Dynamic Visibility of a Point in a Polygon

Karl Cronburg
Tufts University
karl@cs.tufts.edu

Abstract

In this work we present a time-optimal algorithm for maintaining the visibility polygon of a point in a polygon in the presence of small point movements. The work of [1] maintains a space complexity of $O(n)$ (n input polygon vertices) and describes their data structures in the dual. In contrast we spare no expense with space, and instead look at what inherent time complexity limitations arise from the dynamic visibility problem.

Keywords Visibility, polygon, geometry

1. Introduction

Given effectively unlimited precomputation time, and space we show that the visibility of a point inside a simple polygon reduces to (at runtime) the problem of detecting segment intersection with a much smaller convex polygon.

The results of this work are directly applicable to the implementation of games and simulations involving the continuous motion of a point through a closed and finite partition of a 2-dimensional space. For example one might have a particle simulation involving at least $p \in \Omega(n^2)$ particles (n polygon vertices). In this situation, the naiive worst case space complexity requires computing $O(n^2)$ (mostly) distinct visibility polygons.

Furthermore, many dynamic visibility applications realistically have the self-imposed limitation of small point velocities. Relatively low velocities lends itself to amortizing the cost of computing the next segment intersection to cause a *change* in the visibility polygon. By *change* we mean:

- A visibility *window* merges with an edge in the polygon (Figure 1).
- A visibility *window* passes over a vertex in the polygon, changing which edge it lands on (Figure 2).
- A *window* is otherwise created or removed / an edge becomes visibile or invisible (Figure 3).

All of these cases are defined by our point crossing over an infinite ray extending from an edge on a reflex vertex, pointing into the polygon (Figure **??**). As we know from the algorithm for computing the kernel of a polygon, these rays intersect with each other to form a partitioning of the closed space inside our polygon into convex subpolygons. In the *Algorithm* section we discuss how further decomposing these convex subpolygons into Delaunay triangulations allows us to detect visibility changes in constant time.

2. Algorithm & Data Structures

The goal of this algorithm is to have average case constant time visibility polygon updates in the presence of a moving visibility point. This `UpdateVisibility` pseudocode gives a top-level view of what the algorithm does:

```
def UpdateVisibility(point): # O(1)
  move = Edge(point, point.nextPosition):
  # O(1) neighbors
  for neighbor in point.currDelaunay.neighbors:
    if Intersect(move, neighbor):
      return neighbor
  return point.currDelaunay
```

This assumes we have a graph of Delaunay triangles, each of which have associated with them $O(1)$ instructions for how to compute each of the windows in their visibility polygon given the location of a point inside the DT.

2.1 UpdateVisibility Details

In `UpdateVisibility` we are finding the Delaunay triangle (DT) that our point is transitioning into by checking which neighboring DT intersects with the point's segment of movement.

When the point moves from one DT to another we have one of two cases:

1. The new DT is inside the same convex *visibility region* (see Figure 4). In this case we need only recompute where

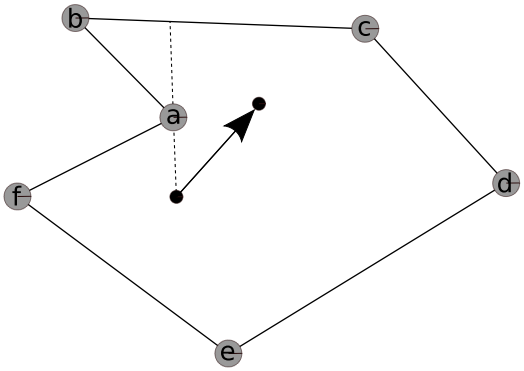

Figure 1. The visibility window (dashed line) passing through polygon vertex a merges with the line segment $\overline{ab}$; i.e. the segment $\overline{ab}$ goes from being invisible to visible. Because of our general position and small motion assumptions, no two windows can change at the same time.

the window points lay on the same edge they were on before. This can be done in $O(w)$ for w window edges in the visibility region, which is necessarily $O(w) \in O(v)$ for v visibility polygon vertices. This is optimal because we must at least read off the v vertices in the visibility solution. If on the other hand we did not need the visibility polygon for a particular time step, the runtime is $O(1)$ to move the point in our Delaunay graph.

2. The new DT crosses a convex visibility region edge into a new region. Given general position, this transition requires the updating of exactly one edge / window in the visibility polygon which can be done online in $O(1)$. Again as in case 1, we also recalculate where the windows land on their respective polygon edges in $O(w)$ time.

2.2 Delaunay Justification

We use DTs to ensure that we look at a constant number of edges (neighboring DTs) for each visibility update. If we had used the entire convex partitions of our polygon, we would be stuck with an $O(\log n)$ average case as required by the visibility complex of [1]. In the Section 3 we discuss possible theories for improving the time bound to be $\theta(1)$ in all cases.

In order to get the DTs and build a meaningful graph of visibility polygons from them, we have to first partition our polygon into convex *visibility regions* (Figure 4). Top-level code for this looks like:

```
def PrecomputeVisibility(polygon):
  delaunays = []
  visGraph  = VisibilityPartition(polygon)
  for convexPoly in visGraph:
```

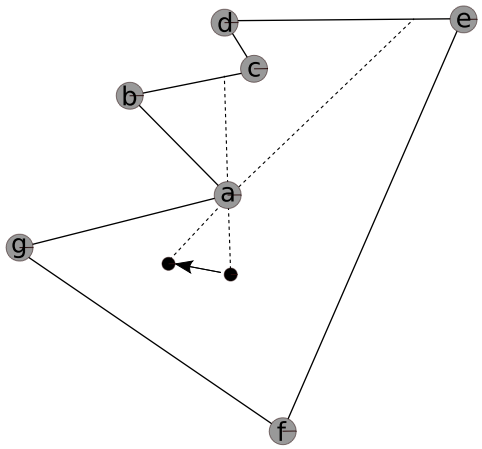

Figure 2. The window in this figure is initially passing through a and landing on the polygonal edge $\overline{bc}$. When we then move the point a short distance to the left, the window passes through point c now landing on edge $\overline{de}$.

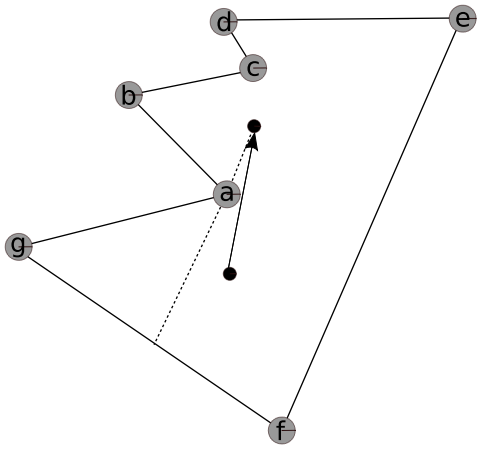

Figure 3. Example point movement where a window is created when edge $\overline{ag}$ goes invisible.

```
    delaunays.append(DelaunayTriang(convexPoly))
  ds = LinkNeighbors(delaunays)
  ComputeVisibilities(polygon, ds)
  return ds
```

VisibilityPartition computes a list of convex polygons like the ones shown in Figure 4. These visibility regions are then further partitioned into their Delaunay Triangulations in the *DelaunayTriang* method. Next we link neighboring DTs in *LinkNeighbors* forming a Delaunay graph. Finally *ComputeVisibilities* informs each of the nodes (DTs) in the Delaunay graph how to compute the visibility polygon of a point inside of it:

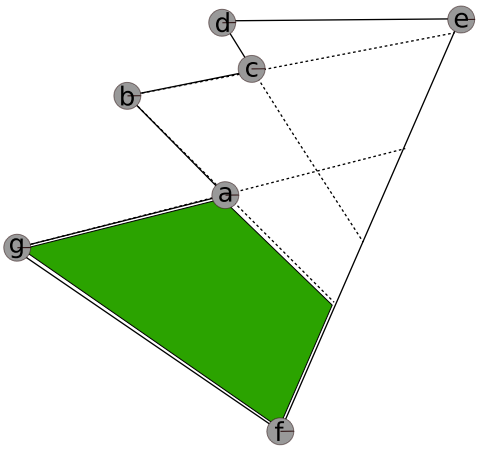

Figure 4. Example decomposition of a polygon into convex *visibility regions*. Highlighted (in green) is one such region. Transitioning from one region to another neighboring region changes at most the visibility of one edge or which edge a window lands on.

```
def ComputeVisibilities(polygon, ds):
  for d in ds:
    ws  = ComputeWindows(polygon, ds)
    vs= VisibleEdges(polygon, ds)
    d.visEdges = RadialSort(d, ws, vs)
    def GetVisPoly(position, d=d):
      visPoly = []
      for e in visEdges:
        if isWindow(e):
          AddWindow(visPoly, point, e)
        else: visPoly.append(e)
      return visPoly
    d.getVisPoly = GetVisPoly
```

3. Future

The core limitation to updating a visibility polygon of $O(v)$ size in $O(1)$ time is that to actually observe the solution necessarily requires $O(w)$ time to compute the window point intersections where in the worst case $w \in O(v)$. Certainly then if the solution is required frequently relative to the distance the visibility travels, any dynamic visibility algorithm degenerates to a best possible case $O(w)$ runtime.

3.1 Practical Improvements

3.2 Example Use Case

We have implemented (in a github repository - see [2]) a simple app allowing a user to control the motion of a point through a polygon.

As the basis for an *art-gallery-guard* game, this app would require low-latency in recomputing the visibility polygon as a guard moves around the polygon. Our *Updat-*

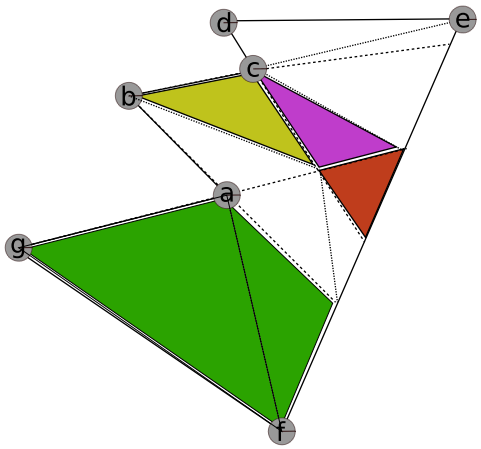

Figure 5. Further decomposition of the visibility regions of Figure 4 into their respective Delaunay Triangulations. One such DT is highlighted in red which has two neighboring DTs. Note that a point movement from the red to yellow DT is possible, but this can be handled in constant time as two transitions using the purple DT as an intermediary.

eVisibility algorithm becomes even more applicable when the game becomes multiplayer and the app needs to compute a visibility polygon for each player. As the basis for a multi-particle simulation the same logic applies - we trade off memory usage in favor of constant-time low latency solutions.

References

[1] Stepahne Rivire. Dynamic Visibility in Polygonal Scenes With the Visibility Complex. In *SCG Proceedings of the thirteenth annual symposium on Computational geometry*.

[2] Karl Cronburg Visibility polygon implementation: https://github.com/cronburg/geo163