# Scenario Week 4 Implementation Report – Group Xerus

## 1. Tools used:

- Language: JavaScript (computation) & HTML canvas (visualization)
- Libraries: General Polygon Clipper (GPC) for JavaScript

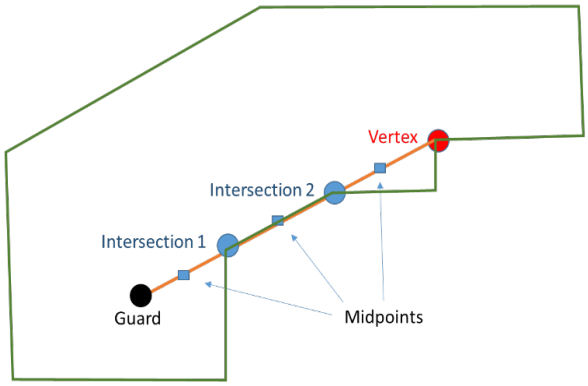
### 1.1. Auxiliary Algorithms:

**Auxiliary algorithm #1: Finding visibility polygons for each guard.**

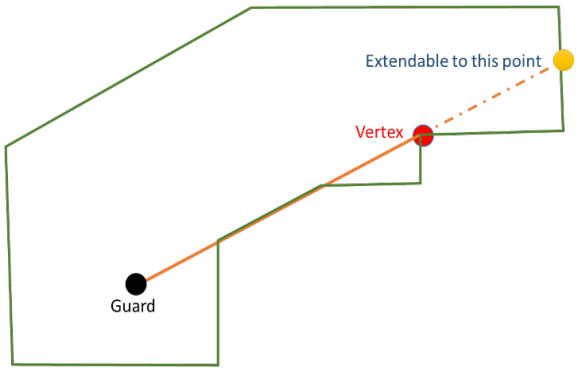
In order to find the visible regions of a polygon we created an algorithm that takes in a polygon object and a list containing the position of the guards. Below is a brief description of the algorithm

- Take in a Polygon object and a list consisting of the position of the guards. The polygon objects attributes consist of a list of its vertices and lines.
- Create an array to store the visibility polygons for every guard.
- Iterate through the guards and for each guard find the reachable vertices, then find the points that are reachable beyond them if there happen to be any:
    - Find the vertices that are visible/reachable:
        1. For each vertex, create a line segment from the guard to the vertex
        2. Find all the intersection points on this line segment with any line segment from the polygon
        3. Sort these intersection points by the distance they are away from the guard – in ascending order.
        4. Iterate through these intersection points in the sorted order and see if the consecutive midpoints lie in the polygon – if any of them do not, the vertex is not visible. Otherwise, the vertex is visible and is pushed to the guard's list of reachable vertices.

           *Orange line is the guard to vertex line segment created in step 1.



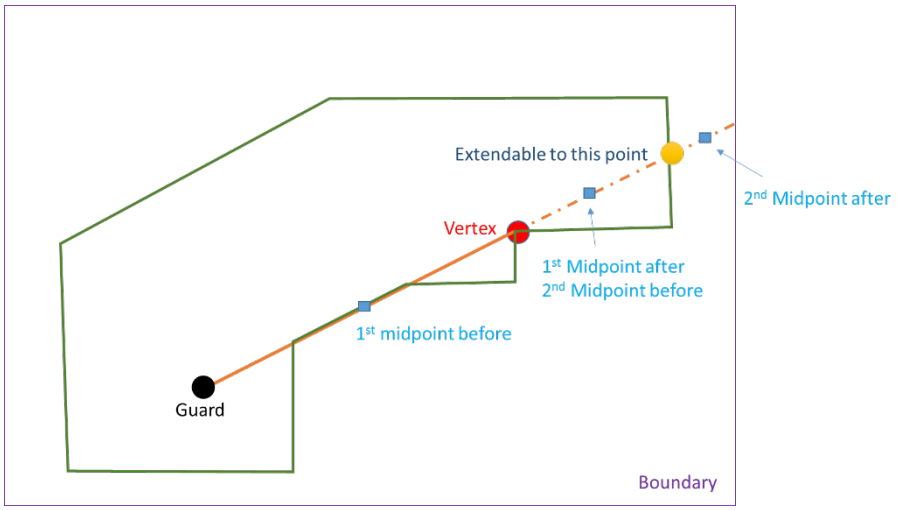

    - For every reachable vertex, check if you can extend the ray from the guard beyond the reachable vertex to an edge on the polygon.




1. Find the gradient of the line between the guard and the vertex
2. Use this to create a line from the vertex to a point beyond the boundary of the polygon.
3. Find the intersection points on the line
4. Sort these intersection points in ascending order

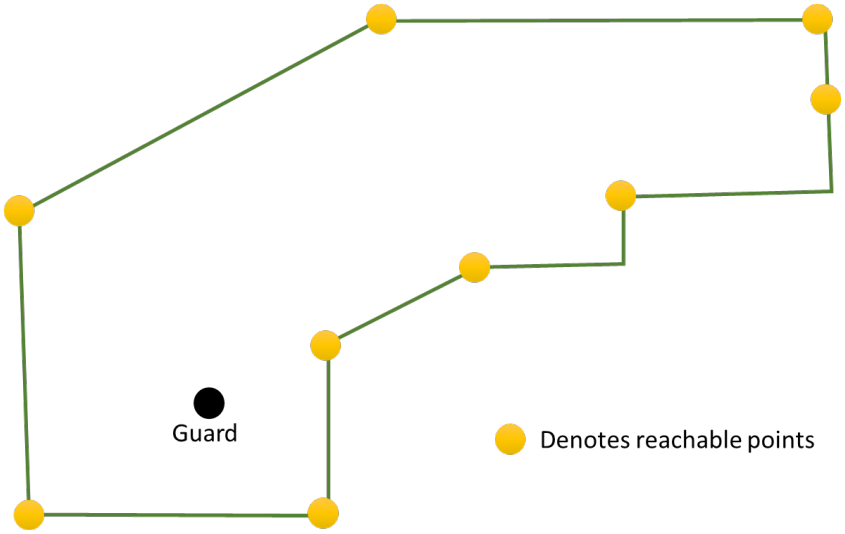
- Have a list with the intersection points in sorted order with the guard and the vertex then inserted as the head of the list.
- Iterate through the list to see if the midpoint of the $i^{th}$ and $i+1^{th}$ index and $i+1^{th}$ and $i+2^{th}$ index both lie in the polygon. If either of them do not, then the point at the $i+1^{th}$ index is the furthest the ray can be extended to. Otherwise continue to the next iteration of the for-loop.

  For instance, when considering the vertex highlighted in red we find that the midpoint before and after - both lie in the polygon - so we continue on to the next iteration (i.e. the next intersection point – in the diagram highlighted as yellow). Here, we find that the midpoint before lies inside the polygon however, the midpoint that immediately follows does not, so hence this point would have to be the furthest point the ray can be extended to.



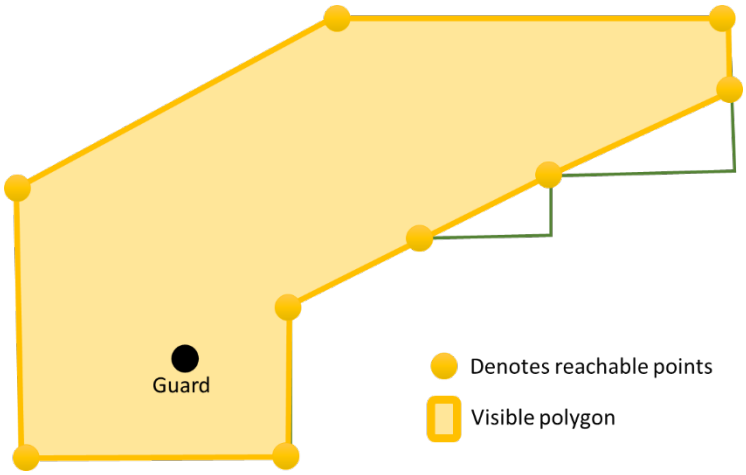
- Now that we have the furthest point it can be extended to. Check if the vertex that has been extended is already in the list of reachable points. If it isn't, then this point is pushed to the list of points in the visible polygon.
- Check also if the extended point (that was just found) - is already in the list of reachable points. If it is not, this point is also pushed to the list of points in the visible polygon.
- We do this to avoid having duplicate points in the list of reachable points.

*We now have all the reachable/ extendable points in an unsorted array as shown below:




- Sort the extended points into an array:
  - Iterate through all of the lines of the polygon in (anti-clockwise) order and for each line:
    1. Find the points in the unsorted extended points array that lie on the line
    2. Sort them in ascending order according to the distance they are away from the start of the line.
    3. Push ordered points into the sorted array.
  - Remove any duplicate points from the sorted array.
- Push sorted array into array consisting of visible polygons for every guard.

*We now have an array of the visibility polygons of each guard as shown below:



**Auxiliary algorithm #2: Finding the intersection point of two line segments**

http://stackoverflow.com/questions/15690103/intersection-between-two-lines-in-coordinates; top rated answer

**Auxiliary algorithm #3: Finding whether a point lies in the polygon**

https://en.wikipedia.org/wiki/Point_in_polygon; the ray-casting algorithm

## 2. Part 1: Finding minimal set of guards (positions) for a polygon

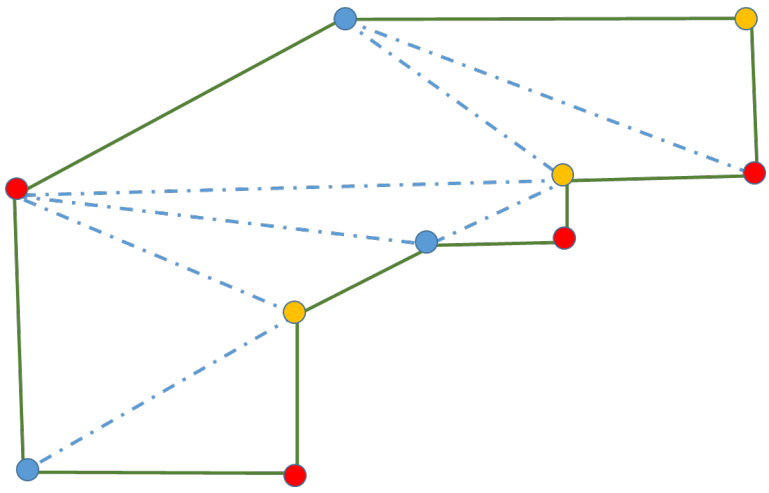
In order to find the minimal set of guards, we came up with a variety of solutions which we implemented in a series of iterations:

**1st solution:** "Colour triangulations and find minimum set

1. Triangulate the polygon based on the ear clipping method.

   This is applicable to our problem set as it works with any simple polygon with at least 4 vertices without holes. Such a polygon has at least two 'ears', which are triangles with two sides being the edges of the polygon and the third one completely inside it. The algorithm then consists of finding such an ear, removing it from the polygon (which results in a new polygon that still meets the conditions) and repeating until there is only one triangle left.

2. Set colour attributes to each of the vertices so that each vertex of the triangles have a unique colour. As shown in the diagram.



3. Find the colour with the least number of vertices. i.e. in this case it would either be blue or yellow with 3 vertices each (whereas there are 4 red vertices).
4. Set the guards on those positions.
5. Remove redundant guards:
   - Three ways in which to order the guards for removal:
     - Original order of the guards

- Random order of guards
- By their visibility area size in ascending order.

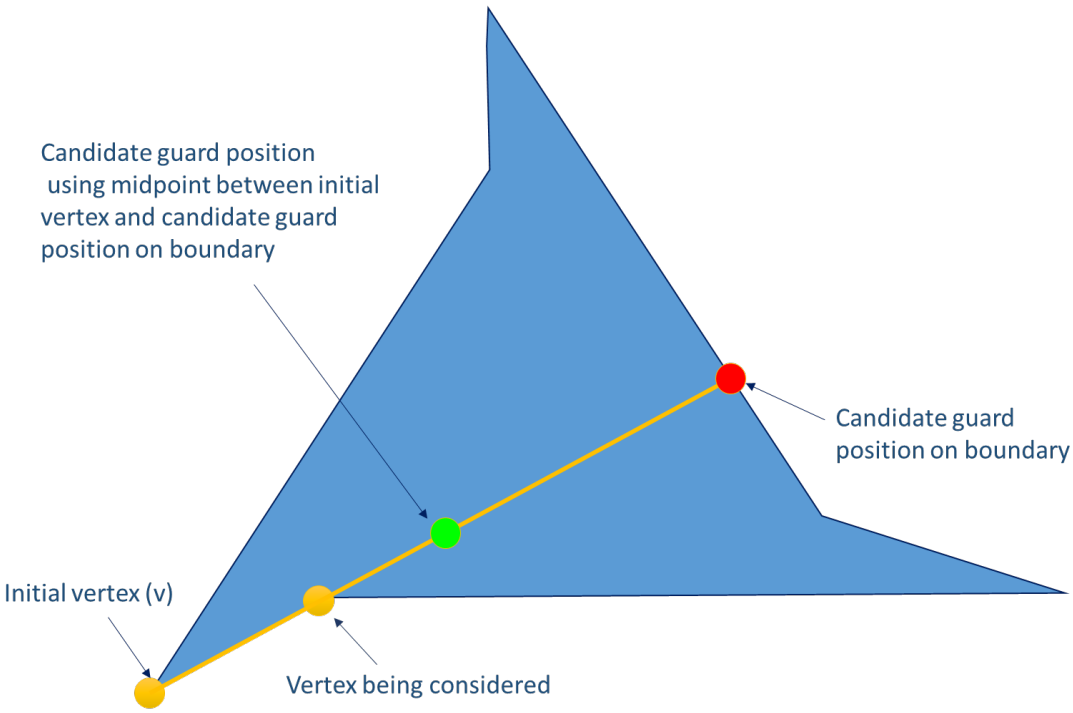
- Remove first guard
- Check to see if there are now non-visible areas within the polygon
- If there is, push the guard back into the list
- Repeat this process for the remaining guards in the list.

**2nd solution:** "Greedy Algorithm on vertices & extended visible points from every vertex"

1. Create an empty list of candidate guards.

Note: There are two different ways in which to select candidate guard positions. The "polygon boundary" or "midpoint" options.

2. For a polygon, iterate through all vertices and for each vertex, v:
    a. Iterate through all the other vertices and on iteration, j, create a line from vertex v passing vertex j to the boundary.




    b. Find all the intersection points of the line with all the other lines.
    c. Check the intersection points that are reachable from vertex v
    d. Of the points that are reachable – add either the polygon boundary point ***or*** the midpoint of the line from vertex v to the intersection, according to the parameter specified. And add them to the list of candidate points for guards.
        - Another version of this algorithm was also written to allow for all the intersections of all the extension lines ***and/or*** intersections of

all lines between all vertices of the polygon which are inside the polygon to be considered as guard candidates. By including these as candidates, we were able to get better solutions than before but at the expense of the performance of the program, see remarks at the end of this chapter for detail.

3. We adopt a greedy algorithm to reduce the size of the candidate set of guards. There are two ways in which we do this:
    a. In terms of the visibility polygon of each guard
        - *Deterministic variation:* For each iteration of the greedy algorithm, selects the candidate which would add the greatest visibility area to the current solution
        - *Probabilistic variation:* For each iteration of the greedy algorithm, assign a weight to each candidate according to the visibility area it would add to the current solution, and pick candidate randomly considering the weights.
    b. **Or** in terms of number of visible candidate guards
        - *Deterministic variation:* For each iteration of the greedy algorithm, selects the candidate which would make the greatest number of additional candidates visible
        - *Probabilistic variation:* For each iteration of the greedy algorithm, assign a weight to each candidate according to the number of additional candidates, and pick candidate randomly considering the weights.

Note: The reason we compute this greedy algorithm is because the next stage is very computationally expensive and is also probabilistic. For that reason, we have to reduce the size of the candidate set of guards.

4. Remove the redundant guards:
    - Three ways in which to order the guards for removal:
        - Original order of the guards
        - Random order of guards
        - By their visibility area size in ascending order.
    - Remove first guard
    - Check to see if there are now non-visible areas within the polygon
    - If there is, push the guard back into the list
    - Repeat this process for the remaining guards in the list.

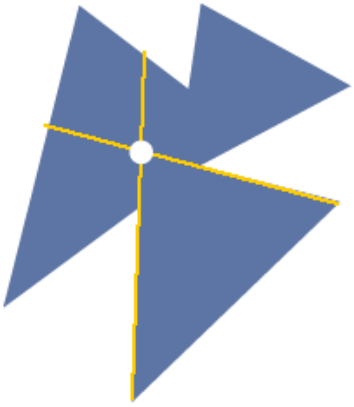
**Brute-force optimization**

In order to compute solution sets to generate the final output file, we developed a brute-force optimization function which chained the algorithms previously discussed in various

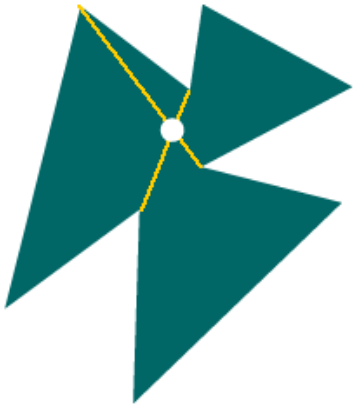
combinations: coloured vertices, deterministic greedy by area, deterministic greedy by number of guards, probabilistic by area and deterministic greedy by number of guards and random removal of redundant guards. The optimization function keeps track of the current best solution and thus tends to the optimal solution set as time progresses due to the probabilistic nature of the algorithms.

**Remarks**

In some cases, it is beneficial to gaining a better solution by considering intersections of all the extension lines ***and/or*** intersections of all lines between all vertices of the polygon which are inside the polygon to be considered as guard candidates. However, computing these sets of points can often be computationally expensive, especially for polygons with a large number of vertices.



Intersection of two extension lines



Intersection of two vertex-to-vertex lines

From experience when testing the program, intersections of all lines between all vertices of the polygon which are inside the polygon performs very poorly when the polygon is a shape made out of convex polygons with a large number of vertices or is itself a convex polygon. This is because the number of lines we are considering in this case is can be as large as n! where n is the number of vertices. On top of that, the algorithm that computes all the intersection points between these lines has a time complexity of $O(n^2)$ where n is the number of lines. Therefore, overall, the worst case time complexity for computing this set of points is $O((n!)^2)$. With this kind of performance, we have no choice but to not consider these points as guard candidates for most of these cases, even though it may be required in order to get a better solution.

## 3. Part 2: Finding a refutation point (uses all the auxiliary algorithms explained earlier)

1. Find the vertices of the non-visible areas of the polygon (given all the visible polygons for each guard)
    - Using the JavaScript GPC library we compute the difference between the whole polygon and each visible polygon. And this gives us the vertices of the non-visible areas of the polygon.
2. Find a refutation point within a non-visible area
    - Iterate through the non-visible areas to find a refutation point

- For each line in the non-visible area find the midpoint of the line, call this line x.
- Iterate through each vertex in the non-visible area and create a line to that vertex from point x. Find the midpoint of this line and see if it falls within the polygon using the ray casting algorithm.
- If a point is found to lie in the non-visible area, return this point.

## 4. Time Complexities

Part 1

- **Polygon triangulation** is achieved by recursive ear-clipping which runs in $O(n^2)$ where $n = number\ of\ vertices$.
- **Triangulation colouring** is computed by recursive colour assignment which runs in O(n) where $n = number\ of\ triangles = number\ of\ vertices - 2$.
- **Redundant guard removal** is achieved by iteratively considering the removal of each guard and checking whether the removal invalidates the correctness of the solution set. This runs in $O(n^3)$ as it considers $n$ number of vertices, and checking the correctness of a solution requires polygon union operations which run in $O(n^2)$.
- **Greedy selection** based on guard visibility greedily selects the candidate which reveals the most guard candidates in addition to the current solution, verifying the current solution with polygon Boolean operations. The runtime of this algorithm is dependent on the input with worst case time complexity being $O(n^4)$.
- **Greedy selection** based on area greedily selects the candidate which reveals the most area in addition to the current solution, also verifying the current solution with polygon Boolean operations. The runtime of this algorithm is dependent on the input with worst case time complexity being $O(n^5)$.

## 5. How were the algorithms for computing/checking guards sets tested?

We tested our solution using the algorithms we came up with in parts 2 and part 3 (visualisation). We made sure that there were no non-visible regions in our solution set.

## 6. How were the input files processed, and output files produced?

Input

1. Once we have the input file we were able to get the lines by splitting the input into lines. This was done by looking for an optional carriage return followed by a new line character regex: \r?\n.
2. For every line:
   - We remove the first part of the line containing the line number followed by the colon.
   - We extract the string containing all the points before the semi-colon (if a semi-colon exists; for part 2) using the regex (/[^;]+/). We then extract the individual co-ordinates using (/\([^\)]+\)/g) and put these into Point objects (that we defined as simply having an x and y attribute) for each point and store these into a list of vertices. These vertices are used to create a Polygon object.

- We extract the string containing all the guard points using (/;.+/). We then extract the individual co-ordinates using (/\([^\)]+\)/g) and put these into a list of Point objects (that we defined as simply having an x and y attribute) of each point and store these into a list of guard points.

Part 1 Output

As better guard sets are discovered during the search, a text area in the GUI gets automatically updated with the current solution set formatted as per the specifications.

Part 2 Output

After the application confirms that the guard sets have insufficient coverage and finds points which refute the incorrect solution, a text file containing these points is generated and downloaded through the browser.

## 7. How the design/implementation workload split between the members of the team?

- Bandi and Gordon: input parsing, output generation, visualization and implementation of art gallery algorithms, finding refutation points
- Shivam and Osman: finding visibility polygons, integration of third party polygon library for computation of polygon Boolean operations

## 8. A link to the repository with the development

https://github.com/bandienkhamgalan/comp205p