A simple kinetic visibility polygon

Samuel Hornus
iMAGIS-GRAVIR/IMAG-INRIA
Samuel.Hornus@imag.fr

Claude Puech
iMAGIS-GRAVIR/IMAG-INRIA
Claude.Puech@imag.fr

ABSTRACT

Given a set of moving obstacles in the plane, we propose a method for maintaining efficiently the visibility polygon of a (possibly moving) viewpoint. We consider both smooth-convex, and simply-polygonal obstacles.

Categories and Subject Descriptors

F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Non-numerical Algorithms and Problems*

Keywords

kinetic data structure, visibility polygon

1. INTRODUCTION

Visibility computations are central in many computer graphics algorithms and in robot motion planning. A very useful tool is the determination of the objects visible from a viewpoint. Thus, in the plane, the visibility polygon is an important visibility structure. It is a star shaped polygon centered at the viewpoint, whose edges are the visible parts of the objects in the scene, and whose interior intersects no object.

Efficient algorithms exist to compute the visibility polygon in the static case, but applications of this problem may apply to moving objects. Computing the visibility polygon at various times on a static " snapshot " of the scene is inefficient, since we do not take into account the temporal coherence that arise from the continuity of the movements of the objects (and possibly the viewpoint): if the time step is too small, we will compute many times the very same (combinatorial) visibility polygon.

We use the *kinetic data structure* framework introduced by Guibas [2, 1] to propose a *simple* algorithm that maintains the visibility polygon of a view point in a 2-dimensional scene when all objects may move. The structure maintained is in fact a weak radial decomposition of the scene. Section 2 treats the case of smooth convex obstacles. Section 3 examines the case of simple (non auto-crossing) polygons.

Kinetic data structures (KDS) are a way to efficiently and accurately maintain an attribute built on top of continuously moving items (e.g. a convex hull). In order to maintain an attribute $\mathcal{A}$ over a set of moving items (items are generally points), each test in the proof of correctness of the construction of $\mathcal{A}$ is analyzed to detect the time at which it will fail. The idea is that maintaining the validity of all those tests (called *certificates*) guarantees that the attribute $\mathcal{A}$ is maintained also, since the certificates provide a proof of correctness. Certificates are ordered in a priority queue, according to their failure time. When the simulation time passes above the first certificate's failure time, the attribute is modified, and the proof is updated (i.e. some certificates disappear, others are created, and their failure time is computed). This method only requires that the motion of the items be known in the short term. For short, one could say that kinetic data structures get rid of step-by-step simulations, and implement in fact time sweep algorithms.

We now get interested in maintaining the visibility polygon of a scene. Here, in the KDS terminology, the items are the convex smooth objects, or the polygons' vertices. We maintain a weak radial decomposition of the scene, thus, the certificates we use take care of the well ordering (i.e. cyclically sorted) of the segments in the decomposition. Finally the radial decomposition of the scene allows us to quickly build the visibility polygon.

2. CONVEX OBSTACLES

Let $\mathcal{O}$ be a set of n convex obstacles in the plane. Let $\mathcal{F}$ be the " free space ": the complement of the union of the obstacles in the plane. Let V be a point in $\mathcal{F}$. We aim at maintaining the visibility polygon of V when V and elements of $\mathcal{O}$ move in the plane. We assume we can compute in constant time the visibility tangents of an obstacle, that are defined as the two tangents to the object passing through the view point V. Let $\mathcal{T} = \{t_0, t_1, \dots, t_{2n-1}\}$ the t_uple of the visibility tangents, sorted in the counter-clockwise order.

Let $u \in \mathcal{S}^1$ be a direction. We denote by $V(u)$ the obstacle seen by V in the direction u. $V(u)$ can possibly be the " blue sky " that we denote ∞. One important observation is that $V(u)$ is constant between two consecutive visibility tangents. Thus, a way to define the visibility polygon around V, is to see it as the function $\mathcal{P} : \mathcal{T} \mapsto \{\mathcal{O} \cup \infty\}$ so that $\mathcal{P}(t_i) = V(t_i^+) = V(t_{i+1}^-)$ where t_i is seen as a direction pointing away from the view point V.

2.1 Kinetic visibility polygon

The visibility polygon (VP) changes only when two visibility tangents (VT) cross each other (but two VTs may cross each other without affecting the VP). Thus, we can maintain the VP by detecting when two consecutive VTs will cross, then updating the VP according to the kind of both VT, and swapping the two VTs involved to keep them sorted in counter-clockwise order.

However, having computed the VP at a given time is not sufficient to maintain it efficiently when obstacles move. We need some additional data that will have to be maintained also: for each VT t_i, we maintain its *hit-item*, which is the obstacle that is hit by the VT beyond the tangency point (it can be ∞). In fact, we maintain a weak radial decomposition of the scene, where only the far object hit by a VT is recorded and not the near object.

For each crossing, the update of the visibility polygon is done in constant time, by distinguishing 8 cases. First, we need to characterize the VTs. Half of them will be *Left* if (seen from V), they pass to the left of the obstacle. The other half will be *Right* visibility tangents.

We explain the naming of the crossing events with an example. Figure 1a shows an $\widehat{L}R$ and an $\widehat{L}L$ crossing events (from left to right, the figure presents the obstacles involved in the event, just before, « during » and after the crossing). $\widehat{L}R$ means that the first VT (in counter-clockwise order) is a *Left* tangent, the consecutive VT is a *Right* tangent, and the hat on $\widehat{L}$ means that, when the crossing occurs, the tangency point of the *Left* tangent is farther from V than the tangency point of the *Right* tangent. Hence, the 8 cases are named $L\widehat{L}$, $\widehat{L}L$, $L\widehat{R}$, $\widehat{L}R$, $R\widehat{L}$, $\widehat{R}L$, $R\widehat{R}$, $\widehat{R}R$.

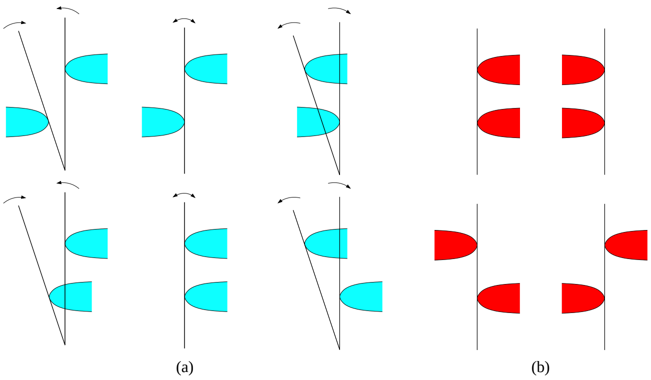

Figure 1: (a) example of a $\widehat{L}R$ event (up) and a $\widehat{L}L$ event (down); (b) all events

Now we need to update the VP and the *hit-items* when the $\widehat{L}R$ crossing occurs, see Figure 2. The VTs that cross each other are consecutive in our t_uple of ordered VTs. Let t be the time at which the crossing occurs. Then at times t^- and t^+, no other VT can lie between the two VTs we are interested in. Therefore we can be sure that any other obstacle (different from G or D in Figure 2) either completely crosses the angular section E, or has no intersection with it. This ensures the correctness of the update process.

First we check if an object C exists between the two points of tangency at time t. To do so, we just need to check whether `g.hit-item` (the hit-item of the *Right* tangent of obstacle G) is the same as `d.hit-item` or not. If so, then C does not exists, else, C exists. Note that `{g,d}.hit-item` can be ∞. This information is enough to update the *hit-items*.

Seen from V, the foremost obstacle among those in the figure is G. Therefore, if there is a change in the visibility for E, this change makes G visible. To know whether G becomes the visible object in E, we simply check whether there exists another obstacle in front of G at time t, by comparing the obstacle visible in E to C or S (depending on the existence of C).

The algorithm is described in Figure 3.

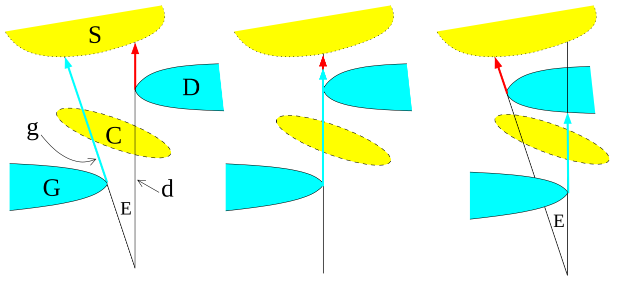

Figure 2: Update of a $\widehat{L}R$ event, the arrows point to the *hit-items* in the case C does not exists

The seven other cases are processed in the very same way, by just changing the roles in Figure 3.

```
g.tangent-item == G;
d.tangent-item == D;
--------------------
if ( G.hit-item == S ) {
    // C does not exists
    g.hit-item = d.tangent-item;
    if ( E.vis-item == S )
        E.vis-item = g.tangent-item;
}
else { // C exists, C == g.hit-item
    if ( E.vis-item == g.hit-item )
        E.vis-item = g.tangent-item;
}
```

Figure 3: Processing an $\widehat{L}R$ event

We know how to maintain the weak radial decomposition of our scene. Without more work, we can compute the visibility polygon only in linear time, by looking at the visible item between each visibility tangent, and « merging » the same consecutive values. This is not very efficient, but we easily remove this problem by performing a first « run-length-encoding » of the consecutive visible items, and by maintaining this encoding each time the VP is changed when a crossing-event occurs. This is done in constant time.

2.2 Complexity

We express the complexity of this kinetic data structure using terms proposed by Guibas and Basch [2, 1]. Our data structure is optimal in size since it is linear in the size of the scene (the set of all obstacles). It is *responsive*, meaning that the cost of processing a certificate failure is small: constant-time in our case. This KDS is *local*, meaning that the number of certificates that involve a single object is small; it is $O(1)$ in our case, with max. 4 certificates per obstacle.

However, it is not optimal since we may have to update many certificates in a move while none of these affect the visibility polygon. Imagine lots of small discs vertically aligned above a big disc, and the view point traversing the plane horizontally under the big disc. Using Guibas and Basch terminology, our KDS is not *efficient*, since the total number of events processed may be of a higher order as the number of changings in the VP. An optimal algorithm would update as many certificates as there are changes in the visibility polygon during the animation. Hall-holt and Rusinkiewicz [3, 4] propose such an algorithm, but are limited to convex smooth obstacles. They do process only one certificate failure for each change in the VP, but the cost of processing one event is not constant in time. However, the overall cost of processing all events for a simple motion (of the observer only) is significantly better in their algorithm.

3. SIMPLE POLYGONAL OBSTACLES

We now present an adaptation of the method to simple polygonal obstacles. A simple polygon can be concave, but none of its edges cross one another. We consider that obstacles in the initial set are in general position, meaning that no pair of vertices is aligned with the observer V.

The basic idea is the same. For each vertex v, we keep track of the ray starting at V and passing through v, which makes, by a slight abuse of language, our new visibility tangent. For each vertex (or VT, since vertices and VTs are in bijection) we also keep track of its *hit-item* and of the type of the vertex. Here, the type of a VT is a bit more complex: Figure 4 shows how we name the type of a vertex depending on the position of both its adjacent edges relatively to the VT passing through it.

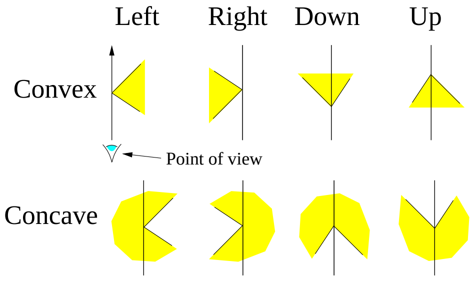

Figure 4: Various types of vertices

Note that the *hit-item* of a vertex v can be irrelevant (and even wrong) if the part of its VT beyond v goes through the interior of the adjacent polygon of v. It is yet correctly updated when the VT gets anew in free space.

The certificates that must stay true are the same as in the case of convex smooth obstacles: we schedule a crossing event when two consecutive vertices (which are kept sorted in counter-clockwise order around the view point) get aligned with the observer V. In Section 2, both VTs of the same obstacle could not cross each other. This is not the case here, since two consecutive vertices (consecutive in the cyclic order and on a polygon boundary) can get aligned with V.

Thus, we have two kinds of update when a certificate fails. If the certificate concerns vertices consecutive on the border of a polygon, then we have to update their type, and possibly their *hit-items* and the VP; this again, is done in constant time. In other cases, the update is similar to those in Section 2, with some more cases because we have to take into account other types for a vertex, namely *Up* and *Down*.

Figure 5 shows how the type of a vertex changes when V crosses the supporting line of one of its adjacent edges. Edges of a polygon are oriented so that the inside of the polygon lies to the left of its edges. When the crossing occurs, one vertex is nearer to V than the other: it will be said *near*, and the other, *far*; one vertex is following the other on the polygon's boundary: it will be said *next*, and the other, *prev*. This is the terminology used in Figure 5 to decide which transition we should target.

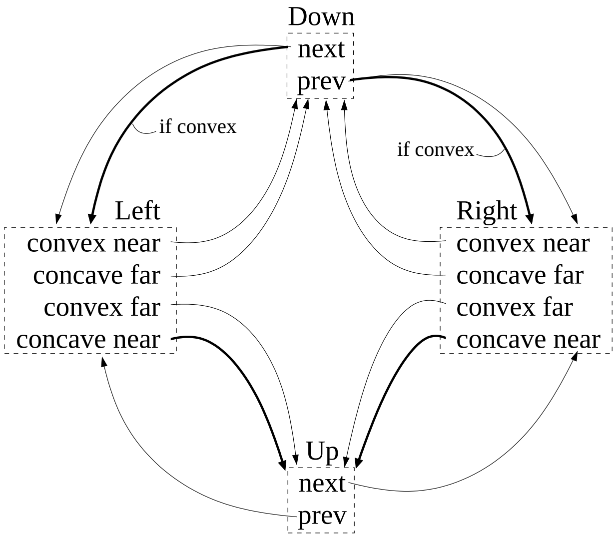

Figure 5: Updating a vertex's type. Fat arrows represent transitions where *hit-item* must be updated

The complexity of the structure for polygonal objects is the same as for convex smooth objects.

Hall-Holt proposes a refinement of this method, which processes exactly as many events as there are changes in the VP. The cost of processing one event is larger, but for the same motion of all items (obstacles and the viewpoint), his algorithm is less costly in time because of a relaxation on the constraints on the shape of the scene decomposition. However, he designed his algorithm only for convex obstacles.

The algorithm proposed by Hall-Holt can in fact be adapted to simple polygonal obstacles using a radial decomposition of the polygonal scene where each edge of a polygon is considered separated of the others, so that the radial segments would lie “ in ” the polygons as well, and not only in free space.

4. CONCLUSION AND FUTURE WORK

We have presented a simple kinetic data structure that maintains the visibility polygon of a moving point in a planar scene of moving obstacles (convex-smooth or simply-polygonal). A change in the visibility polygon is processed in constant time. The size of the structure is optimal (linear in the size of the scene). However it processes too many events: among all the events processed, lots can have no effect on the VP. However the number of events processed remains optimal if the scene is sparse, because nearly all obstacles become visible.

These algorithms could perhaps be accelerated by representing polygons with various level of details (perhaps even aggregating polygons that are closed to each other and far away from the view point), and using a sufficiency criteria to increment or decrement the LOD for some (groups of) polygons.

We may also want to describe a 3-dimensional visibility polyhedron, and extend it to the KDS framework. This looks much more difficult than in the 2d case.

5. REFERENCES

[1] J. Basch. *Kinetic Data Structures.* PhD thesis, Stanford University, June 1999.

[2] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th Symposium on Discrete Algorithms (SODA'97)*, pages 747–756, 1997.

[3] O. Hall-Holt. Kinetic visible set maintenance in the plane. submitted for publication, 2001.

[4] O. Hall-Holt and S. Rusinkiewicz. Visible zone maintenance for real-time occlusion culling. submitted for publication, 2001.