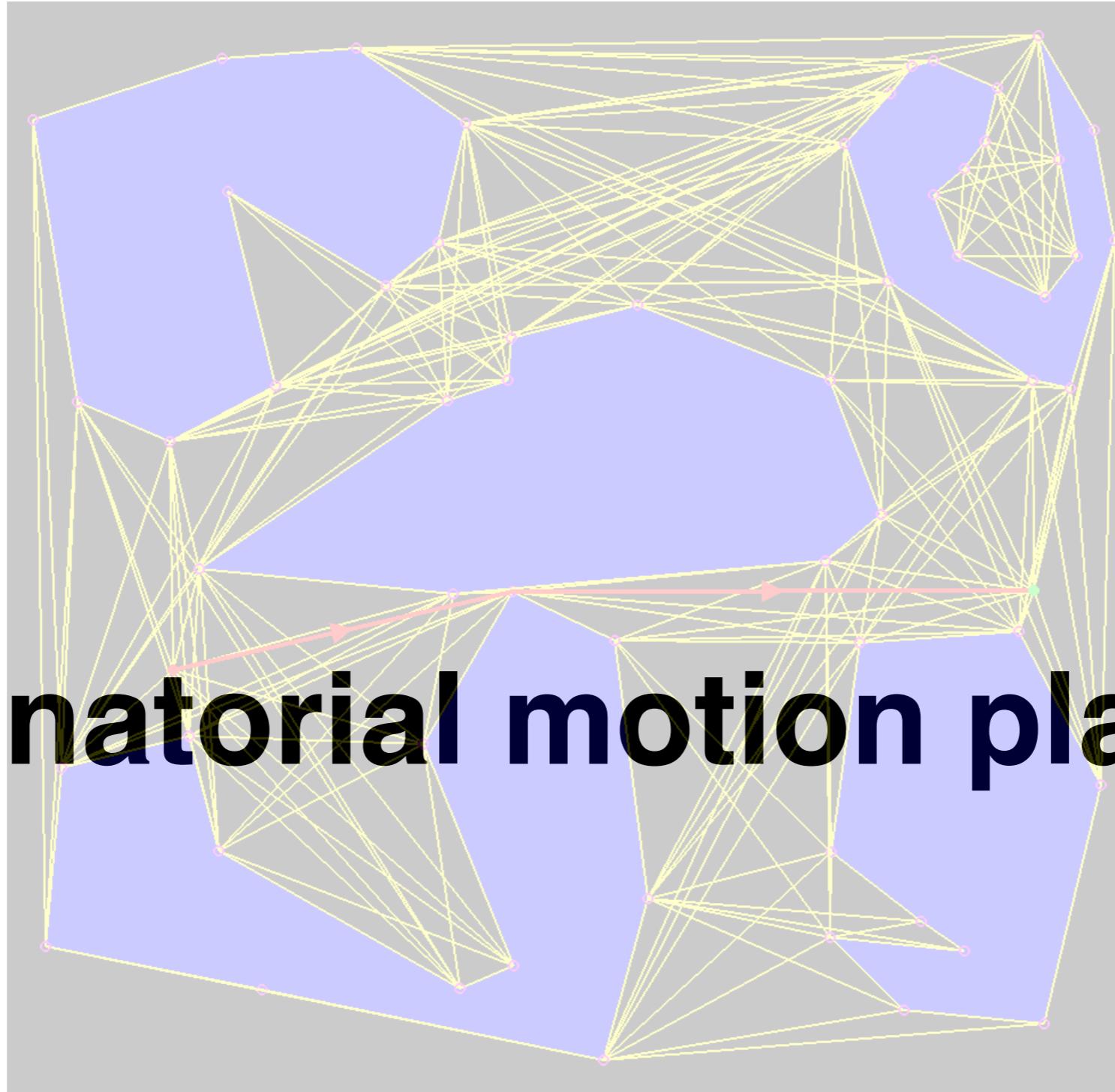
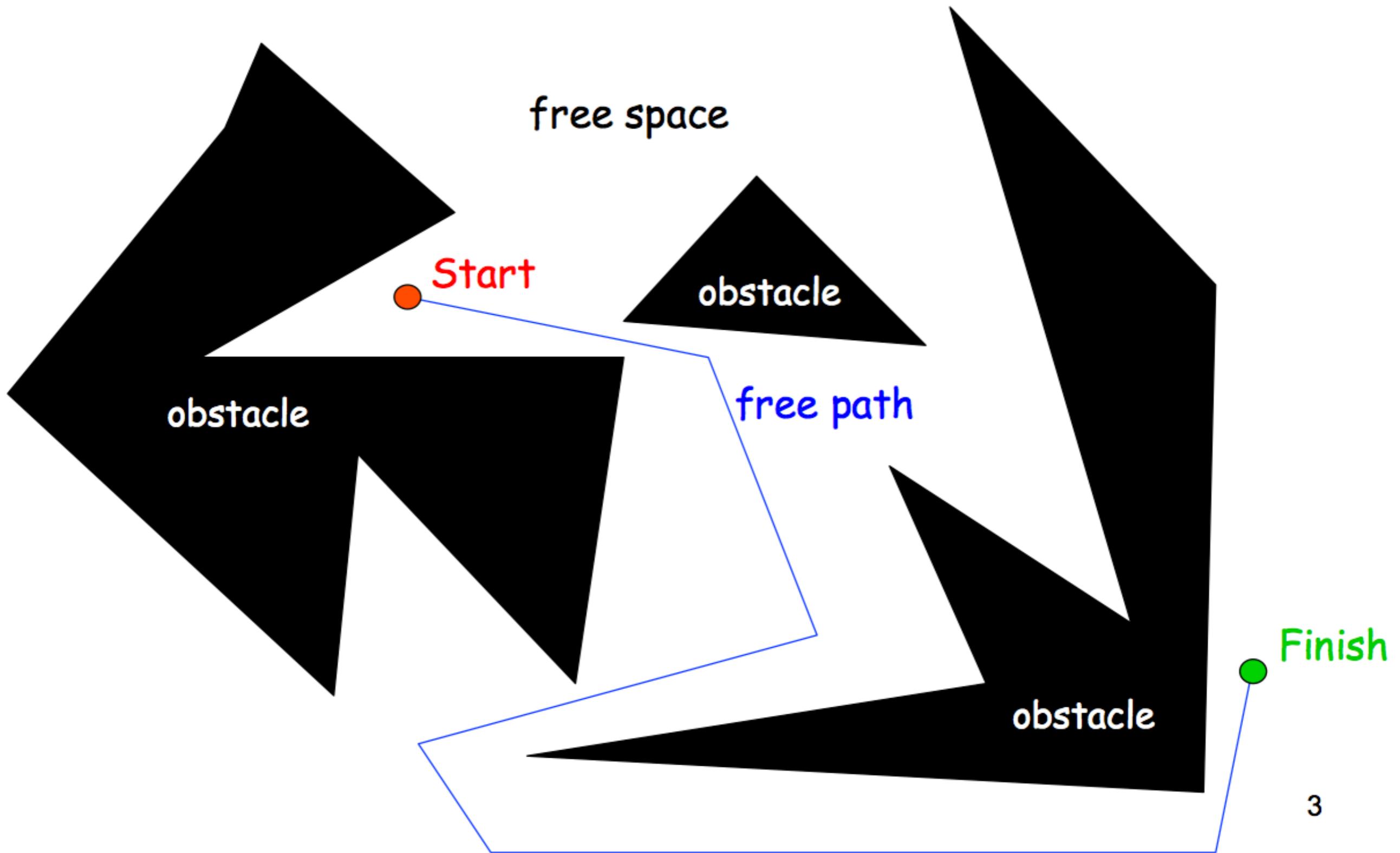


# Combinatorial motion planning



Computational Geometry  
csci3250  
Laura Toma  
Bowdoin College

# Problem



# Motion Planning

## Input:

- a robot  $R$
- start and end position
- a set of obstacles  $S = \{O_1, O_2, \dots\}$

Find a path from start to end (that optimizes some objective function).

## Parameters:

- geometry of obstacles (polygons, disks, convex, non-convex, etc)
- geometry of robot (point, polygon, disc)
- robot movement —how many degrees of freedom (dof); 2d, 3d
- objective function to minimize (euclidian distance, nb turns, etc)
- static vs dynamic environment
- exact vs approximate path planning
- known vs unknown map

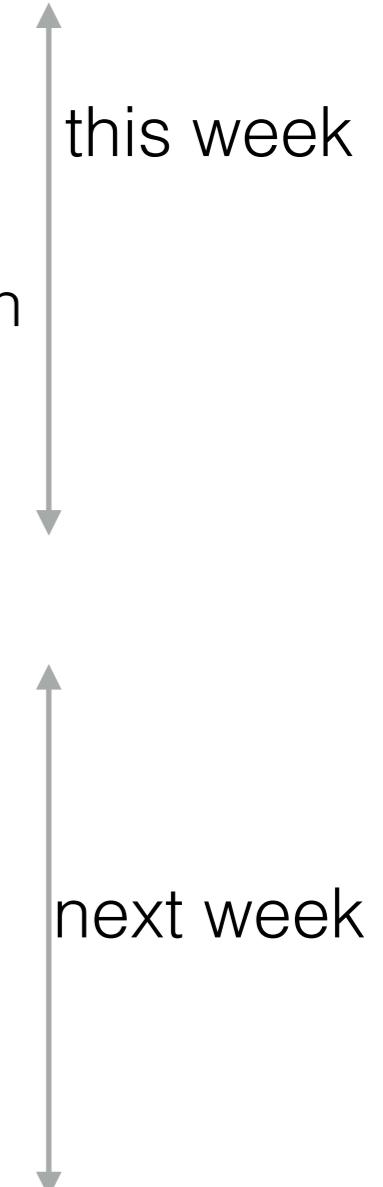
# Motion Planning

algorithm that finds a path

Ideally we want a planner to be complete and optimal.

- A planner is complete:
  - it always finds a path when a path exists
- A planner is optimal:
  - it finds an optimal path (wrt objective function)

# Approaches

- Combinatorial (exact)
    - Used for 2D path planning
    - Idea:
      - Compute an exact representation of free space and a graph that represents the free space
      - Find a path using the graph
  - Approximate
    - sampling-based, search based, space decomposition, probabilistic, potential functions
- 

## 2D path planning problems

- point robot moving inside an arbitrary polygon
- point robot moving among (arbitrary) polygons
- disk robot moving among (arbitrary) polygons
- polygonal robot moving among (arbitrary) polygons
  - translation, translation+rotation
- robot with arms and articulation

harder



# Overview: this week

Combinatorial motion planning in 2D

- Point robot moving among polygonal obstacles in 2D
  - trapezoidal decomposition
  - visibility graph
- Polygonal robot moving among polygonal obstacles in 2D
  - C-space, C-obstacles, ...
  - translation only
  - translation + rotation

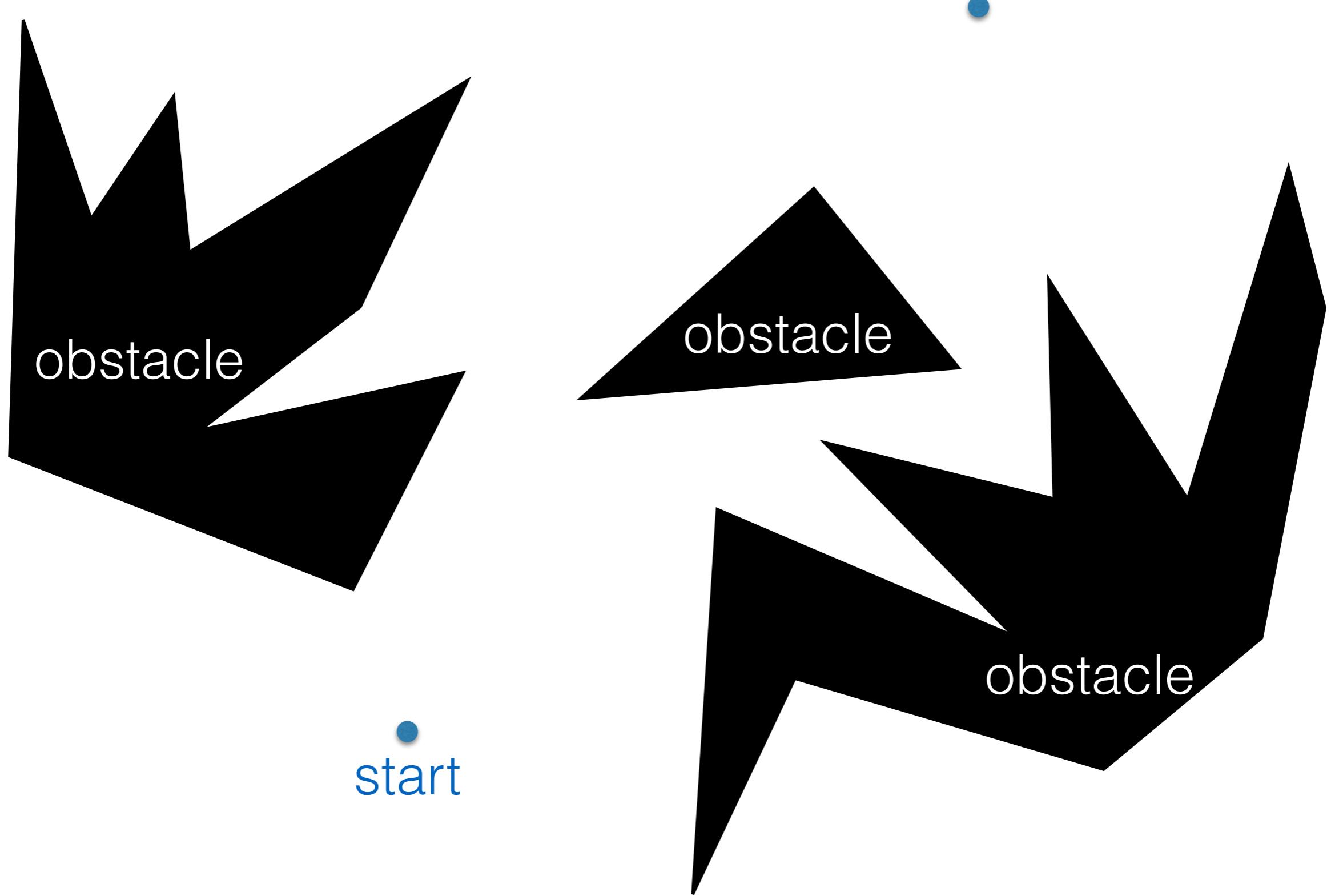
# Point robot moving among polygonal obstacles in 2D

Input:

- start and end position
- a set of polygonal obstacles  $S = \{O_1, O_2, \dots\}$

Find a path from start to end.

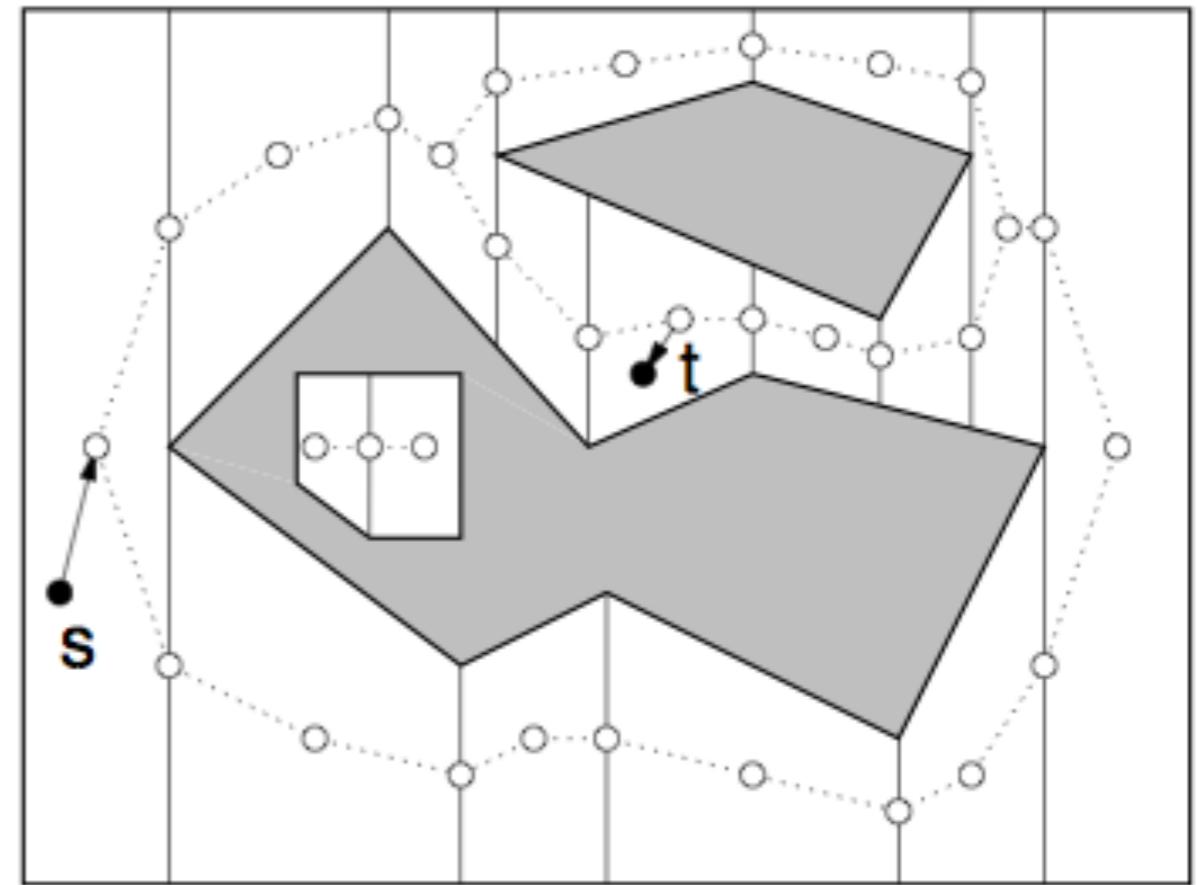
# Point robot in 2D



# Point robot in 2D

## General idea

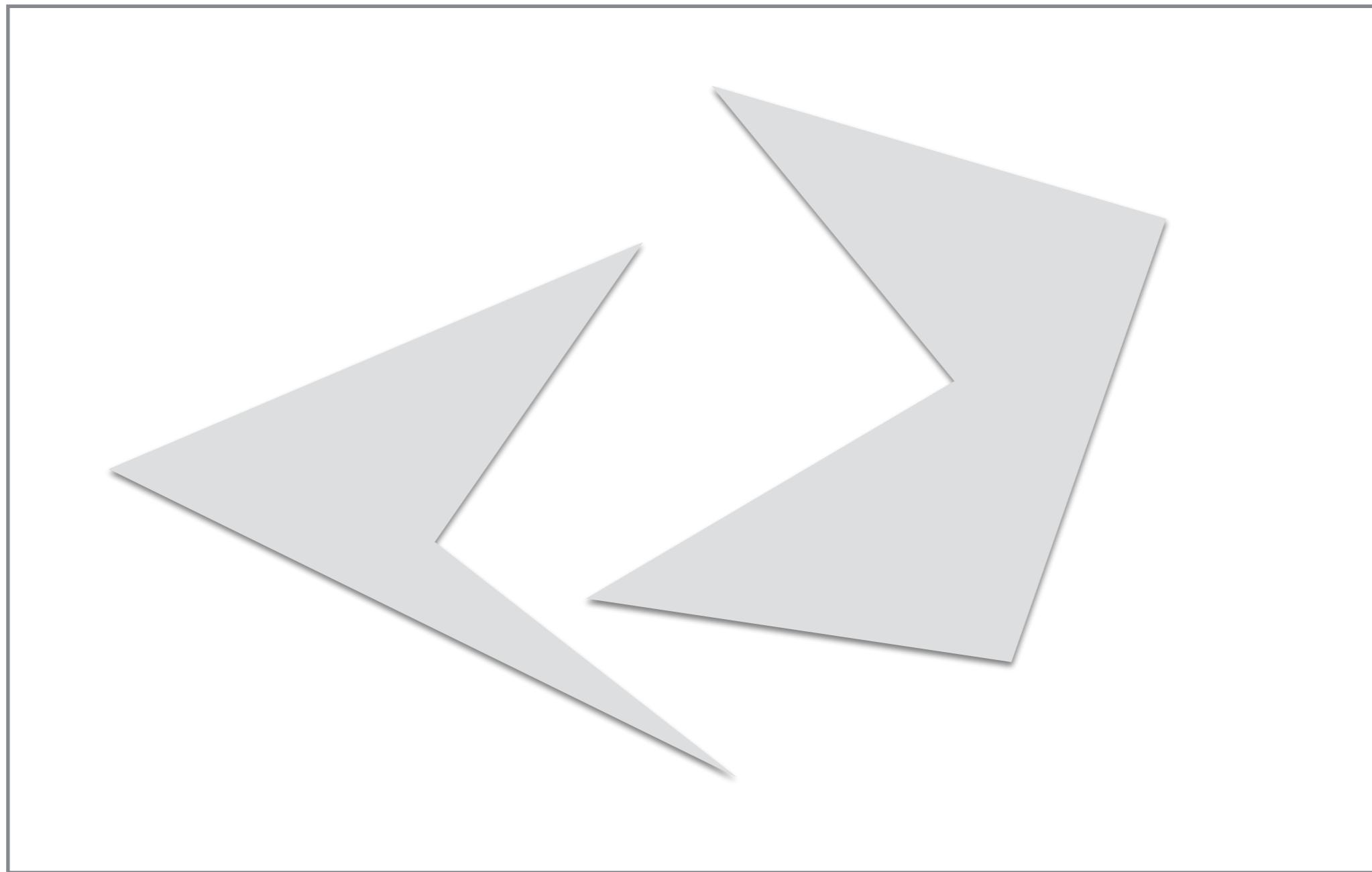
- Compute a representation of free space
- Build a graph of free space
- Search graph to find path



(screenshot from O'Rourke)

- Planning a path for point robot in 2D is reduced to graph search
- Questions: How? How long? Size?

Compute a trapezoid decomposition of free space and the corresponding graph.



# Point robot in 2D

## General idea

- Compute a representation of free space
  - Trapezoid decomposition
- Build a graph of free space
- Search graph to find path
  - BFS



Has size  $O(n)$  and can be computed in  $O(n \lg n)$  time



Has size  $O(n)$  and can be computed in  $O(n)$  time



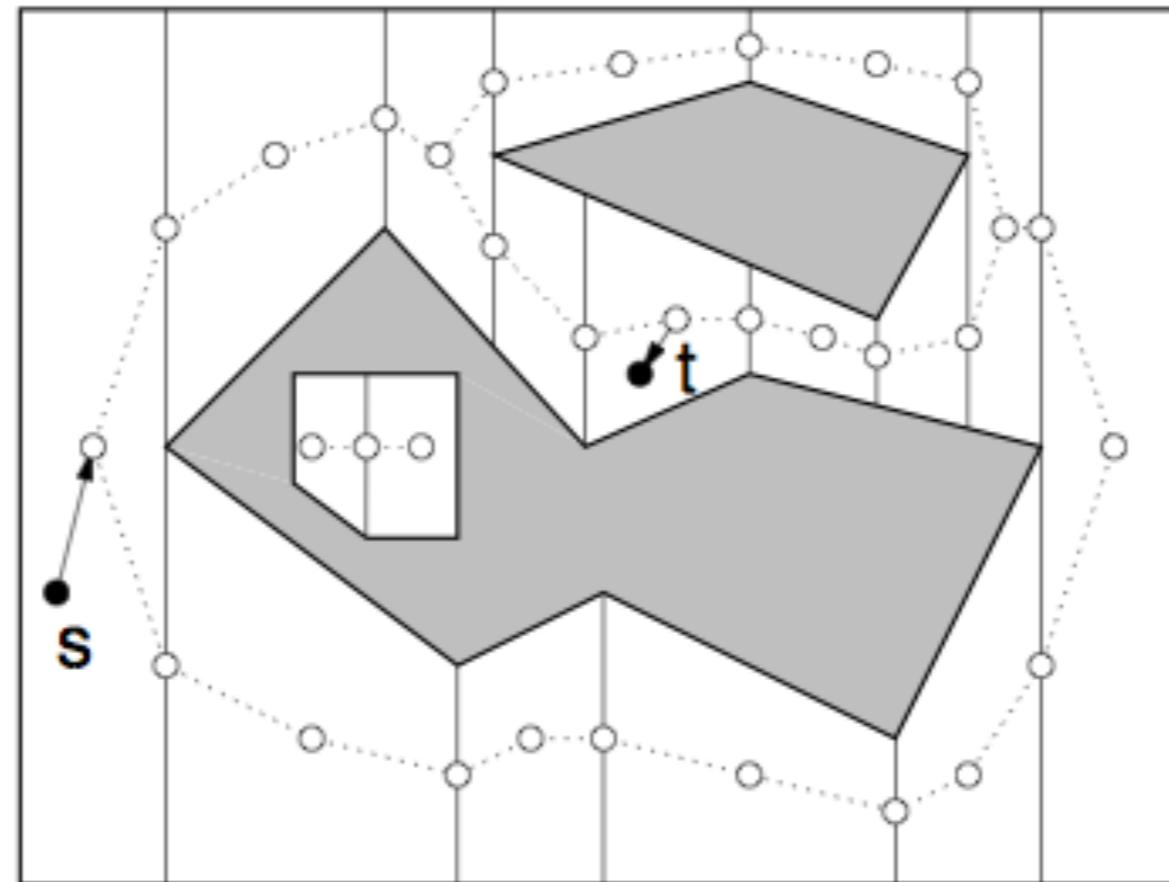
$O(n)$  time

# Point robot in 2D

$n$  = complexity of obstacles  
(total number of edges)

**Result:** Let  $R$  be a point robot moving among a set of polygonal obstacles in 2D with  $n$  edges in total. We can pre-process  $S$  in  $O(n \lg n)$  expected time such that, between any start and goal position, a collision-free path for  $R$  can be computed in  $O(n)$  time, if it exists.

Is this complete?



Is this optimal?

# Point robot in 2D

$n$  = complexity of obstacles  
(total number of edges)

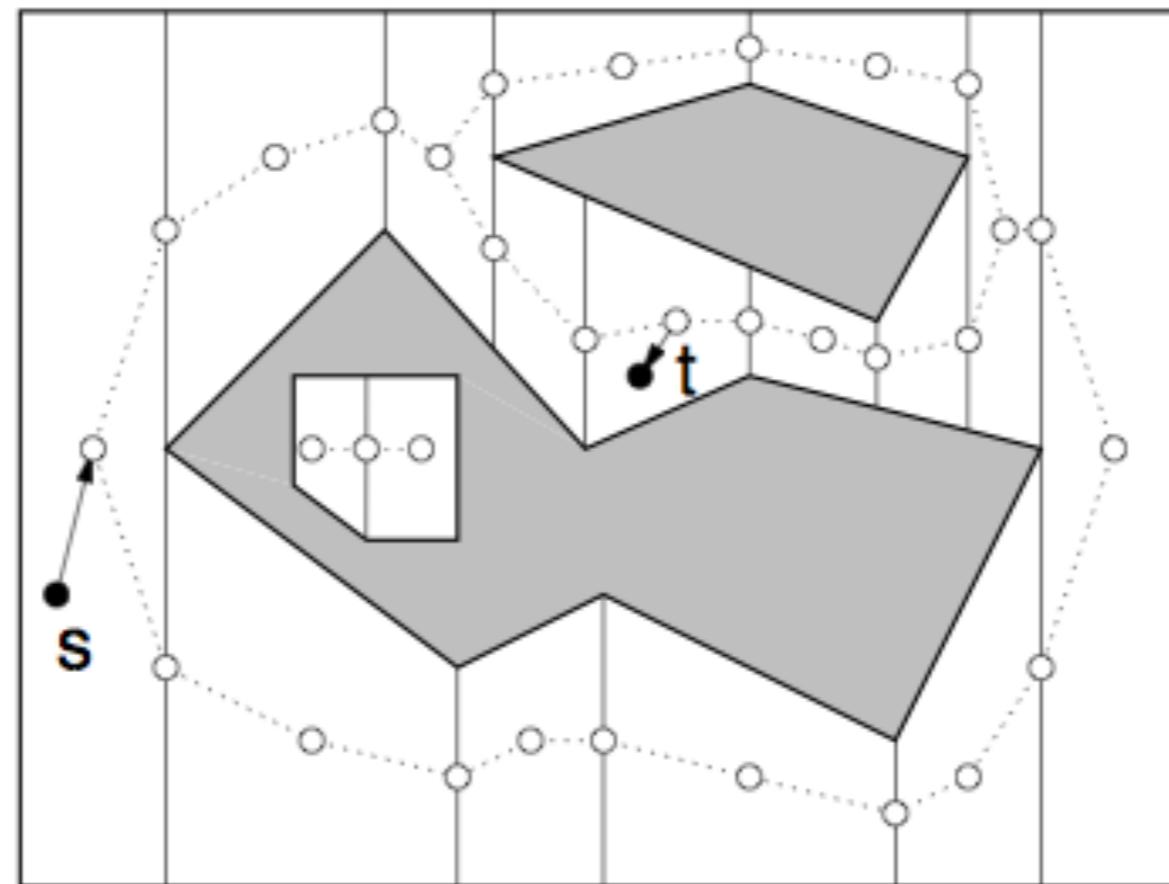
**Result:** Let  $R$  be a point robot moving among a set of polygonal obstacles in 2D with  $n$  edges in total. We can pre-process  $S$  in  $O(n \lg n)$  expected time such that, between any start and goal position, a collision-free path for  $R$  can be computed in  $O(n)$  time, if it exists.

Is this complete?

YES

Is this optimal?

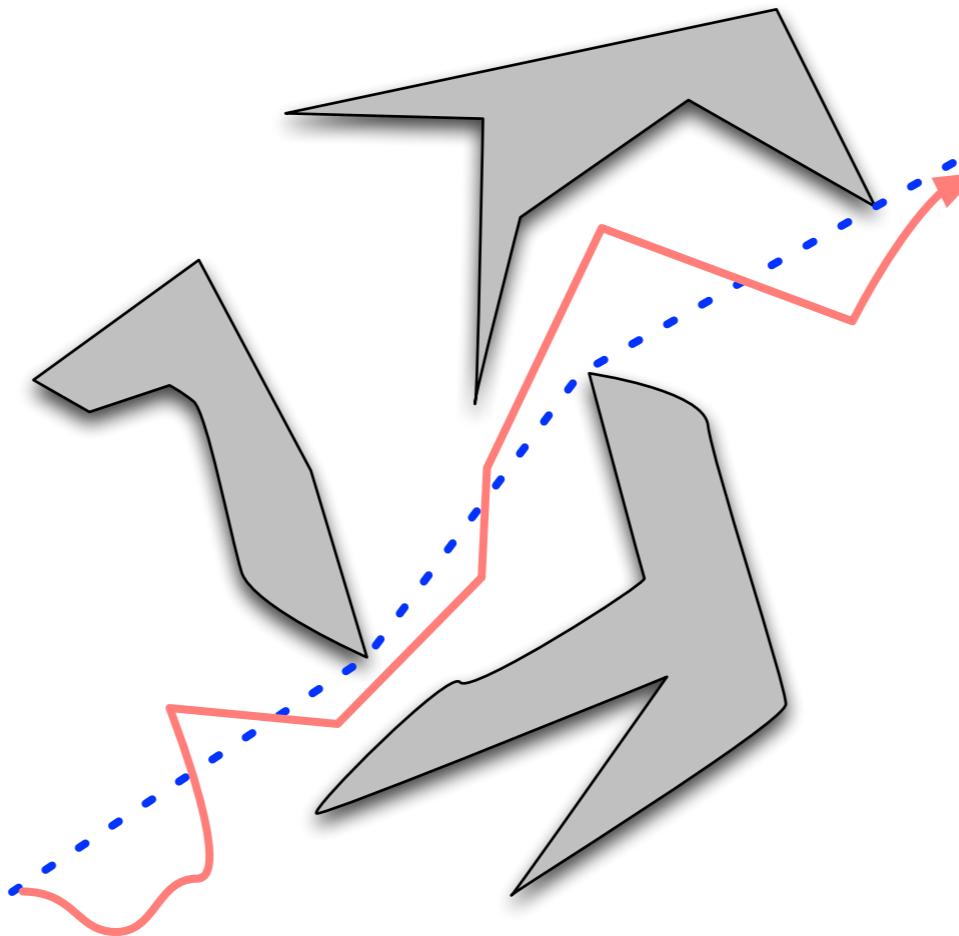
No



## Classwork

Show that the trapezoid map is not optimal by giving a scene where it does not give the optimal (shortest) path

## Point robot in 2D

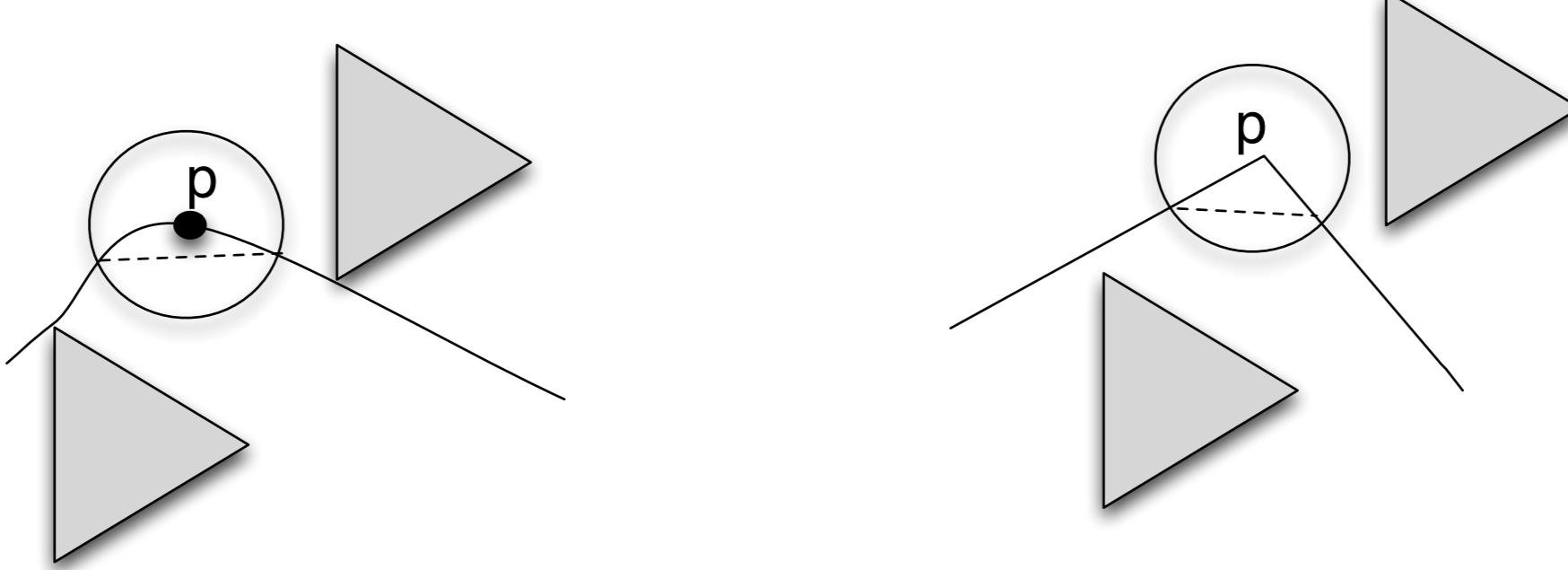


What if we wanted the shortest path?

# Point robot in 2D with shortest paths

Theorem: Any shortest path among a set  $S$  of disjoint polygonal obstacles

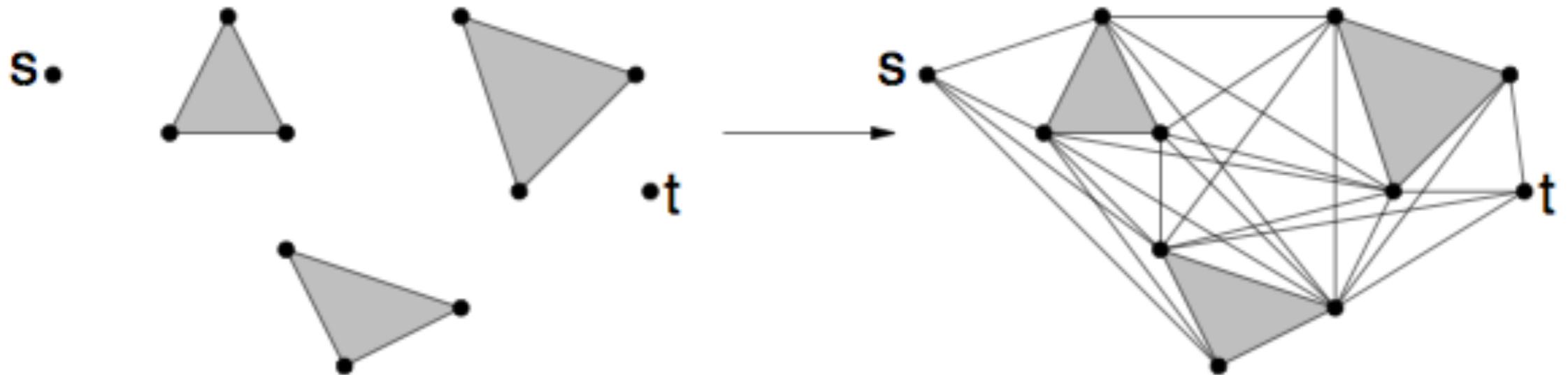
1. is a polygonal path (that is, not curved)
2. its vertices are the vertices of  $S$ .



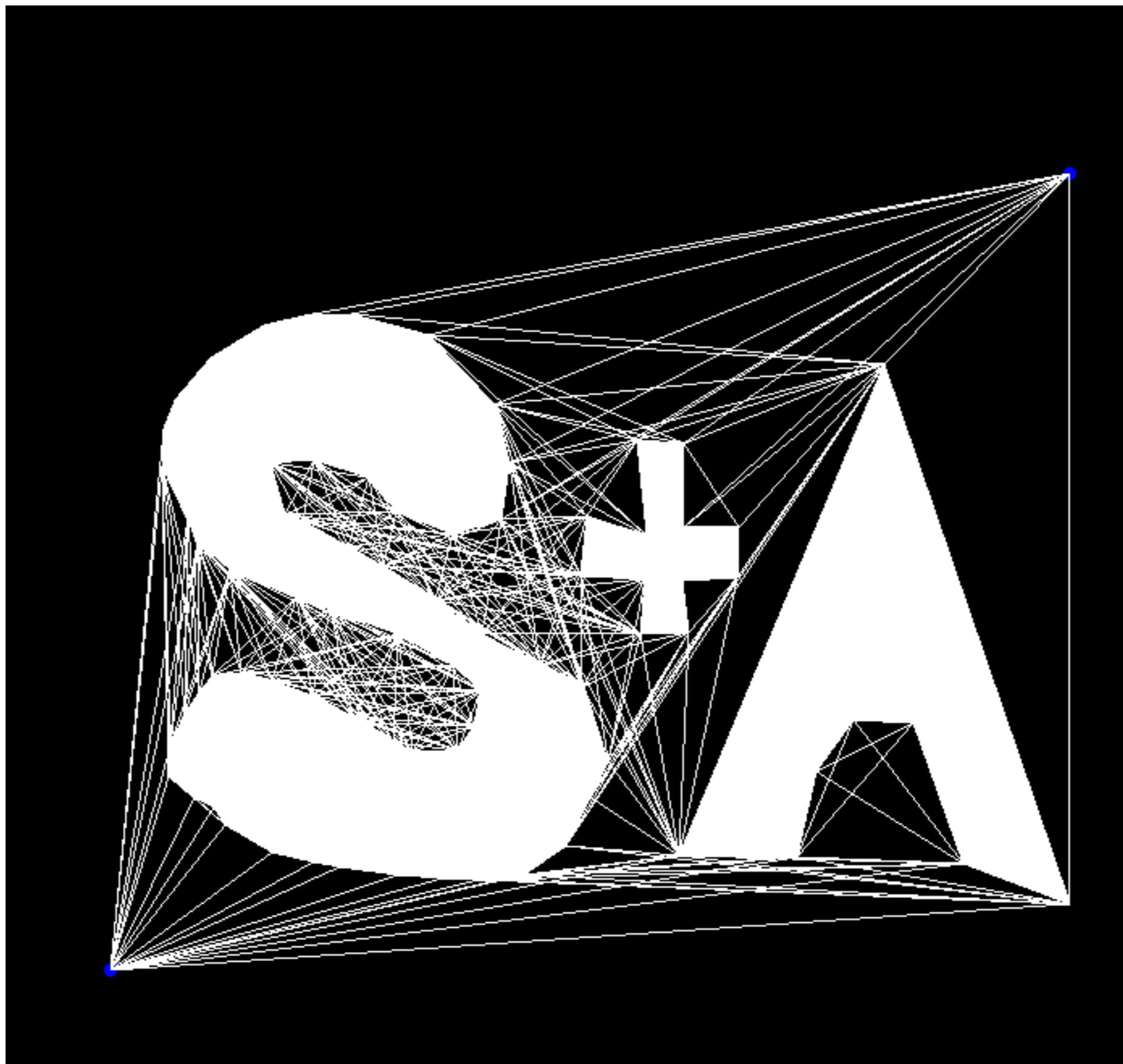
How to find a shortest path from start to end?

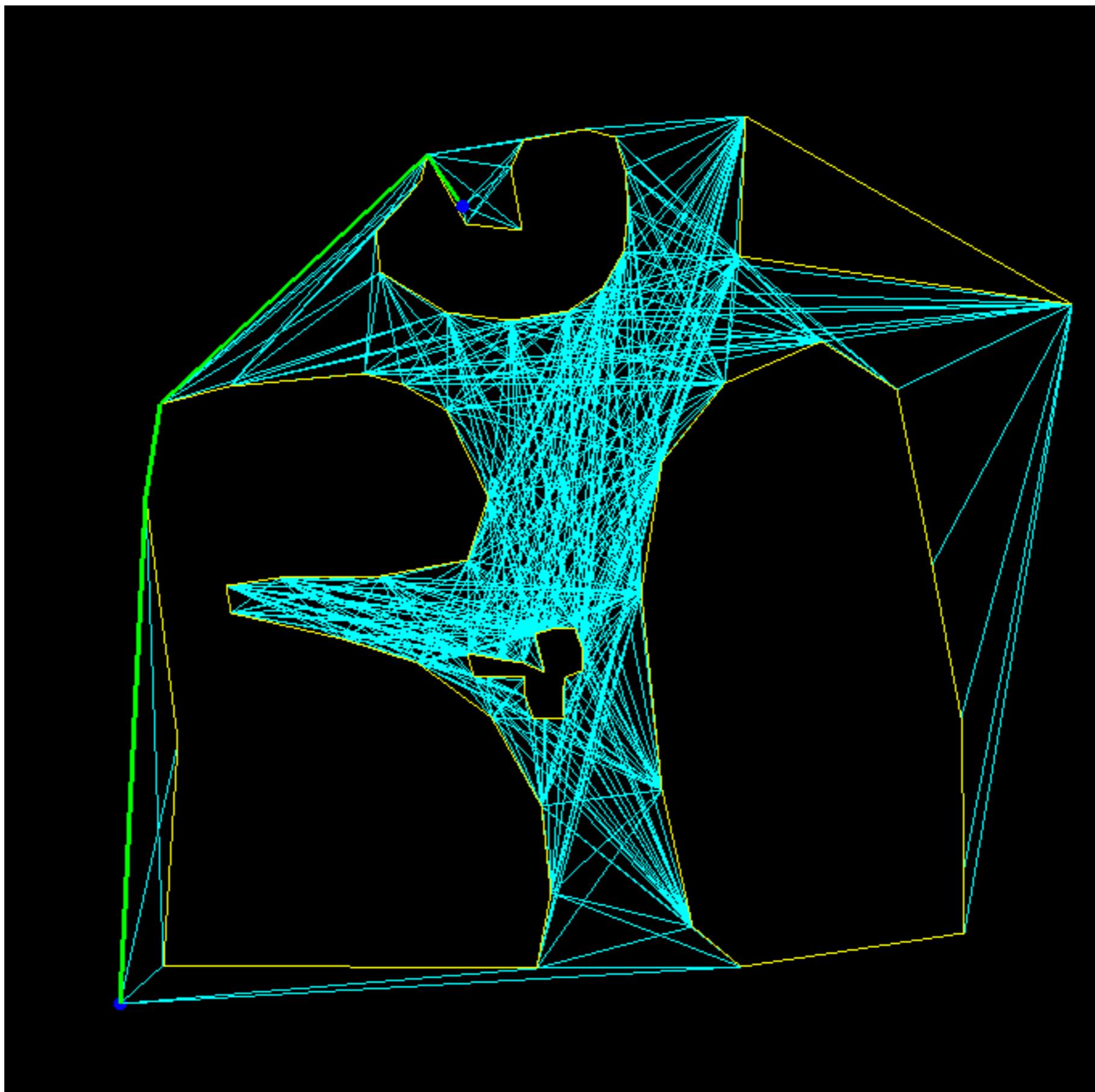
# Point robot in 2D with shortest paths

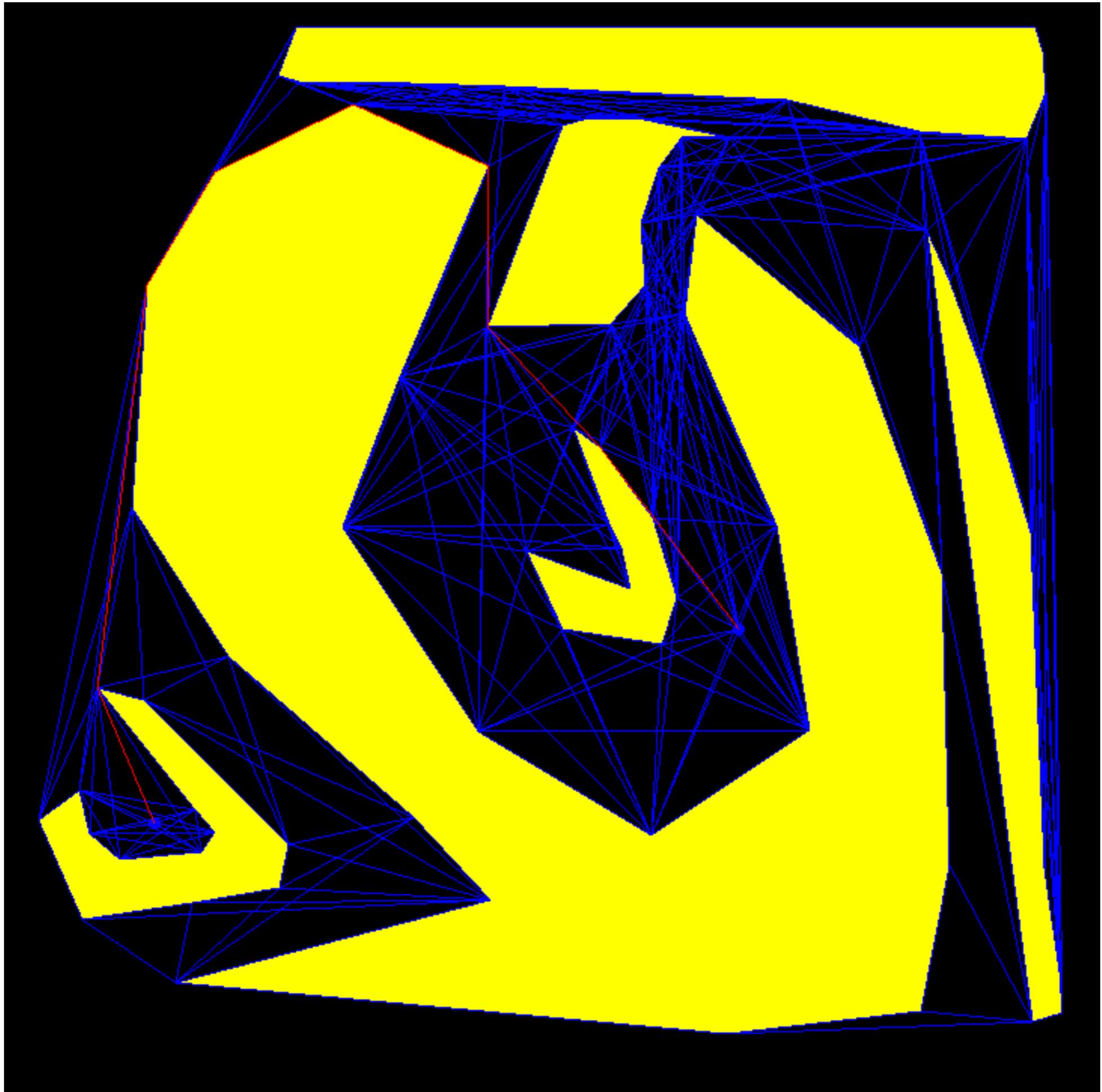
- Idea: Build the visibility graph (VG)
  - represents all possible ways to travel between the vertices of the obstacles

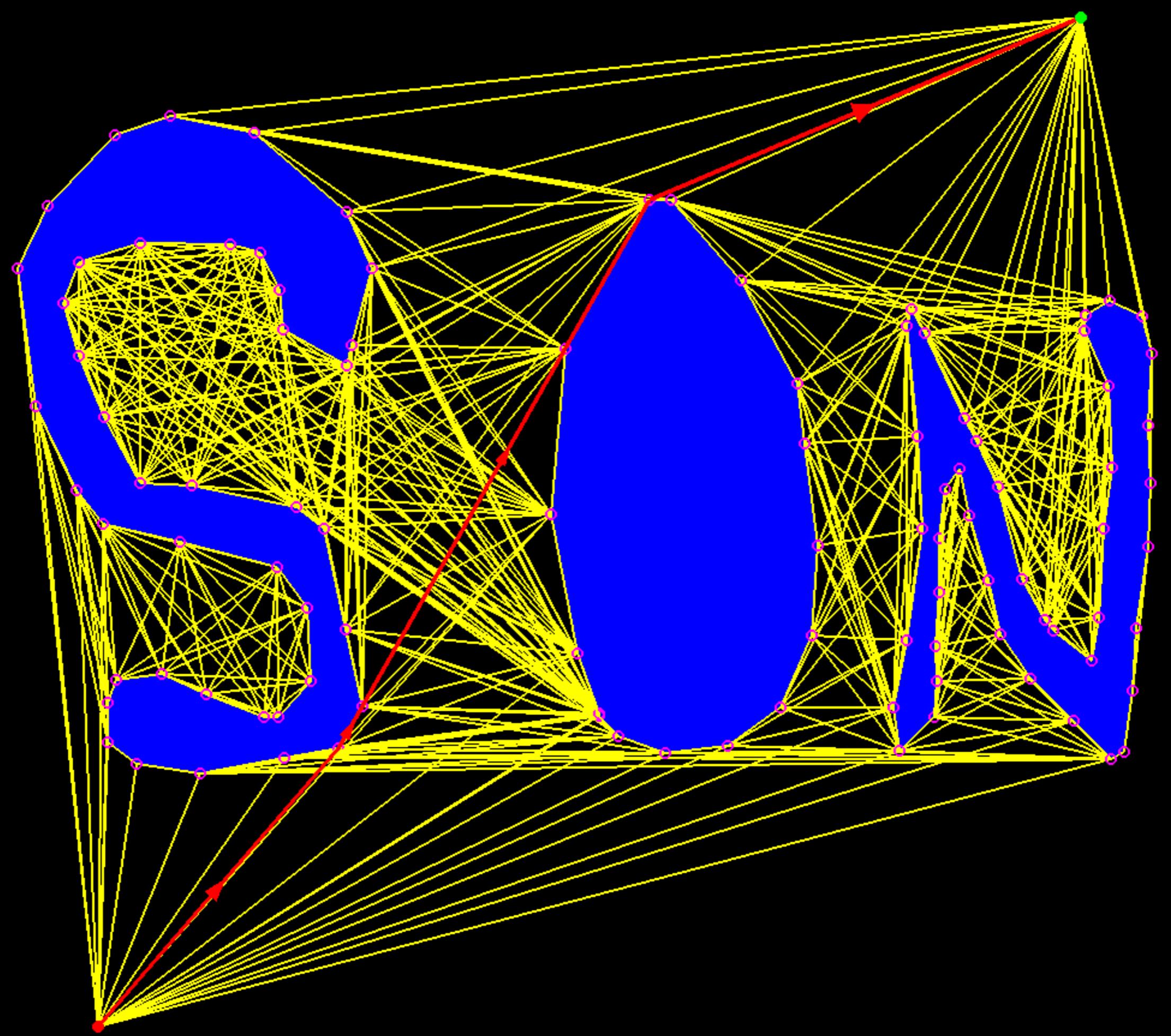


- Claim: any shortest path must be a path in the VG









$n$  = complexity of obstacles  
(total number of edges)

# Point robot in 2D with shortest paths

## Algorithm

- Compute visibility graph
  - $V = \{\text{set of vertices of obstacles} + p_{\text{start}} + p_{\text{end}}\}$
  - $E = \{\text{all pairs of vertices } (u,v) \text{ such that } uv \text{ are visible to each other}\}$
- SSSP (Dijkstra) in VG from start to finish

- Questions to answer
  - What's the size of the VG?
  - How to compute it and how long?

$n$  = complexity of obstacles  
(total number of edges)

## Classwork

- Consider a scene where the total size of the obstacles is  $n$ . Come up with an example that triggers smallest/largest number of edges in VG (up to a constant factor).

# Classwork

- Come up with a straightforward algorithm to compute VG and analyze it

# Classwork

- How long does it take to run Dijkstra's algorithm on VG?

$n$  = complexity of obstacles  
(total number of edges)

## Point robot in 2D with shortest paths

- Complexity of VG
  - $V_{VG} = O(n)$ ,  $E_{VG}=O(n^2)$  <----- can have quadratic size
- Computing VG
  - naive: for each edge, check if intersects any obstacle.  $O(n^3)$
  - improved:  $O(n \lg n)$  per vertex,  $O(n^2 \lg n)$  total ← later
- Dijkstra on VG:  $O(E_{VG} \lg n) = O(n^2 \lg n)$

$n$  = complexity of obstacles  
(total number of edges)

## Summary

Optimal planning for point robot in 2D

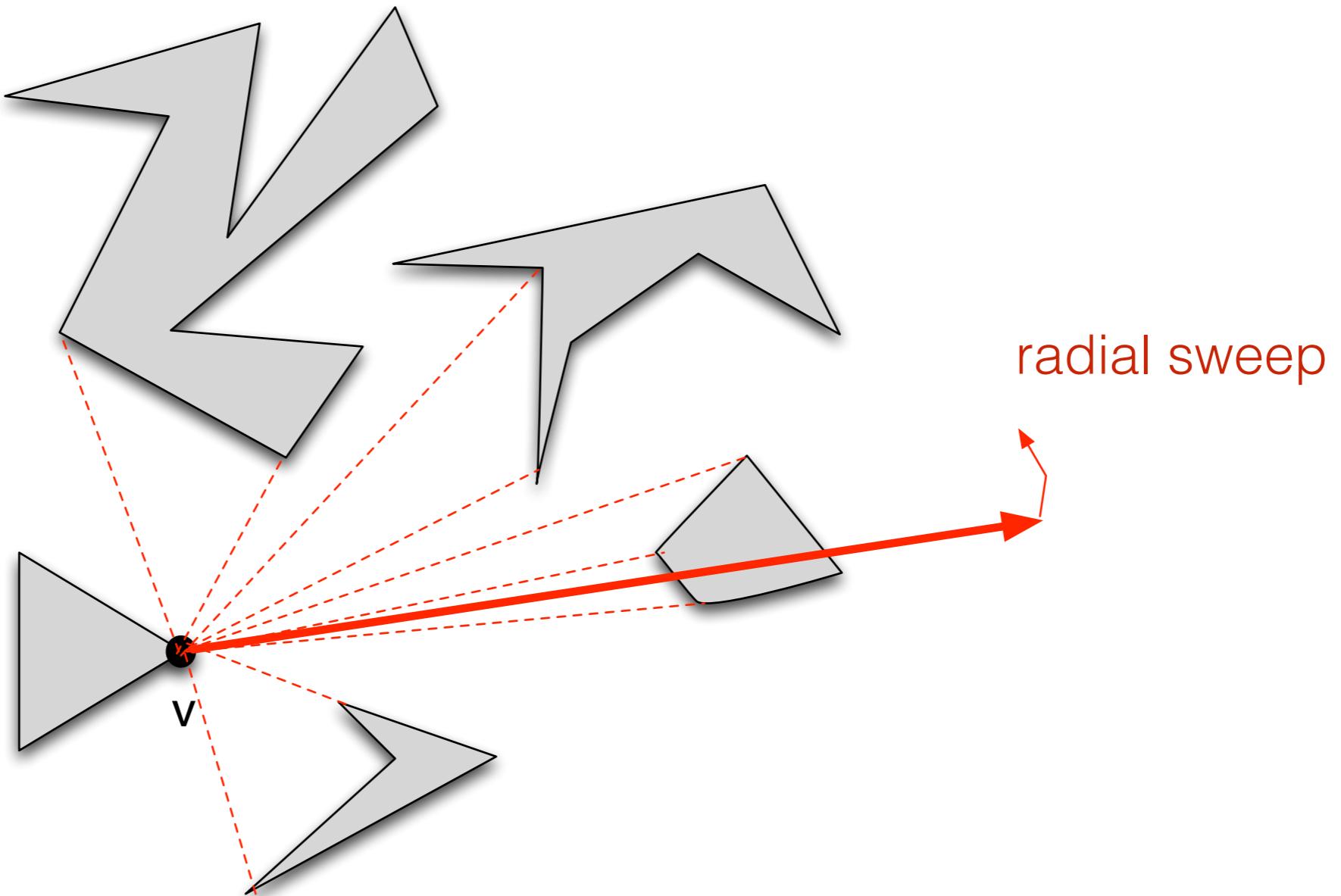
- Compute visibility graph  $\longleftrightarrow O(n^2 \lg n)$
- SSSP (Dijkstra) in VG  $\longleftrightarrow O(E_{VG} \lg n)$

**Theorem:** Given a set of polygonal obstacles with  $n$  edges, a shortest path between two points can be computed in  $O(E_{VG} \lg n) = O(n^2 \lg n)$  time.

# Computing the visibility graph

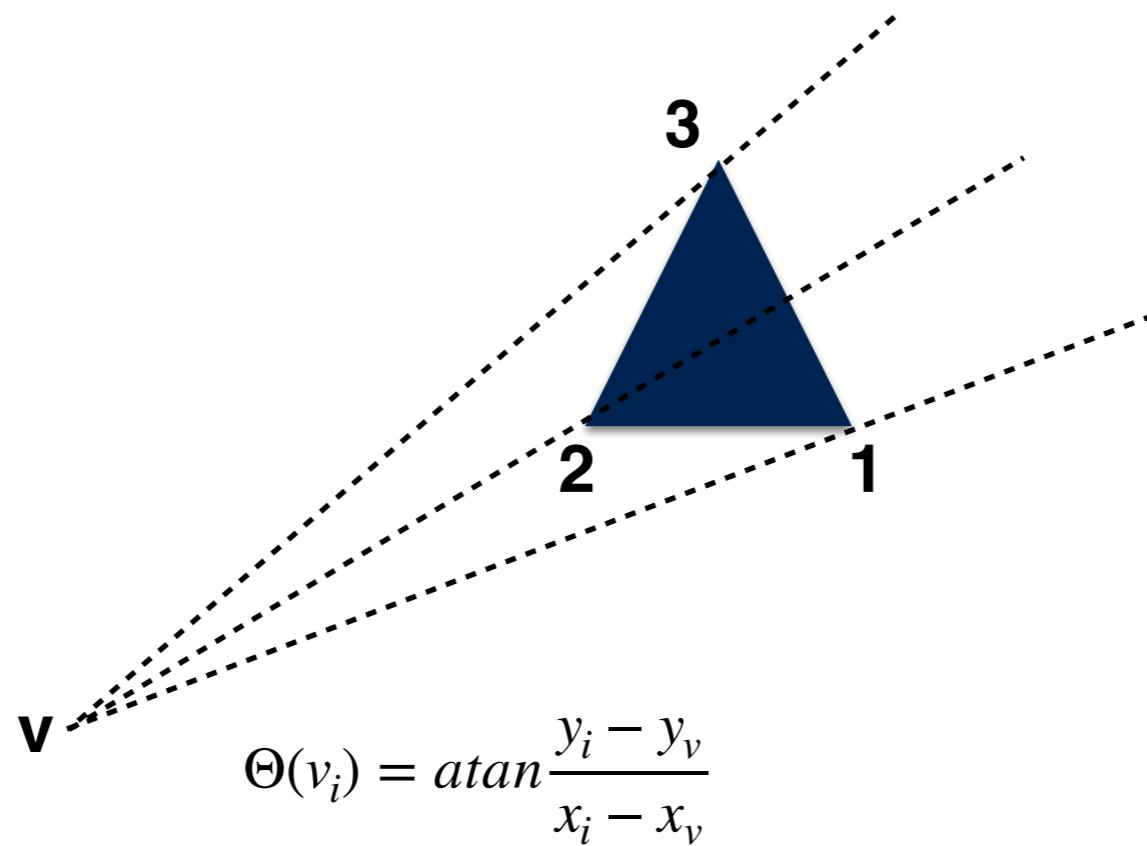
# Improved computation of VG

- For every vertex  $v$ : compute all vertices visible from  $v$  in  $O(n \lg n)$



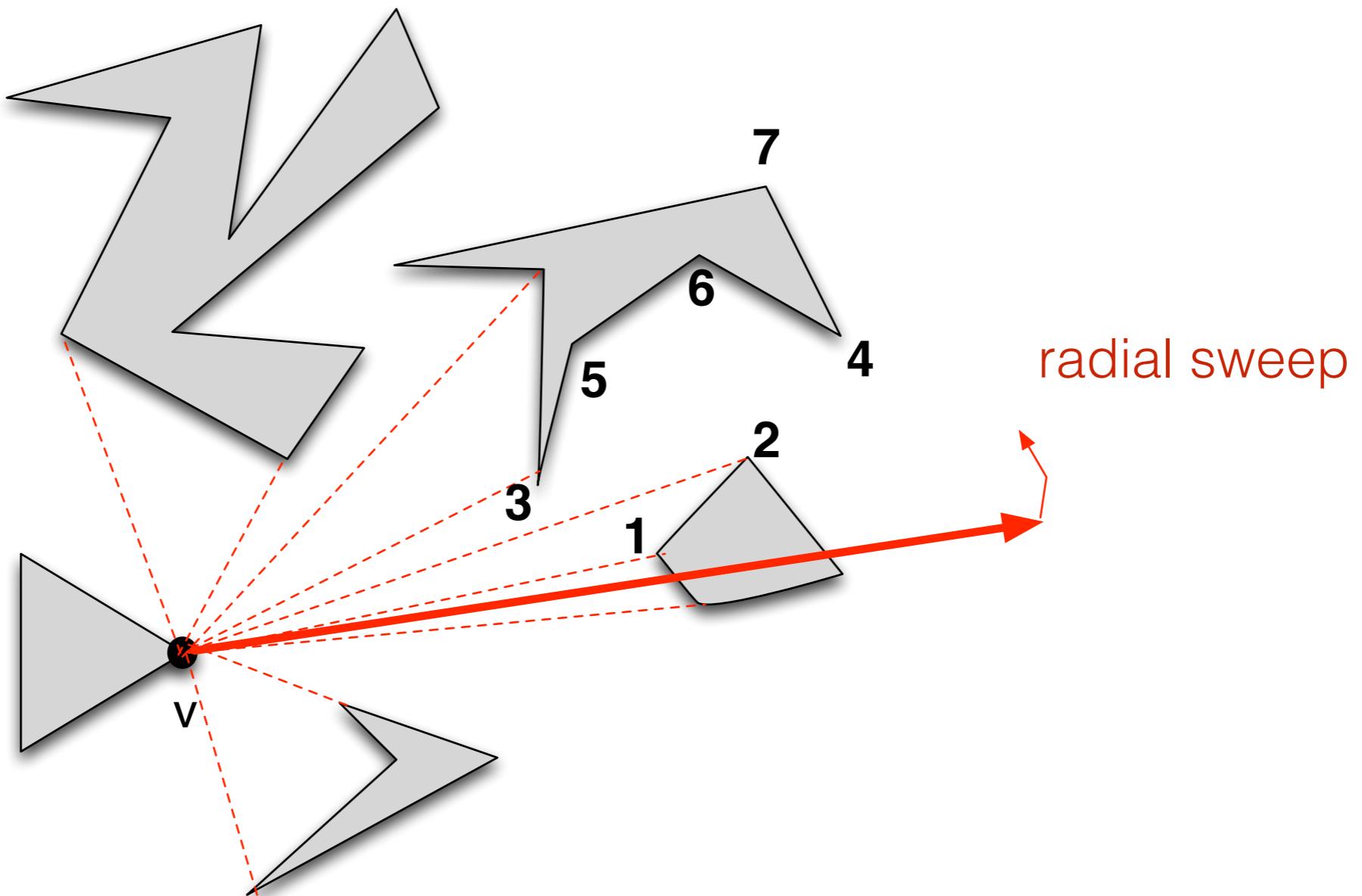
# Improved computation of VG

- Radial sweep: rotate a ray centered at  $v$
- Events: vertices of polygons (obstacles) sorted in radial order
  - for = angles, sorted by distance from  $v$

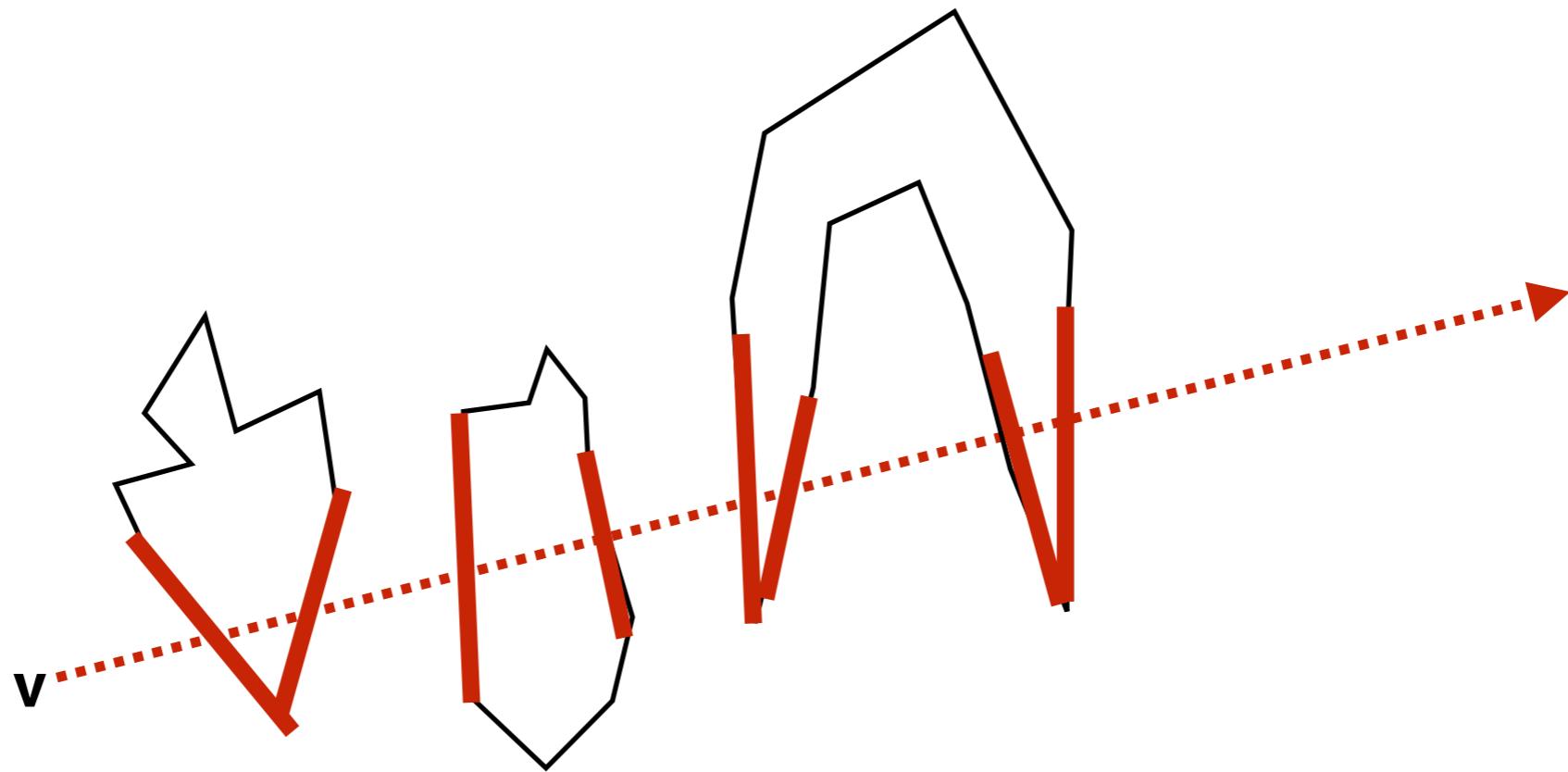


# Improved computation of VG

- Radial sweep: rotate a ray centered at  $v$
- Events: vertices of polygons (obstacles) sorted in radial order
  - for = angles, sorted by distance from  $v$

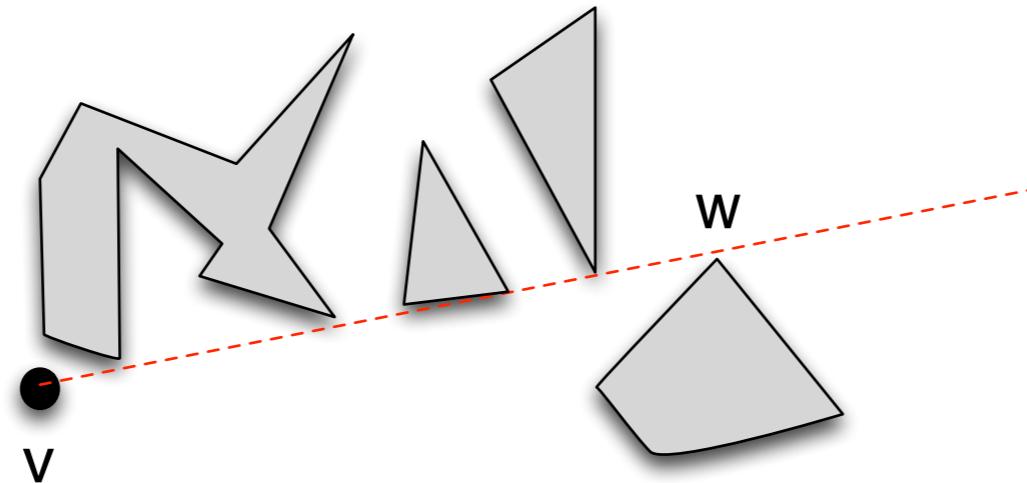


# Improved computation of VG



Active structure (AS) stores all the edges that intersect the sweep line,  
ordered by distance from v

# Improved computation of VG



w visible if vw does not intersect the interior of any obstacle

# Improved computation of VG

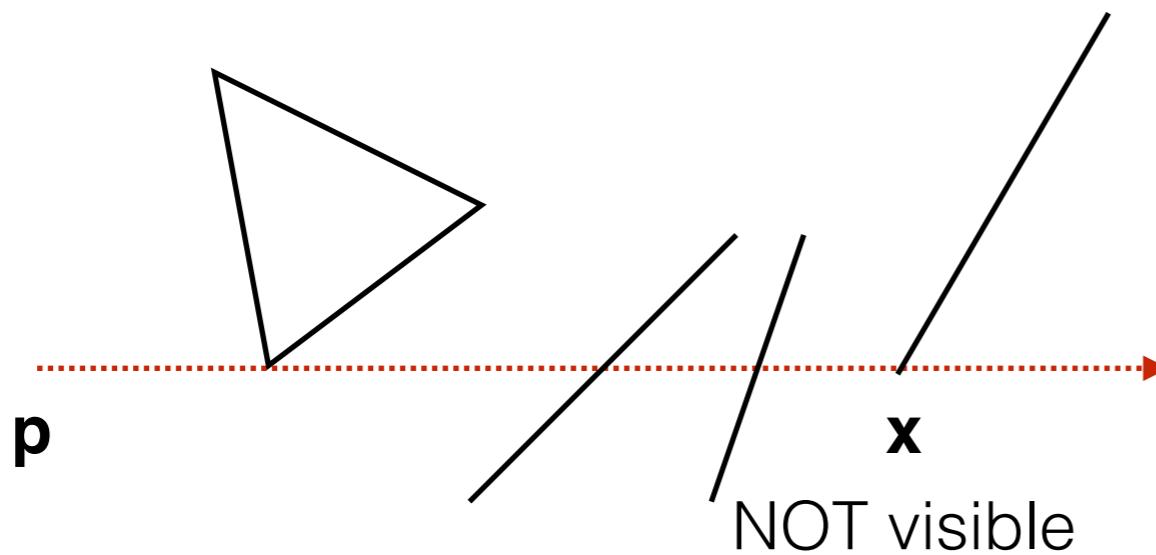
RadialSweep(polygon vertices V, vertex p)

- sort V radially from p, and secondarily by distance from p
- initialize AS with all edges that intersect the horizontal ray from p
- For each vertex v in sorted order:
  - determine if v is visible from p
  - figure out if the edges incident to v are above/below the sweep line.  
If above -> insert edge in AS. If below => delete edge from AS

Runs in  $O(n \lg n)$  time

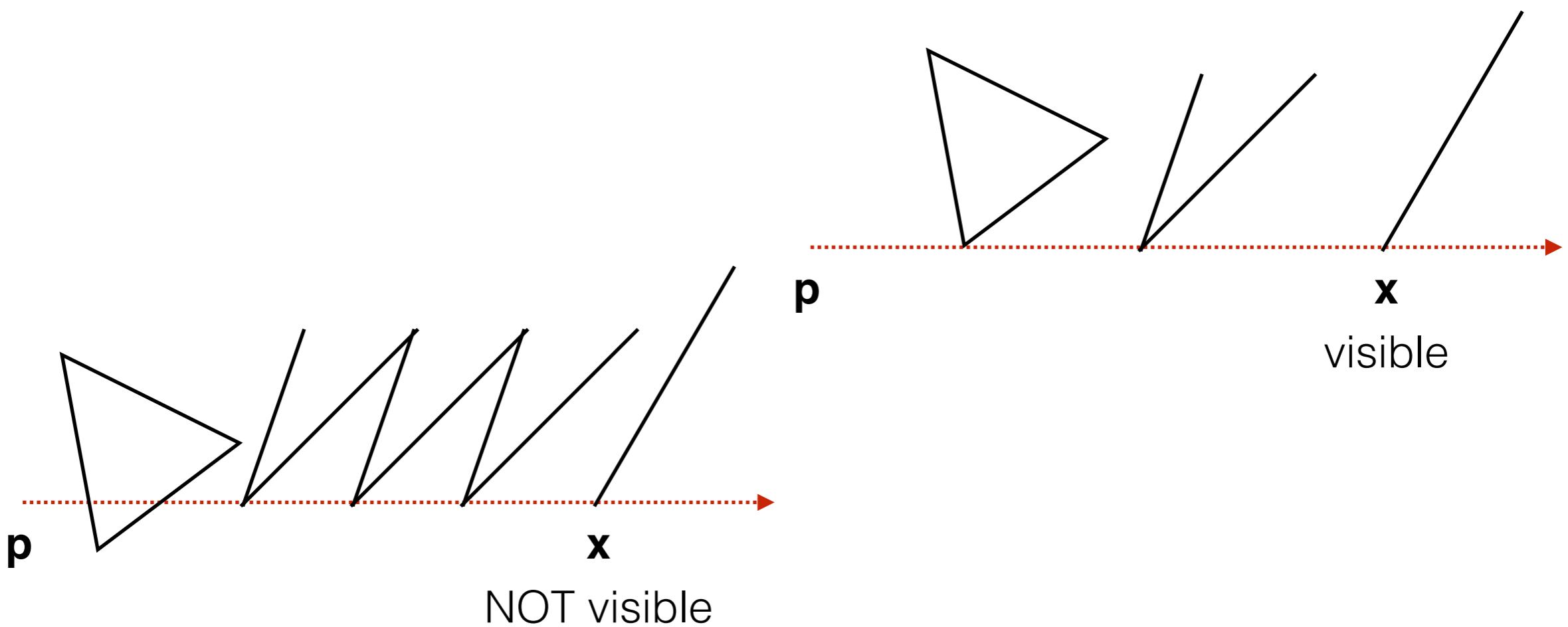
# Is vertex x visible from p?

some cases



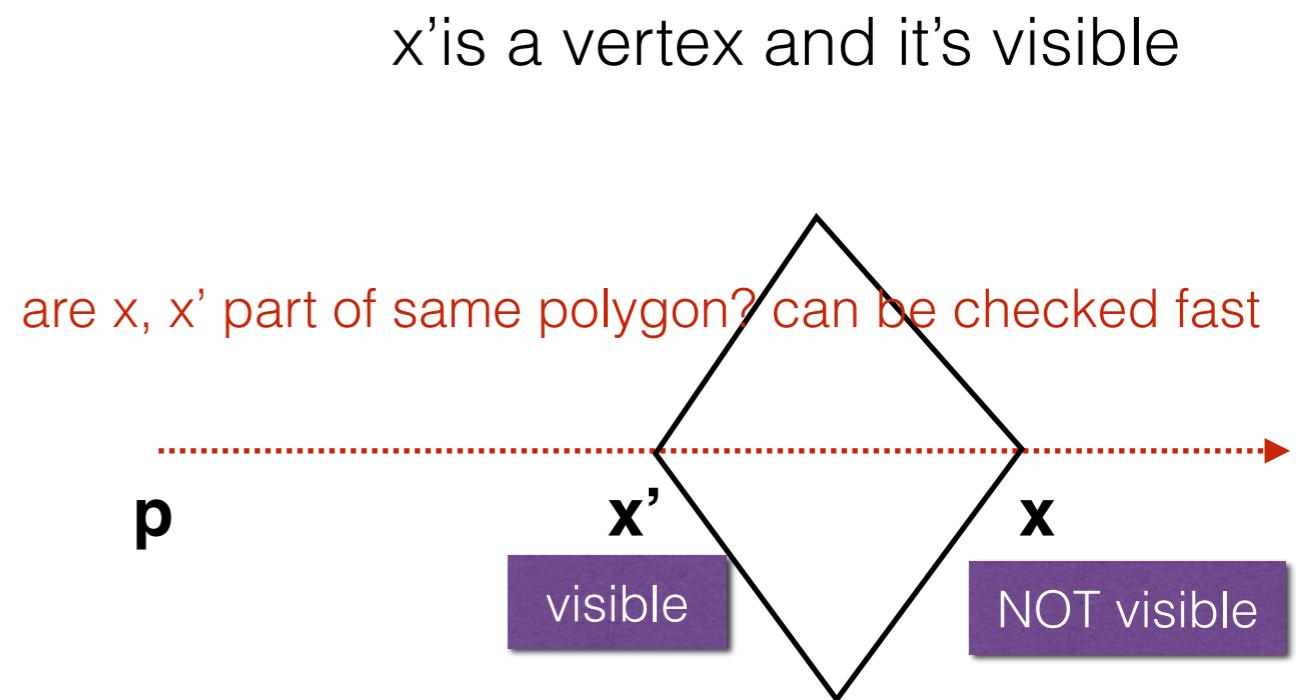
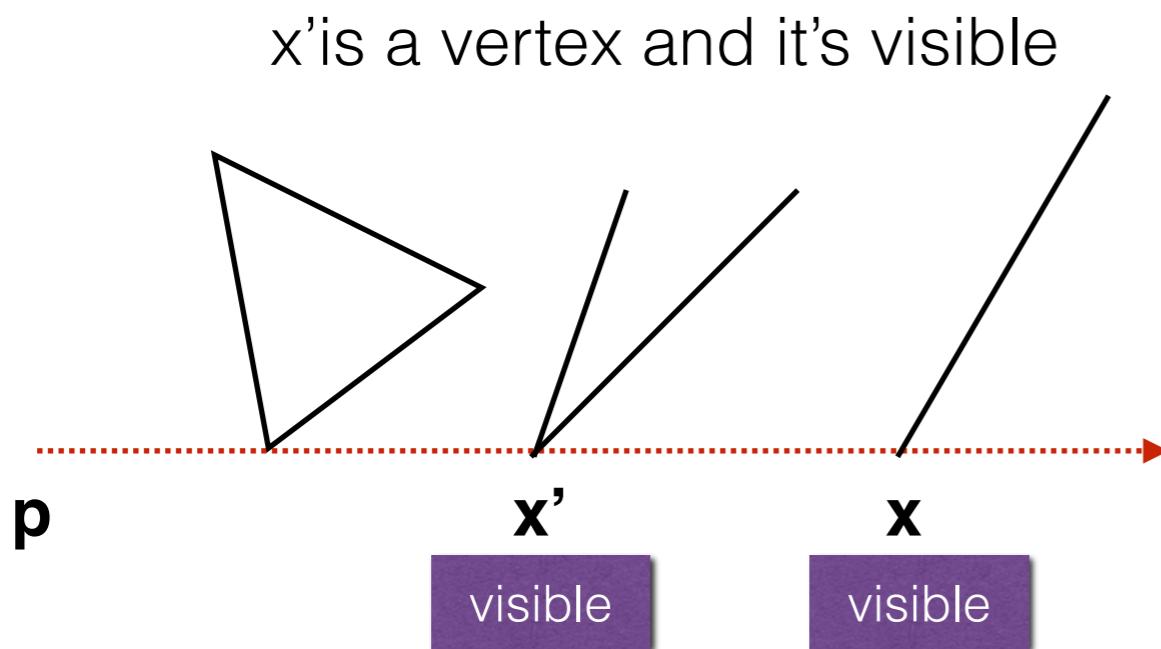
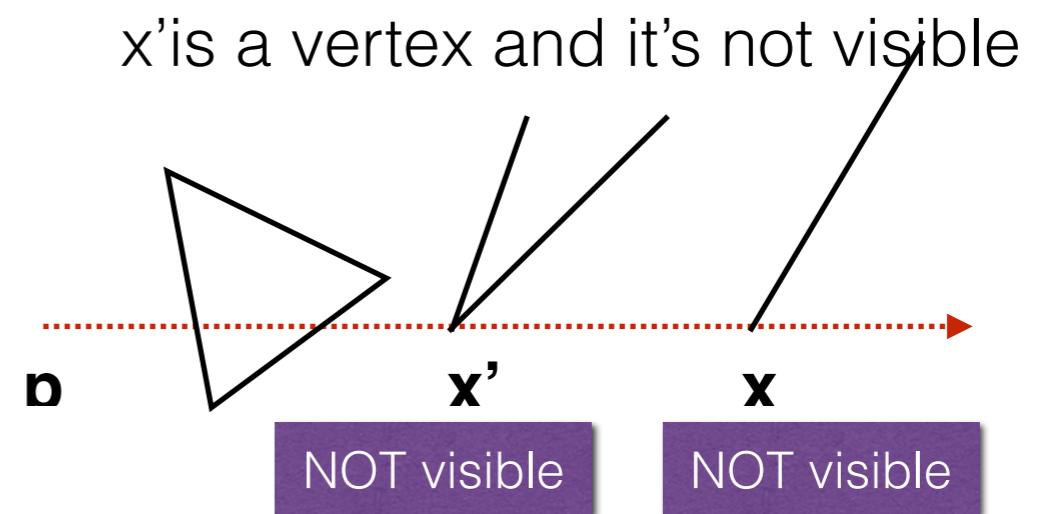
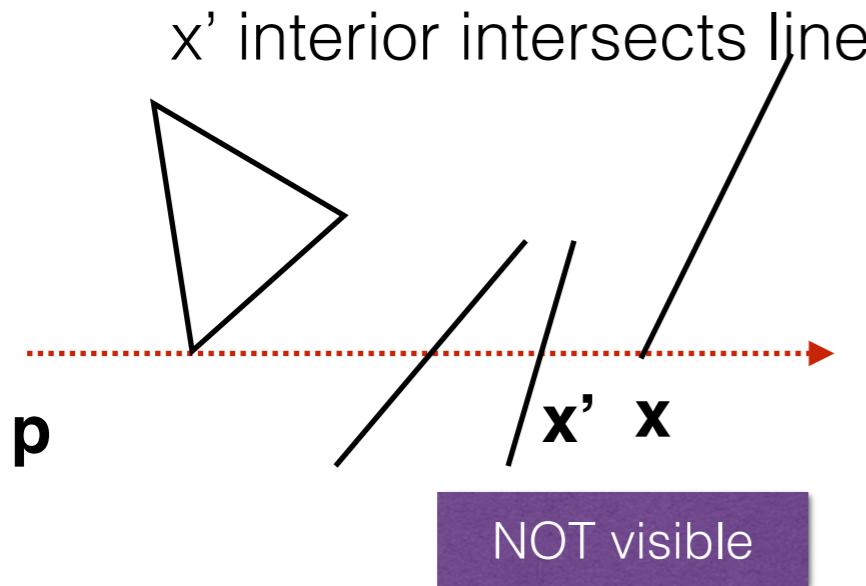
NOT visible:

If there is any edge in AS left of  $x$ , whose interior intersects the line



# Is vertex $x$ visible from $p$ ?

Let  $x'$  be the edge just before  $x$  in the AS,  $x' = \text{AS.predecessor}(x)$



## Is vertex x visible from p?

- check the event just before x in AS (AS.predecessor(x)). Call this x'.
- if x' is an edge whose interior intersects sweep line => x is not visible
- if x' has a vertex on the sweep line then:
  - if x' is not visible => x not visible
  - if x' is visible => x visible, unless they are both on the same polygon

Runs in  $O(\lg n)$  time

Computing the visibility graph

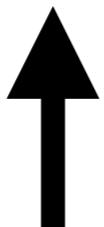
END

# Summary

- Point robot moving among polygonal obstacles
  - Not optimal planning:
    - compute the trapezoid decomposition of free space and a graph that represents it:  $O(n \lg n)$
    - BFS in this graph in  $O(n)$  time
- • Optimal (shortest paths) planning:
  - Compute visibility graph  $\leftarrow O(n^2 \lg n)$
  - SSSP (Dijkstra) in VG  $\leftarrow O(E_{VG} \lg n)$

## Motion planning via Visibility Graph

- + Optimal and complete
- + VG needs to be computed only once, so we can think of it as pre-processing
- VG may be large,  $\Omega(n^2)$



path planning via VG doomed to quadratic complexity

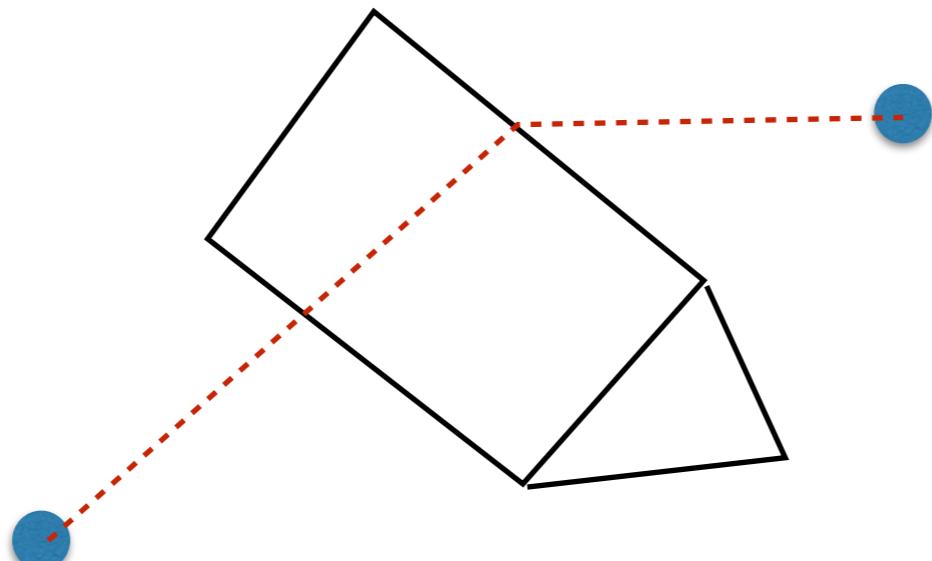
## Some history..

- $O(n^2)$  algorithm (also with a radial sweep)
- Quadratic barrier broken by Joe Mitchell: SP of a point robot moving in 2D can be computed in  $O(n^{1.5+\epsilon})$
- Continuous Dijkstra approach: SP of a point robot moving in 2D can be computed in  $O(n \lg n + k)$  [Hershberger and Suri 1993]
- Special cases can be solved faster:
  - e.g. SP inside a simple polygon w/o holes:  $O(n)$  time

# Visibility Graph in 3D ?

- Inflection points of SP are not restricted to vertices of S, can be inside edges
- Shortest paths in 3D much harder
  - Computing 3D shortest paths among polyhedral obstacles is NP-complete
  - Complete and optimal planning in 3D is hopeless

**VG does not generalize to 3D**



# Where are we?

## 2D path planning problems

- point robot moving inside an arbitrary polygon
- point robot moving among arbitrary polygons
- disk robot moving among arbitrary polygons
- polygonal robot moving among arbitrary polygons
  - translation, translation+rotation
- robot with arms and articulation



next



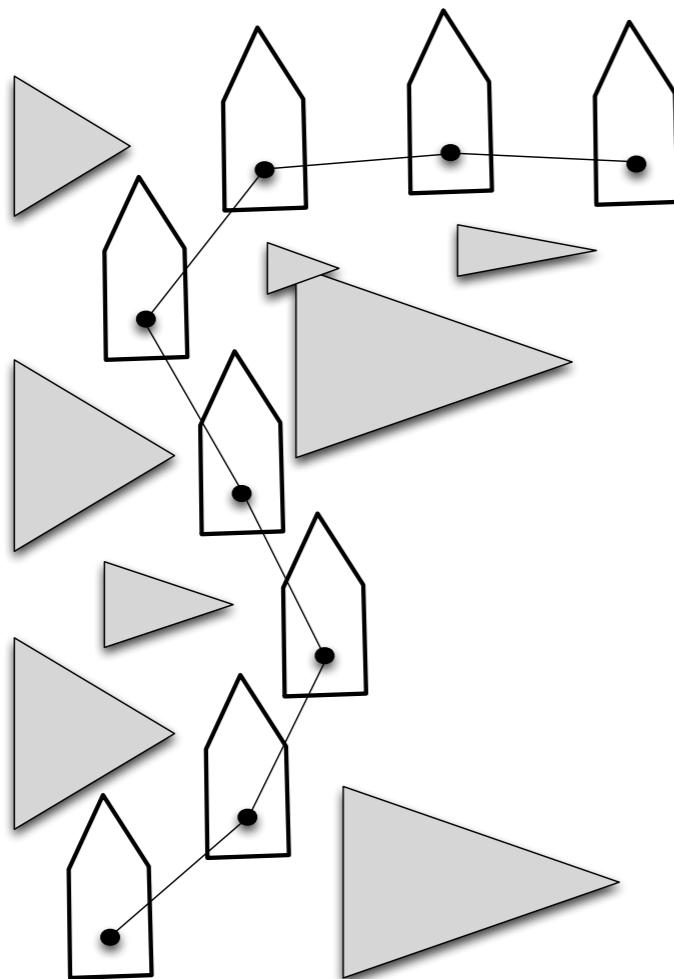
harder



Polygonal robot moving among obstacles in 2D

# Convex polygon moving in 2D

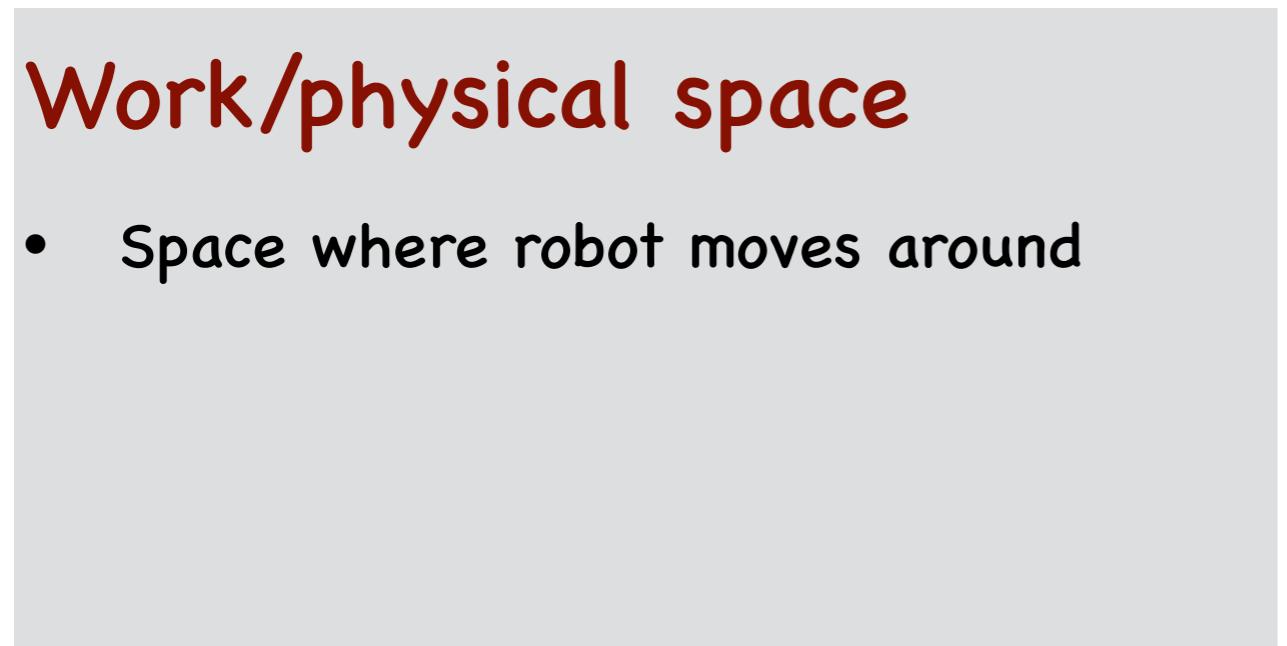
- How can the robot move?
  - Translation only
  - Translation + rotation



screenshot from internet

## Work/physical space

- Space where robot moves around

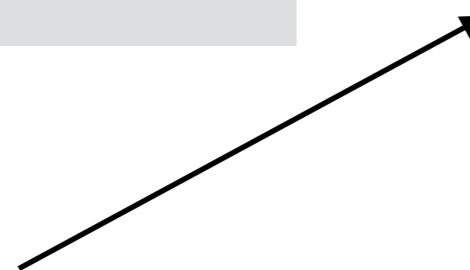


A placement of robot is specified by the degrees of freedom (dof) of the robot

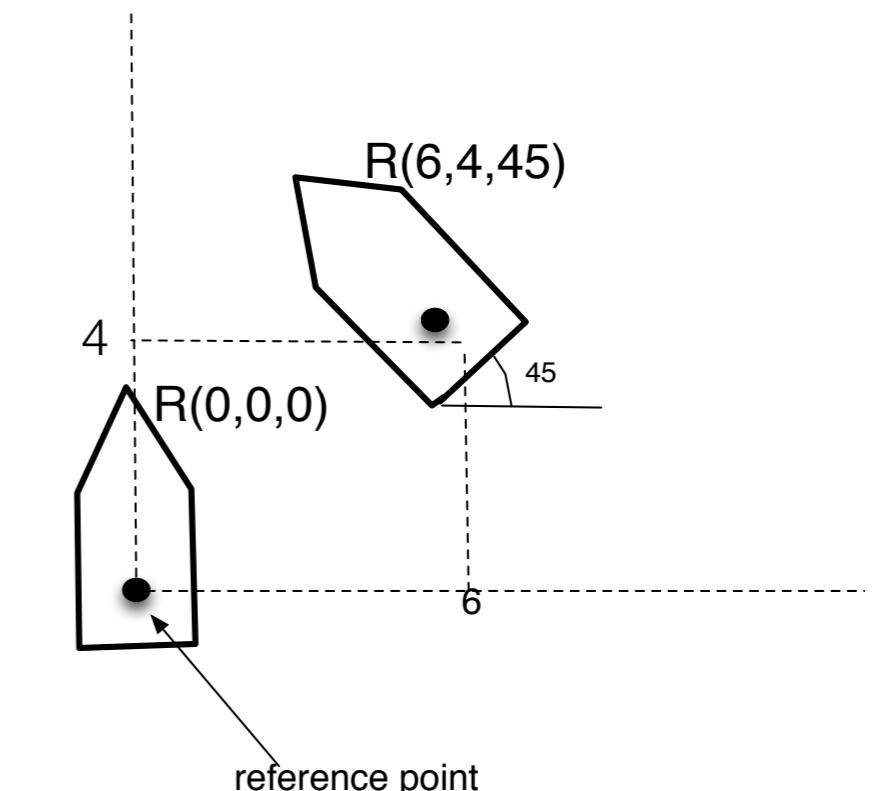
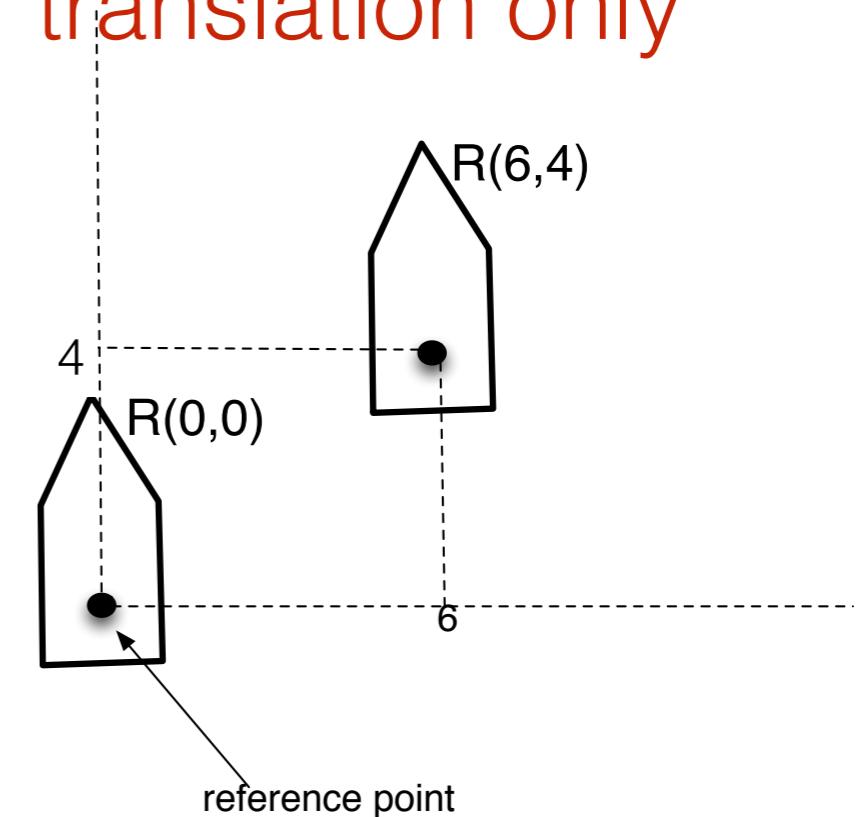
- Example:

$$R(x,y)$$

$$R(x,y,\theta)$$



translation only



translation + rotation

## Configuration space (C-space)

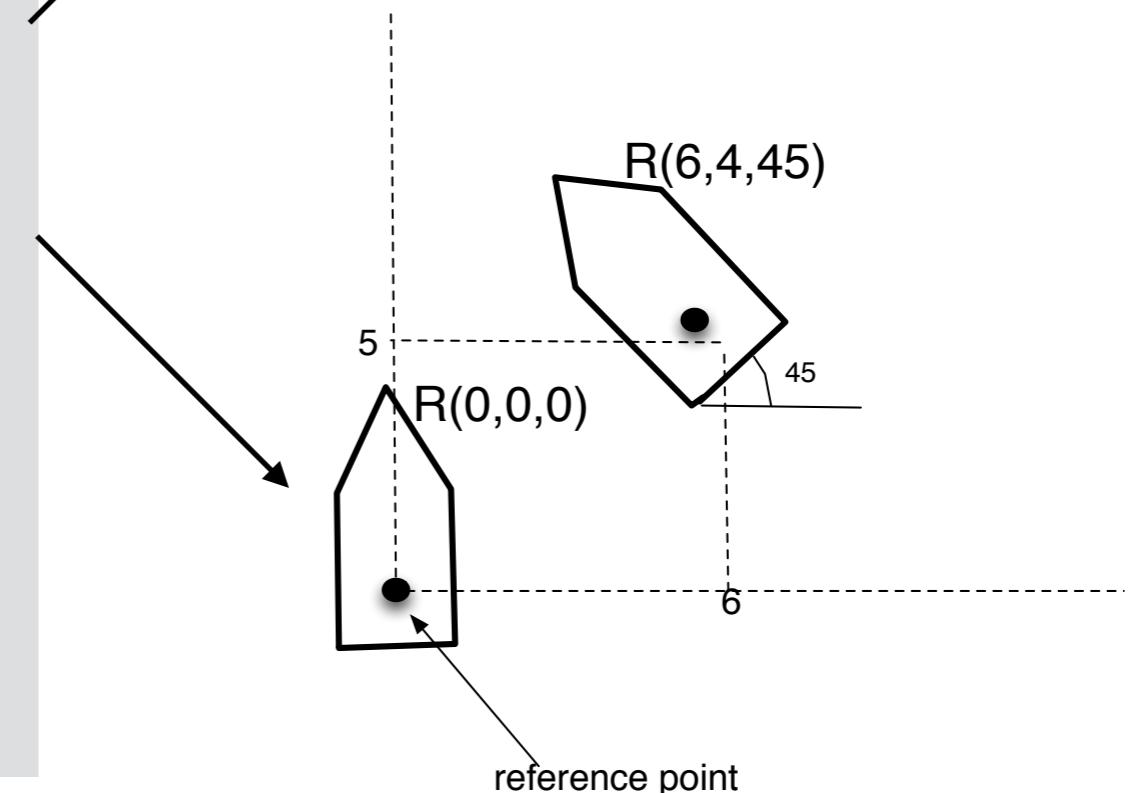
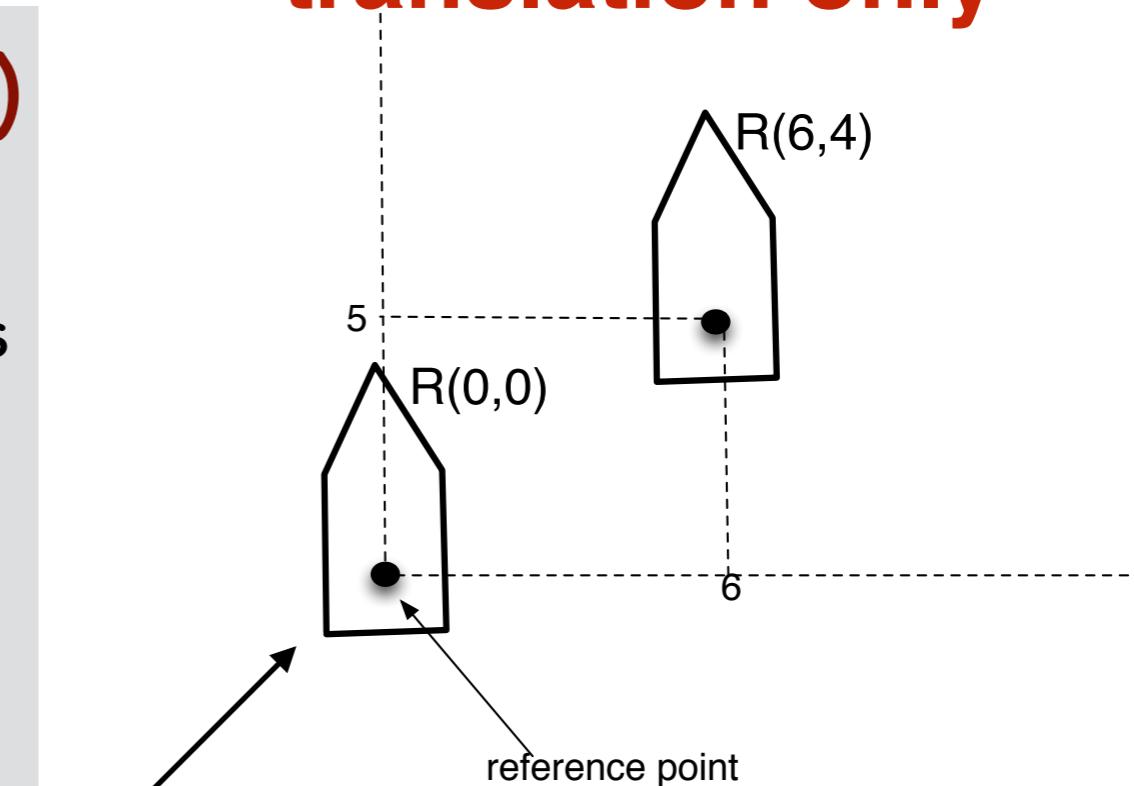
- C-space: The parametric space of the robot = space of all possible placements of the robot
- A point in C-space corresponds to placement of the robot in physical space

### Examples:

2D, translation only  $\leftrightarrow R(x,y)$

2D, transl.+ rot.  $\leftrightarrow R(x,y, \theta)$

translation only



translation + rotation

# Physical Space and C-space

robot	physical space	C-space
polygon, (translation only)	$R(x,y)$	2D

# Physical Space and C-space

robot	physical space	C-space
polygon, (translation only) $R(x,y)$	2D	2D
polygon, (translation + rotations) $R(x,y, \theta)$	2D	3D

# Physical Space and C-space

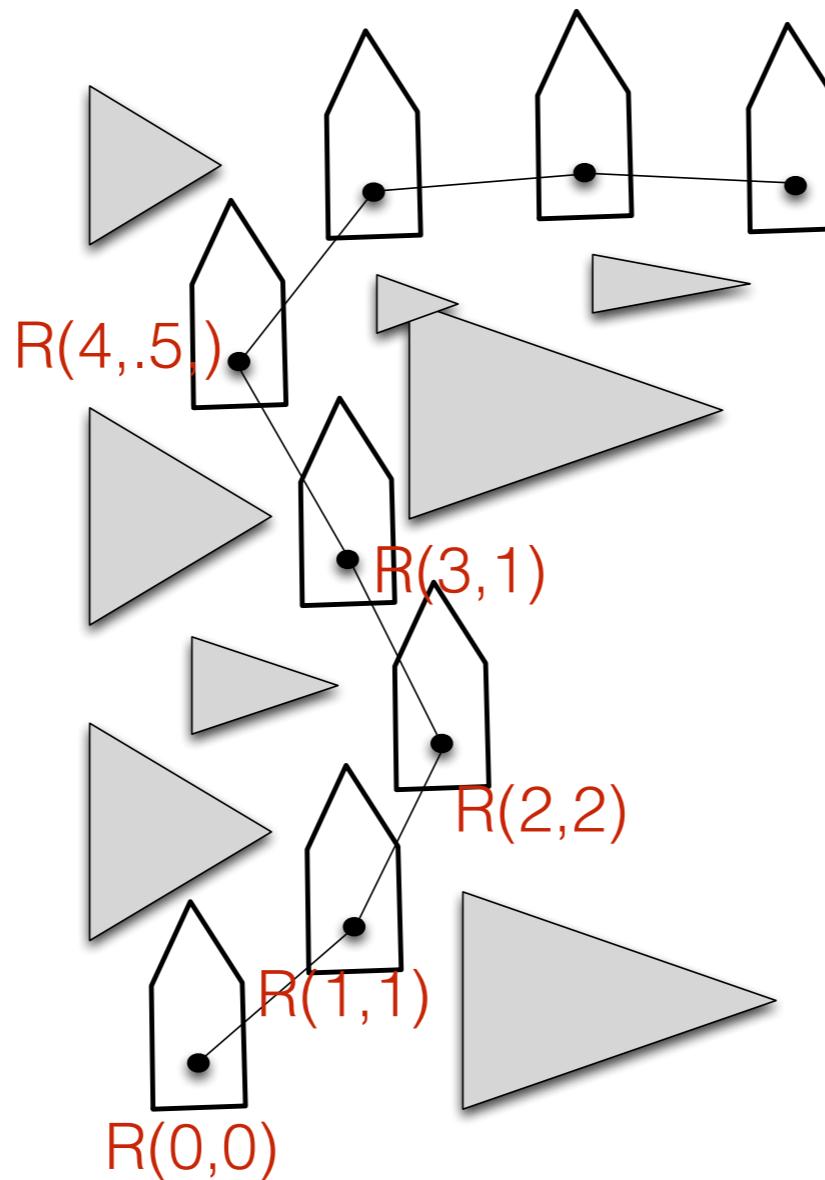
robot	physical space	C-space
polygon, (translation only) $R(x,y)$	2D	2D
polygon, (translation + rotations) $R(x,y, \theta)$	2D	3D
polygon (translation, rotations)	3D	6D

# Physical Space and C-space

robot	physical space	C-space
polygon, (translation only)	2D	2D
polygon, (translation + rotations)	2D	3D
polygon (translation, rotations)	3D	6D
Robot arm with joints	3D	#DOF

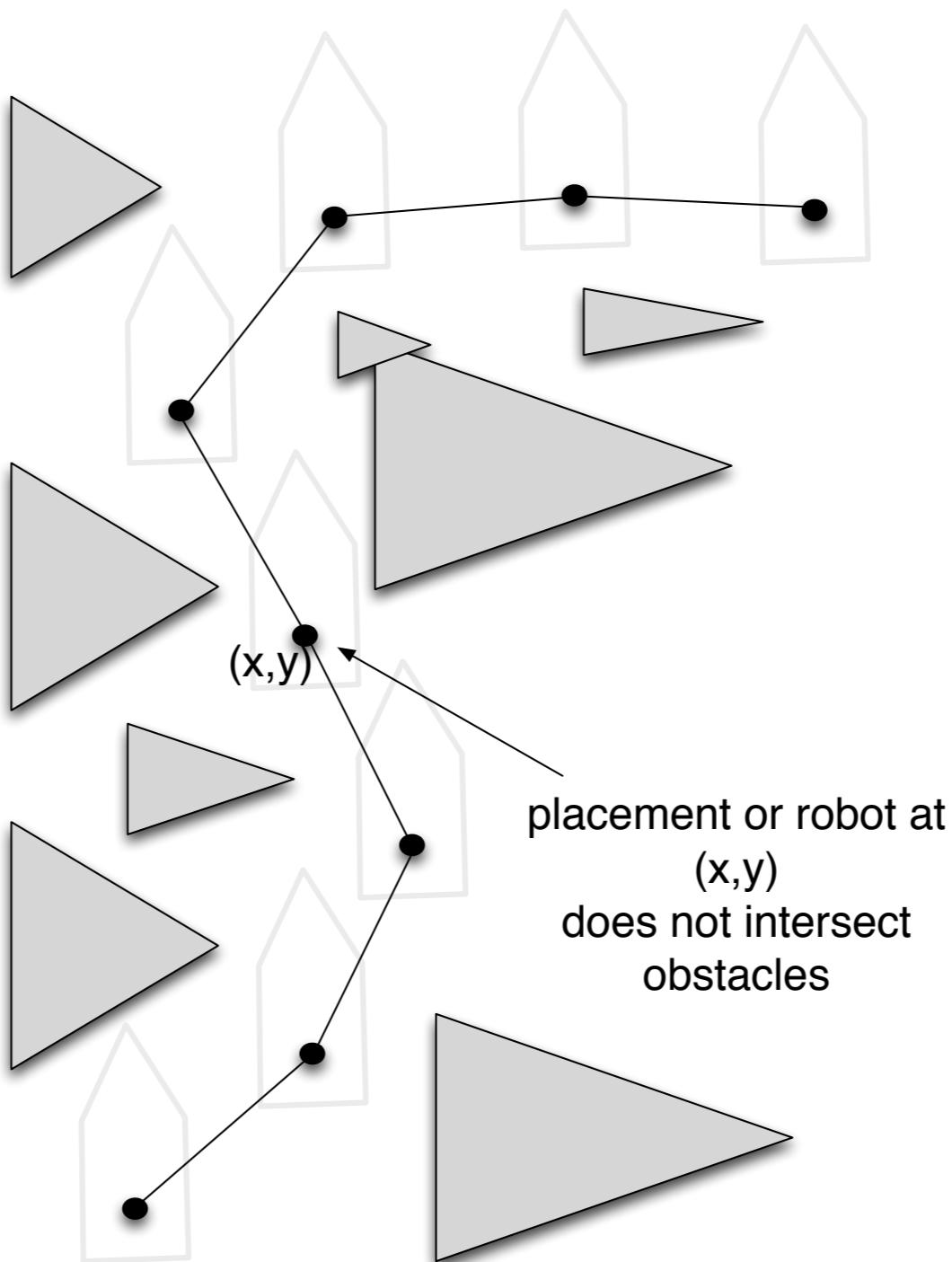
# Path planning in C-space

- Any path for  $R$  corresponds to a path for  $R$  in C-space
- Path planning  $\Rightarrow$  path planning in C-space



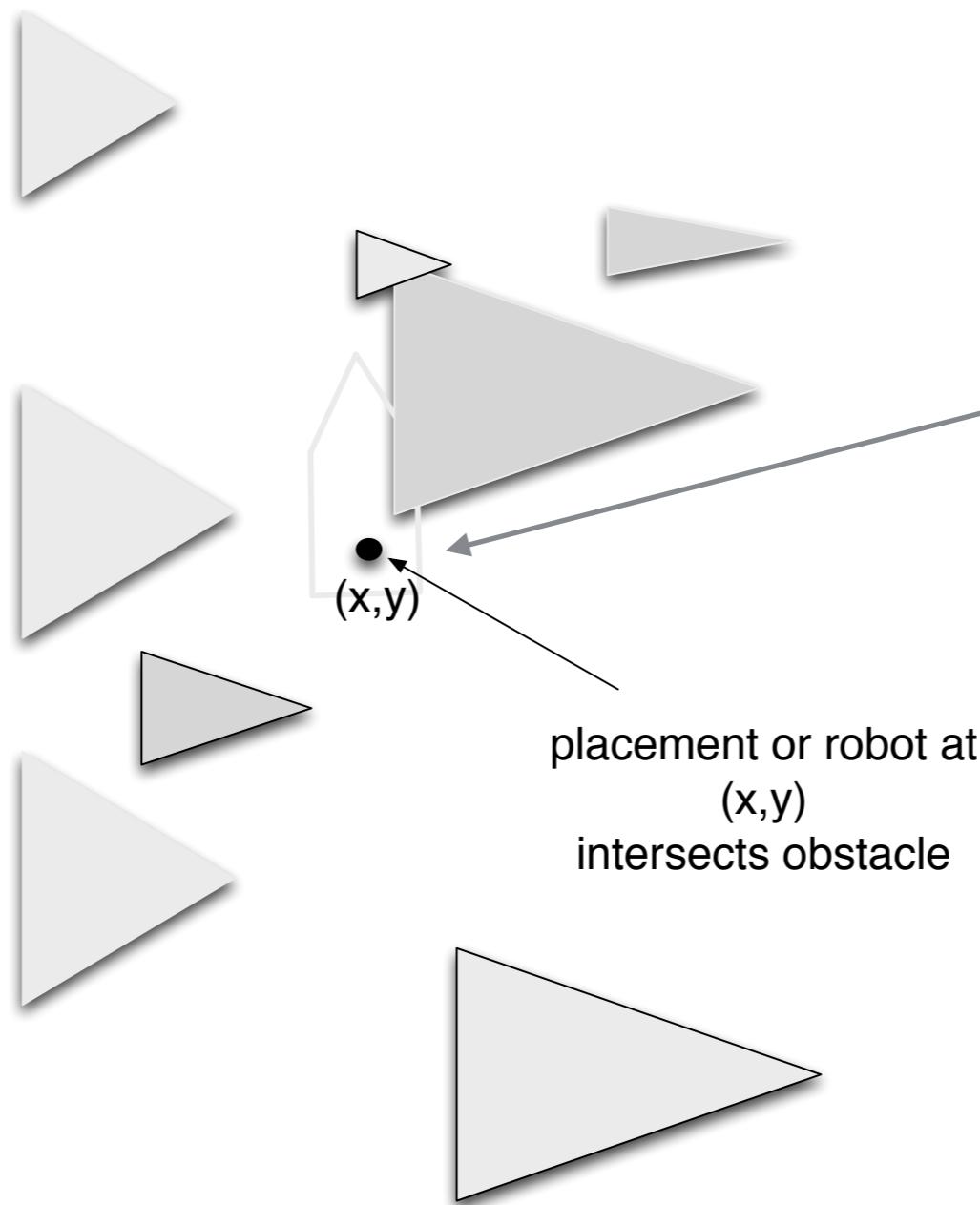
# Free C-space

A point  $(x,y)$  is in free C-space if placing  $R(x,y)$  does not intersect the obstacles



# Forbidden C-space

A point  $(x,y)$  is in forbidden C-space if it is not in free C-space.

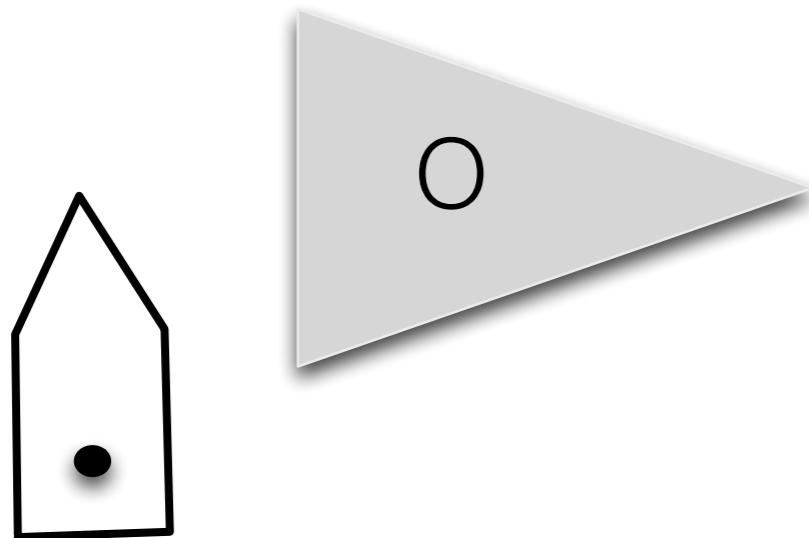


**forbidden C-space:**  
**placements  $(x,y)$**   
**where robot intersects**  
**with obstacle**

placement or robot at  
 $(x,y)$   
intersects obstacle

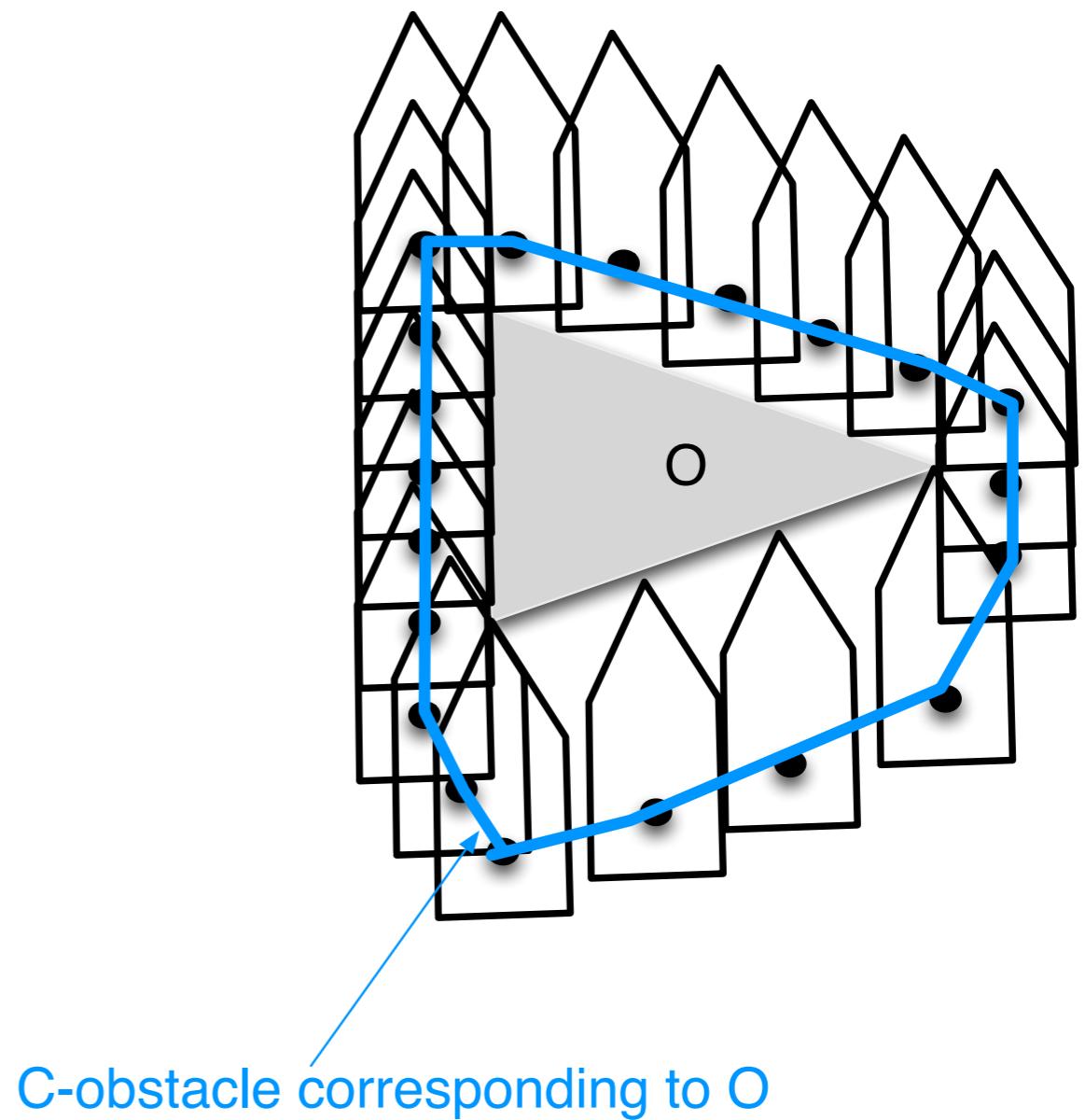
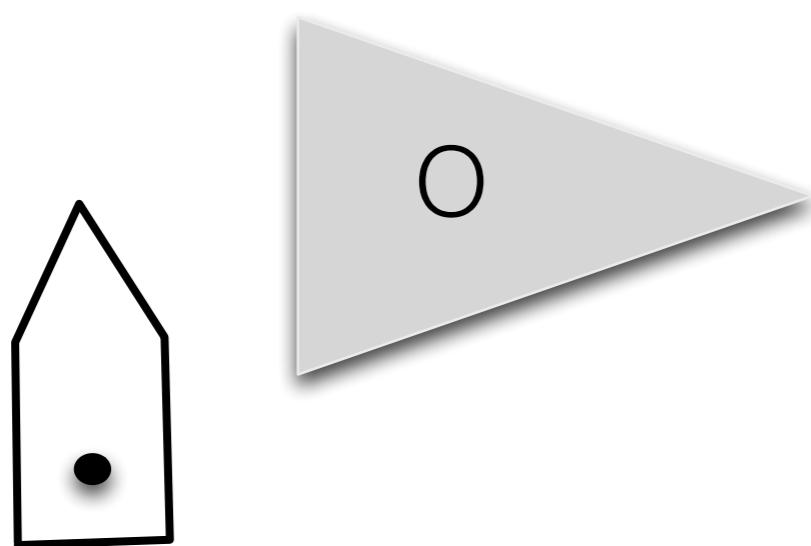
# Extended obstacles or C-obstacles

- Given obstacle O and robot R
  - C-obstacle = the placements of R that cause intersection with O



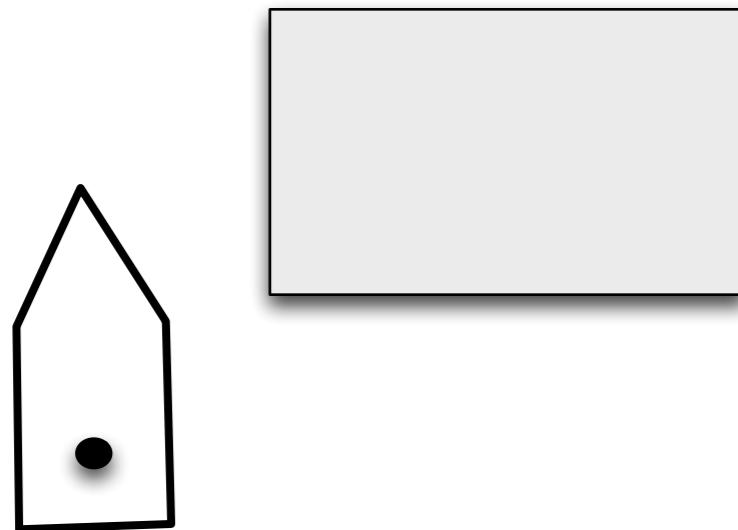
# Extended obstacles or C-obstacles

- Given obstacle  $O$  and robot  $R$ 
  - C-obstacle = the placements of  $R$  that cause intersection with  $O$



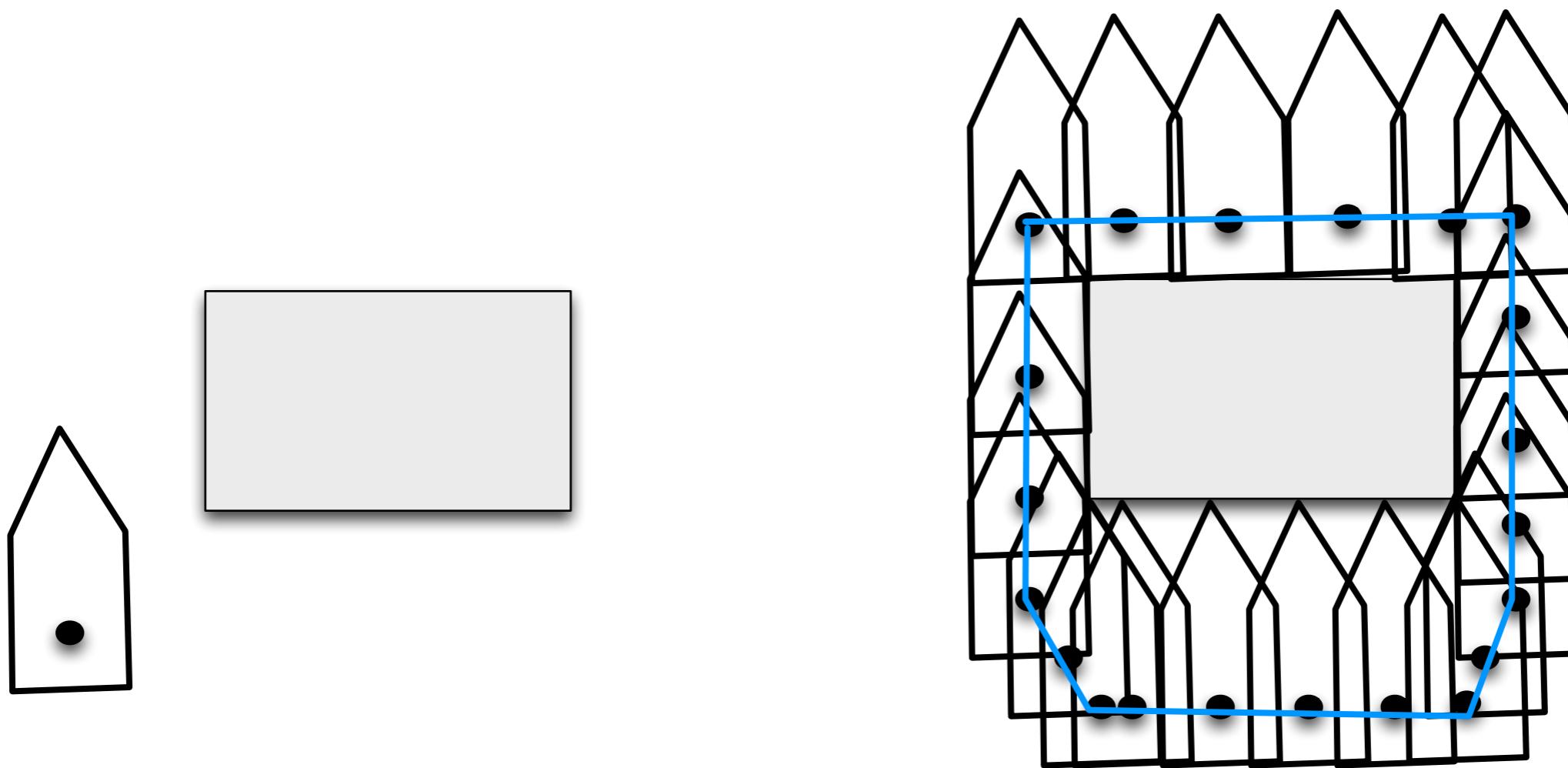
# Extended obstacles or C-obstacles

- Given obstacle O and robot R
  - C-obstacle = the placements of R that cause intersection with O



# Extended obstacles or C-obstacles

- Given obstacle O and robot R
  - C-obstacle = the placements of R that cause intersection with O



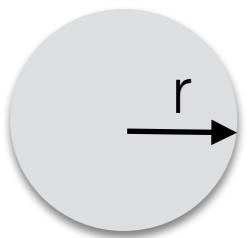
## Class work

- Consider a rectangular robot. Draw a small set of obstacles such that their C-obstacles overlap.

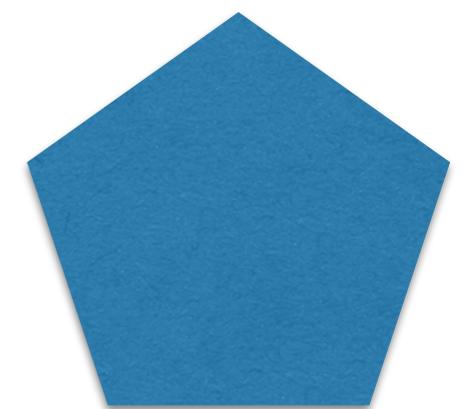
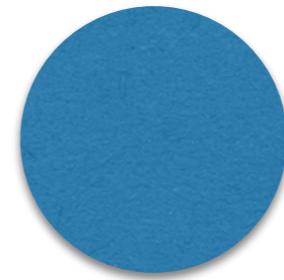
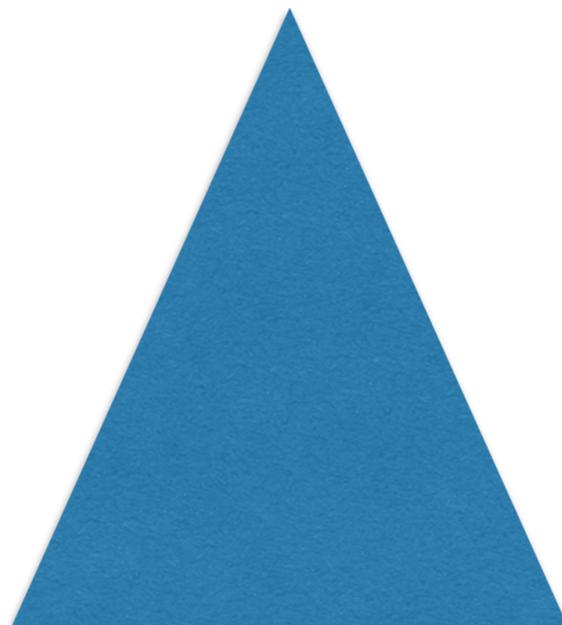
## Class work

- Consider a rectangular robot. Draw a scene of obstacles such that free physical space is not disconnected, but the the free C-space is disconnected.

# Class work

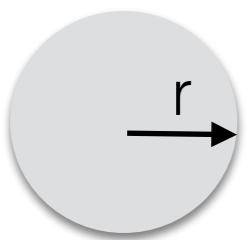


robot

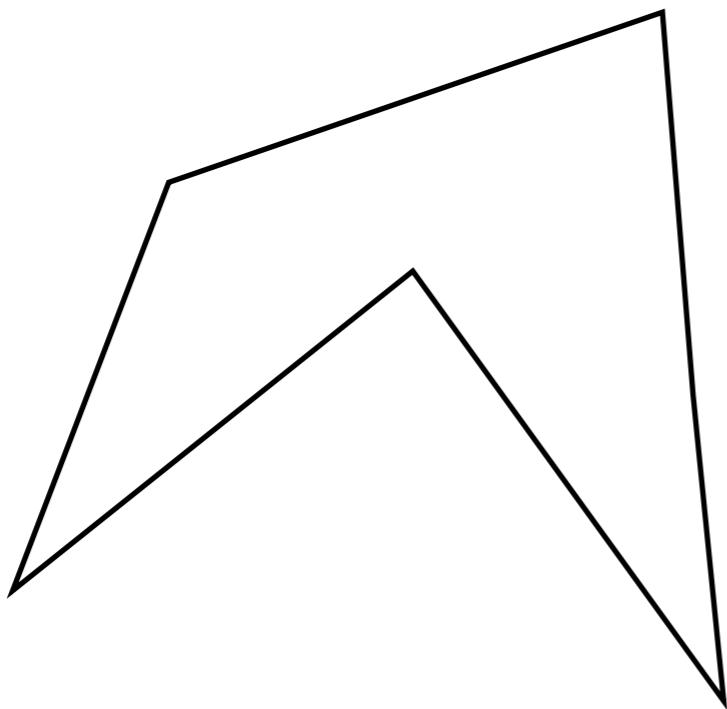


Show the corresponding C-obstacles for a disc robot.

# Class work



robot

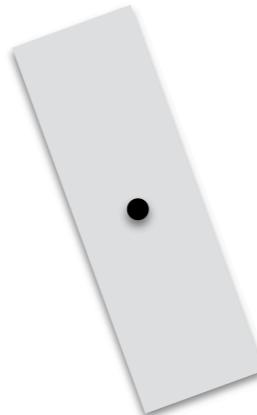


obstacle

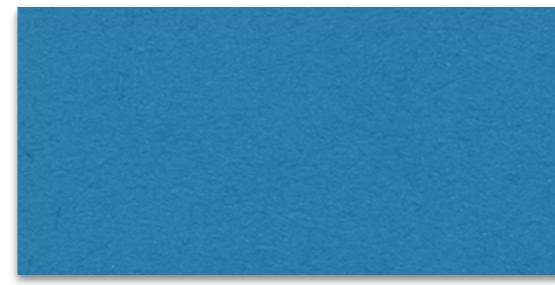
extended obstacle

# Class work

translation only



robot



Show the corresponding C-obstacle.

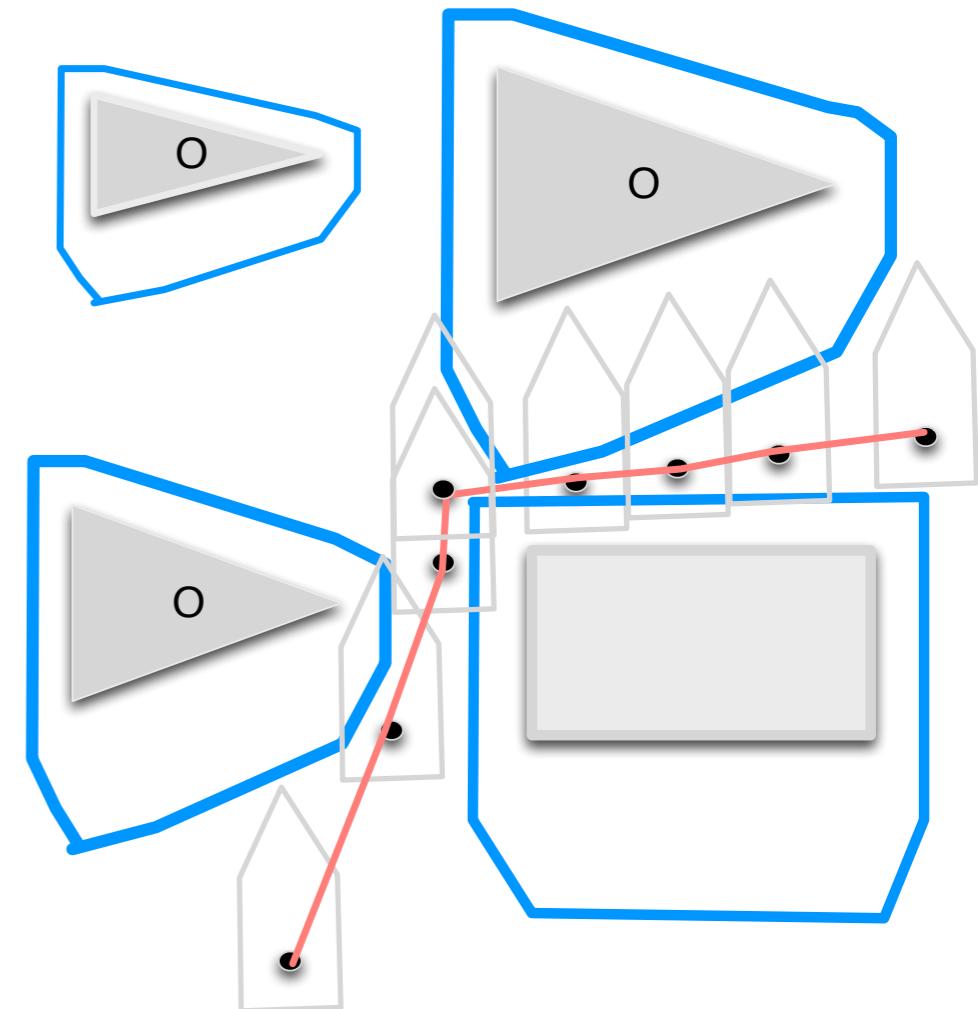
Polygonal robot translating in 2D

# Polygonal robot translating in 2D

## Algorithm

Complete, non optimal.

- For each obstacle  $O$ , compute the corresponding C-obstacle
- Compute the union of C-obstacles
- Compute its complement. That's the free C-space
  - //problem is reduced to a point robot
  - //moving in free C-space
- Compute a trapezoidal map of free C-space
- Use it to compute a roadmap of free space



**How and how fast can this be done?**

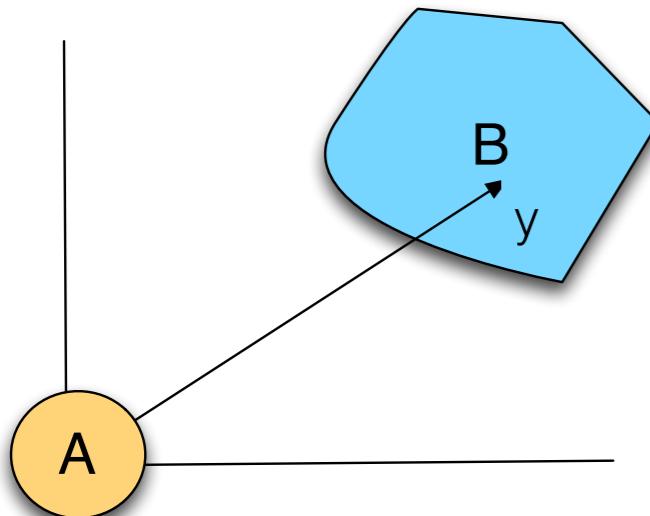
How to compute C-obstacles?

# Minkowski sum

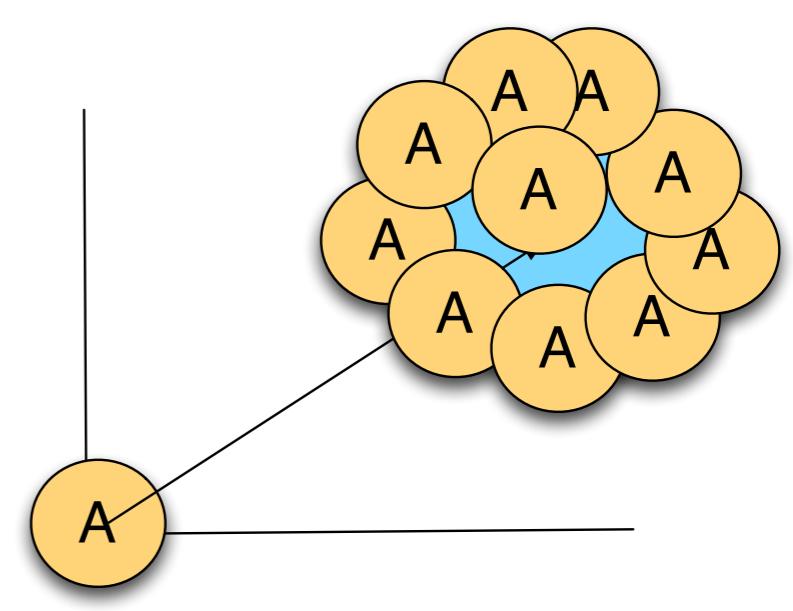
- Let  $A, B$  two sets of points in the plane
  - Define  $A \odot B = \{x + y \mid x \text{ in } A, y \text{ in } B\}$  ← Minkowski sum

The diagram illustrates vector addition. On the left, the text "vector sum" is written below a black arrow pointing diagonally upwards and to the right. On the right, the text "x, y vectors" is written below a black arrow pointing vertically upwards.

- Interpretation: consider set A to be centered at the origin. Then  $A \oplus B$  represents many copies of A, translated by y, for all y in B; i.e. place a copy of A centered at each point of B.

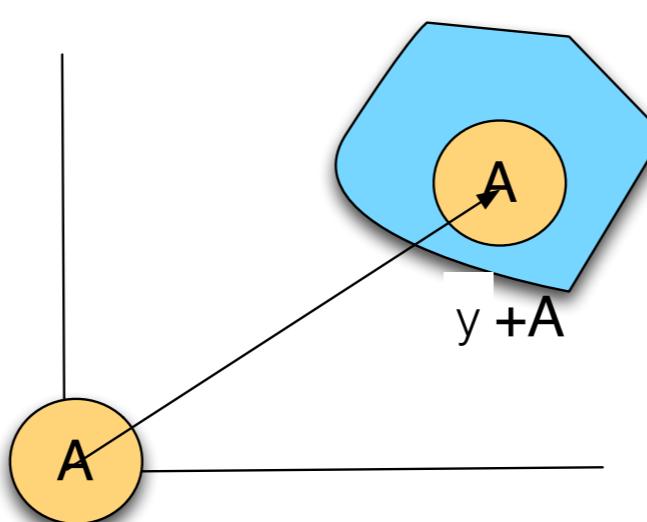
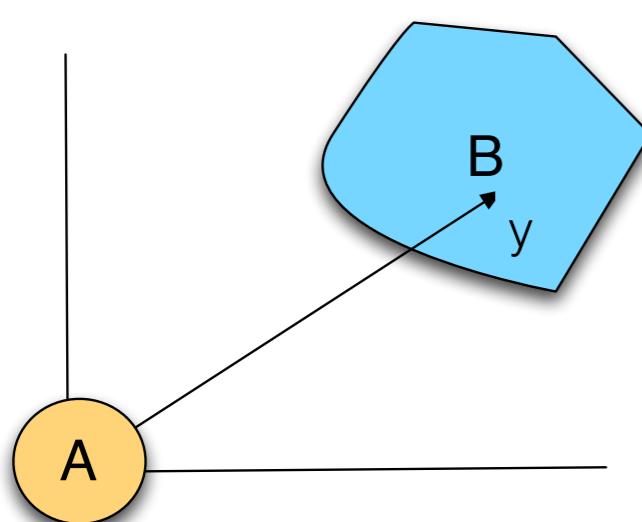


The diagram shows a point  $y$  represented by a black dot on a horizontal axis. A vector  $A$  is shown as a line segment originating from  $y$  and pointing upwards and to the right. The resulting position,  $y + A$ , is indicated by a blue shaded hexagon representing a small neighborhood around the tip of the vector sum.

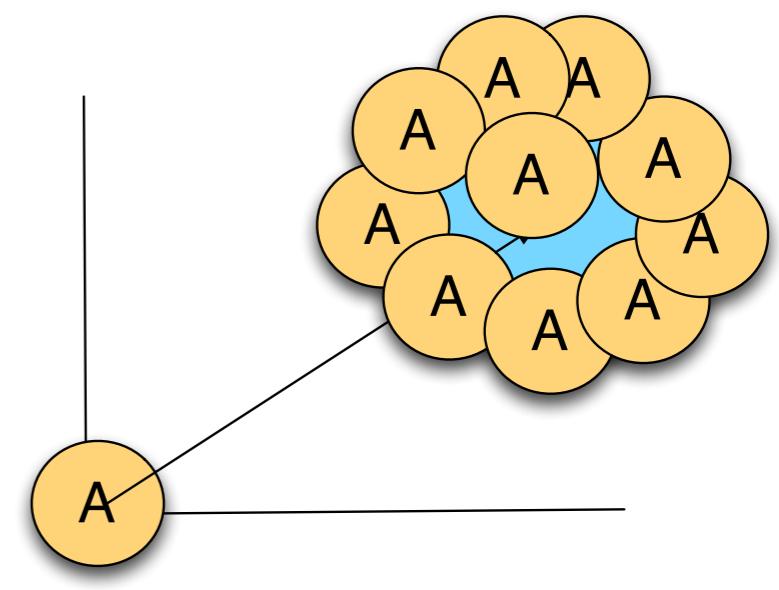


# Minkowski sum

- $A \oplus B$ : Slide A so that the center of A traces the edges of B



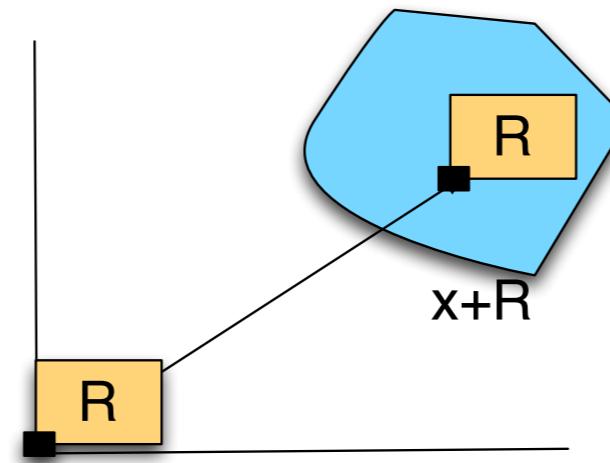
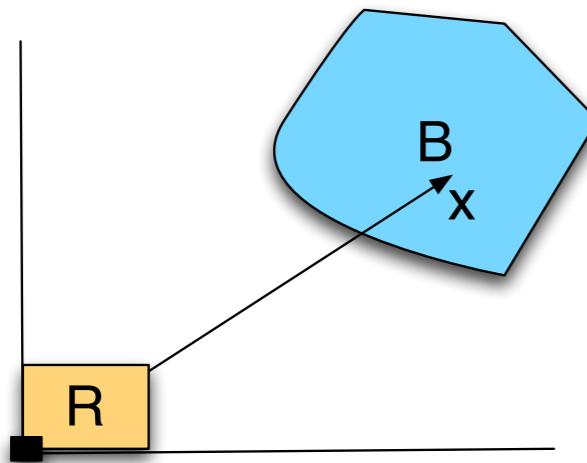
A translated by  $y$



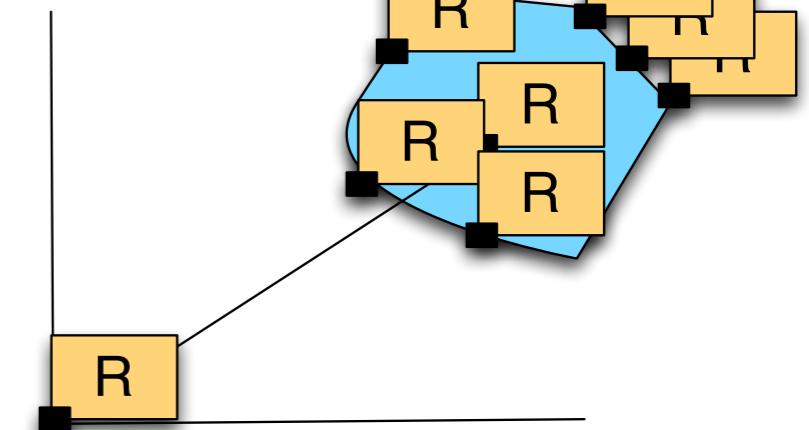
$B \oplus A$

# C-obstacles as Minkowski sums

- Consider a robot  $R$  with the reference in the lower left corner



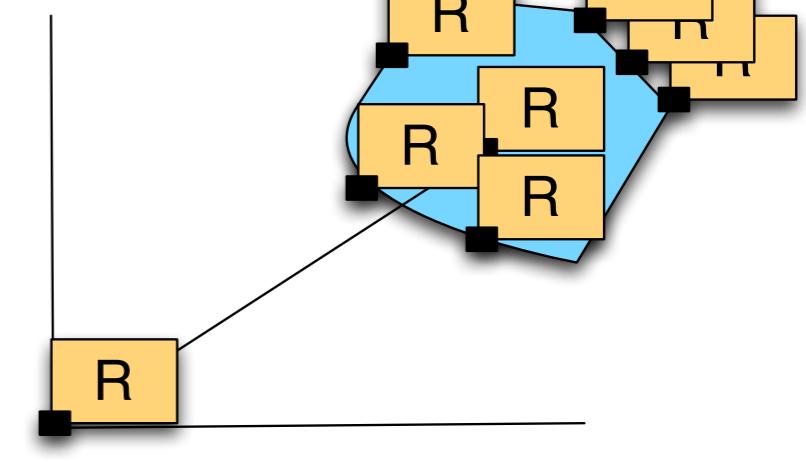
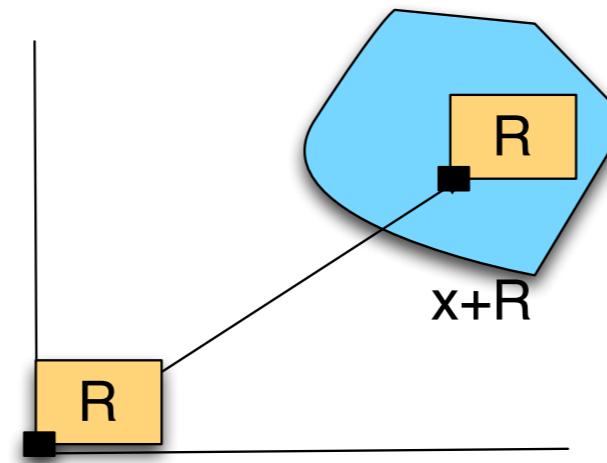
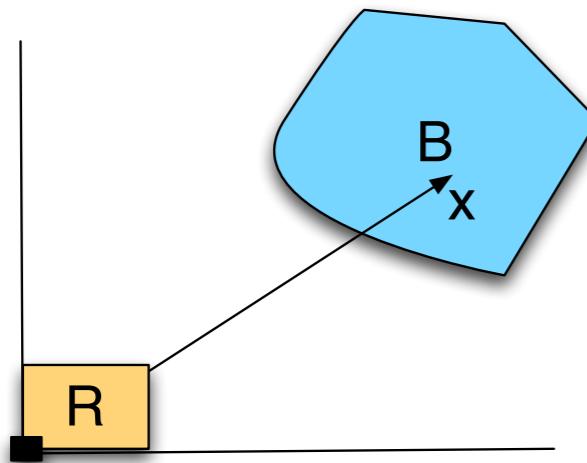
$R$  translated by  $x$



$B \oplus R$

# C-obstacles as Minkowski sums

- Consider a robot  $R$  with the reference in the lower left corner

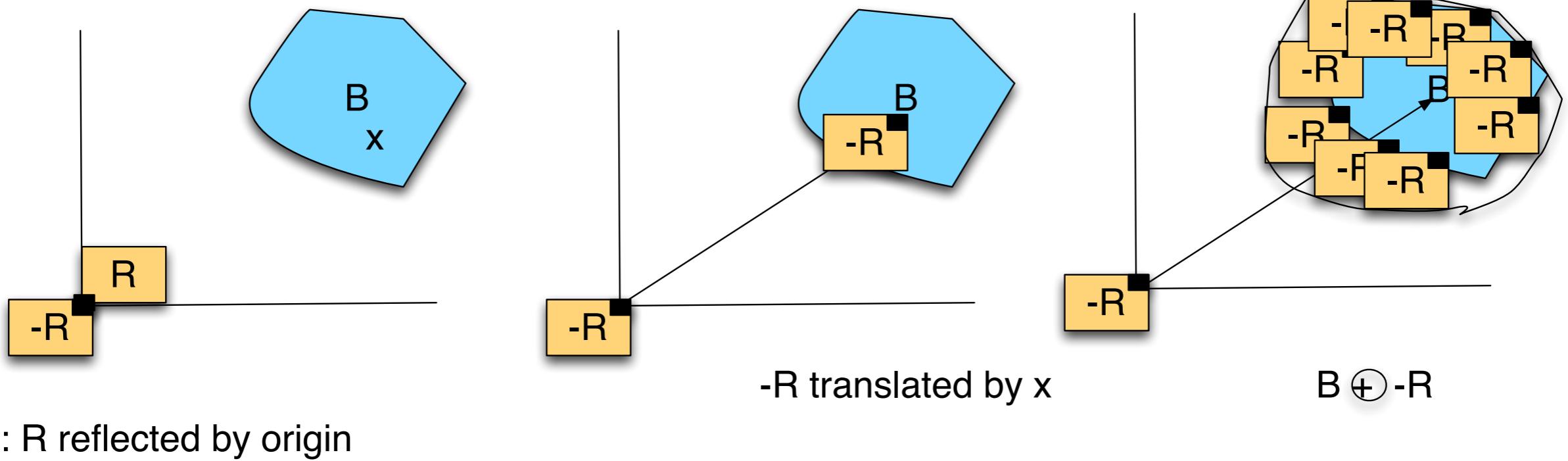


$R$  translated by  $x$

$B \oplus R$

$B \oplus R$  is not quite the C-obstacle of  $B$

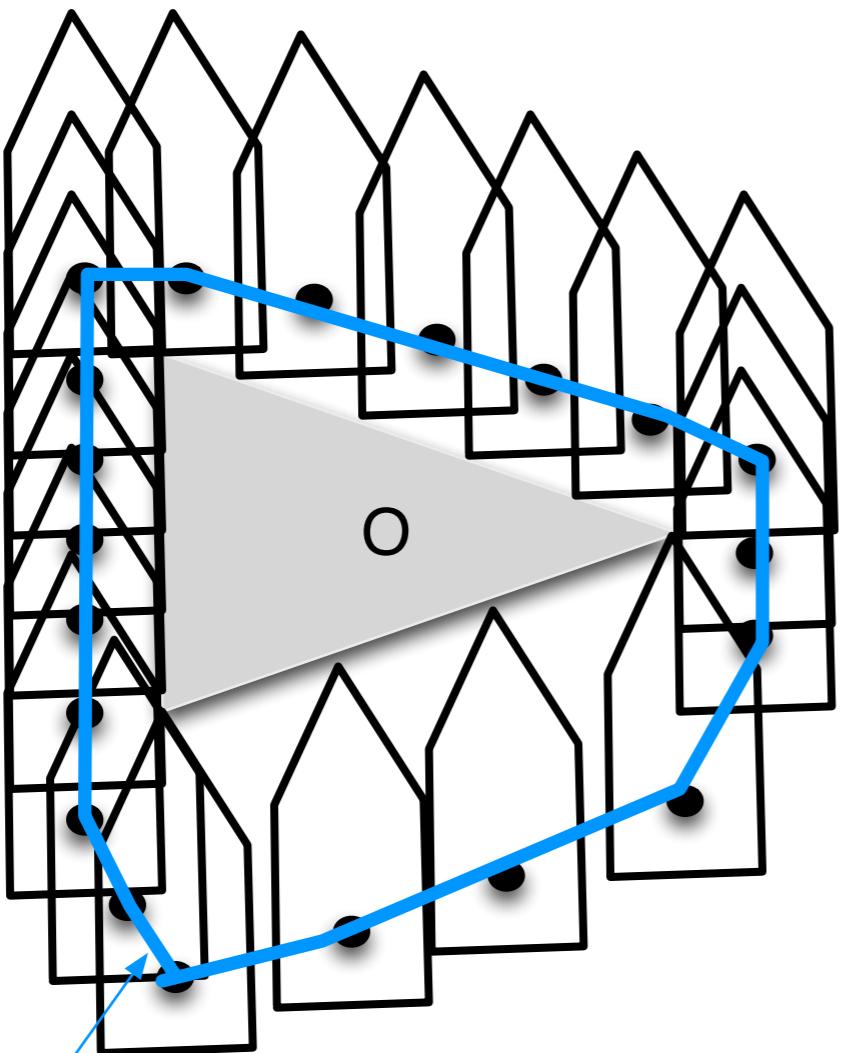
# C-obstacles as Minkowski sums



The C-obstacle of  $B$  is  $B \oplus (-R(0,0))$ .

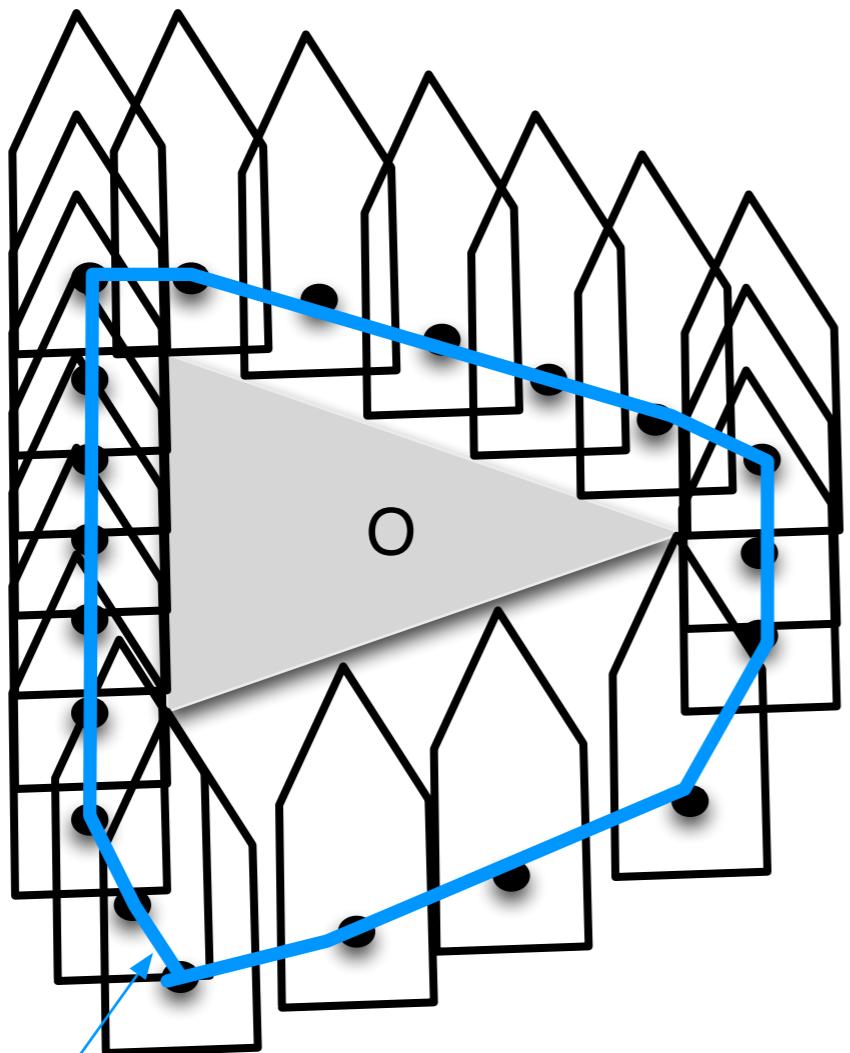
Slide so that R touches the obstacle

Find  $O + (-R)$



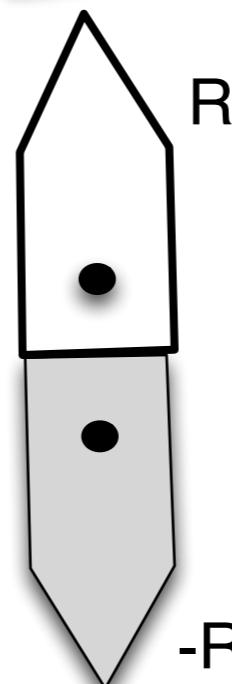
C-obstacle corresponding to O

Slide so that R touches the obstacle

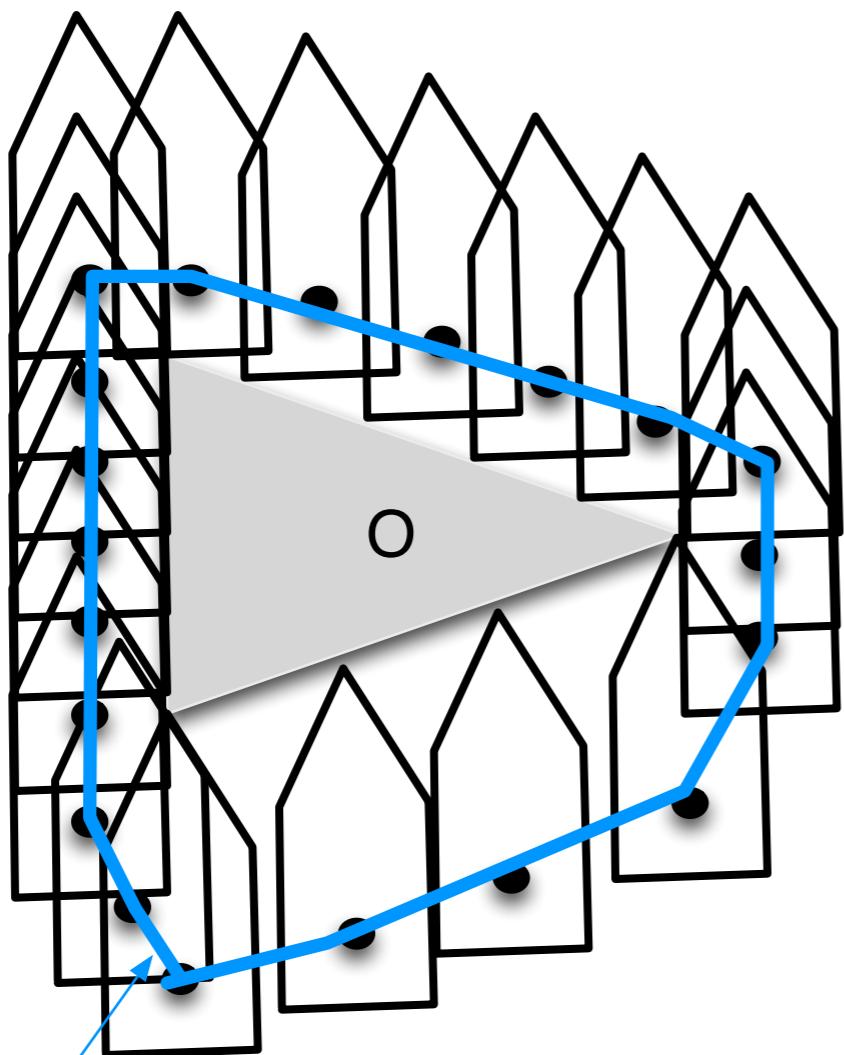


C-obstacle corresponding to O

Slide so that centerpoint of  $-R$  traces the edges of obstacle

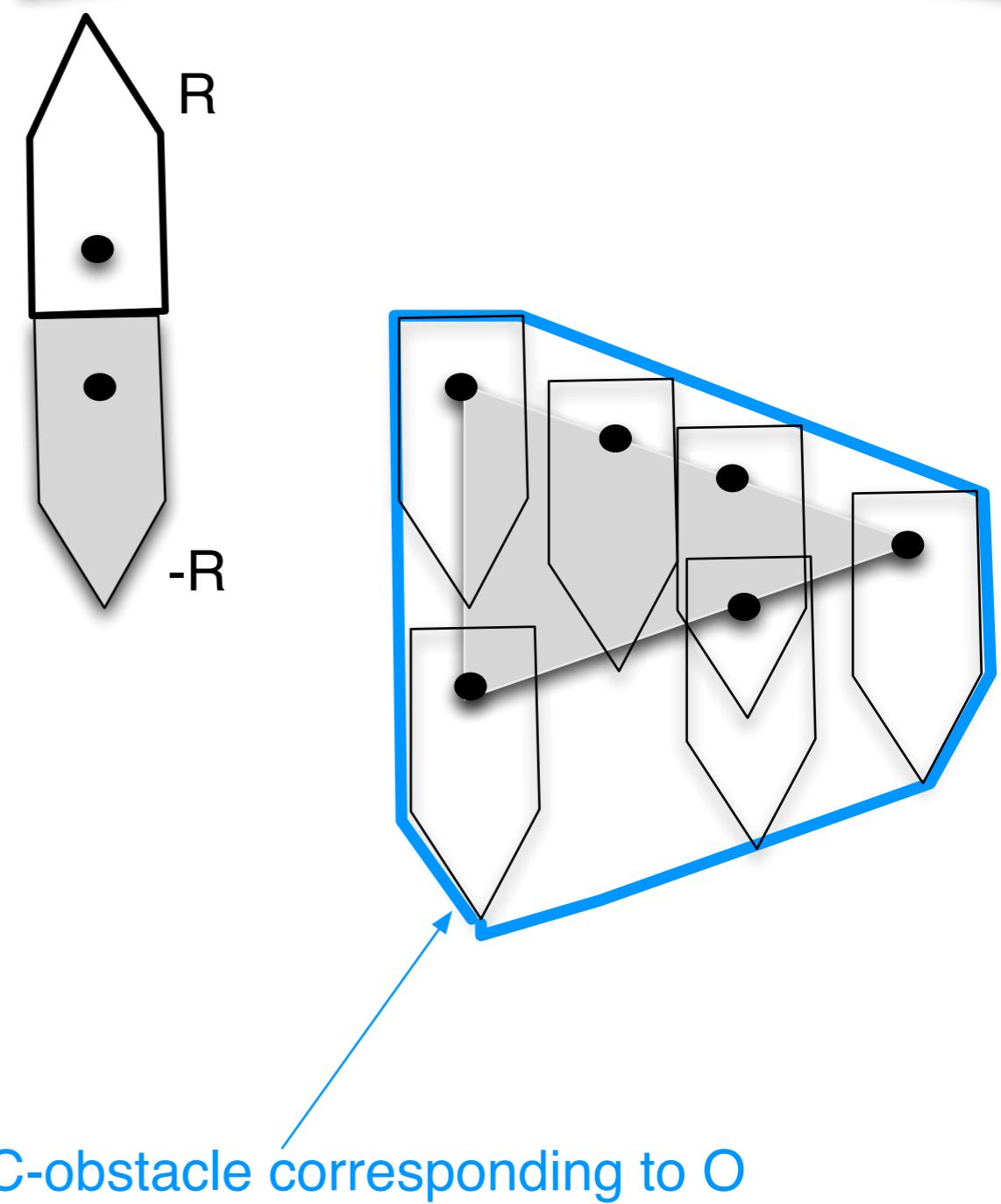


Slide so that R touches the obstacle

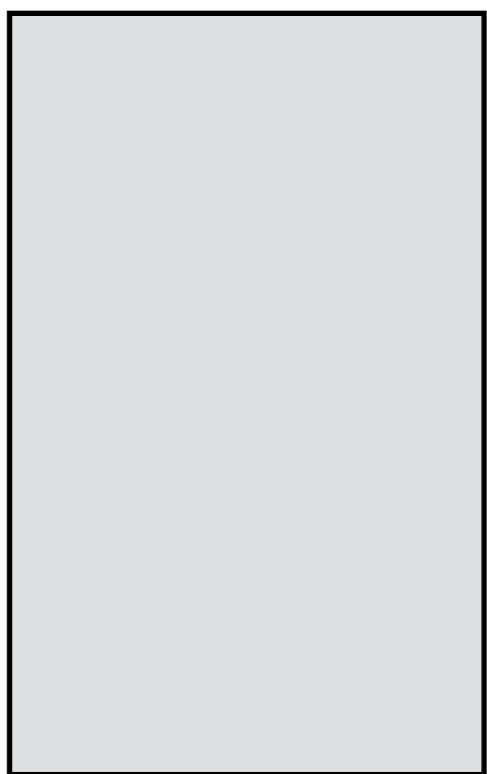
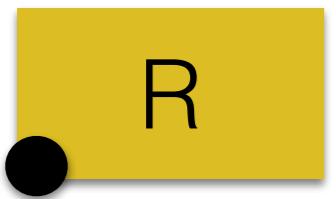


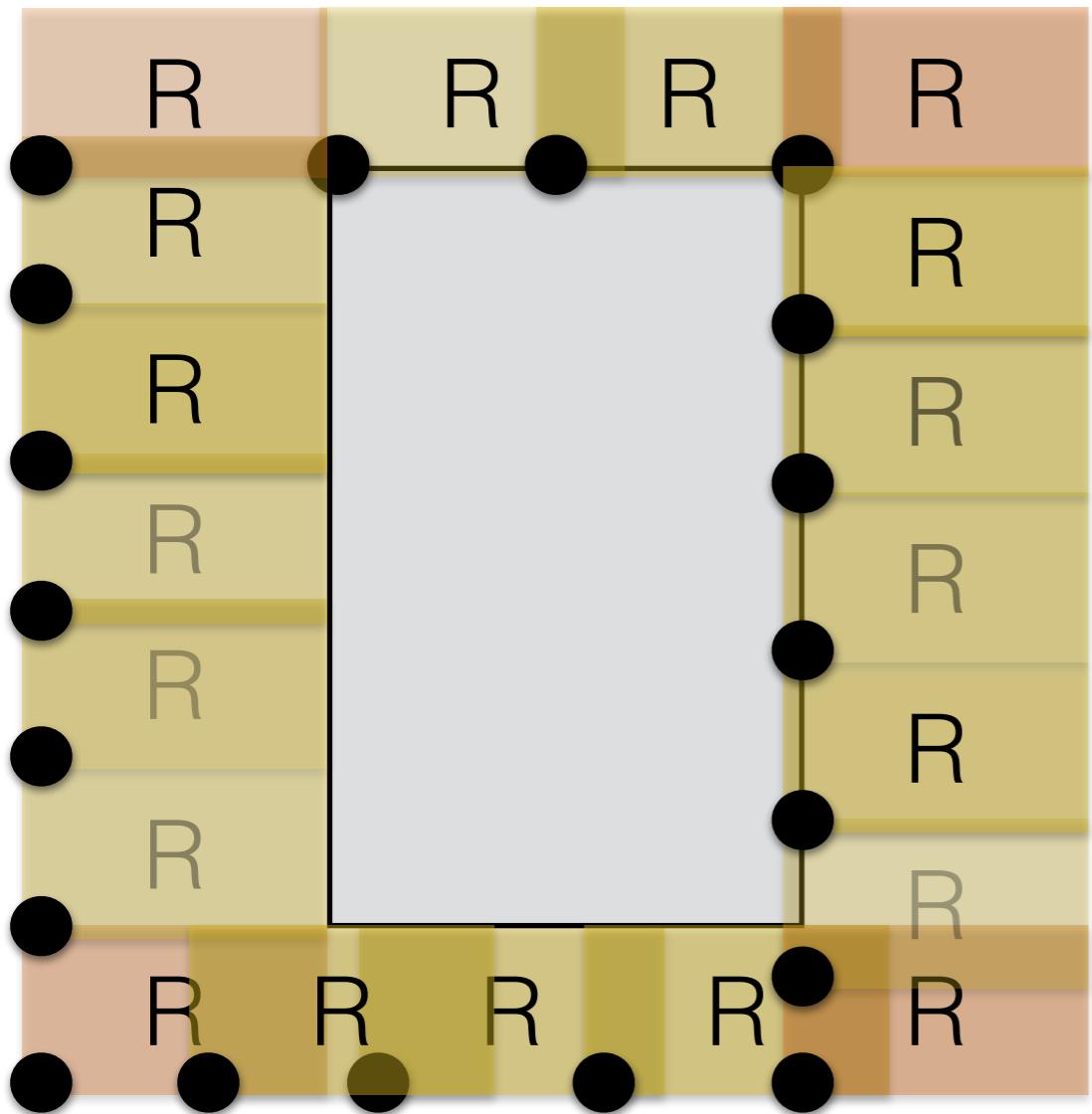
C-obstacle corresponding to O

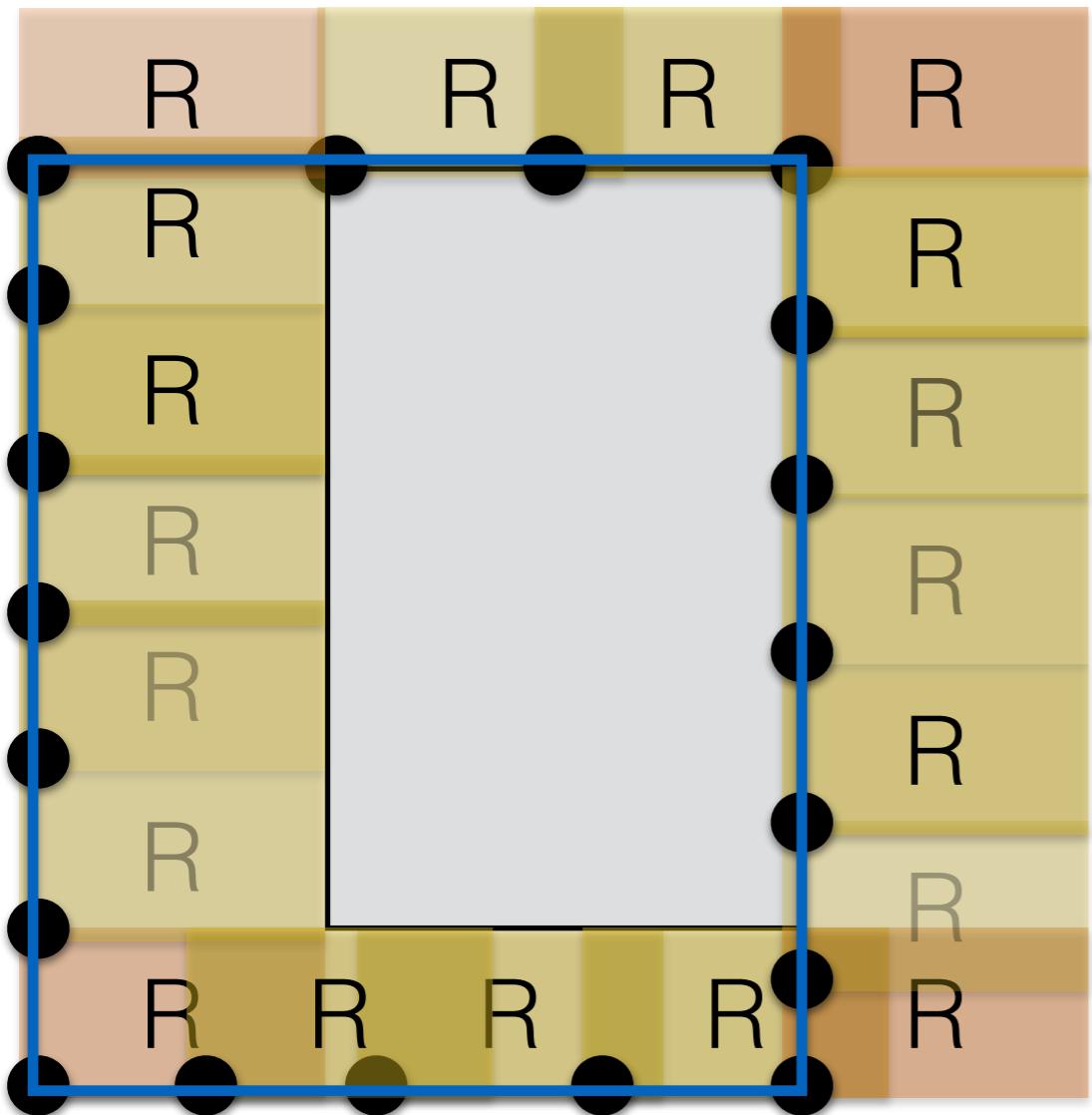
Slide so that centerpoint of  $-R$  traces the edges of obstacle

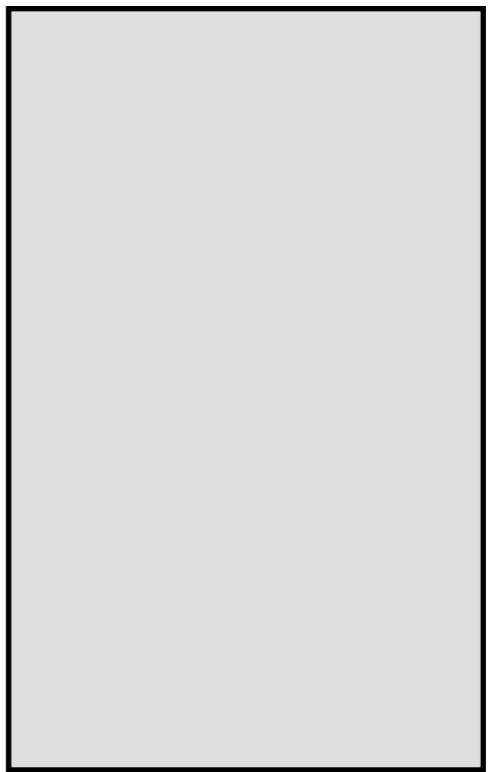
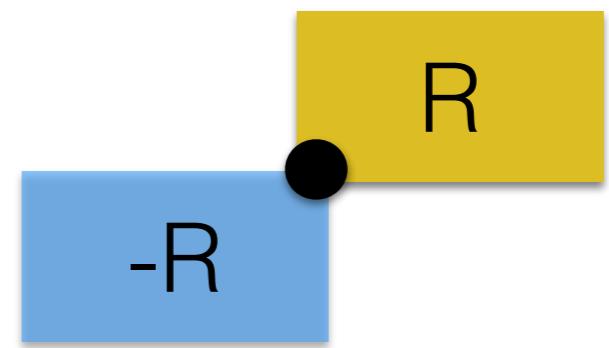


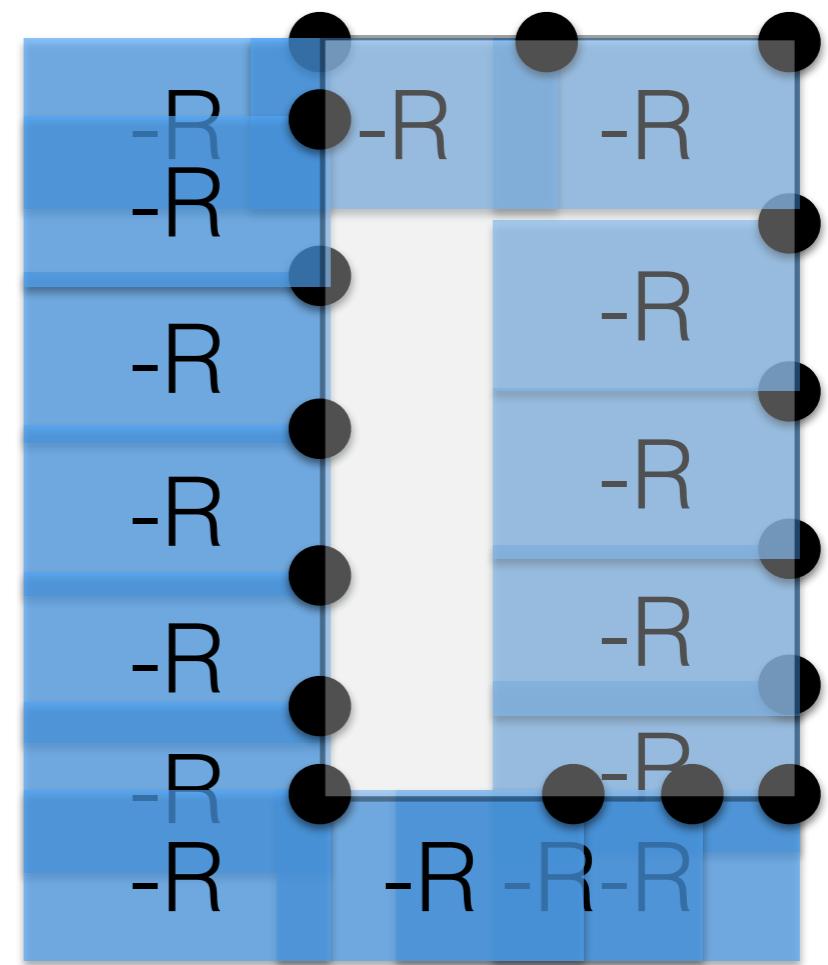
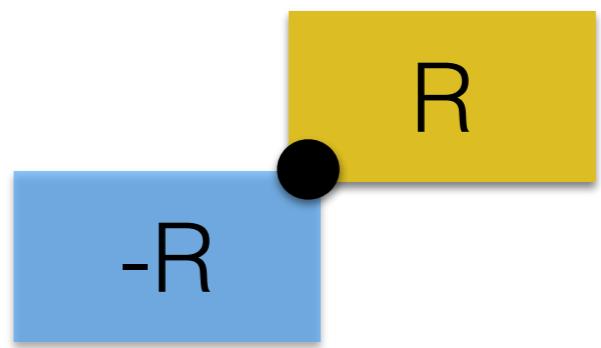
C-obstacle corresponding to O

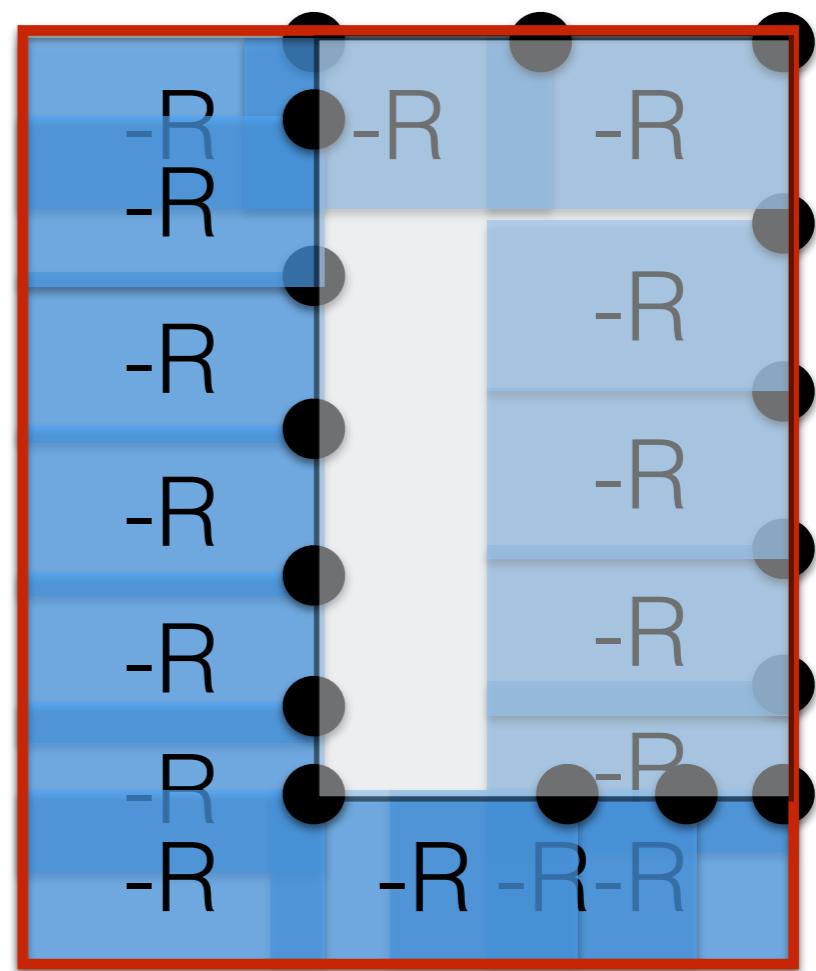
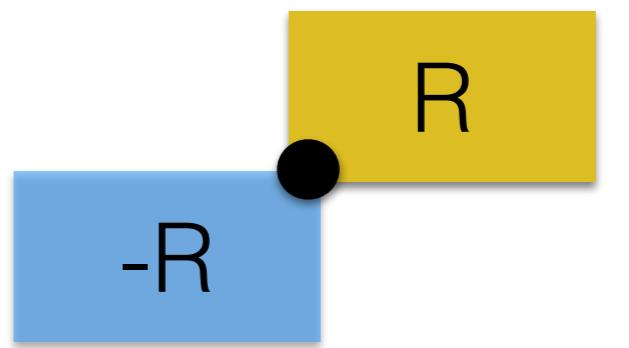


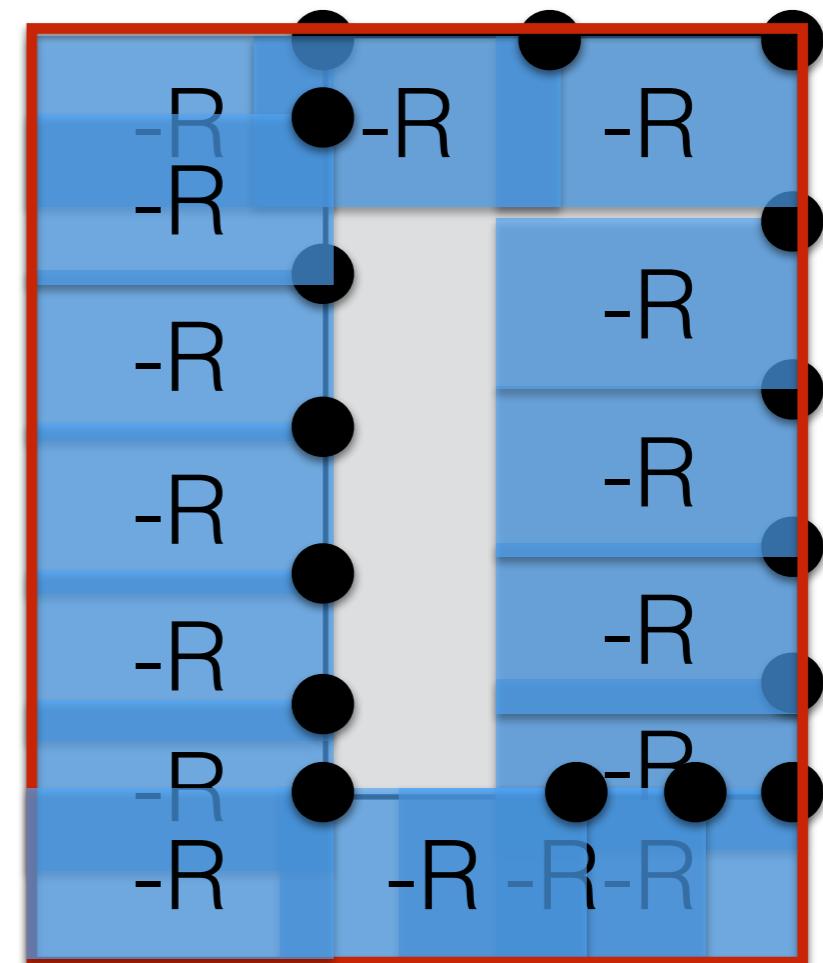
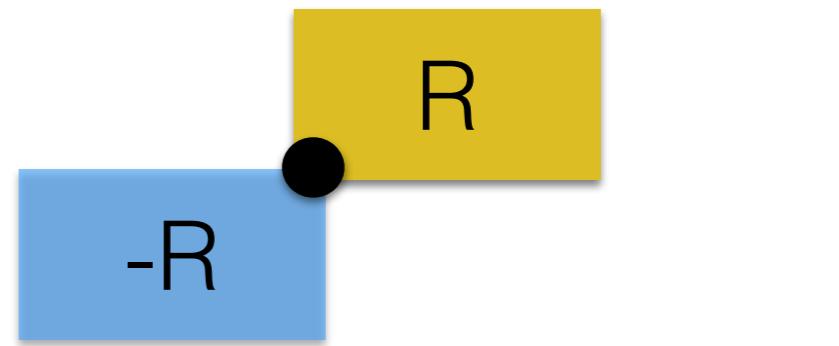
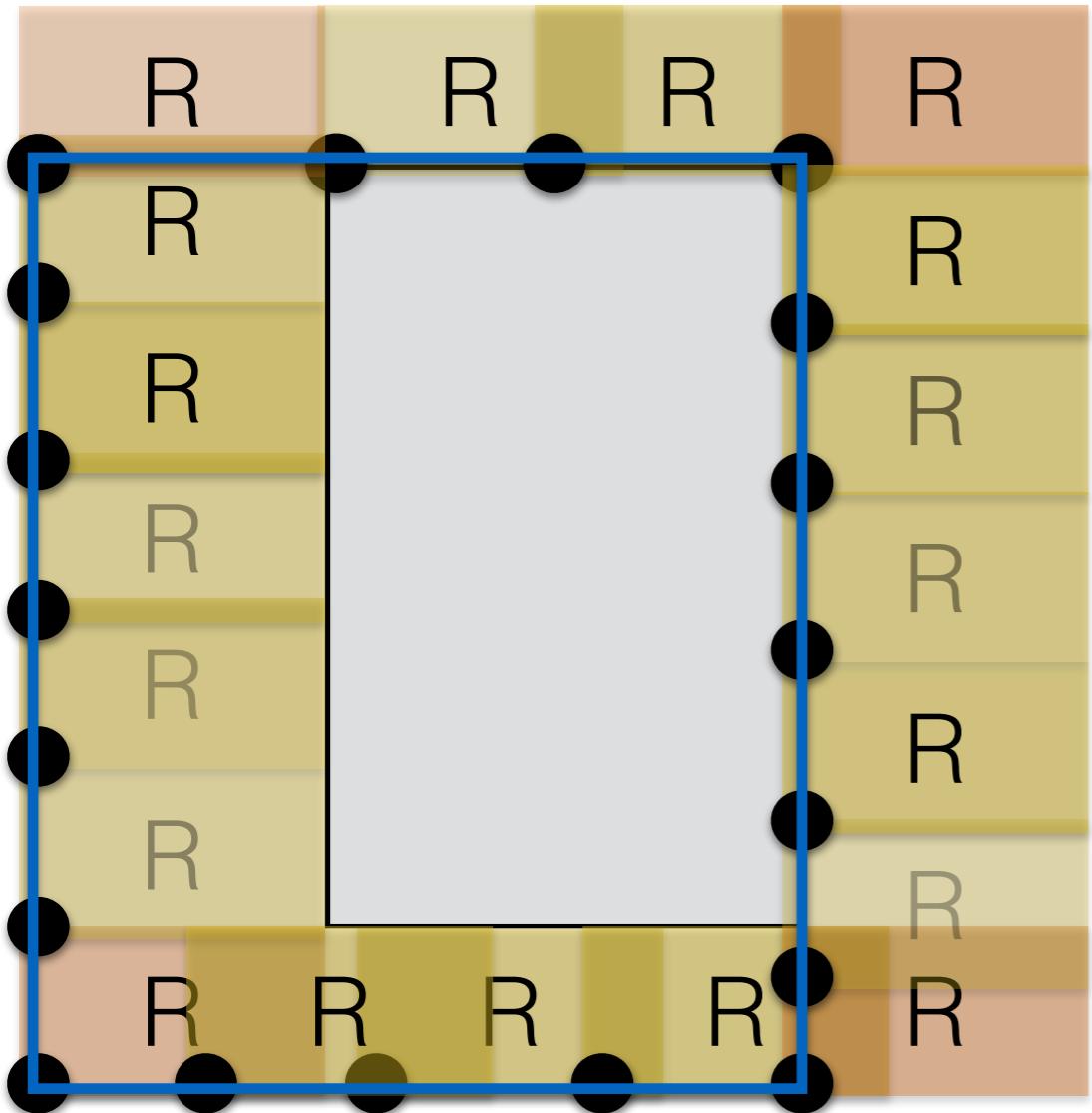


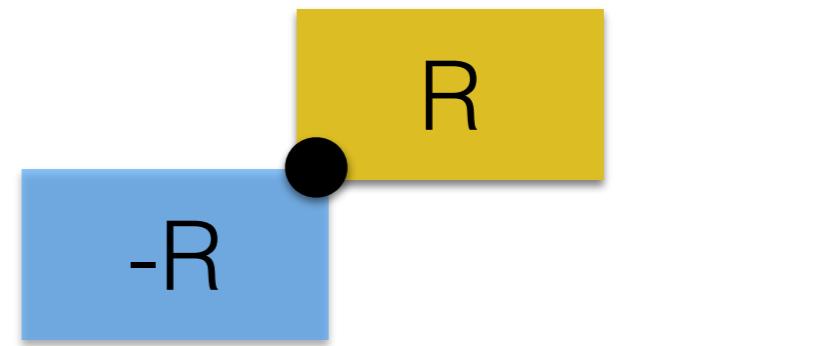
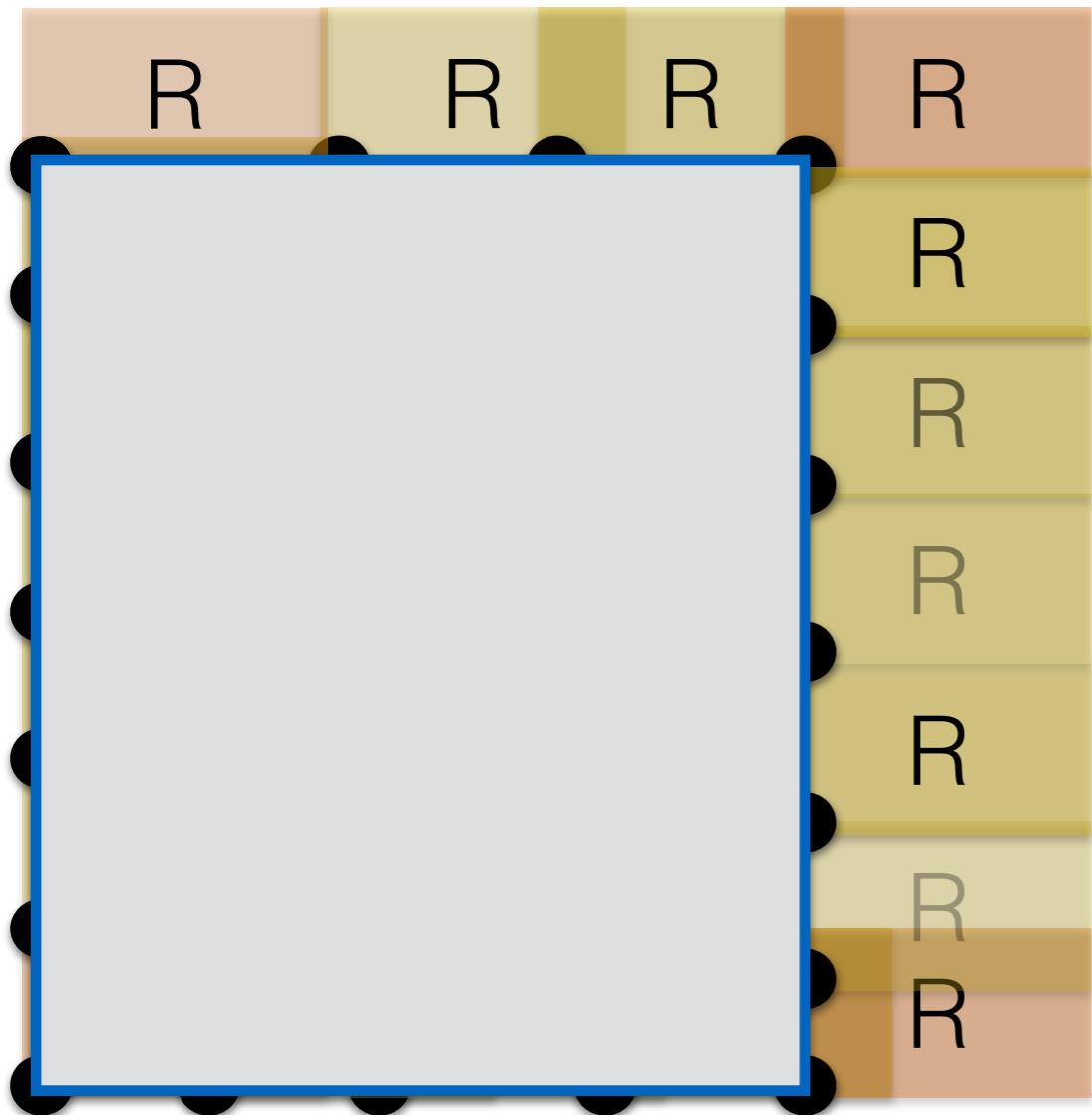


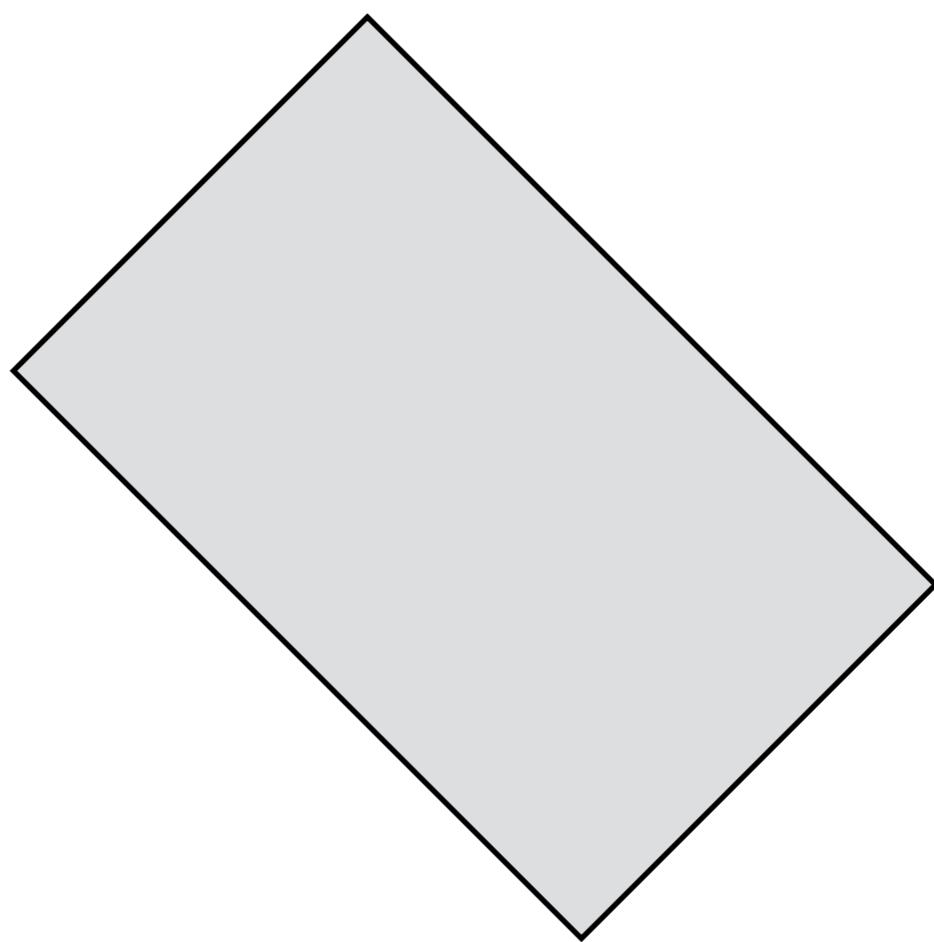
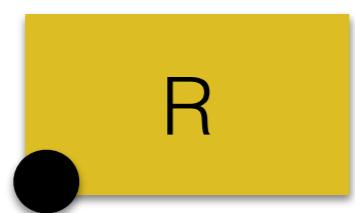


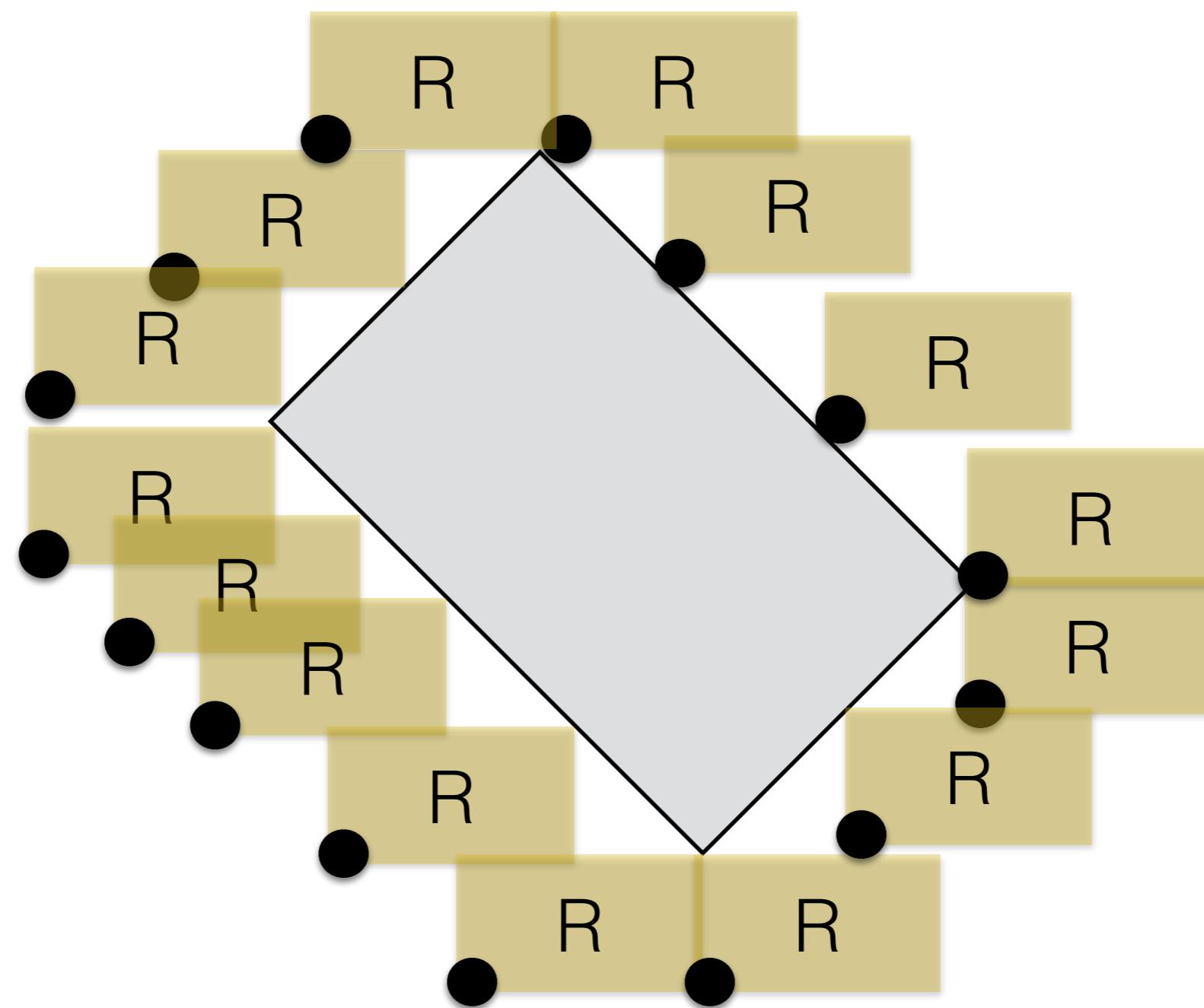


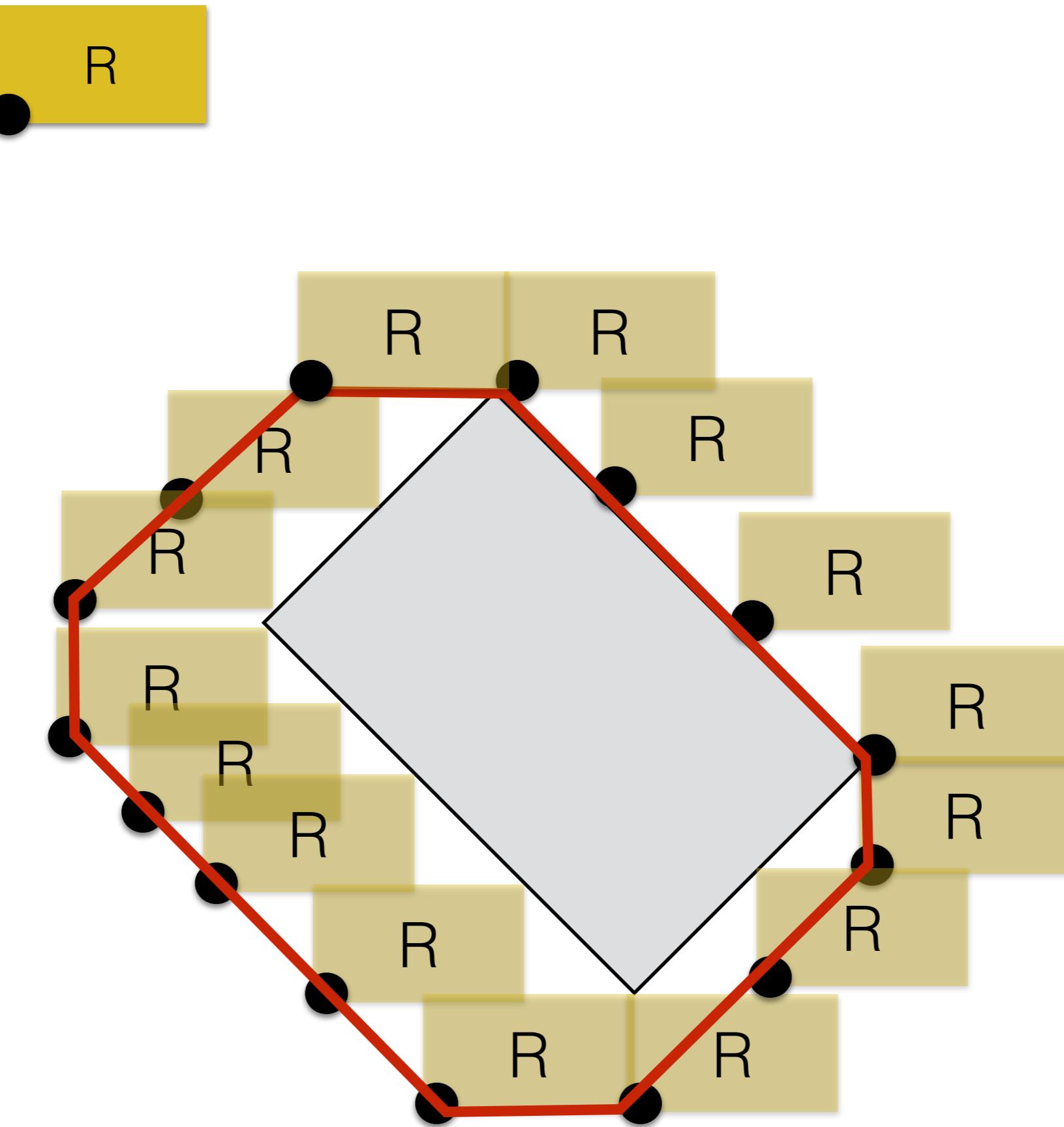






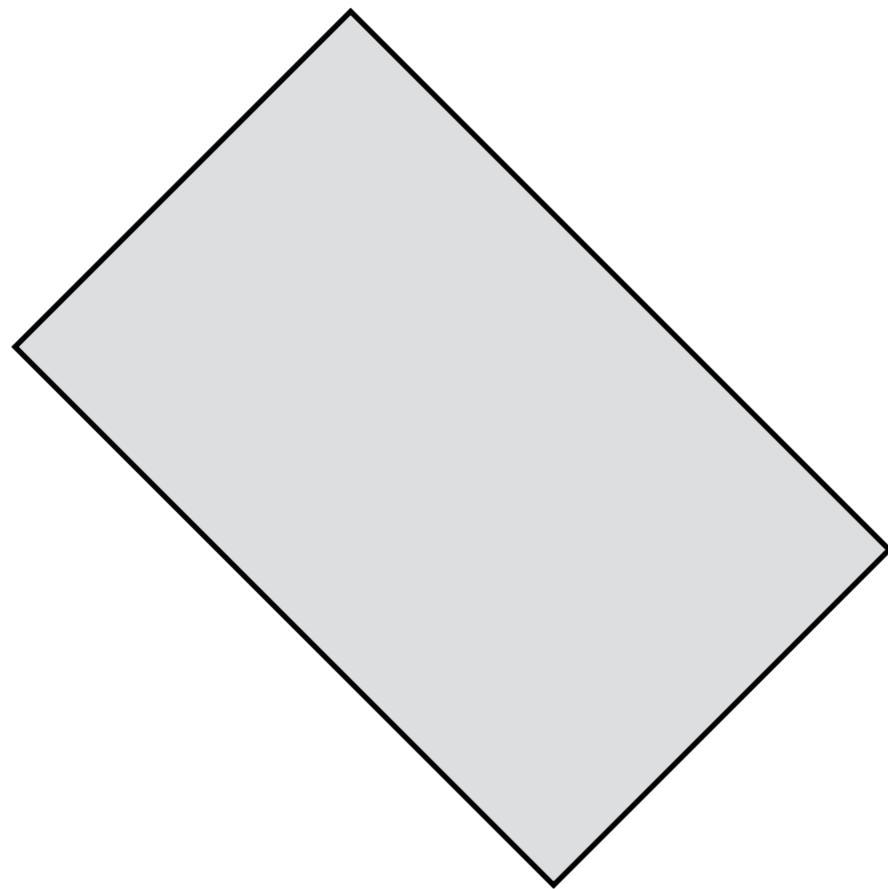


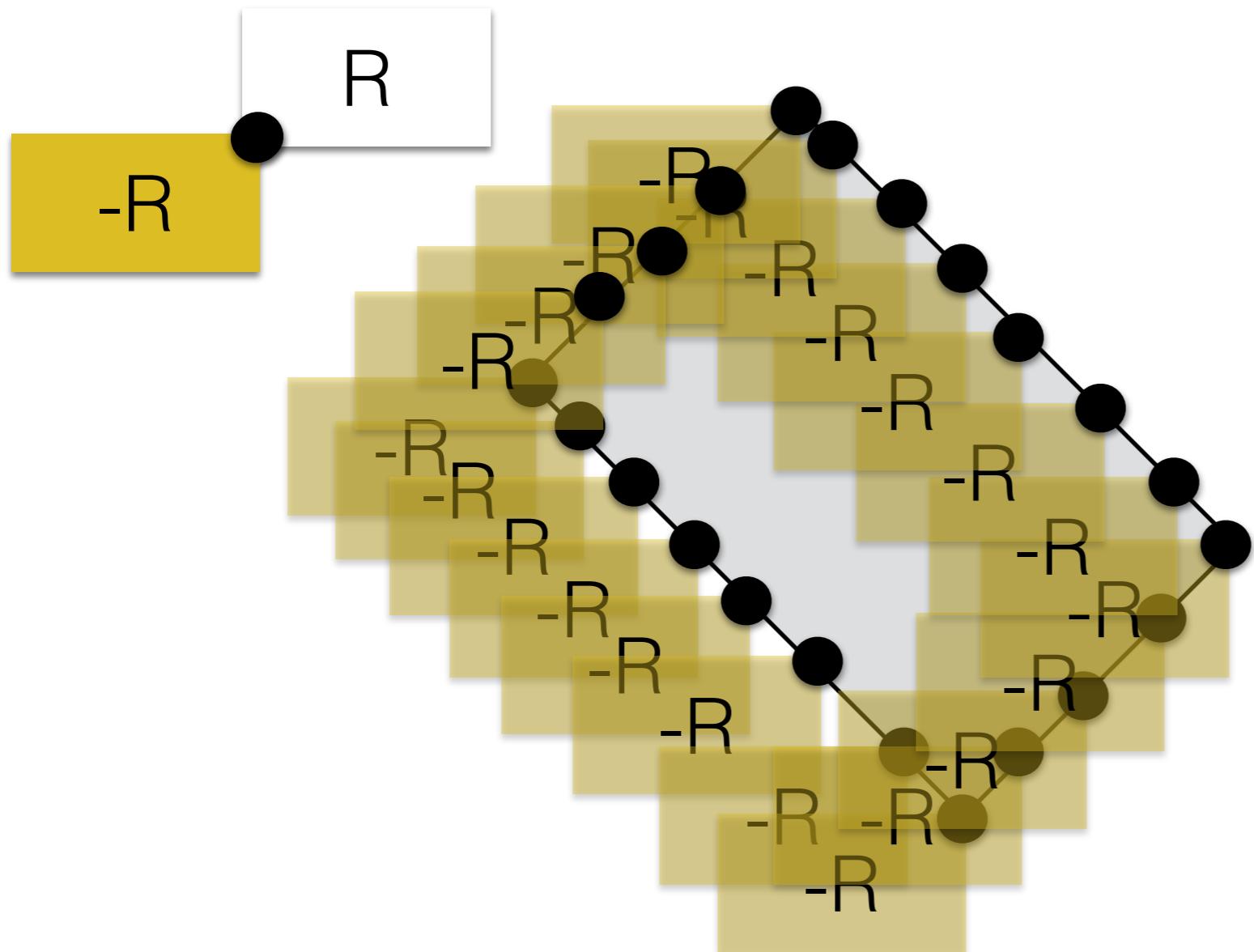


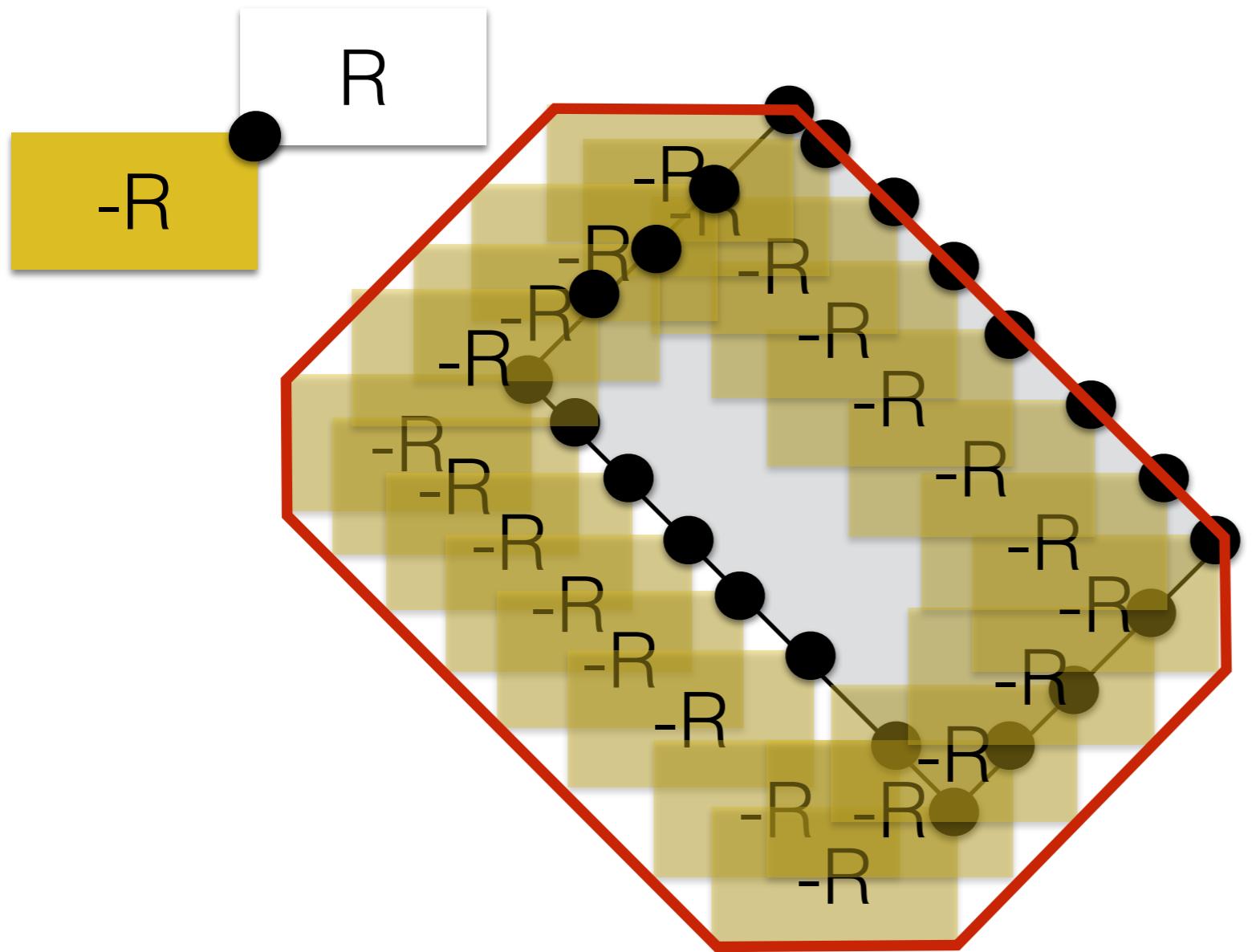


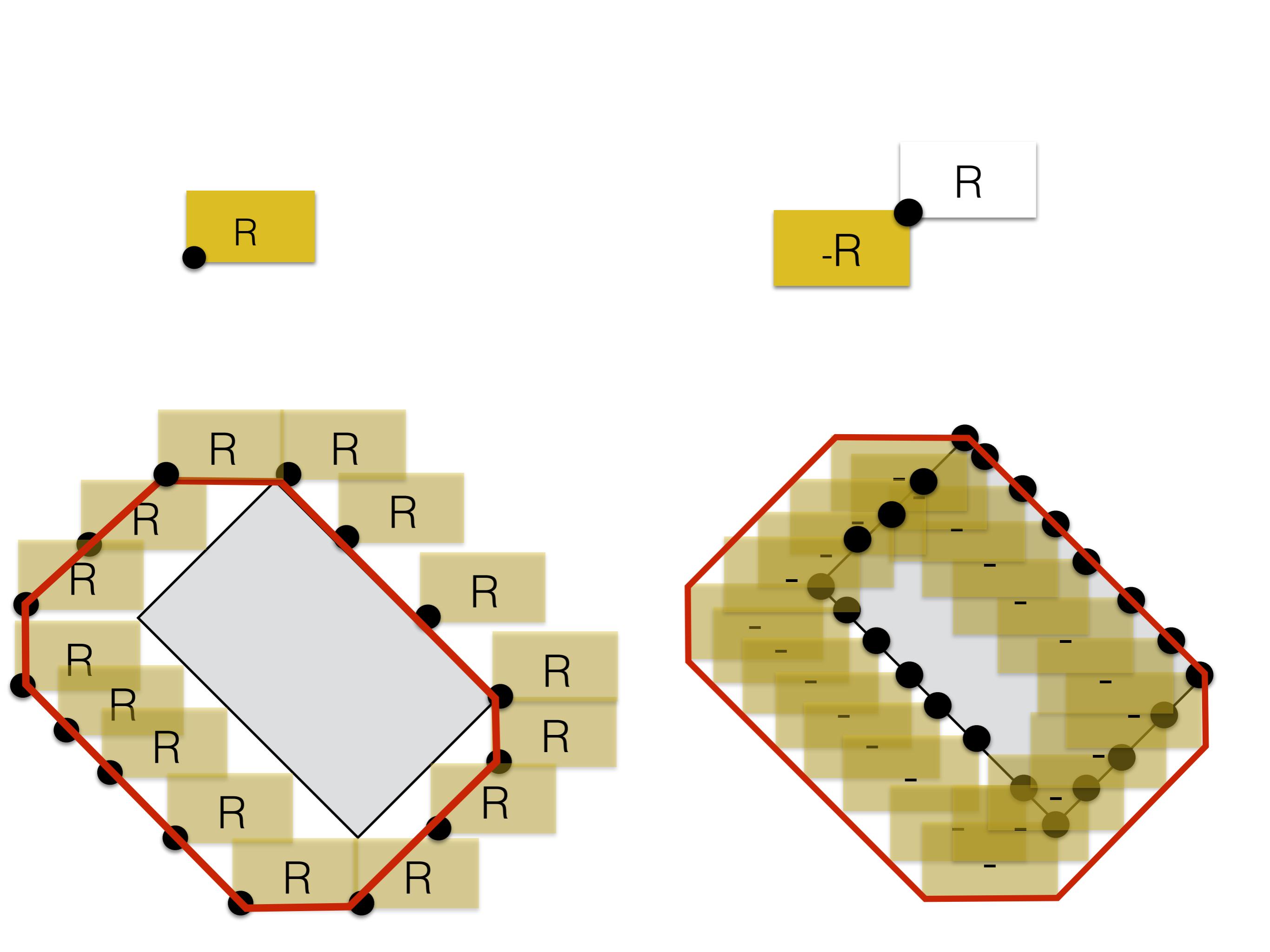
$-R$

$R$

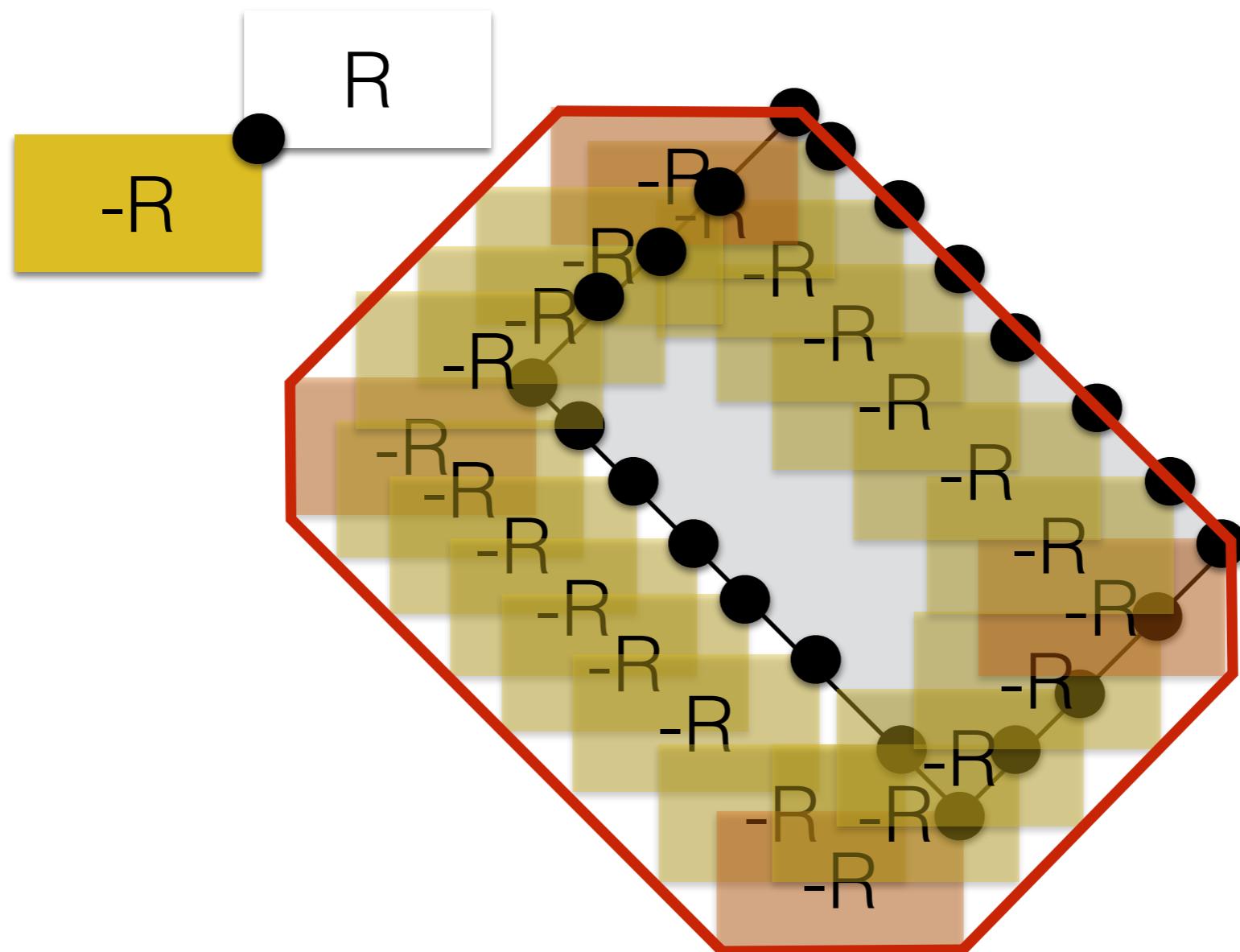




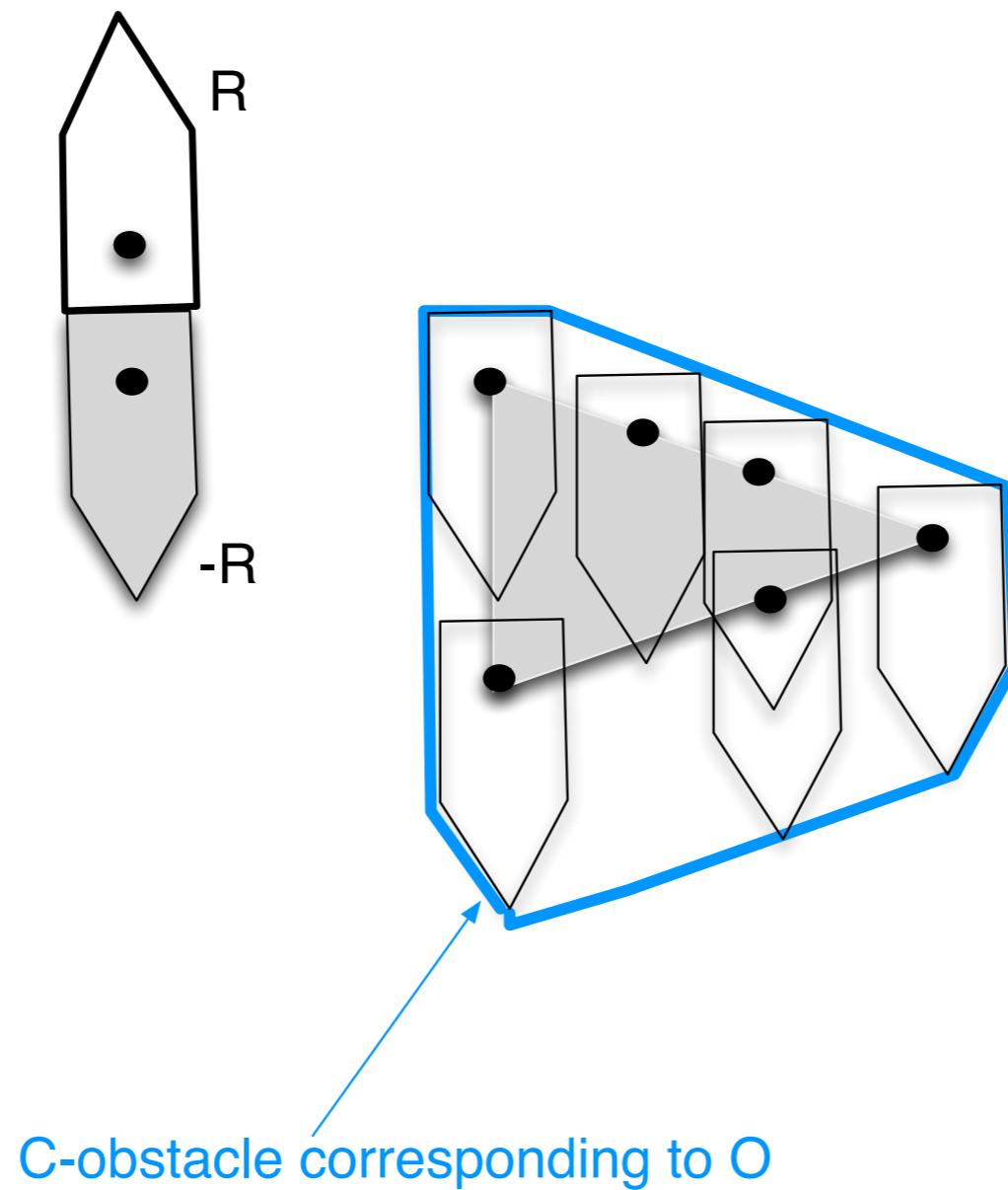




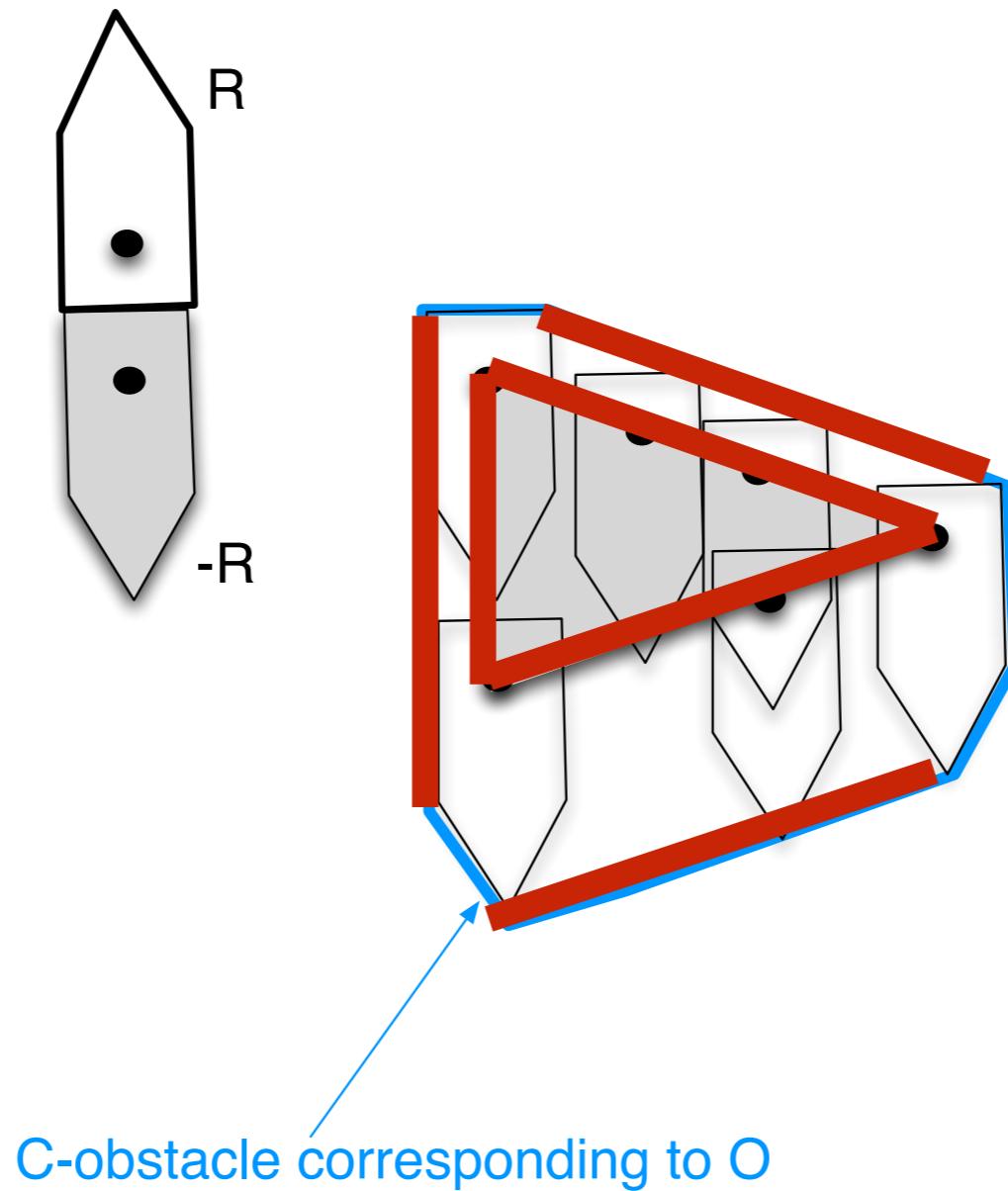
# How do we compute Minkowski sums?



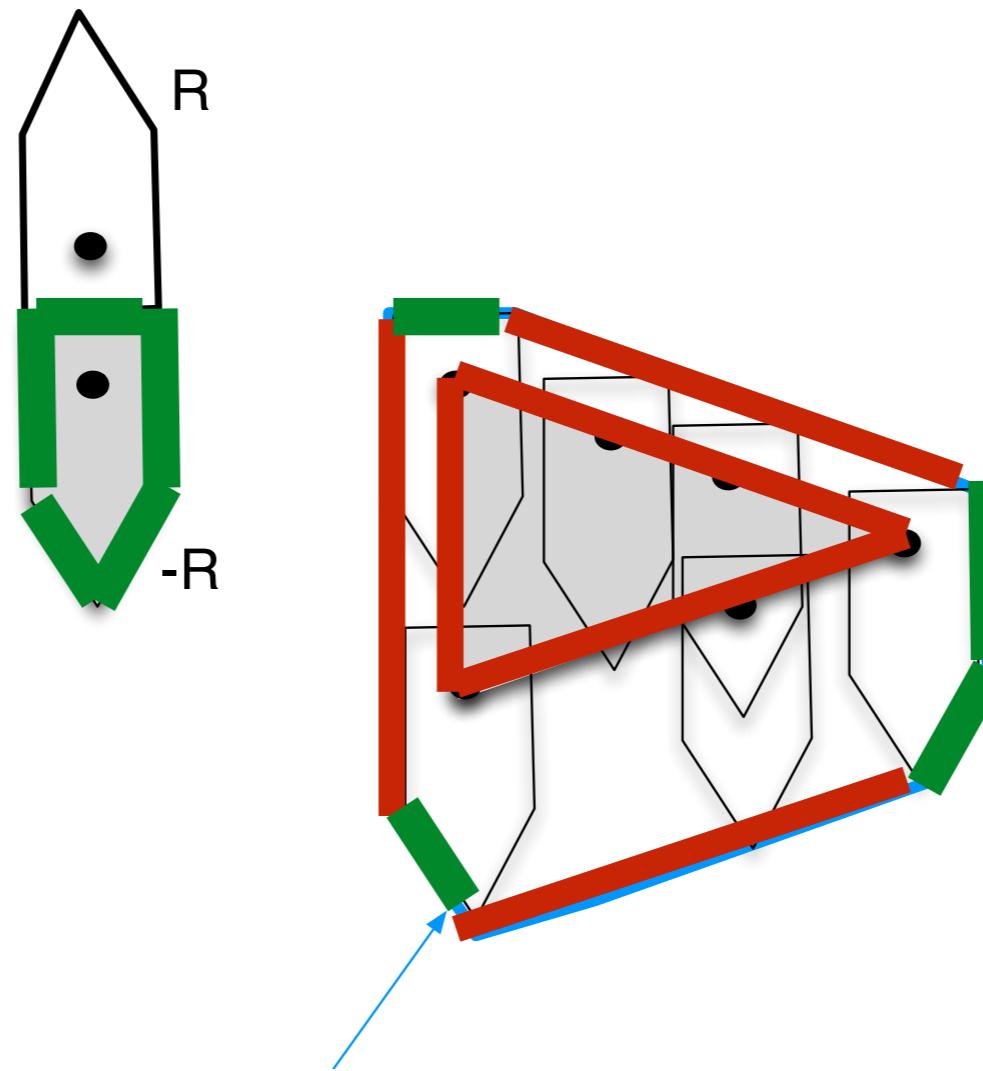
# How to compute Minkowski sums?



## CASE 1: Convex robot with convex polygon

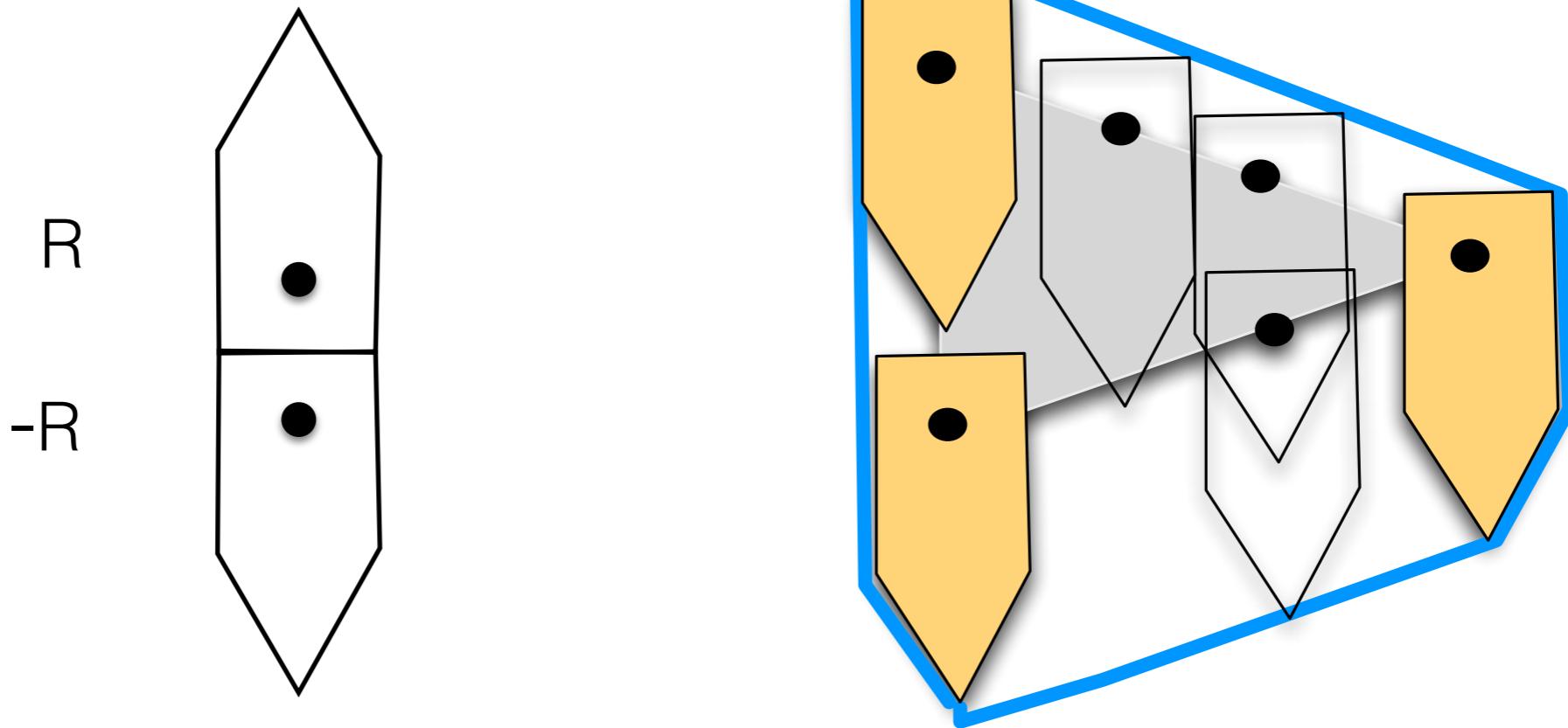


## CASE 1: Convex robot with convex polygon



- Each edge in  $R, O$  will cause an edge in  $R+O$
- $R+O$  has  $O(m+n)$  edges

## CASE 1: Convex robot with convex polygon



- To compute: Place  $-R$  at all vertices of  $O$  and compute convex hull
- Possible to compute in  $O(m+n)$  time by walking along the boundaries of  $R$  and  $O$

# In general...

## 2D

- convex + convex polygons
- The Minkowski sum of two convex polygons with  $n$ , and  $m$  edges respectively, is a convex polygon with  $n+m$  edges and can be computed in  $O(n+m)$  time.
- convex + non-convex polygons
  - Triangulate and compute Minkowski sums for each pair [convex polygon, triangle], and take their union
  - Size of Minkowski sum:  $O(m+3)$  for each triangle  $\Rightarrow O(mn)$
- non-convex + non-convex polygons:
  - size of Minkowski sum:  $O(n^2m^2)$

## 3D

- it gets worse . . .

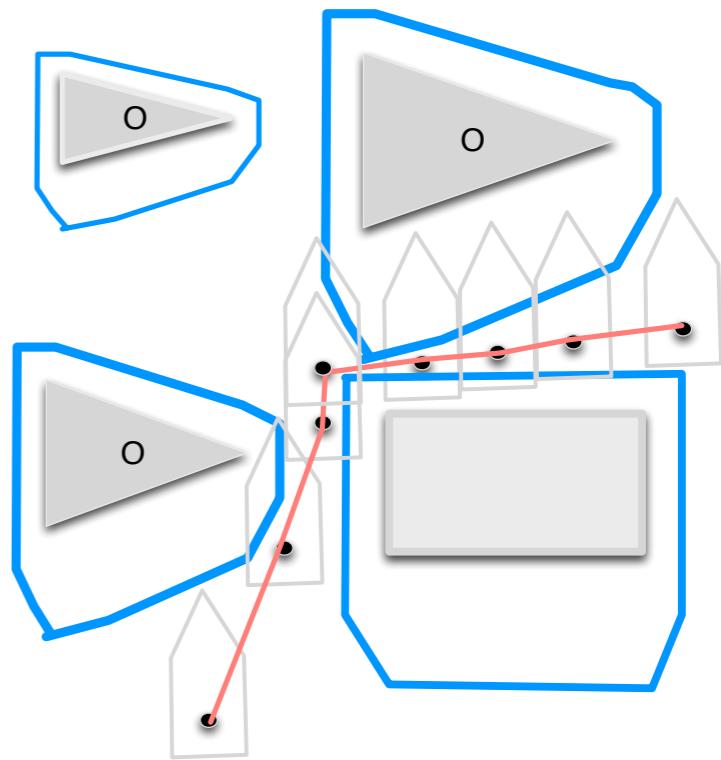
Back to planning

# Polygonal robot translating in 2D

Algorithm

Complete, non optimal.

- For each obstacle  $O$ , compute the corresponding C-obstacle
  - Compute the union of C-obstacles
  - Compute its complement. That's the free C-space
- //problem is reduced to a point robot
- //moving in free C-space
- Compute a trapezoidal map of free C-space
  - Use it to compute a roadmap of free space



How fast can this be done?

# Polygonal robot translating in 2D

Algorithm

Complete, non optimal.

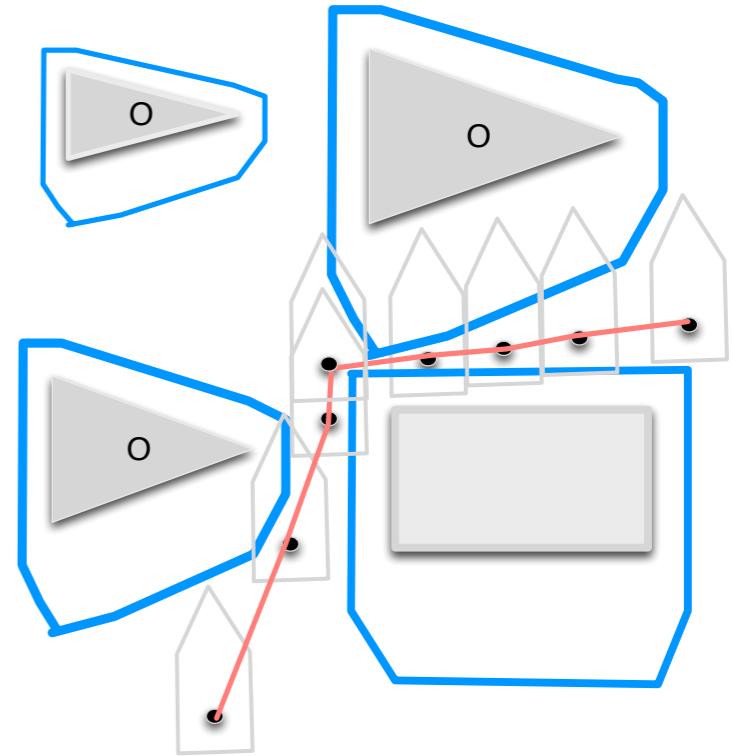
- For each obstacle  $O$ , compute the corresponding C-obstacle
- Compute the union of C-obstacles
- Compute its complement. That's the free C-space

//problem is reduced to a point robot

//moving in free C-space

- Compute a trapezoidal map of free C-space
- Use it to compute a roadmap of free space

How fast can this be done?



For a **convex** robot of  **$O(1)$  size**

- Free C-space can be computed in  $O(n \lg^2 n)$  time.
- With  $O(n \lg^2 n)$  time pre-processing, a collision-free path can be found for any start and end in  $O(n)$  time.

So far we've considered only translation



Next: Translation + Rotation

# Polygonal robot in 2D with rotations

- Physical space is 2D
- A placement is specified by 3 parameters:  $R(x,y, \theta)$  ==> C-space is 3D.



# Polygonal robot in 2D with rotations

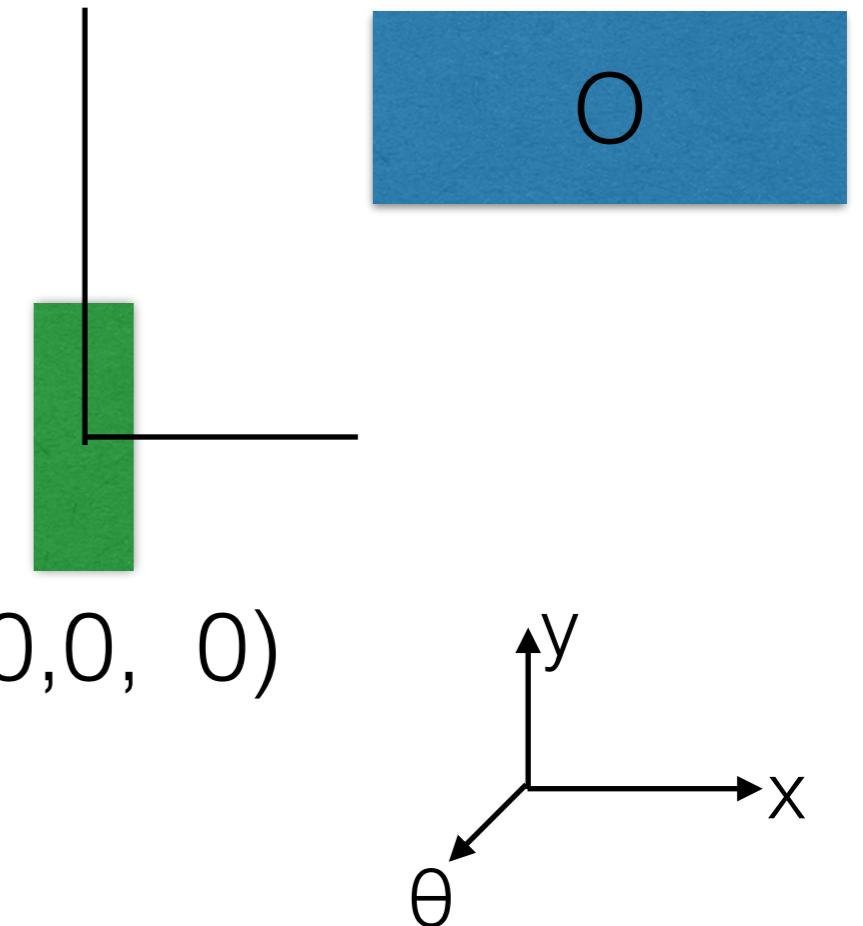
- We'd like to extend the same approach:

Reduce to point robot moving among C-obstacles in C-space.

- Compute C-obstacles
- Compute free space as complement of union of C-obstacles
- Decompose free space into simple cells
- Construct a graph(roadmap)
- BFS on roadmap

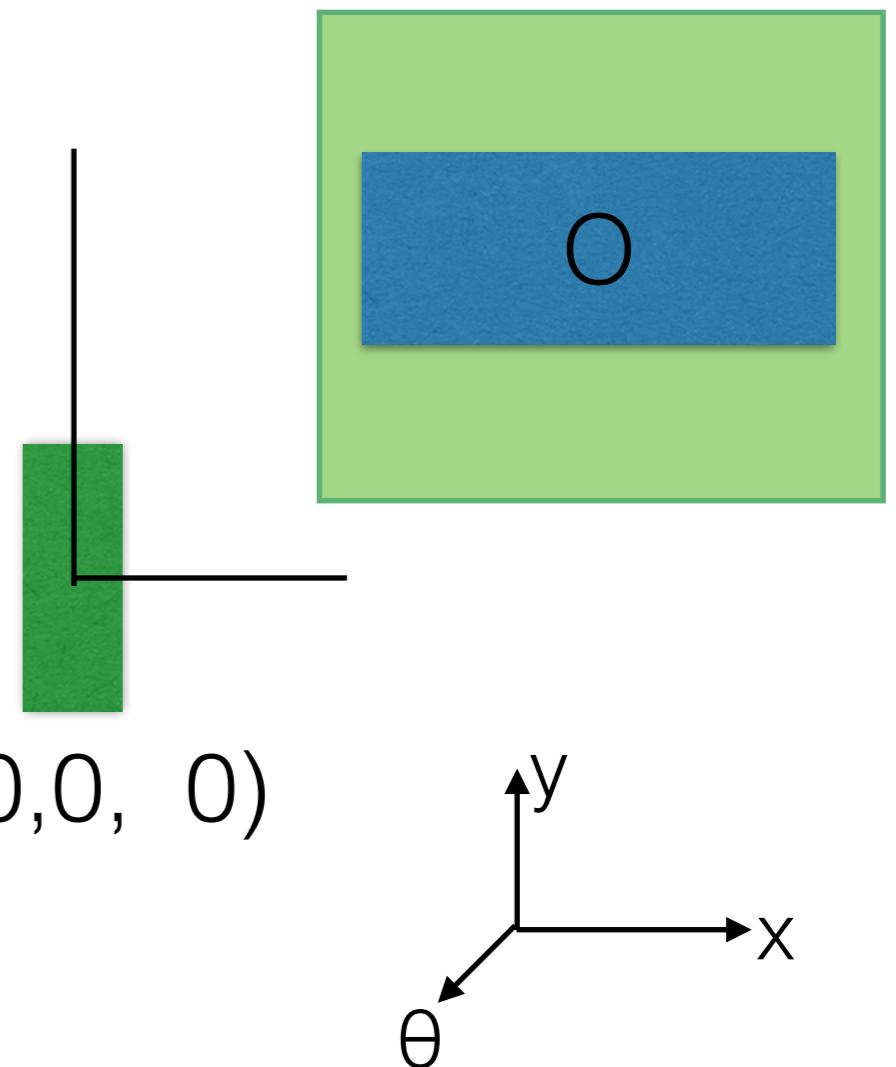
# Polygonal robot in 2D with rotations

- What does a C-obstacle look like when rotations are allowed?



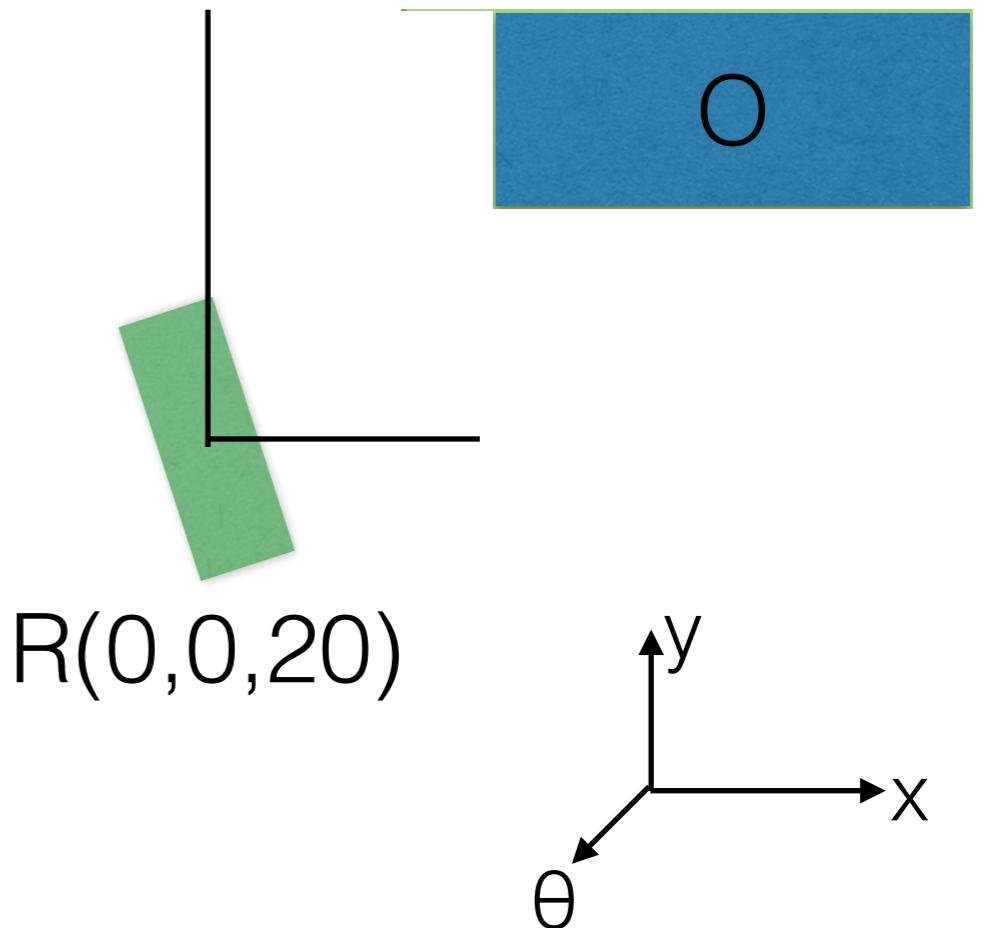
# Polygonal robot in 2D with rotations

- What does a C-obstacle look like when rotations are allowed?



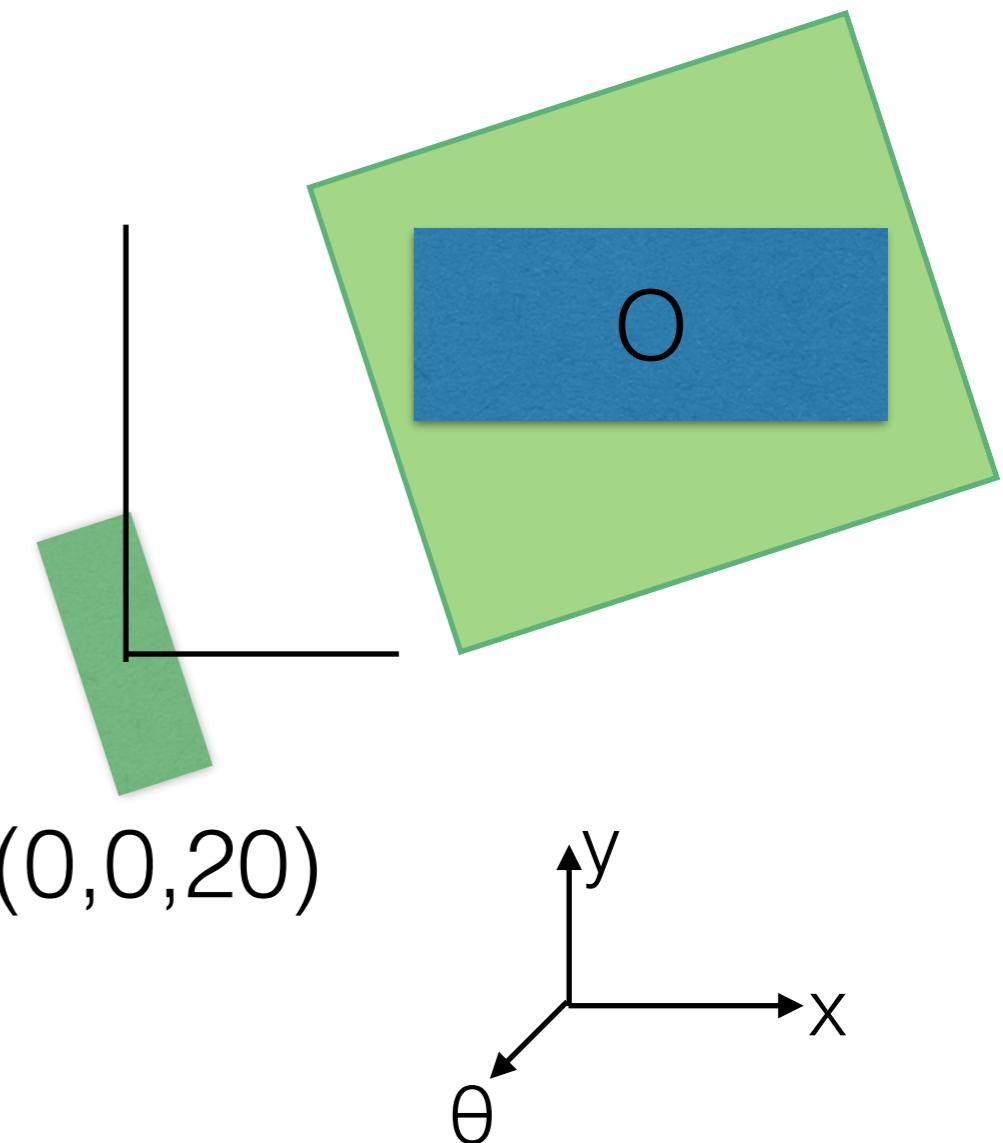
# Polygonal robot in 2D with rotations

- What does a C-obstacle look like when rotations are allowed?



# Polygonal robot in 2D with rotations

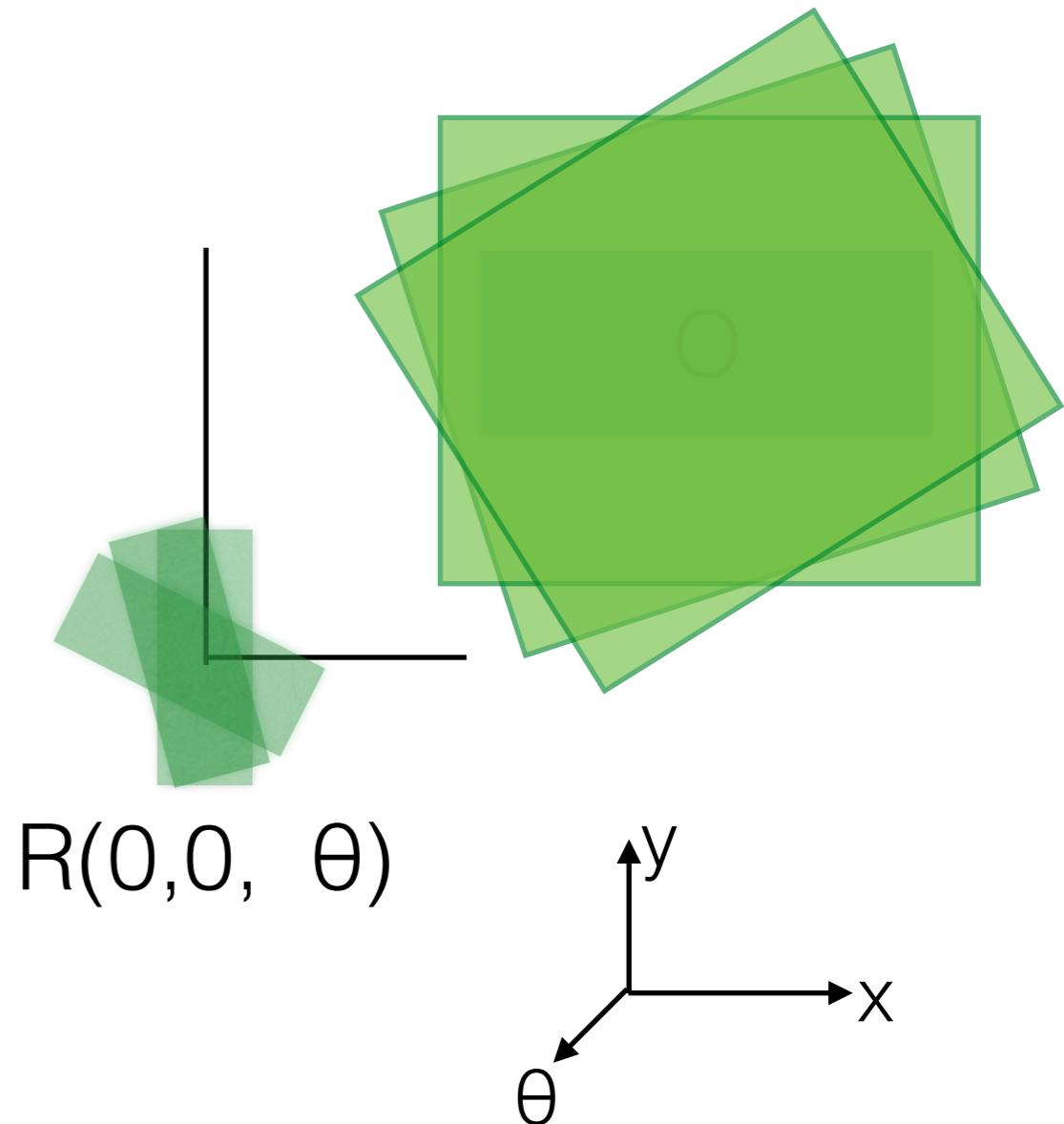
- What does a C-obstacle look like when rotations are allowed?



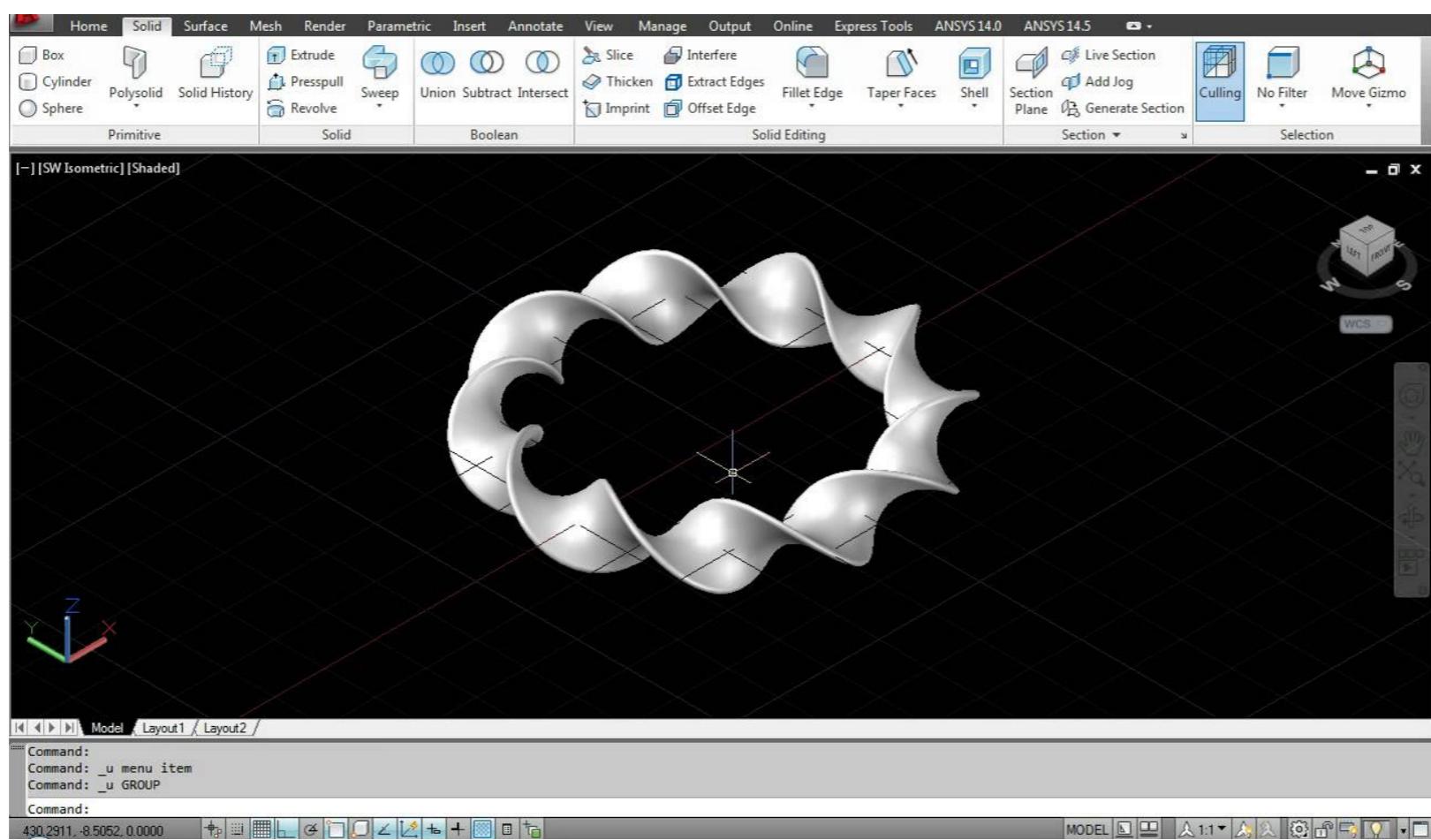
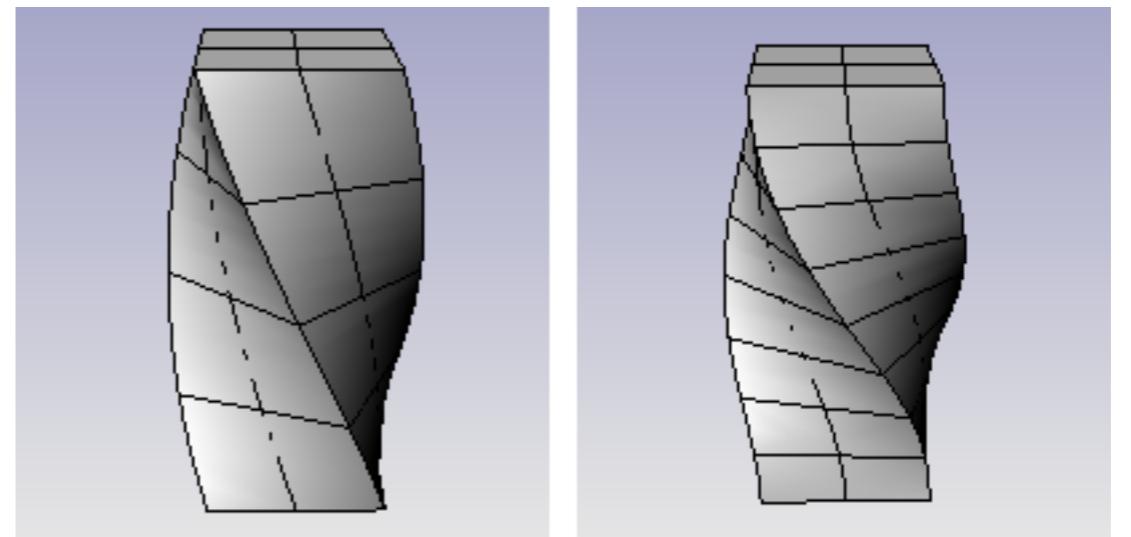
# Polygonal robot in 2D with rotations

- Imagine moving a vertical plane through C-space. Each position of the plane will correspond to a fixed theta.
- Each cross-section of a C-obstacle is a Minkowski sum  $O \oplus R(0,0,\theta)$
- => twisted pillar

**A C-obstacle is a 3D shape.**



the closest i could find ..



# Polygonal robot in 2D with rotations

What's known:

- C-space is 3D
- Boundary of free space is curved, not polygonal.
- Combinatorial complexity of free space is  $O(n^2)$  for convex,  $O(n^3)$  for non-convex robot

# Polygonal robot in 2D with rotations

What's known:

- C-space is 3D
- Boundary of free space is curved, not polygonal.
- Combinatorial complexity of free space is  $O(n^2)$  for convex,  $O(n^3)$  for non-convex robot
- Extend same approach:
  1. Compute C-obstacles and C-free
  2. Compute a decomposition of free space into simple cells
  3. Construct a roadmap
  4. BFS on roadmap

← space is 3D



Difficult to construct a good cell decomposition for curved 3D space

# Polygonal robot in 2D with rotations

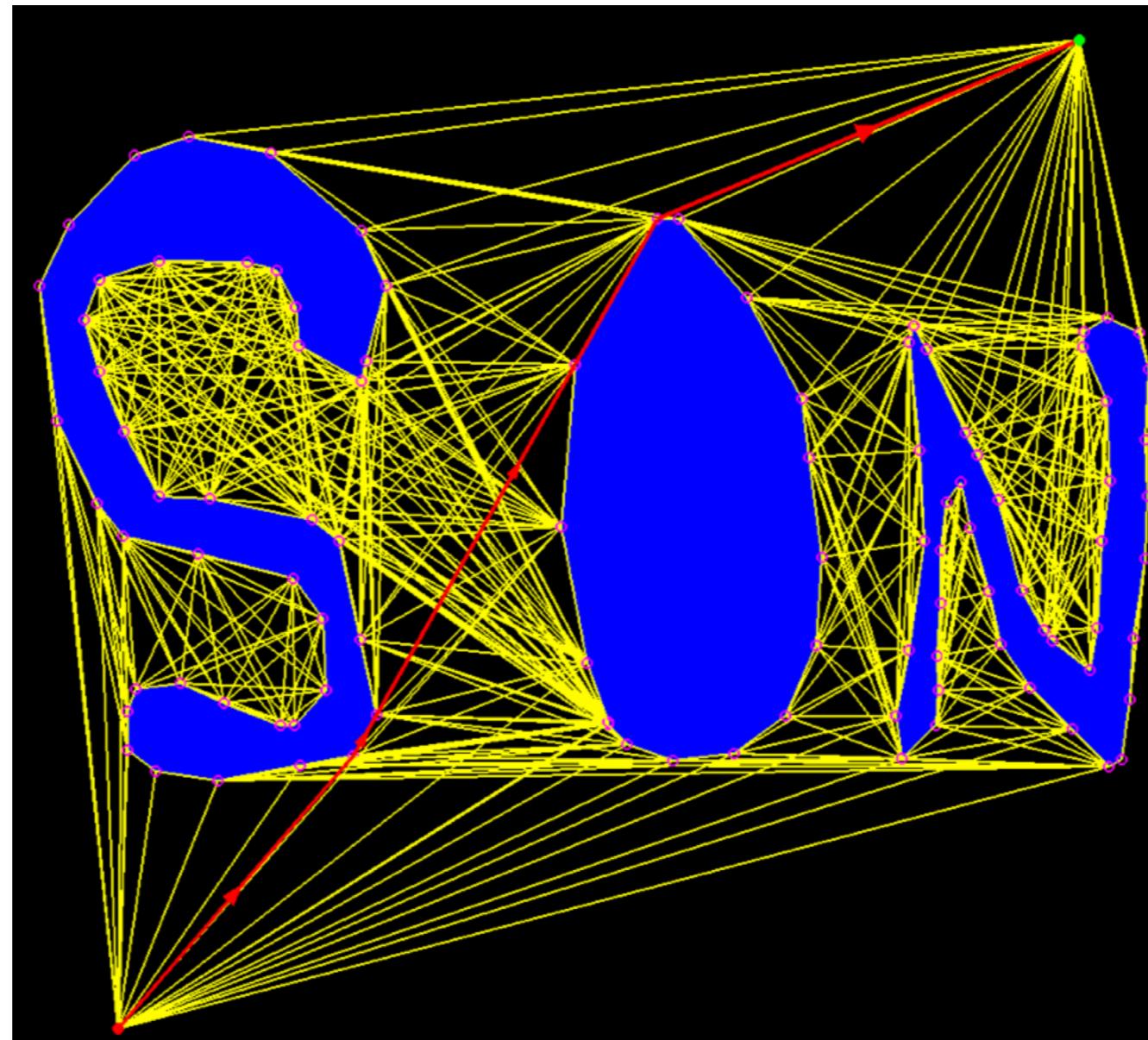
- Difficult to construct a good cell decomposition for curved 3D space
- One possible approach:
  - Discretize rotation angle and compute a finite number of slices, one for each angle
  - For a fixed angle: you got translational motion planning
  - Construct a trapezoidal decomposition for each slice and its roadmap
  - Link them into a 3D roadmap: Add “vertical” edges between slices to allow robot to move up/down between slices; these correspond to rotational moves.
- Example: Consider two angles  $a$  and  $b$ . If placement  $(x,y)$  is in free space in slice  $a$ , and  $(x,y)$  is in free space in slice  $b$ , then the 3D roadmap should contain a vertical edge between slice  $a$  and  $b$  at that position
- Is this complete?

# Combinatorial motion planning : Summary

- **Idea: Compute free C-space combinatorially (exactly)**
- Point robot moving among polygonal obstacles in 2D
  - trapezoidal decomposition
  - visibility graph
- Polygonal robot moving among polygonal obstacles in 2D
  - C-space, C-obstacles, ...
  - translation only, translation + rotation
- **Comments**
  - Complete
  - Works beautifully in 2D and for some (simple) cases in 3D
    - Worst-case bound for combinatorial complexity of C-obstacles in 3D is high
  - Unfeasible/intractable for high #DOF
    - A complete planner in 3D runs in  $O(2^{n^{\#DOF}})$

# Project preview

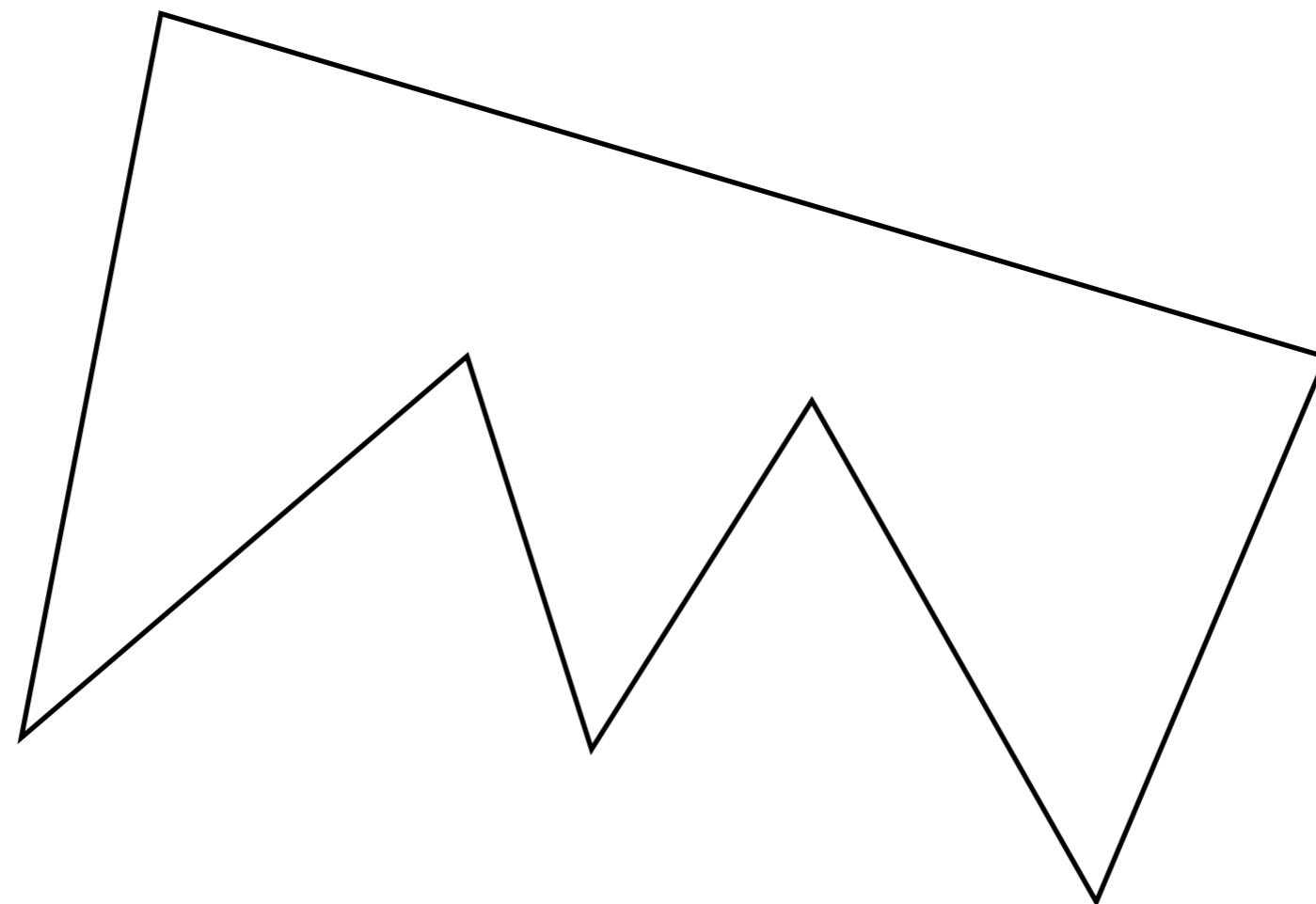
1. Generate a scene consisting of multiple polygonal obstacles and a start and end position. Have a pre-set scene when you start, but also allow the user to change the start and end position for the same scene, and to reset the whole scene and start from scratch entering polygons.
2. Compute and render the visibility graph.
3. Run Dijkstra's algorithm on the VG and render the resulting path (for e.g. in a different color and different line width).



## Additional exercises

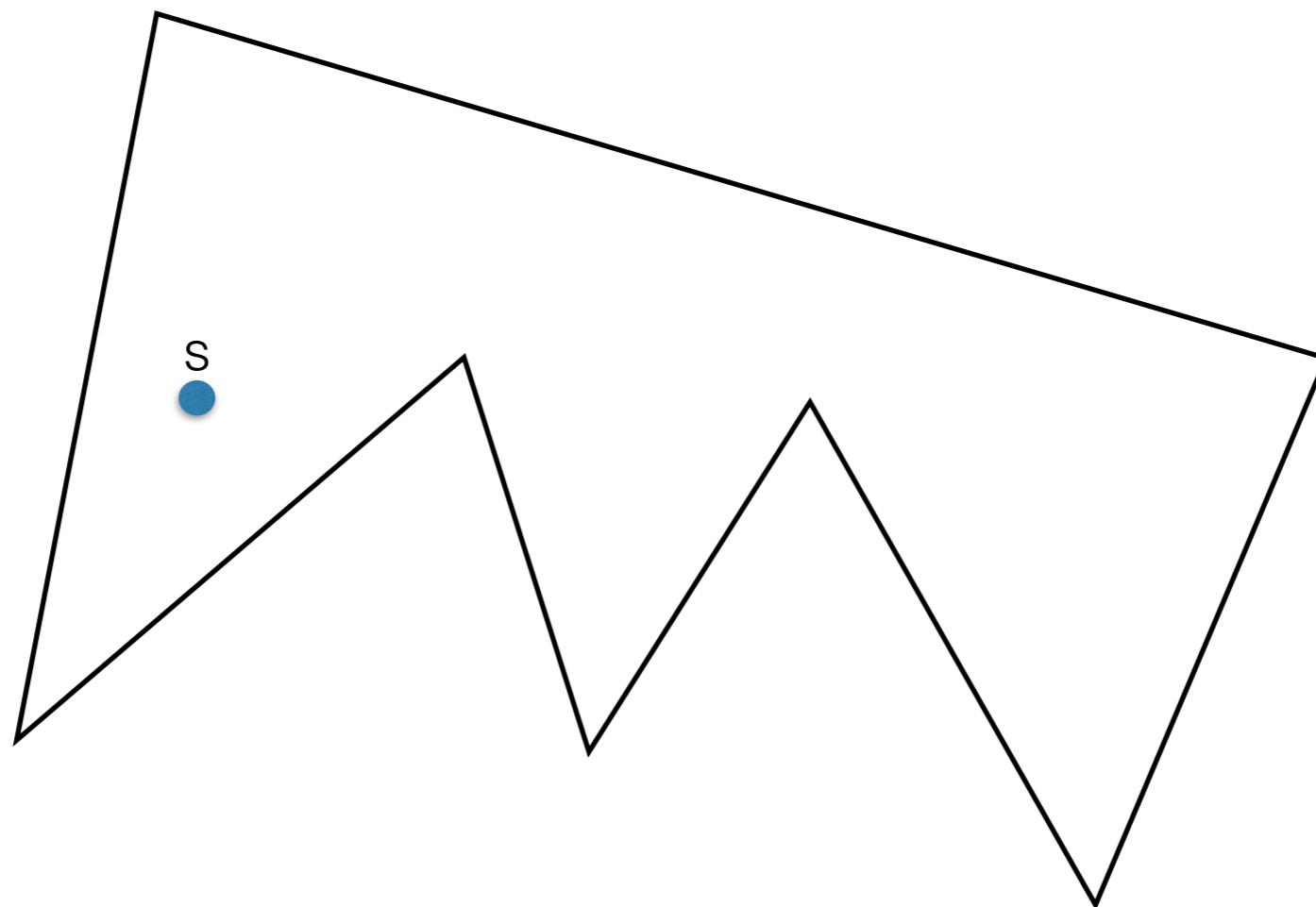
Consider arbitrary two points inside this polygon, and draw the shortest path between them.

What can you claim about the shortest path inside a polygon? (in terms of its intermediate points)



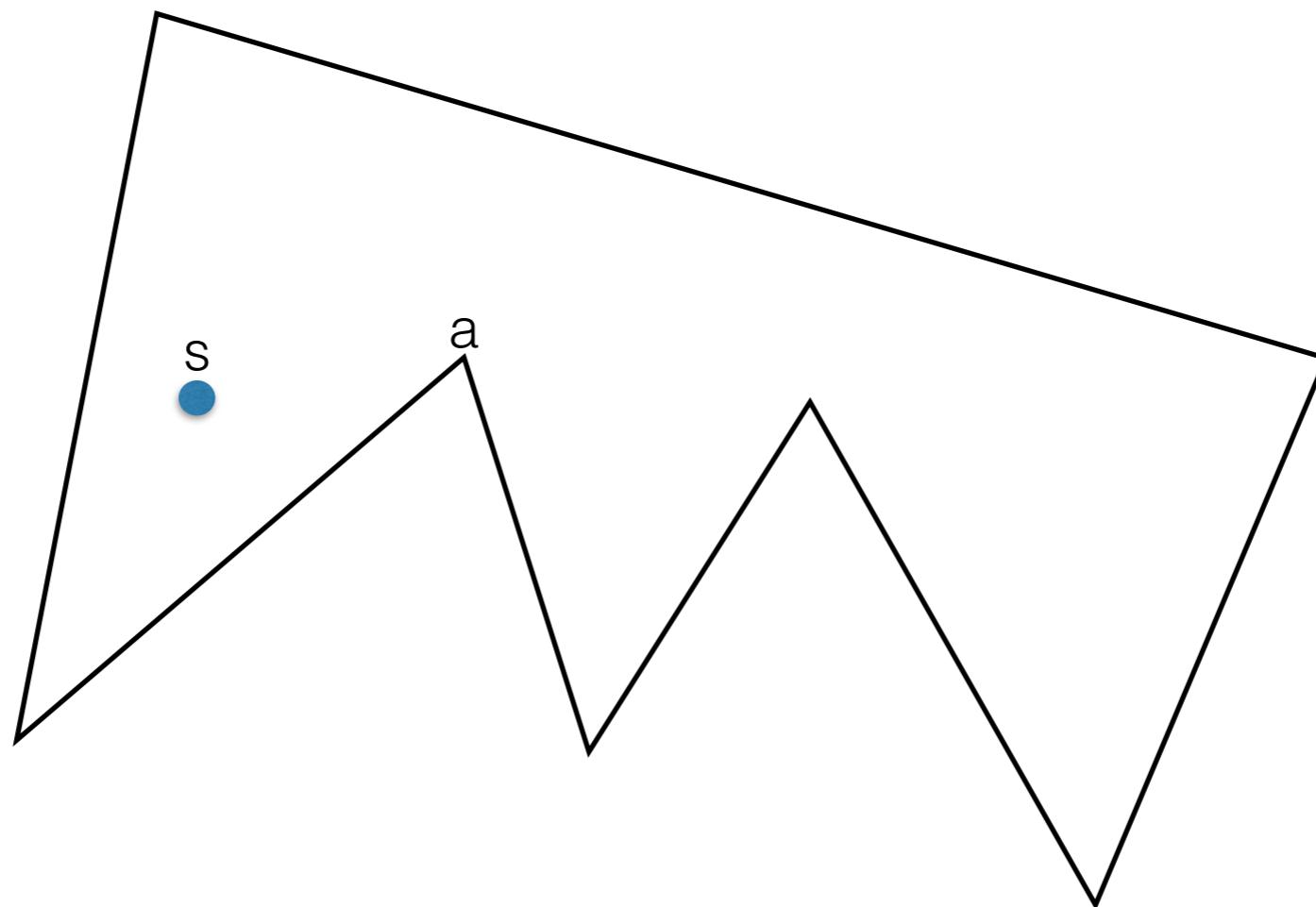
Consider a point s as below.

Draw the region of the polygon that contains all points p such that the shortest path from s to p consists of the straight line segment sp.



Consider a point s as below.

Draw the region of the polygon that contains all points p such that the shortest path from s to p consists of the straight line segment sa plus the straight line segment ap.



Consider a point s as below.  
Draw the shortest path map of s.

