



# Outline

- Why have a world map?
- What's in the vex-aim-tools world map?
- The data association problem
- The kidnapped robot problem
- Obstacle detection

# Why Have A World Map?

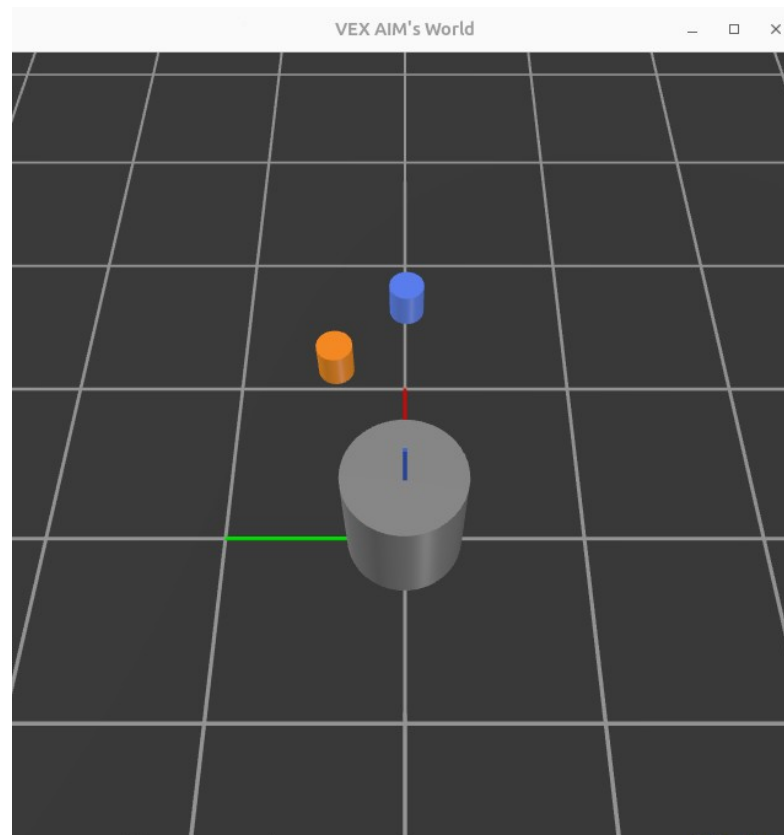
- Represent objects available to the robot, such as barrels and sports balls.
- Represent landmarks to be used for localization, i.e., walls or markers.
- Obstacle avoidance during path planning.

# The vex-aim-tools World Map

- Barrels, balls, AprilTags from “aivision”
- Aruco markers from OpenCV
- Coming soon: walls / doorways
- Future: cliffs
- To access the world map:
  - `robot.world_map` is a `WorldMap` object
- Objects are in:
  - `robot.world_map.objects`

# World Map Coordinates

- $+x$  is North (0 degrees)
- $+y$  is West (+90 degrees)
- $+z$  is Up
- Origin at (0, 0, 0)
- Units are mm and radians.

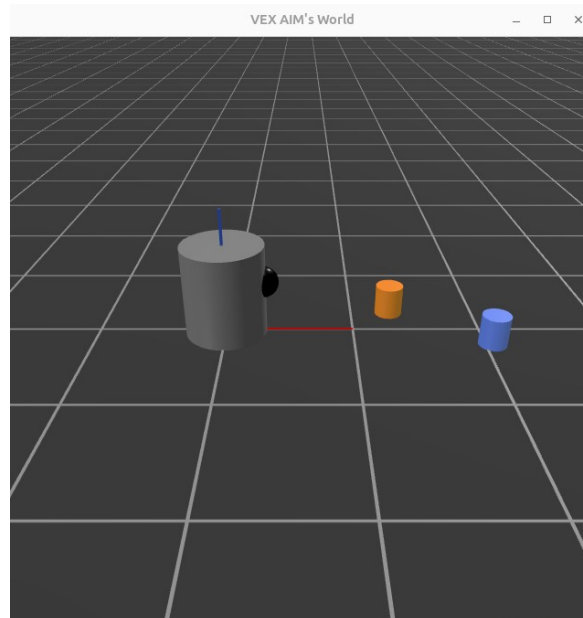
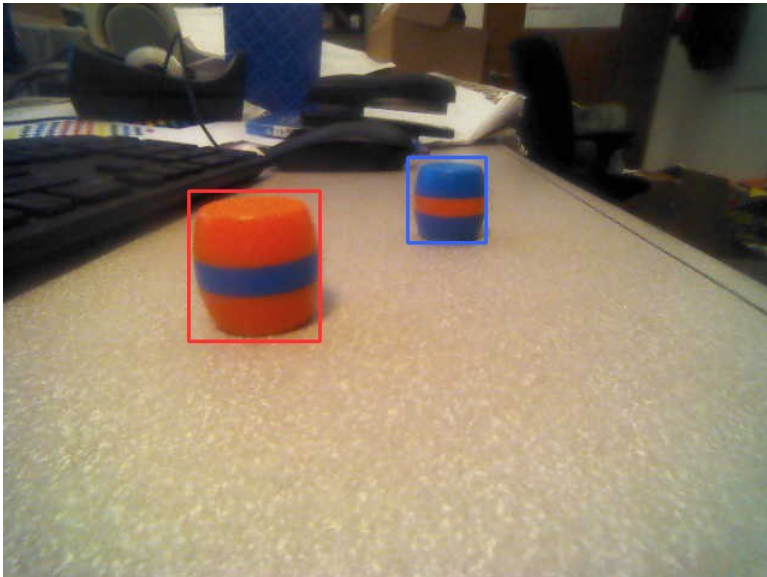


# Object Pose

- A Pose is in  $(x,y,z,\theta)$  coordinates.
  - Angles are in radians
- Pose is defined in `utils.py`.
- The robot and all world map objects have positions that are instances of `utils.PoseEstimate`.
  - Uses a Kalman filter to estimate position

# Converting Camera Coordinates to World Coordinates

- The VEX AIM vision system reports the bounding box of an object in the current camera image. Units are in pixels.
- How do we compute the world coordinates, which are in millimeters?

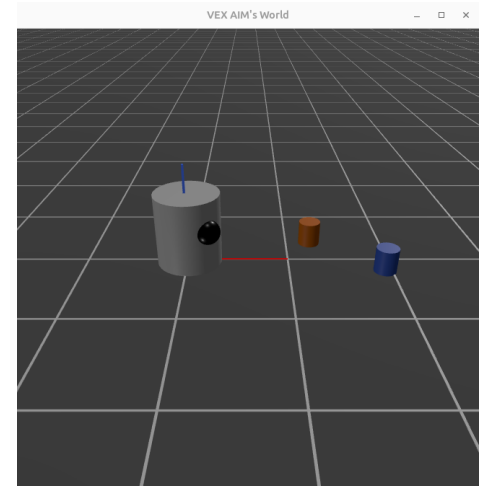
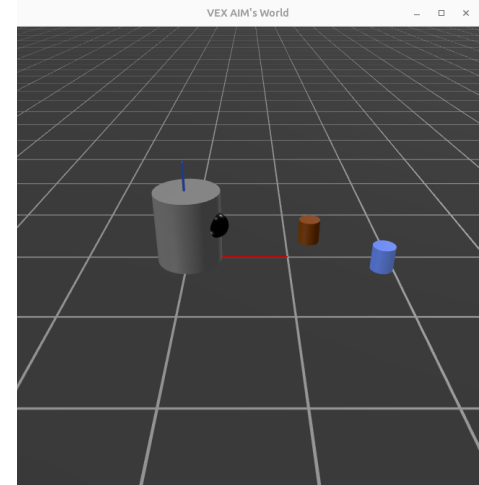
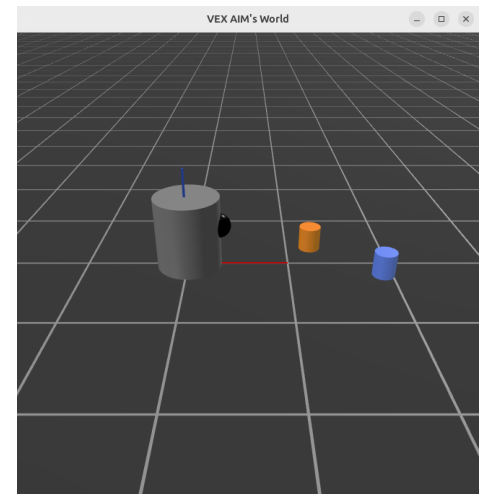


# Converting Camera Coordinates to World Coordinates

- Assume object is on the ground. Use height of bottom edge to estimate distance from robot.
- Requires the following information:
  - Camera parameters: resolution, focal length, field of view
    - From `robot.camera`
  - Camera pose (height and tilt angle)
    - Kinematic tree (“show kine camera”)
  - Robot pose (location and heading)
    - From particle filter (“show pose”)
- See `aim_kin.project_to_ground()` for details.

# Object State

- **visible**: the object is visible in the current camera image.
- **unseen**: the object is not currently visible because the camera isn't facing it.
- **missing**: the object should be visible, but isn't. It could be occluded, or someone could have moved it. Its pose reflects its last known location.



# The Data Association Problem

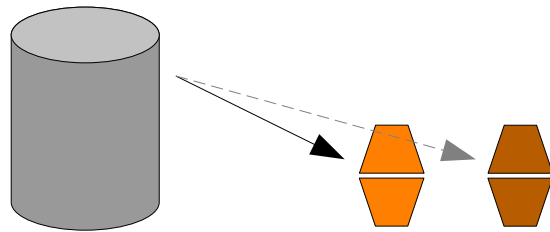
- You see an orange barrel.
  - OrangeBarrel.a
- You move to another position.
- You see an orange barrel again.
- Is this the same barrel, or a new one?
  - Compare world (not camera) coordinates.
  - Requires accurate robot position info.
- Did someone move the first barrel?
- How many orange barrels are there?

# Current vex-aim-tools Solution

- 1) For each new image, find all the objects and calculate their world coordinates.
- 2) Match new objects to world map objects if coordinates are “close enough”.
- 3) Calculate which world map objects *should* be visible given robot pose and note any that weren't matched. Assume they moved somewhere; mark as “missing”.
- 4) If there are seen objects remaining, assign them to missing world map objects.
- 5) For any remaining seen objects, create new world map objects.

# Future Improvements

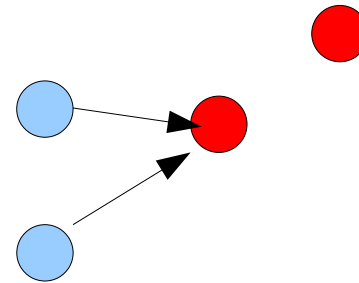
- Take occlusion into account when calculating which objects “should” be visible from current location.



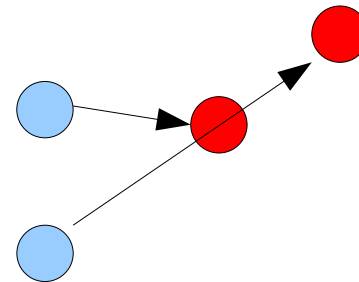
- Use a smarter algorithm to match new objects to world map objects, instead of current greedy algorithm.
  - The Hungarian algorithm (Munkres algorithm) does a better job.

# Data Association

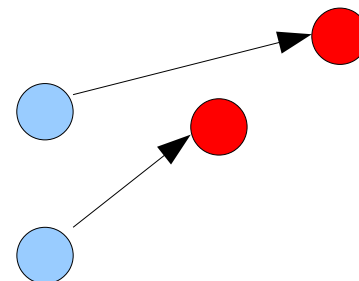
- Independent (greedy) closest match:



- Closest match without replacement:



- Hungarian algorithm:



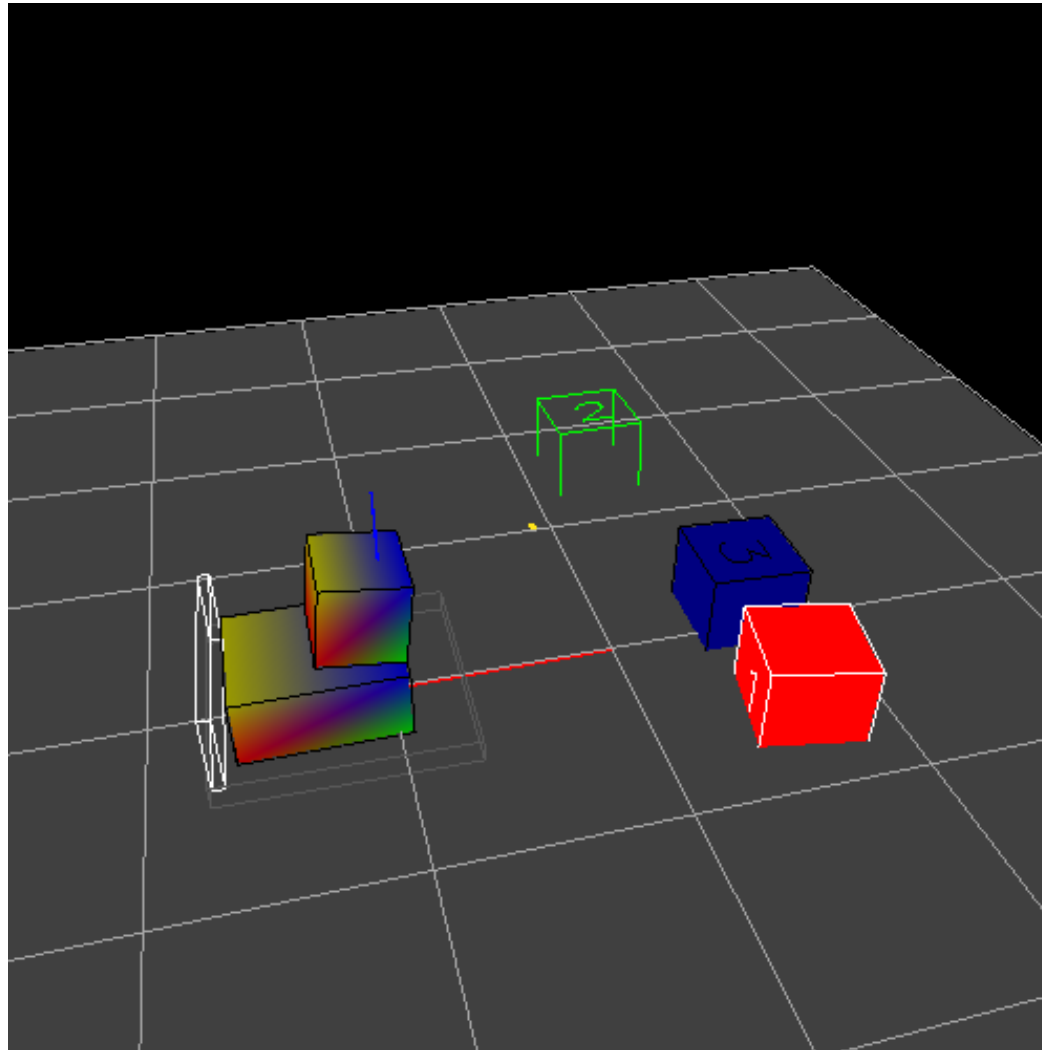
# The Kidnapped Robot Problem

- What do we do when the robot is picked up and placed in an unknown location?
- If familiar landmarks are present, we can randomize the particles and relocalize.
- What if we don't see any landmarks, but we do see new objects?

# Origin ID (Cozmo)

- The robot's origin\_id starts at 1.
- Every time Cozmo is picked up and put down, he may get a new origin\_id value.
- Landmarks can pull him back to an old id.
- Every time the robot sees an object, that object's origin\_id is updated to match the robot's.
- Object poses are only valid if their origin\_id matches the robot's.

# show worldmap\_viewer



# Celeste and the World Map

- When a request is sent to Celeste (meaning GPT-4o), we also send a textual description of the current world map.
- This includes: coordinates of all objects and their visibility status, the visibility of any landmarks, the robot's pose, whether the robot is holding anything, etc.
- See `get_prompt()` in `worldmap.py`.

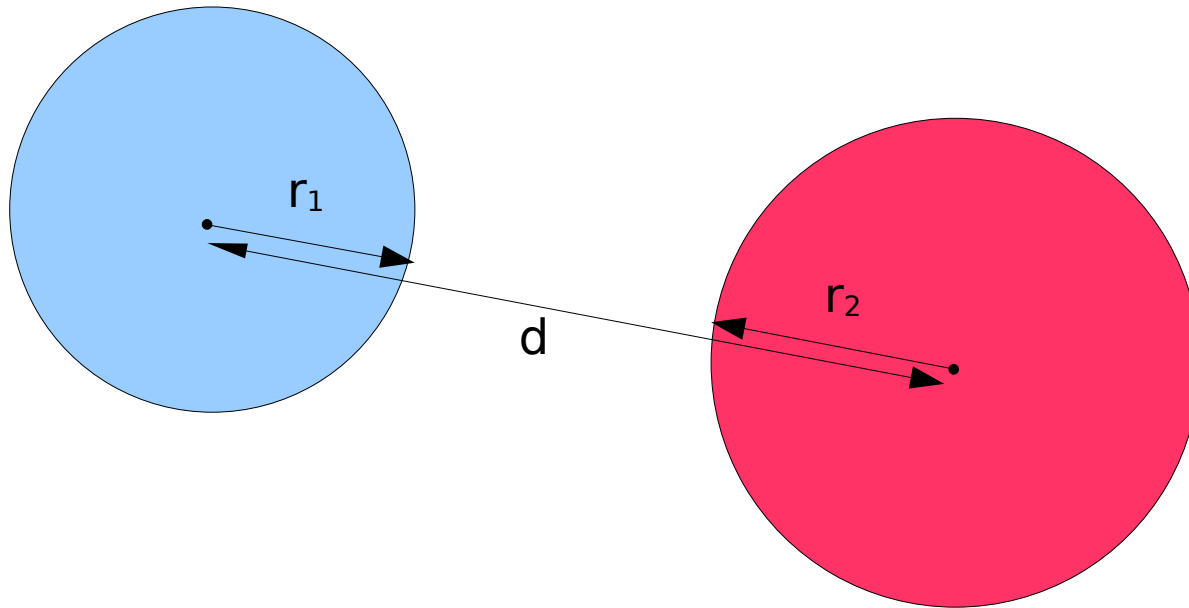
# Obstacle Avoidance

- Having a world map allows us to do path planning to avoid obstacles.
- The path planner needs a way to detect when two objects would collide.
- How can we detect collisions?

# Collision Detection

- Represent the robot and the obstacles as **convex polygons**.
- In 2D, use the Separating Axis Theorem to check for collisions.
  - Easy to code
  - Fast to compute
- In 3D, things get more complex.
  - The GJK (Gilbert-Johnson-Keerthi) algorithm is used in many physics engines for video games.

# Collision Detection: Circles

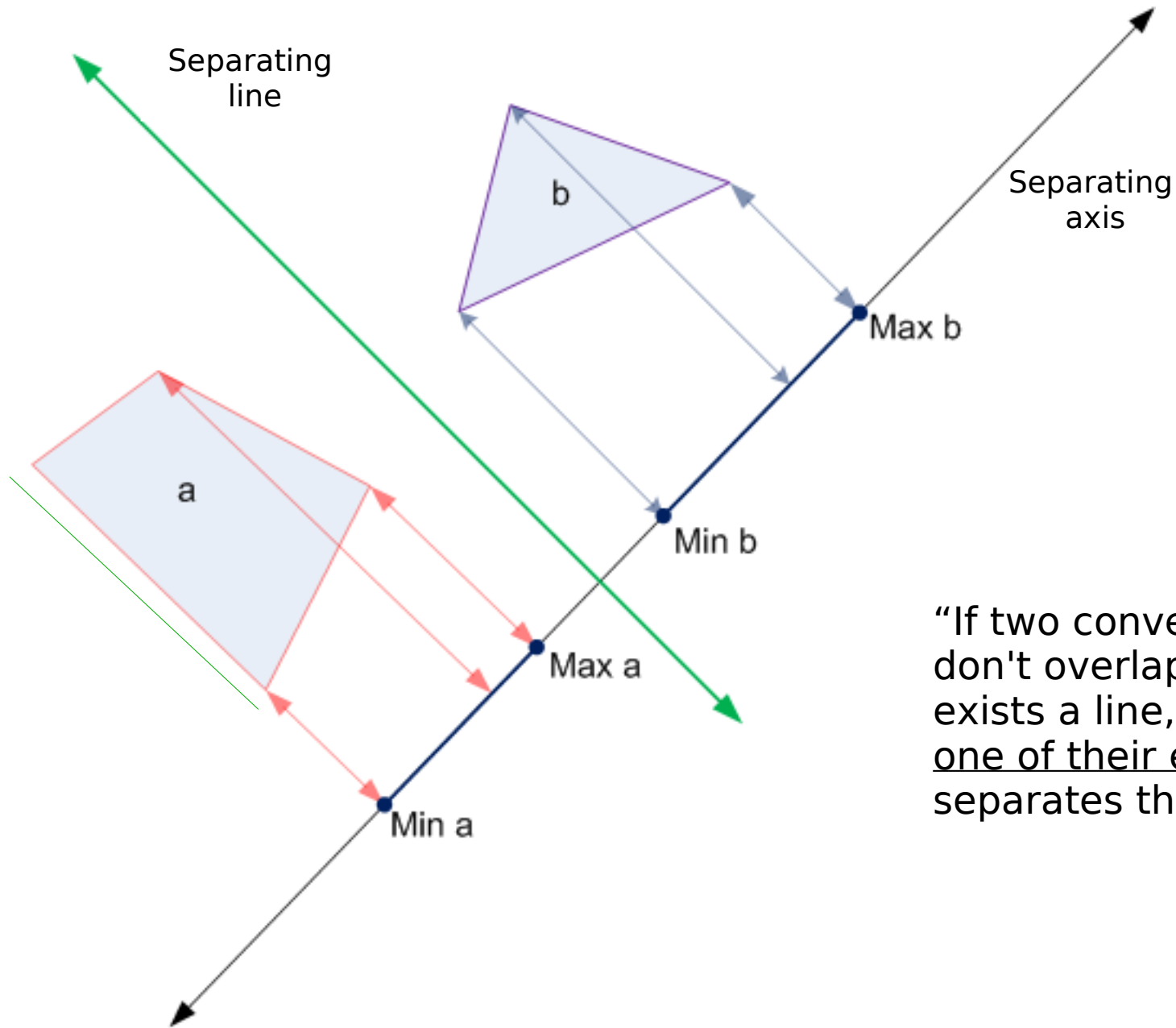


- Let  $d$  = distance between centers
- Let  $r_1, r_2$  be the radii
- No collision if  $d > r_1 + r_2$

# Collision Detection: Two Convex Polygons

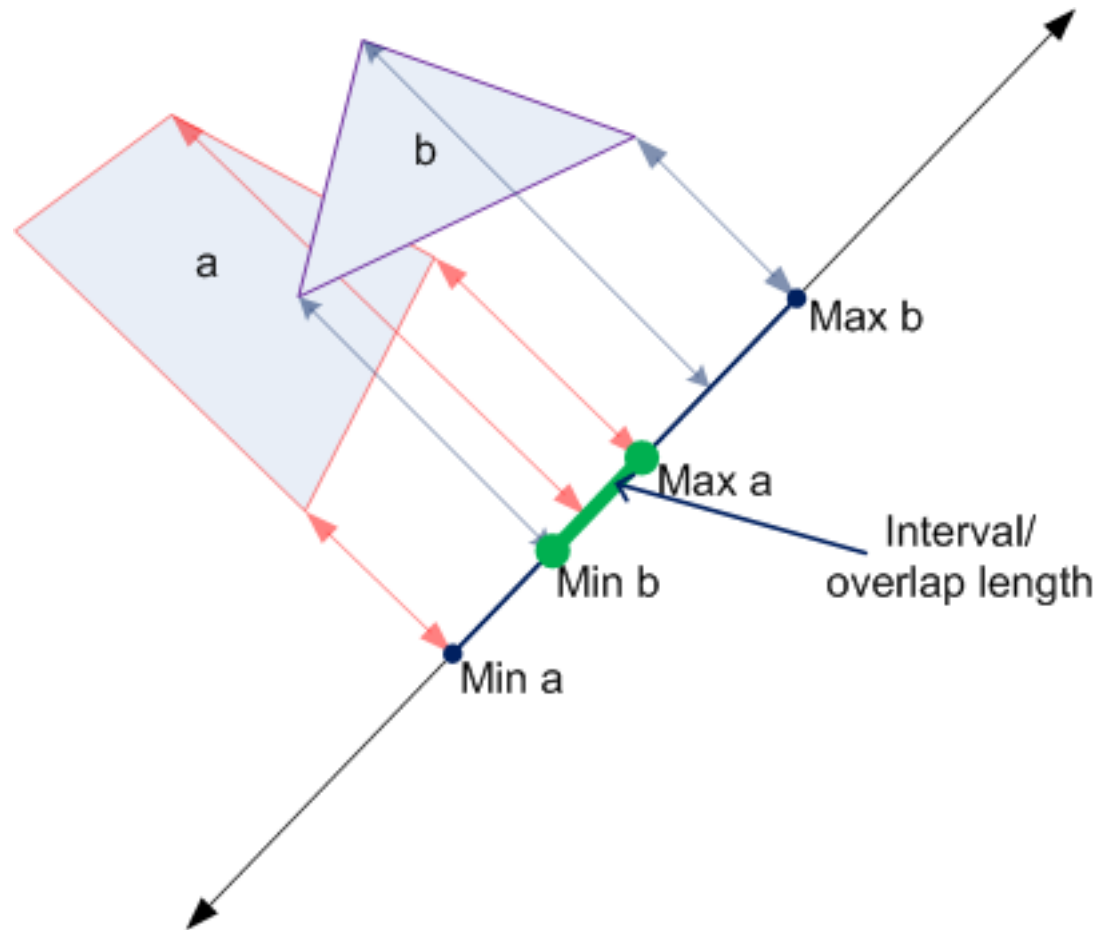
- The Separating Axis Theorem can be used to detect collisions between two convex polygons.
- Time is proportional to the number of vertices.
- To handle non-convex polygons, decompose them into sets of convex polygons and check for collisions between any two components.

# Separating Axis Theorem

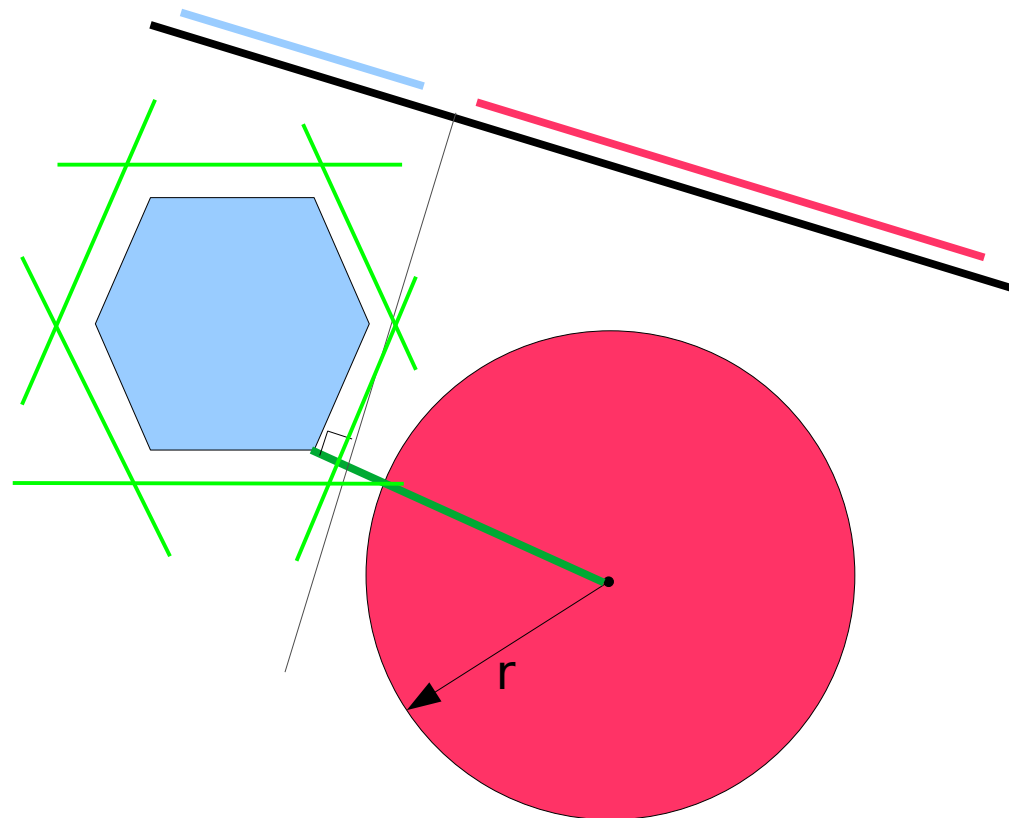


“If two convex polygons don't overlap, then there exists a line, parallel to one of their edges that separates them.”

# Separating Axis Theorem



# Collision Detection: Circle and Convex Polygon



- Separating axes to check are parallel to the edges of the polygon or orthogonal to the line joining the nearest vertex to the center of the circle.

# Collision Detection Algorithm

We only need to find one separating axis to be assured of no collision.

```
def collision_check(poly1, poly2):  
    for axis in Edges(poly1) ∪ Edges(poly2):  
        base = perpendicular_to(axis)  
        proj1 = project_verts(poly1, base)  
        proj2 = project_verts(poly2, base)  
        if not overlap(proj1, proj2):  
            return False  
    return True
```

# How To Build A World Map

- SLAM: Simultaneous Localization and Mapping algorithm.
- Each particle stores:
  - a hypothesis about the robot's location  $(x, y, \theta)$
  - a hypothesis about the map, e.g., a set of landmark identities and locations:  $\{ (i, x_i, y_i, \theta_i) \}$ .
- Particles score well if:
  - Landmark locations match the sensor values predicted by the robot's location.