Exact Visibility Maintenance in Planar Polygonal Scenes in Practical Applications

Alireza Zarei*
zarei@mehr.sharif.edu

Mohammad Ghodsi*
ghodsi@sharif.edu

Computer Engineering Department, Sharif University of Technology, Tehran, Iran
Institute for Studies in Theoretical Physics and Mathematics (IPM), Tehran, Iran

Abstract

In this paper, we are concerned with computing and maintaining the exact visibility polygon $V(q)$ of a moving point observer q in planar scenes represented by a polygon with holes with complexity of n. We propose an algorithm that applies and computes each change to $V(q)$ in constant time as q moves to a new neighboring position. In fact, in our method changes to $V(q)$, as the observer q moves, are applied in linear time in terms of the number of required changes, which is normally small for each movement. The best known algorithms maintains $V(q)$ as a sequence of polygon edges seen by q, and changes to $V(q)$ is computed in $O(\log(|V(q)|)$. But in these methods, the computation of exact visibility, needed by the practical applications, requires another trace of $V(q)$. We use a representation of the visibility graph from which the visible area of an observer is updated by merely applying the necessary changes to the visible area as the observer moves. This is done in optimal time and space without maintaining a queue of future events as previous algorithms do.

Keywords: *Exact visibility maintenance, moving observer, planar polygonal scene, visibility polygon*

1 Introduction

A planar polygonal scene, also known as polygon with holes, is a 2D region with a polygonal border with some disjoint polygonal objects inside. These objects, called holes, act as occluders which observers can not see through. The complexity of the polygon, 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, denoted by $V(q)$. A moving observer can move in any direction and in a speed which is defined by a constant degree algebraic function over time.

In this paper, we are concerned with computing and maintaining the exact visibility polygon, $V(q)$, of a moving point observer q in a polygon. This problem has obviously many applications in computer graphics, computer games, machine vision, robotics and motion planning.

This problem has been considered by many researchers. The main existing solutions for this problem are based on different notions including visibility decomposition, visibility complex, tangent visibility graph, kinetic data structures, topological map and radial subdivision [2, 3, 4, 5, 6, 7, 8, 9, 10]. These methods normally have preprocessing steps to prepare useful information about the visibility coherence of the scene. Cost of this step varies in different approaches from $O(n^3 \log n)$ time and $O(n^3)$ space to $O(n \log n)$ time and $O(n)$ space, respectively. The prepared information of this step is then used to efficiently find the initial value of $V(q)$ and update it as the observer q moves. In these methods, the initial $V(q)$ is computed in $\Omega(|V(q)| \log n)$ time. During the motion, a queue of $O(n)$ events is built by which each visibility change is handled in time of $O(\log n)$ to update and maintain $V(q)$. However, in about all these methods, some of the processed events are unnecessary and do not affect $V(q)$.

The main deficiency of the existing results is that only the combinatorial structure of $V(q)$ is maintained and only the changes to this structure are handled. What is maintained for $V(q)$ is a se-

*This work has been partly supported by a grant from IPM school of CS (No. CS1385-2-01) and Iran Telecommunication Research Center(ITRC).

quence of vertices and edges of the scene which are (partially) visible to q. The exact visible portion of each edge of this sequence, needed in real applications, can be determined by a linear trace of this sequence.

The focus of this paper is to improve the exact computation of $V(q)$ as q moves, even when only the visible portion of an edge is changed by this movement. Such changes that do not alter the combinatorial structure of $V(q)$ occur most often in real applications. Our algorithm maintains a list of edges whose visible parts are subject to change when the observer moves. Hence, the application developer can simply use it to have the exact $V(q)$ in each time stamp.

We propose an algorithm that applies and computes each change to $V(q)$ in constant time as q moves to a new neighboring position. To be exact, our method applies changes to $V(q)$, as the observer q moves to a new location, in linear time in terms of the number of required changes done to the exact visibility, which is normally small for each small movement. Another considerable result of this method is that the visibility events are detected and handled in constant time. Our method uses a special version of the visibility graph as its prepared data structure. We represent the visibility graph in a special way and store some more information in it to detect and handle the events efficiently. It is noticeable that this method can be used in conjunction with other alternative methods, however by some adjustments.

In the rest of this paper, we will first overview our representation of the visibility graph in Section 2. The new method is described in Section 3 and its efficiency will be analyzed in Section 4. The materials will be summarized in Section 5.

2 The visibility graph

The result presented in this paper, needs another algorithm to find the initial $V(q)$ as well as some data structures for predicting and determining visibility events during the observer motion. Any one of the methods of [2, 11, 12, 13, 14] can be used here. However, we use the method of [13] which efficiently computes $V(q)$ for a query point q. Also, we use the visibility graph data structure for detecting events during the motion. In the visibility graph of a scene S which is denoted by $VG(S)$, there is a vertex for each vertex of the scene and an edge between two vertices if their corresponding vertices of the scene see each other. Trivially, size of this data structure is between $O(n)$ and $O(n^2)$

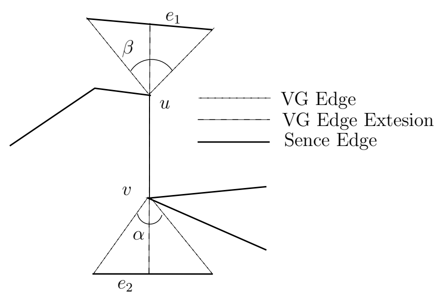

Figure 1: The special representation of the visibility graph.

and it can be constructed in optimal time of $O(n^2)$ [1].

However, we need a special representation of the visibility graph by which we can handle a visibility event in constant time. In this representation, incident edges with each vertex are sorted clockwise and 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.

There is another thing that we need in our representation of the visibility graph. For each edges of the visibility graph we maintain the ranges of its endpoints that the supporting line of this edge passes through from both sides. This is shown in Figure 1. For the edge vu, the ranges α and β are those that the supporting line of vu pass through and must be maintained along with this segment.

Another thing that we need to maintain in this representation is the first edge of the scene that any one of these two extensions of each edge of the visibility graph encounters. For example, in Figure 1 we maintain edges e_1 and e_2 along with the segment uv which is an edge of the visibility graph.

All of these information can be embedded into the classic visibility graph without increasing its space complexity. However some changes are required to the algorithms for computing the visibility graph to generate these extra information. These changes can be applied to the method of Pocchiola and Vegter [1] easily and this can be done without increasing the time complexity of their method.

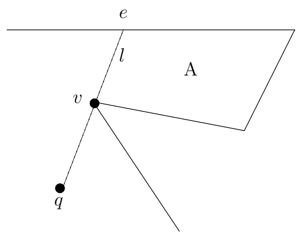

Figure 2: When q moves, only the visible portion of e in $V(q)$ is changed. A is a cave and v, l and e are its vertex, window and edge, respectively.

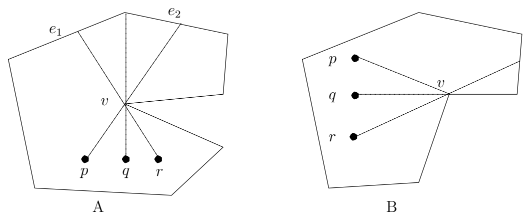

Figure 3: Cave events: in part A(B), when the observer moves from p to r or from r to p a transform(add-remove) event occurs at point q.

3 Maintaining the exact visibility

When an observer q moves inside a scene, only some parts of $V(q)$ are changed. Therefore, online drawing or illumination of $V(q)$ can be done by updating only the changed areas. These areas always belong to some reflex vertices which are visible to q. The vertex v in Figure 2 is an example of such vertices. In this figure, only the left portion of edge e which is bounded by the supporting line of qv, is visible to q. When q moves, this visible portion, based on the motion direction, is increased or decreased. The region A in Figure 2 is called a *cave* and v, l and e are its vertex, window, and edge respectively. We denote the vertex, window, and edge of a cave c by v_c, w_c and e_c.

Online maintenance of the exact visibility polygon, or exact $V(q)$, is equivalent to determination and maintenance of 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. Therefore, after initializing $V(q)$ for the initial location of the observer q, a list of the cave vertices and edges in $V(q)$ must be determined. During the online maintenance, this list specifies the parts of $V(q)$ which must be updated as q moves around. This list is denoted by $C(q)$ in which the caves are stored according to their order in $V(q)$. Each cave is specified by its vertex, window and edge.

Initializing $C(q)$ is straightforward by a linear trace on $V(q)$. $V(q)$ contains the clockwise(counter-clockwise) ordered sequence of visible edges and vertices. In this sequence, an edge following (proceeding) a vertex whose is not adjacent in the scene, compose a cave in $C(q)$.

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 it. We call the events which change $C(q)$ *cave events*. There are three types of cave events which are shown in Figures 3 and 4.

The first type of these events, called *transform*, is shown in part A of Figure 3. In this figure, when the observer moves from p to r, before arriving to the point q, e_1 is the edge of a cave and after crossing q, the edge e_2 will be the cave edge. When the observer moves from r to p a similar event happens. In a transform event a new edge appears in $V(q)$ and will be the edge of a cave, or, a cave edge disappears from $V(q)$ and its adjacent edge will be the cave edge.

The second type of cave events, which is shown in part B of Figure 3, is called *add-remove* event. According to this figure, when the observer moves from point p to r, before arriving to the point q, there is no *active* cave assigned to the reflex vertex v. But after passing the point q and in point r there is a cave on this vertex. Therefore, on point q the new cave must be added to $C(q)$. This scenario happens in reverse when the observer moves from point r to point p; here, on point q the cave of vertex v must be removed from $C(q)$.

The third type of cave events is called *split-merge*. In these events two caves are merged to a single one or a cave is split into two distinct caves. Figure 4 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 on point r and on 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 cave deletion or insertion occurs and its related changes must be handled.

The details of our method are discussed in the

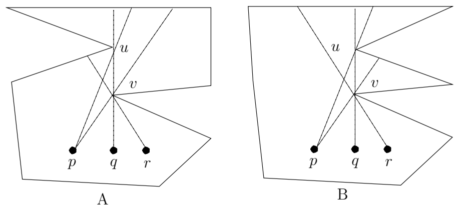

Figure 4: Cave events: in part A and B, when the observer moves from p to r or from r to p a split-merge event occurs at point q.

following subsections: We first describe the initialization tasks for a given query point. Then, the continuous update and maintenance of the exact visibility polygon is described. Finally, the details of event handling for all types of events will be discussed. We also have a short discussion on rational numeric operations. The proposed method is then analyzed in the next section.

3.1 The algorithm initialization

Before starting the motion, the initial view of the observer must be computed. For an observer q, $V(q)$ of its initial placement is computed by any one of the present algorithms like [13]. According to these algorithms, $V(q)$ is the ordered sequence of edges and vertices which are visible to q. We must also prepare $C(q)$ which is the ordered sequence of the caves in $V(q)$. For any one of the reflex vertices in $V(q)$ we have an entry in $C(q)$. Also, for any cave c of these caves we maintain the range α of v_c in our representation of the visibility graph in which w_c lies. We maintain direct access links between $C(q)$, $V(q)$ and edges and vertices of the scene. These links help us to directly navigate from cave edges or vertices of $C(q)$ to their positions in $V(q)$ in constant time. Also, it links the edges or vertices of $V(q)$ to their occurrence in the scene data structure.

3.2 Continuous update of the view

As the observer moves, $V(q)$ must be adjusted accordingly to match the exact visibility polygon of q in each time stamp. This is done by continuously updating caves while the observer moves. For each cave c in $C(q)$, the intersection point of e_c and the supporting line of qv_c is computed in each time stamp. 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 $C(q)$.

However, this type of updates is valid only while the list $C(q)$ is valid. But as discussed before, when the observer moves, some caves may be eliminated from $C(q)$ and new caves may appear. Hence, the correctness of this method depends on correct cave event detection and handling.

3.3 Handling transform events

At each time step and on updating exact $V(q)$, each cave edge is checked to see whether its other endpoint is reached. When this happens, it specifies a cave event and $C(q)$ must be updated accordingly. As shown in part A of Figure 3, assume that the observer has started its motion from point p and it is now on point q. This is a transform event. The reason is that the adjacent edge(e_1) of the cave edge(e_2) will be visible to the observer when it continues its motion. To handle this event, the previous edge of the cave (e_2) is replaced by the new edge of the cave (e_1). Also, $V(q)$ is updated by adding the new edge. For motion in the opposite direction (starting from r) $V(q)$ and $C(q)$ are updated by substituting e_1 with e_2.

On the other hand, a transform event happens when the window of a cave passes over a segment of the visibility graph of the cave vertex. Therefore, having the previously described representation of the visibility graph and the containing range of each cave window, we can detect the time at which a transform event occurs by comparing the current and previous ranges of the cave window which can be done in constant time.

3.4 Handling add-remove events

While we maintain an entry in $C(q)$ for all reflex vertices, these events can also be detected by checking the (imaginary) window of a cave to see whether it leaves its previous containing range or not. If so, based on the type of the event(add or remove), the cave will be activated by specifying its edge or it will be deactivated by removing its edge.

If the event is an add event, $V(q)$ is also updated by removing from it the proper adjacent edge and vertex of the cave vertex. For the case of remove event a new edge and vertex must be added to $V(q)$.

3.5 Handling split-merge events

Cave events occur when a cave window passes on a vertex. This is equivalent to what described in previous subsections for detecting transform and add-remove events. Therefore, it is enough to detect the time at which a cave window leaves its containing range of the visibility graph and enters into an adjacent one. Assume that an observer has started its motion from point r in part A of Figure 4. The time at which this split-merge event happens is identified by this method.

However, event handling is different here. The old cave must be removed from $C(q)$ and two new caves with u and v as their vertices must be added instead. The vertices of these caves are known when the event happens but their edges are not known and must be computed. In these cases, uv is an edge of the visibility graph of the scene. It is enough to know the first edge of the polygon that is encountered by extensions of each edge of the visibility graph in both endpoints. While we have these information as discussed in Section 2, we can build the new caves in constant time whenever such events happen.

There is another kind of split-merge events which is handled by another different method. Assume that the observer has started its motion from point p in part A or B of Figure 4. When the observer reaches point q, its two caves are merged into a single new one. Fortunately, this happens only for adjacent caves in $C(q)$.

Lemma 1 . *Only adjacent caves in $C(q)$ are subject to be merged.*

Proof. Trivially, when two windows of two caves overlap it means that there is nothing between them. □

In these merge events, it is simple to build the new cave. Its vertex is the closer vertex of the previous two caves to the observer and based on the position of the caves, its edge is the closer or farther edge of the old caves to the observer. Therefore, such events can be handled by removing the merged caves from $C(q)$ and inserting the new cave into their position in $C(q)$.

Finally, assume that an observer lies on point r in part B of Figure 4 and moves to q and then to p. At point q, the old cave with vertex v is split into two new caves with u and v vertices. These cases are also detected in the same way because uv is a segment in the visibility graph. When this event happens, the old cave is removed from $C(q)$ and the new two caves are added to $C(q)$. To do this event handling efficiently, we use the extra information we maintained in our visibility graph representation. The vertex, edge, and window of both these caves and the containing range of the cave with vertex v is already known. The containing range of the other cave is exactly the containing range of the extension of the segment vu from endpoint u. While we already have this data in our representation of the visibility graph, the containing range of the window of the other cave is also known. So, the new caves can be constructed in constant time.

3.6 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 and the lost digits in these computations may lead to abnormal effects on the whole algorithm accuracy.

In our algorithm, a noticeable issue is about the way that events are detected. While the time elapses continuously, $V(q)$ is updated in discrete time stamps. In a naive implementation of this algorithm, the events which occur between theses time stamps may be lost and not be processed. However, the solution to this problem is simple. It is enough to check in each time stamp whether an event has occurred between the previous time stamp and the current one. If such an event has occurred it is handled at the current time stamp.

4 Complexity analysis

Our algorithm is composed of two phases: the preprocessing phase and the running phase. In the preprocessing phase, a set of useful information is gathered to facilitate the next phase. In this section, time and space complexities of both phases are analyzed.

Lemma 2 . *Size of the data structures produced in the preprocessing phase for a planar polygonal scene S of n vertices is $O(|VG(S)|)$ which is between $O(n)$ and $O(n^2)$.*

Proof. We use only constant data on each edge of the visibility graph. So, the size of the preprocessing data structure is proportional to the size of the visibility graph of the scene. □

Lemma 3 . *The time complexity of the preprocessing phase is $O(n^2)$.*

Proof. This is exactly the time complexity for computing the visibility graph. Our extra information and our representation of this data structure does not increase the total time complexity of this computation because it only adds some constant extra processes for each segment of the visibility graph. □

Lemma 4 . *At each time stamp which the exact $V(q)$ must be updated, the required time of this update is linear in terms of $|C(q)|$ and this time bound is the best possible.*

Proof. At each time stamp which we need to update the exact $V(q)$, the visible portions of the cave edges are subject to change. So, we need to check them with respect to the observer location at the current time stamp. Only a constant number of events are associated with a cave and checking whether any one of these events has happened requires $O(1)$ time. Hence, processing a cave event requires constant time and the total time which is required in each time stamp is $O(|C(q)|)$.

Trivially this bound is the best possible because any one of the caves must be processed to find the visible part of it in each time stamp. □

We summarize these lemmas as the following theorem which describes the efficiency of this method:

Theorem 1 . *A planar polygonal scene of total n vertices and h objects can be processed in $O(n^2)$ time to prepare data structures of total size of $O(n^2)$ such that the exact $V(q)$ for an arbitrarily observer q can be maintained and updated continuously as the observer moves in any direction in $O(|C(q)|)$ time.*

5 Conclusion

In this paper we considered the notion of exact visibility in a planar polygonal scene. The aim is to maintain the exact view of a moving observer in such environments. The non-exact solutions to 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, after preprocessing the scene, the changes in the visible areas of a moving observer are detected and handled efficiently in linear order of the number of changes occurred. In this algorithm, there is no restriction on the direction and speed of the observer and they can be changed in any time without requiring adjustment in the current observer data structures.

This algorithm prepares a special representation of the visibility graph in its preprocessing step. The possible and interesting extensions of this work can be applied for dynamic environments or line segment observers. Also, other alternative visibility coherence data structures can be used instead of the visibility graph as the preprocessing information. Adjustments of this algorithm to use these alternatives and comparing their efficiencies can be done following the method of this paper.

References

[1] M. Pocchiola and G. Vegter, *Computing the visibility graph via pseudo-triangulations* , In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages 248–257, June 1995.

[2] A. Zarei and M. Ghodsi, *Efficient Computation of Query Point Visibility in Polygons with Holes*, Proc. 21st Annual Symposium on Computational Geometry, June 6-8, 2005, Pisa, Italy.

[3] 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, Jan 24–26, 2006, pp. 32-39.

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

[5] Olaf H. Holt, *Kinetic Visibility*, PhD Thesis, 2002.

[6] K. Nechvile and Petr Tobola, *Local approach to dynamic visibility in the plane*, In Seventh Int. Conf. in Central Europe on Computer Graphics and Visualization, WSCG '99, February 1999.

[7] S. Rivi'ere, *Dynamic visibility in polygonal scenes with the visibility complex*, In Proc. 13th ACM Sympos. Comput. Geom., pages 421-423, 1997.

[8] Rivi'ere, S. *Walking in the Visibility Complex with Applications to Visibility Polygons and Dynamic Visibility*, Proc. Canadian Conf. on Comp. Geom., 1997.

[9] Samuel Hornus, Claude Puech, *A Simple Kinetic Visibility Polygon*, 18th EWCG'02 proceedings, page 27-30 - 2002.

[10] S. Ghali and A. J. Stewart, *Incremental update of the visibility map as seen by a moving viewpoint in two dimensions*, In Seventh International Eurographics Workshop on Computer Animation and Simulation, pages 1-11, August 1996.

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

[12] M. Pocchiola and G. Vegter, *The visibility complex*, Internat. J. Comput. Geom. Appl., 6(3):279308, 1996.

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

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