Jenifer Cochran

Final Project Report

CS 633

Mathematical Morphology: Minkowski Sum

Abstract:

Mathematical Morphology is a set of mathematical operations performed on geometric shapes. This is widely used in digital images but is also employed on vectors, shapes, and other spatial structures. In this paper, we will discuss the operations performed on geometric shapes instead of imagery. The basic operations of mathematical morphology are erosion, dilation, opening, and closing. Opening and closing operations are just a combination of dilation and erosion. This paper will discuss a few approaches on how to dilate geometric shapes and how to modify one of the approaches slightly to erode the two shapes instead. The approaches can be applied to two convex polygons as well as two non-convex polygons on a two-dimensional plane. It will also discuss the pitfalls and simple mistakes can be made when implementing the algorithms. There are many use cases for these basic operations such as removing distortion and noise from an image, collision detection, geometric modeling, spatial planning, biological form description and understanding, crystallography, textured object modeling, and more.

Motivation and Introduction:

The primary motivation of this project was to understand and implement the closing and opening mathematical morphology operations without using popular libraries like CGAL (Computational Geometry Algorithms Library). I also wanted to learn an algorithm that could be applied to GIS (Geographic Information System), since my current position involves creating GIS software applications. This project does use Esri Runtime .Net 10.2.7 for the mapping components and basic data structures for the geometries. The goal is to allow a user to create two polygons on a map and use that as input to calculate the dilation (also known as Minkowski addition) and the erosion (also known as Minkowski difference) of the two shapes drawn. The calculation must be able to accept two convex or non-convex

polygons (in any combination) and be able to visually display the results on the map. The key algorithm for this project to be successful is to be able to accurately calculate the dilation operation since the rest of the operations depend on it.

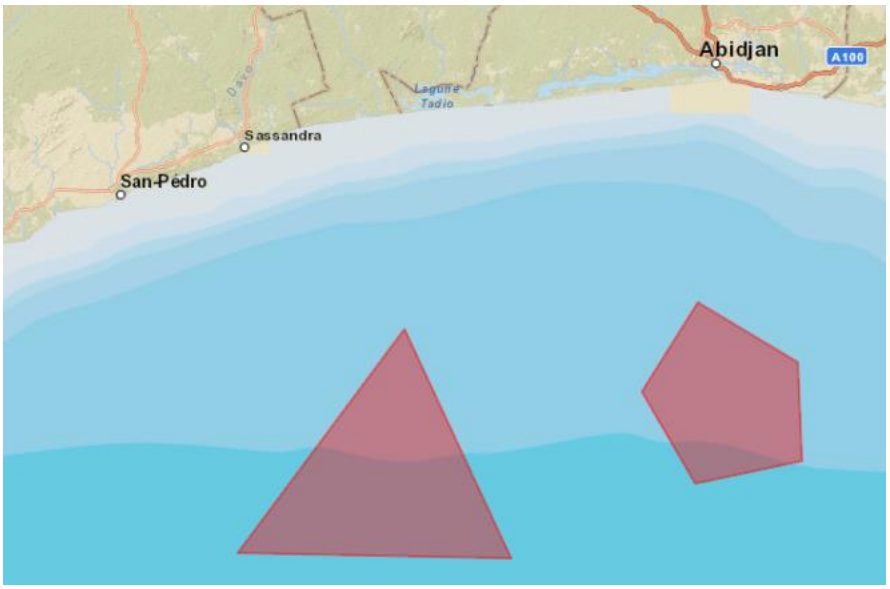
There will be three approaches covered on how to calculate dilation of two polygons. The first approach is simple and easy to implement but will not work for non-convex polygons. The second approach is a different than the first approach but only works for convex polygons as well. The last approach will work on both convex and non-convex polygons and can even be used in the three-dimensional plane. After understanding these approaches, we will discuss how it can be easily translated into a solution for the erosion of two polygons as well by making one simple change.

Another motivation as described in Ghosh and Haralick's paper *Mathematical Morphological Operations of Boundary-Represented Geometric Objects* is the resemblance between morphology and the theory of numbers. They discussed on how there are similarities to theoretic results and morphological theorems. They were able to map different mathematical morphologic equations to operations with natural numbers. One example was that $A \oplus B$ is alike to $m \cdot n$. The paper unfortunately did not discuss why such a relationship exists but did discuss how the different mathematical morphologic operations could be mapped to integer operations.

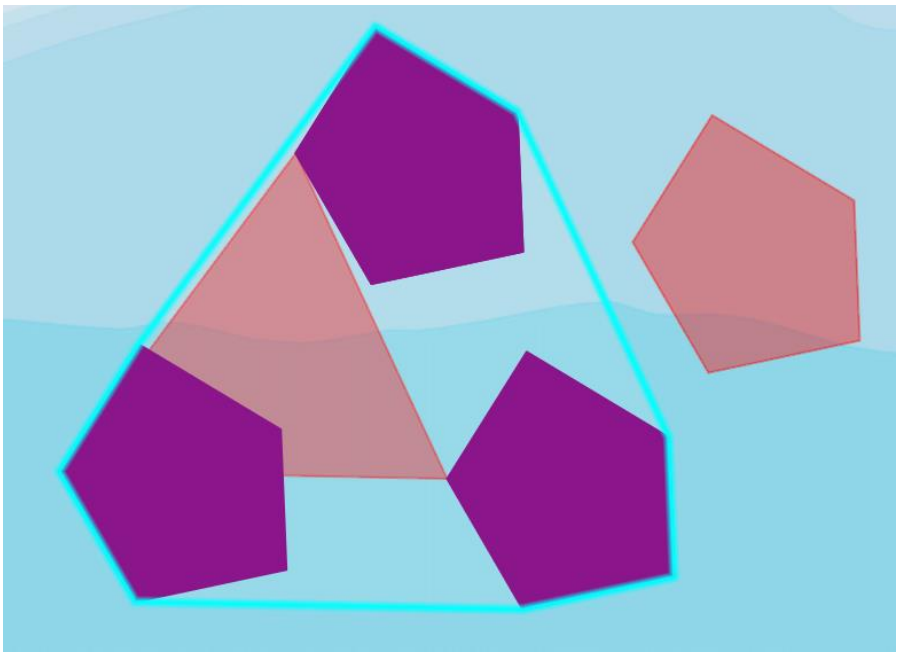
Methods Used and Studied:

In this paper, we will discuss three different approaches to calculate the Minkowski sum of two polygons. The first of which only is applicable to convex polygons. We will discuss why this particular algorithm falls short when applying this to non-convex shapes.

The first approach will take one of the two given polygons (let us name them Polygon A and Polygon B respectfully) and match up a vertex of Polygon A to every vertex in Polygon B.

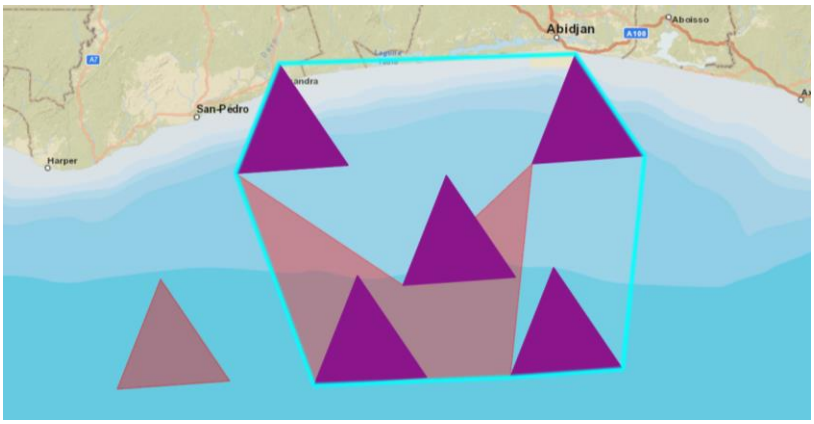


The next step would be to take the Convex hull of all the points and the resulting Polygon will be the Minkowski sum of Polygon A and Polygon B.

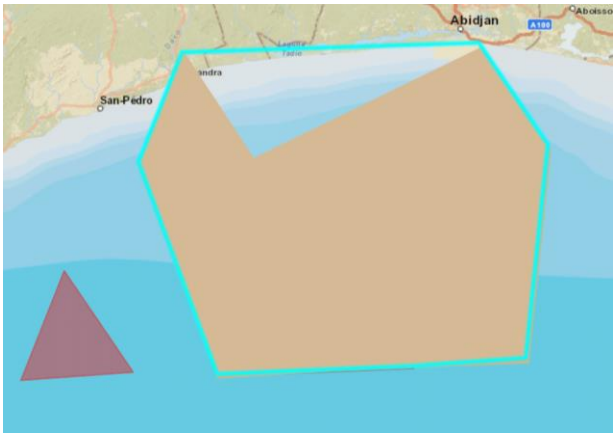


The blue highlight represents the resulting polygon after computing the Minkowski sum on the purple pentagons with the red triangle. This will work for any pair of convex polygons. Unfortunately, as you

will see in the following images how applying the convex hull can lead to errors in the Minkowski sum operation on non-convex shapes. Below you see the same steps applied to a triangle with a non-convex 5-sided polygon.



Since it takes the convex hull of the shape, it makes the Minkowski sum larger than it should be. The blue line represents what the result is when using the convex hull approach and the tan polygon represents the correct answer of the Minkowski sum (in the image below).



There is a triangular shape that the convex hull approach includes in the final answer that should be left out (the space between the blue line and the tan shape at the top). Due to this, we must try a different approach to polygons that are non-convex shapes.

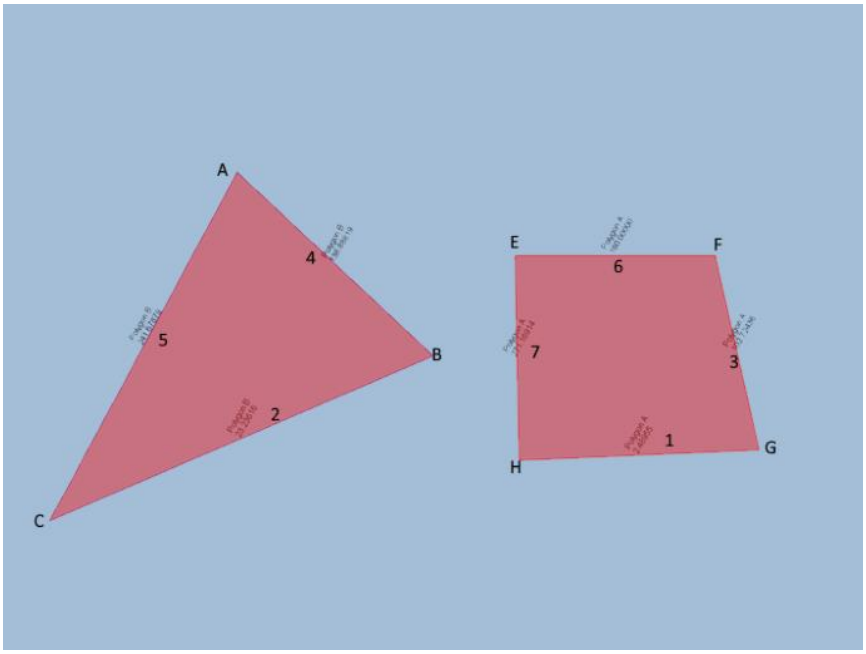
The second approach that we will discuss also only works for convex polygons but we will alter it in the third approach slightly to make it work for non-convex polygons as well. In the second approach, instead of taking one polygon and setting it on all the vertices of the other, we will sort the segments of both polygons by their angle with respect to the horizontal plane.

Below is my snippet of code that does exactly this:

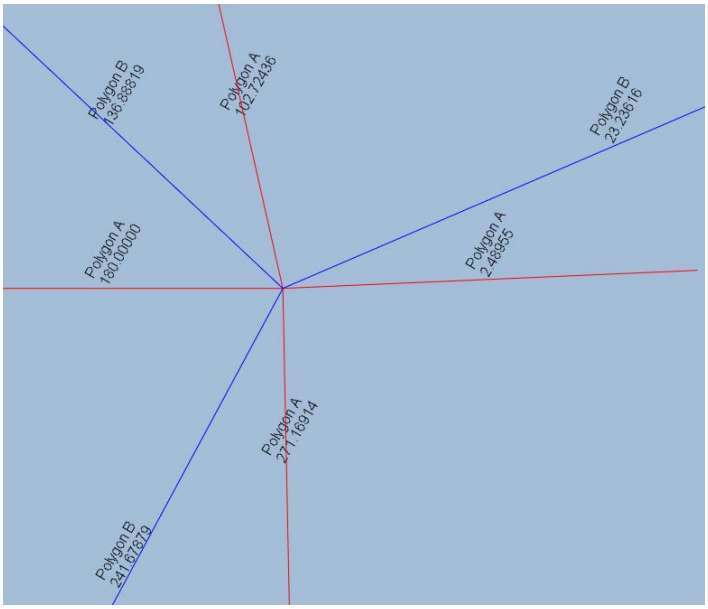
```
var opposite = segment.EndPoint.Y - segment.StartPoint.Y;
var adjacent = segment.EndPoint.X - segment.StartPoint.X;
var radians = Math.Atan2(opposite, adjacent);
var degrees = RadianToDegree(radians);
return ConvertRange(degrees);
```

First step is to calculate the opposite distance over the adjacent distance. A very important note is to take Math.Atan2 over Math.Atan because Math.Atan2 calculates the arctangent with respect to all four quadrants, where as, Math.Atan only accounts for quadrants 1 and 4. Since we are ordering our segments clockwise, it is important that we consider if the segment angle is in quadrant 2 versus 4 as well as the difference if the segment was in 1 versus 3. I initially made this mistake of using Math.Atan and quickly discovered that it does not work.

After ordering both polygon A and polygon B's segments by their angle, we traverse the segments in order determining which vertex to add to that segment's start and end point (pictures will better explain this). To determine which vertex should be added to the segment's start and end point, we must look at the previous and next segment that belong to the opposite polygon of the current segment. Those two segments that sandwich the current segment are from the opposite polygon. The two segments will share a vertex, and this is the vertex that is added to the current segment's start and end point. Let's do an example. In the picture below we have two convex polygons with their segment angles calculated (the label is placed at the midpoint of the segment).



We then translate them to the same origin to order the segments. Below is a photo of the angles translated for better comparison.



For each segment starting with angle 2.48955° from Polygon A (Segment 1), we figure out which vertex to add to the start and end point of this segment. To determine which vertex to add to the segment, we look at the two segments that segment 2.48955° from Polygon A lies between from Polygon B. The two segments are between Polygon B 241.67879 (Segment 4) and Polygon B 23.23616 (Segment 2) and those two segments share a vertex point B. This is the vertex we use to offset Segment 1 from Polygon

A. We simply add the x and y coordinate values of vertex B to both the start and end points of the Segment 1. You continue this until you have reached the end of the segment list. The final output will be your Minkowski sum.

There are problems with this approach when it comes to applying it to non-convex shapes however. This only will work with convex shapes. Therefore, we must alter the approach even further to account for non-convex shapes.

The third approach is called the Convolution. The process is similar to the second approach but instead of traversing all the segments in angle order, we will traverse the vertices of one polygon and then traverse the vertices of the second polygon. It is important that we traverse the polygon in counter-clockwise order. I had made the mistake in assuming that my data structure that I received from the user's input would always be in counter-clockwise order. The following snippet of code detects whether or not the points in the polygon are in clockwise or counter-clockwise order:

```
public bool IsCounterClockwise(Polygon polygon)
{
   var sum = 0.0;
   var points = polygon.Parts.First().GetPoints().ToList();
   var previousPoint = points.First();
   for(var index = 1; index < points.Count(); index++)
   {
      var nextPoint = points[index];
      sum += (nextPoint.X - previousPoint.X) * (nextPoint.Y + previousPoint.Y);
      previousPoint = nextPoint;
   }
   return sum <= 0.0;
}
```

It is a clever algorithm in which works for both convex and non-convex polygons. If you sum over the edges $(x_2 - x_1)(y_2 + y_1)$, if the result is positive, the polygon is sorted in clockwise order, otherwise it is in counter-clockwise order.

After placing the points in the correct order, it is important to check if any of the segment's angles in polygon A are too close to any angle segment in polygon B. This can make the future steps much more difficult if there are parallel lines. Therefore, if there are any line segments whose angles are too close between the two polygons, you would simply rotate one of the polygons by a small angle until they are no longer too close. Here is the code to rotate a polygon given an angle:

```
public static Polygon RotatePolygon(this Polygon polygon, double angle = RotationAngle)
{
    angle = DegreeToRadian(angle);
    var rotatedPolygon = new PolygonBuilder(polygon.SpatialReference);
    foreach(var point in polygon.Parts.First().GetPoints())
    {
        rotatedPolygon.AddPoint(new MapPoint(point.X*Math.Cos(angle) - point.Y*Math.Sin(angle),
                                             point.X*Math.Sin(angle) + point.Y*Math.Cos(angle)));
    }
    return rotatedPolygon.ToGeometry();
}
```

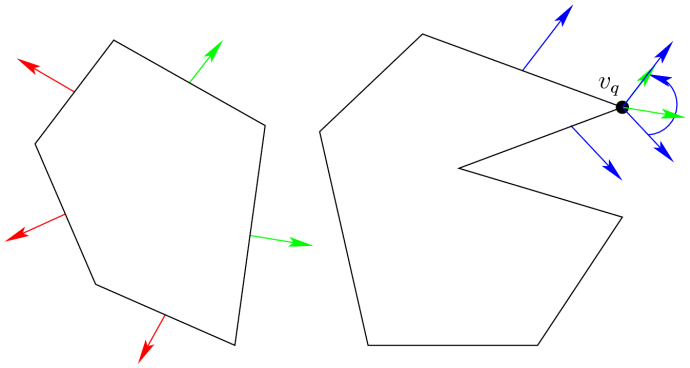
Here we are simply applying a simple rotation algorithm:

$$x' = x \cos \phi - y \sin \phi$$

$$y' = y \cos \phi + x \sin \phi$$

We continue to rotate one polygon until there is enough difference between the segments across polygons such that there are no near parallel segments.

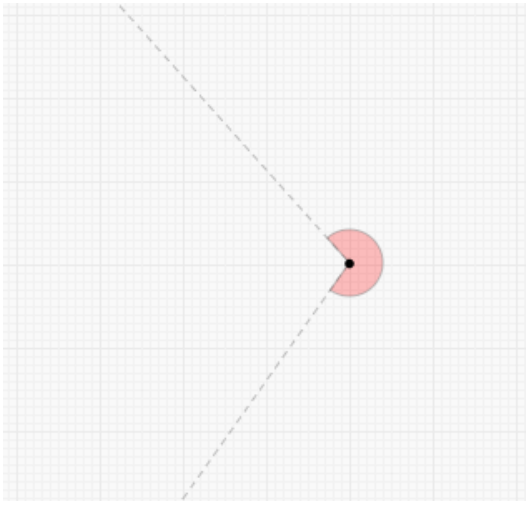
Now that we know the vertices are in the correct order and that there are no two segments across the two polygons in which they are nearly parallel, we can traverse the vertices of the first polygon A. We will have an angle range (a lower and upper bound) that this vertex affects the segments in polygon B. An angle range is determined by the previous and next segment's angle (the previous will be the lower bound and the next segment will be the upper bound). For each vertex in polygon A, we find which segments in B have an angle that is within the range of the vertex in A. If the segment is within that range, we will add the vertex from A to the start and end points of that segment from B.



Looking at the picture shown above we have a vertex v_q with an angle range noted by the blue vectors. The range is attained from the connected segments to the vertex. Notice out of the 5 segments of the opposite polygon, only two segments have an angle that is within the range of v_q. The green vectors represent the segments that are added to vertex v_q and the red vectors represent the segments that are not added to vertex v_q. After we iterated through all the vertices in A and adding the segments in B that

are within the angle ranges we repeat the process, but traverse through the vertices in B and add the segments in A that are within the angle ranges. There are some things that can get tricky when performing this operation. For example, when a vertex is at a reflex point, the lower bound angle is larger than the upper bound angle. Refer to the image below:

The reflex angle has an angle range where the lower bound is ~206° and an upper bound of ~152°. Looking at the numbers it seems strange that the lower bound is larger than the upper bound.

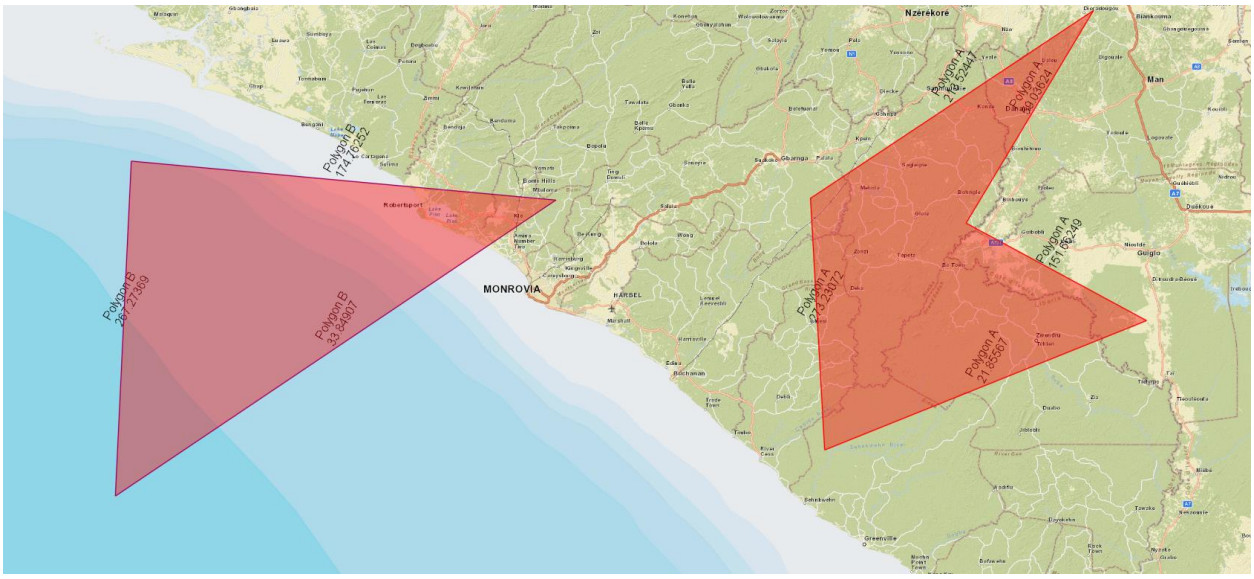


This is because the angle range is on the inside of the polygon versus outside. Therefore when we are asked to see if a segment is effected in an angle range where the lower bound is larger than the upper bound, we must check if the segment's vector is greater than the lower bound or lower than the upper bound. Because the normal check is to make sure the angle is above the lower bound and strictly lower than the upper bound. Here is the snippet of code to determine if a segment is affected by a vertex's angle range.

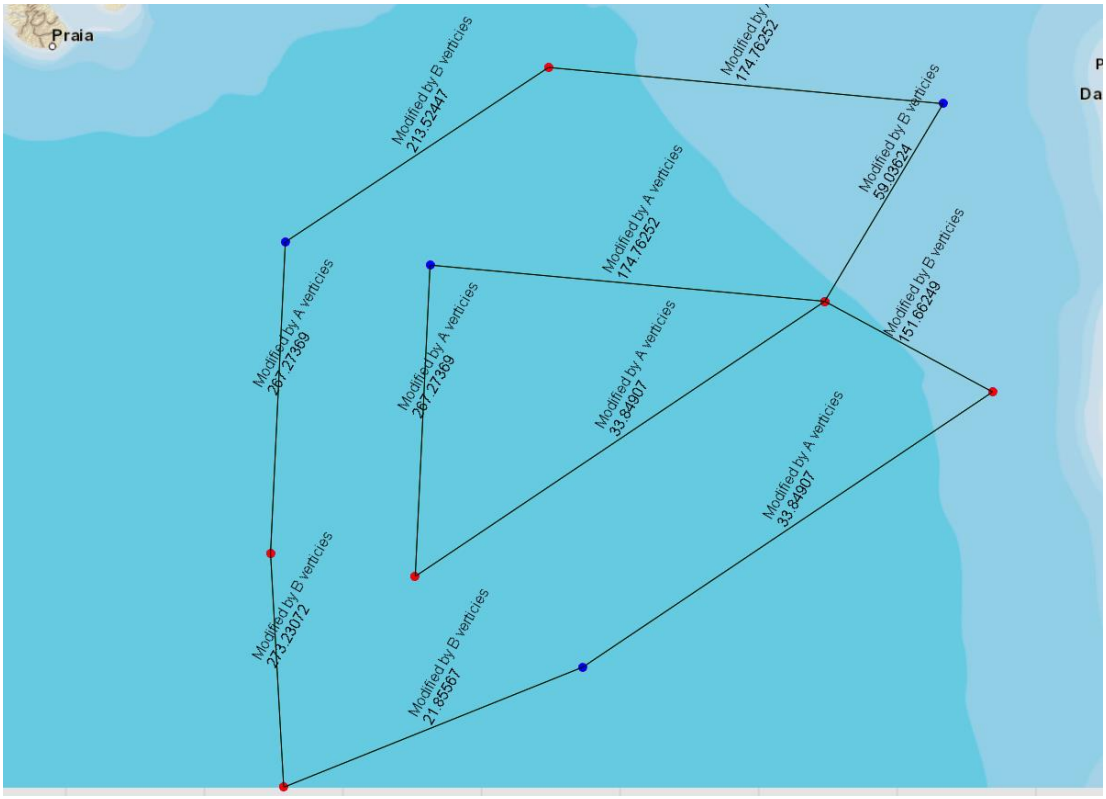
```
public static List<Segment> GetSegmentsWithinRange(double lowerBound, double upperBound,
Polygon polygon)
{
    var segments = new List<Segment>();
    foreach(var segment in polygon.Parts.First())
    {
       var angle = segment.CalculateAngle();
       //check if reflex angle
       if(lowerBound > upperBound)
       {
          if(angle >= lowerBound || angle <= upperBound)
          {
              segments.Add(segment);
          }
       }
       //otherwise treat as normal
       if(angle >= lowerBound && angle <= upperBound)
       {
          segments.Add(segment);
       }
   }
   return segments;
}
```

This will return the list of segments within the polygon that are affected by that vertex's angle range. After we collect all the segments altered from both polygons, we will end up with a collection of segments that only some of which are part of the final solution. Here is an example of what the process looks like below:

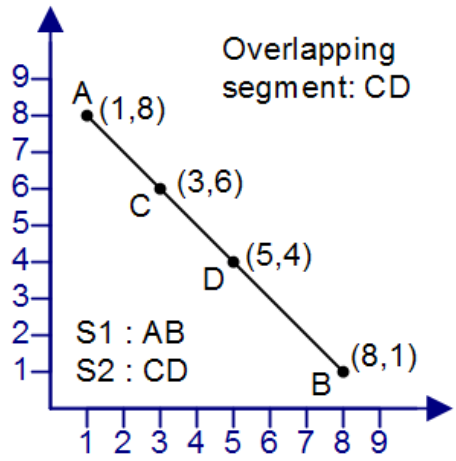
Original Polygons:



Collection of segments after being modified:

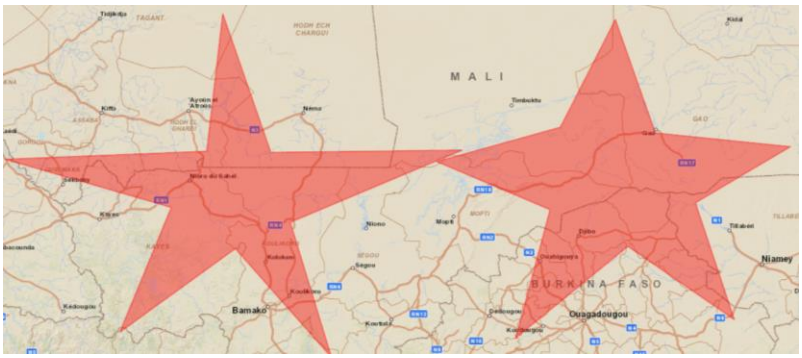


If you notice, there are segments that need to be removed in the collection in order to get the correct Minkowski sum polygon. Before we can traverse through these segments we need to calculate the Arrangement of the line segments. The Arrangement is calculating all the intersections for all the line segments and placing a vertex at each intersection point. If two segments happen to overlap we would break the segment into three parts:

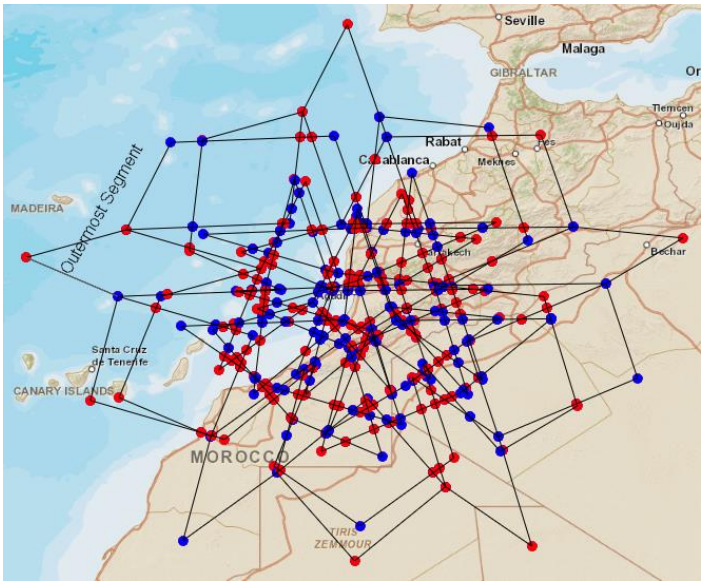


The three new segments that are a part of the Arrangement (in the image above) would be segment AC, CD, and DB. The other cases to consider are where the segments either cross to form an "X" which will be broken up into 4 segments or if they simply meet like the letter "T" in which case you break it up into 3 segments. It is also important we do not count if an intersection occurs at the start or end point of a segment. This can cause the intersection method to loop infinitely since there are a lot of segments that are connected together (like in the shape of an "L"). Though normally you would consider this an intersection, this is part of the Arrangement and does not need to be broken down any further. Here is an example of a more complicated calculated arrangement:

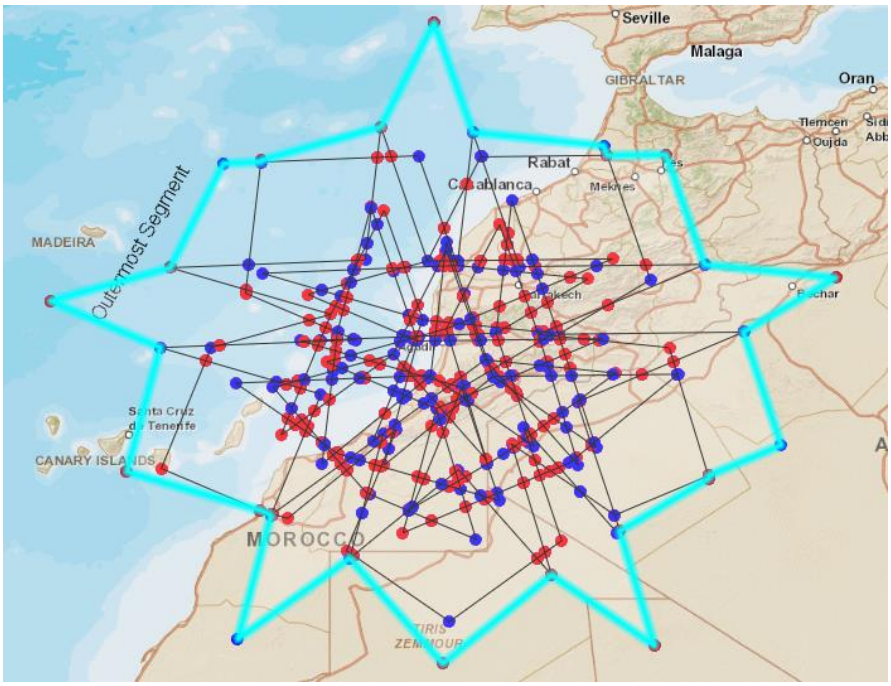
Input:



Output Arrangement:



Minkowski Sum:



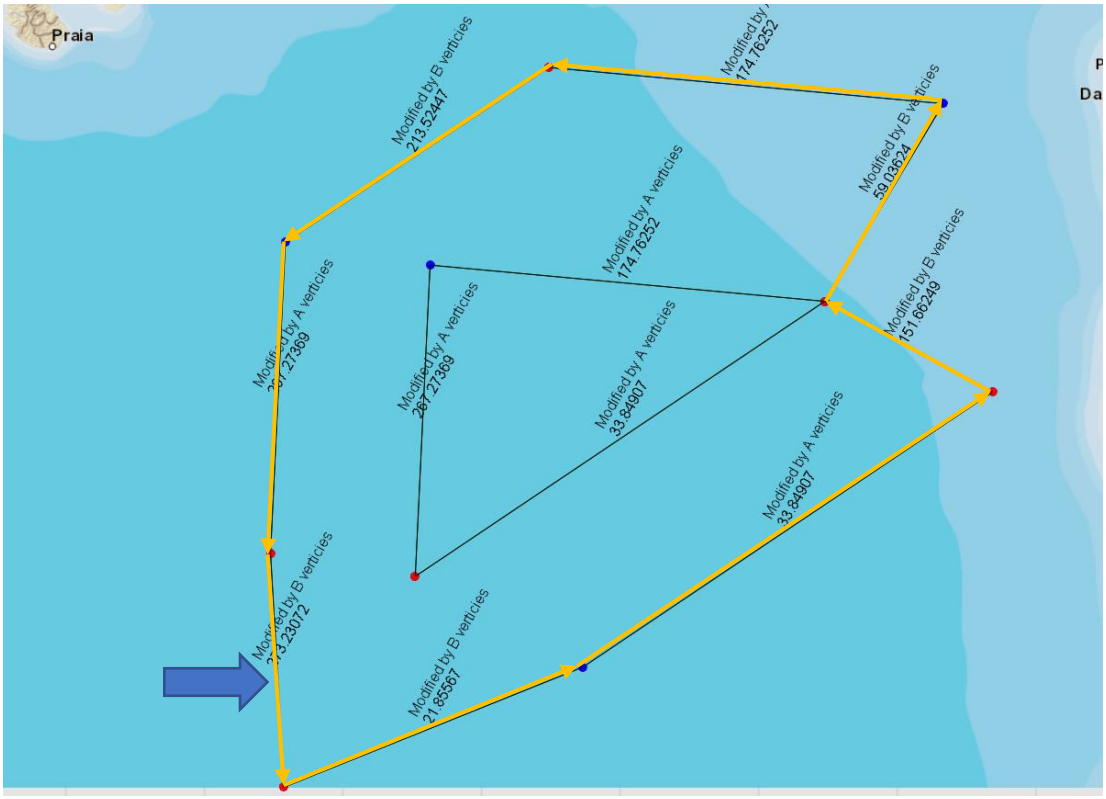
It is important to break up the segment where the start and end points are since when we calculate the boundary of the polygon, it will make the correct turns and possibly cross the interior of the polygon.

After calculating the Arrangement, the last step is detecting the boundary of the polygon. First, we need to identify a segment that is part of the boundary. The following snippet of code traverses through the segments to find the outer most segment:

```
public static Segment GetOuterMostSegment(List<Segment> segments)
{
  //create a list of points from the given segment collection
  var points = new List<MapPoint>();
  foreach(var segment in segments)
  {
    points.Add(segment.StartPoint);
    points.Add(segment.EndPoint);
  }
  //find the most eastern point
  var minXPoint = GetMinXPoint(points);

  //find all points that have a segment whose start or end point connects to the most eastern point
  var pointsConnectedTo = segments.Where(segment => segment.StartPoint.MapPointEpsilonEquals(minXPoint) ||
                                  segment.EndPoint  .MapPointEpsilonEquals(minXPoint))
```

```
                    .Select(segment => segment.StartPoint.MapPointEpsilonEquals(minXPoint) ?        segment.EndPoint : segment.StartPoint);

  var outerMostEnd = pointsConnectedTo.First();
   //for all the points that are connected to the most eastern point, find the point
  // that is the farthest east and lowest
  foreach(var point in pointsConnectedTo)
  {
      //point is more eastern and lower
    if(point.X < outerMostEnd.X &&
       point.Y < outerMostEnd.Y)
    {
       outerMostEnd = point;
    }
    //same eastern but if lower chose that point
    else if(GeometryUtility.IsEqual(point.X, outerMostEnd.X) &&
          (point.Y < outerMostEnd.Y))
    {
       outerMostEnd = point;
    }
  }

  return new Esri.ArcGISRuntime.Geometry.LineSegment(minXPoint, outerMostEnd);
}
```

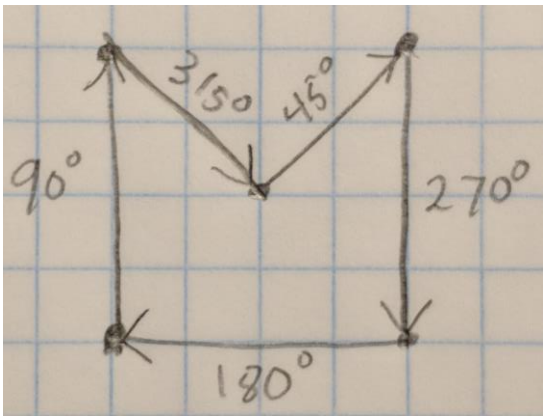
Now that we have a segment that is guaranteed on the boundary of the polygon, we need to traverse the segment going in counter-clockwise direction to find the rest of the segments that are on the boundary. We will look at the end point of the outermost segment and find any segments that are connected to that end point. If we have more than one segment that is connected to that end point we pick the right-most segment.



The segment that the blue arrow is pointing to is the outer most segment that we found with the algorithm provided earlier. The yellow arrows are the segments that are part of the Minkowski sum final solution. If you notice at each vertex that contains multiple segments connected to it, we picked the segment that was the right most turn.

Those segments are part of the final solution. Something to note, this algorithm not only works for both convex and non-convex polygons, but also works in the 3D plane. Lastly, we can modify this algorithm only slightly and we can calculate the erosion between the two polygons instead.

When calculating the Convolution of the two polygons we calculate the segment's angle in counter-clockwise order. To find the erosion of polygon A to polygon B we simply take the complement of polygon B and take the dilation of polygon A to polygon B's complement. This simply means that the segment's angles are calculated in clockwise direction instead of counter-clockwise.



Then you perform the rest of the calculations as you normally would for Minkowski sum using these angles.

Future Work:

The future work would be to see how to implement this algorithm when the polygons have holes in them as well as implement the algorithm in a 3D space. I would be interested in seeing how the holes would affect the algorithm and try to understand how to handle the more complicated shape.

Conclusion:

In conclusion, I found that calculating the Minkowski sum to be deceptively difficult to implement in code. There were a lot of nuances that can easily break getting the correct final polygon for the Minkowski sum when doing the third approach. There were a lot of minute details such as the difference in using Math.Atan2 versus Math.Atan when calculating the angles. As well as when angle

ranges reach a reflex point that the lower bound value is larger than the upper bound value. I also found that calculating the arrangement to be difficult since a single segment could have many intersections or be collinear, in which case must be broken into three segments. I would like to optimize the code for calculating the intersections as well. I currently have it working by looping through the pairs of segments and breaking the loop when the first intersection is found. Then I call it recursively until there are no more intersections found. This part of the code is where it takes up the most processing time and would be the first thing I would fix.

I also made poor assumption on the ordering of my points in my Polygon object (an Esri runtime .Net data structure) would always be counter-clockwise, where it could be either way. I had to discover how to determine in which order the points were traversing before starting my calculation on Minkowski sum. Another obvious oversight I had made on my part was when detecting which segments were connected to vertex, I had not use an epsilon value to compare my double values. Therefore, when I was attempting to either find the outermost segment or traversing the boundary to find connected segments, I was getting incorrect answers due to precision. Overall, I found that implementing the algorithm made me understand and appreciate the algorithm more than I expected to. I had not realized how in Computational Geometry that visually the concepts seem straight forward and simple, but when getting to the details it can get complicated quickly.

Works Cited:

- TONS of help from Professor Jyh-Ming Lien
- Ghosh, P.K. & Haralick, R.M. J Math Imaging Vis (1996) 6: 199. https://doi.org/10.1007/BF00119839
- "How to Determine If a List of Polygon Points Are in Clockwise Order?" Math - How to Determine If a List of Polygon Points Are in Clockwise Order? - Stack Overflow, stackoverflow.com/questions/1165647/how-to-determine-if-a-list-of-polygon-points-are-in-clockwise-order.
- "Mathematical Morphology." Wikipedia, Wikimedia Foundation, 22 Oct. 2017, en.wikipedia.org/wiki/Mathematical_morphology.