Analysis of Occlusion Angle Algorithm

Yao Chen

Introduction

The provided C++ program with this report generates N (user-defined) random circles in cartesian coordinate system that do not overlap with each other or with the origin. For each of these circles, a tilted rectangle is generated within its boundary. The purpose of the algorithm is to compute the occlusion angles caused by these rectangles when viewed from the origin.

Approach

Definitions

1. **Point**: Represents a location in a cartesian coordinate system.
2. **Circle**: Defined by its center (a **Point**) and a radius.
3. **Rectangle**: Defined by its four vertices, which are **Points**.
4. **Coverage Angle**: Represents the angular segment that a shape covers when viewed from the origin.

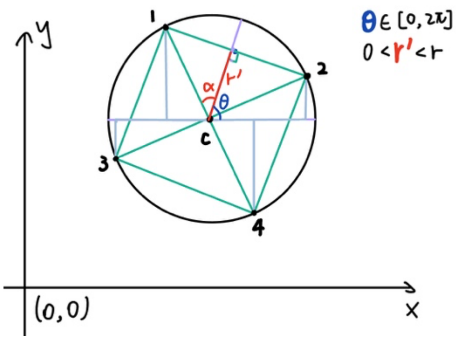
Methods

1. `isOverlapping`: Checks if two randomly generated **circles** overlap.
2. `overlapsWithOrigin`: Checks if a **circle** overlaps with the origin.
3. `rectangleInCircle`: Generates a tilted **rectangle** inside a circle.
4. `angleToOrigin`: Computes the angle from a **Point** to the origin.
5. `coverageAnglesFromRectangleToOrigin`: Finds the **coverage angles** for each vertex of a rectangle and determines the total range of angles covered by the rectangle.

Flow of Algorithm

1. **Random Circle Generation**: Circles are generated with random radii from 0.1 to 2.0 and centers (Points) until N non-overlapping circles are obtained. The circles also do not overlap with the origin.

2. **Random Rectangle Generation**: For each circle, a tilted rectangle is generated within its boundary.

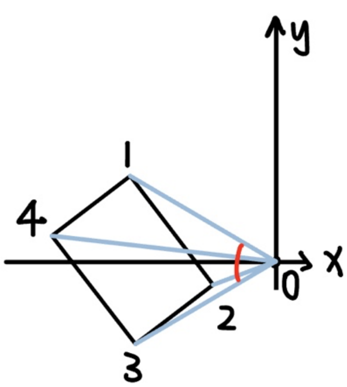


$$p1: (x, y) = (x_c, y_c) + r * (-\cos(\pi - \theta - \alpha), \sin(\pi - \theta - \alpha))$$
$$p2: (x, y) = (x_c, y_c) + r * (\cos(\theta - \alpha), \sin(\theta - \alpha))$$
$$p3: (x, y) = (x_c, y_c) + r * (\cos(\pi - \theta - \alpha), -\sin(\pi - \theta - \alpha))$$
$$p4: (x, y) = (x_c, y_c) + r * (-\cos(\theta - \alpha), \sin(\theta - \alpha))$$

Choose a random tilt angle (theta) and find a tilted line at that angle.
Pick a point (from c to r') on this tilted line.
Find a line (from point 1 to point 2) that's perpendicular to the tilted line and goes through the point.
This line crosses the circle at point1 and point2.

3. **Compute Coverage Angles**: For each rectangle, the angles from the origin to its vertices are computed to determine the angular segments that the rectangle covers.

The **angles** from the origin to its vertices are calculated via `atan2` function in C++. Given the range from $-\pi$ to π, the coverage angle for one rectangle is from the minimum angle to the maximum angle. In special cases, if a rectangle wraps around that boundary, i.e., a rectangle in both quadrant 2 and quadrant 3, its coverage angle will essentially be in two segments. For example,



Angel from point1: 2.62

Angel from point2: -2.82

Angel from point3: -2.65

Angel from point4: 3.02

The coverage range for this rectangle is (2.62, π) and ($-\pi$, -2.65) in this program.

4. **Determine Occlusions**: For each pair of rectangles, their coverage angles are compared to determine overlaps or occlusions.

5. **Output Data**: The details of the generated circles and rectangles are printed out, along with the sum of occluded angles.

Time Complexity Analysis

Below is a pseudo code version of the main logic. There is a nested for loops because it iterates through all the angle ranges of rectangles from a vector of 0s in range $(-\pi, \pi)$. If the rectangle covers some angles that has not already covered by any rectangle, increment from 0 to 1. If overlapping angles are found, we then iterate over this overlapping range in steps defined by the RESOLUTION = 0.01 rad. We use a coverage array to keep track of which parts of the view (in terms of angles) are occluded by any rectangle and update from 1 to 2 to prevent double-counting the overlapped angle.

For each coverage angle that hasn't been previously marked as occluded (coverage = 1), we mark it (coverage = 2) and increase the accumulated angle of occlusion (sumAngle) by the RESOLUTION = 0.01 rad.

```
sumAngle = 0
for i from 0 to N-1:
   for each range_i in rectangleAngleRanges[i]:
          coverageStart = start of coverage angle of range_i
          coverageEnd = end of coverage angle of range_i
                 for k from coverageStart to coverageEnd by RESOLUTION:
                        if coverage[i][k] is not covered:
                               mark coverage[i][k] as covered;
                        else if coverage[i][k] is covered by another rectangle:
                               mark coverage[i][k] as overlapped;
                               increase sumAngle by RESOLUTION
return sumAngle
```

1. **Circle Generation**: In the worst case, generating each circle can take O(N) time (as we need to check overlaps with all previously generated circles). Thus, generating N circles can have a worst-case time complexity of $O(N^2)$.

2. **Rectangle Generation**: Each rectangle is generated in constant time for each circle. So, the time complexity is $O(N)$.

3. **Compute Coverage Angles**: For each rectangle, the angles to its vertices from the origin are computed in constant time. Thus, $O(N)$.

4. **Determine Occlusions**:

Looping from 0 to $N-1$: $O(N)$.

For each i , iterating over each `range_i` in `rectangleAngleRanges[i]`: Each rectangle has a maximum of 2 coverage angle ranges as explained in Flow of Algorithms 3. Compute Coverage Angles. So this loop is $O(1)$ since it's constant.

For each angle range `range_i` iterating over all coverage angles by RESOLUTION =0.01, the total range is $(-\pi, \pi)$ and the number in worst case is 628. So this loop is $O(1)$ since it's constant.

From the above analysis, the most significant term is <u>the circle generation</u> $O(N^2)$. Thus, the overall time complexity of the algorithm is $O(N^2)$.

Conclusion

The algorithm efficiently generates N random non-overlapping circles and rectangles and computes the angles of occlusion caused by the N rectangles when viewed from the origin. The primary computational cost comes from ensuring non-overlap during circle generation, leading to a time complexity of $O(N^2)$. This complexity means that as the number of circles (and hence rectangles) increases, the execution time will grow quadratically.