Step by step calculating visibility of a polygon from a point using Triangular Expansion method

This deep dive started with @AmitP (Amit Patel) and
his legendary site RedBlobGames which has taught me a lot over the years..
Credit to this article for explaining the visibility problem very well.
Uses a different solution but links to the paper implemented here.
https://www.redblobgames.com/articles/visibility/https://www.redblobgames.com/articles/visibility/

This is an implementation of the Triangular Expansion method from this paper
https://scholar.google.com/scholar?cluster=11179288360296920832
“Efficient computation of visibility polygons.”
Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, Alexander Kroller
March 18, 2014

Before you go down the same rabbit hole as me: The paper has already been implemented in CGAL (C++)
https://doc.cgal.org/latest/Visibility_2/

There is also a C# wrapper of CGALDotNet
https://github.com/Scrawk/CGALDotNet/wiki/Introduction
https://github.com/Scrawk/CGALDotNet
(No nuget so far)

At this point the sensible choice would be to implement the CGAL library and get their solution
with all the optimisations from the experts etc

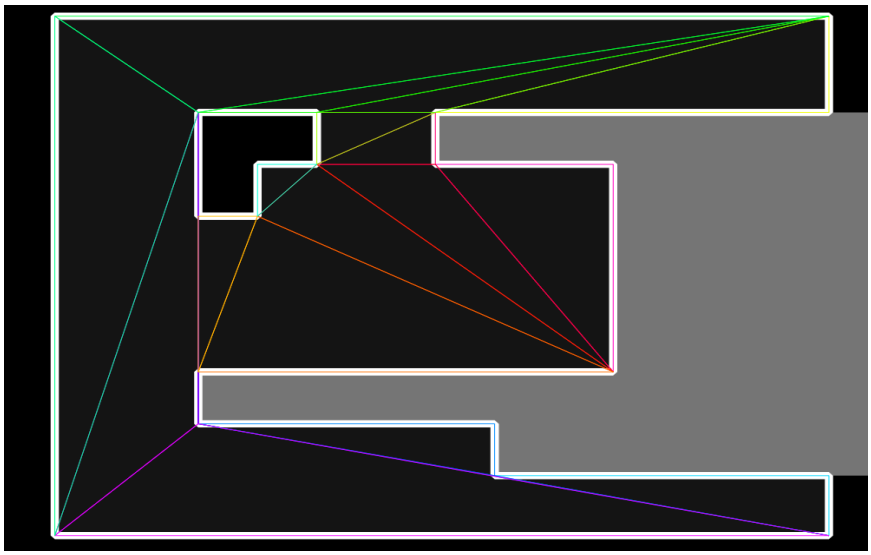
But instead I will try to implement the triangular expansion method myself, certainly not as well.
Mostly to learn more about how it works.
This document is my own step by step guide trying to understand the algorithm from the research paper.

Some other interesting approaches to line of sight calcuation on grids
http://www.roguebasin.com/index.php?title=Field_of_Vision
http://www.adammil.net/blog/v125_Roguelike_Vision_Algorithms.html
https://www.albertford.com/shadowcasting/

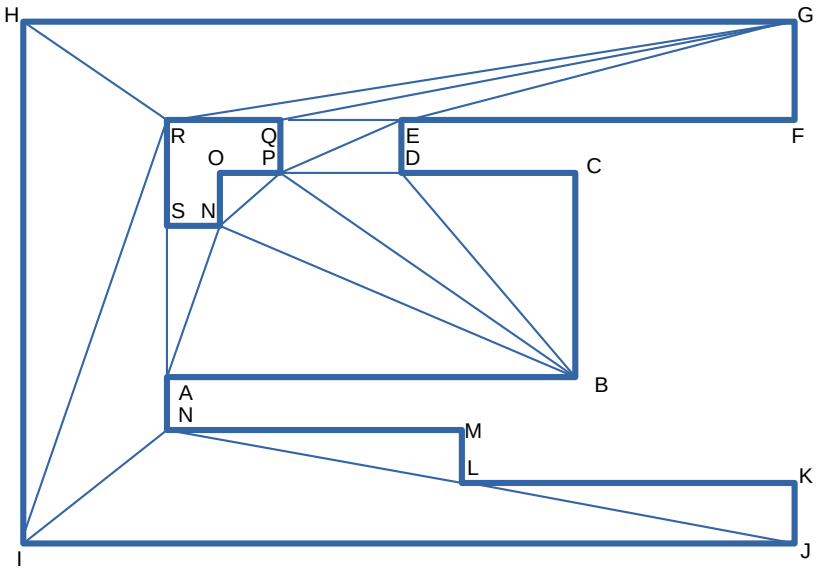
Goal:
Given an area bounded by a polygon, which may have holes.
Calculate what is visible of that polygon from a single point.
The result should be another polygon representing the visible area.

Usecases:
The bounded area typically represents a valid area for an agent to move
and the edges of the polygon are walls or other obstructions.
Useful for lighting, line of sight and AI related calculations.
For example you could test an AI agent is within the line of sight area allowing it to attack.
You can also difference another polygon over the top to create EG a cone shaped of line of sight.

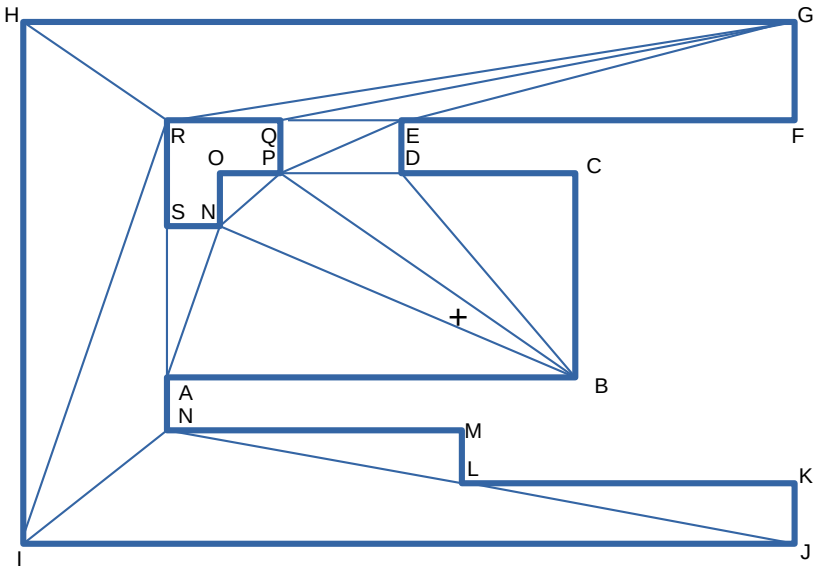
The polygon area, real usecase from an app under development



1. Triangulate the polygon. There are lots of standard triangulation methods out there, I'm gonna use LibTessDotNet. Points are labelled for reference.

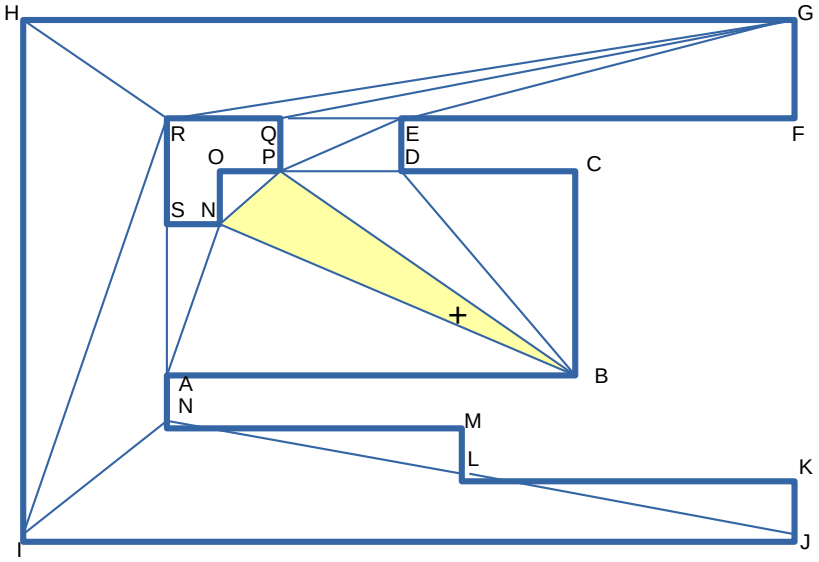


2. Point + is the eye point we want to check visibility from



3. The result we want in the end is a list of points that make up triangles in the visibility polygon.
We start by finding which triangle the eye point is in and adding those points to the result list. In this case BPN.
Because no other triangle can be obscuring them the three points making up this starting triangle can always be seen.
We also add all three edges to our list of sides to explore further,

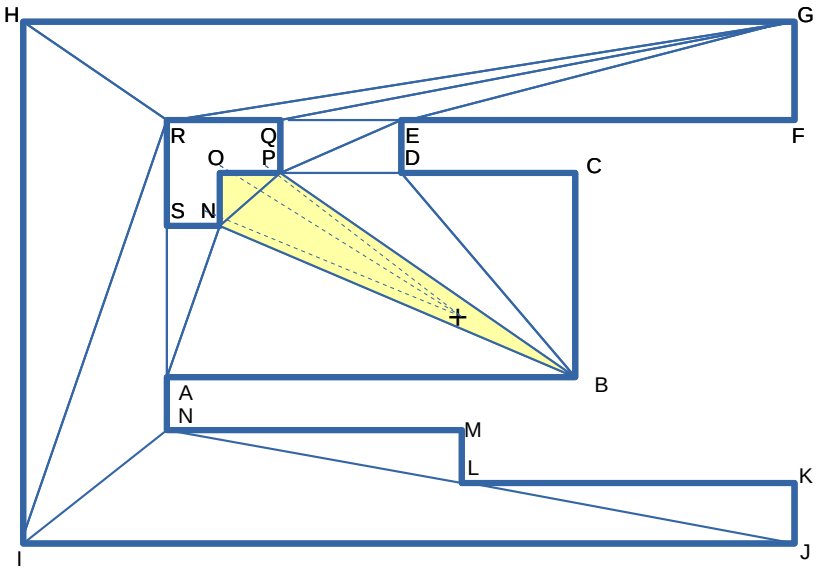
Results
BPN



Edges To Explore
BP, PN, NB

4. For each edge in the explore list we check if its connected to a further triangle. CASE: EDGE IS CONNECTED
In this case PN is connected to O.
We now need to check if the eye point can see O. How do we do that? We check the angle of P, N and O to the eye.
As long as O falls in the middle the entire triangle can be seen. CASE: THIRD POINT VISIBLE
So we add both new edges to the result list and cross off the edge we explored.
Because PN is already on the result list w just add O, in the correct pos between N and P

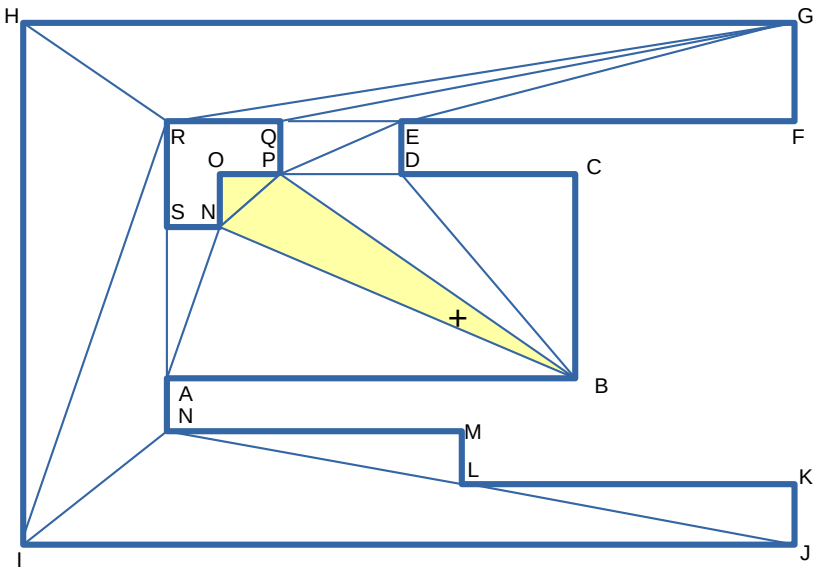
Results
BNOP



Edges To Explore
BP, ~~PN~~, NB, OP, ON

5. We check the next edges, if we check OP and ON we find they are not connected to another triangle
We simply cross them off the list.. CASE: EDGE IS NOT CONNECTED

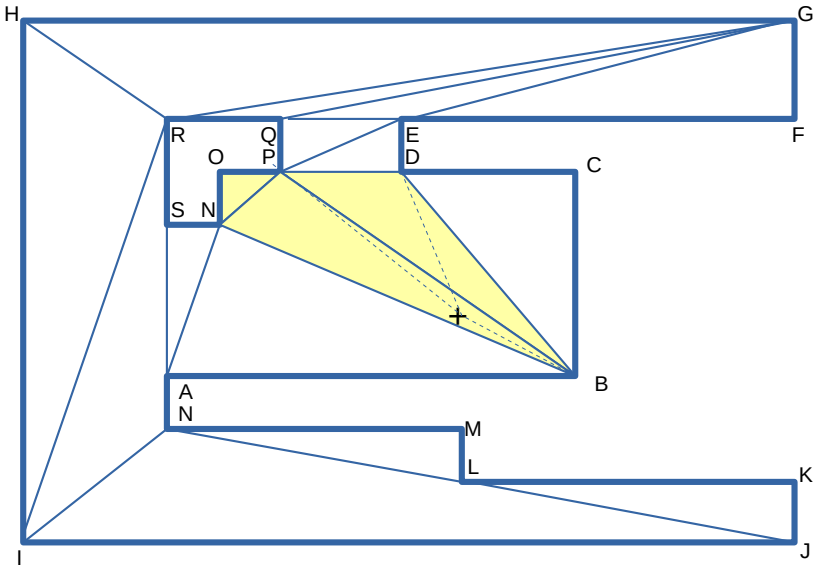
Results
BNOP



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~

6. Next we check BP from the explore list. This is connected to D (CASE: EDGE IS CONNECTED)
and angle of D is between P and B (CASE: THIRD POINT VISIBLE)
so the triangle is completely visible and added to the list, as well as its edges to the explore list.

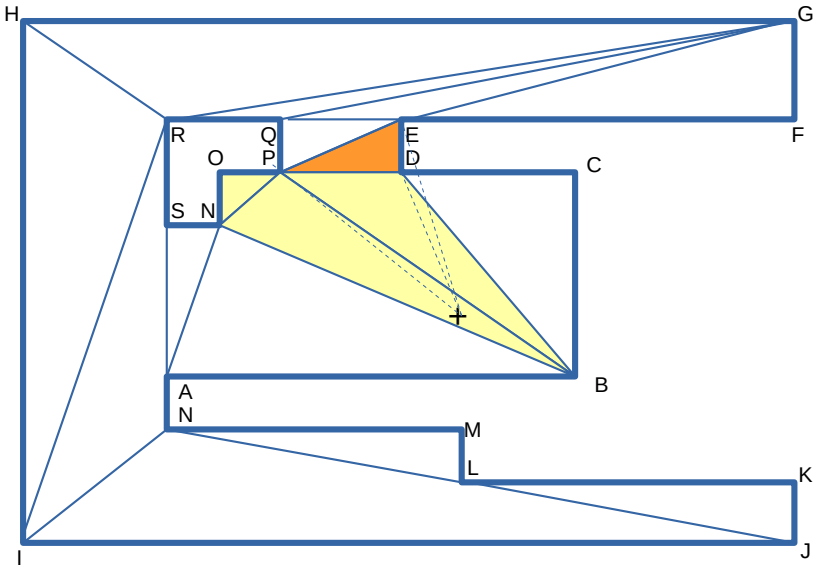
Results
BNOPD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, PD

7. Next we check PD from the explore list. This is connected to E (CASE: EDGE IS CONNECTED)
This time E is not visible from the eye point, its to the angular right of D. (CASE: THIRD POINT > RIGHT)
We dont add anything to the results list
On the Edge list we -do not need to add ED as that edge is totally out of view. (dont add the exceeded point pair)
-We add PE but make a note that E is constrained by D. (noted here as PE(D))

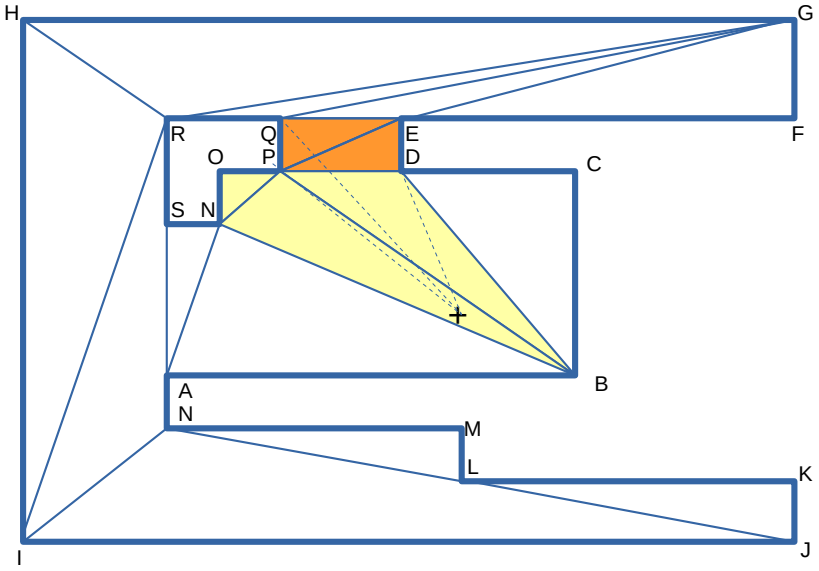
Results
BNOPD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD,~~ PE(D)

8. Next we check PE(D) from the explore list. This is connected to Q (CASE: EDGE IS CONNECTED)
When we check the third point (Q) we check between P and D because we noted already E is constrained by D.
Q is between P and D so (CASE: THIRD POINT VISIBLE)
We add Q to the results list.
On the Edge list we add the edges but continue to note E is constrained by D, inheriting that from the input.

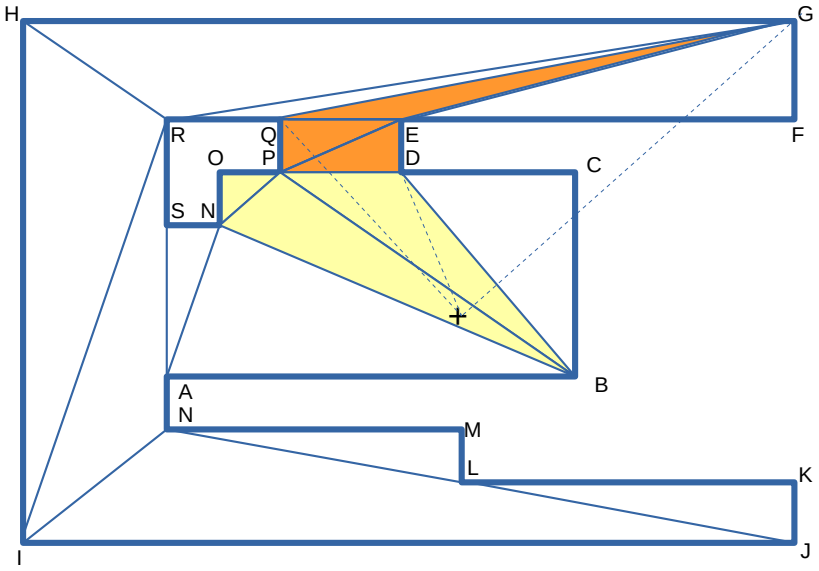
Results
BNOPQD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ, QE(D)

9. Next we check QE(D) from the explore list. This is connected to G (CASE: EDGE IS CONNECTED)
When we check the third point (G) we check between Q and D because we noted already E is constrained by D.
G is right of D (CASE: THIRD POINT > RIGHT)
We do not add to the results list
On the edge list we only need to add the result connecting the point that wasn't exceeded, so just QG,
noting G is constrained by D.

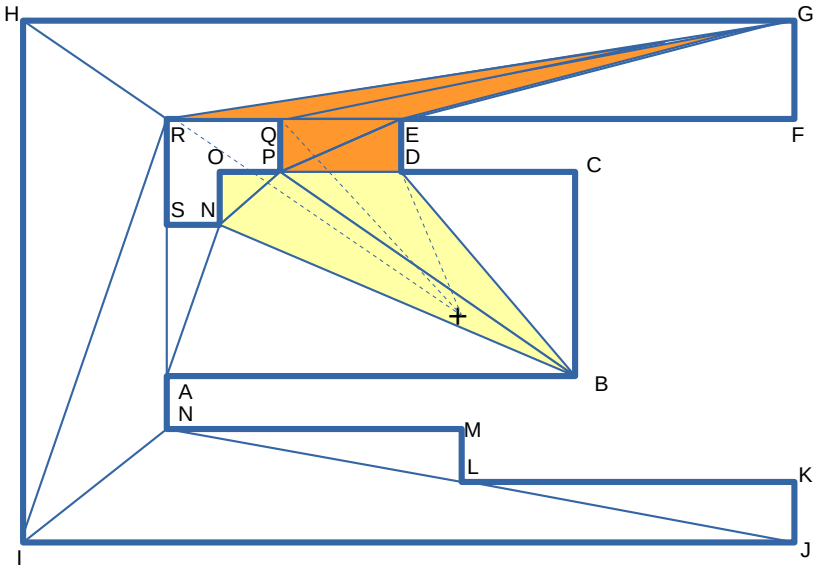
Results
BNOPQD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ,
~~QE(D)~~, QG(D)

10. Next we check QG(D) from the explore list. This is connected to R (CASE: EDGE IS CONNECTED)
When we check the third point (R) we check between Q and D because we noted already G is constrained by D.
R is left of Q (CASE: THIRD POINT < LEFT)
Dont add to results list.
On the edge list we only need to add the result connecting the point that wasn't exceeded, so just GR,
noting both G and R are constrained so G(D)R(Q)

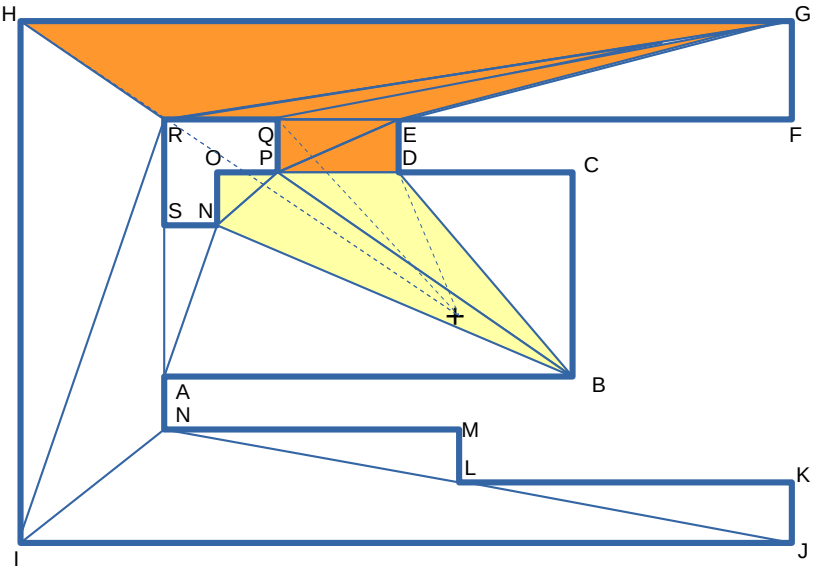
Results
BNOPQD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ,
~~QE(D)~~, ~~QG(D),~~
G(D)R(Q)

11. Next we check G(D)R(Q) from the explore list. This is connected to H (CASE: EDGE IS CONNECTED)
When we check the third point (H) we check between Q and D because we noted already G is constrained by D.
And R is constrained by Q (CASE: THIRD POINT < LEFT)
Do not add to results list
On the edge list we only need to add the result connecting the point that wasn't exceeded, so just GH,
noting both G and H are constrained so G(D)H(Q)

Results (in view)
BNOPQD

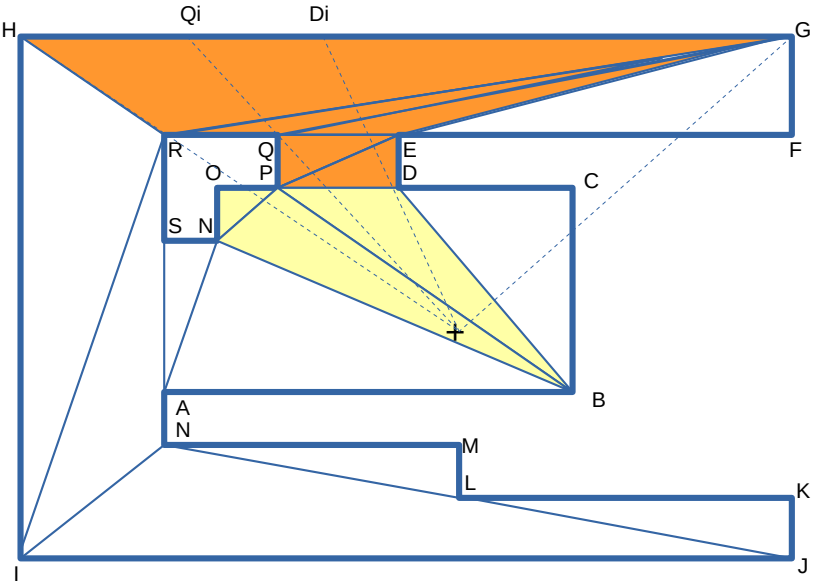


Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ,
~~QE(D)~~, ~~QG(D),~~
~~G(D)R(Q),~~
G(D)H(Q)

12. Next we check G(D)H(Q) from the explore list. This is not connected,
(CASE: EDGE IS NOT CONNECTED)
BUT there are constraints from earlier points, we have to solve the constraints now no new point is carried over.
(CASE: SOLVE LEFT CONSTRAINT) (CASE: SOLVE RIGHT CONSTRAINT)
In this case we calculate both the constraints as projected intersections in the line GH.
Written as Qi Di
We add these intersection as NEW points to the result list with new indices.

When adding these new intersection points to the result list we have to determine if they go
after their constraint point or before.
We can check that by checking the angle of intersection point against the original edge point it's constraining.
In this case Di against G. (G constrained by (D) was in this edge definition)
If the shortest angular distance between Di and G is clockwise than Di goes before D.
For QI and H the shortest angular distance is counterclockwise so QI goes before Q.
This relies on the total result polygon being clockwise.
I think… If your polygon winding is counterclockwise than it's the reverse.

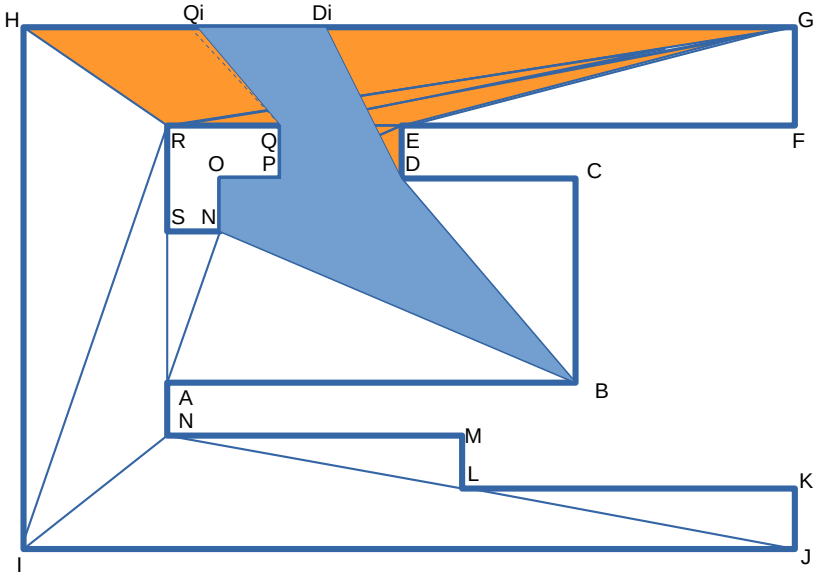
Results
BNOPQQiDiD,



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ,
~~QE(D)~~, ~~QG(D),~~
~~G(D)R(Q),~~
G(D)H(Q)

13. Normally you continue until all edges processed.
We now have all the rules so lets see what happens if we finish now
Last step is take the result points and build a polygon.

Results
BNOPQQiDiD



Edges To Explore
BP, ~~PN~~, NB, ~~OP, ON~~
BD, ~~PD, PE(D),~~ PQ, ~~QE(D)~~, ~~QG(D),~~ ~~G(D)R(Q),~~ ~~G(D)H(Q)~~