Point-in-Polygon Tests by Determining Grid Center Points in Advance

Jing Li* and Wencheng Wang†
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
E-mail: lij@ios.ac.cn Tel: +86-010-62661655
† State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
E-mail: whn@ios.ac.cn Tel: +86-010-62661611

***Abstract*—This paper presents a new method for point-in-polygon tests via uniform grids. It consists of two stages, with first to construct a uniform grid and determine the inclusion property of every grid center point, and the second to determine a query point by the center point of its located grid cell. In this way, the simple ray crossing method can be used locally to perform point-in-polygon tests quickly. When $O(N_e)$ grid cells are constructed, as used in many grid-based methods, our new method can considerably reduce the storage requirement and the time on grid construction and determining the grid center points in the first stage, where N_e is the number of polygon edges. As for answering a query point at the second stage, the expected time complexity is $O(1)$ in general. Experiments show that our new method can be several times faster than existing methods at the first stage and an order of magnitude faster at the second stage. Benefited from its high speed and less storage requirement, our new method is very suitable for treating large-scale polygons and dynamic cases, as shown in our experiments.**

I. INTRODUCTION

The point-in-polygon test, which determines whether a point is inside a polygon, is one of the basic problems in computational geometry. It has wide applications in many areas such as geographic information system (GIS) and computer graphics [1]. Though many efficient methods have been developed, it is still necessary to find more efficient solutions to the problem because the test is the basis of many other algorithms, such as polygon clipping and ray tracing solids, and has substantial influence on their efficiency.

Among the existing methods, those without preprocessing usually compute a certain parameter for determining the inclusion property of a query point, such as the ray-crossing number [2], the sum of angles [3][4], the sign of offset [5] and the summation of the sign of the triangles [6][7]. As every polygon edge must be checked, their time complexity are all $O(N_e)$, where N_e is the number of the polygon edges. To speed up the computation of the relevant parameter, many methods have been proposed to decompose polygon into simpler components, such as trapezoids [1][8], convex polygons [9] and layers of edges [10]. Besides them, various spatial subdivision structures have also been introduced to reduce the complexity of point-in-polygon tests, for example grids [11] [12], and hierarchical subdivision by means of tri-cones [13]. They can have time complexity smaller than $O(N_e)$ for an inclusion test, and are very useful for testing many points against the same polygon. For example, both the methods based on trapezoids and convex polygons have a time complexity of $O(\log N_e)$ for an inclusion test. However, these methods generally take a long time to create the subcomponents in preprocessing, and need $O(N_e \log N_e)$ time. This prevents them from treating dynamic situations when the polygon is either deformable or moving. For example, they may seriously lower the efficiency of finding the safe sites in a flooded city, where the polygons for approximating the flooded region may change frequently. Point-in-polygon methods without preprocessing [14] may handle dynamic cases efficiently when the number of query points is small. However, the high complexity of their inclusion test can notably decrease the total efficiency when answering a large number of query points.

Grid-based methods are an important approach for inclusion tests. Though their expected time complexities for an inclusion test are not as good as the methods based on trapezoids or convex polygons, they have widespread applications, especially for dynamic situations, due to their ease of implementation.

Zalik and Kolingerova [11] gave a *CBCA* (Cell-based Containment Algorithm) method for preprocessing a polygon by subdividing its bounding rectangle into uniform-sized cells, recording each edge of the polygon in the cells crossed by the edge and classifying the empty cells as inside or outside the polygon, where an *empty cell* refers to a grid cell without any edges crossed. Let q be a query point and $Cell(q)$ be the cell containing q. If $Cell(q)$ is empty, the inclusion property of q is obtained directly from that of $Cell(q)$. Otherwise, the edge (or vertex) stored in $Cell(q)$ that is closest to q is found, and the inclusion property is then determined from the orientation of q with respect to the closest edge (or the convexity property of the closest vertex). This method has expect time complexities of $O(N_e^{3/2})$ and $O(N_e^{1/2})$ for preprocessing and an inclusion test respectively, and requires $O(N_e)$ storage. However, this may fail when the closest point is out of $Cell(q)$ or when the adjacent edges of the vertex are nearly collinear. With regard to this, Yang et al. [12] proposed using a quasi-closest point to represent the closest point in a certain direction, and then presented a criterion for obtaining the inclusion property from the quasi-closest point.

In 1994, it was proposed to combine the ray crossing method with the grid structure by pre-computing the inclusion property of the corners of grid cells [15], which performs an inclusion test by producing a line segment between the test point and a corner of the grid cell containing the test point. However, such an idea has not been investigated comprehensively, whose potentials have not been exploited.

This paper makes a deep study on combining the ray crossing method with the grid structure and proposes a grid-based center point method (GCP). We compute in advance the inclusion property of the center points instead of the corners of grid cells, optimize the grid resolutions, efficiently handle the singular cases and reveal $O(1)$ time complexity for an inclusion test in general. By the comparison results with state-in-art methods, our method can considerably reduce the time cost on preprocessing and inclusion tests while having a very low storage requirement. In particular, the more the edge, the more acceleration we can gain. All these show that our method is much superior to the existing techniques.

In the remainder of the paper, we will describe the new method and its implementation in Section II, and then use experiments to compare it with existing methods in Section III. Finally, conclusions are drawn in Section IV.

II. Our New Method

Our new method consists of two stages, the first for constructing the grid and determining the inclusion properties of the center points, and the second for answering inclusion queries.

A. Grid Construction

In grid construction, we need to decide the grid resolution for subdividing the grid's bounding box into uniform-sized cells, and then record each cell which edges cross it. We set the grid's bounding box a little bit bigger than the polygon's bounding box to avoid the problems caused by finite arithmetic.

After the cells are produced, every edge is examined to find which cells it crosses. There have been many investigations on this problem [16], and the Digital Differential Analyzer (DDA) method by Cleary and Wyvill [17] is regarded very efficient, which works by marching cell by cell along an edge from a vertex. As this method is simple to implement, we use it to detect the cells that are crossed by an edge.

B. Determining Center Points

In determining the center points, we try to take advantage of the neighboring points whose inclusion property are known, by which the ray crossing method can be used locally to save cost. As illustrated in Fig. 1, we first treat the cells on the boundary of the grid. This is because for such boundary cells, it is easy and cheap to determine the center points by their neighboring points outside the grid, which must be outside the polygon. Then starting from these boundary cells, we treat the remaining cells in sequence until all cells have been treated, e.g. treating the cells along the traverse paths shown by blue arrows in Fig. 1(a).

Of course, other traverse paths can also be used, as illustrated with an example in Fig. 1(b). In general, we prefer to use vertical or horizontal line segments to form traverse paths because this is very efficient to apply the ray crossing method, where only the edges in at most two cells should be considered for determining a center point.

In comparison with the method of [15], our method can be more efficient on pre-computation. This is because the method of [15] is based on corner points and a corner point may be shared by four cells, all of which should be considered to determine the inclusion property of the corner. Due to the same reason, we can also be faster on inclusion tests.

Here, a singular case may occur that a center point is on a polygon edge or overlaps a vertex of the polygon. For this, we just mark such a center point as "singular" and then treat the next center point along the corresponding traverse path. As the count results of the crossed edges for determining previous center points can be reused, such a singular case will not impact the computational efficiency of determining the next center point. As illustrated in Fig. 1(a), suppose we know that center point O is non-singular and inside the polygon and center point A is singular, we want to determine the inclusion property of center point B along their traverse path. Because we know that line segment OA crosses no edge and so the count number of crossed edge is 0, we can then easily know that line segment OB crosses one edge and so the counted number becomes 1. As point O is inside the polygon, we know that point B is outside the polygon. In this way, we can determine all center points each with the edges of two cells, no matter whether there are singular cases.

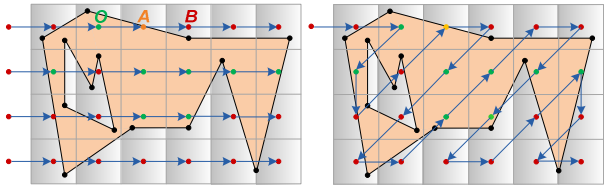

Fig. 1 The center points have their inclusion properties determined one by one along the traverse path represented by blue arrows, starting from the points outside the bounding box of the grid, which must be outside the polygon.

C. Inclusion Tests

We perform an inclusion test by linking a line segment from the test point to the center point of the cell that contains the test point. As illustrated in Fig. 2, for test point q_1, the line segment connecting it with center point c_1 crosses two edges, and c_1 is outside the polygon, so that q_1 is outside the polygon. When the center point is singular, we search cells one by one from the cell containing the test point in a vertical or horizontal direction, until a non-singular center point is found. Then, we link a line segment from the test point to the found non-singular center point to determine the inclusion property of the test point. In Fig. 2, center point c_2 of the cell containing the test point q_2 is singular. However, we can find the non-singular center point c_3 when we search to the right

from the cell containing q_2. Thus, by the line segment q_2c_3, we can compute the inclusion property of q_2. Because singular center points are very few, a non-singular center point can be quickly found by searching very few cells, which is to be discussed in Section II.D. In our tests, we can get a non-singular center point after searching at most one or two nearby cells. Thus, singular center points will have little impact on the efficiency of inclusion test.

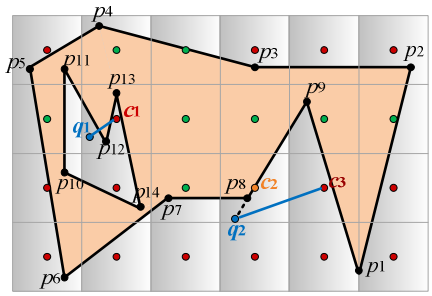

Fig. 2 The inclusion property of a test point is determined by one of its nearby non-singular center points. Here, the center points that are inside the polygon, outside the polygon and singular are marked in green, red and yellow respectively,

D. *Algorithm Analysis*

Probability for singular center points

The probability for center points to be singular is actually the probability for center points to be on polygon edges, denoted by p. In implementation, a point is usually judged as overlapping a line segment if the distance between them is less than a tolerance value ε. Therefore, for each edge, widening the edge ε distance from it on its both sides will form a region to cause singular cases. Thus p equals the area ratio between the widened regions of all edges and the grid's bounding box. As ε is usually very small, the value of p is very low in general. When a singular center point is met, the expected number of the cells to be searched for getting a non-singular center point, N_c, will be $N_c < 1*(1-p) + 2*(1-p)*p + 3*(1-p)*(1-p)*p + \cdots < 1/(1-p)$. Because p is very small, not greater than 0.5 in general, N_c will be 2 at most. Though very rare, there may exist an extreme case that many nearby cells have singular center points. With regard to this, we can adopt the measure that Haines took [15], reducing singular center points by reconstructing the grid structure by changing the grid resolutions or enlarging the bounding box a little. However, in fact, we did not experience such a case in our tests.

Storage and time complexities

For any method using a grid structure, its time and storage complexities are dependent on the grid resolution, namely the number of grid cells M. Suppose an edge intersects f_e cells on average. Then, a cell contains f_eN_e/M edges on average. With these, we first analyze our storage complexity, and then our time complexity in the following.

Storage complexity. In our method, every cell needs two bits for its center point, with one marking whether it is singular and the other recording its inclusion property. Thus, we need $O(M)$ bits for M center points. Besides this, we need $O(N_e)$ storage for recording polygon vertices and $O(f_eN_e)$ storage for edges in the cells. As $f_e \geq 1.0$, our storage requirement is $O(M)+O(N_e)+O(f_eN_e) = O(M+f_eN_e)$ in total.

Time complexity. Our time complexities for preprocessing and inclusion tests are analyzed respectively in the following paragraphs.

In preprocessing, we have three main operations. The first one is constructing the grid, which has a complexity of $O(M)$. The second one is inserting edges into cells. We use the method in [17] to treat edges, which has a complexity linear with the number of the cells intersected by edges. Thus, we need $O(f_eN_e)$ time for this operation. The third one is computing the inclusion properties of center points. For this operation, an edge in a cell will be computed twice at most, with one for determining the center point of the current cell and the other for determining the center point of the next cell along the related traverse path. Therefore, every edge will be visited $2f_e$ times, so we need $O(f_eN_e)$ time for computing the inclusion properties for all center points. In sum, we need $O(M+f_eN_e)$ time for preprocessing.

For an inclusion test, we first need to find the cell containing the test point, which has $O(1)$ time complexity. Next, we need to check all the edges in the cells that are intersected by the line segment from the test point to a non-singular center point. According to the analysis in the above, normally only fewer than 2 cells are expected to be investigated before a non-singular center point is found. As a cell has f_eN_e/M edges on average, we need $O(f_eN_e/M)$ time. So all together we need $O(1+f_eN_e/M) = O(f_eN_e/M)$ time for an inclusion test.

In sum, our method needs $O(M+f_eN_e)$ storage, $O(M+f_eN_e)$ time for preprocessing and $O(f_eN_e/M)$ time for a query. When $M=O(N_e)$, as other grid-based works did [11][12], the complexity of storage, preprocessing time and a query time are $O(f_eN_e)$, $O(f_eN_e)$ and $O(f_e)$ respectively. As discussed in [11][12], f_e is $O(N_e^{1/2})$ in the worst case. However, except extreme cases, f_e is $O(1)$ in general, which was verified by many tests in this paper. Though our method has the same time complexity with the methods using grids [11][12], we can still get much acceleration by using simple calculations, which will be discussed in Section III.

E. *Optimizing Grid Construction*

According to the analysis in the last subsection, we can reduce f_e to lower the complexities. When the cells are in a shape consistent with the directions of most edges, f_e could be reduced. As illustrated in Fig. 3, to the same polygon, f_e is 3.5 when the blue grid is used and 2.83 when the red grid is adopted, though both the grids have 6 cells. Thus, we develop the following measure to optimize grid construction to promote inclusion tests.

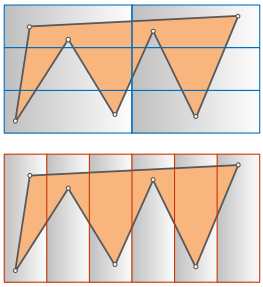

Fig. 3 Different grid construction leads to different *fe*, the average cell number intersected by an edge.

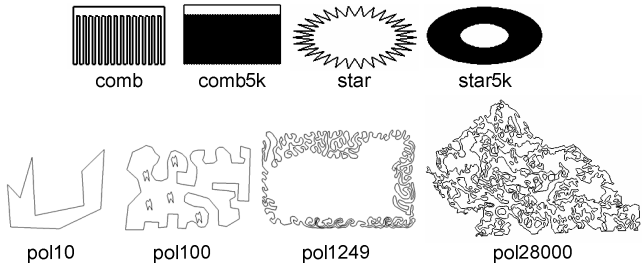

Fig. 4 Polygons used in test.

TABLE I
STATISTICS FOR DIFFERENT GRID CONSTRUCTION STRATEGIES

	comb5k		pol28000		star5k	
	old	ours	old	ours	old	ours
N_e	5000		28012		5000	
l_x/l_y	0.0016		1.0		2.0	
f_e	23.44	2.64	1.2	1.2	42.0	27.0
M_x*M_y	91* 54	2282* 2	209* 133	207* 135	100* 49	70* 70
T_p	0.02	0.002	0.009	0.009	0.02	0.02
T_q	0.6	0.2	0.1	0.1	5.2	2.2
S	961	133	500	501	1161	1097

Note: old and ours refer to the grid resolutions computed with existing methods ((2), k=1) and our improved measure ((1)) respectively. M is set to N_e for both equations. The difference between M and M_x*M_y is due to the rounding operation. T_p and T_q refer to preprocessing time (second) and test time (second) for testing 1,000,000 random points uniformly sampled in the grid. S is the storage cost for grid structures (KB).

Suppose the grid resolutions along X and Y directions are M_x and M_y respectively and $M=M_x*M_y$. Let $P_i(x_i,y_i)$ and $P_{i+1}(x_{i+1}, y_{i+1})$ be the two vertices of the edge P_iP_{i+1} ($1\le i\le N_e$ and $P_{N_e+1}=P_1$). Let l_x and l_y denotes the average edge length along X and Y direction respectively. Then

$l_x = (\sum_{i=1}^{N_e} |x_i - x_{i+1}|)/N_e$, $l_y = (\sum_{i=1}^{N_e} |y_i - y_{i+1}|)/N_e$. Thus, we can get the following equation

$$f_e = l_xM_x/W + l_yM_y/H + 1,$$

where W and H are the width and height of bounding box of the grid. To promote inclusion tests, the value of f_e should be lowered. As we know, $f_e = l_xM_x/W+l_yM_y/H+1 \ge 2(Ml_xl_y/(WH))^{1/2}+1$, and f_e is the smallest when $l_xM_x/W = l_yM_y/H$. Thus, we can reformulate M_x and M_y in the following equation to minimize f_e,

$$M_x = \sqrt{\frac{MW}{H}\frac{l_y}{l_x}},\ M_y = \sqrt{\frac{MH}{W}\frac{l_x}{l_y}} \quad (1)$$

Therefore, we can use Equation (1) to optimize grid construction. With this, our grid construction is different from the usual way taken by existing methods [11][12] to calculate M_x and M_y by the following equation,

$$M_x = k\sqrt{\frac{MW}{H}},\ M_y = k\sqrt{\frac{MH}{W}} \quad (2)$$

(k is an empirical value), which do not consider the impact of direction of edges on inclusion tests. We made tests to investigate the improvement of our grid construction strategy. Three typical test polygons are selected from Fig. 4 to perform the test and the statistics are listed in Table I. The first one is "comb5k", which has a similar shape with polygon "comb" but with more edges. The second one is "pol28000". And the third one is "star5k", which is similar with polygon "star" except for the number of edges. From Table I, it is clear that when l_x and l_y differ greatly, like polygon comb5k, our strategy can efficiently decrease the value of f_e. Therefore, both the time and the storage performance can be greatly improved. For pol28000, whose l_x nearly equals l_y, the two strategies has similar performance. That is to say, by taking edge direction into consideration, our optimized grid partition strategy can handle more type of polygons nicely and help to improve the inclusion test.

III. RESULTS AND COMPARISON

To test efficiency of the new method, we performed tests on a personal computer with an Intel(R) Core(TM)2 2.83GHz CPU and 4GB RAM. We compared our method with several typical methods, namely the basic ray-crossing method [15], the method by convex decomposition (CD) [9] and the cell-based containment algorithm (CBCA) [11]. As the center point approach can also be applied to other spatial subdivision structures, it is interesting to compare such methods with our grid based method. Therefore, we also implemented and compared the quad-tree based method using the ray crossing method locally with the center points of the leaf nodes determined in advance (quad-tree). In the following, we will first discuss our new method (GCP) in comparison with the other methods, then present the experimental results to show the advantages of GCP over these methods. Afterwards, we will discuss an application for treating dynamic cases.

A. Comparison with Typical Methods

The basic ray-crossing method [15] acts as the benchmark for point-in-polygon tests. It answers a query point by shooting a ray from the query point to intersect polygon edges. If there are an odd number of edges intersected, the query point is inside the polygon; otherwise, it is not. This method is easy to implement and does not need preprocessing, requiring $O(N_e)$ time to answer an inclusion query. In comparison, GCP applies ray-crossing locally by only checking the edges in one or at most a few cells in

singular cases, with expected time complexity $O(1)$ in general. Thus, GCP can be much faster than ray-crossing.

The method by convex decomposition (CD) [9] works by decomposing the polygon into convex polygons and uses a tree structure to manage the convex polygons hierarchically. It has time complexities for preprocessing and an inclusion test of $O(N_e \log N_e)$ and $O(\log N_e)$ respectively, the same as that for the popular trapezoid-based method. However, CD is faster than the trapezoid-based method because there are usually much fewer convex polygons created than trapezoids. For an inclusion test, CD first finds the convex polygon that may contain the query point by traversing the tree structure, and then uses a simple method to decide whether the query point is inside the convex polygon. In comparison with CD, GCP has a lower time complexity for preprocessing, and has a higher determination speed in general since the tree traversal of CD may take much time, except in the case that the polygon has very few edges making the traversal very fast. This will be demonstrated by the tests that follow.

The cell-based containment algorithm (CBCA) [11] is a popular grid-based method. For a fair comparison, our GCP method uses the same grid resolutions as the CBCA method in the tests. In preprocessing, GCP is more efficient than CBCA. The main reason could be that CBCA uses the flood-fill algorithm [2] to determine whether an empty cell is inside the polygon, which requires many decision statements and branches and has a relatively random memory visit pattern. In contrast, GCP preprocesses in a fixed sequence, with fewer branches and more regular memory visits, which is better for optimizing CPU execution. As for computing the cells crossed by an edge, the Code-based algorithm [16] used in CBCA has a similar performance as the DDA method [17] used in GCP. For an inclusion test, GCP only uses the simple ray-crossing method locally, benefitting from the predetermined center points, while CBCA needs a more complex computation for finding the closest edge to a query point and determining the inclusion property based on the query point's position relative to that edge. As a result, GCP obtains significant acceleration over CBCA.

The quad-tree has been used a lot for inclusion tests. It takes the bounding box as the root node, and iterates by subdividing a node into four child nodes until the leaf nodes each contain very few edges. This approach is very useful for treating polygons whose edges do not distribute evenly. Clearly, our strategy for the GCP method can be integrated with a quad-tree by determining the inclusion property of the center points at leaf nodes and answering a query point by the edges in the leaf node that contains the query point. However, to arrive at leaf nodes, many inner tree nodes have to be checked starting from the root node. Due to this, constructing a quad-tree may cost more time because for each edge we need know the leaf nodes it crosses, and more time is needed to locate the leaf node that contains the query point. In addition, to transmit the property from a determined center point to its neighboring center points, complex computation may be required for moving between neighboring leaf nodes in the tree. Thus, generally speaking, our GCP method should be faster than the quad-tree based method. Two parameters for the tree structure, the tree depth and the number of edges crossing a leaf tree node, have a large influence on the efficiency of inclusion tests. In our comparison tests, we ran many trials to select the parameters for which the tree structure achieves the best performance given a concrete polygon.

Test results were obtained for the four polygons used in Zalik and Kolingerova [11] , as illustrated in Fig. 4, where pol1249 and pol28000 were obtained from a GIS database. We implemented ray-crossing, CD, GCP and quad-tree method, downloaded the programs for CBCA from http://www.iamg.org/index.php?option=com_content&ask=view&id=175&Itemid=165. All the programs are serial programs, not parallel. For the performance of the inclusion tests, we randomly sampled 1,000,000 query points against every polygon respectively and checked the time for answering them all. Table II lists the statistics for the comparison tests.

The data in Table II show that GCP has similar storage requirements as CBCA. This is because the two methods have the same storage request for the edge references since they take the same grid resolutions. As the number of edges increases, GCP's storage requirement increases more slowly than CD. For the GIS polygon pol28000 in particular, its complex shape forces CD to use about 3.4 times more storage than GCP. Though we take the data for the quad-tree with the best performance, the quad-tree based method needs more storage than GCP in all cases.

As for the time spent on preprocessing and inclusion tests, our method is faster than the other methods, even by one to several orders of magnitude, and is very suitable for dealing with large polygons and a great number of query points. The results show that the acceleration ratios for GCP against CBCA are all over 9 for the inclusion tests. We didn't make tests to compare our method with some recently proposed methods. However, according to the results with the newest one [12], our GCP should be faster than it because its acceleration ratios against CBCA are below 6 for the same test polygons.

TABLE II
THE STATISTICS FOR THE COMPARISON TESTS

Polygon	Algorithm	*Tp*	*Rp*	*Tq*	*Rq*	*Tt*	*Rt*	*S*	*Rs*
pol10	ray-crossing	--	--	0.17903	2.5	0.17903	2.5	--	--
	CD	0.00003	0.5	0.05186	0.03	0.05189	0.03	0.48	-0.4
	CBCA	0.00021	9.5	0.73187	13.5	0.73208	13.5	0.772	-0.005
	quad-tree	0.00003	0.5	0.11161	1.2	0.11164	1.2	0.964	0.2
	GCP	0.00002	--	0.05046	--	0.05047	--	0.776	--
pol100	ray-crossing	--	--	3.04593	63.5	3.04593	63.3	--	--
	CD	0.00122	8.4	0.10216	1.2	0.10338	1.2	7.740	-0.2
	CBCA	0.00037	1.8	0.48632	9.3	0.48669	9.3	9.220	0.01
	quad-tree	0.00050	2.8	0.83129	16.6	0.83180	16.6	14.100	0.5
	GCP	0.00013	--	0.04725	--	0.04737	--	9.136	--
pol1249	ray-crossing	--	--	23.78159	495.9	23.78159	487.4	--	--
	CD	0.00970	10.7	0.20552	3.3	0.21521	3.4	57.480	-0.04
	CBCA	0.00277	2.3	0.44929	8.4	0.45206	8.3	60.340	0.001
	quad-tree	0.00514	5.2	6.70552	139.1	6.71066	136.8	112.396	0.9
	GCP	0.00083	--	0.04786	--	0.04869	--	60.256	--
pol28000	ray-crossing	--	--	426.54656	3905.1	426.54656	3656.9	--	--
	CD	0.24846	32.5	0.22009	1.0	0.46855	3.0	1466.28	3.4
	CBCA	0.02200	2.0	1.73531	14.9	1.75731	14.1	337.348	0.001
	quad-tree	0.08867	11.0	18.62170	169.5	18.71037	159.5	435.076	0.3
	GCP	0.00741	--	0.10920	--	0.11661	--	337.024	--

Tp is the preprocessing time, including grid construction and determining center points (seconds). *Tq* is the inclusion test time for all query points (seconds). *Tt* is the total time and *Tt*=*Tp*+*Tq* (seconds). *S* is the storage for the auxiliary structure (KB). *Rp*, *Rq*, *Rt* are the acceleration ratio for *Tp, Tq* and *Tt* repectively. *Rs* is the reduction ratio for *S*. *Rp*=(*Tp*[old]-*Tp*[GCP])/*Tp*[GCP], *Rq*=(*Tq*[old]-*Tq*[GCP])/*Tq*[GCP], *Rt*=(*Tt*[old]-*Tt*[GCP])/*Tt*[GCP], *Rs*=(*S*[old]-*S*[GCP])/*S*[GCP], where [old] and [GCP] correspond to the data of the comparison methods and of GCP respectively.

B. Treating Dynamic Cases

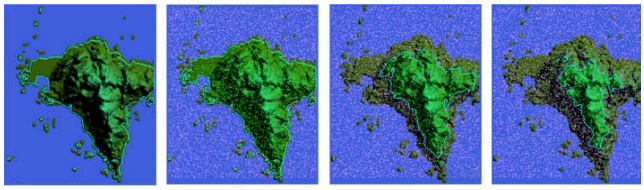

Fig. 5 Some results of the GCP method for determining safe sites in a flood-hit region. (a) is the island and (b), (c) and (d) are the test results at three different moments. Green lines marks the water level, green points are safe points while purple points are unsafe.

TABLE III
TIME STATISTICS FOR THE THREE DIFFERENT MOMENTS ILLUSTRATED IN FIG. 5.

	Fig.5(b)	Fig.5(c)	Fig.5(d)
Time for grid construction and determining center points (s)	0.000461	0.000232	0.000247
Time for inclusion tests (s)	0.001566	0.001294	0.001229

As GCP can run fast in both of its two stages, it is very suitable for handling dynamic cases such as moving or deforming polygons. Every time the polygon changes, the corresponding grids are constructed immediately so that inclusion tests can be executed in time. For example, one application is to decide whether some sites are safe in a flood-hit region. We downloaded and tested data for an island located at 1°N 126°E from ASTER Global Digital Elevation Model (ASTER GDEM) (2009). When floods hit the island, the water level and the polygons for the regions above water change frequently. We randomly sampled 10,000, 100,000 and 1,000,000 query sites on and near the island, and determined whether they are safe for people to stay with the water level changes. Fig. 5 illustrates the situations at three moments when 10,000 points are tested and Table III shows their corresponding times. The data show that monitoring the flood-hit areas can be done efficiently, and even in real time in some cases, using our method. Full tests are included in three videos attached to this paper (http://lcs.ios.ac.cn/~lij/GCP/point10K.avi, http://lcs.ios.ac.cn/~lij/GCP/point100K.avi, http://lcs.ios.ac.cn/~lij/GCP/point1M.avi).

IV. CONCLUSIONS

We have proposed a new method for fast point-in-polygon testing. By determining grid center points in advance, the new method can efficiently use coherence between the center points in grids and the query points to run the ray-crossing

method locally, and achieve a $O(1)$ time complexity on inclusion testing in general. Compared with the typical existing methods, our new method was faster in almost all our tests, by up to several times in preprocessing and by one or more orders of magnitude in inclusion tests, while keeping the storage cost to a low level. It is very suitable for treating large scale problems, which have a great number of edges and query points, and for dynamic applications.

As for the future issues of the new method, one point is to realize the new method in parallel to achieve greater acceleration, for example, by determining the center points in parallel. Another issue is optimizing the method for polygons whose edges are distributed unevenly. In treating such polygons, hierarchical grids have shown their potential and we will further study the integration of this new method with hierarchical grids.

ACKNOWLEDGMENT

We would like to thank the reviewers for their valuable comments. This work is partially supported by Natural Science Foundation of China (60873182, 61379087).

REFERENCES

[1] De Berg M, Van Kreveld M, Overmars M, Schwarzkpf O., *Computational Geometry: Algorithms and Applications*. 2nd ed., Springer, Berlin, 2000.

[2] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics-Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA, 1990.

[3] F. P. Preparata, M. I. Shamos., *Computational Geometry: An Introduction*. Springer, New York, 1985.

[4] K. Hormann, A. Agathos, "The point in polygon problem for arbitrary polygons," *Computational Geometry: Theory and Applications*, vol. 20(3), pp. 131-144, 2001.

[5] G. Taylor, "Point in polygon test," *Survey Review*, vol.32, pp.479-484,1994.

[6] F. R. Feito, J. C. Torres, A. Urena, "Orientation, simplicity, and inclusion test for planar polygons," *Computers & Graphics*, vol. 19 (4), pp. 595-600,1995.

[7] F. R. Feito, J.C.Torres, "Inclusion test for general polyhedral," *Computers and Graphics*, vol.21(1), pp. 23 – 30, 1997.

[8] B. Zalik, A. Jezernik, K. Rizmanzalik, "Polygon trapezoidation by sets of open trapezoids," *Computers & Graphics*, vol. 27(5), pp. 791-800, 2003.

[9] J. Li, W. C. Wang, E. H. Wu, "Point-in-polygon tests by convex decomposition," *Computers & Graphics*, vol. 31 (4), pp. 636-648, 2007.

[10] W. C. Wang, J. Li, E. H. Wu, "2D point-in-polygon test by classifying edges into layers," *Computers & Graphics*, vol. 29 (3), pp. 427-439, 2005.

[11] B. Zalik, I. Kolingerova, "A cell-based point-in-polygon algorithm suitable for large sets of points". *Computers & Geosciences*, vol. 27(10), pp. 1135-1145, 2001.

[12] S. Yang, J. Yong, J. Sun, H. Gu, J. C. Paul, "A point-in-polygon method based on a quasi-closest point," *Computers & Geosciences,* vol. 36 (2), pp. 205-213, 2010.

[13] J. J. Jiménez, F. R. Feito, R. J. Segura, "A new hierarchical triangle-based point-in-polygon data structure," *Computers & Geosciences*, vol.35 (9), pp.1843-1853, 2009.

[14] J. J. Jiménez, F. R. Feito, R. J. Segura, "Robust and Optimized Algorithms for the Point-in-Polygon Inclusion Test without Pre-processing," *Computer Graphics Forum*, vol.28 (8), pp.2264-2274, 2009.

[15] E. Haines, "Point in Polygon Strategies," in *Graphics Gems IV*, P. Heckbert, Eds. Morgan Kaufmann, 1994, pp.24-46.

[16] B. Zalik, G. Clapworthy, C. Oblonsek, "An efficient code-based voxel-traversing algorithm," *Computer Graphics Forum*, vol.16 (2), pp.119–128, 1997.

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