Project Report: Polygon Triangulation

Harsh Vardhan (210050064), Hitesh Kumar (210050066)

Part A

This report analyzes the implementation of a polygon triangulation algorithm using the Doubly Connected Edge List (DCEL) data structure. The algorithm consists of two main phases:

1. Partitioning a simple polygon into monotone pieces
2. Triangulating each monotone piece

1 Data Structures

1.1 DCEL Components

The implementation uses three core classes to represent the DCEL:

- **Vertex Class**
  - Stores coordinates (x, y)
  - Contains a pointer to an outgoing half-edge
  - Classifies vertex type (START, END, SPLIT, MERGE, REGULAR)
  - Provides geometric operations and comparisons
- **HalfEdge Class**
  - Represents directed edges in the DCEL
  - Contains pointers to twin, next, and previous edges
  - Maintains references to origin vertex and parent edge
- **Edge Class**
  - Represents undirected edges
  - Contains references to both vertices
  - Stores a pointer to one of its half-edges

1.2 DCEL Class

The DCEL class manages the complete representation:

- Stores vectors of vertices, edges, and half-edges
- Maintains face information
- Implements a binary search tree for sweep line algorithm
- Provides construction from vertex lists and edge lists

2 Algorithm Implementation

2.1 Monotone Partitioning

Implemented in `Monotone.cpp` using a sweep line algorithm:

1. **Initialization**
   - Processes vertices to determine their types (start, end, split, merge, regular)
   - Sorts vertices by y-coordinate (with x as tie-breaker)
2. **Sweep Line Processing**
   - Uses a priority queue to process vertices from top to bottom
   - Maintains a binary search tree of active edges
   - Handles each vertex type differently:
     - **Start Vertex**: Adds edge to status structure
     - **End Vertex**: Removes edge and potentially adds diagonal
     - **Split Vertex**: Adds diagonal to helper of edge to the left
     - **Merge Vertex**: Adds diagonal to helper of edges to the left and right
     - **Regular Vertex**: Handles based on interior position
3. **Diagonal Addition**
   - Adds diagonals to split the polygon into monotone pieces
   - Creates new edges in the DCEL

2.2 Monotone Triangulation

Implemented in `Triangulate.cpp`:

1. **Face Processing**
   - Processes each face (monotone polygon) separately

- Orders vertices by y-coordinate (with x as tie-breaker)

2. **Stack-Based Triangulation**
    - Uses a stack to maintain vertices on one side of the polygon
    - Processes vertices in order:
        - If on opposite sides of the chain, adds diagonals to all stack vertices
        - If on the same side, adds diagonals while maintaining convexity
    - Adds final diagonals to complete triangulation
3. **New Edge Creation**
    - Combines original edges with new diagonals
    - Constructs a new DCEL with the triangulated mesh

3 Key Functions and Methods

3.1 Vertex Classification (`DCEL.cpp`)

- `set_type()`: Determines vertex type based on neighbors
- Uses `is_below()` and `counter_clockwise()` for geometric tests

3.2 DCEL Construction (`DCEL.cpp`)

- Two constructors:
    1. From simple polygon (vertex list)
    2. From general graph (vertex list + edge list)
- Handles edge ordering and half-edge linking

3.3 Sweep Line Operations (`Monotone.cpp`)

- `add_search_edge()` / `remove_search_edge()`: Manage active edges
- `find_previous_index()`: Locates edge to the left of a point

3.4 Triangulation (`Triangulate.cpp`)

- Processes each face independently
- Uses vertex ordering and stack to add diagonals

4 Geometric Predicates

Several key geometric operations are used throughout:

- `counter_clockwise()`: Determines orientation of three points
- `is_below()`: Compares vertex positions
- `get_angle()`: Computes polar angle for radial sorting
- Various comparison operators for vertex/edge ordering

5 Results and Complexity Analysis

The implementation successfully realizes the polygon triangulation algorithm with the following characteristics:

5.1 Algorithmic Correctness

The implementation follows the standard algorithms for polygon triangulation:

1. First partitions the polygon into monotone pieces using a sweep line algorithm
2. Then triangulates each monotone piece using a greedy stack-based approach

The DCEL data structure effectively maintains the topological relationships between vertices, edges, and faces throughout the process. The code demonstrates proper handling of all vertex types and edge cases in the partitioning phase, and efficiently processes the monotone polygons in the triangulation phase.

5.2 Time Complexity Analysis

The overall time complexity of the algorithm is $O(n \log n)$, which can be broken down as follows:

- **Monotone Partitioning Phase**:
  - Sorting vertices: $O(n \log n)$
  - Processing each vertex in sweep line:
    - BST operations (insert/delete/search): $O(\log n)$ per operation
    - n vertices processed $\Rightarrow O(n \log n)$ total
  - Adding diagonals: $O(1)$ per diagonal, $O(n)$ total
- **Triangulation Phase**:
  - Sorting vertices within each monotone polygon: $O(n \log n)$ total
  - Stack operations: $O(1)$ per vertex, $O(n)$ total

  - Diagonal addition: $O(1)$ per diagonal, $O(n)$ total

This leads to the overall complexity:

$$T(n) = O(n \log n) + O(n \log n) + O(n) = O(n \log n)$$

5.3 Space Complexity

The space complexity is $O(n)$ due to:

- DCEL storage: $O(n)$
- Priority queue: $O(n)$
- Status structure: $O(n)$
- Stack for triangulation: $O(n)$

5.4 Theoretical Validation

The implementation maintains the theoretical guarantees of Seidel's algorithm:

- Correctness through proper handling of all vertex types (start, end, split, merge, regular)
- Optimal $O(n \log n)$ time complexity
- Robustness through careful geometric predicate implementation

The implementation achieves $O(n \log n)$ time complexity while correctly handling all cases of simple polygon triangulation, validating both its theoretical foundations and practical utility.

Part B

This report analyzes the implementation of a polygon triangulation algorithm using the Doubly Connected Edge List (DCEL) data structure. The algorithm is based upon Raimund Seidel's 1991 paper — *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons* [1], available at this link.

This is a **detailed explanation** of the implemented **polygon triangulation algorithm**, which handles both **simple polygons** (no holes) and **polygons with holes**. The algorithm follows a **trapezoidal decomposition** approach followed by **monotone mountain triangulation**, ensuring $O(n \log^* n + h \log h)$ time complexity for simple polygons and **subquadratic time** for polygons with holes. $\log^*$ is the iterated logarithm

6 Algorithm Overview

The algorithm consists of **four main phases**:

1. **Trapezoidal Decomposition** (Partitioning the polygon into trapezoids)
2. **Selecting Inside Trapezoids** (Filtering out trapezoids outside the polygon)
3. **Forming Monotone Mountains** (Converting trapezoids into monotone polygons)
4. **Triangulating Monotone Mountains** (Splitting each mountain into triangles)

7 Detailed Algorithm Steps

7.1 Trapezoidal Decomposition

The goal is to **partition the polygon into trapezoids** using a **randomized incremental algorithm** (expected $O(n \log n)$ time).

Key Steps:

- Randomize the edges of the polygon.
- Insert vertices into the search structure.
- Insert edges, splitting trapezoids as necessary.

Pseudocode:

```
def trapezoidation(polygonal_area: PolygonalArea) -> list[Trapezoid]:
    edges: list[Edge] = polygonal_area.get_edges()

    search_tree = Node(trapezoid=Trapezoid())
    already_inserted: set[Vertex] = set()

    def insert_vertex_if_necessary(vertex: Vertex) -> bool:
        if vertex in already_inserted:
            return False

        search_tree.insert_vertex(vertex)
        already_inserted.add(vertex)
        return True

    for edge in edges:
        top_just_inserted = insert_vertex_if_necessary(edge.top_vertex)
        bottom_just_inserted = insert_vertex_if_necessary(edge.bottom_vertex)

        search_tree.insert_edge(edge, top_just_inserted, bottom_just_inserted)
    return search_tree.get_all_traps()
```

Search Structure:

- **Vertex nodes** split space into above/below regions.
- **Edge nodes** split space into left/right regions.
- **Trapezoid nodes** are leaves representing final regions.

Edge Insertion:

Edges are inserted by locating intersected trapezoids and splitting them.

7.2 Selecting Inside Trapezoids

After decomposition, trapezoids outside the polygon are filtered using the **odd-even rule**.

Key Idea:

A trapezoid is **inside** if crossing its left edge toggles the inside/outside state.

```
def select_inside_trapezoids(all_trapezoids: list[Trapezoid]) -> list[Trapezoid]:
    return [trap for trap in all_trapezoids if trap.is_inside]
```

How `is_inside` Works:

- If no left/right edge exists, the trapezoid is outside.
- Otherwise, check the left neighbor's state and alternate accordingly.

7.3 Forming Monotone Mountains

A **monotone mountain** is a polygon with one base edge and a chain of strictly increasing or decreasing vertices.

Key Steps:

- Group vertices by base edges.
- Construct vertical monotone chains.

```
def make_monotone_mountains(trapezoids: list[Trapezoid]) -> list[MonotoneMountain
    ]:
    above_vertex_by_base_edge: dict[Edge, dict[Vertex, Vertex]] = (
        group_vertices_by_mountain(trapezoids)
    )

    monotone_mountains: list[MonotoneMountain] = []

    for base, above_vertex_mapping in above_vertex_by_base_edge.items():
        below_monotone_vertex: MonotoneVertex | None = None
        current_vertex = base.bottom_vertex
        monotone_mountain_created = False

        while current_vertex is not None:
            current_monotone_vertex = MonotoneVertex(
                vertex=current_vertex, below=below_monotone_vertex
            )
            if below_monotone_vertex:
                below_monotone_vertex.above = current_monotone_vertex

            above_vertex = above_vertex_mapping.get(current_vertex, None)
            current_vertex = above_vertex
            below_monotone_vertex = current_monotone_vertex

            if not monotone_mountain_created:
                monotone_mountains.append(
                    MonotoneMountain(current_monotone_vertex, base)
                )
                monotone_mountain_created = True

    return monotone_mountains
```

How It Works:

Trapezoids contribute left/right edges. Vertices are linked vertically to form chains. The base edge becomes the degenerate side.

7.4 Triangulating Monotone Mountains

Monotone mountains are triangulated in $O(n)$ time using a greedy approach.

Key Steps:

- Process vertices in order.
- Use a stack to detect convex angles.

- Clip ears to form triangles.

```
def triangulate_monotone_mountain(mountain: MonotoneMountain, triangles: list[
    Triangle]):
    if mountain.is_degenerated(): return

    first_non_base_vertex = cast(MonotoneVertex, mountain.bottom_vertex.above)

    convex_order = counter_clockwise(mountain.base.top_vertex, mountain.base.
        bottom_vertex, first_non_base_vertex.vertex,)

    current_vertex = first_non_base_vertex

    while not current_vertex.is_base_vertex():
        below = cast(MonotoneVertex, current_vertex.below)
        above = cast(MonotoneVertex, current_vertex.above)
        current_vertex_convex = counter_clockwise(below.vertex, current_vertex.
            vertex, above.vertex) == convex_order

        if not current_vertex_convex:
            current_vertex = above
            continue

        vertices_counter_clockwise = (below.vertex, current_vertex.vertex, above.
            vertex) if convex_order else (below.vertex, above.vertex,
            current_vertex.vertex)

        triangles.append(Triangle(vertices_counter_clockwise))
        below.above = above
        above.below = below
        current_vertex = above if below.is_base_vertex() else below
```

Key Concepts:

- Convexity ensures a triangle can be formed.
- Chain is updated after each triangle is formed.

8 Handling Polygons with Holes

The algorithm supports holes since:

- Trapezoidal decomposition respects holes.
- The point-in-polygon test uses parity to detect inside regions.

9 Complexity Analysis

Step	**Time Complexity**	**Space Complexity**
Trapezoidal Decomposition	$O(n \log n)$ (expected)	$O(n)$
Select Inside Trapezoids	$O(n)$	$O(n)$
Monotone Mountain Formation	$O(n)$	$O(n)$
Triangulate Mountains	$O(n)$	$O(n)$
Total (Simple Polygon)	$O(n \log n)$	$O(n)$
Total (With Holes)	**Subquadratic**	$O(n)$

10 Result

This implementation result includes:

- **Efficient triangulation** for polygons with and without holes.
- **Robust handling** of degenerate cases.
- **Modular structure** for maintainability.

References

[1] Raimund Seidel, *A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons*, Computational Geometry: Theory and Applications, Volume 1, Issue 1, 1991, Pages 51–64, https://www.sciencedirect.com/science/article/pii/0925772191900124.