How Reliable are Practical Point-in-Polygon Strategies?[⋆]

Stefan Schirra

Otto von Guericke University, Department of Computer Science, Magdeburg, Germany, `stschirr at ovgu.de`

Abstract. We experimentally study the reliability of geometric software for point location in simple polygons. As expected, the code we tested works very well for random query points. However, our experiments reveal that the tested code often fails for more challenging degenerate and also nearly degenerate queries.

1 Introduction

Assume you would like to test points for inclusion in a simple polygon. Most likely, you will end up using one of the so-called practical point-in-polygon strategies instead of implementing one of the more sophisticated theoretically optimal point location data structures developed in computational geometry. Code for such practical point-in-polygon strategies is available on the WWW. Or you might decide to use components from CGAL [3], LEDA [10] or some other software library providing code for point-in-polygon testing or more general point location queries. Most of the available floating-point based code is very efficient and works well for query points chosen uniformly at random inside the bounding box of the polygon.

As we shall see in Section 4 most of the existent code produces wrong results for query points near or on the polygon edges, however, see also Fig. 1 where queries answered correctly are marked by a grey box ■, false positives by a red disk ●, and false negatives by a green disk ●. If you know that the coordinates of query points and polygon vertices are inaccurate anyway, you might be willing to accept this. Unfortunately, sometimes there are errors not only for such problem-specific degenerate queries, but also for algorithm-specific degeneracies, cf. Fig. 7 in more or less rare cases. Are you willing to accept this for your applications? And what if your data is not subject to uncertainty at all? This is the case we are most interested in. In this paper, we consider simple closed polygons and the corresponding binary point-inclusion predicate only. This is the most important case and can be used for point location in polygons with holes, too. Furthermore, point-in-polygon testing is a subtask in landmarks algorithms for point location in arrangements of straight lines [8].

After a very brief look at related work in the next section, we report on experimental studies regarding the reliability of practical point-in-polygon testing

⋆ Partially supported by DFG grant SCHI 858/1-1

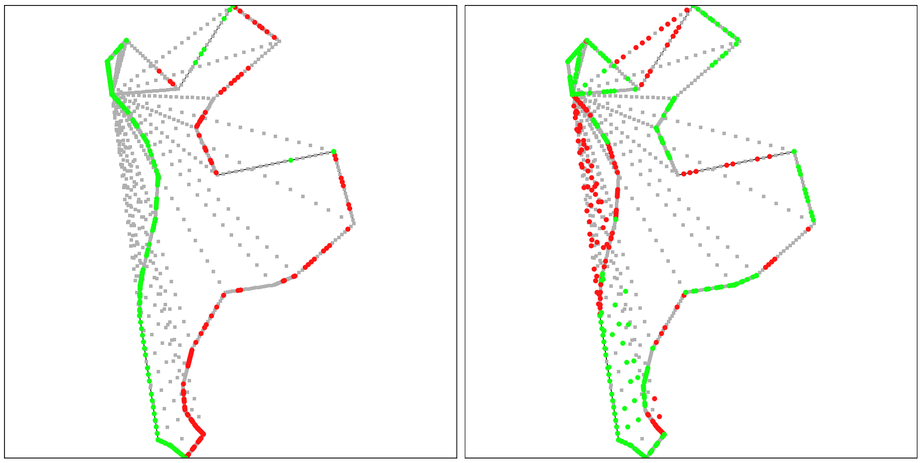

Fig. 1. Results for query points near or on the edges and the diagonals of a real-world polygon with 30 edges for strategies *crossings* (left) and *halfplane* (right).

software. The studies include code from [7], code available on the WWW, and code provided by computational geometry software libraries. Furthermore, we briefly discuss how to achieve full reliability without paying too much for this benefit in Section 5 .

2 Related Work

Testing a query point for inclusion in a polygon is a fundamental problem in computational geometry with many applications, e.g. in computer graphics and geographic information systems, and has been the subject of many research papers in computer science and related application disciplines. For an overview we refer to Snoeyink's survey paper [15]. Probably the most common algorithm for point-in-polygon testing without preprocessing is the crossing number algorithm. Interestingly, already the first description of the algorithm by Shimrat [14] contained a flaw fixed later by Hacker [6]. It is well known that handling degenerate cases in a crossing number algorithm is not obvious. Forrest [4] nicely illustrates the problems involved.

While previous case studies on practical point-in-polygon testing, e.g. [7,16], focus on query time and sometimes on memory usage, here we are concerned with the correctness or at least the numerical stability of practical point-in-polygon algorithms.

3 Experimental Setup

We concentrate on practical point-in-polygon algorithms with no or little preprocessing without sophisticated data structures. Our selection of existent code includes the fastest algorithms from the beautiful graphic gems collection of Haines [7], namely *crossings*, a "macmartinized" crossing number algorithm, see also [1], the triangle-fan algorithms *halfplane*, *barycentric*, and *spackman*, and finally *grid*. Crossing number algorithms compute the parity of the number of intersections of an (horizontal) ray with the polygon boundary. Triangle-fan algorithms consider the collection of triangles formed by an initial vertex v and each polygon edge that is not incident to v. Then they check how many triangles contain the query point. Again, the parity tells us about the relative position of the query point with respect to the polygon. *Halfplane* precomputes line equations for faster triangle testing, *barycentric* and *spackman* compute barycentric coordinates in addition to point location. This might be useful for some applications. *Grid* imposes a 20×20 grid on the bounding box of the polygon and uses a crossing number like strategy to resolve those cases where the grid cell containing the query point is not completely contained in the interior or exterior of the polygon. For the sake of completeness, we also tested Weiler's code [17] which computes the winding number using quadrant movements. Furthermore, we consider Franklin's PNPOLY code [5], which is another crossing number based algorithm available on the WWW. Walker and Snoeyink [16] use CSG representations of polygons for point location. We include their code[1], *csg*, in our case study because of the reported efficiency, although the code is not publicly available. Finally, we also consider point location code for polygons from CGAL using the obvious inexact geometry kernel with double precision floating-point coordinates. Of course, with an exact kernel, CGAL's point location code for polygons is fully reliable. The same holds for LEDA's rat_polygon code (floating-point filtered rational kernel).

Regarding polygon data, we use both artificial "random" polygons and real world data. The random polygons are generated from a random set of vertices using the 2-opt heuristic [2]. The vertices are generated uniformly at random inside the unit square. The real world polygons we use are city and village boundaries from south-west of Germany, scaled to the unit square.

We use different methods to generate query points, cf. Fig. 2. For points generated uniformly at random inside the unit square using a generator from CGAL, all tested code usually works without any problems. Thus we challenge the code with problem-dependent (near) degeneracies. We use CGAL's point generator for generating points "on" a line segment. Given endpoints v and w of a segment and a number $n > 2$, this generator creates a sequence of equally spaced points on the segment $\overline{vw}$, more precisely, it creates points $v + \frac{i}{n-1}(w - v)$ for $i = 0, \ldots, n-1$ and hence the generated set of points includes the segment endpoints. Because we use double precision coordinates, usually not all points are exactly on the segment, but only very close to it. Next we consider potentially algorithm-dependent degeneracies. We create points on the vertical and horizontal lines through the polygon vertices. These are potentially degenerate cases for the crossing number algorithms. Furthermore we generate query points "on" the diagonals bounding the triangles considered by the triangle-fan algorithms,

[1] Thanks to Robert Walker for making the code available to us.

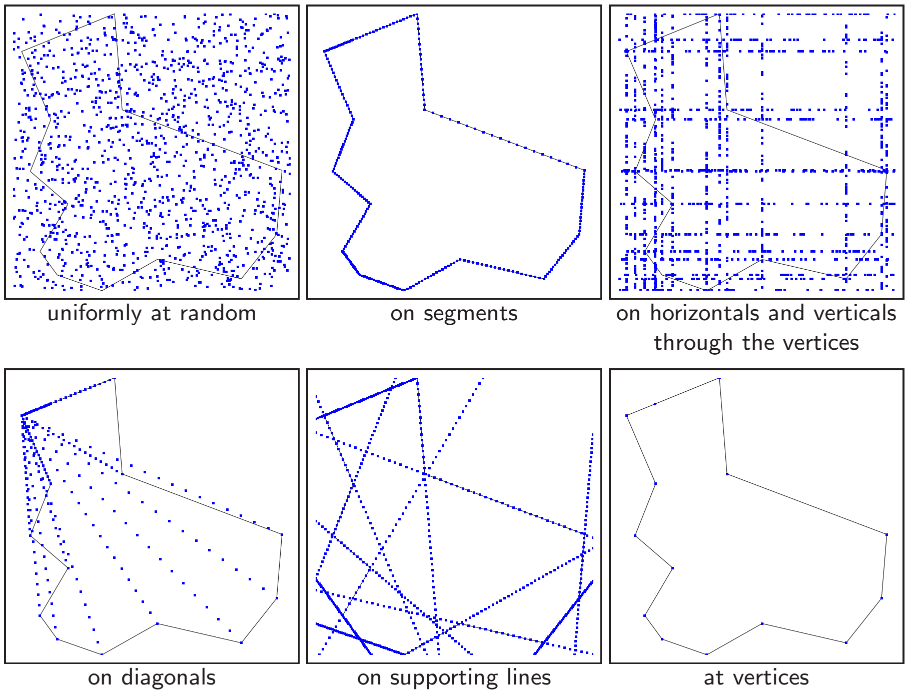

Fig. 2. Query point generators.

again using a generator from CGAL. These are potentially degenerate cases for such algorithms. Besides this we generate points "on" the lines supporting the polygon edges in order to challenge the code based on CSG representations. Finally we use the polygon vertices as query points.

A strong point of this case study is its independence: we test only code not written by the author! For the sake of reproducibility, testbed code and data are available online [12].

4 Experimental Results

The reported results hold on any machine compliant to the IEEE 754 double precision floating-point standard. As stated above, all tested code works very well for random query points. Thus we turn to more interesting query points. Points (almost) on the polygon edges are challenging for all strategies. Fig. 3 and Fig. 4 show typical results for a real-world and a random polygon, respectively.

All tested code produces some false results. Table 1 shows the percentage of false positives and false negatives for query points near the polygon edges for the polygons from Figs. 3 and 4. For all but the CGAL code the percentage of false results is significant. However, the CGAL code is about an order of magnitude

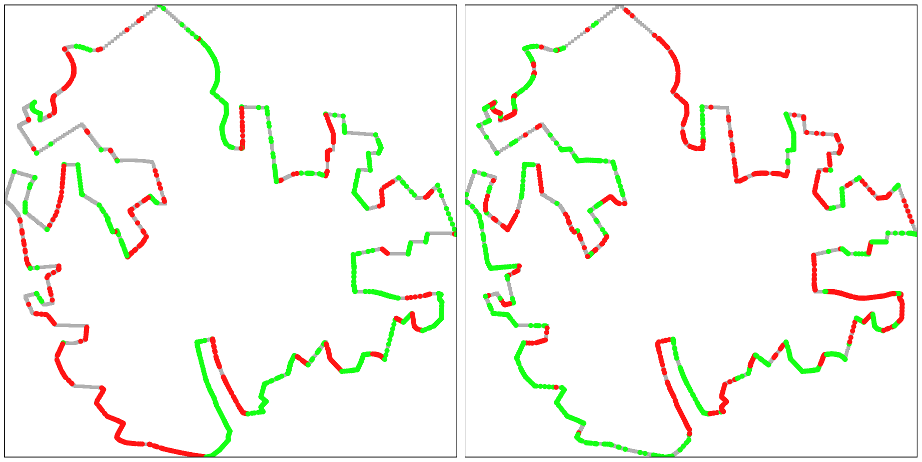

Fig. 3. Results for query points near or on the edges of a a real-world polygon with 304 vertices for strategies PNPOLY (left) and *grid* (right).

slower than the fastest algorithms from [7], but it is also much more reliable for (nearly) degenerate query points. Especially, polygon vertices cause problems for all but the CGAL code, see also Table 4.

Unfortunately, it is not obvious how to measure sensitivity to errors as a function of closeness to degeneracy, because, as illustrated by Kettner et al. [9] for the orientation test, the set of floating-point points near an edge where a floating-point implementation of a geometric predicate fails does not necessarily form a homogeneous sleeve around the edge.

Table 1. Results for points "on" polygon edges.

	real-world polygon 304 vertices		random polygon 64 vertices	
	false positives	false negatives	false positives	false negatives
crossings	12.5 %	17.8 %	18.4 %	18.8 %
weiler	11.9 %	18.0 %	13.4 %	23.0 %
halfplane	15.2 %	24.0 %	16.3 %	24.8 %
barycentric	20.2 %	17.5 %	17.9 %	14.1 %
spackman	20.2 %	18.5 %	19.6 %	15.2 %
grid	16.7 %	15.9 %	17.0 %	19.7 %
PNPOLY	11.9 %	18.0 %	13.5 %	23.4 %
csg	23.9 %	23.7 %	26.0 %	18.0 %
CGAL	0.4 %	0.0 %	3.4 %	0.6 %

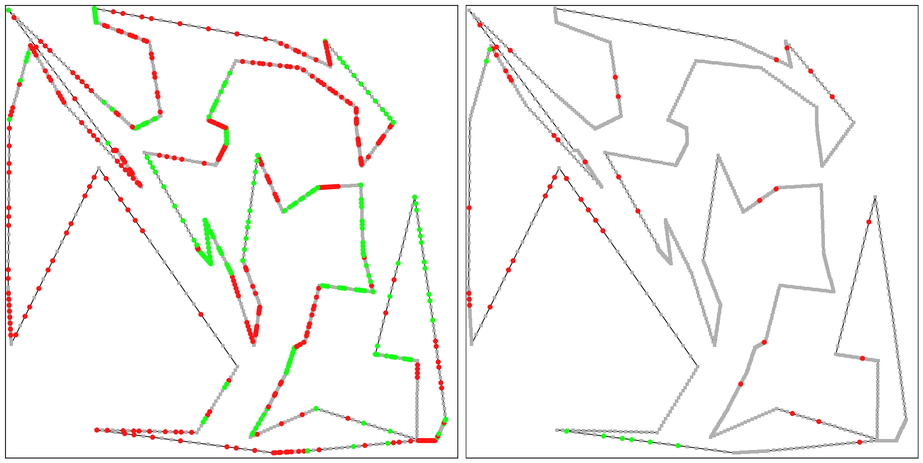

Fig. 4. Results for query points near or on the edges of a random polygon with 64 vertices for strategies *csg* (left) and CGAL (right).

Interestingly, Shimrat [14] already states that his crossing number algorithm does not apply to query points on the boundary of the polygon. Haines [7] writes

> "*When dealing with floating-point operations on these polygons we do not care if a test point exactly on an edge is classified as being inside or outside, since these cases are extremely rare.*

However, our experiments show that we get false results not only for points *exactly* on the boundary. Second, for polygons with axis-parallel edges like the polygon in Fig. 5, points exactly on the edges are not unlikely to arise in real-world applications.

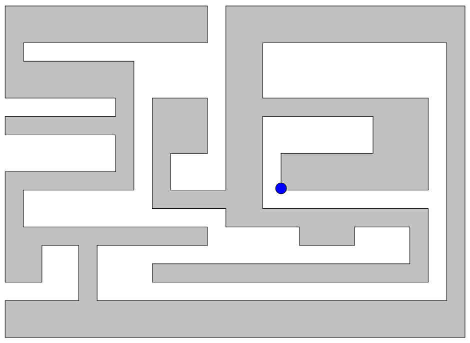

Fig. 5. For polygons with axis-parallel edges query points exactly on an edge are not unlikely.

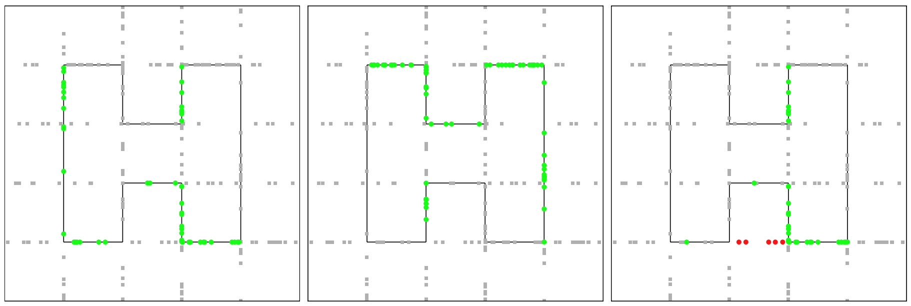

Fig. 6. Results for query points on verticals and horizontals through the vertices of a polygon with axis-parallel edges for *crossings* (left), PNPOLY (middle), and *barycentric* (right)

Next we turn to algorithm-dependent degeneracies. We create points on the vertical and horizontal lines through the polygon vertices. These are potentially degenerate cases for the crossing number algorithms. Because of a conceptual perturbation, namely considering vertices on the ray as being infinitesimally above the ray, both crossing number algorithms work very well for random query points on the horizontals and verticals through vertices, unless we have axis-parallel edges, see Fig. 6.

Next we generate query points (almost) on the diagonals that bound the triangles considered in the triangle-fan algorithms. Fig. 7 illustrates some results. Table 2 shows the percentage of false positives and false negatives for query

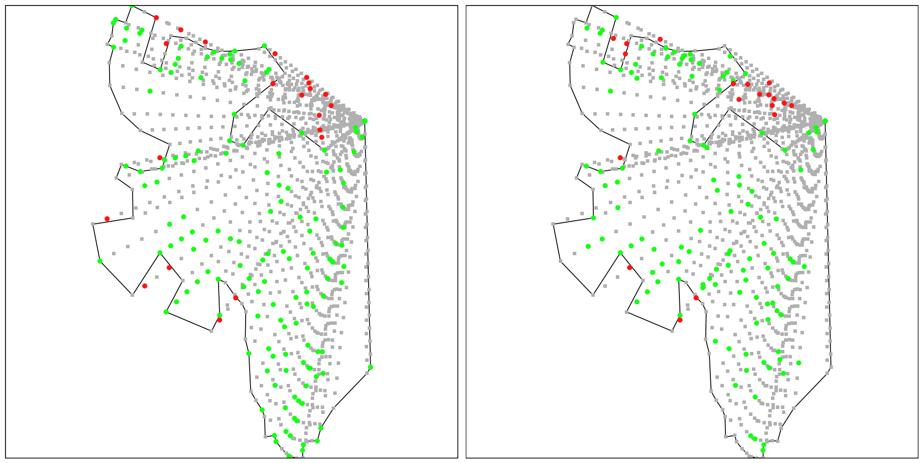

Fig. 7. Results for query points on diagonals of a real-world polygon with 78 vertices for *spackman* (left) and *barycentric* (right).

points generated as described above for the real world polygon from Fig. 7 and for the random polygon from Fig. 4. As we have suspected the triangle-fan algorithms err for many points near or on the edges of the triangles considered by the algorithms. We have many false results, both false positives as well as false negatives. The percentage is higher for the triangle-fan algorithms compared to their competitors. Haines [7] admits that the triangle-fan based code "*does not fully address this problem*". Again, the problems occur not only for points exactly on triangle edges.

Table 2. Results for points "on" diagonal edges.

	real-world polygon 78 vertices		random polygon 64 vertices	
	false positives	false negatives	false positives	false negatives
crossings	0.6 %	7.8 %	0.8 %	3.3 %
weiler	0.2 %	6.8 %	0.7 %	3.4 %
halfplane	5.5 %	25.5 %	16.1 %	25.6 %
barycentric	1.2 %	11.9 %	6.4 %	12.8 %
spackman	1.4 %	13.5 %	6.7 %	14.0 %
grid	0.6 %	7.7 %	0.4 %	2.9 %
PNPOLY	0.3 %	6.8 %	0.7 %	3.4 %
csg	0.0 %	7.7 %	0.0 %	8.3 %
CGAL	0.0 %	0.0 %	0.0 %	0.0 %

There are usually more false-negative results, because the total part of diagonal edges inside a polygon is usually larger. The false results of the remaining methods are mainly caused by query points on the first diagonal which coincides with the first edge of the polygon and by the endpoints of the diagonals, i.e., polygon vertices. Since vertices are false negative if misclassified, this explains the larger percentage of false negatives for the remaining non triangle-fan based methods.

We move on to query points on the lines supporting the polygon edges. The corresponding generator was added to the test set in order to challenge the *csg* method. Fig. 8 shows results for *csg* for the real-world and the random polygon discussed in Table 3. For the real-world polygon from Fig. 8, the percentage of both false negatives and false positives is below 1% for all competitors. Indeed, *csg* reliability is slightly worse than the reliability of the crossing number algorithms. In some examples, however, the triangle-fan algorithms are worse. As can be seen in Fig. 8, *csg* errs only for query points near the actual polygon edges, not elsewhere. The same behavior shows up in all our other experiments with query points on supporting lines. In all examples we study, CGAL produces the largest number of correct results.

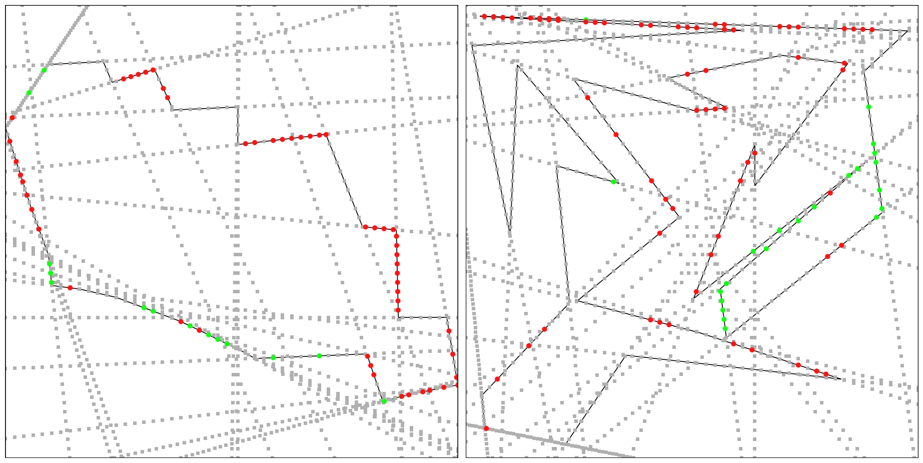

Fig. 8. Results for query points on supporting lines of polygon edges for a real-world polygon with 34 vertices (left) and a random polygon with 32 vertices (right) for *csg*.

Table 3. Results for points "on" supporting lines of polygon edges.

	real-world polygon 34 vertices		random polygon 32 vertices	
	false positives	false negatives	false positives	false negatives
crossings	1.1 %	0.6 %	1.7 %	1.4 %
weiler	0.4 %	0.9 %	0.8 %	1.4 %
halfplane	0.8 %	1.9 %	3.6 %	7.2 %
barycentric	2.2 %	0.7 %	5.1 %	2.2 %
spackman	2.1 %	0.8 %	5.4 %	2.7 %
grid	1.2 %	1.0 %	2.8 %	1.8 %
PNPOLY	0.4 %	0.9 %	0.8 %	1.2 %
csg	3.1 %	0.9 %	4.2 %	1.4 %
CGAL	0.3 %	0.0 %	0.6 %	0.0 %

Finally, we take a closer look at polygon vertices as query points. Since we consider our polygons as topologically closed, there can be false negatives only. Unfortunately, most algorithms produce many wrong results, as shown in Table 4. Besides CGAL, all err for about half the query points! If we would consider polygons to be open, the other "half" of the vertices would be misclassified. Besides CGAL, all produce inconsistent results for polygon vertices as query points. If you are not willing to accept this, you have to add a separate routine for checking whether a query point coincides with a polygon vertex.

Table 4. Results for polygon vertices as query points.

	real-world polygon 304 vertices	random polygon 256 vertices
	false negatives	false negatives
crossings	52.3 %	51.6 %
weiler	48.4 %	49.6 %
halfplane	53.3 %	54.7 %
barycentric	41.4 %	45.3 %
spackman	48.0 %	48.0 %
grid	45.4 %	50.4 %
PNPOLY	48.4 %	48.0 %
csg	42.8 %	49.2 %
CGAL	0.0 %	0.0 %

5 Reliability and Numerical Stability

For all tested programs there are query points where they produce incorrect results. Besides the triangle-fan algorithms, all algorithms are apparently numerically stable, i.e., whenever they err there is a point close to the current query point for which the computed result is the correct one. The triangle-fan based programs, however, also compute incorrect results far away from the boundary edges, cf. Fig. 1 (right) and Fig. 7. Hence, they are not numerically stable. Although the remaining programs are numerically stable, they show fairly different behavior for challenging queries.

Wesselink's CGAL code is by far the most reliable, but is also much slower, although it is an implementation of the crossing number algorithm as well. The code does a lot of reasoning based on comparison of coordinates. In terms of efficiency this is not a good idea as the branching breaks the pipelining in the processor. In terms of reliability, in IEEE 754 compliant architectures, it is a great approach, because coordinate comparisons are always exact thanks to denormalized floating-point numbers. CGAL competitors prefer to perform numerical computations instead of coordinate comparisons. These computations are fast, but error-prone. The crossing number algorithms test whether a horizontal leftward ray r starting at query point $q = (q_x, q_y)$ intersects a polygon edge s. This is often implemented by computing the intersection point p of the supporting line of r and the supporting line of s and then testing whether p lies on both r and s. MacMartin et al. [11] observe that s cannot intersect r if the y-coordinates of both endpoints of s are smaller or larger than q_y. CGAL uses comparison of x-coordinates to save further calculations as well, assuming that we already did the comparison of y-coordinates. Then, if the x-coordinates of both endpoints of s are smaller than q_x, there is no intersection, and if both are larger, there is one. If these comparisons do not suffice to decide the test, CGAL uses an orientation test to check whether q is to the left of s.

The straightforward approach to implement geometric algorithms reliably is to use exact rational arithmetic instead of inherently imprecise floating-point arithmetic. Unfortunately, this slows down the code by orders of magnitude. As suggested by the exact geometric computation paradigm [18] a better approach is to combine exact rational arithmetic with floating-point filters, e.g. interval arithmetic, in order to save most of the efficiency of floating-point arithmetic for nondegenerate cases. This approach is implemented in the exact geometry kernels of CGAL [3] and LEDA [10]. The use of adaptive predicates à la Shewchuck [13] is highly recommended. Interestingly, exact rational arithmetic does not suffice to let the tested crossing number code always produce correct results. Due to the conceptual perturbation of the vertices, vertices as query points still cause incorrect results many times.

We briefly describe an alternative reliable implementation of the crossing number algorithm. We suggest to add some preprocessing to compensate for more expensive arithmetic. Use an interval skip list (or interval tree) to store the y-ranges of all nonhorizontal polygon edges. In order to handle degeneracies correctly, store half-open intervals: Only the y-coordinate of the first endpoint is included, the y-coordinate of the second endpoint is not. Here we assume that polygon edges are consistently oriented along the polygon boundary. Use another interval skip list (or a dictionary data structure) to store the y-coordinates of all vertices and all horizontal edges. The CGAL library provides a flexible and adaptable implementation of interval skip list. Note that all operations on the interval skip lists are exact, because we need only comparisons of floats besides arithmetic on small integers.

To answer a query for $q = (q_x, q_y)$, we use the second interval skip list (or a dictionary data structure) to check exactly whether q lies on a horizontal ray or coincides with a polygon vertex. If not, we use the first skip list to get candidate edges for intersection with the leftward horizontal ray starting at q and use the comparison-based strategy described above for testing for intersection. Thanks to the half-openness of the intervals, we count intersections at vertices only once. In pathological cases we still have to consider a linear number of edges and vertices, in practice, however, we get only a few, leading to good performance for most random and real-world polygons.

6 Conclusions

Our experiments show that the tested practical point-in-polygon code often produces wrong results for challenging queries, where we find inconsistent handling of vertices as query points most annoying. The experiments also show that the triangle-fan based code is not even numerically stable. Furthermore, the experiments show that the slower CGAL code is much more reliable.

References

1. T. Akenine-Möller and E. Haines. *Real-Time Rendering (2nd Ed.)*. AK Peters, Ltd., 2002.
2. T. Auer and M. Held. *Heuristics for the generation of random polygons.* In Proc. of CCCG 96, pages 38–44. 1996.
3. CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.
4. A. R. Forrest. *Computational geometry in practice.* In R. A. Earnshaw, editor, Fundamental Algorithms for Computer Graphics, volume F17 of *NATO ASI*, pages 707–724. Springer-Verlag, 1985.
5. W. R. Franklin. PNPOLY–point inclusion in polygon test. http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html.
6. R. Hacker. Certification of algorithm 112: position of point relative to polygon. *Commun. ACM*, 5:606, 1962.
7. E. Haines. *Point in polygon strategies.* In P. Heckbert, editor, Graphics Gems IV, pages 24–46. Academic Press, Boston, MA, 1994. http://tog.acm.org/editors/erich/ptinpoly/.
8. I. Haran and D. Halperin. *An experimental study of point location in general planar arrangements.* In Proc. of ALENEX 06, pages 16–25. 2006.
9. L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap. *Classroom examples of robustness problems in geometric computations.* Comput. Geom. Theory Appl., 40(1):61–78, 2008.
10. K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.
11. A. Nassar, P. Walden, E. Haines, T. Dickens, R. Capelli, S. Narasimhan, C. Jam, S. MacMartin. *Fastest point in polygon test.* Ray Tracing News, 5(3), 1992.
12. S. Schirra. Companion pages to *How reliable are practical point in polygon strategies?* http://wwwisg.cs.uni-magdeburg.de/ag/pointInPolygonReliability/.
13. J. R. Shewchuk. *Adaptive precision floating-point arithmetic and fast robust geometric predicates.* Discrete & Computational Geometry, 18(3):305–368, 1997.
14. M. Shimrat. Algorithm 112: position of point relative to polygon. *Commun. ACM*, 5:434, 1962.
15. J. Snoeyink. *Point location.* In J. E. Goodman and J. O'Rourke, editors, Handbook of Discrete and Computational Geometry (2nd Ed.), chapter 34, pages 767–786. CRC Press LLC, Boca Raton, FL, 2004.
16. R. Walker and J. Snoeyink. *Practical point-in-polygon tests using CSG representations of polygons.* In Proc. of ALENEX 99, pages 114–123. 1999.
17. K. Weiler. *An incremental angle point in polygon test.* In P. Heckbert, editor, Graphics Gems IV, pages 16–23. Academic Press, Boston, MA, 1994.
18. C.-K. Yap. *Towards exact geometric computation.* Comput. Geom.–Theory and Appl., 7:3–23, 1997.