Journal for Geometry and Graphics
Volume 13 (2009), No. 1, 1–16.

A Practical Approach for Planar Visibility Maintenance*

Alireza Zarei, Mohammad Ghodsi

Computer Engineering Department, Sharif University of Technology
IPM School of Computer Science, Tehran, Iran
email: zarei@mehr.sharif.edu, ghodsi@sharif.edu

Abstract. In this paper, we propose a method for maintaining the region visible from a moving point observer inside a planar scene. In this method, we check the observer position in discrete time-stamps to detect and apply changes to the visible or illuminated region of a moving point observer q, or $VP(q)$. We efficiently maintain a list $C(q)$ of edges in $VP(q)$ which are subject to change during the motion. In each time-stamp that $VP(q)$ is to be updated, we only refine and redraw the view against the edges of $C(q)$ that indicate the positions of the visibility changes. We build an enriched representation of the visibility graph in a preprocessing step to apply the required updates on $C(q)$ efficiently and ready to be used in the next time-stamp. Using these structures, the exact visible regions are updated in each time-stamp in $O(|C(q)|)$ for sufficiently small values of time-stamp intervals. This is the best possible and superior to the current solutions. Although the time-stamp intervals are small enough in real applications, our method will still remain superior even if the intervals were relatively long in cases with high-speed observer or in dense scenes. The results of our implementation prove efficiency of our method in practice.

Key Words: Computational geometry, exact visibility maintenance, moving observer, planar polygonal scene, visibility polygon

MSC 2000: 68U05, 65D18

1. Introduction

In many application areas like computer graphics, computer games, machine vision, robotics, and motion planning, we are asked to draw the illuminated region from a light source. Equivalently, we need to determine the view of an observer inside a two or three dimensional scene. There are many variants of this problem and different solutions have been proposed. In this

* This research was in part supported by a grant from IPM (No. CS1386-2-01).

ISSN 1433-8157/$ 2.50

paper, we only consider the problem of maintaining the view from a moving point observer inside a 2D scene.

A 2D scene, also known as a polygonal domain or a polygon with holes, is defined by a set of nonintersecting planar objects inside a simple polygon. For simplicity, we assume that the objects are represented by some simple polygons. These objects, called holes, act as occluders that our observer can not see through. The complexity of the scene, n, is the number of the vertices of its border and holes.

A point observer q in such an environment sees a point p if pq does not intersect the boundary and the scene objects. The set of these visible points, composed as a star-shaped polygon, is the visibility polygon of q and is denoted by $VP(q)$. The observer moves in any direction with different speeds. We assume that the observer does not intersect the outer boundary of the scene or edges of the holes during its motion. Our problem is to maintain and update $VP(q)$ of such an observer while it moves inside the scene. We assume that there is a sequence of discrete time-stamps at which (when it arrives) we get a new position of the observer and our algorithm must draw or compute the exact view of the observer for that pair of location and time.

This problem has been considered thoroughly before in two separate theoretical and practical approaches. The main theoretical solutions are based on notions including visibility decomposition, visibility complex, tangent visibility graph, kinetic data structures, topological map and radial subdivision [1, 9, 14, 19, 23, 18, 10, 5, 22]. These methods usually have a preprocessing phase to prepare useful information about the visibility coherence of the scene. Cost of this phase varies in different solutions from $O(n^3 \log n)$ time and $O(n^3)$ space to $O(n \log n)$ time and $O(n)$ space. The prepared information of this phase is then used to efficiently find the initial $VP(q)$ and to update it as the observer q moves along. In the efficient methods, the initial $VP(q)$ is computed in $\Omega(|\,VP(q)|\log n)$ time. During the motion, a queue of $O(n)$ events is built by which each visibility change is handled in time $O(\log n)$ in order to update and maintain $VP(q)$. However, in about all these methods, some of the processed events are unnecessary and do not affect $VP(q)$.

The main deficiency of these results is that only the combinatorial structure of $VP(q)$ is maintained. That is, only the changes to this structure are handled. What is maintained for $VP(q)$ is a sequence of vertices and edges of the scene which are (partially) visible to q. Therefore, if we have to draw the exact visible portion of each edge of this sequence, as needed in all real applications, we must do a linear trace over this sequence and compute and draw the exact visibility polygon. This extra process should be done in each time-stamp that we want to draw the view.

This along with the implementation complexity prevent application developers to use these methods in practice. Therefore, several simple but inefficient methods are usually used in real and practical implementations in which the view is re-computed and re-drawn from scratch at each time-stamp and some hardware or cache mechanisms are used to improve the efficiency.

In this paper, we propose a theoretical method that computes the exact data of $VP(q)$ instead of its combinatorial structure. Despite current theoretical solutions, this method, like the practical approaches, produces the exact visibility polygon at each time-stamp and it is theoretically efficient as well.

To achieve this goal, we maintain a list $C(q)$ of the edges of $VP(q)$ that are not completely visible from q. This list defines the edges from the exact view of q that face changes as q moves. For example, for observer q in Fig. 1, $C(q) = \langle e_1, e_2 \rangle$. At the next time-stamp that

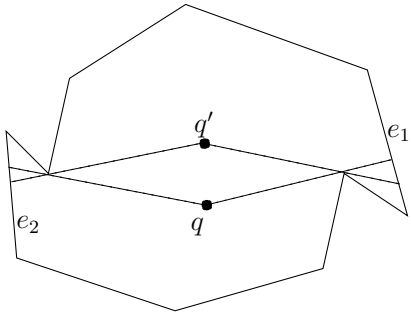

Figure 1: When the observer moves from point q to point q', its visibility polygon is changed only near edges e_1 and e_2

observer lies at point q', the exact view are drawn by just determining the new visible portions of the edges e_1 and e_2. Such updates that do not alter the combinatorial structure of $VP(q)$, occur most often in real applications. Hence, the application developers can simply use our method to have the exact $VP(q)$ at each time-stamp.

As q moves further, some edges may be eliminated from $C(q)$ and new edges must be inserted into this list. To do these updates efficiently, we build an enriched version of the visibility graph data structure of the scene. This data structure is built in the preprocessing phase of our algorithm containing the required data in order to detect and handle the updates in constant time. The size of this structure is equal to the size of the original visibility graph which is something between $O(n)$ and $O(n^2)$. Using this method, updates of $VP(q)$, as the observer q moves to a new location, is done in linear time in terms of the number of the required changes of the exact visibility. Trivially, this is equal to $|C(q)|$ whenever the time interval between the consecutive time-stamps is small. Therefore, the update time is normally low for small observer movement.

Comparing to current theoretical methods, the best current algorithm maintains $VP(q)$ as a sequence of polygon edges visible from q, and updates on $VP(q)$ are done in $O(\log(n))$. But, in these methods, the computation of the exact visibility region, needed by the practical applications, requires another trace of $VP(q)$ which is done in $O(|\,VP(q)|)$. While $C(q)$ is a subset of $VP(q)$, the efficiency of our method is always superior. Specifically, assume that q moves along a given line segment st and during this movement, there are k time-stamp intervals, and $VP(q)$ changes combinatorially l times. The time complexity of the best current methods is $O(k\overline{|\,VP(q)|} + l \log n)$ where $\overline{|\,VP(q)|}$ is assumed to be the average size of $VP(q)$ along st. In contrast, after the first initialization of $VP(q)$, the total processing time of our method, when q moves along st, is $O(k\overline{|C(q)|} + l)$ where $\overline{|C(q)|}$ is the average size of $C(q)$ along st. Also, the processing time of our method is completely output-sensitive and does not depend on the size of the scene. Moreover, if the observer changes its movement direction, the current methods require $O(n \log n)$ processing time to rebuild their event queue, while our method does not require such extra processing times.

Our method is simple for implementation and our experimental results verify its efficiency and applicability for scenes containing many objects inside.

The rest of this paper is organized as follows: we describe the preprocessing phase of the algorithm and our representation of the visibility graph in Section 2. The new method is described and analyzed in Sections 3 and 4. Our experimental results are shown in Section 5 and the materials are summarized and concluded in Section 6.

2. The preprocessing phase

Our algorithm is composed of two main phases: preprocessing and running. In the preprocessing phase, a set of useful information is gathered to facilitate the next phase. In this section, we describe the structures prepared in the preprocessing phase of the algorithm and we analyze the time and space complexity of this phase.

Our method, uses an existing algorithm to compute the initial $VP(q)$. Any one of the methods in [22, 8, 17, 20, 2] can be used here. We use the method of [2] which efficiently computes $VP(q)$ for a single point observer q. This method does not require a preprocessing and computes $VP(q)$ in $O(n \log n)$ time.

Data structures are needed to predict and determine the visibility events during the motion of the observer. We use the visibility graph for this purpose. The visibility graph of a scene S, denoted by $VG(S)$, is a graph that contains a vertex for each vertex of the scene and an edge between two vertices if their corresponding vertices see each other. For a scene S of total n vertices, size of $VG(S)$ is something between $O(n)$ and $O(n^2)$ depending on the number of the edges in this graph.

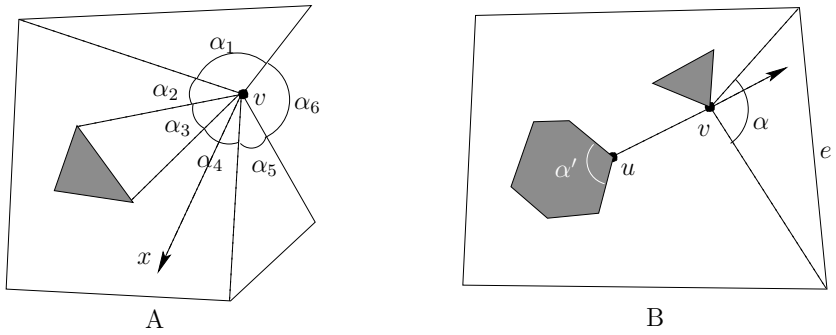

Figure 2: Special representation of the visibility graph

This definition of the visibility graph does not answer all requirements of our algorithm. We need a special representation for this graph by which a visibility event can be handled in constant time. For this purpose, the incident edges of each vertex are assumed to be sorted counter-clockwise such that these radial edges subdivide the unit circle around each vertex into several ranges. Having these sorted ranges, we can navigate from one range to an adjacent one in constant time. This sorted list of ranges for a vertex v is denoted by R_v and the range containing a ray $\overrightarrow{vx}$ is denoted by $R_v(\overrightarrow{vx})$. For example, R_v and $R_v(\overrightarrow{vx})$ of the vertex v shown in Fig. 2A are respectively equal to $\langle \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6 \rangle$ and α_4.

Another requirement in our representation of the visibility graph is to determine the ranges from R_v and R_u through which the supporting line of the edge uv of the visibility graph passes from its both end points. We refer to these ranges by $R_v(\overrightarrow{uv})$ and $R_u(\overrightarrow{vu})$. For example, $R_v(\overrightarrow{uv})$ and $R_u(\overrightarrow{vu})$ in Fig. 2B are respectively equal to α and α'. Using the ordered lists of R_u and R_v, these ranges can be computed in logarithmic time. We must determine these ranges in constant time during the motion. We thus compute $R_v(\overrightarrow{uv})$ and $R_u(\overrightarrow{vu})$ for each edge uv of the visibility graph in the preprocessing phase and maintain them in our representation of the visibility graph.

Moreover, for any edge of the visibility graph, we must be able to determine in constant time the first edges of the scene intersected by the supporting line of this segment from its both endpoints. We denote these intersected edges by $e_{\overrightarrow{uv}}$ and $e_{\overrightarrow{vu}}$. For example, $e_{\overrightarrow{uv}}$ in Fig. 2B is equal to e and while $\overrightarrow{vu}$ passes through the outside of the scene after vertex u,

$e_{\overrightarrow{vu}}$ is set to empty. We compute $e_{\overrightarrow{uv}}$ and $e_{\overrightarrow{vu}}$ for each edge uv of the visibility graph in the preprocessing phase and maintain them in our representation of the visibility graph to be accessed in constant time during the motion.

Finally, to facilitate navigation between these structures, we create direct links between edges and vertices of the visibility graph and their corresponding vertices and edges of the scene and their adjacent ranges defined above.

We call this version of the visibility graph as *enriched visibility graph* and denote it by $EVG(S)$ for a scene S. As described above, we add constant data items to each edge of the visibility graph. Hence,

Lemma 1 *$EVG(S)$ and $VG(S)$ data structures have equivalent space complexity.*

Therefore, the size of the required space of the preprocessing phase of our method for a scene of total n vertices is something between $O(n)$ and $O(n^2)$.

We must compute and embed our extra data into the known visibility graph data structure. Before describing these details, we introduce the current algorithms of computing the visibility graph of a planar scene. The visibility graph problem itself has long been studied and has been applied to a variety of areas and problems like motion planning, shortest path and art gallery problems. D.T. Lee proposed the first nontrivial solution to this problem running in $O(n^2 \log n)$ time [13]. Then, a series of $O(n^2)$ time algorithms proposed by E. Welzl [21], Asano et. al. [3], Edelsbrunner and Guibas [4] and Overmars and Welzl [15] which require $O(n^2)$ [21, 3], or $O(n)$ [4, 15] working space. Finally, several output-sensitive algorithms proposed by Overmars and Welzl [15], Ghosh and Mount [6, 7], Kapoor and Maheshwari [11, 12], Pocchiola and Vegter [16] and Riviére [19] which run in $O(e \log n)$ [15] or $O(e + n \log n)$ [6, 7, 11, 12, 16, 19] where e is the number of the edges in the visibility graph.

We can pick one of the simple algorithms from the above-mentioned to build the visibility graph. For example, if we use the method by D.T. Lee [13], we can sort the adjacent edges of each vertex in total $O(|e| \log n)$ time. Having the sorted list R_v for all vertices v, we can build the other $R_v(\overrightarrow{uv})$ and $e_{\overrightarrow{uv}}$ data for all vertices and edges in $O(e+n)$ time to be described later. Therefore, the total time complexity of the preprocessing phase will be $O(n^2 \log n)$ and it can simply be implemented for practical usage.

To be efficient, we use the algorithm proposed by Kapoor and Maheshwari [12] to build and prepare our extension of the visibility graph. In this method, the adjacent edges of a vertex v are computed and reported sorted by their angular order around v (Lemma 4.2 in [12]). Having this sorted list, we propose a method to compute $R_v(\overrightarrow{vu})$ data items for all edges vu of the visibility graph adjacent to a vertex v by a linear trace on this list.

Assume that $E_v = \langle vu_1, vu_2, \ldots, vu_k \rangle$ is the list of adjacent edges for vertex v in $VG(S)$ computed by the algorithm proposed in [12]. The order of edges of E_v corresponds to their counter-clockwise order around v. We define a list $R_v = \langle \alpha_1, \alpha_2, \ldots, \alpha_k \rangle$ where α_i, $(1 \leq i < k)$ is the counter-clockwise angle defined by edges vu_i and vu_{i+1} and α_k is this angle for vu_k and vu_1. Having these two lists, we can compute $R_v(\overrightarrow{u_iv})$, $1 \geq i \leq k$, as follows. We first find $\alpha_i = R_v(\overrightarrow{u_1v})$ by a naive linear trace or an efficient binary search on R_v. Assume that we have a new list $R'_v = \langle \alpha_i, \alpha_{i-1}, \ldots, \alpha_1, \alpha_k, \alpha_{k-1}, \ldots, \alpha_{i+1} \rangle$. Then, we can simultaneously trace R'_v and E_v lists to find $R_v(\overrightarrow{u_iv})$ of all vu_i edges of E_v.

The $e_{\overrightarrow{uv}}$ data can be computed by applying small changes to the base algorithm in [12]. We change this algorithm to maintain the segment e_i which is visible from each vertex v through any two neighboring edges vu_i and vu_{i+1} of its edges in VG, i.e., for any two edges

vu and vu' of the visibility graph which are neighbors in counter-clockwise ordered list of the adjacent edges of v, we maintain the segment e which is visible to v through the wedge defined by vu and vu'. These information can be computed without increasing the time or space complexities of the base algorithm. We omit the details here since it needs the details of [12]. Having this information, $e_{\overrightarrow{uv}}$ is exactly the segment e which is visible from v through the edges of the range $R_v(\overrightarrow{uv})$.

In this method, we process each edge of the visibility graph a constant number of times and therefore,

Lemma 2 *We can build the enriched visibility graph in $O(e + n \log n)$ preprocessing time.*

Summarizing the discussion of this section we have the following theorem:

Theorem 1 *The size of $EVG(S)$ data structure of a scene S of total n vertices is $O(e + n)$ and it can be constructed in $O(e + n \log n)$ time where e is the number of the edges of $VG(S)$ which is at most n^2.*

Therefore, the space and time complexity of our preprocessing phase is $O(e + n)$ and $O(e + n \log n)$, respectively.

3. Maintaining exact visibility

In existing theoretical solutions, the combinatorial structure of $VP(q)$ is maintained during the motion of the observer. This structure, which is the sequence of the visible edges and vertices from the observer q, is changed in discrete time-stamps. These changes occur whenever a new edge or vertex is added to this sequence or an edge or vertex is removed from it. For example, when the observer in Fig. 1 moves from point q to point q', the combinatorial structure of its visibility polygon does not change. However, when we are considering the exact border of the visibility polygon, we must determine the exact visible portion of edges e_1 and e_2 in this figure. Therefore, if we are asked to maintain or draw the exact visibility polygon at each time-stamp, it is sufficient to identify such edges and compute their visible portions. Then, online drawing or illumination of $VP(q)$ can be done by only updating the changed areas around these edges.

It is easy to show that these changing areas always belong to some reflex vertices visible from q. The vertex v in Fig. 3 is an example of such vertices. In this figure, only the upper portion of edge e which is determined by the supporting line of qv, is visible from q. When q moves, this visible portion, based on the motion direction, is increased or decreased. The region A in Fig. 3 is called a *cave* and v, $\overrightarrow{l}$ and e are its vertex, window, and edge, respectively. A cave c is specified by its vertex, window and edge which are respectively denoted by v_c, w_c and e_c.

The idea of our algorithm for online maintenance of the exact $VP(q)$ is to efficiently determine and maintain the visible portion of the cave edges during the observer motion. In each time-stamp, the actual visibility polygon is computed by finding the visible portion of each cave edge which depends on the position of the observer in that time. The initial value of $VP(q)$ for the initial location of the observer q is computed using one of the current methods. For this initial view, the list of its caves is determined and maintained in a sorted list denoted by $C(q)$ in which the caves are stored according to their appearance order in $VP(q)$. This list specifies the parts of $VP(q)$ which must be updated as q moves around. In the following subsections, we first describe how to initialize $C(q)$ and then, describe the possible changes

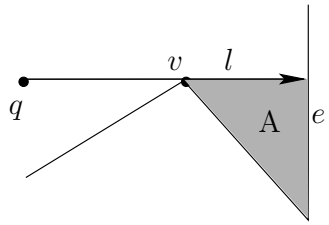

Figure 3: When q moves, only the visible portion of e in $VP(q)$ is changed. A is a cave and v, l and e are respectively its vertex, window and edge.

(events) of this list during the motion. In the next section, we describe the event handling methods for these changes.

3.1. Initialization step of the algorithm

Before starting the motion, $VP(q)$, the initial view of the observer q, is computed for the initial placement according to the method described in Section 2. According to this method, $VP(q)$ is the ordered sequence of edges and vertices which are visible from q. We prepare $C(q)$ as follows. For any one of the reflex vertices in $VP(q)$ there is a cave entry in $C(q)$. These caves are determined by a linear trace of $VP(q)$ and identifying its reflex vertices. For each reflex vertex v_i, its corresponding cave c_i is specified by its vertex, edge and window which are respectively equal to v_i, e_i and $\overrightarrow{v_i v_i'}$. Here, e_i is the edge next to or prior to v_i in the ordered list of $VP(q)$ that is intersected by the supporting line of qv_i, and, v_i' is this intersection point. Having $VP(q)$, the associated values of each cave can be computed in constant time.

Moreover, for each cave window w_c, we compute and maintain $R_{v_c}(w_c)$ which is the range of v_c containing w_c. This can be done in $\log(|R_{v_c}|)$ time by a binary search on R_{v_c}.

Lemma 3 *The time complexity of the initialization step of our algorithm is* $O(|\,VP(q)|\log n)$.

Proof: We can compute $VP(q)$ in $O(|\,VP(q)|\log n)$ time using the solution described in Section 2. According to the above discussions, $C(q)$ is obtained in $O(|\,VP(q)|)$ and the containing ranges of their windows are computed in $O(|C(q)|\log n)$ running time. □

3.2. Continuous update of the view

The initial view must be updated during the motion. We do not have any information about the direction and velocity of the motion and in discrete time-stamps we must redraw the view according to the position of the observer in that moment. This is done by continuously updating caves while the observer moves. At each time-stamp, we first update the cave list and then the view is updated according to the new list of caves and the position of the observer. For each cave c in $C(q)$, the intersection point of e_c and the supporting line of qv_c is computed. This point is an end point of the visible segment of the edge e_c in that time-stamp. Therefore, in each time-stamp at which we need to compute the exact view, this intersection computation must be done for all caves in $C(q)$.

This type of updates is valid while the list $C(q)$ is valid. After its initialization, $C(q)$ is valid only for a while and depending on the movement of q, some new caves must be added to it or some caves must be removed from. Hence, the correctness of this method depends on maintaining the cave list correctly. We call the events which change $C(q)$ *cave events*. There are three types of cave events which are shown in Figs. 4 and 5.

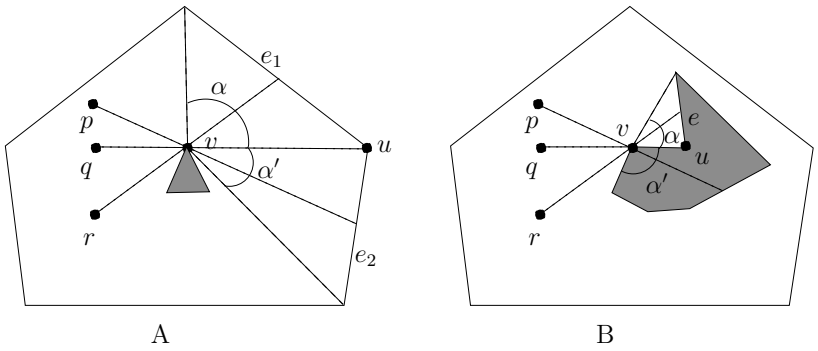

Figure 4: Cave events: in part A(B), when the observer moves from p to r or from r to p, a transform (add-remove) event occurs at point q

The first type of these events, called *transform*, is shown in Fig. 4A. In this figure, when the observer moves from point p to r, before arriving to point q, e_2 is the edge of a cave and after passing q, the edge e_1 will be the edge of that cave. When the observer moves from r to p a similar event happens. In a transform event, a new edge appears in $VP(q)$ and will be the edge of a cave, or, a cave edge disappears from $VP(q)$ and its adjacent edge will be the cave edge.

The second type of cave events, which is shown in Fig. 4B, is called *add-remove* event type. According to this figure, when the observer moves from point p to r, before arriving to the point q, there is no *effective* cave assigned to the reflex vertex v. A cave is effective if its window lies inside the scene. After passing the point q and in point r, there is an effective cave on this vertex. Therefore, on point q the new cave must be added to $C(q)$ as an effective cave. This scenario happens in reverse when the observer moves from point r to point p for which on point q, the cave with vertex v must be specified as an ineffective cave.

The third type of cave events is called *split-merge*. In these events, two caves are merged into a single one or a cave is split into two distinct caves. Figure 5 shows these situations. When the observer lies on point p, it has two caves with vertices v and u. As it moves, on point q, these caves will be merged and, then, on point r only the cave with vertex v remains and the other one is eliminated. On the other hand, when the observer moves from r to p this scenario happens in reverse order. The observer first has a single cave in point r, and in point q, this cave is split into two distinct caves.

Therefore, in a split-merge event, depending on the direction of the observer motion, a

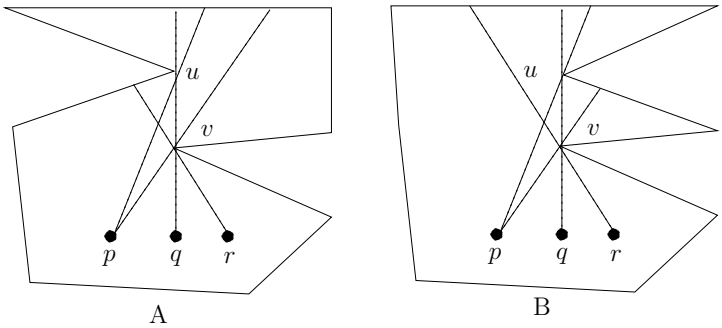

Figure 5: Cave events: when the observer moves from p to r or from r to p, a split-merge event occurs at point q

cave deletion or insertion occurs and its related changes must be handled.

In order to prove the correctness of this method, we must prove that it is enough to identify and handle the cave events.

Theorem 2 *Assuming that we can correctly identify and handle the three types of cave events, the combinatorial structure of VP(q) remains valid during the motion.*

Proof: We prove this by showing that any single combinatorial change of $VP(q)$ corresponds to a single cave event. The combinatorial structure of $VP(q)$ is changed whenever q passes over an edge e of the extended visibility graph. Then, a new vertex and edge will be added to $VP(q)$, or an edge and a vertex must be removed from. Assume that e belongs to vertices u and v, and u is the newly added or removed vertex. Also, assume that just before passing over e, we have the correct list for $C(q)$. Trivially, v is a reflex vertex and just before passing over e, we have an effective of ineffective cave associated to v in $C(q)$, and after passing over e, the edge of this cave is changed. Simple arguments show that these changes belong to a single cave event. Therefore, while q moves, if we detect and handle these cave events according to their happening order, $VP(q)$ will be updated correctly. □

In the next section we describe how to detect and handle the cave events.

4. Handling visibility events

Despite current algorithms, we do not have an event queue in our algorithm. Instead, we identify and handle the occurred events in discrete time-stamps. At each time-stamp, we process the current cave list against the new position of the observer. Tracing this list, we detect the happened changes (events) during this time slice (from previous time-stamp to the current one). For any cave, it is possible to detect several events which must be handled according to their happening order. Therefore, during this linear trace on the cave list, each cave is checked and its associated cave event(if there is any) is handled, and this is repeated until there was no such event on this cave.

For any one of these events, we apply the required changes to $VP(q)$ and $C(q)$ according to the following subsections. Doing this linear trace, we have the correct $VP(q)$ and $C(q)$ for the current time-stamp and we can draw the correct view.

Assume that t is the current time-stamp and c is a cave in this time-stamp and t' is the previous time-stamp. In the following subsections we refer to the edge and window of this cave in time-stamp t' by e'_c and w'_c, respectively.

4.1. Handling transform events

A transform event happens on a cave c whenever w_c passes over one of the edges of $R_{v_c}(w'_c)$, named vu, and $e_{\overrightarrow{vu}}$ is empty. In any time-stamp, such an event on cave c is detected by comparing the new direction of w_c, based on the current position of the observer, and edges of $R_{v_c}(w'_c)$ for the previous direction of w'_c in prior time-stamp. At the new time-stamp, we have $R_{v_c}(w'_c)$ for windows of all caves in prior time-stamp. Therefore, this comparison is done in constant time.

After detecting such an event we must apply the required changes to $VP(q)$ and $C(q)$ lists. As shown in Fig. 4A, assume that the observer has started its motion from point p and it is now on point q. At the first time-stamp after this moment we will detect this transform event and must update $VP(q)$ and $C(q)$ accordingly. At that time-stamp, we handle this

event as follows. Form the result of the prior time-stamp, we have $e'_c = e_2$ and assume that $\alpha' = R_{v_c}(w'_c)$. Then, $R_{v_c}(w_c)$ is equal to α where α is a range in R_{v_c} next to α' in counter-clockwise order. Also, e_c must be set to e_1 which is the other edge of the scene adjacent to vertex u. Having the enriched visibility graph, these updates can be done in constant time. Updating $VP(q)$ is also equivalent to removing e_2 from it which is done in constant time using the direct links between the cave vertex v and its position in $VP(q)$.

For motion in the opposite direction (starting from r in Fig. 4A), the event is handled in a bit different way. $VP(q)$ is updated by adding e_2 to its correct position and $R_{v_c}(w_c)$ is set to α' which is a range in R_{v_c} prior to $R_{v_c}(w'_c)$ in counter-clockwise order.

4.2. Handling add-remove events

An add-remove event happens on a cave c whenever w_c passes over one of the edges of $R_{v_c}(w'_c)$, named vu, and vu is an edge of the scene. Therefore, it can be detected in the same way as transform events. These events can also be handled similarly. As shown in Fig. 4B, assume that the observer has started its motion from point p and it is now on point q. Here, an add event is happened. It is handled by removing edge vu from $VP(q)$. The cave c associated to vertex v is specified as an effective cave and e_c and $R_{v_c}(w_c)$ are respectively set to e and α where e is the other edge adjacent to u and α is a range in R_{v_c} next to $\alpha' = R_{v_c}(w'_c)$ in counter-clockwise order.

For motion in the opposite direction (starting from r in Fig. 4B) the event is handled similarly. $VP(q)$ is updated by adding vu to its correct position, $R_{v_c}(w_c)$ is set to α' and the cave c is set as an ineffective cave.

It is simple to show that using the enriched visibility graph, add-remove events can also be handled in constant time.

4.3. Handling split-merge events

As shown in Fig. 5, a split event happens on a cave c whenever w_c passes over one of the edges of $R_{v_c}(w'_c)$, named vu for a reflex vertex u, such that w_d, the window of the corresponding cave of u, lies inside the scene just after this moment. Like transform and add-remove events, split event on a cave c is detected by comparing the direction of w_c and the edges of $R_{v_c}(w'_c)$ and checking the vertex u and the direction of the new window w_d which all can be done in constant time.

To handle a split event, the new cave d on vertex u must be inserted into $C(q)$ before or after the cave c. Vertices of the caves c and d are known and are respectively equal to v and u. In order to find their edges and the containing ranges of their windows we use the prepared data structure of the enriched visibility graph. Without loss of generality, assume that v is closer to the observer than u. For the case shown in Fig. 5A, e_c and e_d are equal to $e_{\overrightarrow{vu}}$ and for the case shown in Fig. 5B, e_c is equal to one of the edges of the scene which is adjacent to vertex u and e_d is equal to $e_{\overrightarrow{vu}}$. While vu is an edge of the visibility graph, we know the value of $e_{\overrightarrow{vu}}$ from the preprocessing information. Moreover, $R_u(w_d)$ is equal to $R_u(\overrightarrow{vu})$ which is also known from the preprocessing information and $R_v(w_c)$ is equal to one of the adjacent ranges of $R_v(w'_c)$. Therefore, we can compute the parameters of the new and updated caves in constant time. Consequently, $C(q)$ can be updated in constant time whenever a split event is detected and processed. $VP(q)$ can also be updated in constant time. This is done by inserting e_u, u and e_c into $VP(q)$ after of before the position of e'_c in $VP(q)$. However, e_u or e_v may already exits in these positions. If so, they are not inserted twice.

As shown in Fig. 5, a merge event happens on caves c and d whenever both w_c and w_d pass simultaneously over the supporting line of $v_c v_d$ and v_c and v_d lie in the same side of the observer. Fortunately, this happens only on adjacent caves in $C(q)$.

Lemma 4 *Only adjacent caves of $C(q)$ can be merged.*

Proof: We assume that the vertices of the scene are in general position. This implies that a line can not pass through three vertices of the scene. Then, whenever windows of two caves overlap no other vertices exist between them. □

Therefore, the merge events can be detected by considering the consecutive caves of $C(q)$ and this takes constant time for each cave c in $C(q)$.

The merge events are handled as follows. Assuming that v is closer to q than u, $C(q)$ is updated by removing the cave d, which is the corresponding cave on vertex u. For the case shown in Fig. 5A, e_c is set to e'_d and for the case shown in Fig. 5B, e_c is set to the proper adjacent edge of the vertex u. In both cases, $R_v(w_c)$ is set to the proper adjacent range of $R_v(w'_c)$. The view is also updated by removing e'_c, u and e'_d from $VP(q)$ and inserting e_c instead. However, e'_c and e'_d may be equal or e_c may already exists in $VP(q)$ or it may be equal to e'_d. These cases must be considered properly when we update $VP(q)$.

4.4. Algorithm efficiency

It is important to note that at each time-stamp we may handle several events on a cave. The reason is that the time intervals between our time-stamps may be long and several cave events may happen between two consecutive time-stamps. We process these events according to their correct happening order, and therefore, the final result at each time-stamp is correct.

From the previous subsections we have,

Lemma 5 *At each time stamp where the exact view is drawn, the required time for updating $C(q)$ and $VP(q)$ is linear in terms of the number of the visibility change events happened between the previous and the current time-stamp.*

This bound is the best possible. The reason is that any one of these changes must be applied on the exact view of the previous time-stamp to be equal to the view in current time-stamp.

If the time-stamp intervals are sufficiently small, we can assume that at each time-stamp we apply at most one cave event on each cave of $C(q)$. Consequently, the required time to update and draw the exact view at each time stamp is linear in terms of the positions where the view has changed from the last time-stamp.

We summarize the efficiency of our method in the following theorems:

Theorem 3 *A planar polygonal scene can be preprocessed such that the exact $VP(q)$ for an arbitrarily observer q can be maintained and updated in discrete and sufficiently small time-stamps in $O(|C(q)|)$ processing time as the observer moves in any direction.*

Theorem 4 *The total processing time of maintaining the exact visibility polygon of a moving point observer q on a given path, where $VP(q)$ is required in k time-stamps and $VP(q)$ changes combinatorially l times along this path, is $O(k\overline{|C(q)|} + l)$ where $\overline{|C(q)|}$ is the average size of $C(q)$ along this path.*

Our method is completely output-sensitive and it is superior over the best current methods which are event-driven and focus on maintaining the combinatorial structure of the visibility polygon. As discussed in the introduction, the time complexity of the best known methods for a situation described in the above theorem is $O(k\overline{|VP(q)|} + l \log n)$ where $\overline{|VP(q)|}$ is the average size of $VP(q)$ along the path. Moreover, whenever the observer changes its motion direction, these methods require extra processes($O(n \log n)$) to rebuild their event queue, while, our method does not require such extra processes.

4.5. Rationality problems

In geometric algorithms, rational number computations are always a challenging problem. This problem occurs because of the limited accuracy in real number computations. The lost digits in these computations may lead to abnormal effects on the accuracy of the whole algorithm.

In event driven algorithms where there is a queue of future events, the event times are not exact because of these lost digits. Therefore, when we are at the predicted time of an event, the configuration of the input may not correspond to what we expect for that event. For example, we are to process an event at time t where three points are supposed to be collinear. Because of the limited accuracy of real numbers, we have stored this event to be happened at time t' that is not exactly equal to t. Then, at time t' the three points are not collinear and an inconsistency may arise in our algorithm. This issue must be considered whenever we implement and use a geometric algorithm in real applications which complicates the usage of such algorithms.

In our method, we avoided to build an event queue. Instead, we check all events happened during the last interval of the discrete time-stamps. Therefore, all events are detected and handled correctly in this method and it is still simple to be implemented and used in real applications.

5. Implementation result

We have examined efficiency of the proposed method in practice by implementing this method. We used some real data sets as well as virtually generated data to be fed into the algorithm and approximately obtained similar results. We captured the real test data from some settings of objects in a room to evaluate the results obtained for virtual scenes. While the results were closely similar, we only describe the virtual settings here.

The outer boundary of our virtual scene is a square and the obstacles are regular k-gons and k-stars for different values of k. The observer lies initially at the center of the square. Figure 6 depicts a sample virtual scene. We define the following parameters for a scene $\mathcal{S}$:

- **Density** (d)**:** The density of $\mathcal{S}$ is the ratio of the area of the bounding square covered by the obstacles which is formally the area of the obstacles over the area of the outer boundary square.
- **Average obstacle complexity** (n_h)**:** Average obstacle complexity is the average number of the vertices of the obstacles which is formally the total number of the vertices of the obstacles over h, the number of the obstacles.
- **Time-stamp granularity** (t)**:** Time-stamp granularity is the ratio of the distance between the positions of the observer in two consecutive time-stamps and the length of the edges of the bounding square.

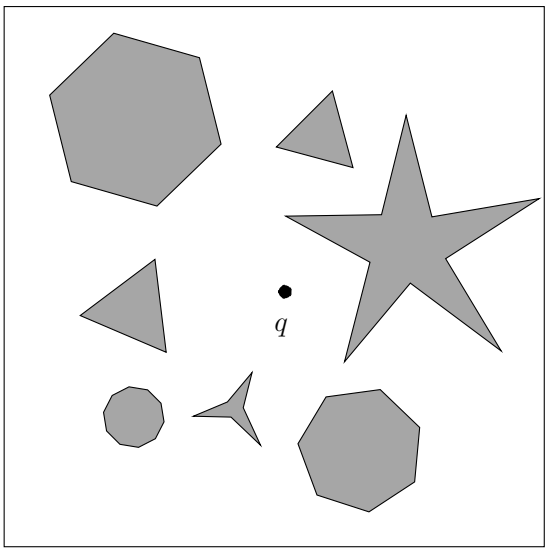

Figure 6: A sample virtual scene

- **Average time-stamp event complexity** (n_e)**:** Average time-stamp event complexity is the average number of visibility change events happened between two consecutive time-stamps. We have computed this parameter for the initial position of the observer and its position in the next time-stamp moving along a diagonal of the bounding square.
- **Average cave complexity ratio** (c)**:** Average cave complexity ratio is defined to be $|C(q)|/|VP(q)|$ for the initial position of the observer q.

The obstacles are placed uniformly inside the square in such a way that they do not intersect each other. We show our implementation results in three diagrams shown in Figs. 7, 8 and 9.

We first checked the relation between n_e and t. For this purpose we built scenes of 1000 obstacles for $d \in \{0.00001, 0.00002, \ldots, 0.0001\}$ and $n_h \in \{5, 10\}$. Figure 7 shows the result. Then, for a fixed $t = 0.0001$ we checked the relation between n_e and h. We built scenes of $h \in \{1000, 2000, \ldots, 10000\}$ obstacles for $d \in \{0.00001, 0.00002, \ldots, 0.0001\}$ and $n_h \in \{5, 10\}$. Figure 8 shows the result for this relation. Finally, we checked the relation between c and other parameters of the scene. We understood that the tangible effective parameter is n_h and number of reflex vertices of the obstacles. In our virtual scenes, there are two obstacle types (k-gon and k-star) for which the average number of reflex vertices is a constant factor of n_h. So, we only checked the relation between c and n_h for which the result has been shown in Fig. 9.

The following interesting observations can be perceived from these diagram:

1. There is a linear relation between n_e and t, *i*.e., for given values of d, h and n_h, n_e grows linearly as t increases linearly.
2. There is a linear relation between n_e and h, *i*.e., for given values of d, t and n_h, n_e grows linearly as h increases linearly.
3. The relation between n_e and d is something greater than linear, *i*.e., for given values of t, h and n_h, as d increases linearly n_e growth ratio is super linear.
4. There is a linear relation between n_e and n_h, *i*.e., for given values of d, h and t, n_e grows nearly linearly as n_h increases linearly.

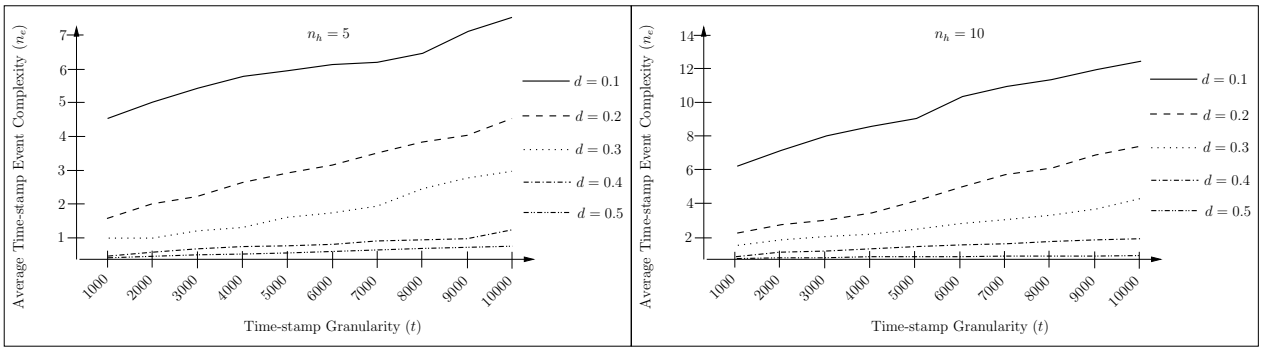

Figure 7: Relation between n_e and t for $h = 1000$

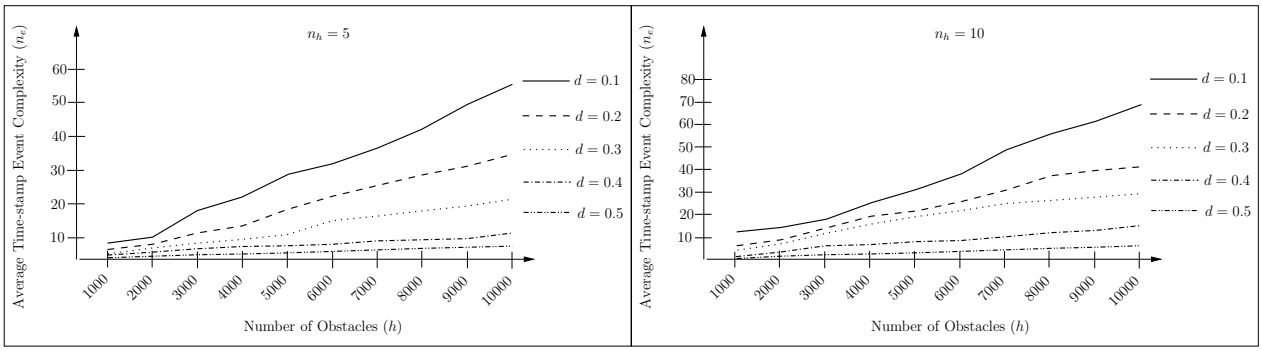

Figure 8: Relation between n_e and h for $t = 0.0001$

5. In a dense scene ($d = 0.5$), n_e is usually constant. Trivially, n_e depends on t, but, for realistic small values of t, ($t = \frac{1}{10h}$), we have $n_e < 1$ for both $n_h = 5$ and $n_h = 10$. Moreover, in a sparse scene ($d = 0.1$), n_e grows rapidly as n_h grows.
6. There is a linear relation between c and n_h, *i*.e., regardless of the values of d, h and t, c grows linearly as n_h increases linearly.

From the above observation we can conclude that reducing the complexity of drawing $VP(q)$ from $|VP(q)|$ to $|C(q)|$ is an acceptable improvement which is done in this paper. Also, nonnecessity of handling visibility event separately from the drawing procedure is another enhancement which is obtained using our method.

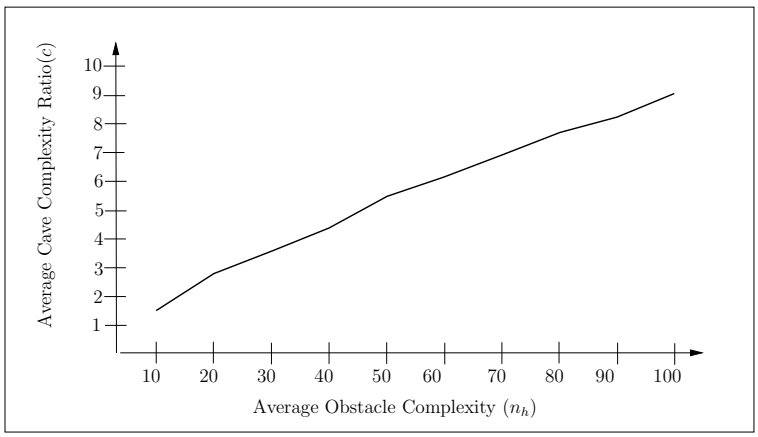

Figure 9: Relation between c and n_h

6. Conclusion

In this paper, we considered the notion of exact visibility in planar polygonal scenes. The aim is to maintain the exact view of a moving observer in such environments. The non-exact solutions of this problem has been considered before from which the exact view can be obtained by doing some more processes. But, such solutions are not efficient and require a queue of events whose maintenance and usage is complicated and unnecessary in real applications.

We have proposed a method that can be used directly in computer graphics and other visibility related applications. In this method, we prepare an enriched version of the visibility graph in the preprocessing phase which contains enough data to be used during the motion. Then, the changes in the visible areas of a moving observer are detected and handled efficiently in linear time in terms of the number of the occurred changes. In this algorithm, there is no restriction on the direction and speed of the observer and they can be changed any time without requiring adjustment in the current data structures. The algorithm draws or produces the exact view in discrete time-stamps of arbitrary distance. This algorithm prepares a special representation of the visibility graph in its preprocessing phase.

The possible and interesting extensions of this work can be applied for dynamic environments or line segment observers. Using this method on 3D scenes is another extension of this method. Reducing the preprocessing cost of this method is an interesting problem which will enhance the efficiency and usability of this method. This goal may be obtained using another alternative preprocessing data structures instead of the visibility graph.

References

[1] B. Aronov, L. Guibas, M. Teichmann, L. Zhang: *Visibility Queries and Maintenance in Simple Polygons*. Discrete and Computational Geometry **27** (4), 461–483 (2002).

[2] T. Asano: *Efficient Algorithms for Finding the Visibility Polygons for a Polygonal Region with Holes*. Manuscript, Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, 1984.

[3] T. Asano, L. Guibas, J. Hershberger, H. Imai: *Visibility of disjoint polygons*. Algorithmica **1**, 49–63 (1986).

[4] H. Edelsbrunner, L. Guibas: *Topologically sweeping in an arrangement*. Proc. 18th Annual Symposium on Theory of Computing, 1986, pp. 389–403.

[5] S. Ghali, A.J. Stewart: *Incremental update of the visibility map as seen by a moving viewpoint in two dimensions*. Seventh Internat. Eurographics Workshop on Computer Animation and Simulation, August 1996, pp. 1–11.

[6] S.K. Ghosh, D.M. Mount: *An output sensitive algorithm for computing visibility graphs*. Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles/CA 1987, pp. 11–19.59.

[7] S.K. Ghosh, D.M. Mount: *An output-sensitive algorithm for computing visibility graphs*. SIAM Journal on Computing **20**, no. 5, 888–910 (1991).

[8] P.J. Heffernan, J.S.B. Mitchell: *An Optimal Algorithm for Computing Visibility in the Plane*. SIAM Journal of Computing **24** (1), 184–201 (1995).

[9] O.H. Holt: *Kinetic Visibility*. PhD. Thesis, 2002.

[10] S. Hornus, C. Puech: *A Simple Kinetic Visibility Polygon*. Proc. 18th EWCG'02, 2002, pp. 27–30.

[11] S. Kapoor, S.N. Maheshwari: *Efficient algorithms for Euclidean shortest path and visibility problems with polygonal obstacles*. Proc. 4th Annual ACM Symposium on Computational Geometry, Urbana/IL 1988, pp. 172–182.

[12] S. Kapoor, S.N. Maheshwari: *Efficiently constructing the visibility graph of a simple polygon with obstacles*. SIAM Journal on Computing **30**, no. 3, 847–871 (2000).

[13] D.T. Lee: *Proximity and reachability in the plane*. PhD. Thesis and Tech. Report ACT-12, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana/IL 1978.

[14] K. Nechvile, P. Tobola: *Local approach to dynamic visibility in the plane*. Seventh Int. Conf. in Central Europe on Computer Graphics and Visualization, WSCG '99, Feb. 1999.

[15] M.H. Overmars, E. Welzl: *New methods for constructing visibility graphs*. Proc. 4th Annual ACM Symposium on Computational Geometry, Urbana/IL 1988, pp. 164–171.

[16] M. Pocchiola, G. Vegter: *Computing the visibility graph via pseudo-triangulations*. Proc. 11th Annual ACM Symposium on Computation Geometry, Vancouver, B.C., 1995, pp. 248–257.

[17] M. Pocchiola, G. Vegter: *The visibility complex*. Internat. J. Comput. Geom. Appl. **6** (3), 279–308 (1996).

[18] S. Riviére: *Walking in the Visibility Complex with Applications to Visibility Polygons and Dynamic Visibility*. Proc. Canadian Conf. on Comp. Geom., 1997.

[19] S. Riviére: *Dynamic visibility in polygonal scenes with the visibility complex*. Proc. 13th Annual ACM Symposium onComputational Geometry, Nice/France 1997, pp. 421–423.

[20] S. Suri, J. O'Rourke: *Worst-Case Optimal Algorithms for Constructing Visibility Polygons with Holes*. Proc. second annual symposium on Computational geometry 1986, pp. 14–23.

[21] E. Welzl: *Constructing the visibility graph for n line segments in $O(n^2)$ time*. Information Processing Letters **20**, 167–171 (1985).

[22] A. Zarei, M. Ghodsi: *Efficient Computation of Query Point Visibility in Polygons with Holes*. Proc. 21st Annual Symposium on Computational Geometry, Pisa/Italy, 2005.

[23] A. Zarei, A.A. Khosravi, M. Ghodsi: *Maintaining Visibility Polygon of a Moving Point Observer in Polygons with Holes*. 11th CSI Computer Conference (CSICC'2006), IPM School of Computer Science, Tehran 2006, pp. 32–39.

Received July 15, 2008; final form March 7, 2009