- Build a polygonal map ➜ construct a visibility graph (VG) ➜ run A* (or Dijkstra) on that graph.
- Just remember: raw VG paths tend to skim obstacle corners, so you must "grow" obstacles by at least the robot radius (configuration-space inflation) before building the graph. Slides 04_05 - Navigation

Here's how I'd structure it end-to-end, with a data format that will make your life easy.

1) Data you give the planner

Instead of screenshots, export coordinates from GeoGebra. Two simple options:

CSV (recommended)

bash — Code kopieren

```
type,id,label,x,y
vertex,poly1,A,0.0,0.0
vertex,poly1,B,2.0,0.0
vertex,poly1,C,2.0,1.0
vertex,poly1,D,0.0,1.0
# ... one row per vertex, polygons must be ordered CCW and closed implicitly
start,,S, -1.0, 0.5
goal,,G,  6.0, 0.5
```

type, id, label, x, y

Ex: vertex, poly 1, A, 50.0, 50.0

vertex, poly 2, F, 40.0, 40.0

start, start_pt, Start, 5.0, 95.0
goal, goal_pt, Goal, 95.0, 5.0

JSON

json — Code kopieren

```
{
  "polygons": [
    {"id": "poly1", "label": "box", "vertices": [[0,0],[2,0],[2,1],[0,1]]}
  ],
  "start": [-1.0, 0.5],
  "goal":  [ 6.0, 0.5],
  "robot_radius": 0.09
}
```

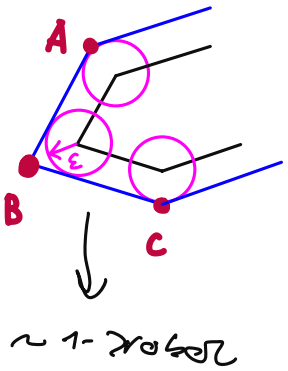
Having an explicit `polygon_id` (or label) per vertex is perfect: it lets us reconstruct polygons unambiguously exactly as you suggested. Include `robot_radius` (or wheelbase half-width if you prefer) so we can inflate obstacles.

2) Pipeline (why + how)

1. **Inflate obstacles** by the robot radius ("grow" them) so that a point-mass path is safe for your real robot. Slides 04_05 - Navigation
2. **Collect candidate nodes** = all (inflated) polygon vertices + start + goal.
3. **Visibility test** between every pair of nodes: a straight segment is valid iff it lies in free space and doesn't intersect any inflated obstacle edge (touching at a common vertex is OK if you allow grazing, but safer to require a tiny clearance ε).
4. **Build graph**: weighted undirected edges with Euclidean distance.
5. **A*** with heuristic = straight-line distance. (Complete and optimal on this graph; A* expands fewest nodes with an admissible heuristic.) Slides 04_05 - Navigation
6. **Output path** as a polyline. (Later, hand the polyline to your path follower / local nav.) Slides 04_05 - Navigation

This lines up with the course notes (plan globally with A*/Dijkstra, VG requires obstacle growth, then follow the path and pair with local avoidance). Slides 04_05 - Navigation

→ how much add ε



~ 1-norm

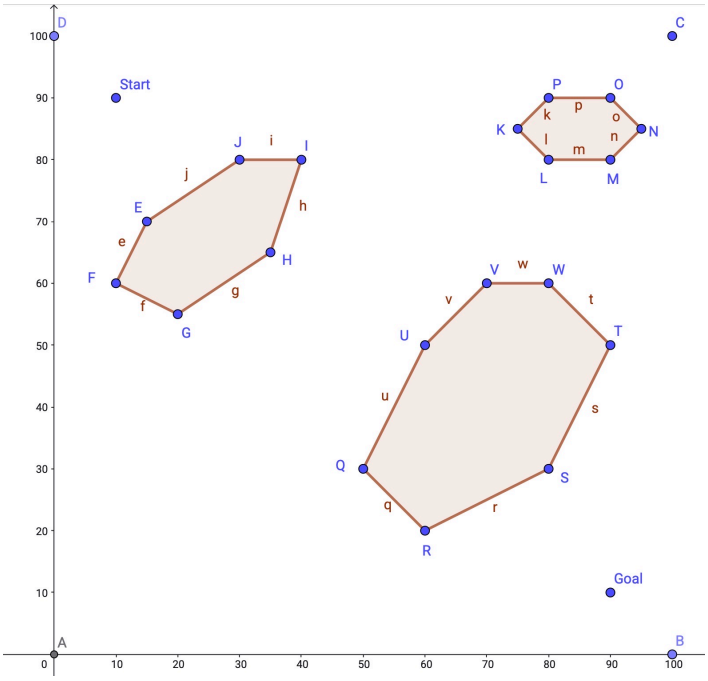
3) Practical gotchas

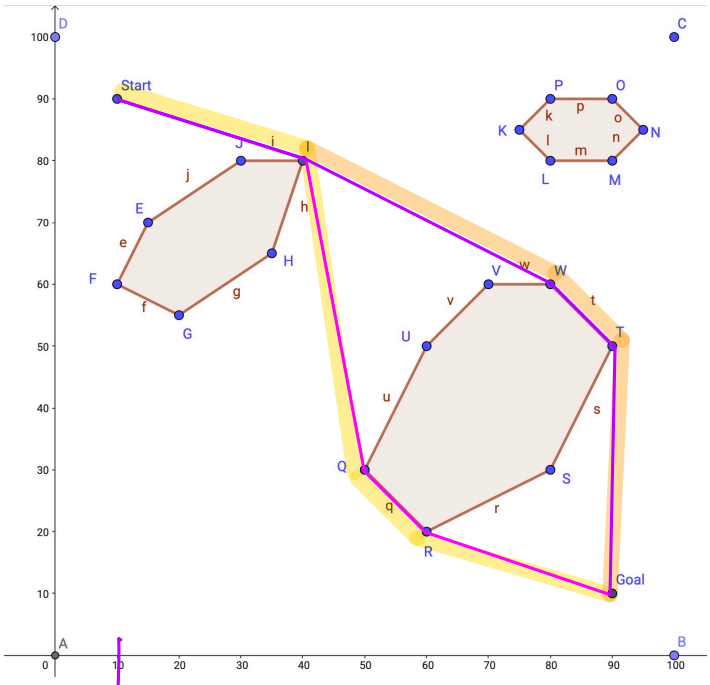
- **Numerical robustness:** treat intersections with a small epsilon; snap nearly collinear points; ensure polygons are CCW and non-self-intersecting. ①
- **Start/goal inside free space:** verify they're not inside any inflated polygon. ②
- **Performance:** naive O(N²) visibility checks are fine for coursework maps (tens–hundreds of vertices).
- **Thymio reality check:** VG gives shortest paths, but often too close to walls—keep the inflation generous to reduce controller stress. Slides 04_05 - Navigation

→ put hypothesis

→ would exist better than $O(N^2)$?

Report:
- hypothesis { CCW, non-self-intresting; free space
- $O(x)$
- how determine ε





ptbm

Python:

have coordinates → draw polygone based edge coordinates ?

list start nodes goal position angle

[ ]