CS 603 Project

Nenavath Preetham, 210050108
Ramavath Sai Srithan, 210050132

1 Part (a)

Given a set of points P in the plane, implement the quadtree data structure.
Input: A finite set P of points in R^2.
Output: A quadtree decomposition of the bounding square of P, with each leaf cell containing at most a constant number of points.

1.1 Algorithm

Algorithm 1 Algo: Quadtree Construction for Point Set

Input: Set of n points P in $\mathbb{R}^2$, threshold k
Output: Root node of the quadtree with all leaf nodes having at most k points

Function `RecursiveSubdivide`(*node, k*)**:**
  if *number of points in node* $\leq k$ **then**
    return
  Divide the node's region into 4 equal quadrants **foreach** *quadrant* q **do**
    $p_q \leftarrow$ points from node that lie inside q
    child $\leftarrow$ `Node`(bounding box of q, p_q)
    `RecursiveSubdivide`(*child, k*)
    Add *child* to node.children

Main Construction:
  Compute the bounding square covering all points in P
  root $\leftarrow$ `Node`(bounding square, P) `RecursiveSubdivide`(*root, k*)
  return *root*

1.2 Implementation

The following code is for recursive sub_divison

```
def recursive_subdivide(node, k):
   if len(node.points)<=k:
       return

   w_ = float(node.width/2)
   h_ = float(node.height/2)

   p = contains(node.x0, node.y0, w_, h_, node.points)
```

```
    x1 = Node(node.x0, node.y0, w_, h_, p)
    recursive_subdivide(x1, k)

    p = contains(node.x0, node.y0+h_, w_, h_, node.points)
    x2 = Node(node.x0, node.y0+h_, w_, h_, p)
    recursive_subdivide(x2, k)

    p = contains(node.x0+w_, node.y0, w_, h_, node.points)
    x3 = Node(node.x0 + w_, node.y0, w_, h_, p)
    recursive_subdivide(x3, k)

    p = contains(node.x0+w_, node.y0+h_, w_, h_, node.points)
    x4 = Node(node.x0+w_, node.y0+h_, w_, h_, p)
    recursive_subdivide(x4, k)

    node.children = [x1, x2, x3, x4]
```

The next is QuadTree Class implementation:

```
class QTree():
    def __init__(self, k, n, points):
        self.points = points
        self.threshold = k

        min_x = min(p.x for p in self.points)
        max_x = max(p.x for p in self.points)
        min_y = min(p.y for p in self.points)
        max_y = max(p.y for p in self.points)

        side_len = max(max_x-min_x, max_y-min_y)
        self.root = Node(min_x, min_y, side_len, side_len, self.points)


    def get_points(self):
        return self.points

    def subdivide(self):
        recursive_subdivide(self.root, self.threshold)

    def graph(self):
        fig = plt.figure(figsize=(12, 8))
        plt.title("Quadtree")
        c = find_children(self.root)
        print("Number of segments: %d" %len(c))
        areas = set()
        for el in c:
            areas.add(el.width*el.height)
        print("Minimum segment area: %.3f units" %min(areas))
        for n in c:
```

```
        plt.gcf().gca().add_patch(patches.Rectangle((n.x0, n.y0), n.width, n.height, fi
    x = [point.x for point in self.points]
    y = [point.y for point in self.points]
    plt.plot(x, y, 'ro') # plots the points as red dots
    plt.show()
    return
```

1.3 Results

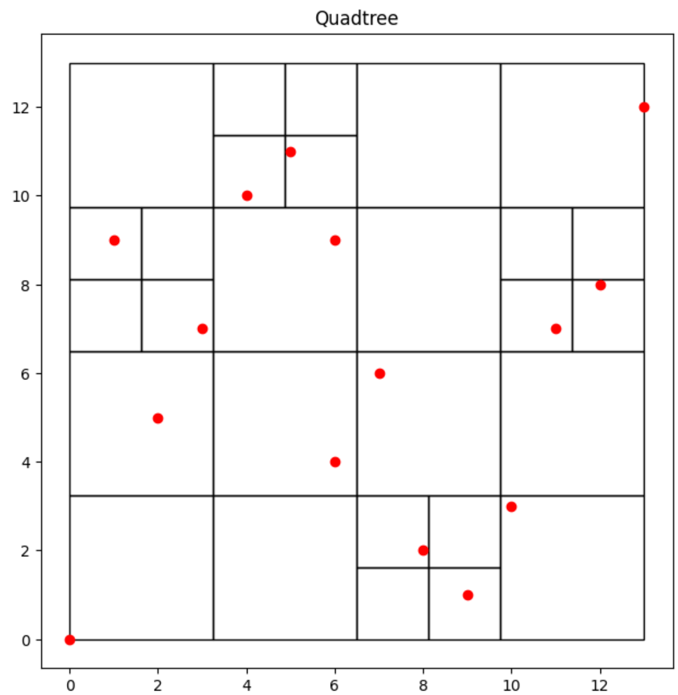

Figure 1: QuadTree of 15 random points

2 Part (b)

Given a set of non-overlapping octilinear polygons in a square domain, implement a quadtree-based triangular mesh generator that produces a valid, non-uniform triangulation. This part extends the quadtree from Part (a) to handle polygon boundaries and generate a triangulation.

2.1 Problem Description

- **Input:** A set of disjoint octilinear polygons with integer coordinates, all contained within a square domain $[0, U] \times [0, U]$, where U is a power of 2. The total polygon perimeter is $p(S)$.
- **Output:** A triangular mesh covering the domain that:
  - Conforms exactly to all polygon boundaries and avoids hanging nodes.
  - Is adaptive (finer triangles near boundaries, coarser elsewhere)

2.2 Algorithm Overview

1. Create a quadtree by recursively subdividing regions intersecting polygon boundaries.
2. Balance the tree using a 2:1 constraint on neighboring cells.
3. Triangulate each leaf based on whether it intersects a polygon or lies fully inside.

2.3 Quadtree Subdivision (Pseudocode)

Algorithm 2 Polygon-aware Quadtree Subdivision

Input: Node n, Polygon list S

Output: Quadtree with refinement near boundaries

```
if node intersects polygon boundary then
    if node width ¿ min size then
        Subdivide node into 4 children  Recurse on each child
else
    if node width ¿ coarse threshold then
        Subdivide node  Recurse on each child
```

2.4 Balancing Phase

To ensure a well-behaved quadtree, we enforce a 2:1 balance condition between adjacent nodes during the balancing phase. The key steps are:

- Initialize a queue with all leaf nodes and iteratively process each node.
- If a neighbor is a leaf and more than one level shallower, subdivide it and add its children back into the queue.
- Repeat the process until no further subdivisions are needed and all adjacent leaves are balanced.

2.5 Triangulation

Each leaf cell is triangulated as follows:

- If it intersects a polygon interior: apply alternating diagonal splits.
- If it doesn't intersect polygon interior: Add a diagonal to quadtree cell

2.6 Code Snippets

Triangulation logic:

```
if interior_intersected:
    if (cell_x + cell_y) % 2 == 0:
        triangles.append((sw, se, ne))
        triangles.append((sw, ne, nw))
    else:
        triangles.append((sw, se, nw))
        triangles.append((se, ne, nw))
```

Neighbor finding logic (used in balancing):

```
def find_neighbors(self, node):
    neighbors = []
    directions = [
        (0, 1), (0, -1), (1, 0), (-1, 0),
        (1, 1), (-1, 1), (1, -1), (-1, -1)
    ]
    for dx, dy in directions:
        check_x = node.x0 + (0.5 + dx * 0.6) * node.width
        check_y = node.y0 + (0.5 + dy * 0.6) * node.height
        neighbor = self._find_leaf_containing_point(check_x, check_y)
        if neighbor and neighbor != node:
            neighbors.append(neighbor)
    return neighbors
```

2.7 Time Complexity Analysis

Let $p(S)$ be the total perimeter of polygons, and U be the domain size (power of 2).

- **Quadtree build:** $O(p(S) \cdot \log(U))$ subdivisions due to recursive splitting near polygon boundaries.
- **Balancing:** Refinement spreads only locally. Still $O(p(S) \cdot \log^2(U))$.
- **Triangulation:** One triangle per leaf cell. $O(p(S) \cdot \log(U))$ total triangles.

$$\boxed{\text{Total Time: } O(p(S) \cdot \log^2(U))}$$

2.8 Quadtree Creation

The `create_quadtree` method builds a quadtree by selectively subdividing cells that intersect polygon boundaries:

```
def _recursive_subdivide(self, node, polygons, U):
    min_size = 1
    coarse_threshold = U // 4

    boundary_intersection = self.check_if_intersects_boundary(node, polygons)

    if boundary_intersection:
        if node.width > min_size:
            children = node.subdivide()
            for child in children:
                self._recursive_subdivide(child, polygons, U)
    elif node.width > coarse_threshold:
        children = node.subdivide()
        for child in children:
            self._recursive_subdivide(child, polygons, U)
```

Complexity Analysis:

- The algorithm starts with a single cell of size $U \times U$.
- Each subdivision divides a cell into 4 smaller cells (quadrants).
- The maximum depth of subdivision is $\log_4(U^2) = \log_2 U$.
- Subdivision is focused on cells that intersect polygon boundaries.
- The number of such cells is proportional to the perimeter $p(S)$.
- For each level of refinement, approximately $\mathcal{O}(p(S))$ cells are processed.

$$\text{Overall complexity: } \mathcal{O}(p(S)\log_2 U)$$

2.9 Quadtree Balancing

The `balance` method ensures a 2:1 size ratio between adjacent cells:

```
def balance(self):
    changed = True
    while changed:
        changed = False
        leaves = self.get_leaves()
        queue = deque(leaves)
        while queue:
            node = queue.popleft()
            neighbors = self.find_neighbors(node)
            for neighbor in neighbors:
                if neighbor.is_leaf and neighbor.depth < node.depth - 1:
                    children = neighbor.subdivide()
                    queue.extend(children)
                    changed = True
    return self
```

Complexity Analysis:

- In the worst case, the balancing phase might add additional cells.
- The total number of nodes are of $\mathcal{O}(p(S) * log(U))$.
- The number of balancing iterations is bounded by the depth of the tree ($\log_2 U$).

$$\text{Overall complexity: } \mathcal{O}(p(S)\log(U) * log_2(U) \implies \mathcal{O}(p(S) * log^2(U))$$

2.10 Triangulation

```
        if interior_intersected:
            if (cell_x + cell_y) % 2 == 0:
                triangles.append((sw, se, ne))
                triangles.append((sw, ne, nw))
            else:
                triangles.append((sw, se, nw))
                triangles.append((se, ne, nw))
```

Complexity Analysis:

- Each leaf node generates a constant number of triangles (typically 2 or 4).
- The number of nodes is $\mathcal{O}(p(S)\log_2 U)$ due to the quadtree structure.
- The vertex map lookup operations are $\mathcal{O}(1)$ on average.

$$\text{Triangulation complexity: } \mathcal{O}(p(S)\log_2 U) \quad \text{Output size: } \mathcal{O}(p(S)\log U)$$

2.11 Results

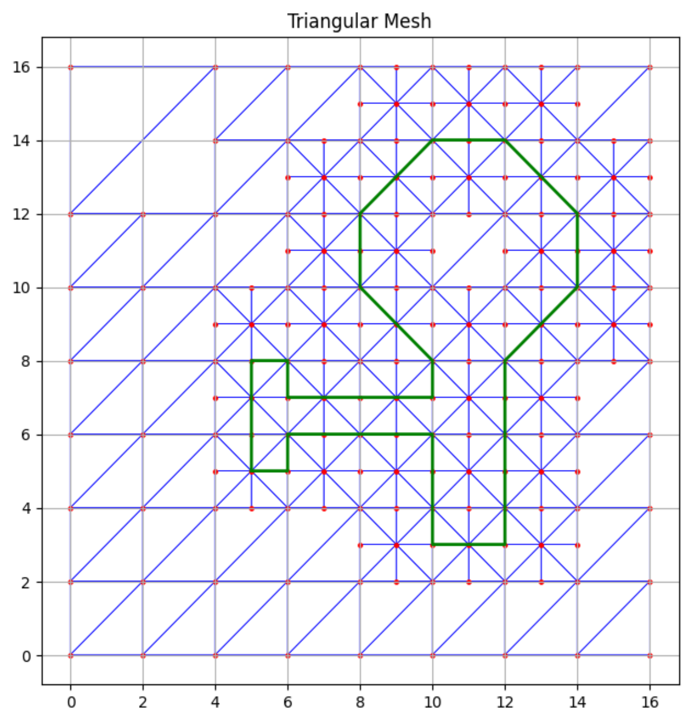

Figure 2: Triangulation of a basic octagon

2.11.1 Observation

- If the Quadtree cell is intersected by any edge is divided into 4 subcells.
- If not, we treat the cell as leaf node and triangulate it by adding a diagonal.
- Hence, we can see that the triangulation is finer near the boundaries of the polygon and is coarser elsewhere, we can see this in further examples next page.

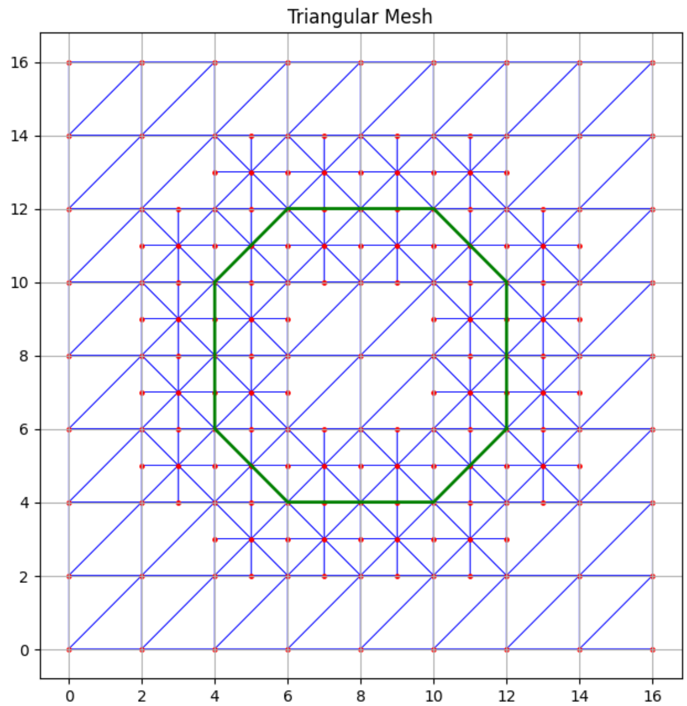

Figure 3: Triangulation of a key-shaped polygon

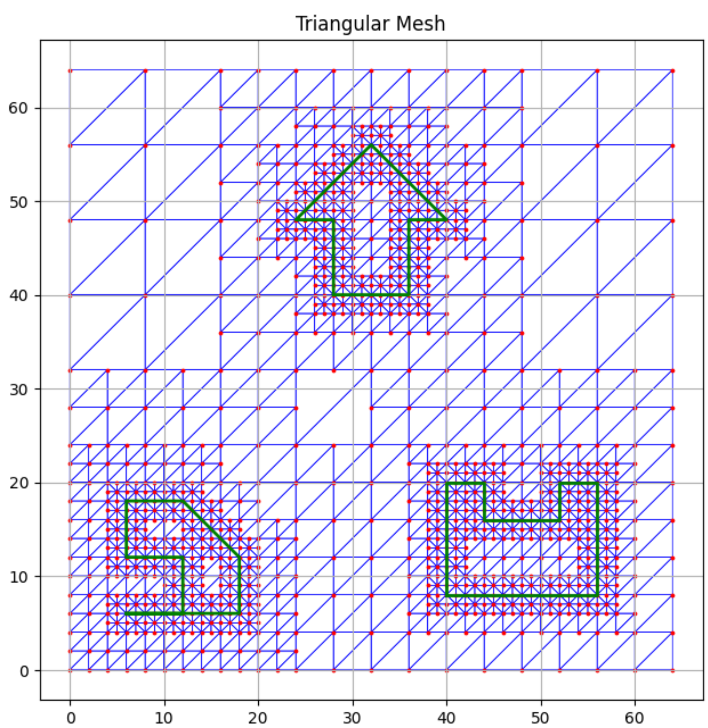

Figure 4: Multiple disjoint polygonal regions