Project Report

EECS Lab II: ECS 330

Under Guidance of : Dr. Pawan Kumar Aurora

Name: Ajay Choudhury **Roll no.:** 18018 **Date:** 4th May 2021

Objective of the Project: Implementing an algorithm for triangulating a simple polygon.

Definitions:

A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a triangulation of the polygon.

Here I have implemented an algorithm to triangulate a simple polygon. Now, a simple polygon is a region enclosed by a single closed polygonal chain that does not intersect itself.

Before proceeding to the methods of triangulation of a simple polygon, we should have an idea of the ways we can triangulate any polygon and the number of triangles to be obtained.

Theorem:

Every simple polygon admits a triangulation, and any triangulation of a simple polygon with **n** vertices consists of exactly **n−2** triangles.

Proof: This theorem can be proved by induction on n. When n=3, then the polygon is a triangle by itself and thus the number of triangles is 1 i.e. (3-2) which states the correctness of the theorem for n=3.

Let n > 3, and assuming that the theorem is true for all m < n, let P be a polygon with n vertices. So we will first prove the existence of the diagonal in P.

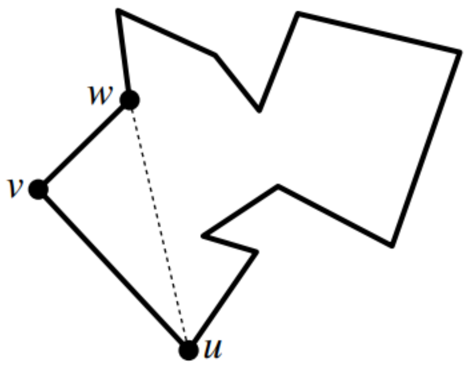

Fig.1: The polygon P with leftmost vertex v.

Let v be the leftmost vertex of P (as shown in the figure above, in case of the same y-coordinates we take the lowest x-coordinate to decide the lowest leftmost vertex).

- **Case I:** Let u and w be the two neighbouring vertices of v on the boundary of P. If the open segment $\overline{uw}$ lies in the interior of P, we have found a diagonal.
- **Case II:** Otherwise, there are one or more vertices inside the triangle defined by the vertices u, w and v or on the diagonal $\overline{uw}$.

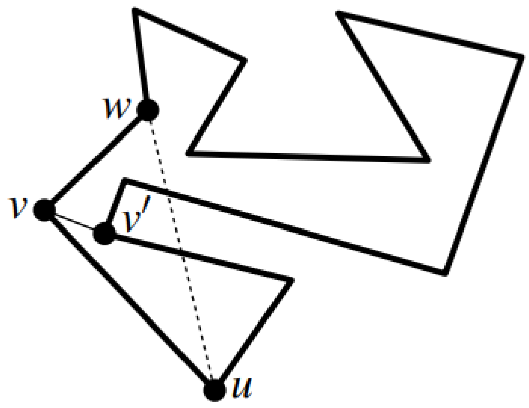

Fig.2: The polygon P with leftmost vertex v and v' farthest from $\overline{uw}$.

Of those vertices, let v' be the one farthest from the line through u and w. The segment connecting v' to v cannot intersect an edge of P, because such an edge would have an endpoint inside the triangle that is farther from the line through u and w, contradicting the definition of v' . Hence, vv' is a diagonal.

Now, we have proved that a diagonal exists. We know any diagonal cuts a polygon into two simple subpolygons, in this case P_1 and P_2. Let m_1 be the number of vertices of P_1 and m_2 be the number of vertices of P_2. Both m_1 and m_2 must be smaller than n, so by our induction statement stated earlier, both P_1 and P_2 can be triangulated.

Let us consider an arbitrary diagonal in some triangulation T_P. This diagonal cuts the polygon into two subpolygons with m1 and m2 vertices, respectively. Every vertex of P occurs exactly in any one of the polygons except the vertices defining the diagonal which occurs in both the subpolygons.

Hence, we obtain an equation: $m_1 + m_2 = n + 2$.
By induction, we know any triangulation of P_i consists of m_i - 2 triangles, which implies that T_P consists of $(m_1 - 2) + (m_2 - 2) = n - 2$ triangles.

Algorithms:
There are many algorithms for triangulation of a polygon. Some of them are discussed below:

- **Ear-clipping method:**

This algorithm works as follows:

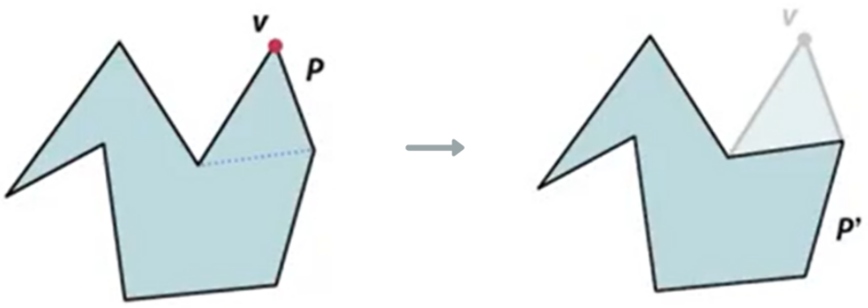

Fig.3: Ear-clipping of a Polygon P

i. Finds an ear
ii. Cut v away with the respective diagonal.
iii. Proceed recursively with the obtained polygon P' that has n-1 vertices.

Here, for a vertex v, we need to check whether the segment with the endpoints at the adjacent vertices of v is a diagonal. After an ear is cut away from P, only two of its adjacent vertices in P can become new ears in the resulting polygon P'.

Afterwards, at each iteration the set of polygon ears can be updated in time linear in the number of vertices of the current polygon.

Time complexity: Time complexity can be calculated by the terms of operations we do. As for every vertex v, we require O(n) and consider the O(n) vertices, time per ear-clipping operation is $O(n^2)$. Now, as the program runs recursively, we obtain a polygon P' with n-1 vertices and hence a recurrence relation as:

$$\begin{cases} O(1), \; n \leq 3 \\ O(n^2) + T(n-1), \; n > 3 \end{cases}$$

This gives the total time complexity: $T(n) = O(n^3)$. However, if separate lists of concave and convex vertices are maintained, the algorithm runs in $O(n^2)$ time.

Implementation: I have written the code for Ear-clipping triangulation of any polygon in Python and depicted them visually using the plotnine library.

Setting up environment:

1. The libraries required to be downloaded are: *pandas* and *plotnine.*
2. The code can be viewed here.

The code for polygon triangulation is:

```
import math
import sys
from collections import namedtuple
import pandas as pd
from plotnine import ggplot, aes, geom_polygon, geom_segment, geom_point
```

```python
Vertex = namedtuple('Vertex', ['x', 'y'])
EPSILON = math.sqrt(sys.float_info.epsilon)


def earclip(polygon):                    # Algorithm to triangulate
    ear_vertex = []
    triangles = []

    polygon = [Vertex(*vertex) for vertex in polygon]

    if isCw(polygon):          # To check if the triangulation is clockwise
        polygon.reverse()

    count_vert = len(polygon)
    for i in range(count_vert):
        prev_index = i - 1
        prev_vertex = polygon[prev_index]
        vertex = polygon[i]
        next_index = (i + 1) % count_vert
        next_vertex = polygon[next_index]

        if is_ear(prev_vertex, vertex, next_vertex, polygon):
            ear_vertex.append(vertex)

    while ear_vertex and count_vert >= 3:
        ear = ear_vertex.pop(0)
        i = polygon.index(ear)
        prev_index = i - 1
        prev_vertex = polygon[prev_index]
        next_index = (i + 1) % count_vert
        next_vertex = polygon[next_index]

        polygon.remove(ear)
        count_vert -= 1
        triangles.append(((prev_vertex.x, prev_vertex.y), (ear.x, ear.y),
(next_vertex.x, next_vertex.y)))
        if count_vert > 3:
            prev_prev_vertex = polygon[prev_index - 1]
            next_next_index = (i + 1) % count_vert
            next_next_vertex = polygon[next_next_index]

            groups = [
                (prev_prev_vertex, prev_vertex, next_vertex, polygon),
                (prev_vertex, next_vertex, next_next_vertex, polygon),
            ]
            for group in groups:
                p = group[1]
                if is_ear(*group):
                    if p not in ear_vertex:
                        ear_vertex.append(p)
                elif p in ear_vertex:
                    ear_vertex.remove(p)
    return triangles


def isCw(polygon):                  # definition of function to check direction
of triangulation
    s = 0
    polygon_count = len(polygon)
    for i in range(polygon_count):
        vertex = polygon[i]
        vertex2 = polygon[(i + 1) % polygon_count]
        s += (vertex2.x - vertex.x) * (vertex2.y + vertex.y)
    return s > 0


def is_convex(prev, vertex, next):
    return triang_sum(prev.x, prev.y, vertex.x, vertex.y, next.x, next.y) < 0


def is_ear(p1, p2, p3, polygon):
    ear = no_int_vert(p1, p2, p3, polygon) and \
        is_convex(p1, p2, p3) and \
        triang_area(p1.x, p1.y, p2.x, p2.y, p3.x, p3.y) > 0
    return ear


def no_int_vert(p1, p2, p3, polygon):
    for pn in polygon:
        if pn in (p1, p2, p3):
            continue
        elif is_int_vert(pn, p1, p2, p3):
            return False
    return True


def is_int_vert(p, a, b, c):
    area = triang_area(a.x, a.y, b.x, b.y, c.x, c.y)
    area1 = triang_area(p.x, p.y, b.x, b.y, c.x, c.y)
    area2 = triang_area(p.x, p.y, a.x, a.y, c.x, c.y)
    area3 = triang_area(p.x, p.y, a.x, a.y, b.x, b.y)
    areadiff = abs(area - sum([area1, area2, area3])) < EPSILON
    return areadiff
```

```python
def triang_area(x1, y1, x2, y2, x3, y3):
    return abs((x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0)


def triang_sum(x1, y1, x2, y2, x3, y3):
    return x1 * (y3 - y2) + x2 * (y1 - y3) + x3 * (y2 - y1)


def heron_area(triangles):
    result = []
    for triangle in triangles:
        sides = []
        for i in range(3):
            next_index = (i + 1) % 3
            pt = triangle[i]
            pt2 = triangle[next_index]
            # Distance between two points
            side = math.sqrt(math.pow(pt2[0] - pt[0], 2) + math.pow(pt2[1] -
pt[1], 2))
            sides.append(side)
        c, b, a = sorted(sides)
        area = .25 * math.sqrt(abs((a + (b + c)) * (c - (a - b)) * (c + (a -
b)) * (a + (b - c))))
        result.append((area, a, b, c))
    triangle_area = sum(tri[0] for tri in result)
    return triangle_area

def wire_frame_pol(polygon):
        x = [i[0] for i in polygon]
        y = [i[1] for i in polygon]
        df = pd.DataFrame({'x': x, 'y': y})
        return df

def wire_frame_triang(triangles):
    x_start = []
    x_end = []
    y_start = []
    y_end = []
    for triangle in triangles:
        for i, pt in enumerate(triangle):
            next_index = (i + 1) % 3
            x_start.append(pt[0])
            x_end.append(triangle[next_index][0])
            y_start.append(pt[1])
            y_end.append(triangle[next_index][1])
    df = pd.DataFrame({'x': x_start, 'y': y_start, 'xend': x_end, 'yend':
y_end})
```

```
    return df

rand_poly = [(200, 180), (180, 190), (165, 210), (165, 235), (150, 250), (100,
170), (120, 120), (150, 100), (170, 140), (130, 150)]
df = wire_frame_pol(rand_poly)
(ggplot(df, aes(x='x', y='y')) + geom_polygon())

triangles = earclip(rand_poly)
df = wire_frame_triang(triangles)
(ggplot(df, aes(x='x', y='y')) + geom_point(color='red') +
geom_segment(aes(x='x', y='y', xend='xend', yend='yend')))
```

Output:

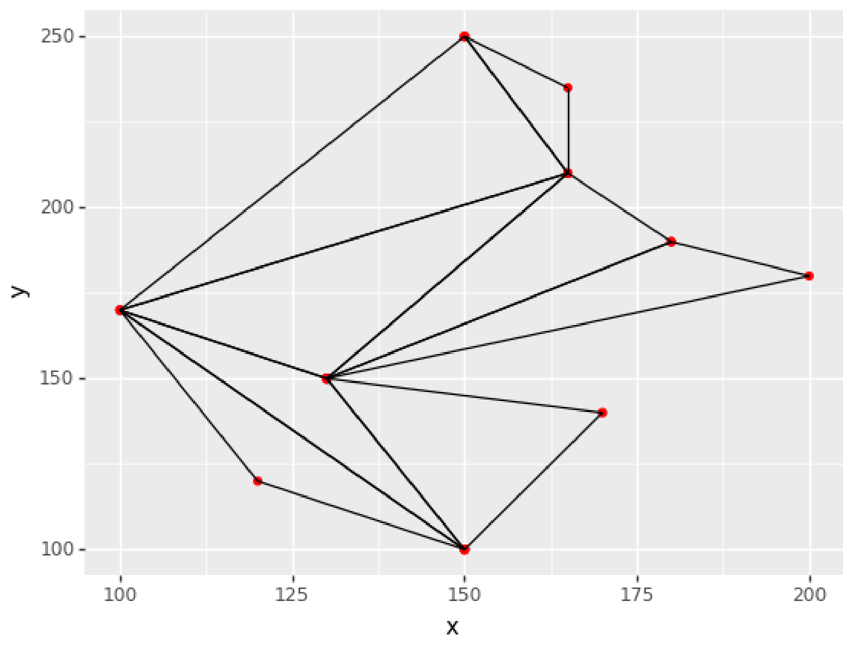

Fig.4: Output of the Triangulation code

- **Delaunay triangulation:**

Delaunay triangulation is a method of making a set of triangles from a discrete set of vertices so that no vertex lies inside the circumcircle of any triangle in the set. A circumcircle of a triangle is a circle passing through all its vertices.

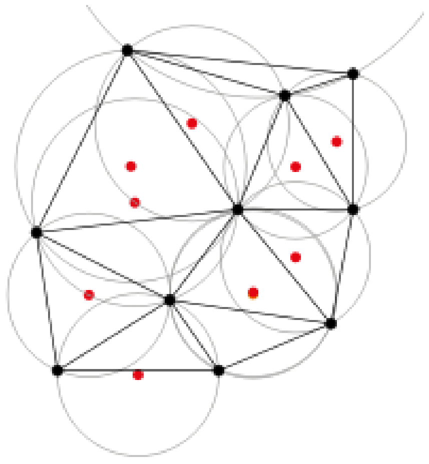

Fig.5: Delaunay triangulation

To understand the working of the algorithm, let us consider 4 random points and then:
i. Consider a triangle which contains all the points in its interior, name it as T.
ii. Select a point and look for triangles whose circumcircle contains the selected point.
iii. Connect all the edges to make a new set of triangles and repeat the same process for all the points.
iv. Finally, delete the triangle T and triangles involving T to get the desired triangulation.

Time complexity: In Delaunay triangulation, using the divide and conquer algorithm, we recursively draw a line to split the vertices into two sets. The Delaunay triangulation is computed for each set, and then the two sets are merged along the splitting line. So the total running time is O($n \log n$).

Implementation: I have implemented Delaunay triangulation in Python using the SciPy spatial library of Delaunay triangulation.

The code for Delaunay triangulation in Python is:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import Delaunay

vertices = np.array([[200, 180], [180, 190], [165, 210], [165, 235], [150,
250], [100, 170], [120, 120], [150, 100], [170, 140], [130, 150]])
tri = Delaunay(vertices)
plt.triplot(vertices[:,0], vertices[:,1], tri.simplices)
plt.plot(vertices[:,0], vertices[:,1], 'o')
plt.show()
```

Output:

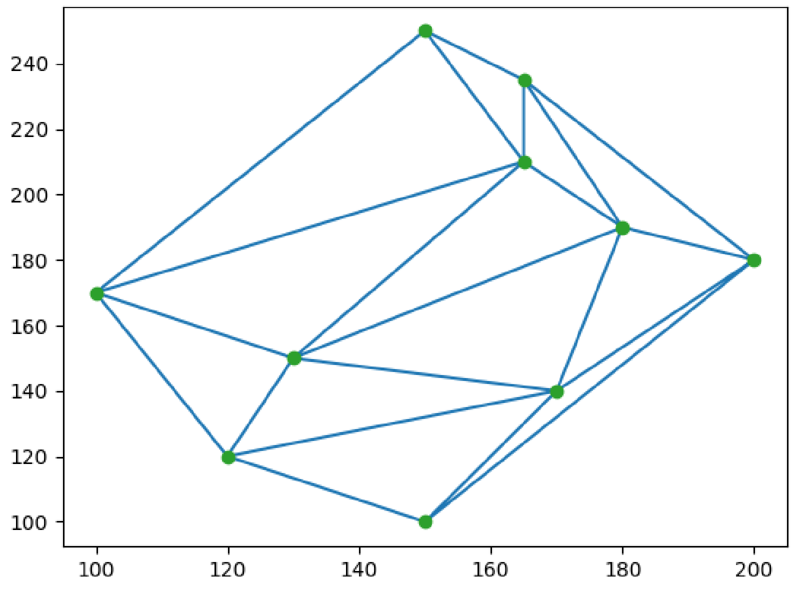

Fig.6: Output of Delaunay triangulation in Python

- **Triangulation of a non-monotone polygon: (Algorithm)**

Triangulating a non-monotone polygon involves two major steps:
i. Decomposing the polygon into y-monotone parts, i.e., into polygons P such that an intersection of any horizontal line L with P is connected.
ii. Triangulating the monotone subpolygons in linear time. (Ear-clipping algorithm can be used).

Monotone partitioning:

We can implement a line-sweep (top down) algorithm to decompose any polygon into monotone polygons.

Classifying vertices:

To have y-monotone pieces, we need to get rid of turn vertices:

a. When we encounter a turn vertex, it might be necessary to introduce a diagonal and split the polygon into subpolygons.
b. But it is not required to add diagonal at every turn vertex.

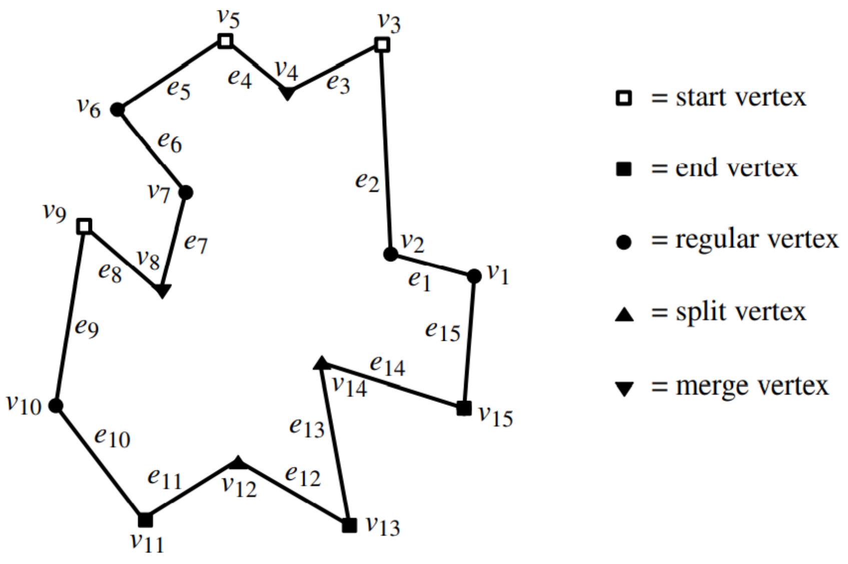

Fig.7: Types of vertices in a non-monotone polygon

Adding diagonals:

To partition P into y-monotone pieces, get rid of split and merge vertices:

a. We need to add a diagonal going upward from each split vertex.
b. Also, we need to add a diagonal going downward from each merge vertex.

Let helper(e_j) be the lowest vertex above the sweep-line such that the horizontal segment connecting the vertex to e_j lies inside P.

a. For a split vertex v_i, let e_j be the edge immediately to the left of it. We need to add a diagonal from v_i to helper(e_j).

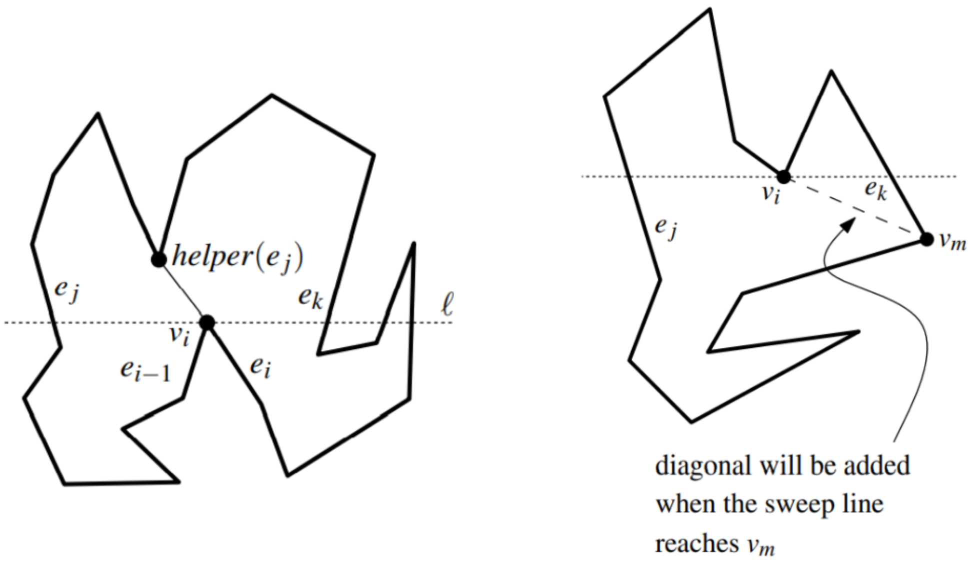

Fig.8: Eliminating Split and Merge vertices

b. For a merge vertex v_i, let e_j be the edge immediately to the left of it. Now, as the sweep line moves down, v_i becomes a helper(e_j).

Whenever the helper(e_j) is replaced by some vertex v_m, add a diagonal from v_m to v_i.
If v_i is never replaced as helper(e_j) , we can connect it to the lower endpoint of e_j.

In this way, when the sweep-line passes over the whole polygon, the entire polygon is decomposed into y-monotone subpolygons which can be triangulated in linear time O(n).

Time complexity: The time complexity of the algorithm mentioned above is O(nlogn) using O(n) spaces.