, P) = \widehat{s}_{n-1} = \frac{1}{4}\sum_{i=0}^{n-1}\widehat{\delta}_i.$$

This leads to the first version of a winding number algorithm (Algorithm 1) which resembles the incremental angle algorithm in [13]. It can further be improved by exploiting the following observation:

$$\sum_{i=0}^{n-1}\delta_i = \sum_{i=0}^{n-1}(q_{i+1} - q_i) = q_n - q_0 = 0 \quad \Longrightarrow \quad \omega(R, P) = \frac{1}{4}\sum_{i=0}^{n-1}\bigl(\widehat{\delta}_i - \delta_i\bigr),$$

i.e., we just need to sum up the differences $\widehat{\delta}_i - \delta_i$, which are nonzero only for $\delta_i \in \{-3, -2, 2, 3\}$.[2] Thus, by defining

$$\bar{\delta}_i = \begin{cases} 1, & \text{if } \delta_i \in \{-3, -2 \text{ ccw}\}, \\ -1, & \text{if } \delta_i \in \{3, 2 \text{ cw}\}, \\ 0, & \text{else}, \end{cases}$$

we get

$$\omega(R, P) = \sum_{i=0}^{n-1}\bar{\delta}_i,$$

[2] And we have $\widehat{\delta}_i - \delta_i = \pm 4$ in these cases.

```
                    ⋮
$\omega = 0$
for $i = 0$ to $n - 1$
  switch $q_{i+1} - q_i$:
    $-3$:  $\omega = \omega + 1$
     $3$:  $\omega = \omega - 1$
    $-2$:  if det($i$) $> 0$:  $\omega = \omega + 1$
     $2$:  if det($i$) $< 0$:  $\omega = \omega - 1$
return $\omega$
```

Algorithm 2. Modification of Algorithm 1.

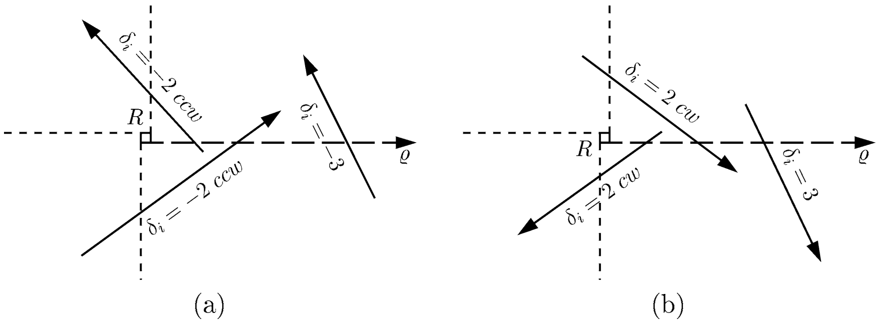

Fig. 5. All edges (arrows indicating direction from P_i to P_{i+1}) crossing from below the ray to above are counted $+1$ (a), the others -1 (b).

which results in a slight modification of the first algorithm (Algorithm 2) and improves the performance by approximately 5%.

However, besides this small acceleration it is far more important to notice that the algorithm has now turned into a ray-crossing method. In fact, by disregarding all edges with $\delta_i \in \{-1, 1\}$, the remaining cases relate to edges that cross the horizontal ray[3] $\varrho = \{R + \lambda\binom{1}{0},\ \lambda \geqslant 0\}$. The difference to the even–odd rule where only the number of crossings is counted is that edges starting below this ray and ending above it are counted $+1$ and the others -1 (cf. Fig. 5). Nevertheless, the winding number ω still shifts from even to odd and back with every crossing, so that testing the parity of ω exactly gives the even–odd definition.

Therefore, we have presented an algorithm that is capable of solving the point in polygon problem for both definitions of the interior of an arbitrary polygon: the one based on the even–odd rule and the other based on the nonzero winding number. In contrast to the statement of O'Rourke, that the determination of the winding number depends on "floating-point computations, and trigonometric computations in particular" [8], this algorithm gets by with integer arithmetic except for the `det` function which needs floating-point operations if the coordinates are non-integer. However, divisions (neither integer nor floating-point) are totally avoided.

[3] This refers to the special choice of $S = \binom{+\infty}{R^y}$ in the definition of the even–odd rule.

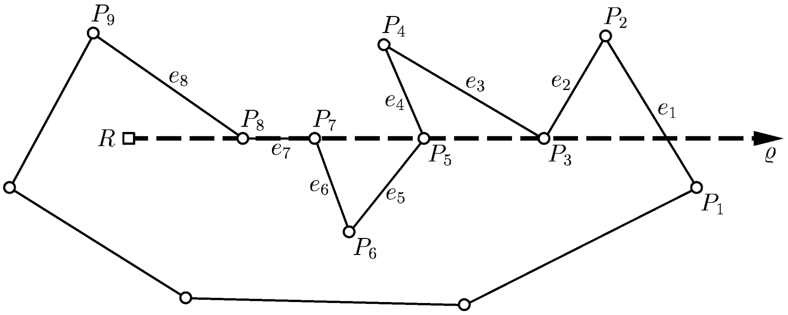

Fig. 6. Degenerate intersections of ϱ and P.

Another advantage of this algorithm is the handling of degenerate cases, which can cause trouble in other algorithms, as, e.g., Foley et al. point out that “the ray must hit no vertices of the polyline” [1]. However, the quadrant-classification of the vertices P_i naturally avoids these degeneracies. In Fig. 6, the regular polygon segment P_1, P_2 as well as the degenerate segment sequence P_6, P_7, P_8, P_9 should count $+1$, while the sequence P_4, P_5, P_6 should count -1 and P_2, P_3, P_4 should not be regarded as a crossing at all, thus resulting in $\omega(R, P) = 1$.

The classification scheme guarantees that no vertex can ever coincide with the ray ϱ, because either $P_i^y \geqslant R^y$, then P_i is above the ray, or $P_i^y < R^y$ which holds for all vertices lying below ϱ. Therefore, the edges e_2, e_3, e_4 and e_7 are ignored as all the vertices adjacent to these edges are classified as 0-quadrant-vertices. On the other hand, edges e_1 and e_6 will be recognized as positive crossings ($\bar{\delta}_1 = \bar{\delta}_6 = 1$) and e_5 as a negative one ($\bar{\delta}_5 = -1$). All other edges, including e_8 do not affect the determination of $\omega(R, P)$.

Another important feature of the algorithm is that it can easily be modified to recognize the special case of R lying on the boundary of P, which may lead to ambiguities in some other algorithms. We distinguish two different cases: firstly, R may coincide with one of the vertices P_i of P, which can be detected by inserting the line

```
if $P_i^x = R^x$ and $P_i^y = R^y$:  exit $\mathit{vertex\_code}$
```

into the quadrant classification loop. Secondly, R may lie on one of P’s edges e_i. In this case, the angle between $\overline{RP_i}$ and $\overline{RP_{i+1}}$ is always $\pm\pi$, so that the classification scheme assigns two diagonally opposite quadrants (0 and 2, or 1 and 3) to the vertices P_i and P_{i+1}, hence $\delta_i = q_{i+1} - q_i = \pm 2$. This always invokes the det function, which returns 0 in this case. Thus, by replacing this function with

```
function det($i$)
  $d = (P_i^x - R^x) * (P_{i+1}^y - R^y) - (P_{i+1}^x - R^x) * (P_i^y - R^y)$
  if $d = 0$
    exit $\mathit{edge\_code}$
  else
    return $d$
```

the algorithm is able to detect this case, too. In the remainder we will refer to this modification as the *boundary version*.

```
function classify($i$)
  if $P_i^y > R^y$
    return $(P_i^x \leqslant R^x)$
  else
    if $P_i^y < R^y$
      return $2 + (P_i^x \geqslant R^x)$
    else
      if $P_i^x > R^x$
        return 0
      else
        if $P_i^x < R^x$
          return 2
        else
          exit vertex_code
```

Algorithm 3. Efficient quadrant classification.

```
$\omega = 0$
for $i = 0$ to $n - 1$
  ⋆ horizontal line crossed? ⋆
  if $(P_i^y < R^y$ and $P_{i+1}^y \geqslant R^y)$ or
     $(P_i^y \geqslant R^y$ and $P_{i+1}^y < R^y)$
    ⋆ crossing to the right? ⋆
    if (det $(i) > 0$ and $P_{i+1}^y > P_i^y)$ or
       (det $(i) < 0$ and $P_{i+1}^y < P_i^y)$
      ⋆ modify winding number ⋆
      if $P_{i+1}^y > P_i^y$
        $\omega = \omega + 1$
      else
        $\omega = \omega - 1$
return $\omega$
```

Algorithm 4. Computing the winding number without quadrant classifications.

3. Efficient implementation

For many applications the algorithms of the previous section are sufficient. They are robust, correct and easy to understand which always helps to reduce the probability of an implementation bug. But in other applications this routine might be called so often that it turns out to be a bottleneck. We now discuss how the basic algorithms can be accelerated, ending up with two very efficient versions: the *efficient standard* algorithm, that is very short but does not care about the special case of R lying on the boundary of P and the *efficient boundary* algorithm, that needs a little more code but handles that special case.

Looking at Algorithms 1 and 2, there are three parts that can be improved: the `det` function, the quadrant classification and the determination of the winding number. The `det` function can be declared as `inline` which saves a few clock cycles for the function call but there is no way to accelerate the actual calculation of this determinant. Likewise one can try to break up the `switch` structure in the third part into a series of sophisticated `if/else` statements but this does not really accelerate the algorithm considerably.

However, the quadrant classification can be improved a lot. In the present version, the average number of comparisons that have to be evaluated for each vertex is 6, assuming the compiler generates *short circuit evaluation*[4]. By simply adding an `else` statement to the end of each line this number can be reduced to 4. A more sophisticated decision tree which only needs slightly more than 2.5 comparisons per vertex and is also able to detect the case of vertex coincidence is shown in Algorithm 3. Note that we have followed the C convention that logical expressions are equal to 1 if they are true and 0 otherwise in order to reduce the length of the code. The use of this classification variant accelerates the basic algorithms of the previous section by more than 30% (cf. Fig. 11).

Unfortunately this is the maximum speed-up we can get out of our basic idea and we need to restructure the algorithm for further improvement. First of all we can combine the two loops because both of them

[4] I.e., the second operand of an `and` operator is only evaluated if the first one is true.

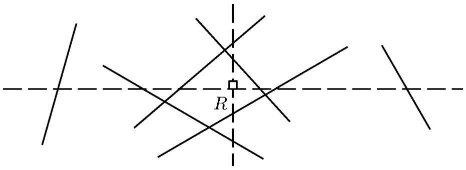

Fig. 7. Edges that fulfill the `crossing` condition.

range over the same interval $i = 0, \ldots, n-1$. Then we can eliminate the array q as we can process the quadrant numbers on the fly and do not need an explicit storage of these values. All we need for each single pass of the loop are the quadrant numbers of the vertex where the edge e_i that is currently processed begins and the one where it ends, namely q_b and q_e. This leads to the following simplification:

```
$q_b =$ classify(0)
for $i = 0$ to $n-1$
  $q_e =$ classify($i+1$)
  switch $q_e - q_b$:
            ⋮
  $q_b = q_e$
```

and reduces memory usage as well as execution time.

Further optimization can be achieved by omitting the quadrant numbers altogether and rather handling the cases in which the winding number changes directly. All edges e_i that relate to these cases have in common that one of their endpoints lies strictly below the horizontal line through R and the other one above or on it (cf. Fig. 5). After using this property as an initial test to distinguish edges that might contribute to a change in the winding number from those who certainly do not, the edge constellations shown in Fig. 7 remain. To further reject those edges that do not modify the winding number, the following observation can be used. Whenever the determinant does not have the same sign as the difference $P_{i+1}^y - P_i^y$, the intersection of the edge with the horizontal line is on the left side of R and the winding number remains unchanged. For the residual edges, the edge direction decides the sign of the modification: if the edge crosses the horizontal line from below ($P_{i+1}^y > P_i^y$), then the winding number is increased, otherwise it is decreased. These considerations are summarized in Algorithm 4, which is 20% faster than Algorithm 2 with the optimal classification scheme (Algorithm 3). This code can be abridged a lot by using the following macros which may seem a little cryptic at first sight:

$$\begin{aligned} \texttt{crossing}\colon\ & (P_i^y < R^y) \neq (P_{i+1}^y < R^y), \\ \texttt{right\_crossing}\colon\ & (\texttt{det}\,(i) > 0) = (P_{i+1}^y > P_i^y), \\ \texttt{modify\_}\omega\colon\ & \omega = \omega + 2 * (P_{i+1}^y > P_i^y) - 1. \end{aligned}$$

They can be used to rewrite Algorithm 4 as Algorithm 5 and accelerate it by more than 30%.

The last improvement on the winding number algorithm can be made by avoiding the rather costly procedure of computing the determinant whenever possible. In Algorithm 5 the determinant computation is invoked for all edges that pass the `crossing` test in order to find out whether they intersect the horizontal line to the left or to the right of R. But for some of these edges this decision can be made in a

```
$\omega = 0$
for $i = 0$ to $n - 1$
  if crossing
    if right_crossing
      modify_$\omega$
return $\omega$
```

Algorithm 5. Using macros to rewrite Algorithm 4.

```
$\omega = 0$
for $i = 0$ to $n - 1$
  if crossing
    if $P_i^x \geqslant R^x$
      if $P_{i+1}^x > R^x$
        modify_$\omega$
      else
        if right_crossing
          modify_$\omega$
    else
      if $P_{i+1}^x > R^x$
        if right_crossing
          modify_$\omega$
return $\omega$
```

Algorithm 6. Efficient standard algorithm.

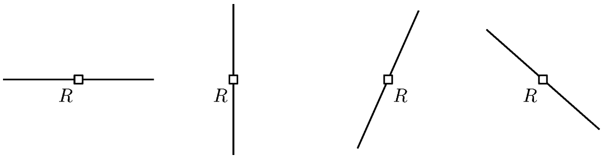

Fig. 8. Different cases of edge coincidences.

simpler way. Referring to Fig. 7, edges like the leftmost one ($P_i^x < R^x$ and $P_{i+1}^x \leqslant R^x$) never change the winding number, whereas those similar to the rightmost one ($P_i^x \geqslant R^x$ and $P_{i+1}^x > R^x$) always do. Both cases can be detected by comparisons and only the edges in the middle require the evaluation of the det function to decide whether they affect the winding number or not. This observation has been realized in the efficient standard algorithm (Algorithm 6). Assuming uniformly distributed polygon vertices, the probability of occurrence of the different edge types is 25% for the leftmost, 25% for the rightmost case and 50% for the edges shown in the middle. Therefore Algorithm 5 evaluates the det function only every second time on average. This improvement is traded in for two extra comparisons and decreases the computational costs by approximately 5%.

Now we will show that only minor modifications of the efficient standard algorithm are necessary in order to handle special cases. First of all, most cases of edge coincidences can be detected by using the modified det function in the right_crossing condition. Only the case of R lying on a horizontal edge (leftmost case in Fig. 8) cannot be recognized this way because this constellation does not pass the crossing condition and can therefore never reach the call of the det function. The other special case, R being identical to one of the vertices P_i, can be detected by checking the equality of both coordinates as in the boundary version of Algorithm 1. The resulting code is shown in Algorithm 7. Note that the vertex coincidence test is prior to the investigation of possible ray intersections, because an edge intersection could be wrongly detected otherwise.

```
if $P_0^y = R^y$ and $P_0^x = R^x$
  exit vertex_code
$\omega = 0$
for $i = 0$ to $n - 1$
  if $P_{i+1}^y = R^y$
    if $P_{i+1}^x = R^x$
      exit vertex_code
    else
      if $P_i^y = R^y$ and $(P_{i+1}^x > R^x) = (P_i^x < R^x)$
        exit edge_code
  if crossing
    if $P_i^x \geqslant R^x$
      if $P_{i+1}^x > R^x$
        modify_$\omega$
      else
        if right_crossing
          modify_$\omega$
    else
      if $P_{i+1}^x > R^x$
        if right_crossing
          modify_$\omega$
return $\omega$
```

Algorithm 7. Efficient boundary algorithm.

4. Evaluation

The timings reported in this section refer to an implementation in C on a 195 MHz SGI R10000 with 128 MB of memory, but similar results were obtained by using *Pascal* and a Pentium PC with 233 MHz and 64 MB of memory. We generated 1000 polygons for different values of n and determined the winding numbers of 1000 reference points for each of these polygons, thus calling the winding number algorithm one million times. The vertices of the polygons as well as the reference points were chosen randomly within the integer square $[-100, 100] \times [-100, 100]$.

Fig. 9 shows that the runtime of the standard algorithms that do not take the special cases into account grows linear with the number of vertices, thus confirming the $O(n)$ complexity of the problem. In contrast, the boundary algorithms behave different (see Fig. 10). As the number of vertices grows, the probability of the reference point to lie on the boundary of the polygon increases. At the same time the chances of the boundary algorithm to exit earlier with the detection of a vertex or an edge coincidence rise and therefore the algorithms do not need to run through the whole loop over the polygon's edges in many cases. Of course, this effect is much less perceivable if the reference points are chosen from a larger domain than the polygon vertices or if the coordinates are floating-point values. Fig. 11 summarizes the timing results of all algorithms for the special choice of $n = 10$.

All algorithms presented in this paper can easily be modified to give the result of the even–odd rule instead of the winding number by replacing every statement that modifies ω, especially the macro `modify_`ω, with $\omega = 1 - \omega$. This simplification saves about 5% of the computation time.

Furthermore we would like to mention that the algorithms can be speeded up considerably by comparing the reference point with the polygon's bounding box first, as it is done, e.g., in the point in

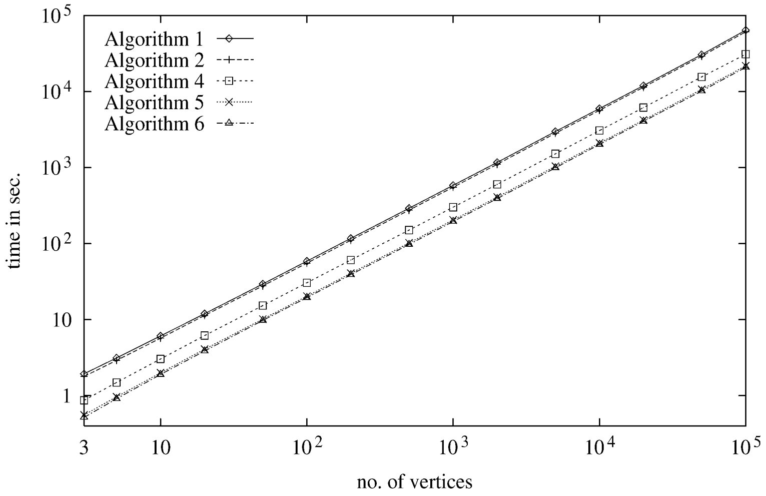

Fig. 9. Execution times of the algorithms that do not handle the special cases.

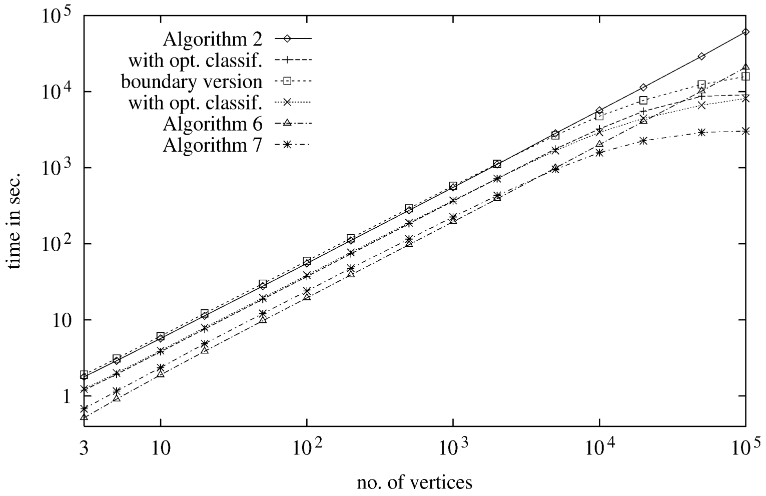

Fig. 10. Execution times of the algorithms that handle the special cases.

polygon algorithm of the C++ library LEDA [6]. The additional costs of the bounding box determination and the test itself pay off after just a few (<10) tests, the precise number depending on the number of vertices as well as the size of the reference point domain compared to the size of the polygon.

We conclude this section by comparing our algorithms to those found in literature. The most thorough comparison of different point in polygon strategies was probably made by Haines in [4], with the result that the ray-crossing strategy performs best “if no preprocessing nor extra storage is available”. Taking a close look at his ray-crossing algorithm it is very similar to Algorithm 6, except that he uses a different

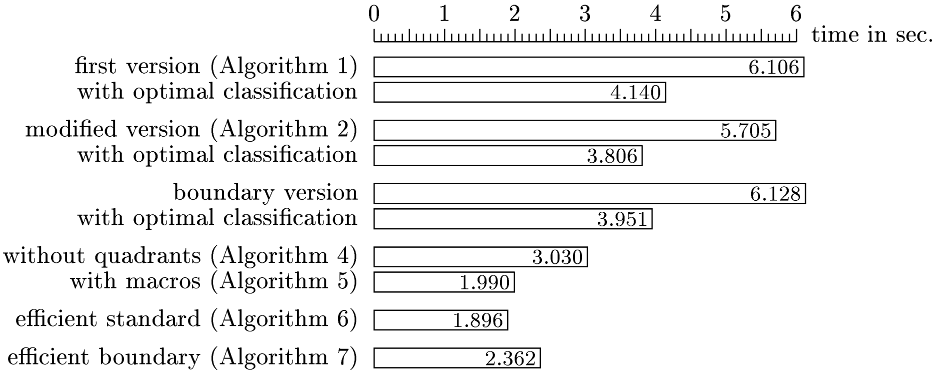

Fig. 11. Execution times of all algorithms for $n = 10$.

test for determining whether an edge crosses the ray to the right. He directly computes the x coordinate of the intersection and compares it to R^x,

$$\texttt{right\_crossing'}: P^x_{i+1} - (P^y_{i+1} - R^y) * (P^x_{i+1} - P^x_i)/(P^y_{i+1} - P^y_i) > R^x.$$

Note that the case of $P^y_{i+1} = P^y_i$, which would cause a division by zero, never passes the prior `crossing` test. We found this version to be approximately 8% slower than the `right_crossing` condition in our testing environment, which is probably due to the division operation. This observation corresponds with Haines' comments on a modified version of his algorithm [3] where he uses the `right_crossing` condition. At the same time he omits the if-statements that filter unnecessary evaluations of this condition, making that modified version identical to Algorithm 5.

The `right_crossing'` condition has also been used in an implementation by Franklin [2] which is otherwise identical to Algorithm 5, as are the algorithm in the LEDA library [6] and the implementation by Stein [11] except that they swap P_i and P_{i+1} if necessary so that they can always assume $P^y_{i+1} < P^y_i$ which simplifies the `right_crossing` condition to ($\texttt{det}\,(i) < 0$) but is about 20% slower in total. Finally, the code given by O'Rourke [8] resembles Algorithm 4 with a `right_crossing'` condition and he gives further optimization ideas as exercises which will eventually lead to Algorithm 5.

5. Conclusion

We have presented a detailed discussion of the point in polygon problem for arbitrary polygons. This problem is well known and has been discussed in many books and papers before. Most of the authors distinguish between two concepts for solving this problem: the even–odd or parity rule and the nonzero winding number. We have shown by mathematical means that both concepts are the same in principle and that the concept of winding numbers encompasses the even–odd idea.

Furthermore, we have developed an algorithm for the determination of the winding number and have improved it step by step up to a very efficient implementation. We have compared our algorithms to those found in literature and can summarize that in our testing environment Algorithm 6 performed best although we admit that Algorithm 5 and the implementations in [2–4] were so close that they might be faster for different machine architectures and compilers. However, a definite advantage of our approach is that it can easily be extended to handle the special case of R lying on the boundary of P (Algorithm 7), an issue that was otherwise taken care of only in [6] and [8], leading to much slower algorithms.

Appendix A

Let $R=(0,0)^{\mathrm{T}}$, $P=(P_x,P_y)^{\mathrm{T}}$ and $Q=(Q_x,Q_y)^{\mathrm{T}}$ be the vertices of a planar triangle and α,β,γ the angles of that triangle at R, P and Q, respectively. Then

$$\cos\alpha=\frac{\langle P|Q\rangle}{\|P\|\,\|Q\|},\qquad \cot\beta=\frac{\langle P-Q|P\rangle}{|D|},\qquad \cot\gamma=\frac{\langle Q-P|Q\rangle}{|D|},$$

with $D=P_xQ_y-Q_xP_y$ and for the linear curve $(x(t),y(t))^{\mathrm{T}}=tQ+(1-t)P$, $t\in[0,1]$, the following equations hold:

$$\begin{aligned}
\int_0^1\frac{\dot y(t)x(t)-y(t)\dot x(t)}{x(t)^2+y(t)^2}\,\mathrm{d}t &= \int_0^1\frac{D}{t^2\langle Q-P|Q-P\rangle+2t\langle Q-P|P\rangle+\langle P|P\rangle}\,\mathrm{d}t\\
&= \arctan\frac{\langle Q-P|Q\rangle}{D}+\arctan\frac{\langle P-Q|P\rangle}{D}\\
&= \operatorname{sign}(D)(\arctan\cot\gamma+\arctan\cot\beta)\\
&= \operatorname{sign}(D)(\pi-\gamma-\beta)\\
&= \operatorname{sign}(D)\alpha.
\end{aligned}$$

References

[1] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics: Principles and Practice, 2nd Edition, Addison-Wesley, 1990.

[2] R. Franklin, pnpoly, http://www.ecse.rpi.edu/Homepages/wrf/geom/pnpoly.html.

[3] E. Haines, CrossingsMultiplyTest, http://www.acm.org/tog/GraphicsGems/gemsiv/ptpoly_haines/ptinpoly.c.

[4] E. Haines, Point in polygon strategies, in: P. Heckbert (Ed.), Graphic Gems IV, Academic Press, Boston, MA, 1994, pp. 24–46.

[5] S. Harrington, Computer Graphics: A Programming Approach, McGraw-Hill, 1983.

[6] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[7] J. Nievergelt, K. Hinrichs, Algorithms and Data Structures: With Applications to Graphics and Geometry, Prentice-Hall, 1993.

[8] J. O’Rourke, Computational Geometry in C, 2nd Edition, Cambridge University Press, 1998.

[9] D.F. Rogers, Procedural Elements for Computer Graphics, McGraw-Hill, 1985.

[10] R. Sedgewick, Algorithms, 2nd Edition, Addison-Wesley, 1988.

[11] B. Stein, A point about polygons, Linux Journal 35 (March 1997).

[12] T. Theoharis, A. Böhm, Computer Graphics: Principles & Algorithms, Symmetria, 1999 (in Greek).

[13] K. Weiler, An incremental angle point in polygon test, in: P. Heckbert (Ed.), Graphic Gems IV, Academic Press, Boston, MA, 1994, pp. 16–23.

[14] M. Woo, J. Neider, T. Davis, OpenGL Programming Guide, 2nd Edition, Addison-Wesley, 1997.