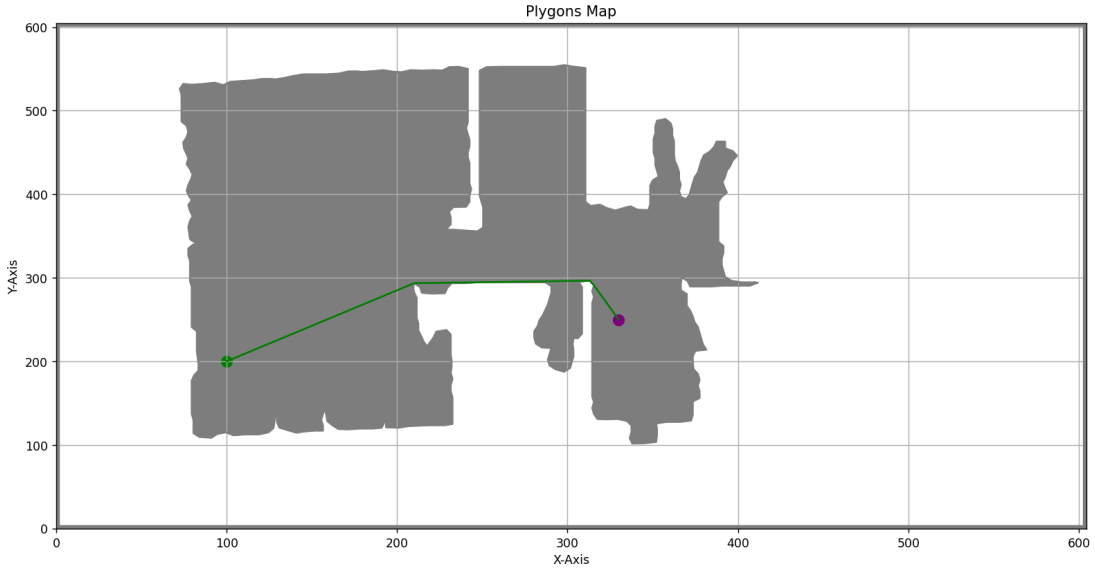
Robotics Lab Project - Report

Submitted by:
Ido peled - 318268299
Noy cohen - 206713307

Goal: Finding the Shortest Path in a given point cloud Room.



Background

Our project focuses on finding the shortest path between two points in a room using various algorithms.

Before diving into the code and implementation details, we want to highlight our primary focus: pathfinding on both convex and non-convex polygonal maps. Throughout the project, we explored this topic extensively, reading several articles and experimenting with different tools, inputs, and map configurations.

Our project consists of a Python-based visualization module and a C++ core module that work together, totaling over 1,500 lines of code. The code is organized into multiple files and classes, which we will explain in detail later.

Our work divided to 3 parts throughout the semester:

1. Random Map Generation and Shortest Path Calculation
2. Transition part- Room Representation with Obstacles
3. Point Cloud Processing- “Real world room”

In the first part of the semester, we concentrated on generating random maps—particularly non-convex polygonal maps with obstacles—and implementing algorithms to find the shortest path between any two points while avoiding obstacles.

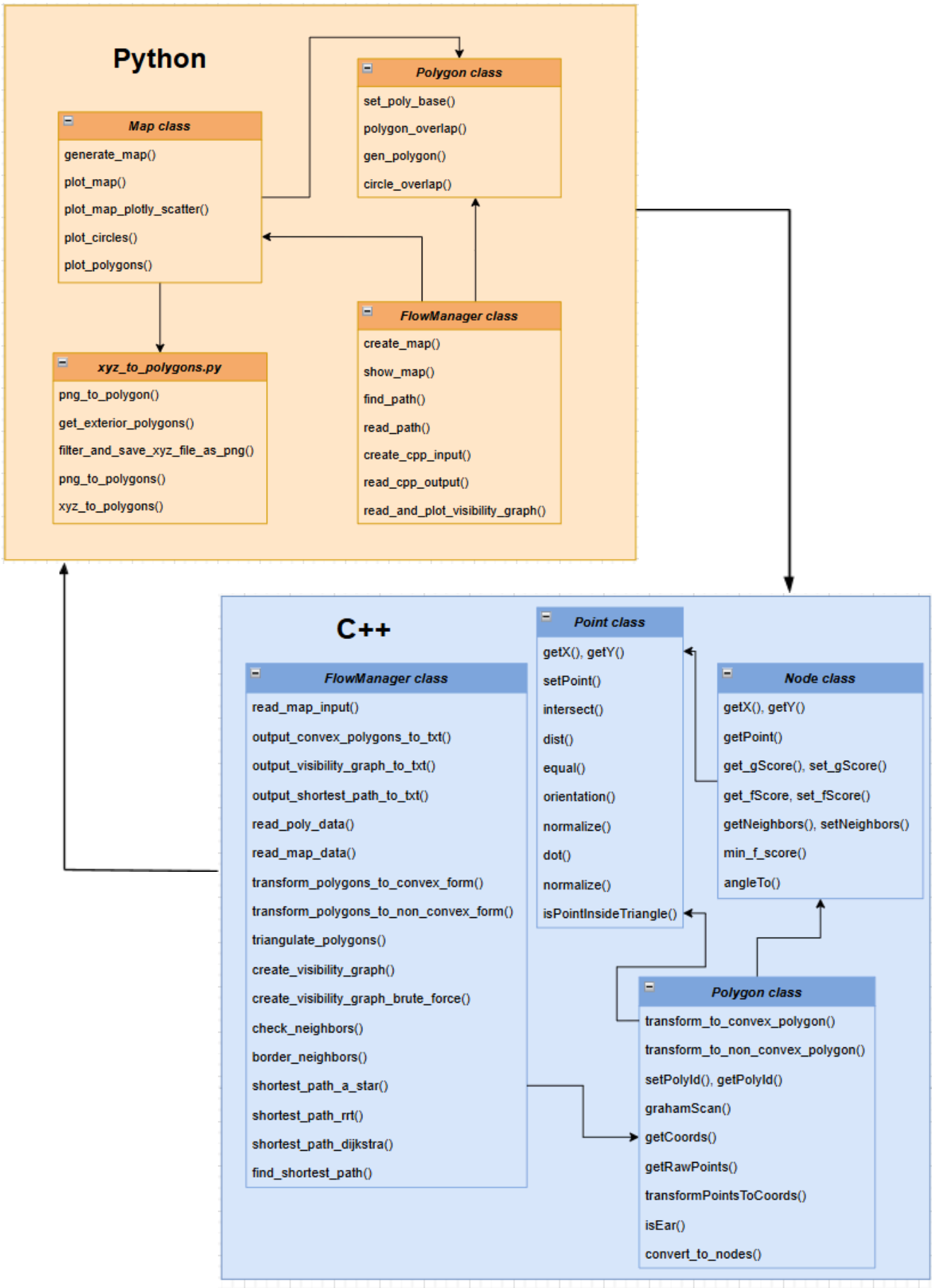
In the third part, following a meeting with Professor Dan Feldman, we expanded the project to handle real-world environments. He provided us with a sample dataset representing a room in the Amir Building lab as an .xyz file. We converted this dataset into a non-convex

polygonal representation with triangulation and applied our algorithms to determine the shortest path between two points within it. We have adjusted the algorithms to work in this new environment.

While we submitted our program using the Amir lab dataset, this report will showcase the full range of capabilities our system offers.

Code structure

Screenshot from .uml file:



Workflow:

The project consists of three main parts:

1. **Random Map Generation and Shortest Path Calculation**
   We developed a software program that generates a randomly populated map with designated start and end points, as well as various obstacles. The program calculates the shortest possible route between the start and end points while avoiding obstacles.

   This part is divided into two stages:

   - A Python script generates a randomly populated map with start and end points, along with polygonal obstacles.
   - The data is transferred from Python to C++ using .txt files.
2. **Transition part- Room Representation with Obstacles**
   We created a randomly generated map that simulates a room with obstacles, demonstrating the algorithm's ability to handle confined spaces.
3. **Point Cloud Processing- "Real world room"**
   We tested the algorithm on a **point cloud (.xyz file)** representing the university lab floor in the **Amir Building, City Campus**, demonstrating its effectiveness in real-world scenarios.

How the 2/3th part works:

Input:

- A point cloud (.xyz file) representing the university lab floor in the Amir Building, City Campus.
- Or a randomly generated map with obstacles.

Output:

- The shortest path between two selected points in the given map.

Program Steps:

1. **Receiving the Input (Python Side)**
    - If a **point cloud** is provided:
        - ➢ Filter the points based on the **z-axis** (keeping values closest to the minimum).
        - ➢ Convert the filtered points into a **binary image**.
        - ➢ Extract the boundary and generate a **polygonal representation**.
    - If a **randomized map** is used:
        - ➢ Generate a random number of polygons.
        - ➢ Randomly generate points to define each polygon.
2. **Passing Data to C++**
    - Python sends the processed map data to C++ via a .txt file.
    - The C++ executable is called to process the data.
3. **Processing in C++**
    - C++ reads the .txt file and parses the data.
    - Perform triangulation on the extracted polygon.
    - Construct the visibility graph using optimized tree-based algorithm ($O(V^2 \log(V+E))$).
4. **Pathfinding Algorithm Selection**
    - Python determines which algorithm to use for shortest path calculation:
        - ➢ A*
        - ➢ Dijkstra
        - ➢ RRT
5. **Returning and Displaying Results**
    - C++ sends the computed path back to Python.
    - Python visualizes the result.

Key Differences Between the the second and the third parts of the Project

In the second part, we had to **adjust our approach** to accommodate real-world point cloud data. Unlike the first part, where we controlled the polygon generation, the second part required filtering noisy data, converting it into a usable polygonal format, and handling irregularities before applying our algorithms.

Feature	First Part: Randomized Maps	Second Part: Real-World Data (Point Cloud)
Map Source	Randomly generated polygons with obstacles	Point cloud (.xyz file) of an actual lab
Obstacle Representation	Defined by manually generated polygons	Extracted from the point cloud data
Processing Method	Triangulation	Triangulation
Visibility Graph	Constructed from randomized polygons	Constructed from real-world extracted polygons
Pathfinding	A*, Dijkstra, RRT	A*, Dijkstra, RRT

Algorithms Used:

- Point cloud processing
- Convex-hull (for first part)
- Triangulation (for 2/3th part)
- Visibility graph
- A*, dijkstra and RRT

Visibility Graphs:

Initially, we used a **brute-force approach** to compute the visibility graph with a time complexity of $\mathbf{O((V+E)^3)}$. However, we later discovered the paper *"An $O(n^2 \log n)$ Algorithm for Computing Visibility Graphs"* and another study that implemented this approach. This led us to adopt and implement the improved $\mathbf{O(n^2 \log n)}$ algorithm, significantly optimizing our computation.

Python:

The python in the project is mostly responsible for the map/room processing and visualization of everything from inputs, processes and final output also.

As you can see in the diagram above it has 3 classes and another .py file for processing the .xyz file input.

Classes roles:

Polygon class: Responsible for generating polygon obstacles on the map. It does this by randomly selecting a circle's base point and radius, then sampling points within the circle to create an obstacle area. These obstacles will later be converted to convex polygons in the C++. The class also ensures that the generated obstacles do not overlap with one another.

Map class: Responsible mainly for visualising any step of the program, and also for creating a non-convex map. With this class we will show the input room and some processing while finding neighbors and also the final path results.

FlowManager class: Responsible for the program flow management. Receiving inputs, creating inputs, communicating with the C++ and storing all the program data to visualize via with Map class.

C++

The C++ is the core code of the program and responsible for all the calculations processes of the map\room input from the python, including searching for valid neighbors, constructing a path, creating inputs back to the python and also processing the obstacles.

It has 4 main classes:

Classes roles:

Point class: Represents a coordinate of x and y, and responsible for all the geometry math calculations related to coordinates (intersection, orientation etc).

Node class: Represents a node in the map which holds a point and his neighbors and also distances. Basically responsible for all the graph logic of the program.

Polygon class: Represents a polygon of nodes, it can be a room polygon or an obstacle polygon.

FlowManager class: Responsible for the program flow managing and main logic of finding neighbors, paths, checking problems and managing inputs and outputs while communicating with the python.

Program Steps Visualization and examples:

1. **Procedural Map Generation (first part of our project)**

We started our work from implementing a feature to generate **random polygonal maps** with obstacles.

- How it works

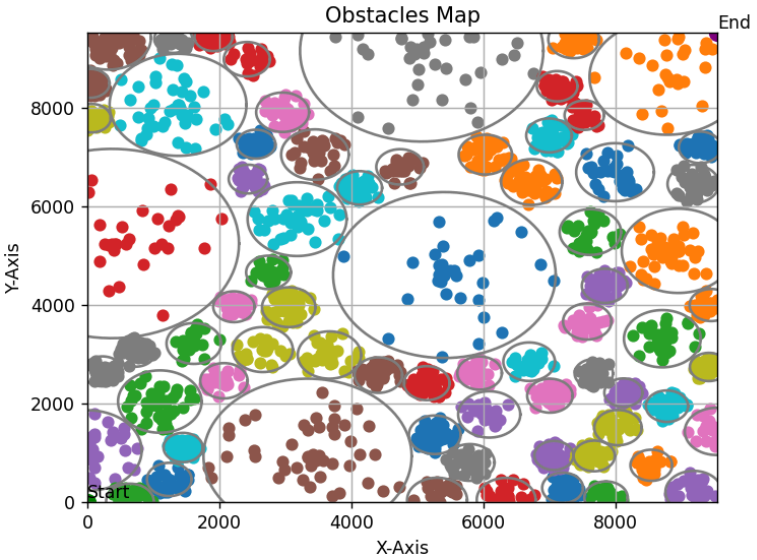
1. **Defining the Map Parameters**
    - Specify the **number of obstacles (polygons)**.
    - Set the **map size, start point, and end point**.
2. **Creating Obstacles**
    - Sample a **random center coordinate** and **radius** for each obstacle.
    - Generate **20-50 random points** inside each circular area.
    - Store the generated points in an array.
3. **Converting to Convex Polygons**
    - The random points are processed in **C++** to form valid polygons.
    - Each set of points is passed to the **Polygon class constructor**.
    - The **Graham Scan algorithm** is applied to convert the points into a **convex polygon** by sorting them based on **clockwise angles**.
    - The convex polygon points are then converted into **Node structures**.

```
Polygon(const std::vector<Point>& pointsVector) {
    std::vector<Point> convex_points = grahamScan(pointsVector);    // create a convex
    this->coords = convert_to_nodes(convex_points);     // converting the convex points to nodes
}
```

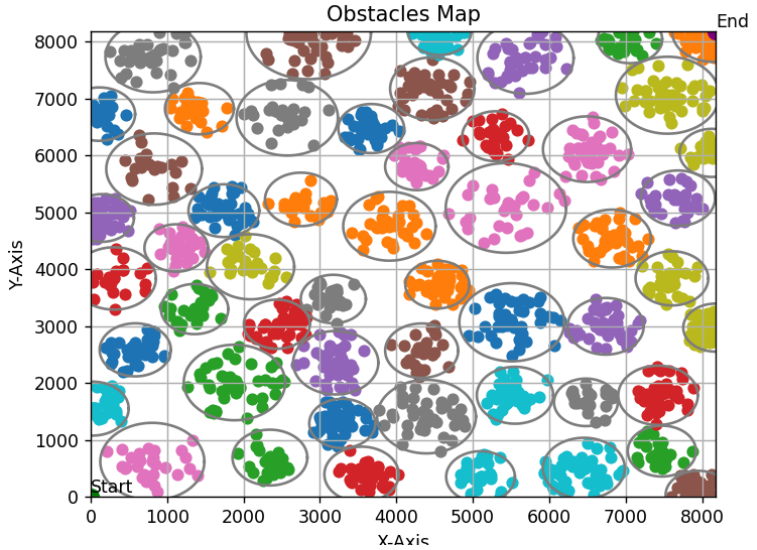
- <u>Obstacle Examples</u>

  Before converting to polygons:

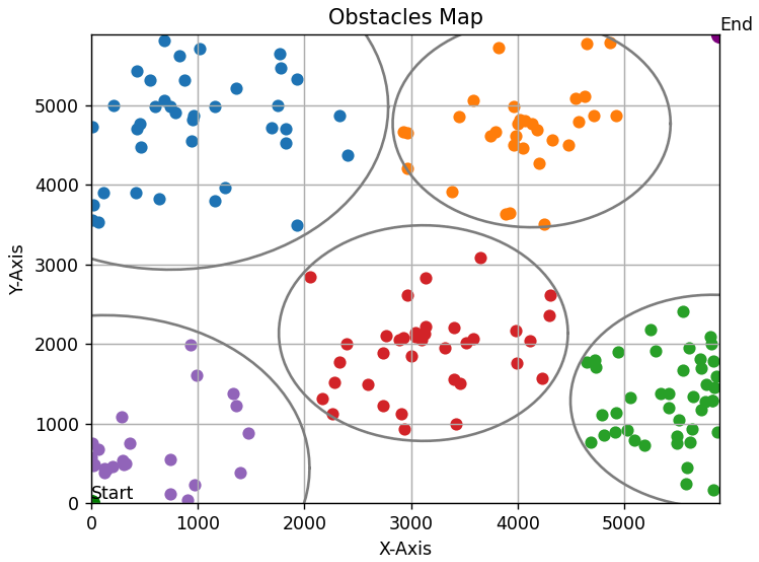
  - Various circle sizes



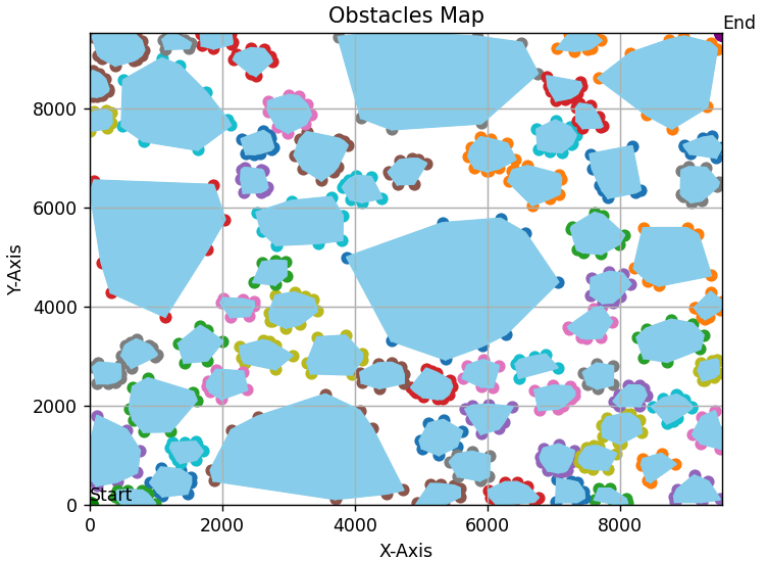
  - 52 small circles:



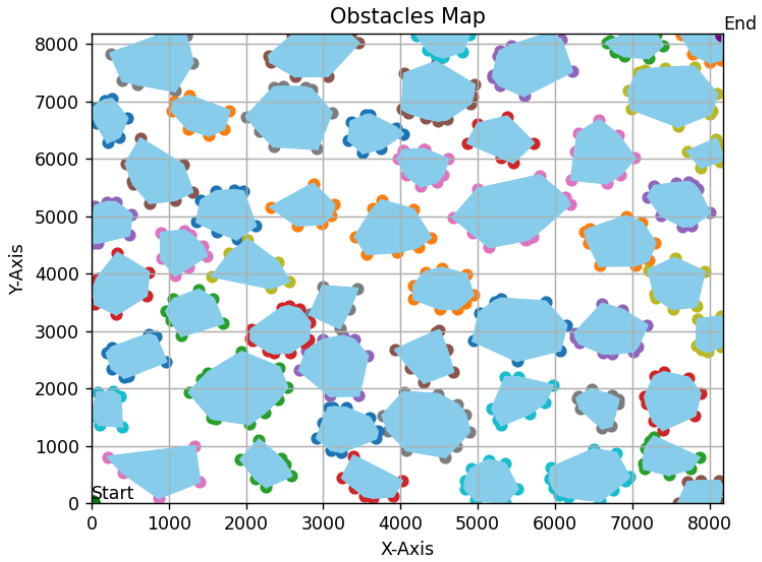
- 5 big circles:



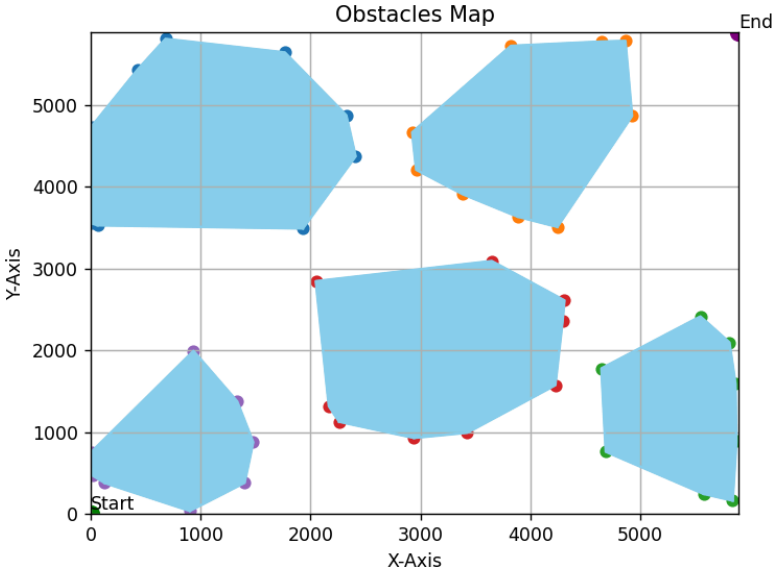
- <u>After applying the **Graham Scan algorithm**, we obtain:</u>
  - Various convex polygon sizes



- 52 small polygons

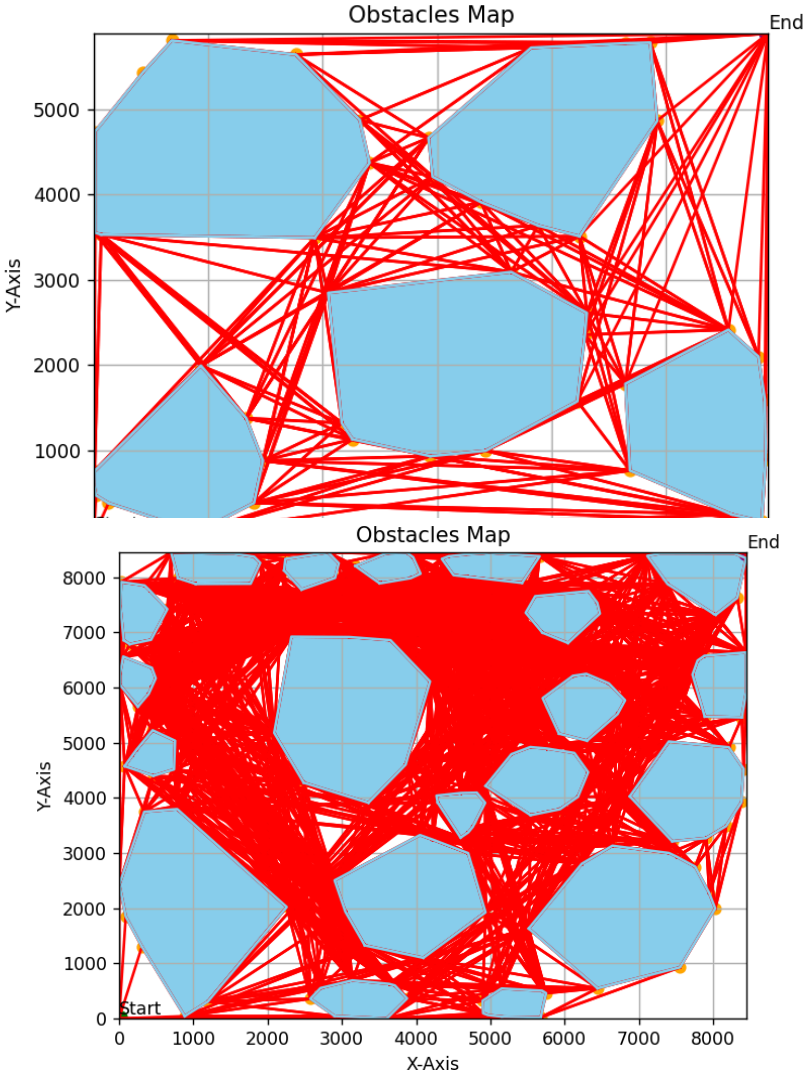


- 5 big polygons:

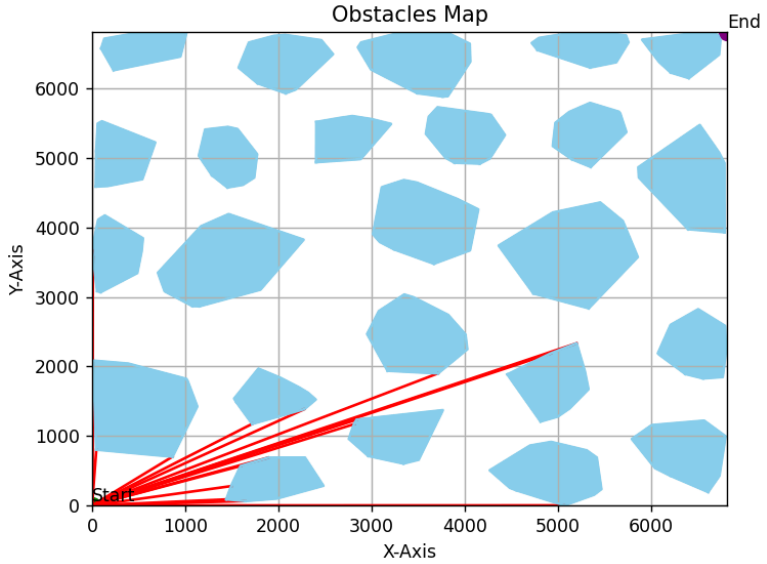


- Visibility Graph for Generated Maps

  For each node, we construct the **visibility graph**, as explained earlier.



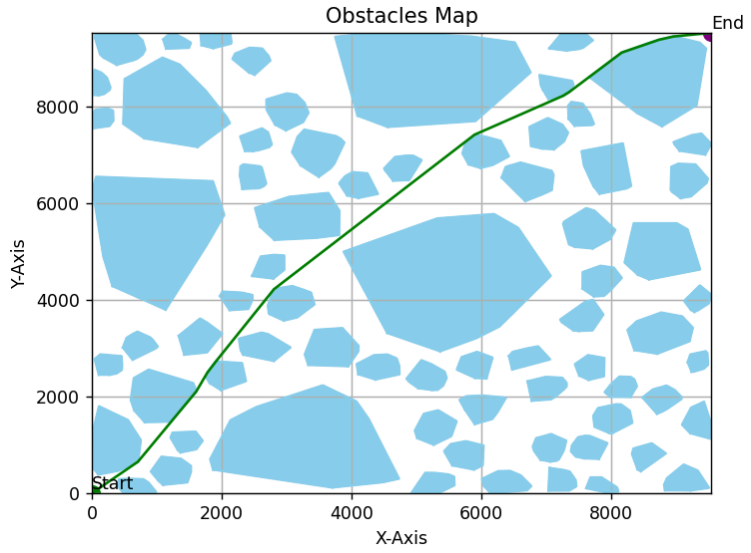
- Example: **Start point visibility**



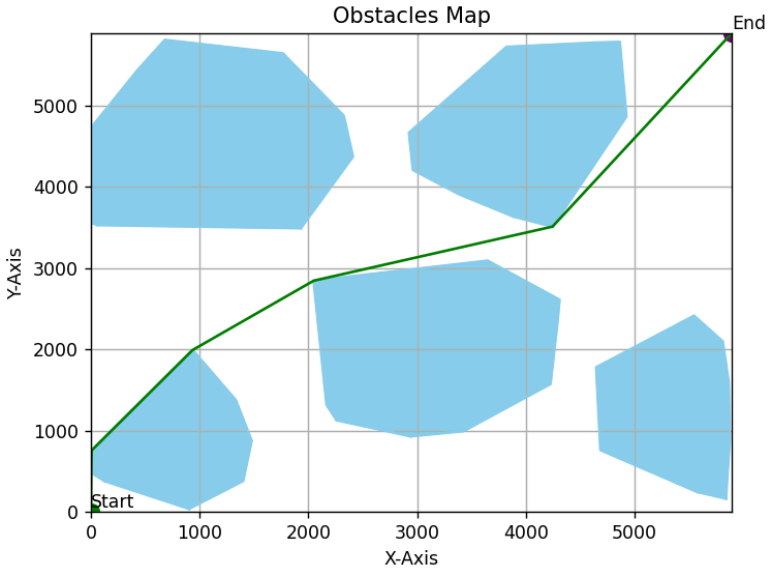
- Pathfinding and Results
    - With the visibility graph complete, we run **pathfinding algorithms** and return the results to Python for visualization.

Example Results

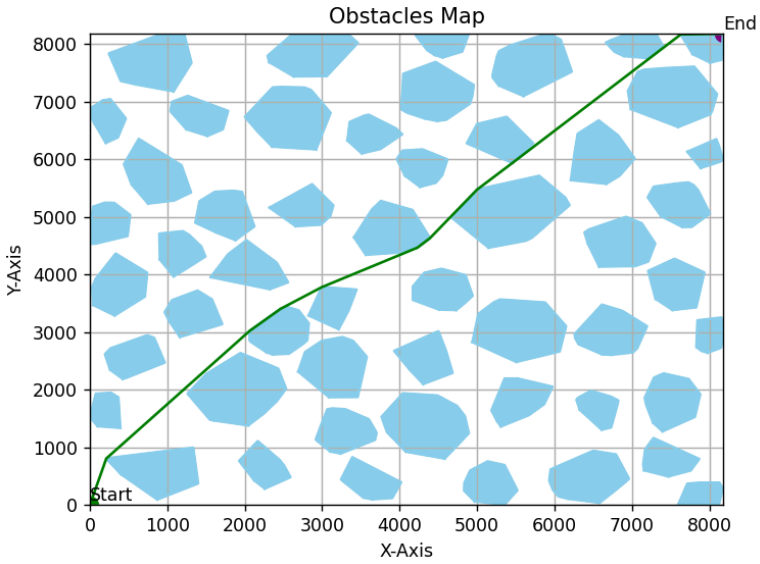
- Different polygon sizes

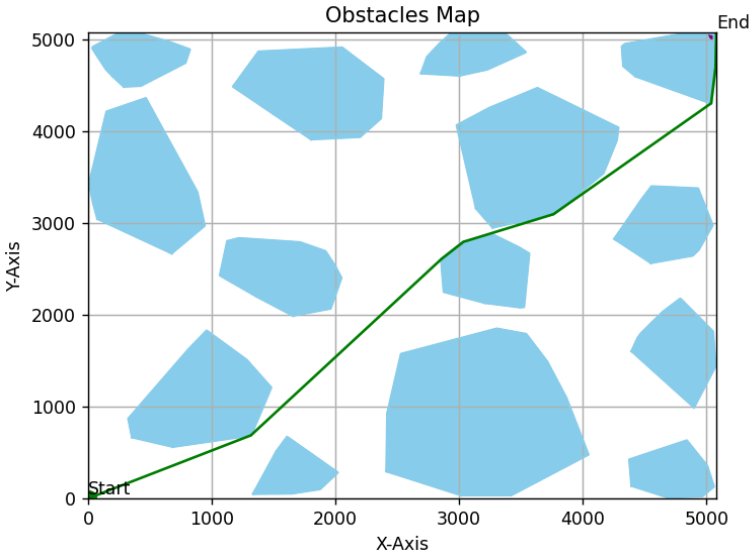


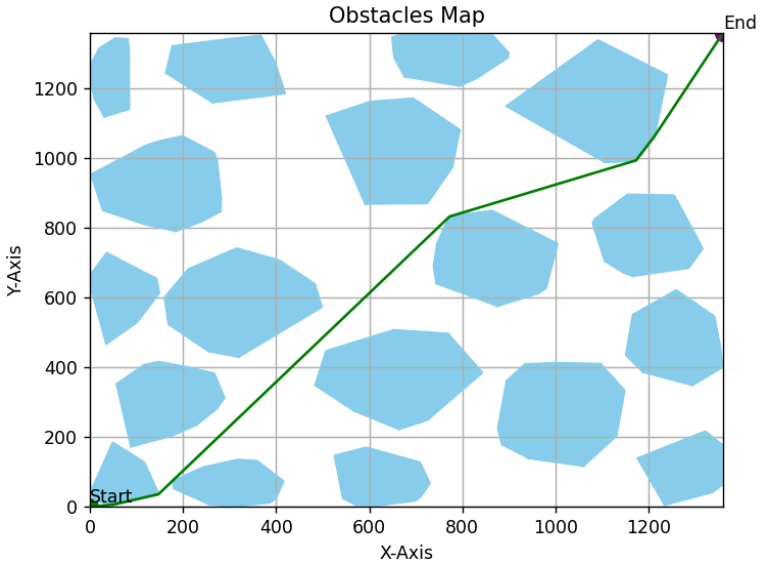
- 5 large polygons



- Additional random map examples



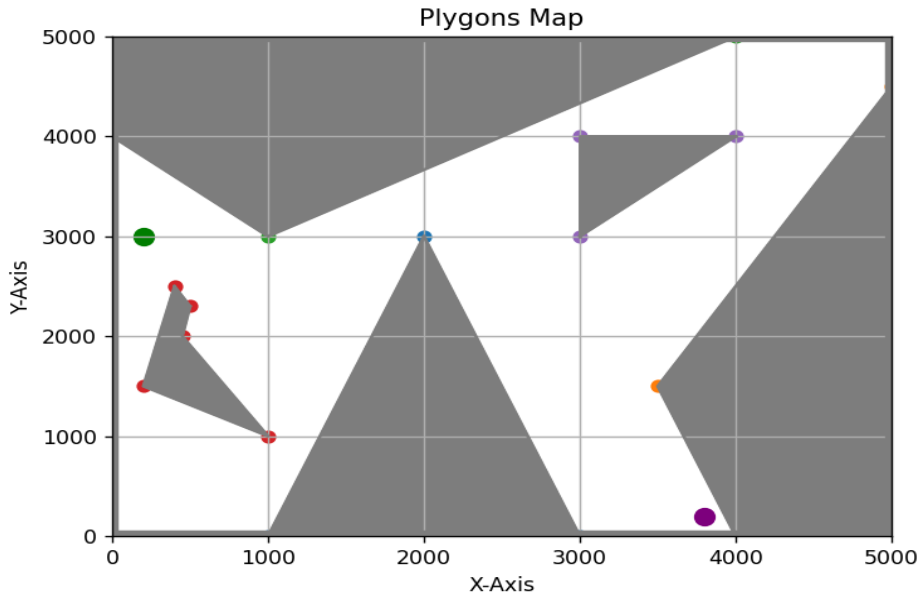




2. Transition to Random Map Input (second part of our project)

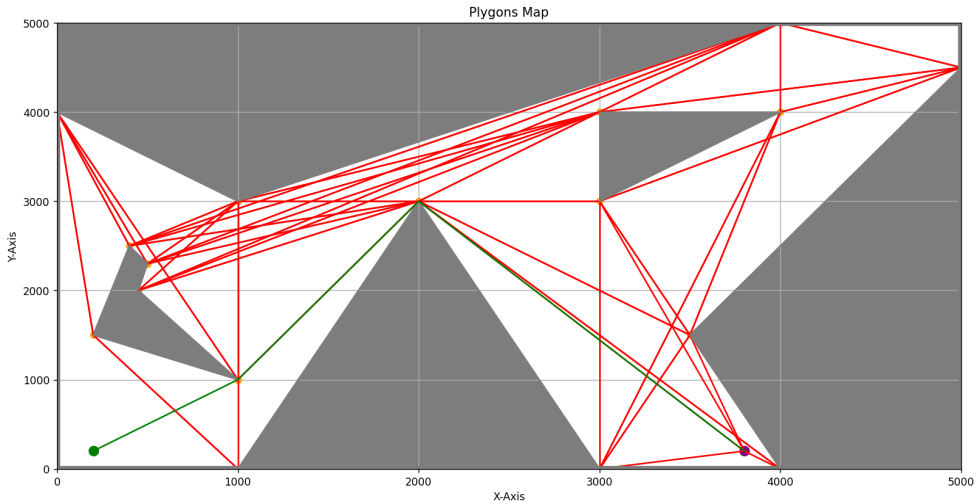
In the second part, we introduced a transitional phase, where we built a randomly generated map and established our pathfinding algorithm.

- Generating a Randomized Map
  - ➢ Generate a **random number of polygons** to serve as obstacles.
  - ➢ Define a **map size**, a **starting point**, and an **endpoint**.
  - ➢ Randomly generate **polygon vertices** to create non-convex obstacles.

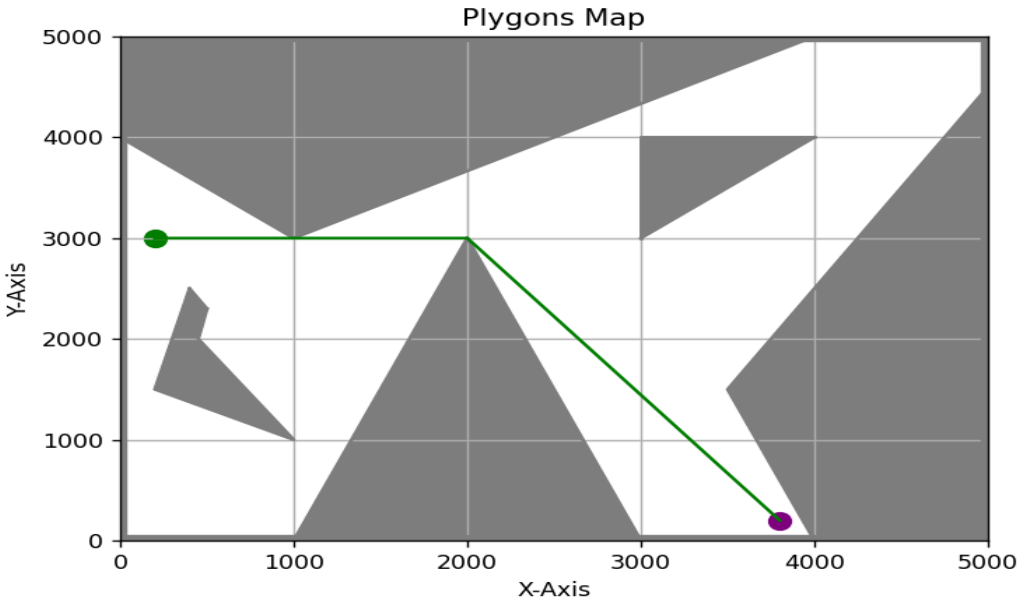


- Triangulation
  - Apply triangulation to the randomly generated map.

- Visibility Graph Construction
  - Each node in the map determines its visible neighbors.
  - A visibility graph is created, representing all possible direct connections.



- Pathfinding and Output

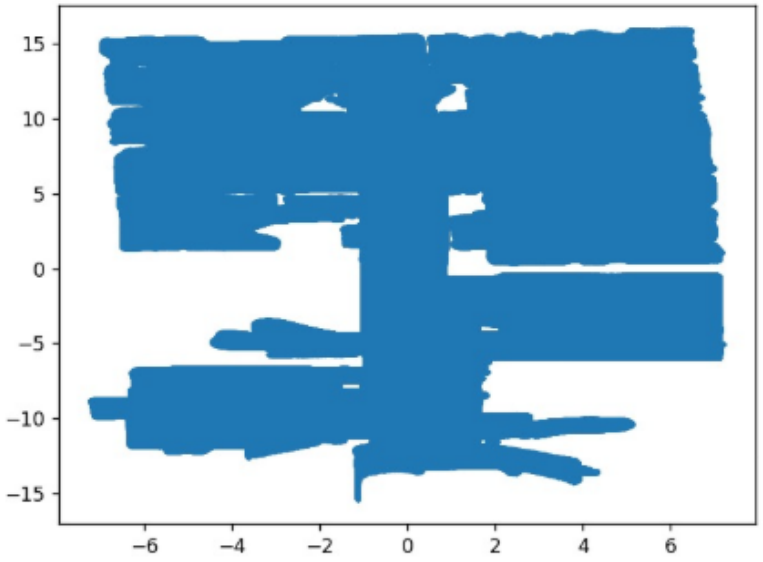


3. **Point Cloud Input (Real-World Data - Amir Building Lab Floor)**:

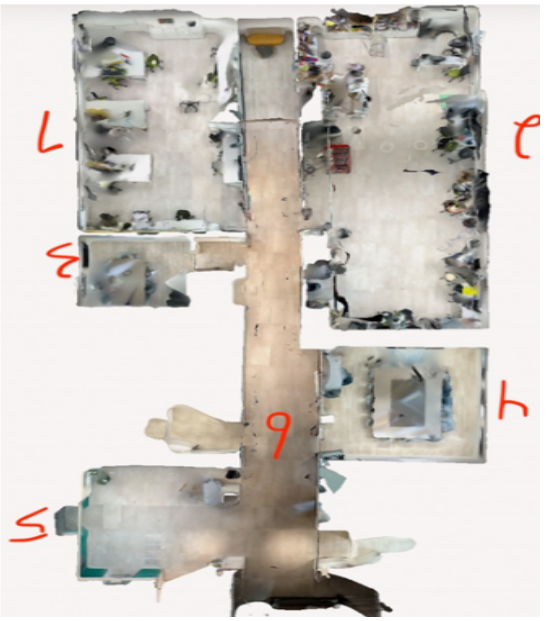
   In the final part, we demonstrated our algorithm's capabilities in a **real-world environment** by applying it to a **point cloud representation of a room**.

   - <u>Receiving and Processing Input</u>

.xyz file:

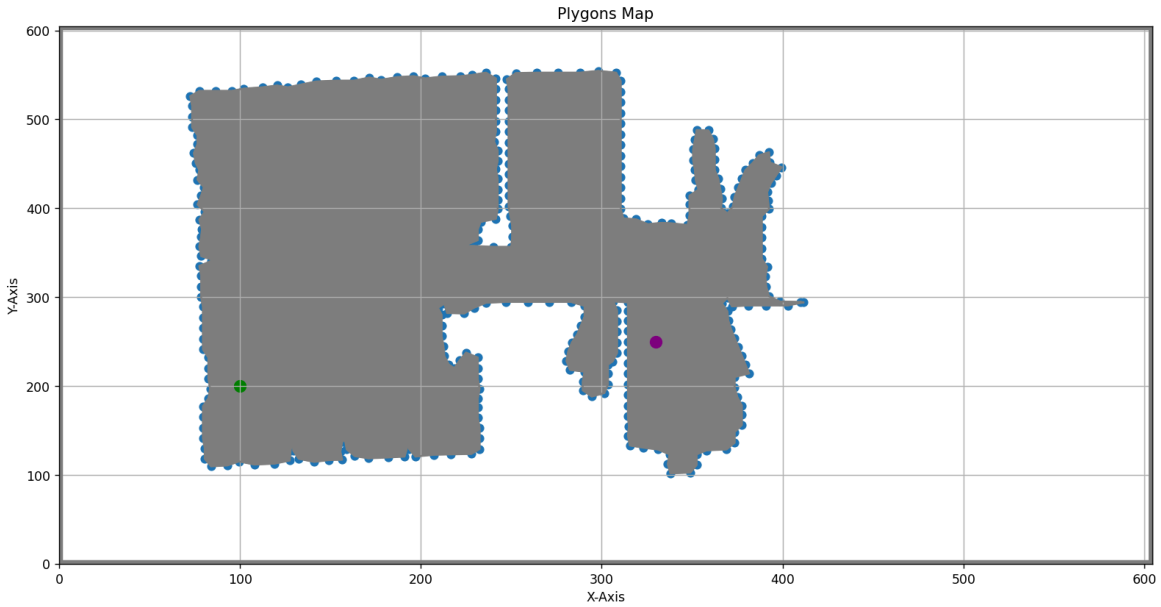


Floor image:

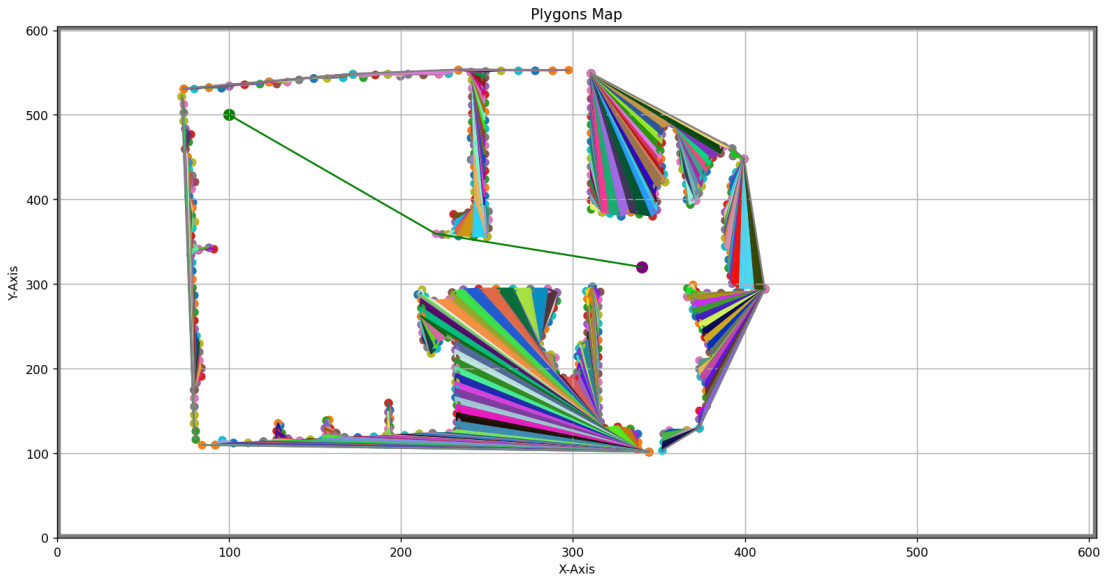


   - <u>Room Representation</u>

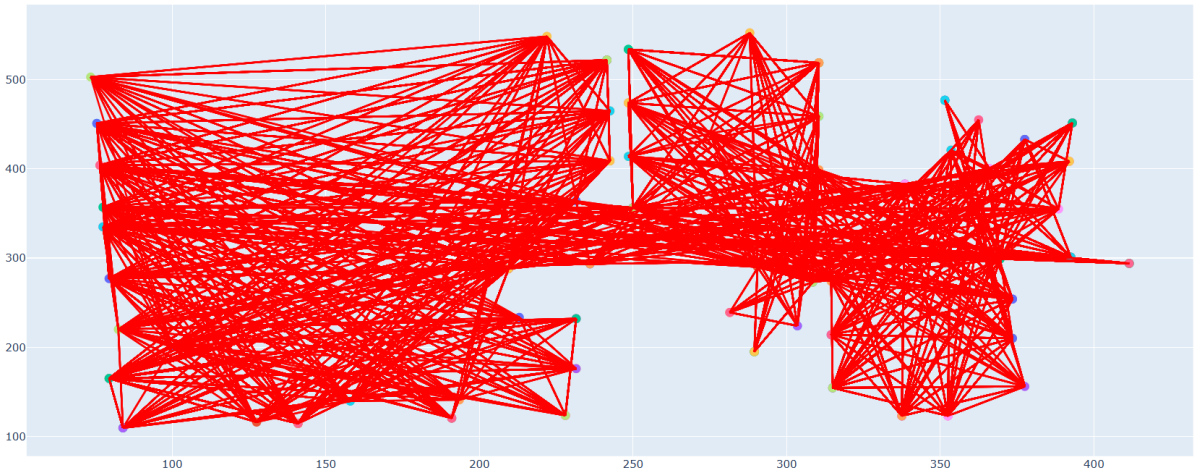
     The **extracted boundaries** are used to create a **polygonal map** representing the actual lab floor.



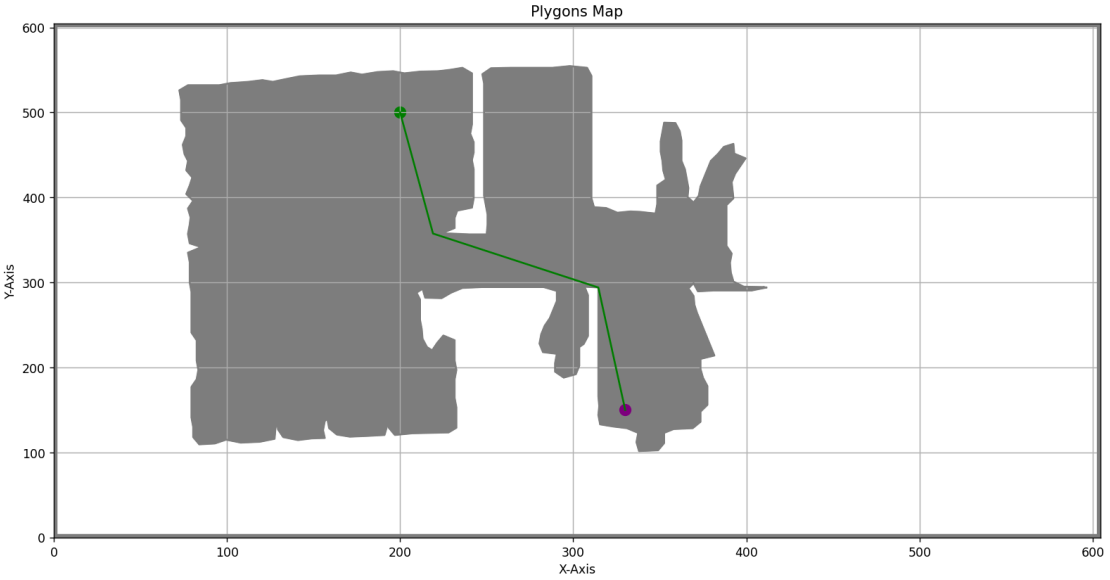
- Triangulation
    - Apply **triangulation** to the **polygonal room** before constructing the **visibility graph**.
    - Triangulation is performed on the **outer part of the polygon**, allowing movement between obstacles while ensuring a structured representation for pathfinding.

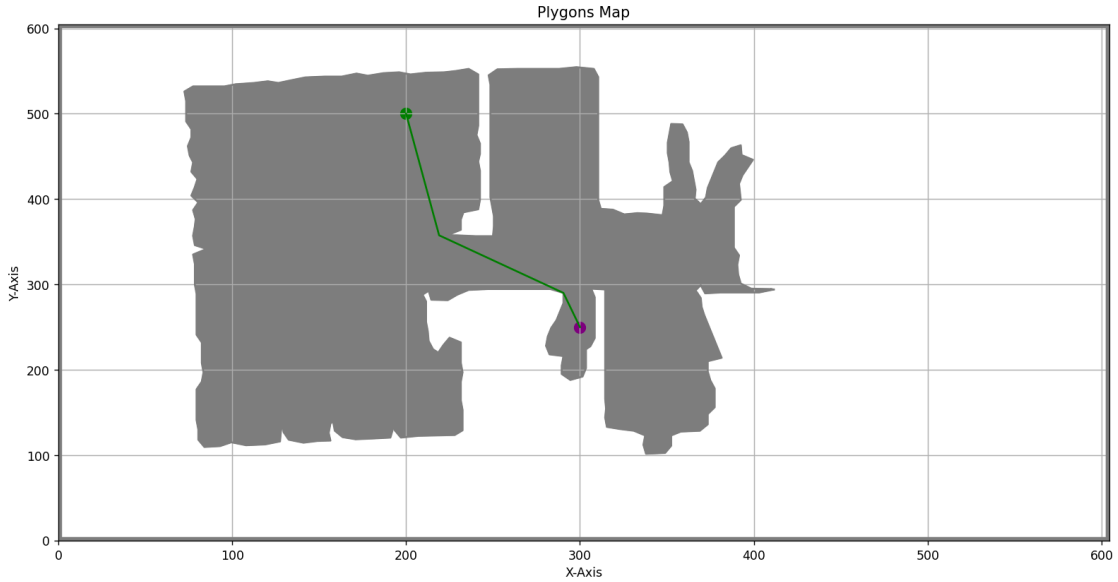


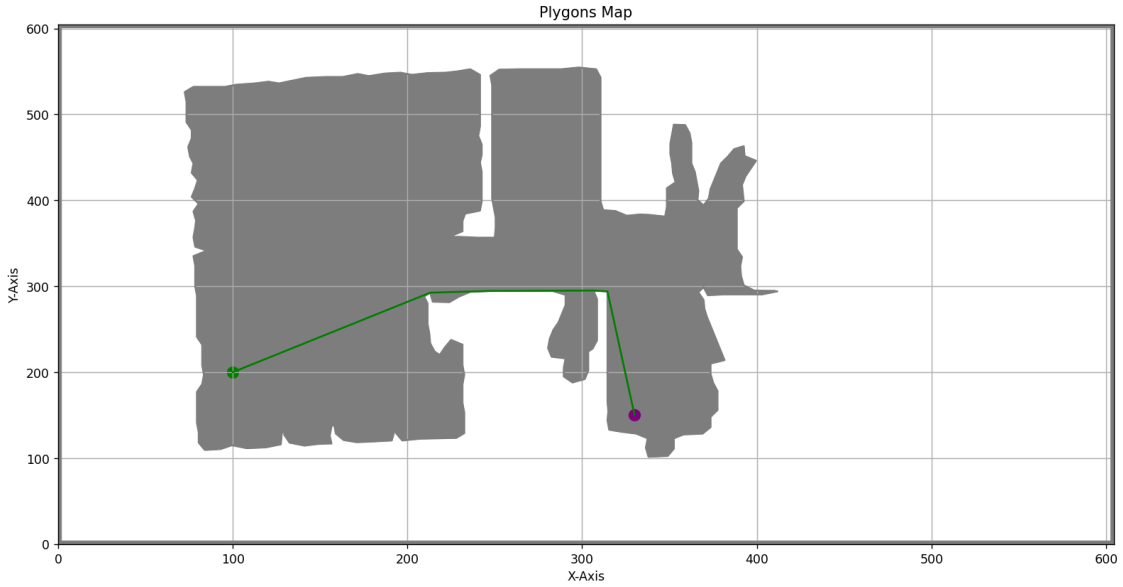
- Visibility Graph Construction
    - Each node in the map determines its **visible neighbors**.
    - A **visibility graph** is created, representing all possible direct connections.



- <u>Pathfinding and Output</u>
    - Python selects the pathfinding algorithm to compute the shortest path:
        - ➢ A* (heuristic-based)
        - ➢ Dijkstra's Algorithm (guarantees shortest path)
        - ➢ RRT (suitable for complex environments)
    - The computed path is returned from C++ to Python for visualization.







How To Run:

1. Build and compile the C++ project in Release mode
2. Paste the .xyz file to work/floor_1_amir/point_cloud.xyz
3. Run python main.py script
4. Enjoy :)

References:

- Environment Detection and Path Planning Using the E-puck Robot Muhammad Saleem Sumbal Department of Electrical, Electronics and Automation Engineering University of Girona, Spain, https://www.researchgate.net/publication/314299113_Environment_Detection_and_Path_Planning_Using_the_E-puck_Robot
- Algorithm HyperLink: https://www.science.smith.edu/~istreinu/Teaching/Courses/274/Spring98/Projects/Philip/fp/algVisibility.htm