Computational Geometry: Project

Halut Celestin, Silva Capitão Dinis,Vervaet Gwenaël

October 2021

In an era where simulations (whether for scientific or entertainment purposes) are becoming more and more realistic, the problem of their computability arises and it is therefore quite natural that a problem as basic as the computation of a field of view is addressed in the modern scientific literature. In this project we will discuss one of the latest methods to tackle simple visibility problems. We will use a scientific publication of 2015:

- Visibility and ray shooting queries in polygonal domains[1]

Visibility problems

This article divides a visibility query algorithm into two parts: preprocessing time and query time. It is particularly on preprocessing that the article will work, using multiple tools to create a new data structure representing the polygonal domain in a much more efficient way to process visibility queries. This particular data structure is called ”extended corridor structure” and separates the free space (F, the space with the obstacles P (the polygons) excluded) into three parts: bays, canals and an ocean (M) [3] (in reference with a classic corridor structure of polygonal domains already used in others papers [7]). Here is how to create this structure.

The first step is to triangulate the free space. It could be done by sweep line algorithm (video on the uv) (or by more basic algorithms as viewed during the classes). This should take $\mathcal{O}(n \log n)$ time, where n is the total number of vertices in all polygons. After we put in place a planar dual graph of this triangulation[1]. Then, we remove all nodes that have a degree less than two. We continue by removing all nodes of degree less than 3 but, we replace the node (and its two edges) by a single edge connecting the neighbours. The set of remaining nodes is the set of junction triangles. Junction triangles partition the free space into subpolygons.

Now, we need to compute bays, hourglasses and canals. We do all of them in a single process, we look at junction triangles and when we remove these triangles we get separate pieces of F, these spaces are called hourglasses. We can find all hourglasses in $\mathcal{O}(n)$ time thanks to the triangulation we did before. Each of them is thus delimited by four parts. Two are edges of the removed triangles (like the $\overline{be}$ and $\overline{af}$ segment on the figure 1) and the other two are the shortest paths between the vertices of these edges that are part of the same obstacle (like the black line connecting $\overline{ab}$ or $\overline{ef}$ on the figure 1). To calculate these shortest paths we can use naive approach that consists in creating a distance-weighted visibility graph for the simple polygon that is the space bounded by the two

[1]A dual graph of G is a graph that has a vertex for each face of G, so here, a vertex in each triangle, if two faces (triangles) have an edge in common, then both vertices corresponding to the faces in the dual are connected by an edge

obstacles and the two edges of the triangle junctions and then find the shortest path between the two vertices of the graph of interest. But this way is in $\Omega(n^2)$ time. An other manner is described in a paper [2] written in 1994 and reduced the time taken to $\mathcal{O}(n \log(n))$ by triangulating the polygon, making as previously a dual graph (which is a tree and thus have only one simple path between two vertices), reduce the potential vertices (included in the shortest path between two points x,y. See figure A for an example) to the vertices belonging to the triangles that form the simple path (between the triangles where x and y are included) and uses an enhanced algorithm for finding the shortest path by doing it incrementally.

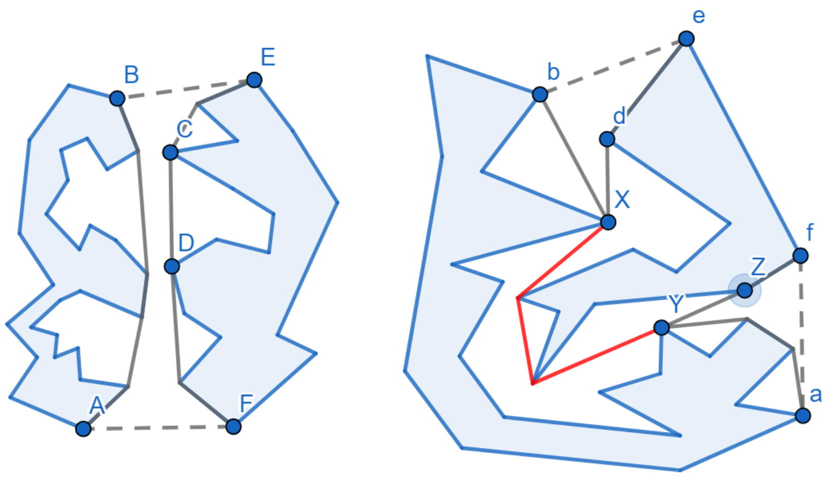

Figure 1: Computation of hourglass, open on the left, closed with a canal on the right

The hourglass can be open or closed. On one hand, if it is open, shortest paths never cross or merge. In this case, the hourglass delimits only bays, there is no canal. On the other hand if it is closed, it means that both join at some point and the shortest paths delimit canals and bays. Canals happen in the merged part of some shortest paths, and it is delimited by canals opening and closure. A canal opening (or closure) is located when the shortest paths (not merged yet) connect two different obstacles. A canal is defined as a simple polygon with an opening and a closure. A bay is defined as a polygon delimited by a single closure formed by a shortest path on its own polygon.The ocean is defined by the free space with exclusion of bays and canals. The computations made to define all these structures can be done in $\mathcal{O}(n + h \log^{1+\epsilon} h)$ where h is the number of obstacles, i.e. the size of P. Now that we have all those structures, we can ask the data structure : what is the visibility polygon of the point q :

1. the point q is in the ocean (M, the free space minus the bays and canals), we compute the visibility polygon of q in the ocean (Vis(q,M)). Then there are multiple scenarios:
   - Vis(q,M) does not intersect with any gate of bay or canal, then Vis(q) = Vis(q,M)
   - Vis(q,M) intersects with a bay gate, the visibility of q in the bay is a set of cones (one with no obstacle) created from q and segments crossed by Vis(q, M) (figure 2). Vis(q) = Vis(q,M) $\cup$ Cones in bay

- Vis(q,M) intersects with a canal gate. The approach is the same than the bay, but we check all the canal gates that intersect Vis(q) = Vis(q,M) ∪ Cones in the canal.

2. q is in a bay, we compute, Vis(q, Bay).
   - if Vis(q,Bay) does not intersect the boundary of the bay : Vis(q)=Vis(q,Bay)
   - if Vis(q,Bay) intersects with the gate, the outside of the bay is visible with a cone from the bay similarly to the above case, but reversed.
3. q is in a canal, we compute Vis(q,Canal), the result is the same as Bay, but we need to look at the two gates

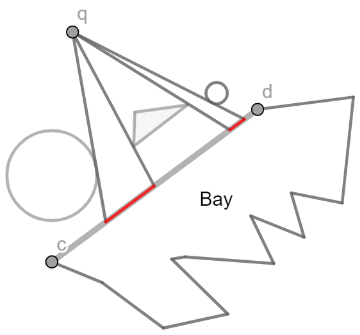

Figure 2: cone of view in a bay with bonus obstacles

But, now, how can we compute those Vis(q,X) ? To do this we will use the visibility complex. It can be created in $\mathcal{O}(n \log n + k)$ time where n is the number of disjoint convex obstacles and k is the output size, and only uses $\mathcal{O}(n)$ space. The visibility complex of a collection of n pairwise disjoint convex obstacles in the plane (our set P) can be considered as a subdivision of the set of free rays. Free rays is the set of rays whose origins are in the free space F (not in the obstacles). The visibility complex is a 2-dimensional regular cell complex where each cell corresponds to a collection of rays which have the same backward and forward views. We will not go deeper into what exactly is a 2-dimensional regular cell complex because it would take too much time and frankly, we are not qualified enough to talk about it, but we recommend to read [5] and [6] if you are interested in it. Thanks to the visibility complex we are now able to build a data structure of size $\mathcal{O}(n + h^2)$ in $\mathcal{O}(n + h^2 \log h)$ time that allows us to compute Vis(q,M) in $\mathcal{O}(|Vis(q, M)| + h' \log n)$ time for any point q in M. h' being the number of obstacles seen by q. The same can be said to compute visibility cones, hence the query time stays the same even when q is in a bay or a canal (this does not give us the visibility polygon inside the bay or the canal, it only gives us the visibility polygon inside M from the point q).

Now we will have to see how to obtain the visibility polygons of bays and canals. As for the ocean M, there is two types of queries when we consider bays and canals, the first type being when the point q is inside a bay or canal and the second type being when q is outside but sees a bay or canal through its gate. For the first type, we will use a quadratic-space data structure defined by Aronov and al in [4]. This is a simpler data structure that we can use in this case because we can consider the bay or canal where q is as a simple polygon viewed from q. The prepocessing time of this will be $\mathcal{O}(n^2 \log n)$ and it will take $\mathcal{O}(n^2)$ space for all bays and canals. With this any Vis(q,Bay)/Vis(q,Canal) can be computed in $\mathcal{O}(\log^2 n + |P|)$. For the second type of queries, we will have to do something a little bit more complex. First, we will define BC(p) as being a set of all bays and canals that have a gate lying on the boundary of a given obstacle p ϵ P. Now, let's say Vis(q, BC(p)) is the union of the visibility polygons of q in all bays and canals of BC(p) ignoring any other obstacle in P (as if they were transparent to q). Let's say C_q is a cone with apex q i.e. a cone through which q is visible to one or more portions of p. Finally we obtain Vis(C_q, BC(p)) which is the intersection between the space covered by the cone C_q and Vis(q,BC(p)). Thanks to all this we can use the same data structure we used for the first type. We can do this for each obstacle p ϵ P in $\mathcal{O}(m^2 \log m)$ time and $\mathcal{O}(m^2)$ space, where m is the total number of vertices in BC(p). We can also obtain Vis(C_q, BC(p)) in $\mathcal{O}(\log m + |Vis(C_q, BC(p))|)$ time. To do this for all bays and canals we get a prepocessing time of $\mathcal{O}(n^2 \log n)$ and we need $\mathcal{O}(n^2)$ space. Now, when we would like to make a query from a point q in M, after we found the visibility polygon in M, we will then see for each obstacle p ϵ P that q sees if there is not an other obstacle that partially blocks the view, i.e., there are some cones through which q is visible to one or more portions of p. For each of those cones, we compute the Vis(C_q, BC(p)). Then we finally have Vis(q).

We can do a generalisation of all this to make cone visibility queries. A cone visibility query is that given a query cone, compute the visibility polygon of the cone apex in the region bounded by the cone [1]. If q is in a bay or a canal, we can use the same data structure we used in the bay/canal case of the visibility query since we are still in a simple polygon (more restricted because of the cone, but still a simple polygon). When q ϵ M, then it is more difficult and once again, we have to use the visibility complex. In this case, we are in a more complicated case than the simple visibility query because the boundaries of the visibility polygon are not only segments of P. Indeed, in the visibility query, all the boundaries are either a segment of an obstacle or a boundary of the free space, but in the cone visibility query we would have up to two "imaginary" boundaries that are made by the rays defining the cone. To solve this problem we will have to add the ray from q to the first obstacle hit by it, i.e. the boundary of the cone, and the ray shooting in the opposite direction as a new "obstacle" to the visibility complex. To do this, we need to add a new data structure to make ray shooting, but the difference in time and space do not change a lot compared to the simple visibility query problem.

Conclusion

This algorithm requires a lot of preprocessing but is optimal in the space used. The query time is very short too and is clearly way faster than a brute force approach to the same problem. We present below the final theorem that summarises the overall performance of the presented data structure.

For a polygonal domain P, we can build a data structure of size $\mathcal{O}(n^2)$ in $\mathcal{O}(n^2 \log n)$ preprocessing time that can answer each visibility query in $\mathcal{O}(\log^2 n + k + min(k, h) \log n)$ time. [1]

Work done

To illustrate those concepts, we programmed a web page that allows you to watch the five main steps of the preprocessing. Then we made a video game showing what we can do with those kind of algorithms. The first part is mainly hardcoded, we computed by hand the algorithm result and encoded it in the source code. It is only to illustrate the algorithm steps. The game works with a much more simpler algorithm than the one we present. It's a brute force algorithm that works in $\mathcal{O}(n^2)$.It works in two steps. First step, we find the point of the visibility polygon by asking all vertices if they are visible. If they are visible, we ensure that they are not tangent. If a vertex is tangent, we need to compute the projection of it to another polygon or the frame. Once we have the set of all the points that will be in the visibility polygon, the second step begins. We sort all the points by angle and we recreate the polygon with two rules :

1. If you are on a projection point/ tangent, (there is always a connection between the tangent and the projection point in the visibility graph) the next point is the corresponding tangent/projection point that creates/is created by the point.

2. If you are not in the first case, then take the first vertex laying on the same polygon and adjacent to the point in the angle-sorted list (you cannot take a point that is already on the path)

The game is quite simple, the only thing you have to do is to find a frog in a maze. It appears only when it lays in the visibility polygon computed from the player's position. When you catch the frog you pass to the next level (there are 3 levels).

A Appendix

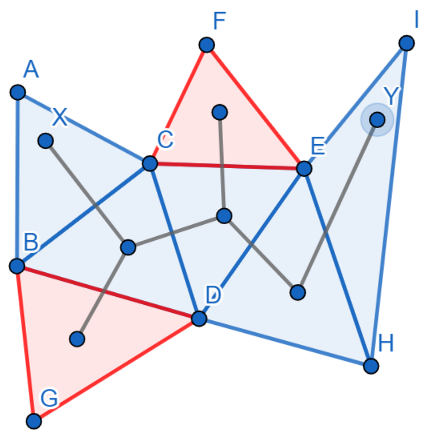

Figure 3: Dual graph reduction of a triangulated polygon

Link to bibliography articles

- Visibility and ray shooting queries in polygonal domains[1] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1cfj0qe/cdi_gale_infotracacademiconefile_A385426356`
- Euclidean Shortest Paths in the presence of rectilinear barriers[2] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1luu2m3/alma991009525171204066`
- Computing the visibility polygon of an island in a polygonal domain[3] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1cfj0qe/cdi_springer_primary_2015_453_77_1_58`
- Visibility queries and maintenance in simple polygons[4] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1cfj0qe/cdi_crossref_primary_10_1007_s00454_001_0089_9`
- Topologically sweeping visibility complexes via pseudotriangulations[5] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1cfj0qe/cdi_springer_primary_1996_454_16_4_BF02712876`
- The visibility complex[6] : `http://dx.doi.org/10.1145/160985.161159` [Not free]
- An efficient algorithm for Euclidean shortestpaths among polygonal obstacles in the plane[7] : `https://cibleplus.ulb.ac.be/permalink/32ULDB_U_INST/1cfj0qe/cdi_crossref_primary_10_1007_PL00009323`

References

[1] Danny Z. Chen and Haitao Wang. "Visibility and ray shooting queries in polygonal domains". In: *Computational Geometry* 48.2 (2015), pp. 31–41. ISSN: 0925-7721. DOI: https://doi.org/10.1016/j.comgeo.2014.08.003. URL: https://www.sciencedirect.com/science/article/pii/S0925772114000777.

[2] FP Preparata DT.Lee. "Euclidean Shortest Paths in the presence of rectilinear barriers". In: *Networks* 14 (1984), pp. 393–410.

[3] H Wang DZ Chen. "Computing the visibility polygon of an island in a polygonal domain". In: *Proc. of the 19th European Symposium on Algorithm(ESA)* (2011), pp. 481–492.

[4] Aronov B. Guibas L. J. Teichmann M. Zhang L. "Visibility queries and maintenance in simple polygons". In: *Discrete and Computational Geometry* 27(4) (2002), pp. 461–483.

[5] G. Vegter M. Pocchiola. "Topologically sweeping visibility complexes via pseudotriangulations". In: *Discrete and Computational Geometry* 16 (1996), pp. 419–453.

[6] Vegter G. Pocchiola M. "The visibility complex". In: *International Journal of Computational Geometry and Applications* 6(3) (1996), pp. 279–308.

[7] J.S.B. Mitchell S. Kapoor S.N. Maheshwari. "An efficient algorithm for Euclidean shortest paths among polygonal obstacles in the plane". In: *Discrete Computanional geometry 18(4)* (1997), pp. 377–383.