Visible Points Searching in Large Obstructed Space

Jianqiu Xu, Ralf Hartmut Güting

Database Systems for New Applications, Mathematics and Computer Science
FernUniversität Hagen, Germany
{jianqiu.xu,rhg}@fernuni-hagen.de

Abstract

This paper proposes a new and fast algorithm to find visible points for an arbitrary query location inside a large polygon with obstacles. The procedure first decomposes the polygon into a set of triangles, and then builds a dual graph on triangles. The process of searching visible points is accessing adjacent triangles based on the graph. The time complexity is $O(N)$ in the worst case, improving the existing method which is $O(N + NlogN)$. The efficiency and effectiveness of the proposed method are verified through extensive experiments.

1. Introduction

Visible points searching (VPS) is a basic problem in the area of computational geometry [14] and geographical databases [16]. That is, given a polygon P with a set of obstacles (holes) represented by polygons $\{O_1$, O_2, ..., $O_n\}$, and an arbitrary query location q inside P, VPS returns all vertices from obstacles fulfilling the condition that the line between q and the vertex does not cross any obstacle. Figure 1 shows an example with three obstacles $\{O_1$, O_2, $O_3\}$. All visible points to q are depicted. VPS is widely used in many applications such as route planning and nearest neighbor searching. For example, in urban areas, q represents the location of a pedestrian who searches the nearest bus stop and the obstacles denote nearby buildings and some streets without crossings. In a battlefield, the path for tanks and soldiers might be blocked by mountains, lakes and construction. In the field of robot motion planning, VPS can help the robot find an optimal route from the start location to the end.

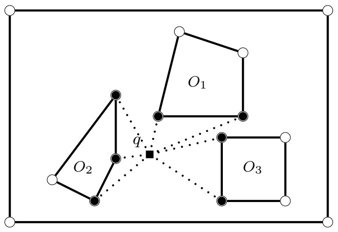


Figure 1: Visible Points Searching

In the literature, VPS has been considered for shortest path searching in the presence of obstacles [13, 9, 8] and visible nearest neighbor searching [11, 6, 5]. However, the results are different from this paper. In the shortest path searching, only a subset of vertices is considered where the methods try to minimize the searching space and the number of visited vertices. A shortcut operation is proposed in [13] to discard portions of the obstacle space when computing the shortest path, and [8] constructs a relevant subgraph based on the overall visibility graph on P. Considering nearest neighbor searching in an obstructed space, the distance between two objects is based on the shortest path between them without crossing any obstacle. Efficient algorithms are proposed to find the results by employing spatial indices such as R-tree to prune the search space [16, 6]. The results are interesting objects instead of obstacle vertices, and distant objects are not considered.

To our knowledge, the only approach that can find all visible points to q is to perform a rotational plane sweep (RPS) algorithm [12]. RPS first sorts all obstacle vertices in counter clockwise order (or clockwise, in the following we use the order counter clockwise) according to q. Afterwards, the vertices are stored in a priority queue according to the angle value. A binary search tree is used for *visible* checking by taking each point from the priority queue. The time complexity for RPS is $O(NlogN + N)$ where N is the total number of obstacle vertices. In detail, $O(NlogN)$ is for sorting and $O(N)$ is for sweep processing. However, the performance decreases for a large dataset and cannot guarantee the efficiency when a large number of vertices (obstacles) are considered, due to the costly sorting procedure and tree operations such as inserting and deleting.

In this paper, we propose a novel and fast algorithm to return all *visible* points for a query location. Initially, we decompose P into a set of triangles and build a dual graph on these triangles where a node corresponds to a triangle and an edge is created if two triangles are *adjacent*. The preprocessing step needs $O(NlogN)$ and is done once before the on-line searching. Afterwards, we locate the triangle that q belongs to, and then the procedure of visible points searching is to access *adjacent* triangles based on the dual graph. The time complexity is $O(N)$ in the worst case. We demonstrate the efficiency of the technique through extensive experiments.

The rest of the paper is organized as follows: Section 2 introduces the framework. The algorithm of VPS is presented in Section 3. We conduct the experiment in Section 4 and conclude the paper in Section 5.

2. The Framework

We let $\{O_1, O_2, ..., O_n\}$ be obstacles inside P and O_0 denote the outer boundary of P. The contour of each O_i consists of a sequence of vertices and we define them in a counterclockwise order. Then, a vertex of P can be represented by v_i = (o_id, id, loc) (o_id, $id \in D_{\underline{int}}$ [1] $\wedge$ $c\_id \geq 0 \wedge id \geq 0$, $loc \in D_{\underline{point}}$) where o_id records the contour id and id defines the vertex order. Figure 2 depicts the polygon in Figure 1 (the point information is omitted) with the proposed vertex representation.

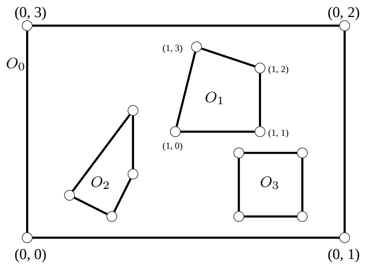


Figure 2: A Polygon with Holes

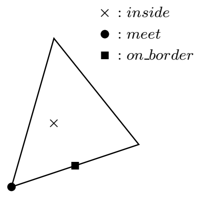


Figure 3: Relationships between q and tri_q

Employing the algorithms from [10, 7], we perform polygon triangulation on P and get a set of triangles Tri each of which is denoted by $tri(v_1, v_2, v_3) \in Tri$. Afterwards, we build a dual graph DG on these triangles

[1]Using the algebraic terminology that for a data type α, its domain or carrier set is denoted as D_α.

where a node represents a triangle and an edge is created for two *adjacent* triangles. Let $(v_i,\ v_j)$ denote a triangle edge. For the query location q, besides the location data we assume its located triangle is also known, denoted by tri_q. Three relationships (shown in Fig. 3) are defined in Def. 2.1 between q and tri_q: (1) *inside*; (2) *meet*; and (3) *on_border*. We give the framework of visible points searching algorithm in Algorithm 1.

Definition 2.1 *Relationships between q and tri_q*

(1) inside: q is inside $tri_q \Leftrightarrow \neg\exists\ (v_i, v_j)$: $i \neq j \wedge (v_i, v_j)$ Contains q.

(2) meet: q meets $tri_q \Leftrightarrow \exists v_i \in tri_q$: $v_i.loc$ = q.

(3) on_border: q is on_border of $tri_q \Leftrightarrow \exists v_i, v_j$: (v_i, v_j) Contains $q \wedge v_i.loc \neq q \wedge v_j.loc \neq q$.

Algorithm 1: VPS(q, DG)

```
1 let $tri_q$ be the triangle for $q$;
2 VP ← ∅;
3 if $q$ is inside $tri_q$ then VP = Inside($q$, DG);
4 if $q$ meets $tri_q$ then VP = Meet($q$, DG);
5 if $q$ is on_border of $tri_q$ then VP = OnBorder($q$, DG);
6 return VP;
```

3. Visible Points Searching

3.1. A Clamp Structure

Before elaborating each sub algorithm, we introduce a structure called *Clamp* to be used for searching visible points. A clamp consists of three points with one being called *apex* and the other two being called *feet*, represented by $cl(a, f_1, f_2)$ (a, f_1, $f_2 \in D_{\underline{point}}$). Two connections are defined among the three points: $\overrightarrow{af_1}$ and $\overrightarrow{af_2}$, see Fig. 4(a). Each connection starts from the *apex* and passes through one of the *feet*. A clamp partitions the space into two parts, $cl.A$ and $\overline{cl.A}$, as shown in Fig. 4(b). We define that $cl.A$ does not include lines $\overrightarrow{af_1}$ and $\overrightarrow{af_2}$. One can calculate the angle of a clamp, that is $\angle f_1af_2$ or $\angle f_2af_1$. We let $cl.\alpha$ denote the angle and define the value to be between $(0, 180)$. Given a clamp and a point q, let β_1=$\angle qaf_1$ and β_2=$\angle qaf_2$ ($\beta_1, \beta_2 \in (0, 180)$) be two angles. We define the *cover* relationship between a clamp and q as below.

Lemma 3.1 *clamp covers q*

A clamp covers q if and only if $cl.\alpha = \beta_1 + \beta_2$. This implies that the line $\overrightarrow{aq}$ is located inside $cl.A$.

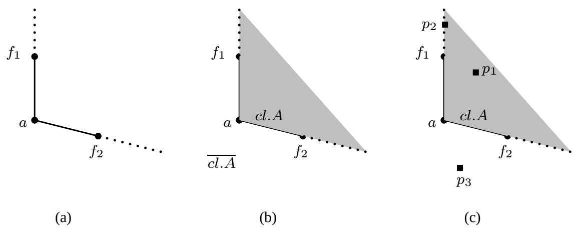


Figure 4: Clamp Structure

We prove that if $\overrightarrow{aq}$ inside $\overline{cl.A}$, the *cover* condition cannot hold.

Proof *Proof by contradiction.*

If $\overrightarrow{aq}$ is inside $\overline{cl.A}$, we have β_2 + β_1 + $cl.\alpha$ = 360. Let β_2 = 360 - (β_1 + $cl.\alpha$). As $cl.\alpha$ = β_1 + β_2 (cover condition), then $cl.\alpha$ = β_1 + 360 - (β_1 + $cl.\alpha$) by replacing β_2. It contradicts. □

Fig. 4(c) depicts an example with three points $\{p_1, p_2, p_3\}$. According to the *cover* condition, the clamp covers p_1 but does not cover p_2 and p_3. The case for p_3 is straightforward. For p_2, as we define $\beta_1, \beta_2 \in (0, 180)$, the *cover* condition does not hold if the three points $\{a, f_1, p_2\}$ are co-linear.

3.2. Inside

If the *inside* condition holds, evidently, all vertices of tri_q are *visible* to q, as shown in Fig. 5. We collect the three points and search more *visible* points inside P if exist. In the following, the notation $O_i.v_j$ (Section 2) is used to denote a vertex of P where i is the contour id and j is the vertex order. We create three clamps by setting q as the *apex* for each case: (1) $cl_1(q, O_2.v_2, O_2.v_3)$; (2) $cl_2(q, O_2.v_2, O_3.v_0)$; (3) $cl_3(q, O_2.v_3, O_3.v_0)$.

The three clamps partition tri_q into three sub triangles. For each sub triangle, we apply the *depth-first* method to search its *adjacent* triangles on DG to find visible points to q. Each sub triangle determines a searching space, and we only tackle the points that are *covered* by the corresponding clamp. Without loss of generality, we take cl_3 as an example to describe the procedure. By searching DG, we find the *adjacent* triangle to $tri(q,\ O_2.v_3,\ O_3.v_0)$, that is $tri(O_1.v_0,\ O_2.v_3,\ O_3.v_0)$. Since the two triangles share two vertices, only $O_1.v_0$ needs to be checked. We denote the point to be checked by p. The following lemma is used to determine whether p is visible to q.

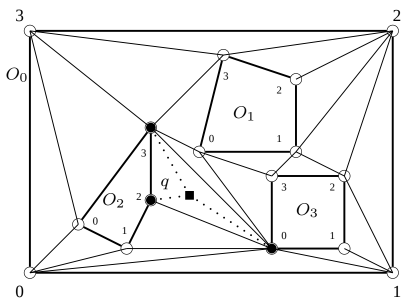


Figure 5: q inside tri_q

```
Algorithm 2: Inside (q, DG)
1 VQ ← ∅;
2 VQ = VQ ∪ {$tri_q.v_1$, $tri_q.v_2$, $tri_q.v_3$};
3 create clamps CL by q and $tri_q$;
4 for each $cl_i \in$ CL do
5     let e($cl_i.f_1$, $cl_i.f_2$) be an edge;
6     VQ = VQ ∪ DepthTraversal(DG, $cl_i$, e);
7 return VQ;
```

Figure 6: the algorithm for the case q inside a triangle

Lemma 3.2 *p is visible to q*

Let $cl(q,\ f_1,\ f_2)$ be the current clamp structure with q being the apex and $Poly_q$ be a polygon with four vertices $\{q, f_1, f_2, p\}$. We say p is visible to $q \Leftrightarrow Poly_p$ is a strictly convex polygon [2].

Note that if q, $f_1(f_2)$ and p are co-linear, p is not visible to q as p is blocked by $f_1(f_2)$. Given a polygon with n vertices, the time complexity for checking convex is $O(n)$. Since $Poly_p$ always has four vertices, the time complexity is $O(1)$. In Fig. 5, $O_1.v_0$ is visible to q and is collected. When p (the checked point) is visible, we split the clamp into two subclamps. For each subclamp, the apex remains the same (q), one foot is the same as before and the other foot is set as p. cl_3 is split into $cl_3^1(q,\ O_1.v_0,\ O_2.v_3)$ and $clp_3^2(q,\ O_1.v_0,\ O_3.v_0)$, shown in Fig. 7(a). For each subclamp, we repeat the same procedure of visiting adjacent triangles to find visible points. Whenever a clamp is split, the angle of a subclamp becomes smaller so that the searching space in the sub procedure is decreased.

Now we consider the case that p is not visible. Considering $cl_3^1(q,\ O_1.v_0,\ O_2.v_3)$, by searching DG we find the triangle $tri(O_1.v_3,\ O_2.v_3,\ O_1.v_0)$ and $O_1.v_3$ is not visible to q. The algorithm checks the edges (p, f_1) and

[2]every internal angle is less than 180 degrees

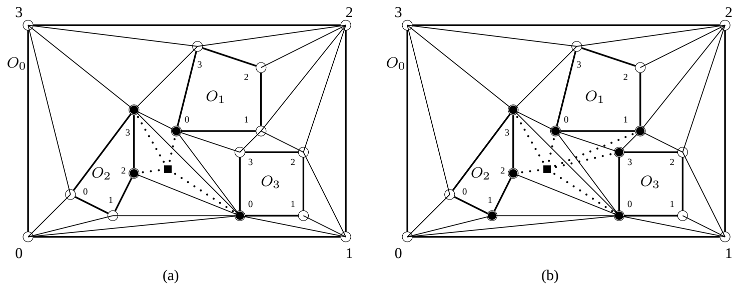


Figure 7: Split the *Clamp*

(p, f_2) to determine whether the procedure of searching *adjacent* triangles should forward or stop. At current state, p (the point to be checked) is $O_1.v_3$, f_1 is $O_2.v_3$ ($O_1.v_0$) and f_2 is $O_1.v_0$ ($O_2.v_3$). The searching stops on the edge if one of the following conditions holds.

- If the edge belongs to the *outer* or *obstacle* contour of P, no *adjacent* triangles can be found.
- If there exists an unvisited *adjacent* triangle to $tri(p, f_1, f_2)$ with the shared edge $(p, f_{1(2)})$, the searching stops when the **internal** angle $\angle q f_{1(2)} p$ in $Poly_q$ is equal or larger than 180. The reason is if $\angle q f_{1(2)} p \geq$ 180, the vertex belonging to the *adjacent* triangle cannot be *covered* by the current clamp (see the *cover* condition in Lemma 3.1).

See Fig. 7(a), $O_1.v_3$ is not visible to q. The searching stops at $(O_1.v_0, O_1.v_3)$ since the edge belongs to the obstacle O_1. The procedure also stops at the edge $(O_2.v_3, O_1.v_3)$ because $cl_3^1(q, O_1.v_0, O_2.v_3)$ does not cover $O_0.v_3$. The vertices of $Poly_q$ are $\{q, O_2.v_3(f_1), O_1.v_0\ (f_2), O_0.v_3\}$ where p is $O_0.v_3$. As a result, the sub procedure for $cl_3^1(q, O_1.v_0, O_2.v_3)$ terminates at $tri(O_1.v_0, O_1.v_3, O_2.v_3)$. For the subclamp $cl_3^2(q, O_1.v_0, O_3.v_0)$, the algorithm finds $tri(O_1.v_0, O_3.v_0, O_3.v_3)$ and checks $O_3.v_3$, which is visible. Then, $cl_3^2(q, O_1.v_0, O_3.v_0)$ is split into two parts and the same procedure is repeated. In the end, the algorithm returns $\{O_3.v_3, O_1.v_1\}$ for $cl_3^2(q, O_1.v_0, O_3.v_0)$, see Fig. 7(b). We summarize the result of $cl_3^1(q, O_1.v_0, O_2.v_3)$ and $cl_3^2(q, O_1.v_0, O_3.v_0)$, and get the visible point set $\{O_1.v_0, O_1.v_1, O_3.v_3\}$ for cl_3. The procedure is the same for cl_1 and cl_2. Notice that, $O_2.v_1$ is visible to q, but it is only found by $cl_2(q, O_2.v_2, O_3.v_0)$ for the reason that cl_1 and cl_3 do not cover $O_2.v_1$. The algorithms are given in Algorithm 2 and Algorithm 3.

Algorithm 3: DepthTraversal (*DG*, cl_i, e)

```
1  VQ ← ∅;
2  let $tri(q, cl_i.f_1, cl_i.f_2)$ denote the created triangle;
3  if exists $tri_i$ in DG that is adjacent to $tri$ on the edge $e$ then
4      let $q = cl_i.a$;
5      get $p$ from $tri_i$;
6      if $p$ is visible to $q$ then
7          VQ = VQ ∪ $p$;
8          split $cl_i$ into $\{cl_i^1, cl_i^2\}$;
9          VQ = VQ ∪ DepthTraversal(DG, $cl_i^1$, $e_1(cl_i^1.f_1, cl_i^1.f_2)$);
10         VQ = VQ ∪ DepthTraversal(DG, $cl_i^2$, $e_2(cl_i^2.f_1, cl_i^2.f_2)$);
11     else
12         if $\angle q f_1 p < 180$ then
13             VQ = VQ ∪ DepthTraversal(DG, $cl_i$, $e_1(cl_i.f_1, p)$);
14         if $\angle q f_2 p < 180$ then
15             VQ = VQ ∪ DepthTraversal(DG, $cl_i$, $e_2(cl_i.f_2, p)$);
16 return VQ;
```

3.3. *Meet*

In this case, q equals to the point of a triangle, i.e., a vertex of P. To determine the searching space, we process as follows. First, all triangles $\{tri_1, tri_i, ..., tri_n\} \subset Tri$ that contain q are collected. This step can be optimized by building an R-tree on Tri as opposed to performing a linear searching. Then, for each tri_i we create a clamp with q being the *apex* and the two triangle vertices (not equal to q) being the feet. In the end, we call the function *DepthTraversal* for each created clamp. In Figure 8, q is located at $O_2.v_2$ and the query point is contained by $tri_1(O_2.v_2, O_2.v_3, O_3.v_0)$ and $tri_2(O_2.v_2, O_2.v_1, O_3.v_0)$. Two *clamps* are created $cl_1(q, O_2.v_3, O_3.v_0)$ and $cl_2(q, O_2.v_1, O_3.v_0)$. The first clamp is split into several parts during searching, demonstrated in the figure. We give the algorithm in Algorithm 4.

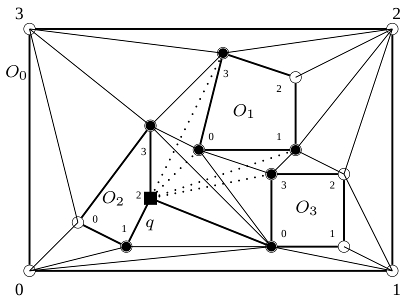


Figure 8: q *meets* a triangle

Algorithm 4: Meet(q, *DG*)

```
1 VQ ← ∅;
2 let $S$ be the set of all triangles containing $q$;
3 for each $tri_i \in S$ do
4     create a $cl$ by $tri_i$ and $q$;
5     let $e(cl.f_1, cl.f_2)$ be an edge;
6     VQ = VQ ∪ DepthTraversal(DG, $cl$, $e$);
7 return VQ;
```

Figure 9: the algorithm for the case q meets a triangle

3.4. *On_Border*

Let (v_i, v_j) denote the edge that q is located on. There are two cases: (1) (v_i, v_j) belongs to one contour of P; (2) (v_i, v_j) does not belong to any contour of P, demonstrated by q_1 and q_2 in Fig. 10. In the first case, two clamps have to be created to partition tri_q into two parts. We have cl_1 = $(q_1, O_3.v_2, O_0.v_2)$, and cl_2 = $(q_1, O_3.v_2, O_0.v_1)$. In the second case, the edge (v_i, v_j) is shared by two triangles both of which are considered to partition the searching space. Each triangle is split into two parts, resulting in four clamps. In this example, we have cl_1 = $(q_2, O_2.v_0, O_2.v_3)$, cl_2 = $(q_2, O_0.v_3, O_2.v_0)$, cl_3 = $(q_2, O_0.v_3, O_1.v_3)$, and cl_4 = $(q_2, O_1.v_3, O_2.v_3)$. The algorithm is given in Algorithm 5.

3.5. *Time Complexity*

The algorithm VPS visits triangles from P to find visible vertices to q. As a consequence, the complexity depends on the number of triangles after the decomposition of P. Let N (≥ 3) be the total number of vertices in P including the outer contour and obstacles, and H be the quantity of obstacles. We use T to denote the number of triangles after polygon triangulation, calculated by a well-known formula (1) T = N+2*H-2.

Now, we have to set the value of H for a polygon with N vertices. The lower and upper bounds can be determined, represented by (2) $H \in$ [0, (N-3)/3]. The lower bound shows a polygon without holes. The upper bound indicates that P has three vertices for the outer contour and all the other vertices are for holes. Combining (1) and (2), we can get the range of $T \in$ [N-2, 5N/3-4]. In the worst case, the algorithm has to visit all triangles, resulting in the time complexity $O(5N/3 - 4)$ = $O(N)$.

Notice that, before running the VPS algorithm two preprocessing steps are needed: (1) polygon triangulation; and (2) building a dual graph. The time complexity for both of them is $O(NlogN)$. This procedure is only done once, and therefore it is not included in the on-line searching.

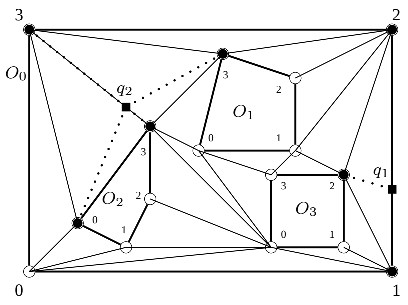


Figure 10: q is *on_border* of a triangle

```
Algorithm 5: OnBorder(q, DG)
1  VQ ← ∅;
2  let $(v_i, v_j)$ be the edge where $q$ is located;
3  if $(v_i, v_j)$ belongs to one contour of $P$ then
4      create $cl_1$ and $cl_2$ by $q$ and $tri_q$;
5      VQ = VQ ∪ DepthTraversal(DG, $cl_1$, $e_1(cl_1.f_1$, $cl_1.f_2)$);
6      VQ = VQ ∪ DepthTraversal(DG, $cl_2$, $e_2(cl_2.f_1$, $cl_2.f_2)$);
7  else
8      let $C$ be the set of created clamps;
9      for each $cl_i \in C$ do
10         let $e(cl_i.f_1, cl_i.f_2)$ be an edge;
11         VQ = VQ ∪ DepthTraversal(DG, $cl_i$, $e$);
12 return VQ;
```

Figure 11: the algorithm for the case q is on_border of a triangle

4. Experimental Evaluation

4.1. Setup and Datasets

We perform the experimental evaluation in this section. The implementation is developed in an extensible database system SECONDO [2] and programmed in C/C++. A standard PC (AMD 3.0 GHz, 4 GB memory, 2TB disk) running Suse Linux (kernel version 2.6.34) is used. The tool MWGen [15] is used to create a relatively large polygon representing the overall walking area for a city. The program takes a set of roads as input and creates pavements and zebra crossings. Two road datasets are used, Berlin [3] and Houston [4]. We show the polygon and graph data in Figure 12. A website [1] is available for providing materials for experiments including datasets, scripts and screenshots of the data.

	Berlin	Houston
X Range	[0, 44411]	[0, 133573]
Y Range	[0, 34781]	[0, 163280]
No. Vertices in P	116,516	437,279
No. Obstacles	14,557	10,171

(a) Polygon

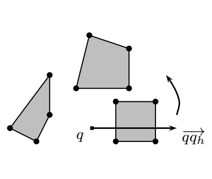
Berlin		Houston	
Nodes	Edges	Nodes	Edges
145,387	159,943	458,810	469,550

(b) Dual Graph

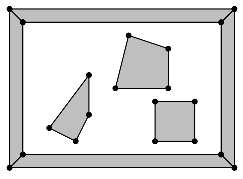
Figure 12: Datasets Statistics

4.2. Competitor's Implementation

Before the evaluation, we present the implementation of the RPS algorithm. Let $\overrightarrow{qq_h}$ be the starting horizontal sweep line ($q_h.x$ is larger than the axis value of q and all obstacle vertices). Initially, the algorithm sorts all obstacle vertices, each of which is denoted by v_i, in counter clockwise according to q (i.e., rotating from $\overrightarrow{qq_h}$ to $\overrightarrow{qv_i}$). After sorting, the vertices are stored in a priority queue according to the angle value. A binary search tree Tr is used for *visible* checking. Each node in Tr stores the segment of an obstacle and the key value is set as the distance between q and the segment. At the beginning, Tr stores all segments that intersect $\overrightarrow{qp_h}$. For each v_i popped from the queue, three operations are performed: (1) *visible* checking; (2) inserts segment v_iv_j into Tr where rotating from qv_i to qv_j is in counter clockwise order; (3) deletes segment v_iv_k from Tr where rotating from qv_i to qv_k is in clockwise order. If v_i is *visible* to q, this means that $\overrightarrow{qv_i}$ neither passes through any obstacles nor contains any obstacle vertices (excluding v_i). Let $min(Tr)$ be the smallest key in Tr and there



(a)



(b)

Figure 13: Rotational Plane Sweep

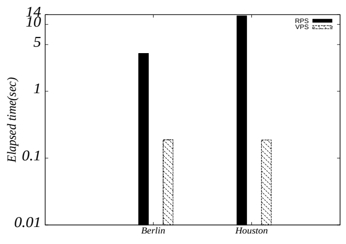
are two cases that v_i is *visible*: (i) $dist(q, v_i) \leq min(Tr)$ and (ii) no obstacle segment in Tr intersects $\overrightarrow{qv_i}$ or the intersection point is q or v_i. Figure 13(a) shows an example.

Although the original RPS method does not have the *outer* boundary covering all obstacles, the algorithm can be adapted as follows: For the points inside P, RPS can be directly applied. While for points located on the boundary, a bounding box can be created containing the *outer* boundary as a region inside and the space between the bounding box and the *outer* boundary is treated as obstacle areas, see Figure 13(b).

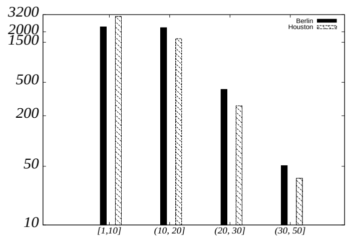
4.3. Evaluation Results

In this part, we evaluate the procedure of searching visible points and compare the performance between VPS and RPS by using the polygon data of Berlin and Houston. In each city, 5,000 random points inside P are generated to be the set of query locations. We run both algorithms to find visible vertices for each query point and measure the execution time. The final result is the average value over all runnings where the time measurements are plotted in logarithmic scale. To have a fair comparison, both algorithms have the same 5,000 query points. The complexity of RPS is $O(NlogN + N)$ for each case, while our algorithm is $O(N)$ in the worst case. The experimental results confirm the efficiency of our algorithm where VPS achieves about an order of magnitude performance improvement, shown in Figure 14(a).

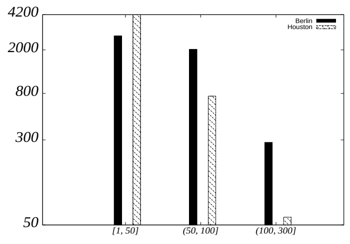
Besides the time cost, for each query location we also record the quantities of visible points and accessed triangles. Figure 14(b) and 14(c) show the distribution of the results. The value of x-dimension means the number of visible points (accessed triangles) and the y-dimension shows the number of query locations that have such results. In both datasets, the number of visible points is less than 20 for 90% of the query locations. In Section 3.5, we analyzed that in the worst case all triangles have to be accessed, but practically the number of visited triangles is less than 100 in most cases (94.3% for Berlin, 98.8% for Houston). The precise values are reported in Figure 15.



(a) Time Cost



(b) Distribution for the Number of Visible Points



(c) Distribution for the Number of Visited Triangles

Figure 14: Results1

	RPS	VPS
Berlin	3.68	0.188
Houston	13.42	0.187

(a) Time (sec)

	[1, 10]	(10, 20]	(20, 30]	(30, 50]
Berlin	2,294	2,242	413	51
Houston	3,044	1,657	263	36

(b) Statistics for Visible Points

	[1, 50]	(50, 100]	(100, 300]
Berlin	2,690	2,025	285
Houston	4,187	754	59

(c) Statistics for Visited Triangles

Figure 15: Results2

5. Conclusions

We study the problem of searching visible points in a large polygon with obstacles. Based on the result of polygon triangulation, we manage the obstructed space by a set of triangles and build a dual graph on them. We search visible points by accessing adjacent triangles. An extensive experimental study is conducted and the results confirm the superiority of the technique over the existing method.

[1] http://dna.fernuni-hagen.de/secondo.html /transportationmode.html.

[2] http://dna.fernuni-hagen.de/secondo.html/index.html.

[3] http://www.bbbike.de/cgi-bin/bbbike.cgi (2012.6.25).

[4] http://www.census.gov/geo/www/tiger/tgrshp2010 /tgrshp2010.html (2012.6.25).

[5] Y. Gao and B. Zheng. Continuous obstructed nearest neighbor queries in spatial databases. In *SIGMOD*, pages 577–590, 2009.

[6] Y. Gao, B. Zheng, W. Lee, and G. Chen. Continuous visible nearest neighbor queries. In *EDBT*, pages 144–155, 2009.

[7] M. Held. Fist:fast industrial-strength triangulation of polygons. *Algorithmica*, 30(4):563–596, 2001.

[8] S. Kapoor, S.N. Maheshwari, and J.S.B. Mitchell. An efficient algorithm for euclidean shortest paths among polygonal obstacles in the plane. *Discrete Comput. Geom.*, 18:377–383, 1997.

[9] J.S.B. Mitchell. Shortest paths among obstacles in the plane. *Internet Journal Comput. Geom.*, 6:309–332, 1996.

[10] A. Narkhede and D. Manocha. *Fast polygon triangulation based on Seidel's algorithm*. Graphics Gems V, Academic Press, 1995.

[11] S. Nutanong, E. Tanin, and R. Zhang. Visible nearest neighbor queries. In *DASFAA*, pages 876–883, 2007.

[12] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15(1):193–215, 1986.

[13] J.A. Storer and J.H. Reif. Shortest paths in the plane with polygonal obstacles. *Journal of the ACM*, 41(5):982–1012, 1994.

[14] T.Asano, S.K.Ghosh, and T.C.Shermer. *Visibility in the plane*. Handbook of Computation Geometry, Elsevier, 2000.

[15] J. Xu and R. H. Güting. MWGen: A Mini World Generator. In *MDM, To Appear*, 2012. Preliminary version: http://dna.fernuni-hagen.de/articles2012.html.

[16] J. Zhang, D. Papadias, K. Mouratidis, and M. Zhu. Query processing in spatial databases containing obstacles. *International Journal of Geographical Information Science*, 19(10):1091–1111, 2005.