

# Webots Basics



# Computer vs. Robotic Programs

---

- **Computer Programs:**

- designed to compute an answer
- data usually valid when available
- predictable program flow
- foreseen errors easily handled

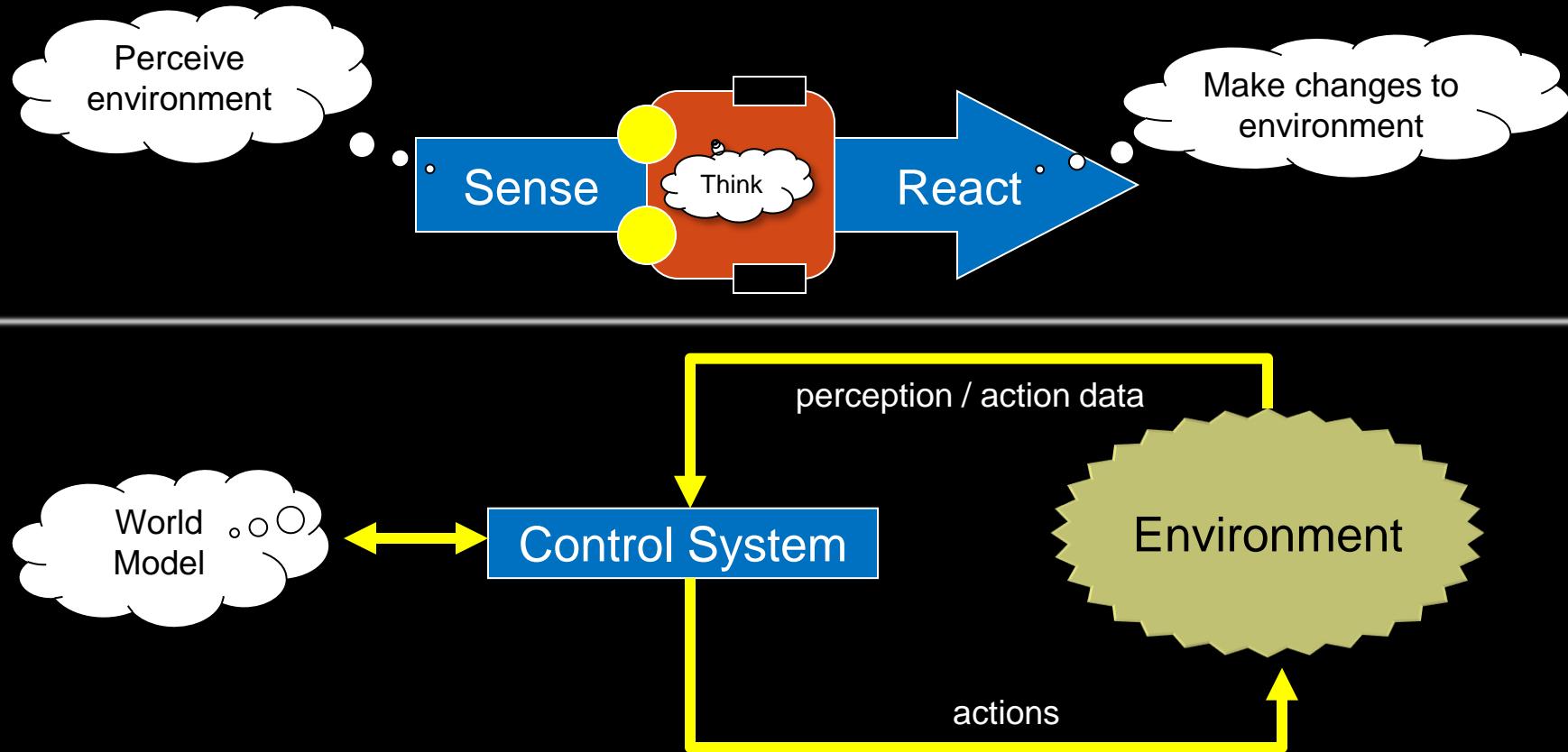
- **Robotic Programs:**

- designed to react to achieve goals, not “an answer”
- sensors often produce invalid data (or data missing)
- unpredictable situations due to dynamic environment  
(e.g., unforeseen obstacles, wrong or missing sensor data, communication outages, hardware failure, etc..)
- program must degrade gracefully in difficult situations



# Robot Processing

- Here is the basic “flow of control” commonly used:



# Robot Control - Pseudocode

Each robot program runs in an infinite loop.

```
MyRobotController() {  
    Set up sensors and motors  
  
    WHILE (TRUE) {  
        detectLeft = read left sensor;  
        detectRight = read right sensor;  
  
        IF (detectLeft) THEN  
            turn = right;  
        ELSE IF (detectRight) THEN  
            turn = left;  
        ELSE  
            turn = none;  
  
        IF (turn == left) THEN  
            turn leftMotor on backwards  
            turn rightMotor on forward  
        ELSE IF (turn == right) THEN  
            turn rightMotor on backwards  
            turn leftMotor on forward  
        ELSE  
            turn rightMotor on forward  
            turn leftMotor on forward  
    }  
}
```

## Initialize

Set up the motors, sensors, variables etc...

## Sense

Read sensors to determine if a collision is detected on the left or right sensor.

## Think

Decide whether robot should go straight or turn away.

## React

Turn left, turn right, or move forward accordingly.

# Robot Control – JAVA code

Informs JAVA compiler where to find libraries of various functions that you will use in your program. You will add lots more **import** statements.

All classes are given a name which must match the filename ...  
**(LabController.java** in this case)

```
import com.cyberbotics.webots.controller.Robot;

public class LabController {
    /* Declare static constants & variables here */
    public static void main(String[] args) {
        Robot robot = new Robot();           // Creates a generic Robot object
        int timeStep = (int) Math.round(robot.getBasicTimeStep());
        /* INITIALIZE ... variables, sensors, motors, etc... */

        while(robot.step(timeStep) != -1) {
            /* SENSE ... Read sensors */

            /* THINK ... Make decision as to what to do */

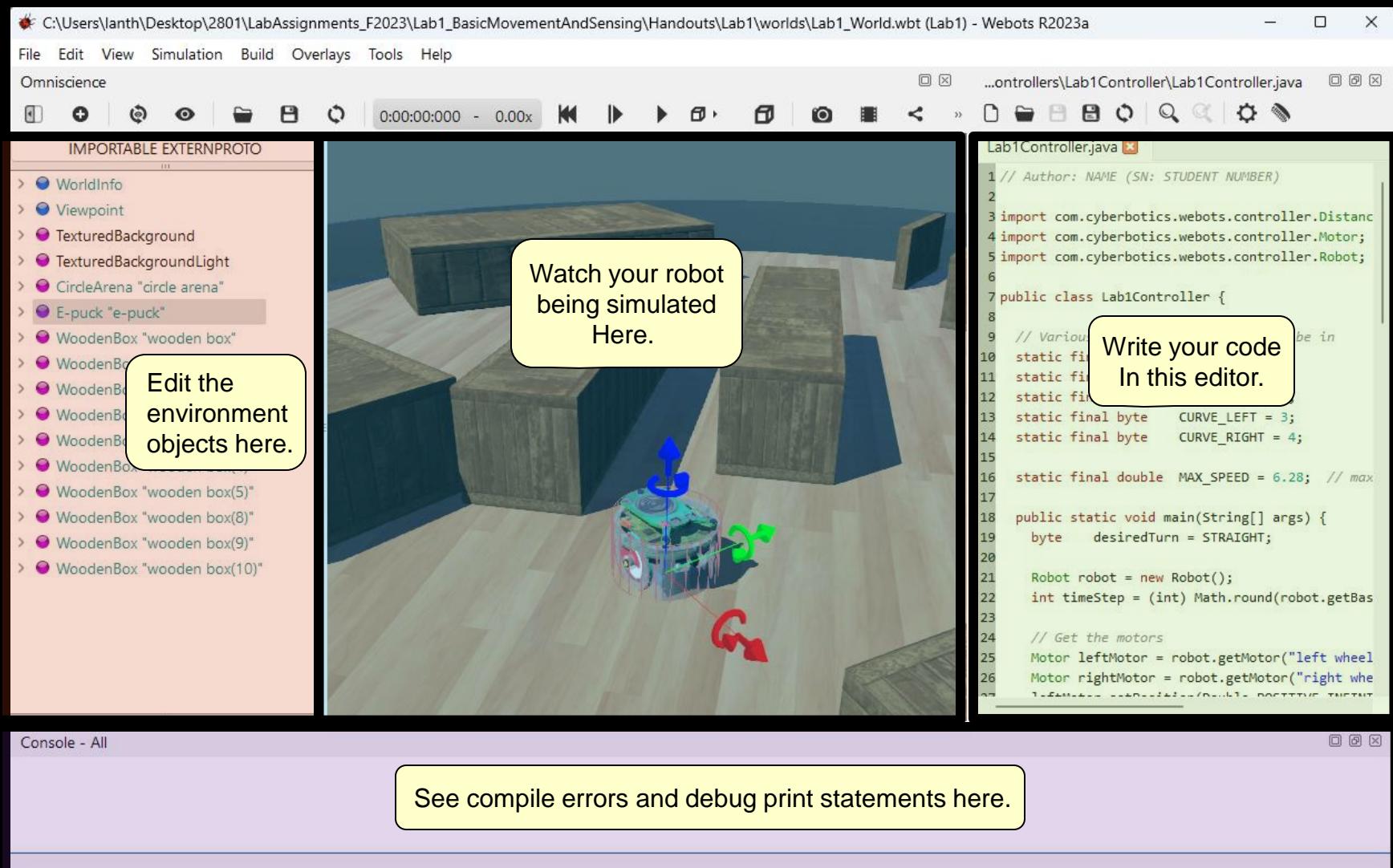
            /* REACT ... Move motors, head, arms, etc... */
        }
    }
}
```

**main** function is starting point of your program.

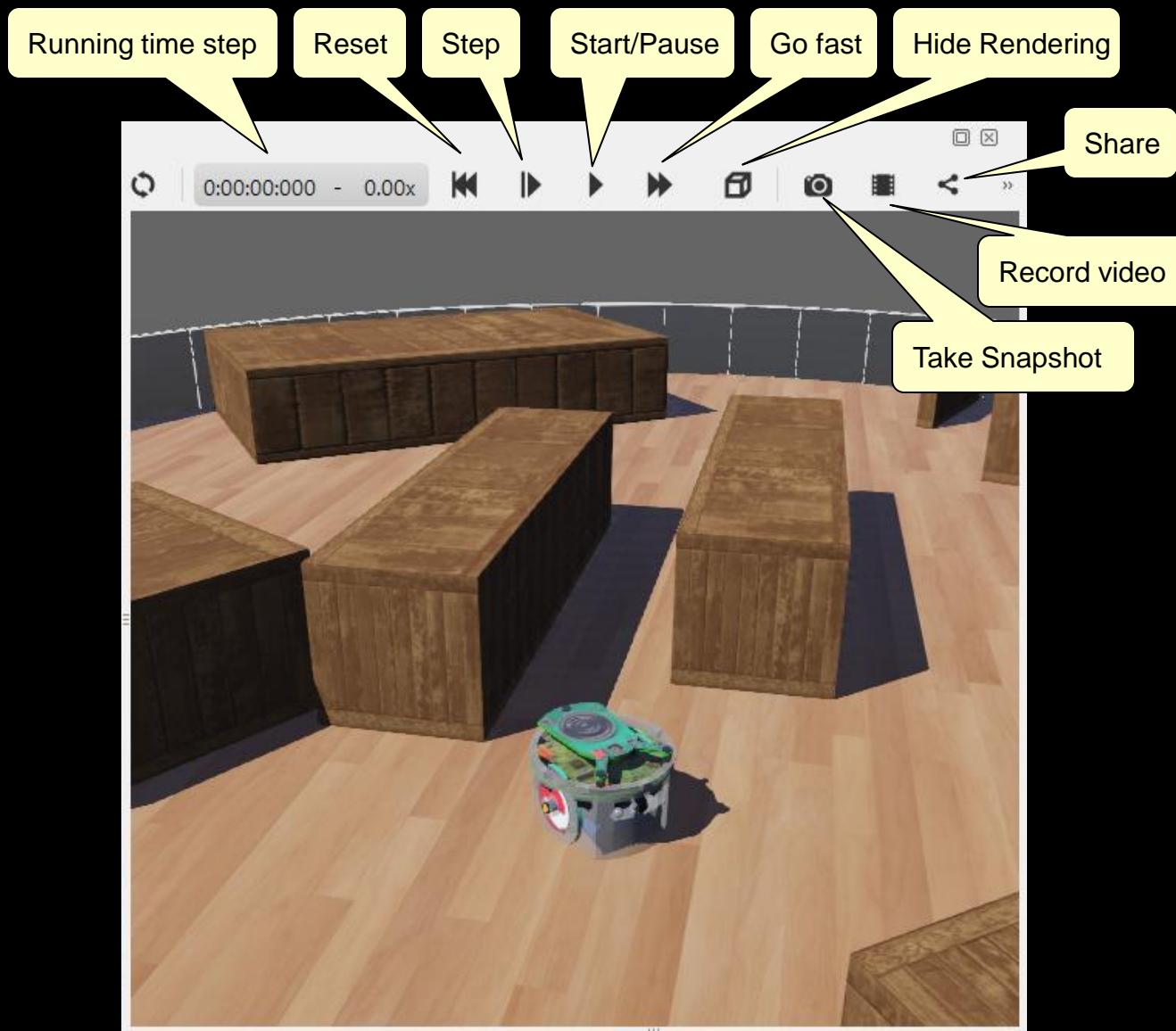
Simulation time-step in (ms).  
You won't change this.

Loops forever (until PAUSED or STOPPED)

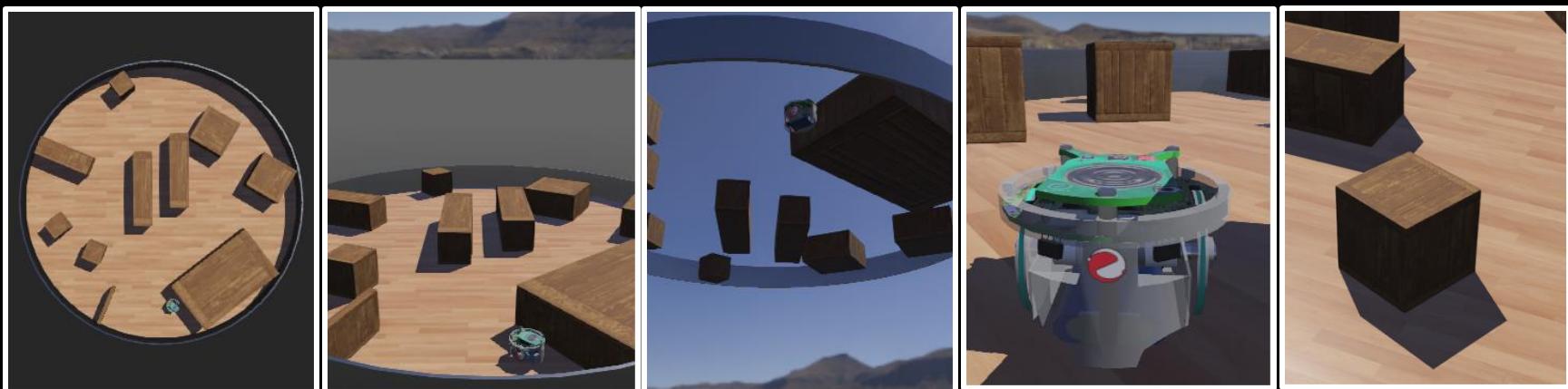
# Webots Interface – main GUI



# Webots Interface – Simulation Controls



# Webots Interface – Changing Viewpoint



Use the mouse to change viewpoint:

**Roll:** left press + up/down

**Spin:** left press + left/right

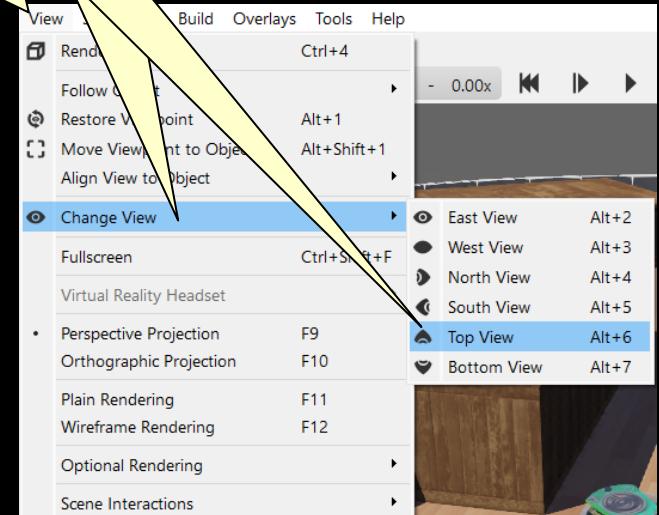
**Translate:** right press + up/down/left/right

**Zoom:** scroll up/down

**Select:** left click

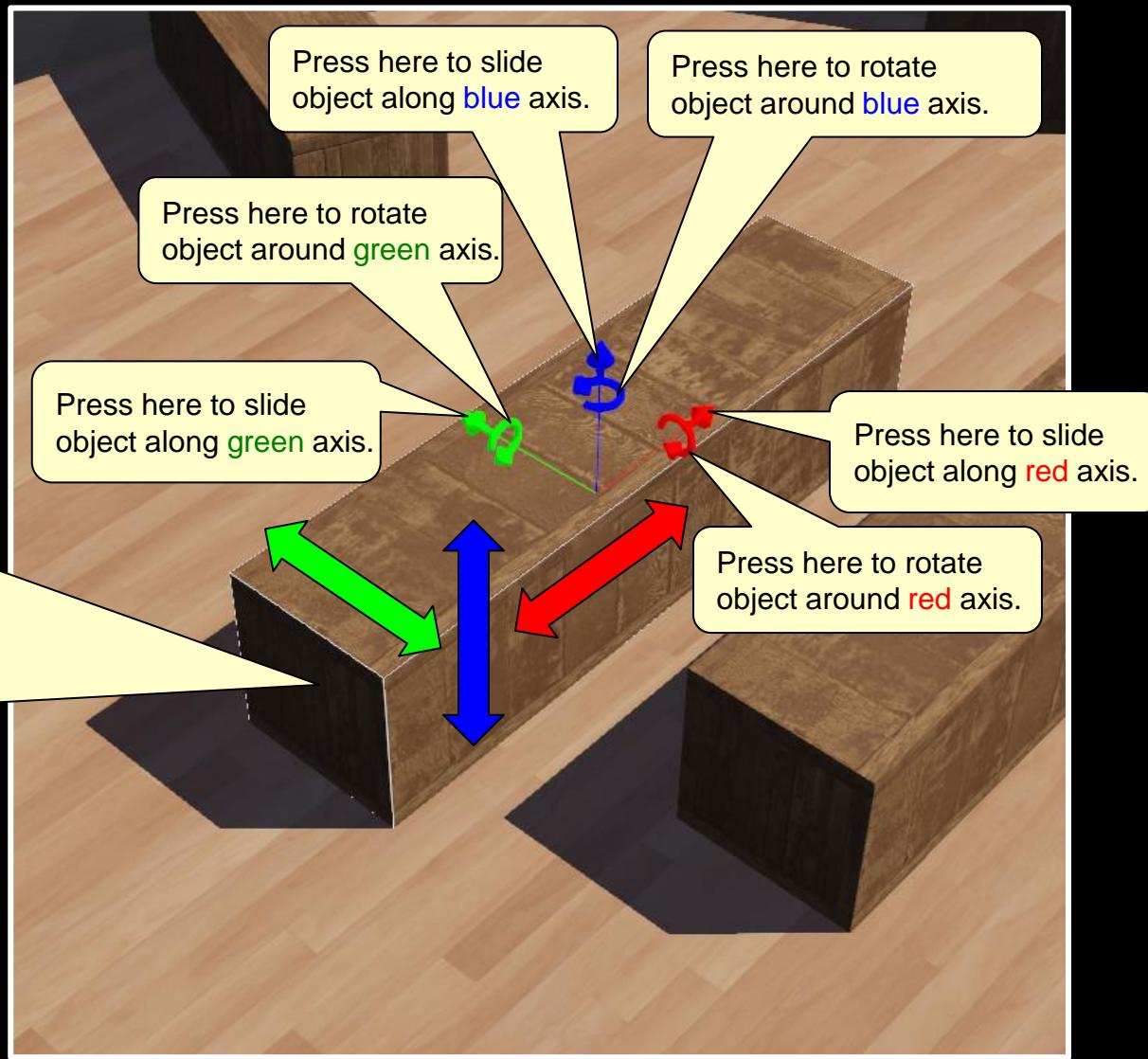
**Menu:** right click

View menu has more options



# Webots Interface – Moving World Objects

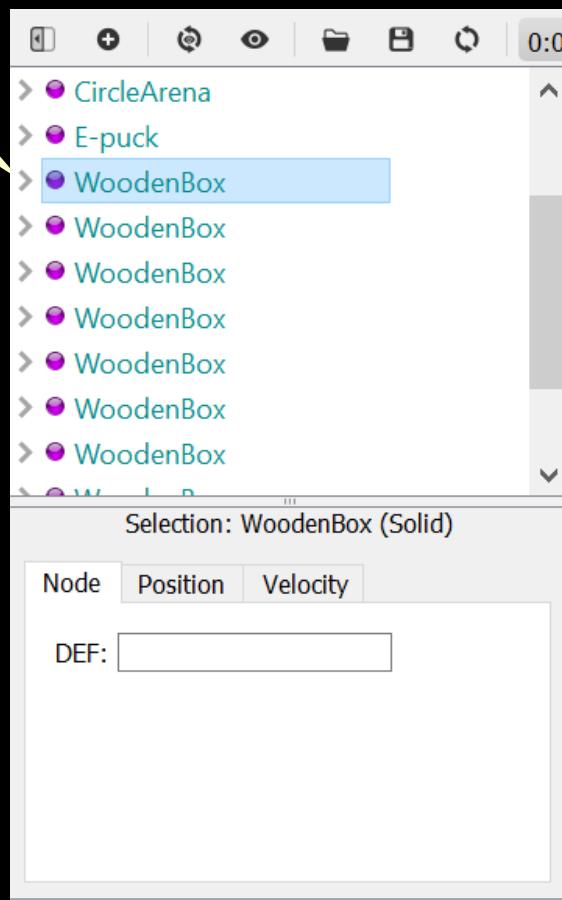
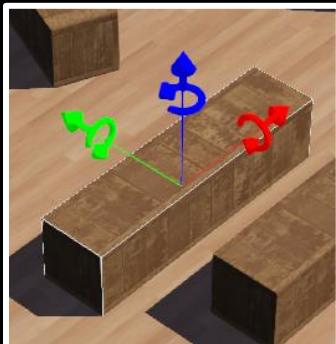
Click anywhere on object to select it, then press and hold left mouse button on one of 6 places to change object.



# Webots Interface – Editing World Objects

1. Click on the > to expand or collapse one object's attribute view.

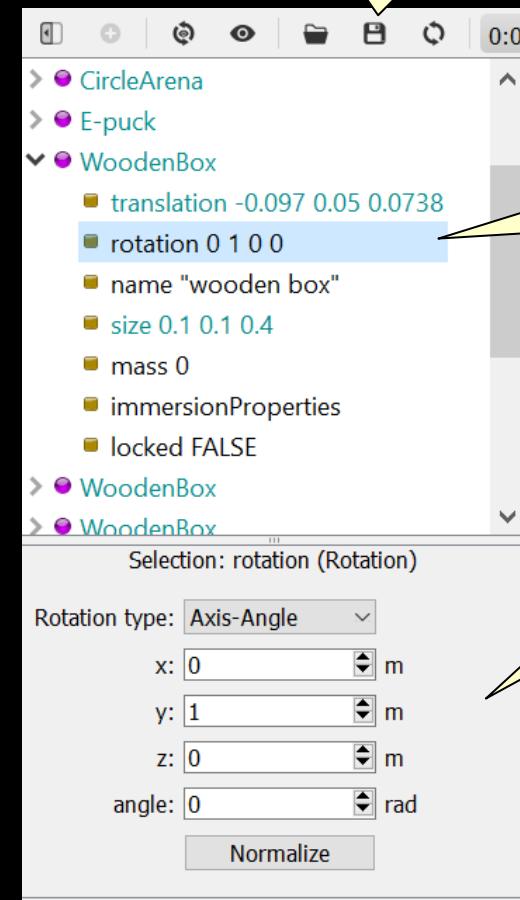
Objects are all listed here in the **Scene Tree**.



The screenshot shows the Webots interface with the Scene Tree on the left and the Attribute Editor on the right. In the Scene Tree, several objects are listed, including 'CircleArena', 'E-puck', and multiple 'WoodenBox' objects. One 'WoodenBox' object is selected, highlighted with a blue border. The Attribute Editor below shows the selected object's attributes:

Attribute	Value
translation	-0.097 0.05 0.0738
rotation	0 1 0 0
name	"wooden box"
size	0.1 0.1 0.4
mass	0
immersionProperties	
locked	FALSE

4. Save your changes.



The screenshot shows the Attribute Editor for a selected 'WoodenBox' object. The 'rotation' attribute is currently selected and highlighted with a blue border. The rotation values are displayed in the editor:

Attribute	Value
x	0
y	1
z	0
angle	0

A yellow callout points to the 'y' value with the instruction '2. Select an attribute.'

A yellow callout points to the 'Normalize' button with the instruction '3. Change its values.'

2. Select an attribute.

3. Change its values.

# Webots Interface – Making New World

1. Select **New Project Directory...** from the **File/New** menu

2. Press **Next.**

3. Enter a folder name.

4. Press **Next.**

5. Enter name for world.

6. Check this.

7. Press **Next.**

8. Press **Finish.**

9. Select arena and right-click to bring up pop-up menu.

10. Select **Add New** to add objects and robots. The menu can take up to 10 seconds to appear!

Here is the result!

C:\Users\lanth\Desktop\2801\LabAssignments\_F2023\Lab1\_BasicMovementAndSensin

File Edit View Simulation Build Overlays Tools Help

New Open World... Open Recent World Open Sample World... Save World

New Project Directory... New World File... New Robot Controller... New Physics Plugin... New PROTO...

New World File... Ctrl+Shift+N New Robot Controller... Ctrl+Shift+R New Physics Plugin... Ctrl+Shift+P New PROTO... Ctrl+Shift+T

Create a Webots project directory

New project creation

This wizard will help you creating a new project folder.

2. Press **Next.**

3. Enter a folder name.

4. Press **Next.**

5. Enter name for world.

6. Check this.

7. Press **Next.**

8. Press **Finish.**

Conclusion

The following folders and files will be created:

C:\Users\lanth\Desktop\2801\TestLab\ C:\Users\lanth\Desktop\2801\TestLab\worlds\ C:\Users\lanth\Desktop\2801\TestLab\worlds\TestWorld.wbt C:\Users\lanth\Desktop\2801\TestLab\controllers\ C:\Users\lanth\Desktop\2801\TestLab\protos\ C:\Users\lanth\Desktop\2801\TestLab\resources\ C:\Users\lanth\Desktop\2801\TestLab\plugins\ C:\Users\lanth\Desktop\2801\TestLab\libraries\

File Edit View Simulation Build Overlays Tools Help

Simulation View

Text Editor

IMPORTABLE EXTERNPROTO

WorldInfo Viewpoint TexturedBackground TexturedBackgroundLight RectangleArena "rectangle arena"

Console - All

C:\Users\lanth\Desktop\2801\TestLab\worlds\TestWorld.wbt (TestLab) - Webots R2023a

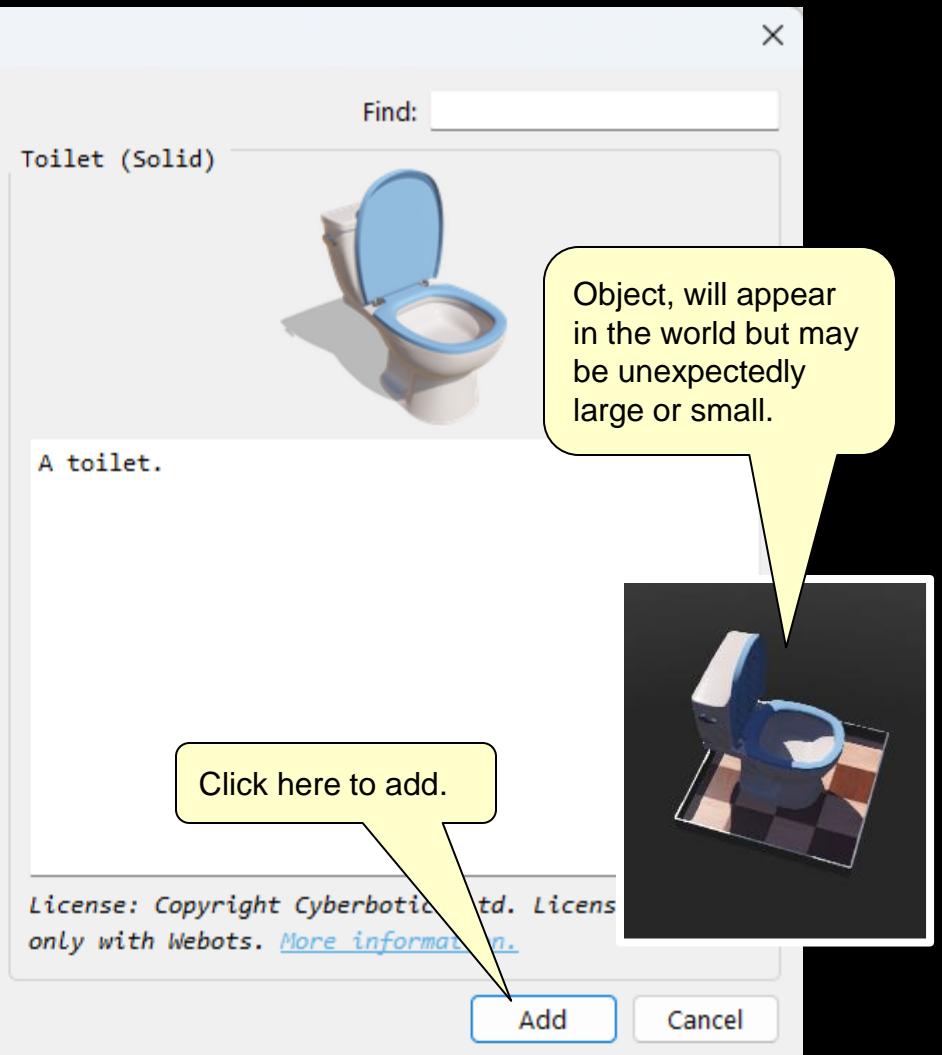
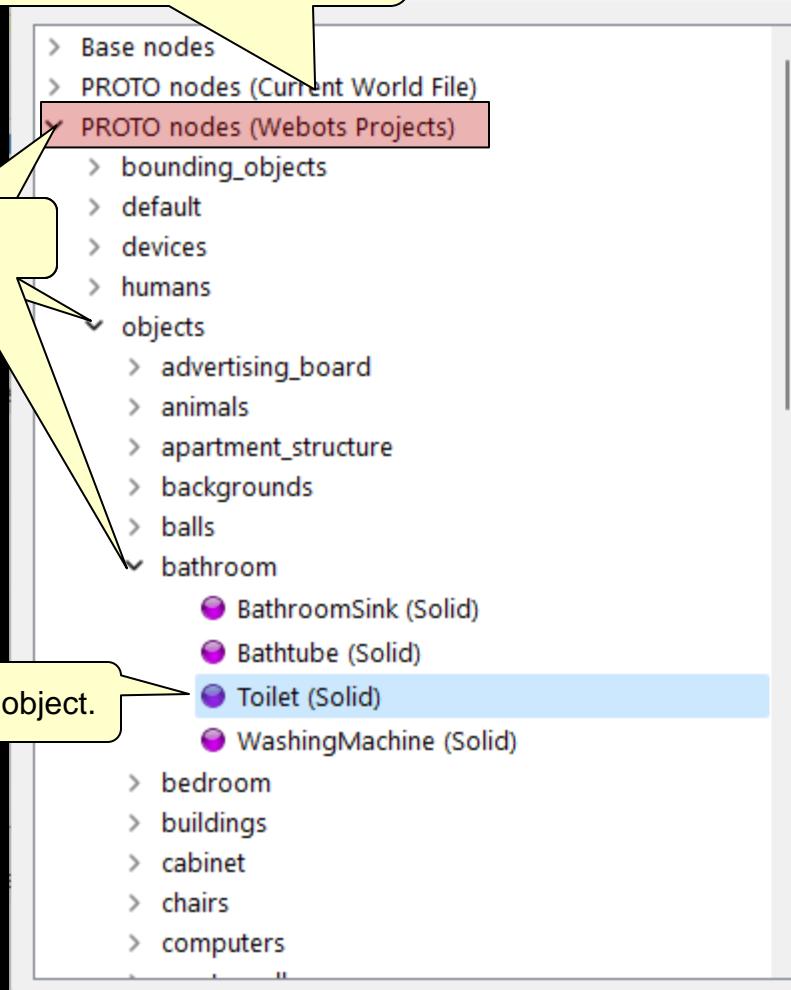
Cut Ctrl+X Copy Ctrl+C Paste Ctrl+V Reset to Default Value Edit... Add New Ctrl+Shift+A Delete Del Move Viewpoint to Object Alt+Shift+1 Align View to Object Follow Object Optional Rendering Edit PROTO Source View PROTO Source View Generated PROTO Node Convert to Base Node(s) Convert Root to Base Node(s) Help...

# Webots Interface – Adding an object

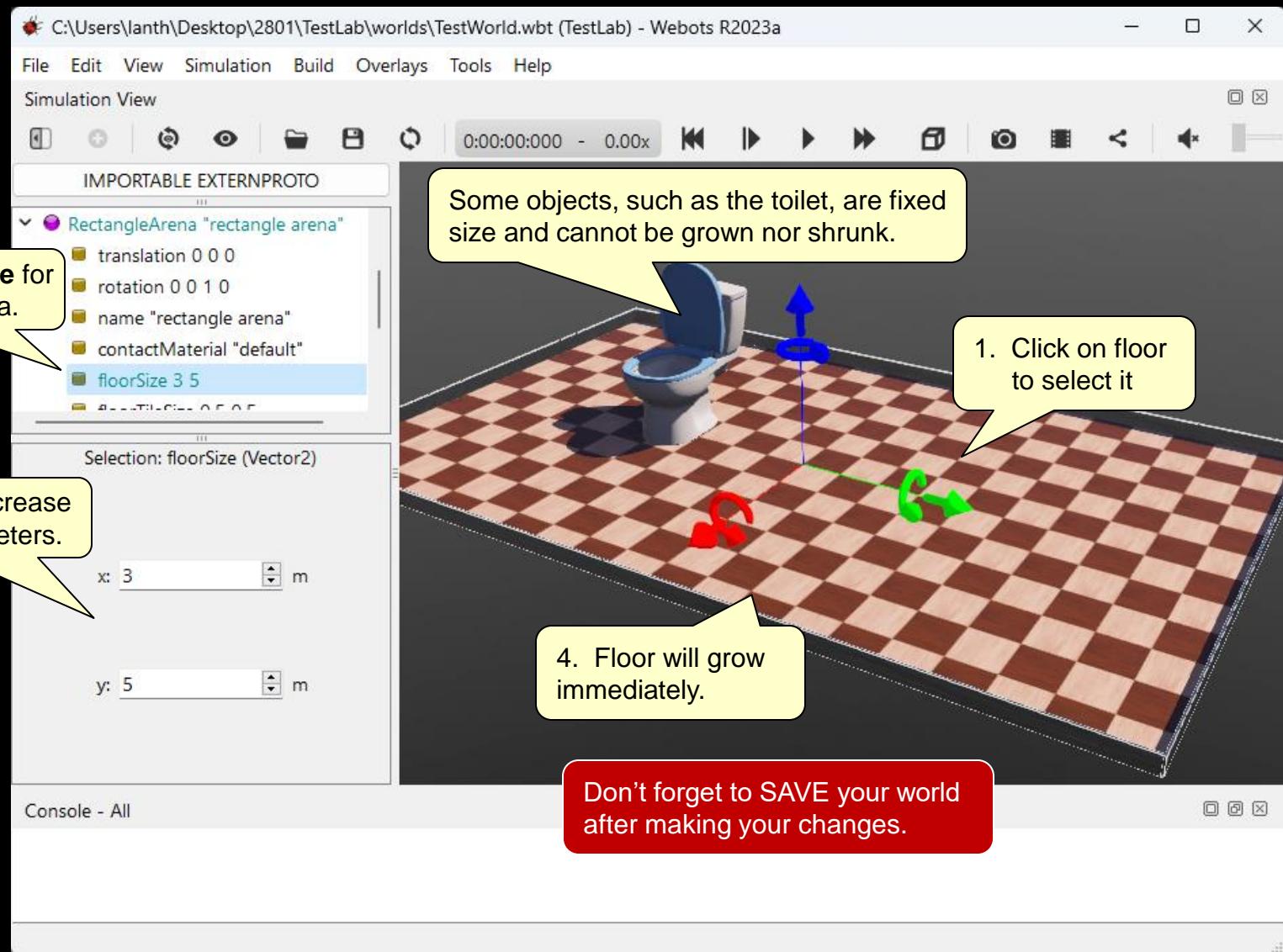
Objects are listed by category under  
PROTO nodes (Webots Projects)

Press > to  
expand.

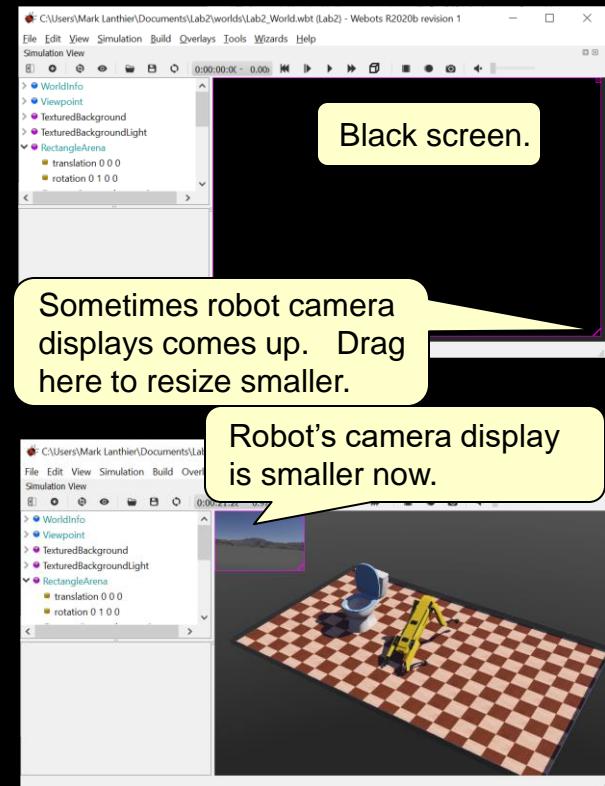
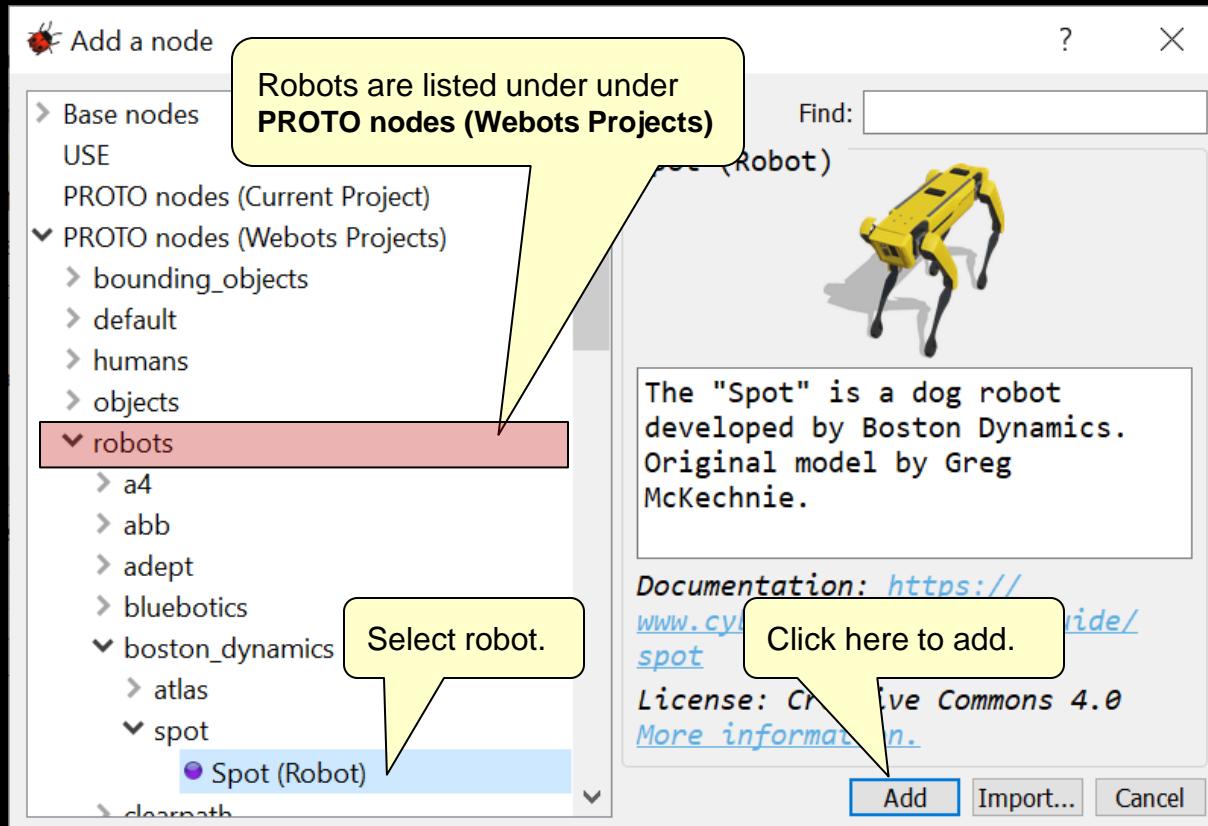
Select object.



# Webots Interface – Resizing an Object



# Webots Interface – Adding a Robot



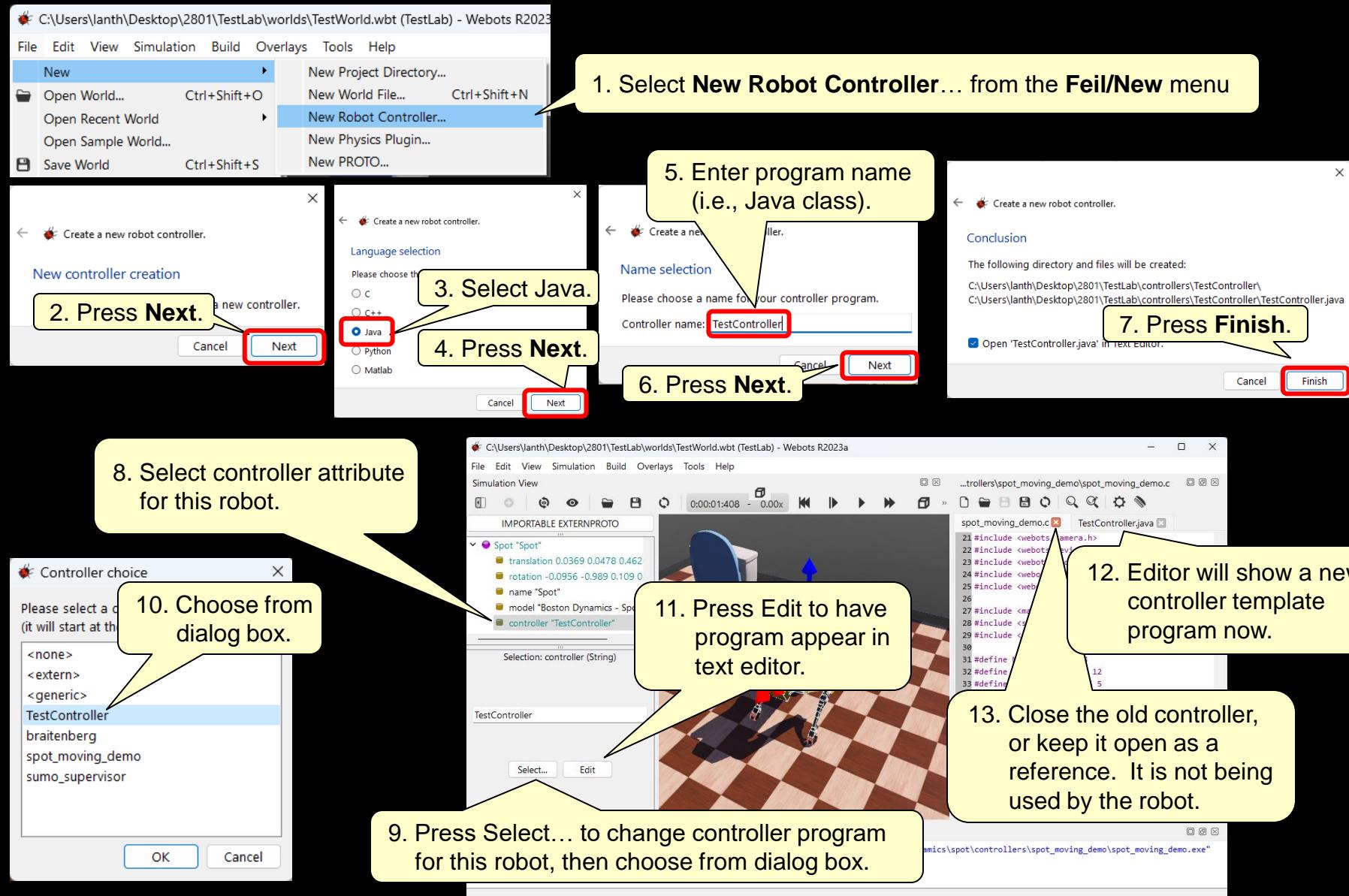
Don't forget to **SAVE** your world after adding your robot.

# Webots Interface – Controller Code

The screenshot shows the Webots R2023a interface. On the left, the 'Simulation View' panel displays a yellow Boston Dynamics Spot robot on a checkered floor, with a blue toilet nearby. A yellow speech bubble points to the 'controller' attribute in the object properties panel, stating: 'controller attribute specifies program that robot will run.' Another yellow speech bubble points to the 'controller' dropdown in the selection panel, stating: 'Click here to select a different program to run on the robot.' A third yellow speech bubble points to the code editor, stating: 'Click here to see the code in the editor.' On the right, the code editor window shows the C code for the 'spot\_moving\_demo'. The code includes includes for webots/camera.h, device.h, led.h, motor.h, and robot.h, along with definitions for NUMBER\_OF\_LEDS, NUMBER\_OF\_JOINTS, and NUMBER\_OF\_CAMERAS, and initializes the robot's information, motors, and cameras.

```
include <webots/camera.h>
22 #include <webots/device.h>
23 #include <webots/led.h>
24 #include <webots/motor.h>
25 #include <webots/robot.h>
26
27 #include <math.h>
28 #include <stdio.h>
29 #include <stdlib.h>
30
31 #define NUMBER_OF_LEDS 8
32 #define NUMBER_OF_JOINTS 12
33 #define NUMBER_OF_CAMERAS 5
34
35 // Initialize the robot's information
36 static WbDeviceTag motors[NUMBER_OF_JOINTS];
37 static const char *motor_names[NUMBER_OF_JOINTS] =
38     "front left shoulder abduction motor", "front le
39     "front right shoulder abduction motor", "front ri
40     "rear left shoulder abduction motor", "rear le
41     "rear right shoulder abduction motor", "rear rig
42 static WbDeviceTag cameras[NUMBER_OF_CAMERAS];
43 static const char *camera_names[NUMBER_OF_CAMERAS]
```

# Webots Interface – Your Controller



# Webots Interface – Editing Code

Press here (or use Cntrl-S) to save your code.

Press here to compile and load your code onto the robot.

Press here to create a new source code .java file.

Press here to open other source code .java files.

ALWAYS put your name and student number at the top of your programs.

Double-click on error line to go there in the editor.

Compile errors appear in the console window. If a compile error occurs, the code is NOT loaded onto the robot, so the robot is running its previous code. Once compiled and loaded, press **Reset** in the dialog box that appears. Press the play button in the simulation view to run it.,

```
File C:\Users\van...ds\TestWorld.wbt (TestLab) - Webots R2023a
C:\Users\van...op2001\testLab\controller
TestController.java X
Auth Mark Lanthier (SN: 100100100)
import com.webrobotics.webbots.controller.Robot;
...
public class TestController {
    public void main(String[] args) {
        // create the Robot instance.
        Robot robot = new Robot()

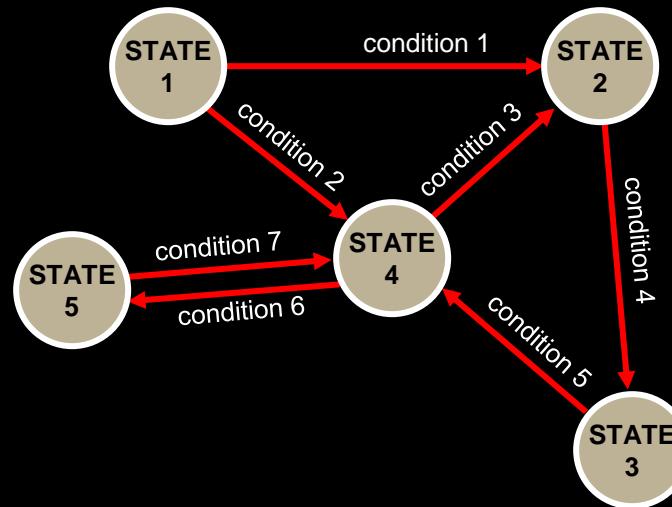
        // get the time step of the current world.
        int timeStep = (int) Math.round(robot.getBasicTimeStep());

        while (robot.step(timeStep) != -1) {
            // Sense
            // Think
            // React
        }
    }
}

Console - All
javac -Xlint -classpath C:\Program Files\Webots\lib\controller\java\Controller.jar;. TestController.java
TestController.java:11: error: ';' expected
    Robot robot = new Robot()
                           ^
1 error
Nothing to be done for build targets.
```

# State Machines

- A robot can be in various **states** at any time
  - e.g., STOPPED, MOVING\_FORWARD, SPINNING\_LEFT, etc...
- A **state machine** is a diagram that explains how a robot moves from one state to another.
  - Each state indicates what the robot is doing at any moment in time.
  - A robot **transitions** from one state to another based on a condition (which is often sensor input).
- We will be using state machines for the first 3 labs so make sure that you understand them.



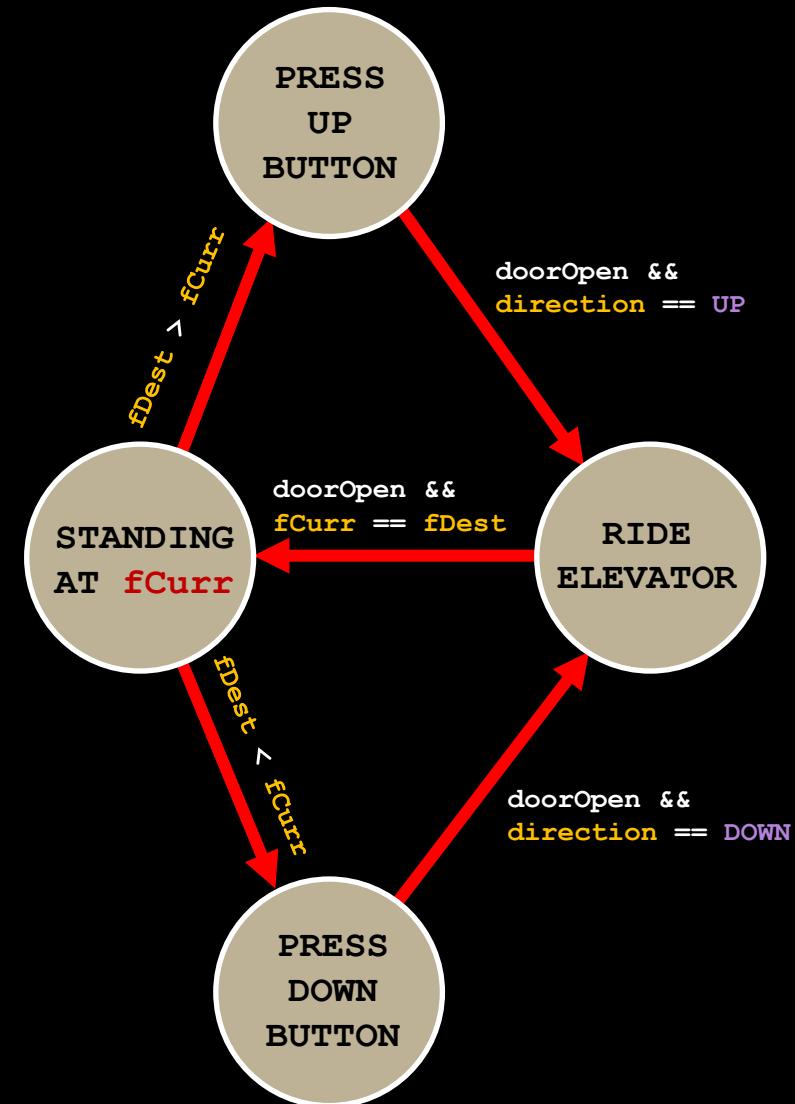
# State Machine - Elevator

- Here is an example of a state machine for using an elevator:

```
// Variables needed to transition
short fCurr = 0;
short fDest = 5;
byte direction = UP;
boolean doorOpen = false;

// Various states
static final byte STANDING_AT_FCURR = 0;
static final byte PRESS_UP_BUTTON = 1;
static final byte PRESS_DOWN_BUTTON = 2
static final byte RIDE_ELEVATOR = 3;

// Direction constants
static final byte UP = 0;
static final byte DOWN = 1;
```



# State Machine – Elevator (Code)

```
// Variables needed to transition
short fCurr; // current floor
short fDest; // destination floor
byte direction = UP; // UP or DOWN
boolean doorOpen = false;

// Various states
static final byte STANDING_AT_FCURR = 0;
static final byte PRESS_UP_BUTTON = 1;
static final byte PRESS_DOWN_BUTTON = 2;
static final byte RIDE_ELEVATOR = 3;

// Direction constants
static final byte UP = 0;
static final byte DOWN = 1;
```

Our SWITCH statements will be like nested IF statements.

The code in each case will be the code that does the action needed for that state (e.g., stop, walk, press, etc..)

```
// REACT
switch(state) {
    case STANDING_AT_FCURR :
        // stop moving
        break;
    case PRESS_UP_BUTTON :
        // reach out and press up button
        break;
    case PRESS_DOWN_BUTTON :
        // reach out and press down button
        break;
    case RIDE_ELEVATOR :
        // get in elevator
        // stop moving
        break;
}
```

```
byte state = STANDING_AT_FCURR;
fCurr = 0; // any floor
fDest = 12; // any floor

while(true) {
    // SENSE
    // THINK
    // REACT
}
```

// SENSE  
doorOpen = checkIfDoorOpen();  
direction = checkElevatorDir();

There will be one IF statement for each arrow in the state machine.

```
// THINK
switch(state) {
    case STANDING_AT_FCURR:
        if (fDest > fCurr)
            state = PRESS_UP_BUTTON;
        if (fDest < fCurr)
            state = PRESS_DOWN_BUTTON;
        break;
    case PRESS_UP_BUTTON:
        if (doorOpen && (direction == UP))
            state = RIDE_ELEVATOR;
        break;
    case PRESS_DOWN_BUTTON:
        if (doorOpen && (direction == DOWN))
            state = RIDE_ELEVATOR;
        break;
    case RIDE_ELEVATOR
        if (doorOpen && (fCurr == fDest))
            state = STANDING_AT_FCURR;
        break;
}
```

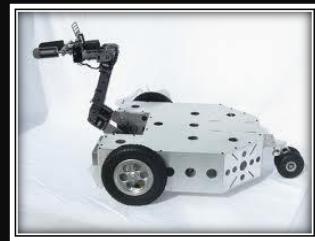
Decide on the new state for the next time through the WHILE loop

Start the  
Lab ...

# **Basic Movement and Sensing**

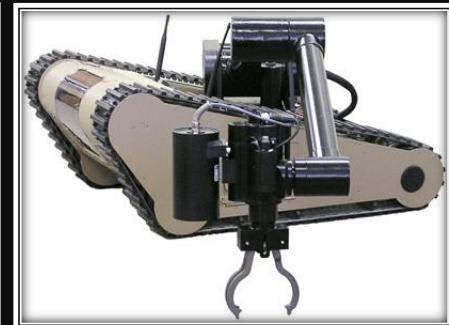
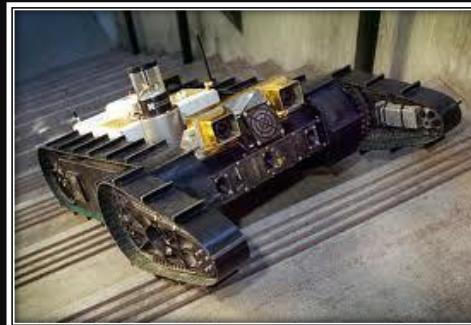
# Wheeled Movement

- 1-wheeled and 2-wheeled robots require balancing sensors.
- 3 wheeled robots usually have 2 drive wheels and one castor wheel for balance.
- 4 or more wheels requires a suspension system.



# Treaded Robots

- Treaded robots have better traction.
- Can also increase stability while allowing smoother spins.
- Can be used to climb over obstacles such as stairs.



# Wheeled Robot Concerns

---

- When designing a wheeled robot, various concerns must be addressed:

- Stability

- Robot can topple over depending on its wheel geometry
    - Low center of gravity helps with stability

- Traction (i.e., Friction)

- Wheel slippage (due to low friction surfaces or steep incline) can lead to positioning errors.

- Maneuverability

- May not be able to turn sharp enough.
    - May not be easy to maneuver through rough or soft terrain.

- Control

- Configuration may not allow sufficient control of robot's speed, causing overshoot or collision.



# Wheel Designs

## ■ Standard

- often used for drive
- sometimes used for steering



## ■ Castor

- used for balancing, not controlled
- problems when changing direction

## ■ Ball

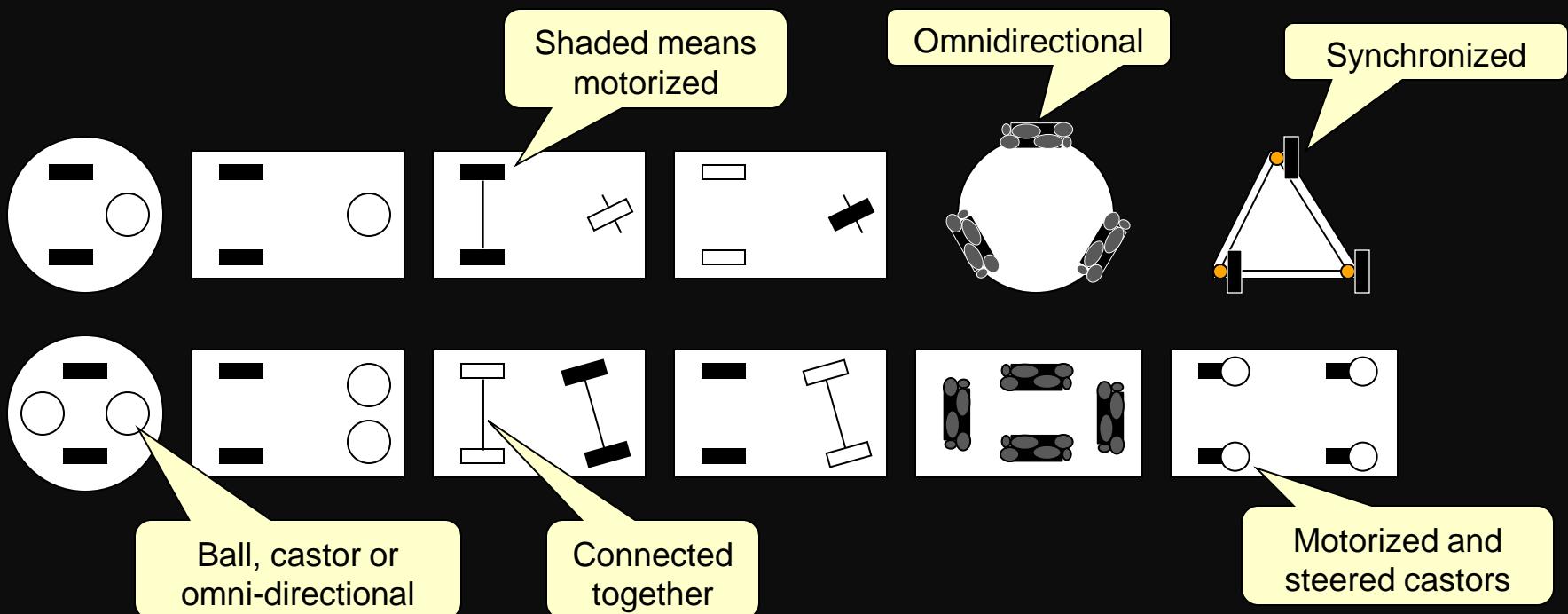
- no direction change problem
- used for balancing, not controlled

## ■ Omni-Directional

- Less friction in “side” directions

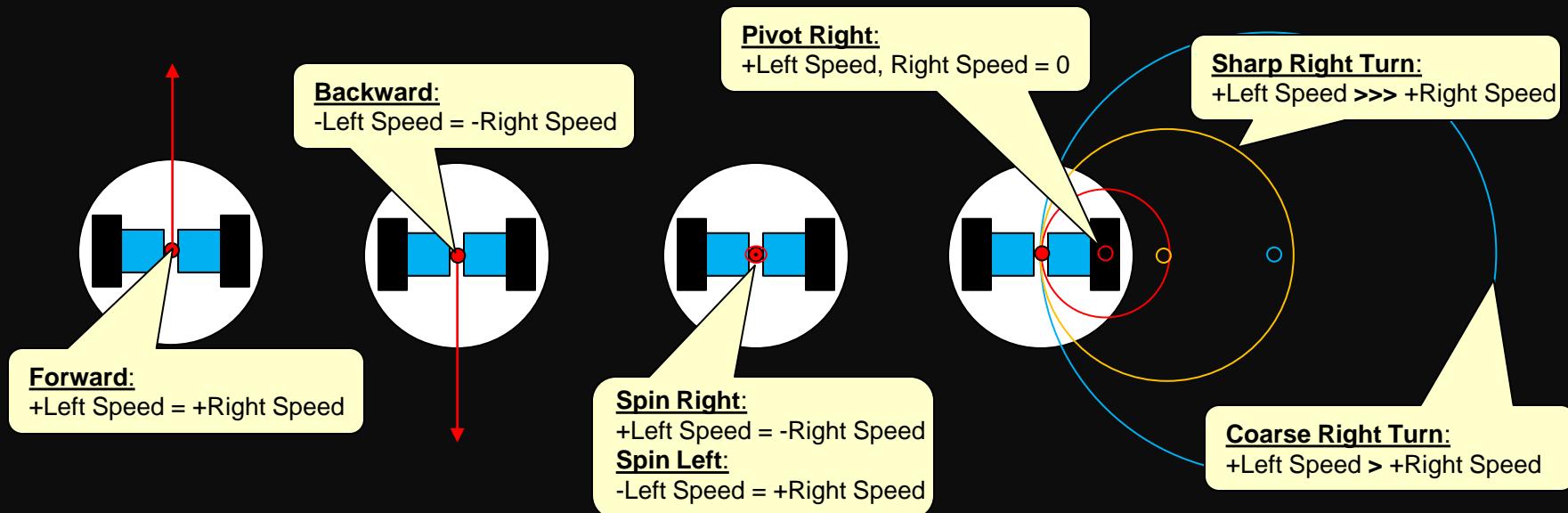
# Wheel Geometry

- Choice of wheels depends on where they are placed on the robot
- Choosing geometry depends on where robot will be used



# Differential Steering Robots

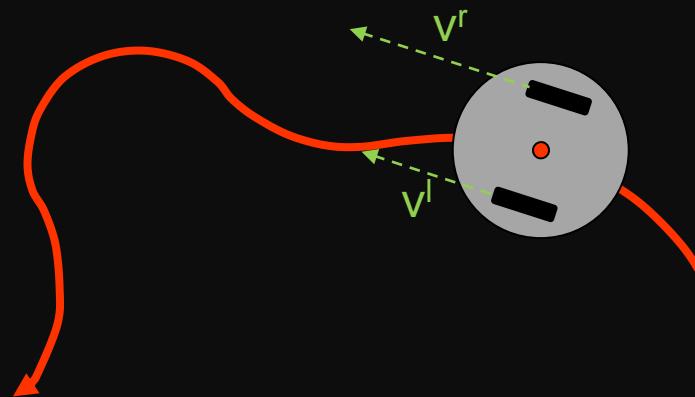
- Robots with two-wheels use *differential steering*  
(i.e., more drive torque is applied to one side of the vehicle than the other side).
- They are simple and easy to maneuver:



# Differential Steering Robots

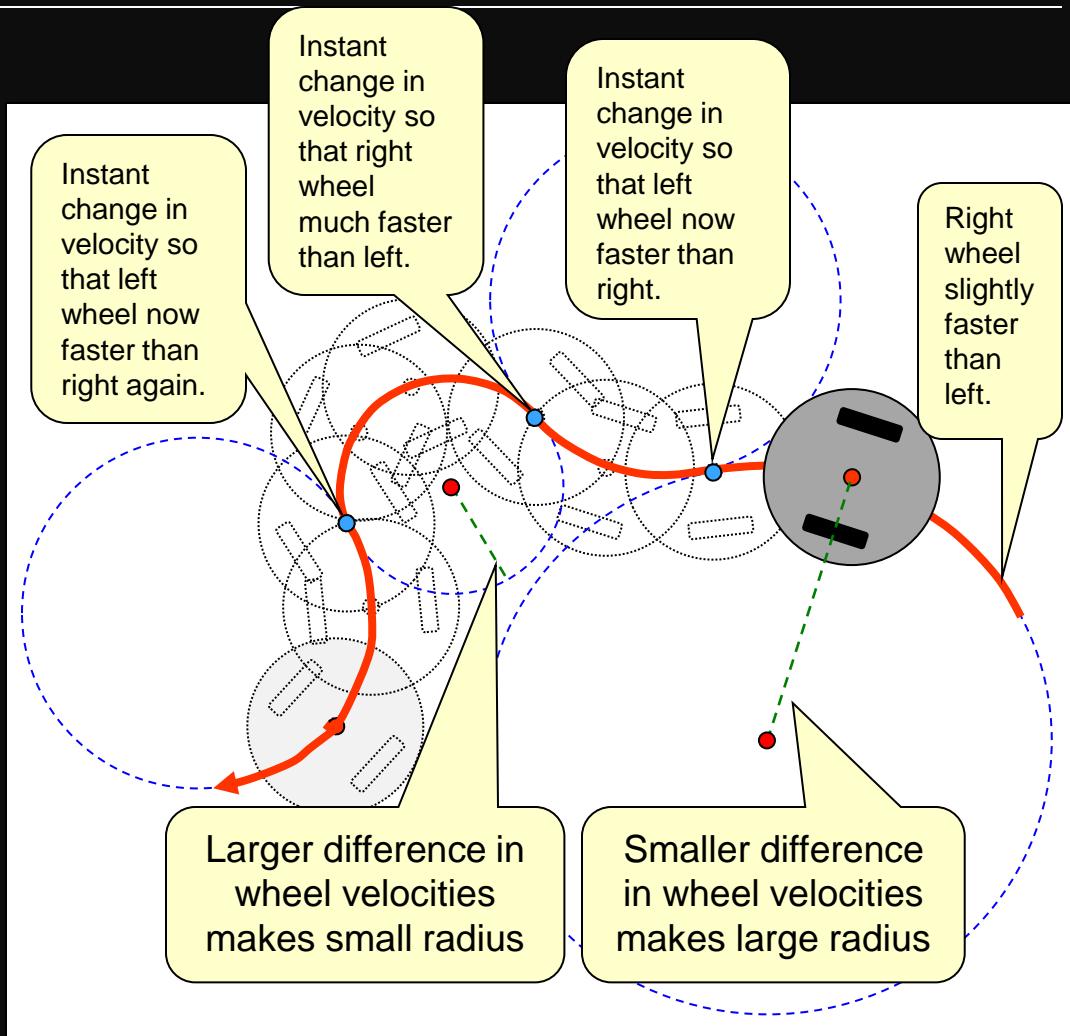
---

- At any instance in time both left and right wheels have their own velocities  $v^l$  and  $v^r$
- Robot forms curves in its workspace depending on these velocities.
- Can produce virtually any desired path since steering can spin on the spot.



# Differential Steering Robots

- As velocity stays constant, the robot path traces out a circle.
- When the velocity changes, the circle changes.
- Any curve can be formed as long as the velocities are known at all times.



# GCtronic e-puck: Motors

- Robot uses two-motor differential steering:

```
import com.cyberbotics.webots.controller.Motor;

static final double MAX_SPEED = 6.28; // maximum speed of the e-puck robot

// Motors are objects
Motor leftMotor, rightMotor;

// Set up the motors
leftMotor = robot.getMotor("left wheel motor");
rightMotor = robot.getMotor("right wheel motor");
leftMotor.setPosition(Double.POSITIVE_INFINITY);
rightMotor.setPosition(Double.POSITIVE_INFINITY);

// Set the motors to 100% of maximum speed
int leftSpeed = 1.0 * MAX_SPEED;
int rightSpeed = 1.0 * MAX_SPEED;

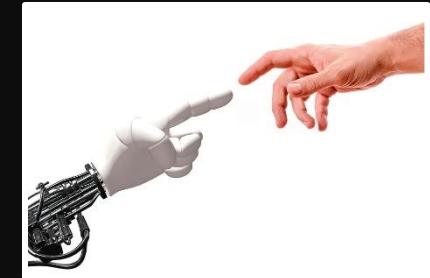
// Make the motors move, if speed > 0
leftMotor.setVelocity(leftSpeed);
rightMotor.setVelocity(rightSpeed);
```

leftSpeed	rightSpeed	RESULT
0	0	STOPPED
+S	+S	FORWARD
-S	-S	BACKWARD
+S	-S	SPIN RIGHT
-S	+S	SPIN LEFT
+S	+T, S>T	CURVE RIGHT
+S	+T, T>S	CURVE LEFT
+S	0	PIVOT RIGHT
0	+S	PIVOT LEFT



# Proximity Sensors

- A **Proximity Sensor** is a sensor that detects the presence of an object within some fixed distance from the sensor.
- Provides a binary “yes/no” reading indicating that an object is either “within range” or “out of range”.
  - **Tactile** – uses physical contact to determine if anything is within a close proximity (e.g., bumpers and whiskers).
  - **Non-Tactile** – sends out an active signal that is received back if object is detected (e.g., sonar sensors, Infrared sensors).
- The **detection range** is defined as the maximum distance that the sensor can detect an object.



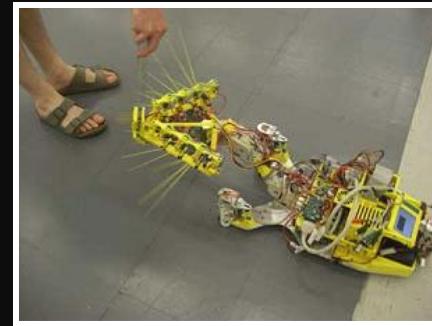
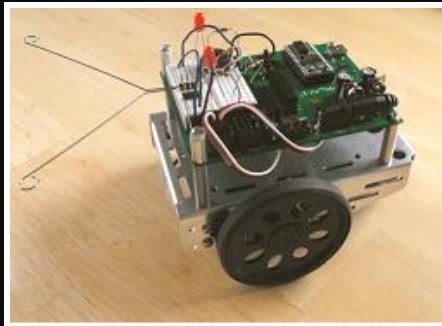
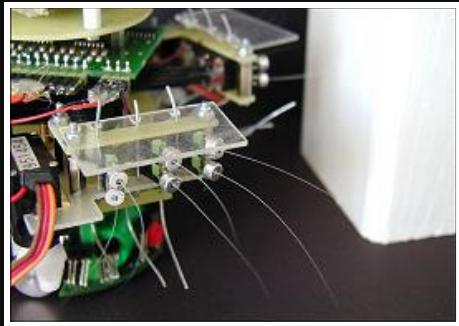
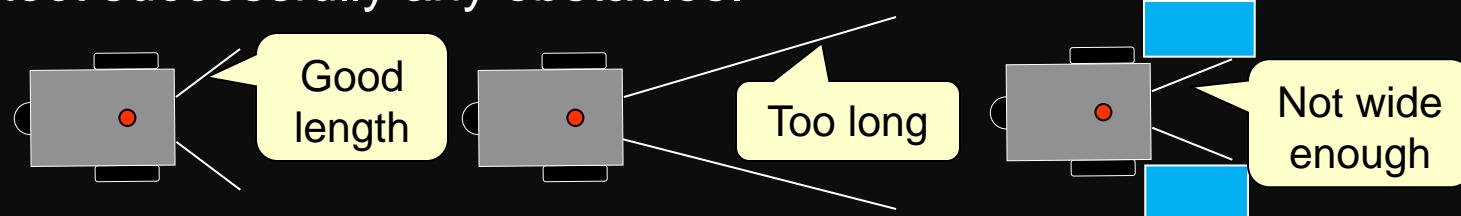
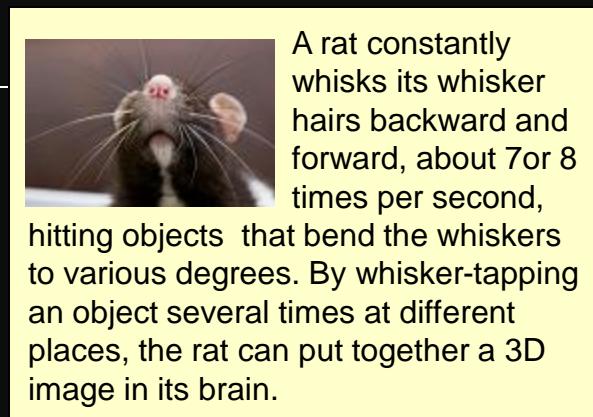
# Tactile: Bumpers

- Bumpers are simple, but they have a short detection range from from 1<sub>mm</sub> to 2<sub>cm</sub>.
- Unfortunately, they require physical contact with the object to detect it:
  - Can cause damage to robot depending on speed
  - Bumpers can break over time
  - Objects can be pushed or damaged



# Tactile: Whiskers

- Whiskers are like flexible bumpers
  - Usually placed at front and extend long enough to ensure safe stopping distance.
  - Should extend the entire body width so as to detect successfully any obstacles.

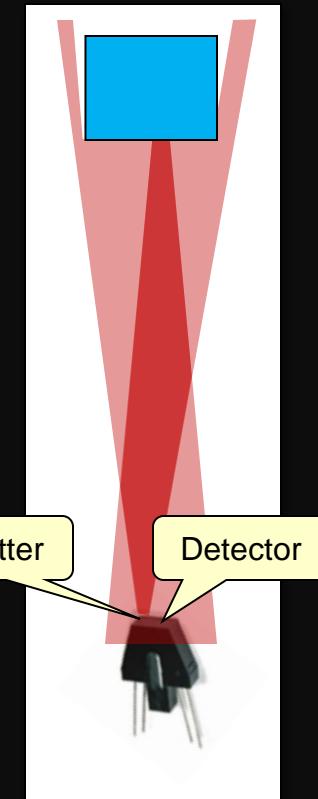


(0:17) <http://www.youtube.com/watch?v=HwC2tZcOKM8>

(0:47) [http://www.youtube.com/watch?v=GTekO\\_RQCzE&feature=player\\_embedded](http://www.youtube.com/watch?v=GTekO_RQCzE&feature=player_embedded)

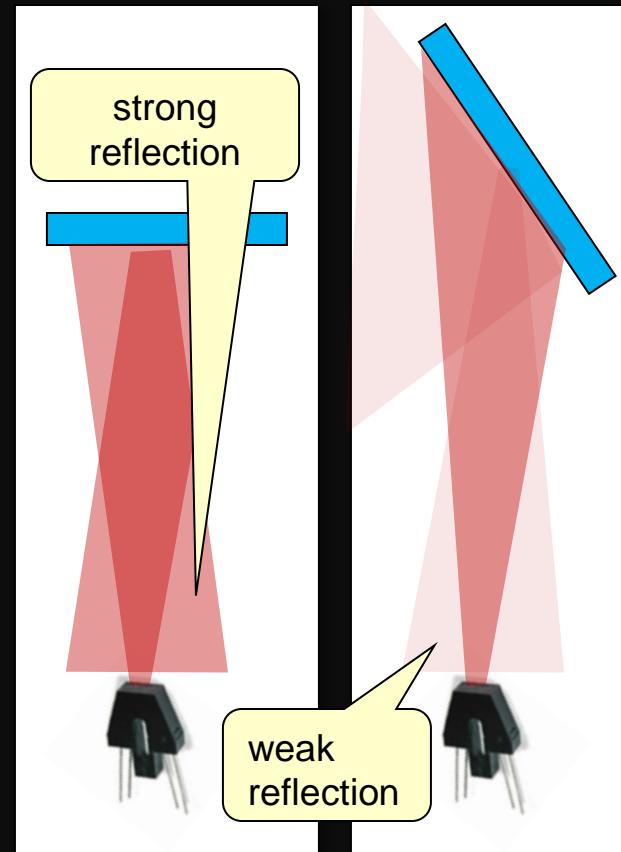
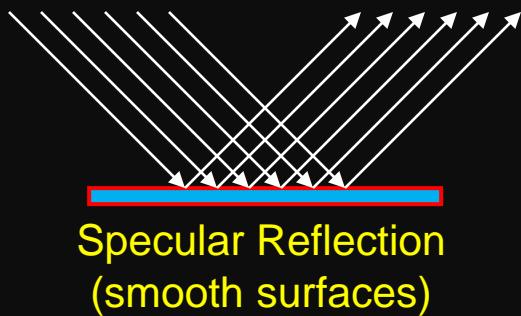
# Non-Tactile: Reflective IR Sensors

- IR proximity detection is simple:
  - Turn on an IR diode (i.e., light)
  - Light is reflected off obstacle, some light returns
  - Receiver measures strength of light returned.
- Range reading is highly dependent on the reflective characteristics of the object:
  - shiny obstacles (e.g., metal) reflect a lot of light
  - rough surfaces (e.g., thick cloth) do not reflect well
  - white/black surfaces report different ranges
  - cannot detect glass, since light shines through it



# Non-Tactile: Reflective Issues

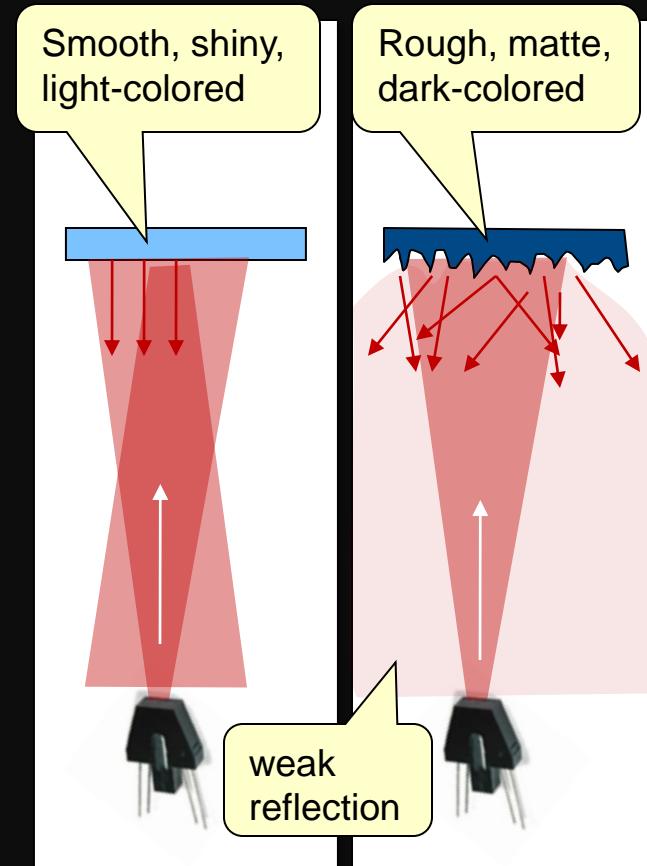
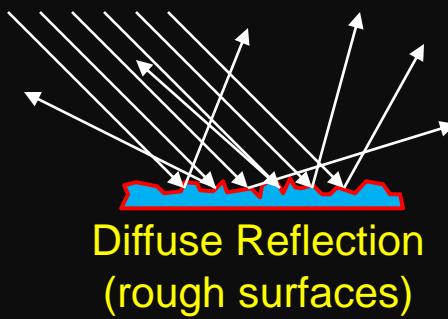
- IR is sensitive to obstacle angle
  - can result in improper detection.
- When beam's angle of incidence falls below a certain critical angle ***specular reflection*** occurs.
  - Object may not be detected



# Non-Tactile: Reflective Issues

- IR is sensitive to obstacle surface
  - can result in improper detection.

- When beam hits rough surfaces (e.g., cloth or carpet), less light is reflected back since *diffuse reflection* occurs.

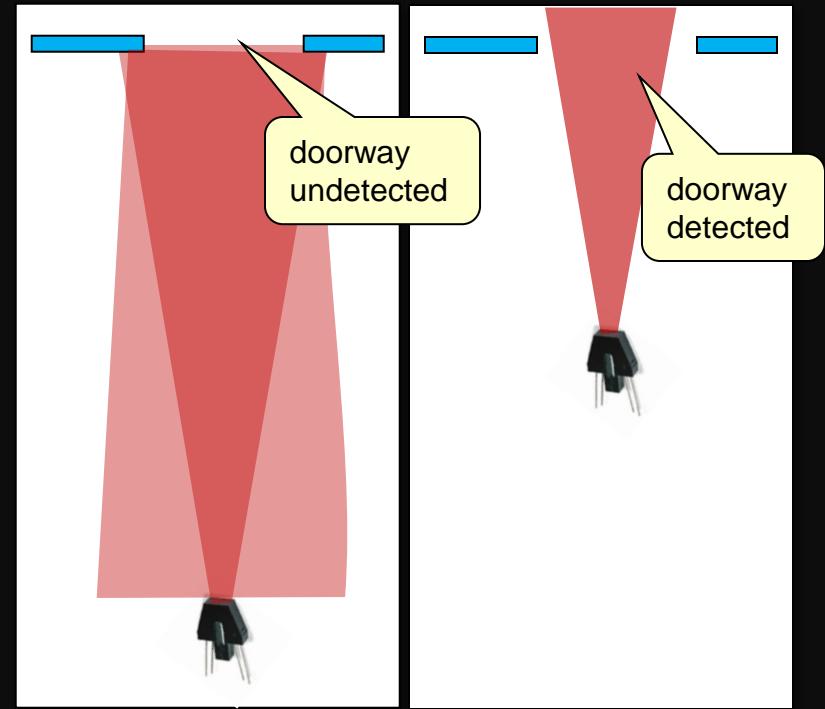
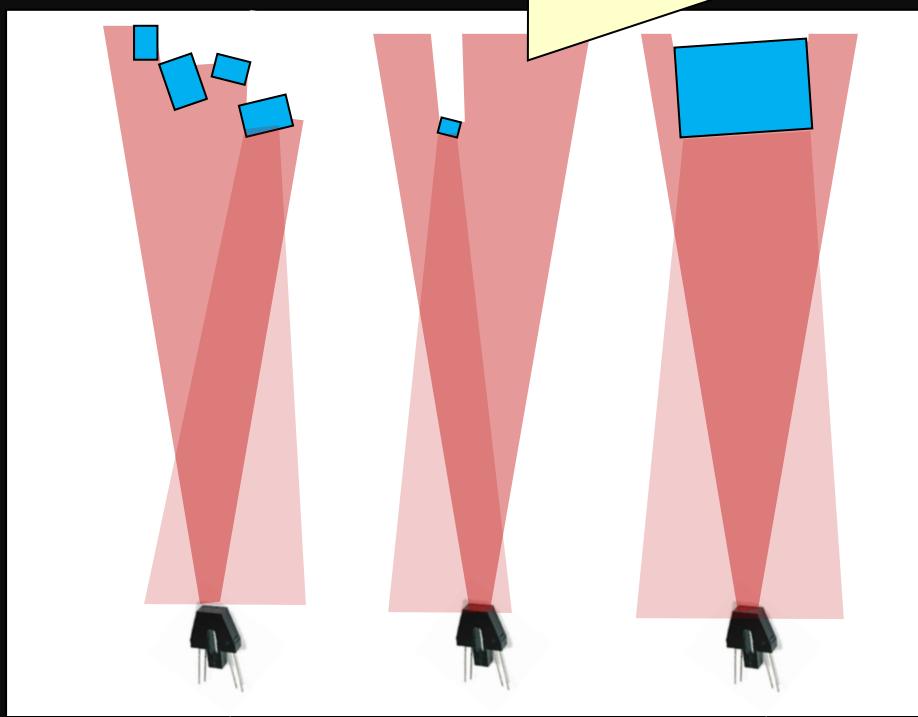


- Color also affects amount of light reflected (e.g., white reflects more than black).

# Non-Tactile: Reflective Issues

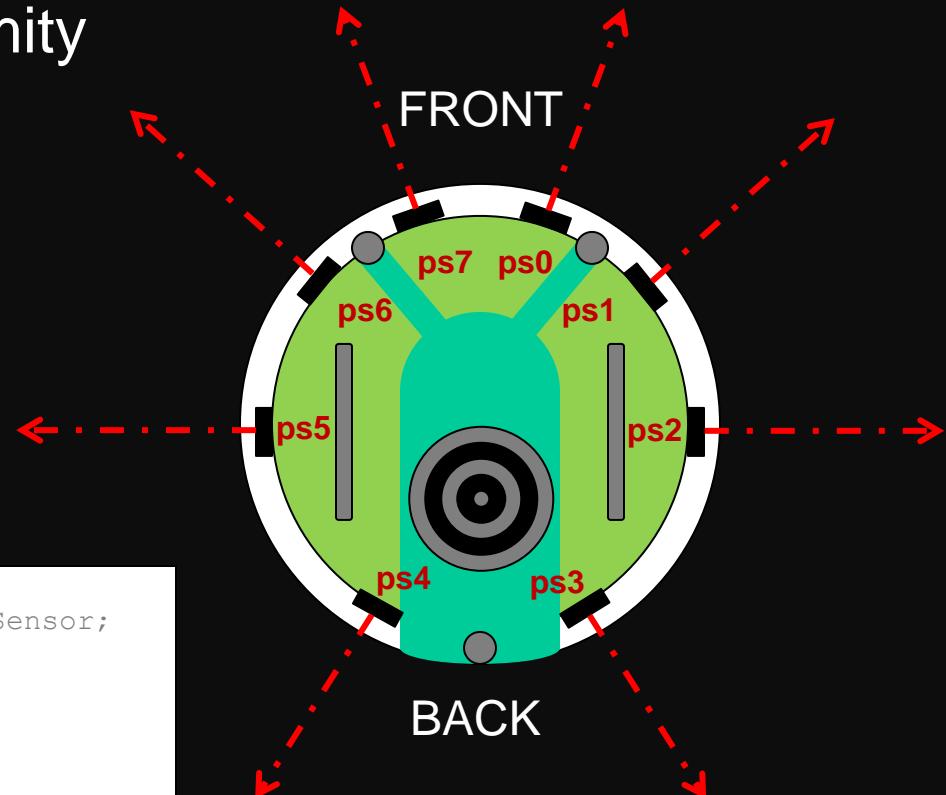
- Beam width can also play a role in obstacle detection:
  - multiple close obstacles cannot be distinguished
  - gaps cannot be detected (e.g., doorways)

Cannot distinguish between these three scenarios.



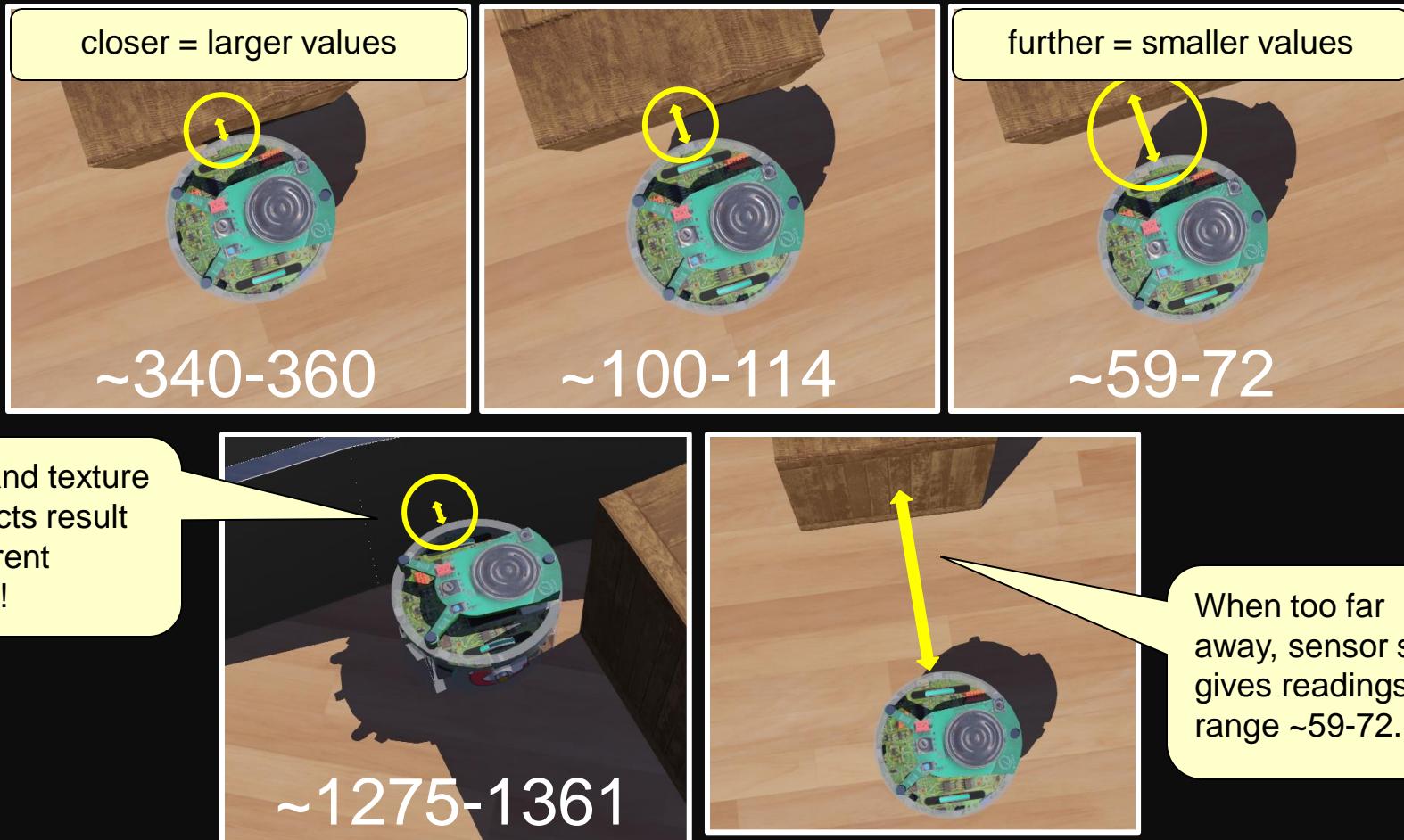
# GCtronic e-puck: IR sensors

- The E-Puck robot has 8 proximity sensors around the outer ring.



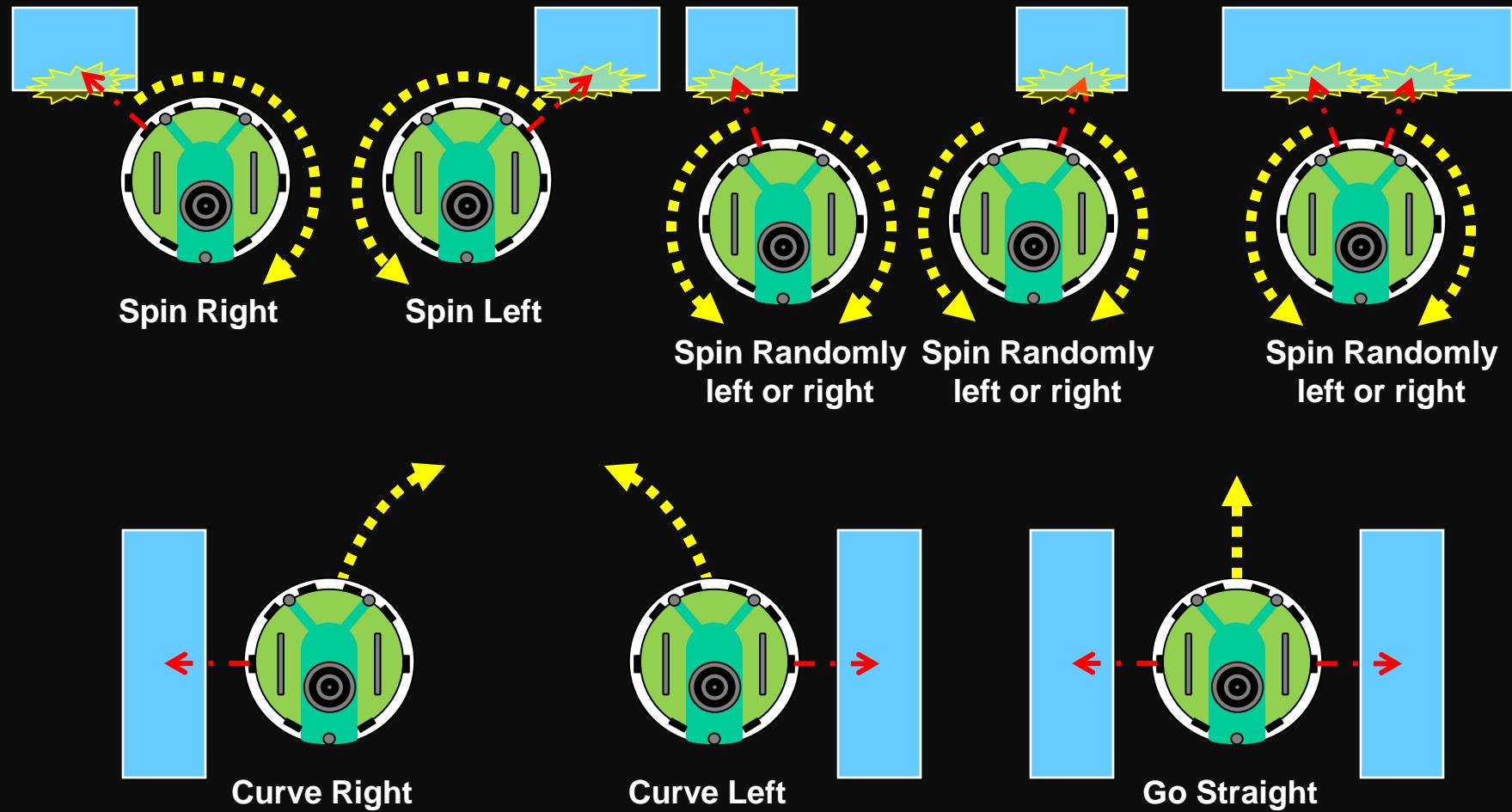
```
import com.cyberbotics.webots.controller.DistanceSensor;  
  
// Sensors are objects  
DistanceSensor sensor7;  
  
// Set up the sensor to be used  
sensor7 = robot.getDistanceSensor("ps7");  
sensor7.enable(timeStep);  
  
// Read the sensor value  
double reading = sensor7.getValue();
```

# GCtronic e-puck: IR sensors



Anything less than 80 seems to indicate that “no object is detected”

# Collision Detection



# Collision Avoidance: Problem

- In corners and tight spaces, the robot may end up oscillating back and forth.

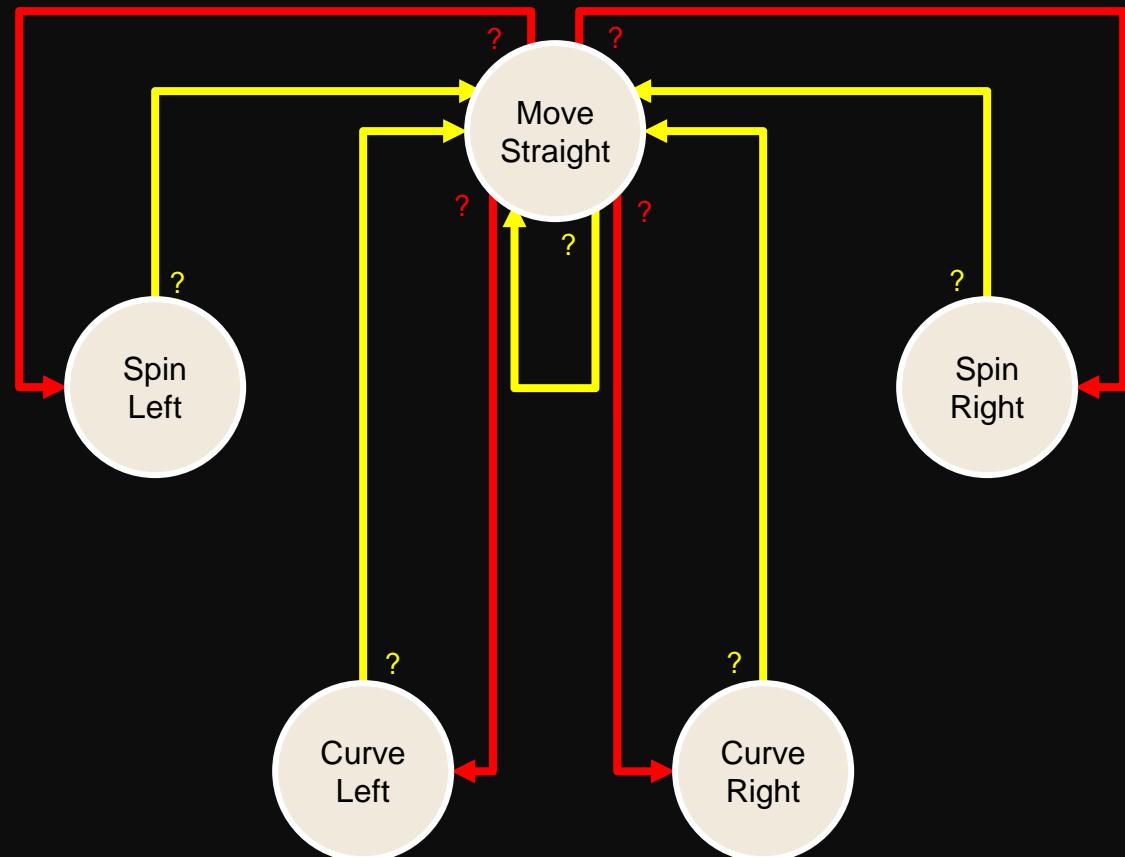
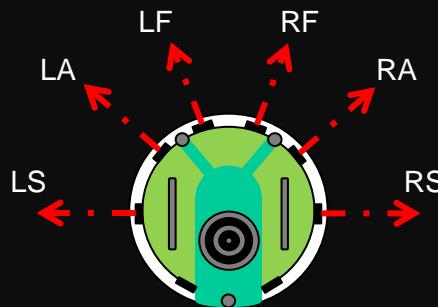


- Solution: Commit to a direction and “stick with it”!!

# Collision Avoidance: Solution

- Need a state machine based on sensor values:

- 5 states robot can be in at any time.
- You decide when it should leave one state and go to another, based on sensor values at any moment in time.



# State Machine Code Structure

- States can be represented as numbers and state machine as SWITCH statement:

```
// States represented as unique #'s
static final byte STRAIGHT = 0;
static final byte SPIN_LEFT = 1;
static final byte SPIN_RIGHT = 2;
static final byte CURVE_LEFT = 3;
static final byte CURVE_RIGHT = 4;

byte state = // some start state
while(robot.step(timeStep) != -1) {
    // SENSE: Read all the sensors
```

```
    // THINK
```

```
    // REACT
```

Each time through the loop the robot moves one timestep. The loop is necessary to make the robot keep moving indefinitely.

Check sensors, then make a decision as to which mode to change to for the next round of the loop.

```
// THINK: Look at the sensors
// and based on the "current"
// state, decide what the
// "next" state should be
switch(state) {
    case SPIN_LEFT:
        // decide on next state
        break;
    case SPIN_RIGHT:
        // decide on next state
        break;
    case CURVE_LEFT:
        // decide on next state
        break;
    case CURVE_RIGHT
        // decide on next state
        break;
    default:
        // decide on next state
        break;
}
```

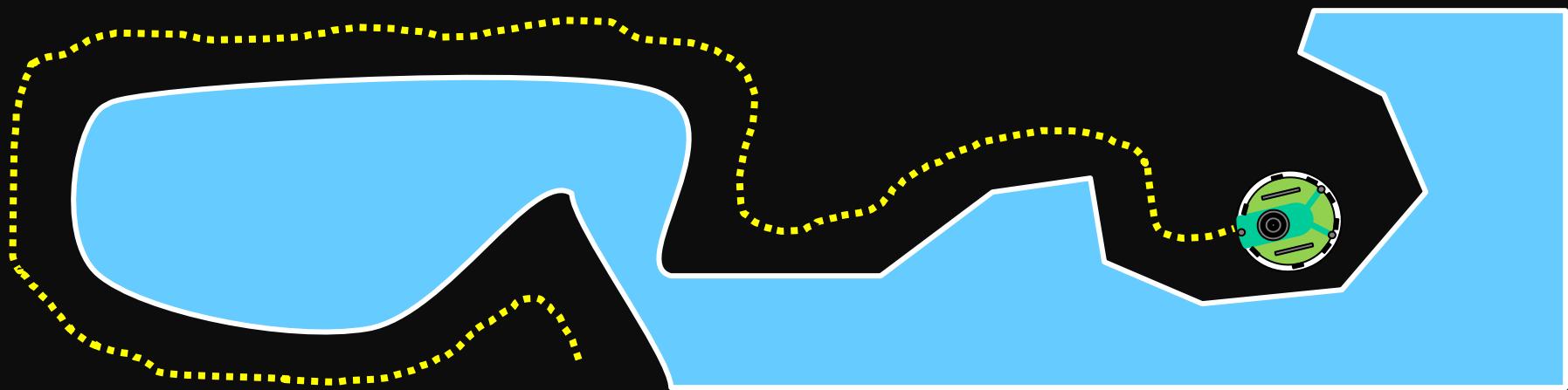
```
// REACT: Move motors by
// setting their speed to what
// the "current" state
// requires
switch(state) {
    case SPIN_LEFT:
        // Set motor speeds ...
        break;
    case SPIN_RIGHT:
        // Set motor speeds ...
        break;
    case CURVE_LEFT:
        // Set motor speeds ...
        break;
    case CURVE_RIGHT:
        // Set motor speeds ...
        break;
    default:
        // Set motor speeds ...
        break;
}
```

Start the  
Lab ...

# Wall-Following

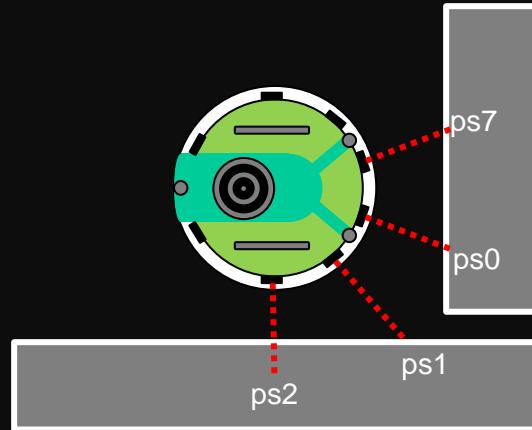
# Wall Following

- Wall following behavior is useful for mapping, navigation, seeking wall outlets, performing cleaning tasks etc...
- Strategy varies depending on types of sensors.
- Robot usually follows wall by keeping itself aligned to the wall on its left or right side



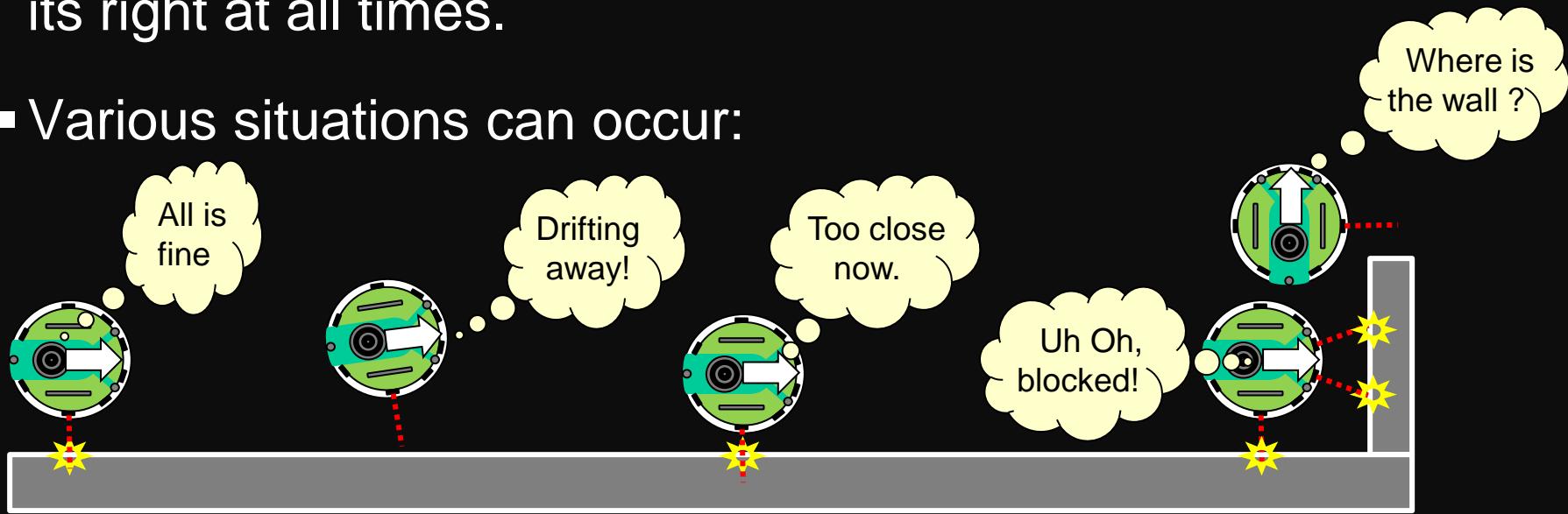
# Wall Following

- At any time, robot simply moves forward or turns right or left depending on the shape of the contour that it is following.
- We will consider right-handed wall following only
- Consider the e-puck robot using 4 sensors as follows:



# Wall Following

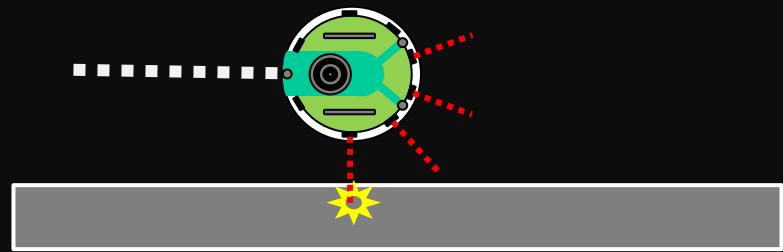
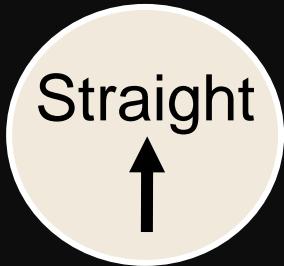
- Robot will try to maintain the same distance from the wall on its right at all times.
- Various situations can occur:



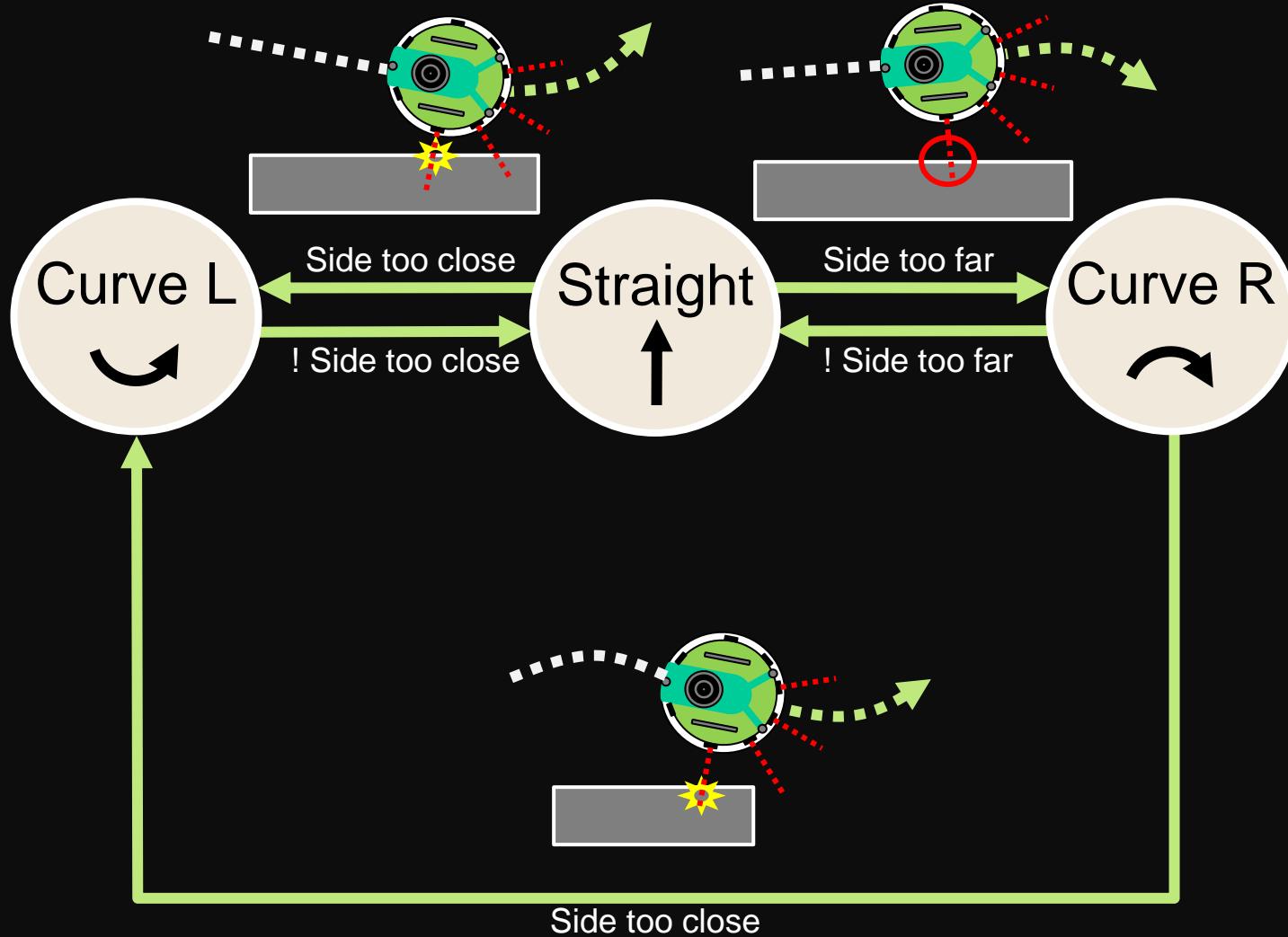
- Robot will thus be in certain *modes* to cause it to travel ahead or make appropriate turns to re-align with the wall or orient itself to a new edge.
- Can use a state machine to do this...

# Wall Following – State Machine

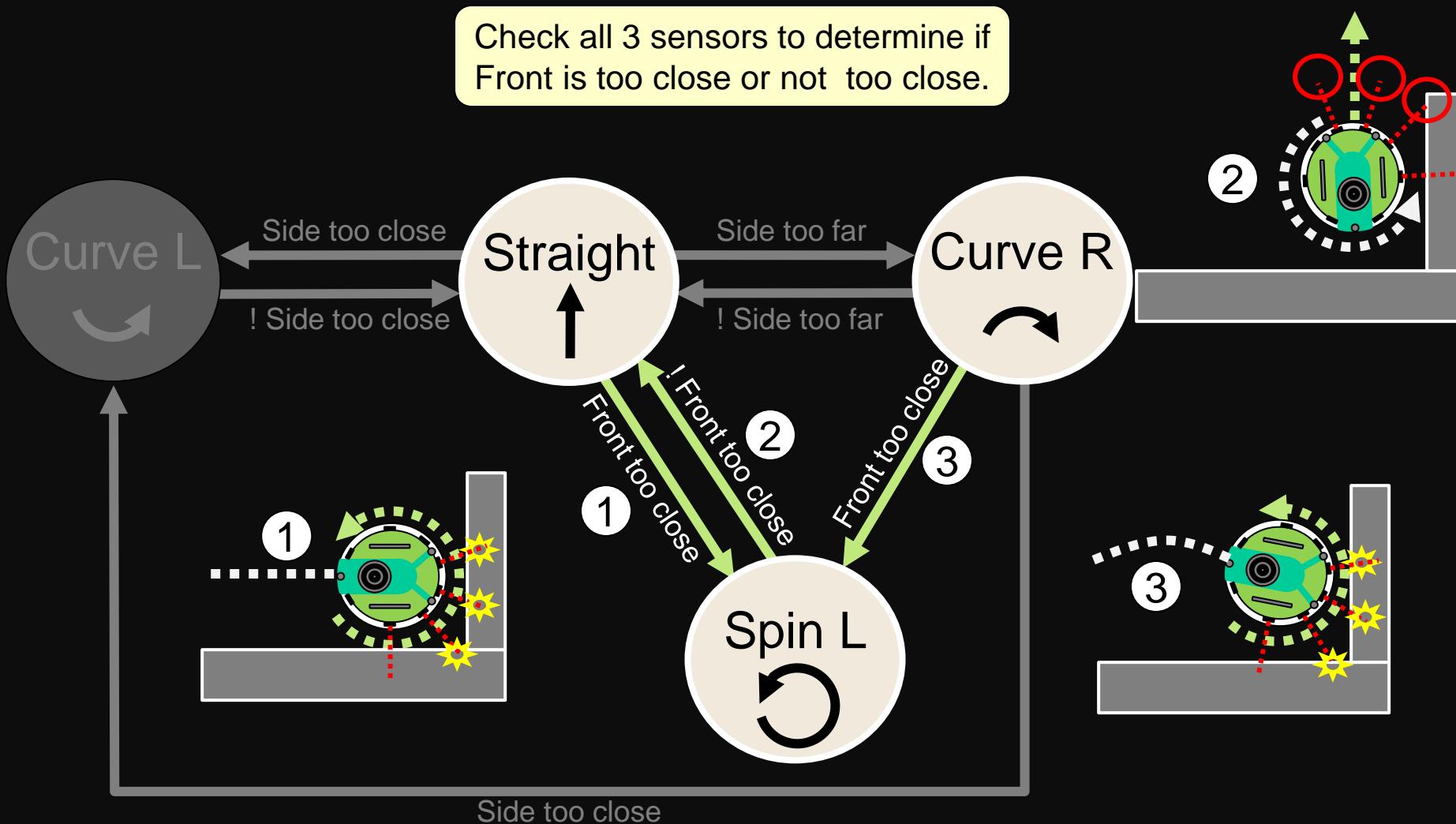
---



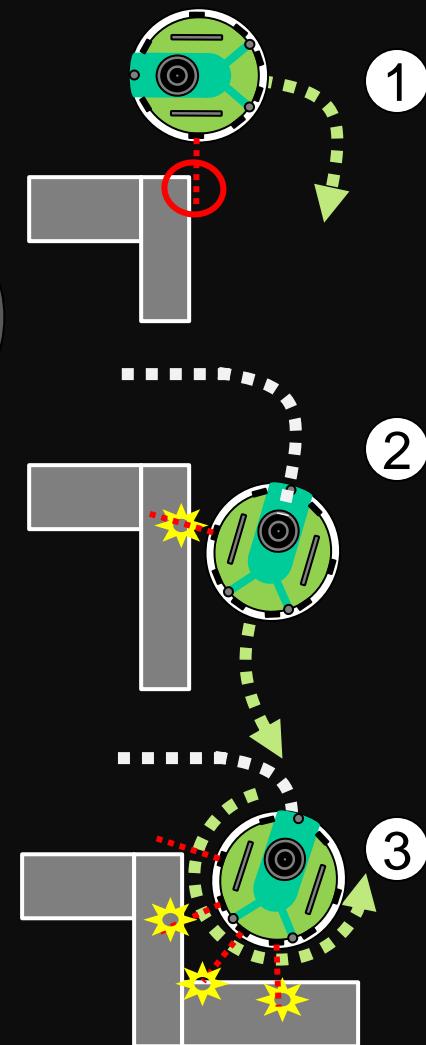
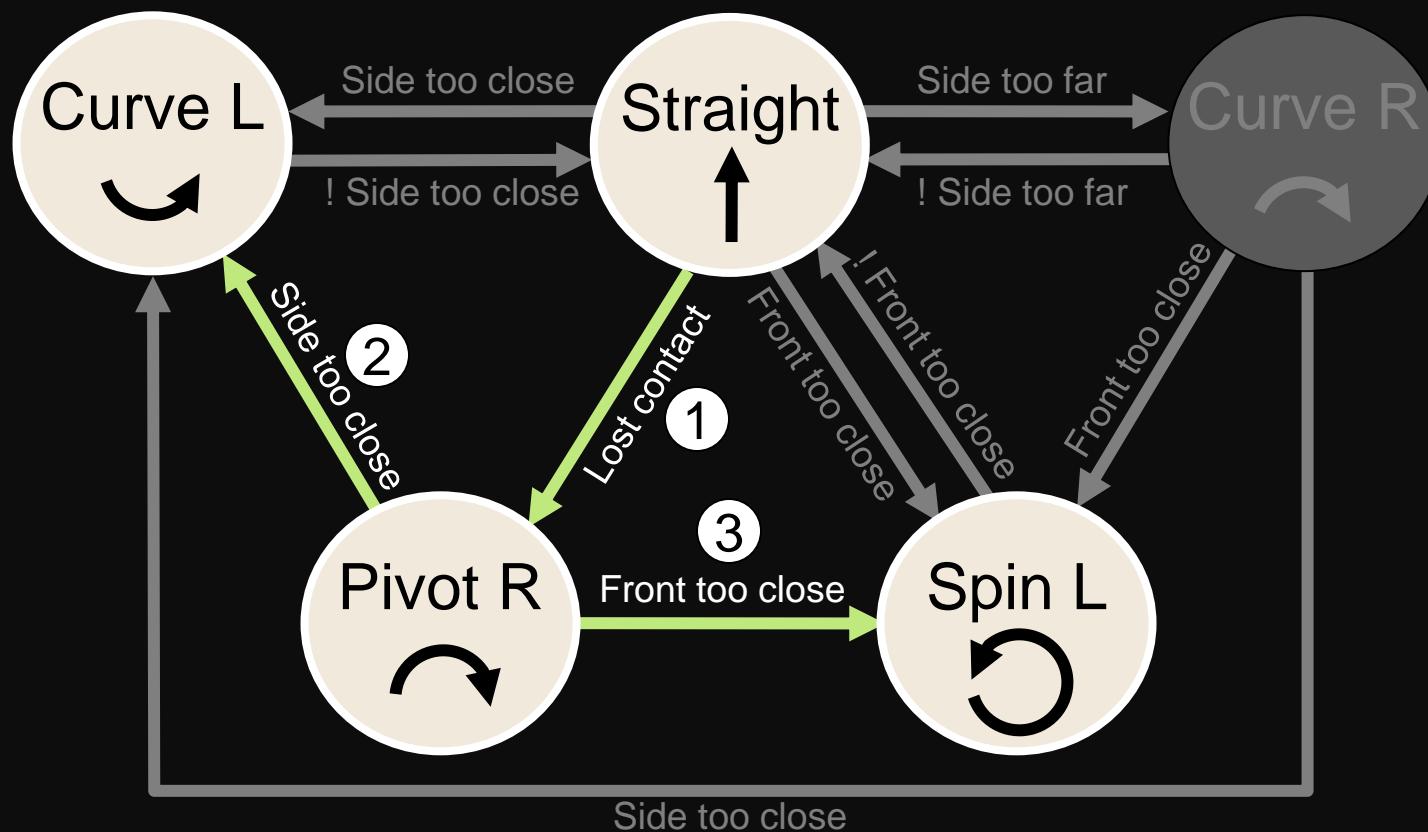
# Wall Following – State Machine



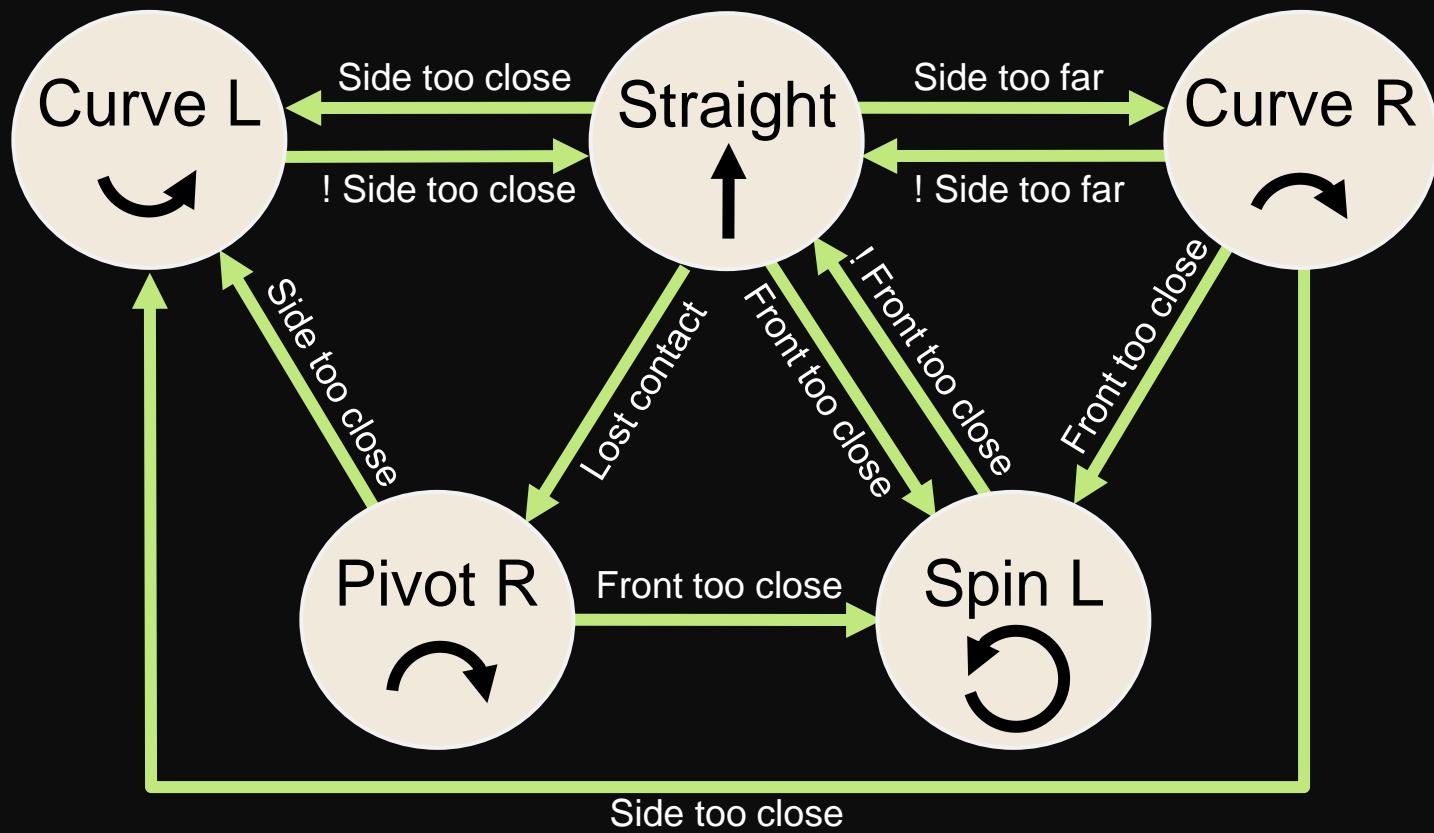
# Wall Following – State Machine



# Wall Following – State Machine



# Wall Following – Completed



# Wall Following – Code Structure

```
static final byte STRAIGHT = 0;
static final byte SPIN_LEFT = 1;
static final byte PIVOT_RIGHT = 2;
static final byte CURVE_LEFT = 3;
static final byte CURVE_RIGHT = 4;

byte currentMode = STRAIGHT;

while(robot.step(timeStep) != -1) {
    switch(currentMode) {
        case STRAIGHT:
            // ... check sensors and make decision to change mode, decide on move to make ...
            break;
        case CURVE_LEFT:
            // ... check sensors and make decision to change mode, decide on move to make ...
            break;
        .
        .
        .
        case PIVOT_RIGHT:
            // ... check sensors and make decision to change mode, decide on move to make ...
            break;
    }
    // ... move the motors right or left accordingly
}
```

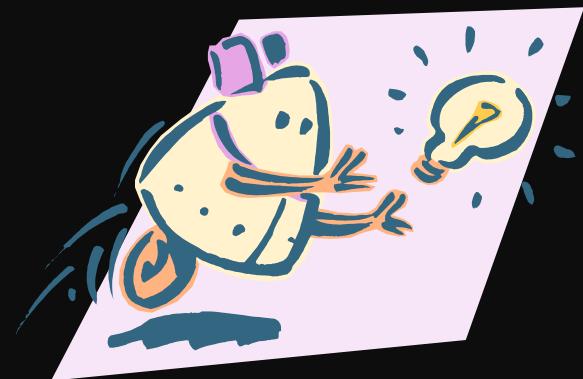
Start the  
Lab ...

# Homing and Tracking

# Homing Behavior

---

- **Homing** involves orienting or directing homeward to a destination
- **Taxis:** A steering toward or away from some directional stimulus or a gradient of stimulus intensity. (E.g., seeking out light, temperature, energy etc..)
- Used to orient the robot towards or away from something progressively.
- There are three main types:
  1. **Klinotaxis**
  2. **Tropotaxis**
  3. **Menotaxis**

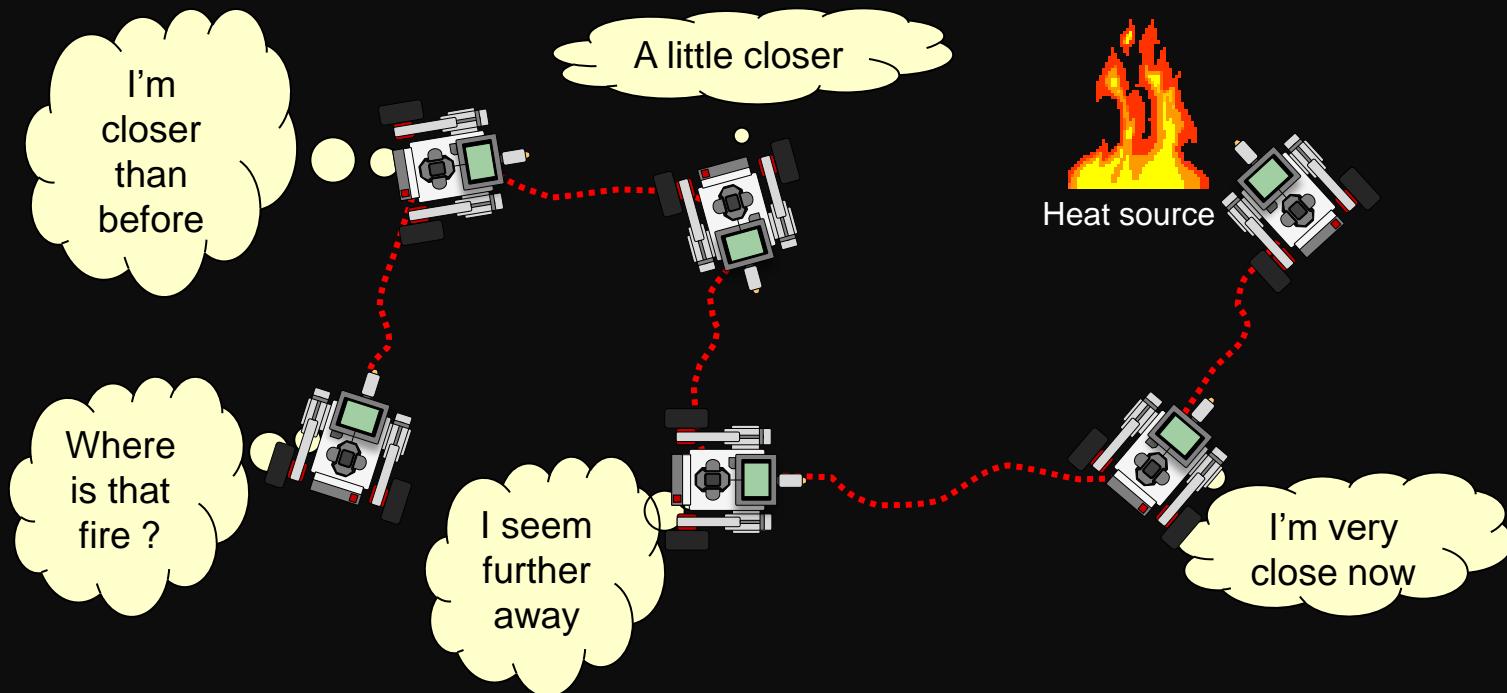


# Homing Behavior

## 1. Klinotaxis:

*Taking sensor readings at various locations in sequence in order to head towards a stimulus*

e.g., Temperature sensing or “sniffing” out chemicals

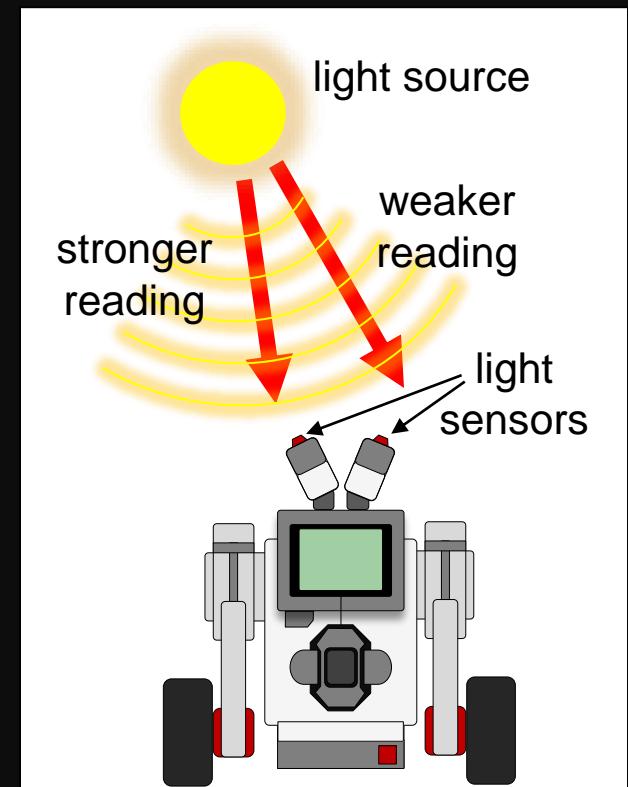
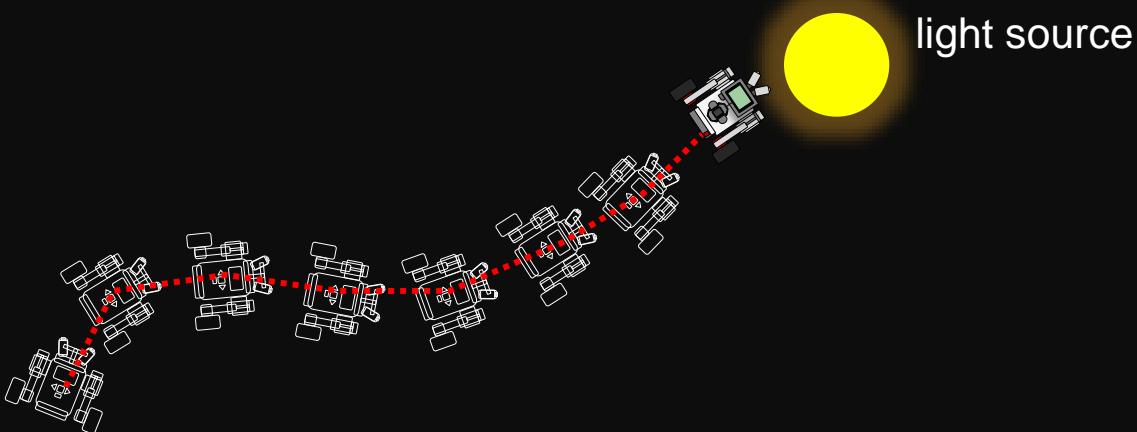


# Homing Behavior

## 2. Tropotaxis:

*Using the difference between two similar sensors to determine the direction of a certain stimulus*

e.g., seeking out a light source



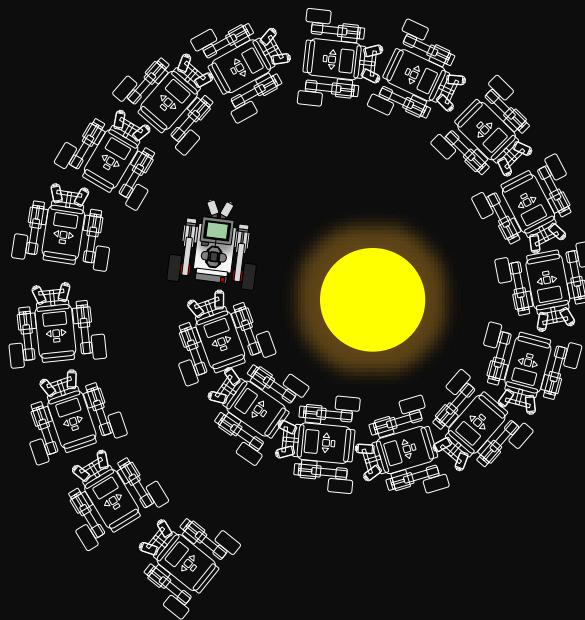
# Homing Behavior

---

## 3. Menotaxis:

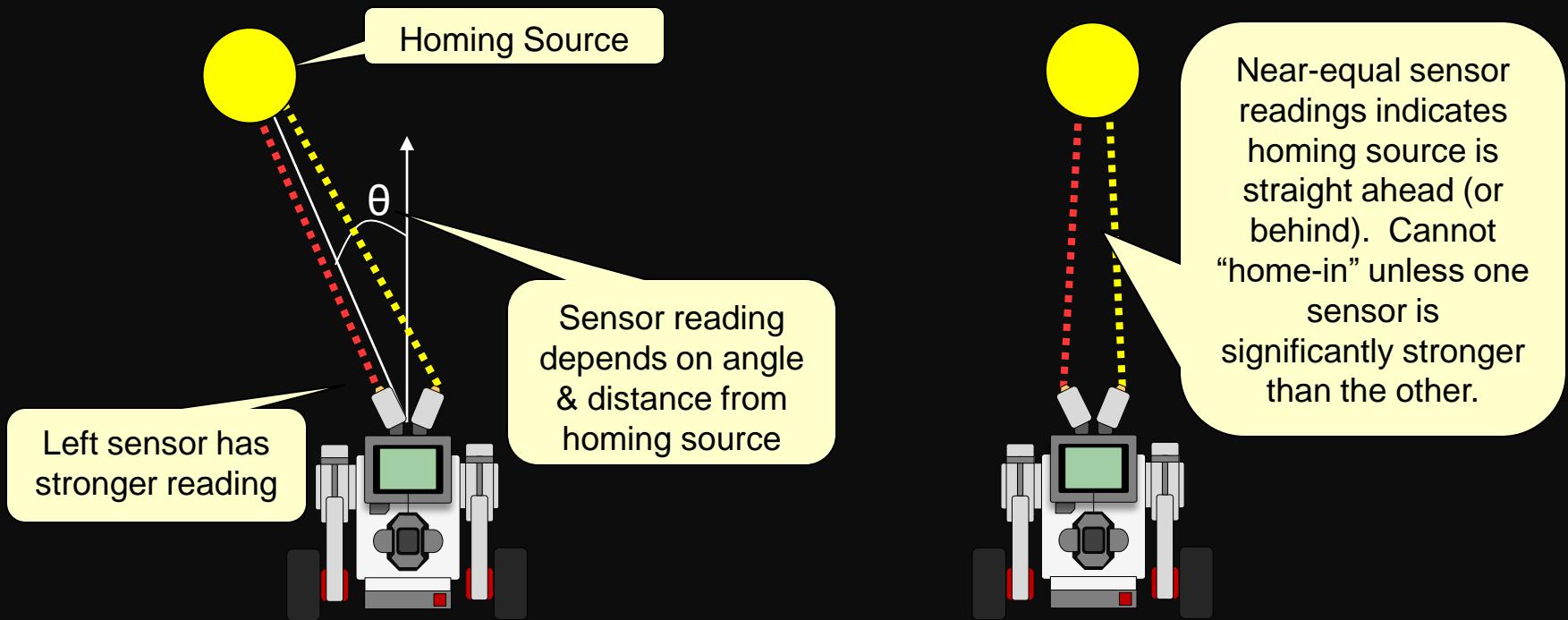
*Maintaining a fixed angle between the path of motion and the direction of the sensed stimulus*

e.g., spiraling around a light source



# Tropotaxis Homing Logic

- Easiest with 2 sensors whose readings increase when pointed towards homing source (tropotaxis):

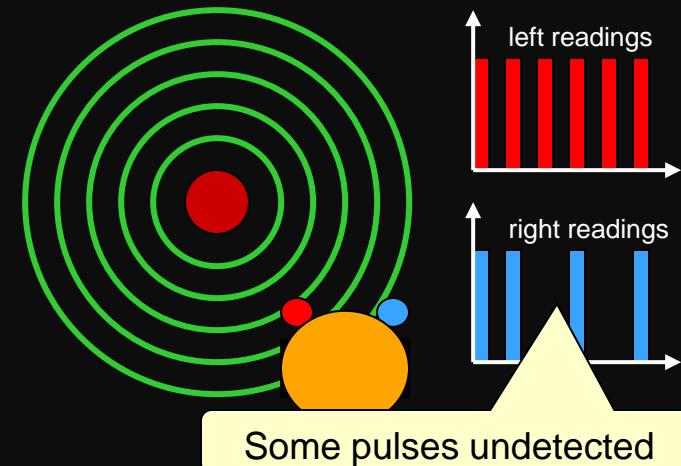


# Other Types of Homing

- Other forms of homing-in:

- Beacon Following

- Beacon emits pulsed signal which is more reliably detected by closer sensor

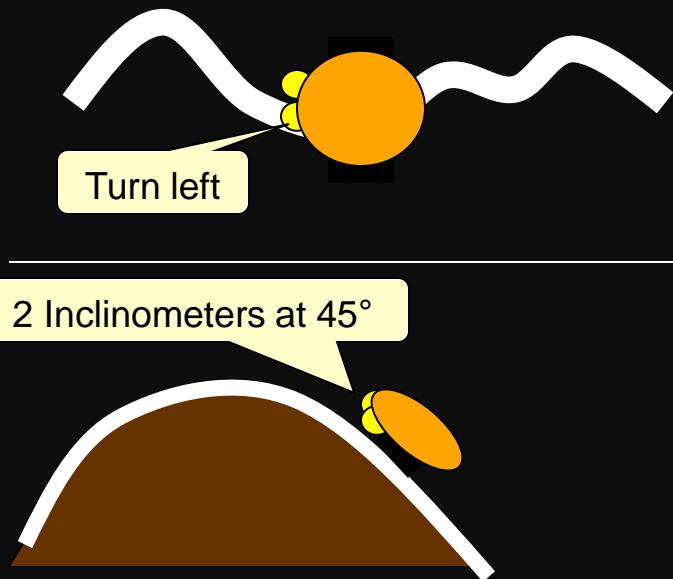


- Line Following

- Photodetectors read stronger on white, can detect when a sensor leaves black line

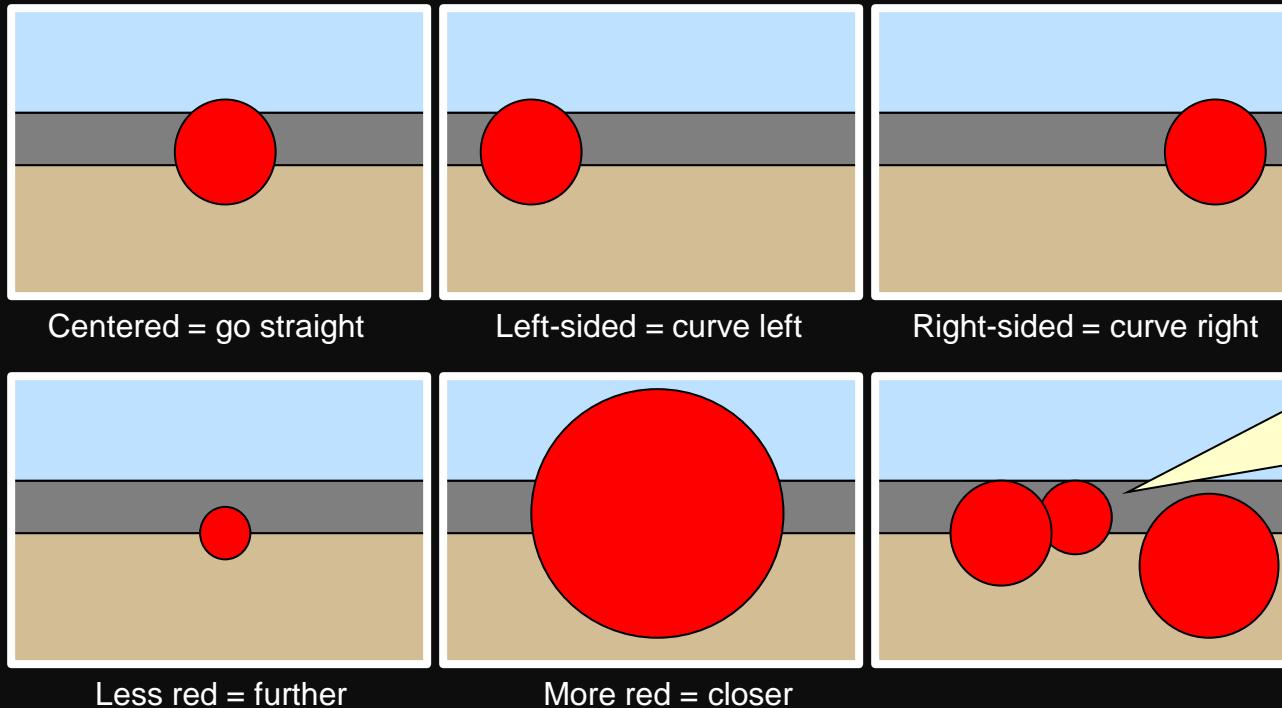
- Hill Climbing

- Climb hill by minimizing roll while keeping pitch positive using inclinometers



# Camera Tracking

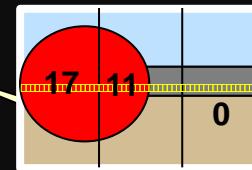
- Cameras are often used to track objects and can be used to find, locate and “home-in” on them.
- Can look for colored “blobs” and make decisions based on their location.



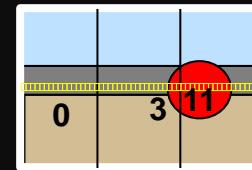
# Basic Tracking: Checking Pixels

- Grab a single row of pixels in the center of the image and count the **amount of red** in each of 3 zones:

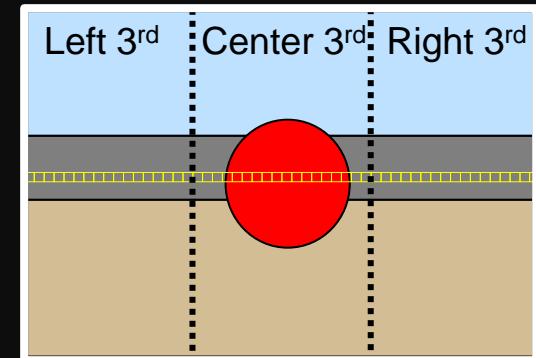
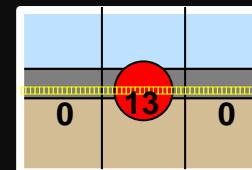
```
If (leftCount > (rightCount + ε))  
    then the object is on  
    the left side
```



```
If (rightCount > (leftCount + ε))  
    then the object is on  
    the right side
```



```
If (centerCount > ε)  
    then there is red ahead
```

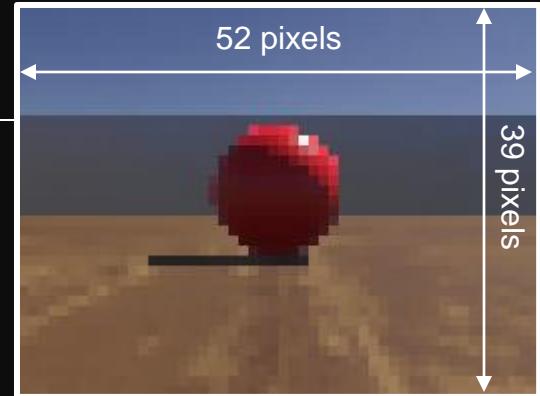


Allow for a small epsilon  $\epsilon$  so that robot doesn't zig-zag back and forth due to minor noise fluctuations.

- If none of above are true, then no object is detected

# E-puck Camera

- The e-puck robot has a color camera that can be used for processing images in a simple manner:

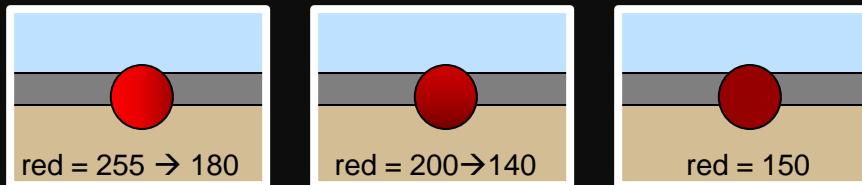


```
import com.cyberbotics.webots.controller.Camera;  
  
// Cameras are objects  
Camera camera;  
  
// Set up the camera  
camera = new Camera("camera");  
camera.enable(timeStep);  
  
// WHILE LOOP {  
  
    // Need to capture an image ... comes back as  
    // a 1D array with rows one after the other  
    int[] image = camera.getImage();  
  
    // Can get the red, green and blue value of a pixel at position (x, y) in the array  
    // where (0,0) is at the top left of the image  
    int r = Camera.imageGetRed(image, 52, x, y); // 52 is image width  
    int g = Camera.imageGetGreen(image, 52, x, y);  
    int b = Camera.imageGetBlue(image, 52, x, y);  
}  
// }
```

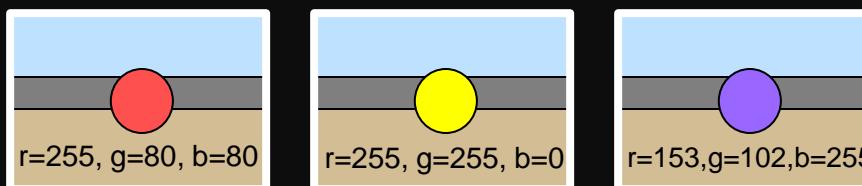
This code MUST be in the main **while** loop, otherwise a runtime exception will occur.

# Camera Colors

- Objects will have shadows... will affect the red color value:



- Also, other colors have red in them:



- So, need to check all 3 color components to decide if red:

```
if ((red > 60) && (green < 100) && (blue < 100)) ...
```

# Acceleration

- Acceleration is the rate of change in velocity
  - Measured in (meters per second) per second. (i.e.,  $\text{m/s}^2$ )
- A “g” is a unit of acceleration equal to the earth’s gravity at sea level:

$$g = 9.81 \text{ m/s}^2 \text{ (or } 32.2 \text{ ft/s}^2\text{)}$$

**1g** - Earth’s gravity

**2g** - Passenger car when taking a corner

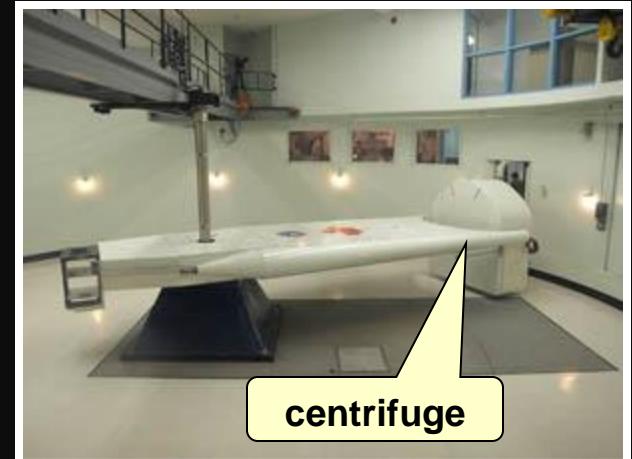
**2g** - Bumps in the road

**3g** - Indy car driver when taking a corner

**5g** - Bobsled rider when taking a corner

**7g** - Unconsciousness

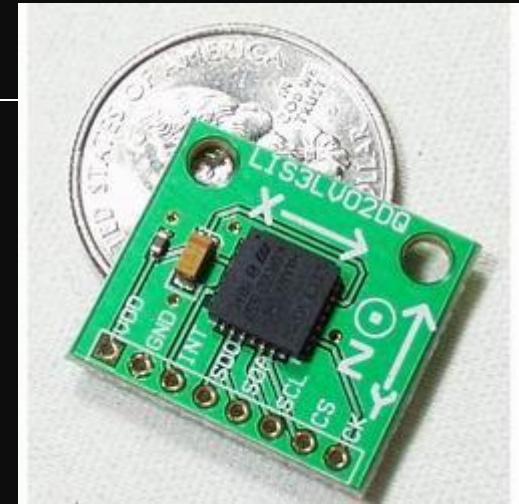
**10g** - Space shuttle



# Accelerometers

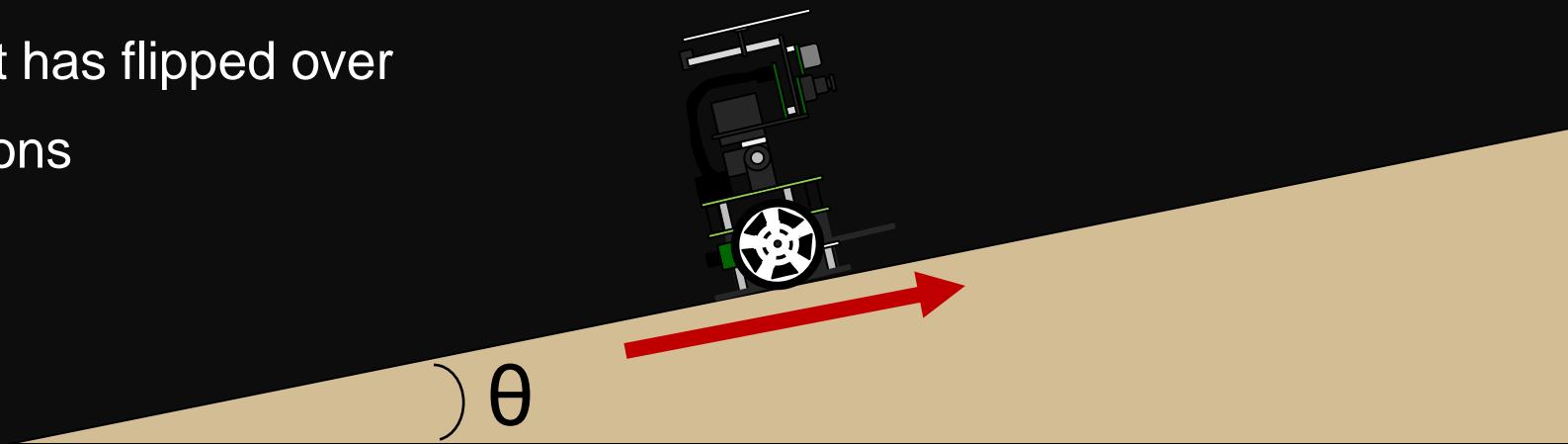
- An accelerometer can measure:

- Static acceleration forces
  - (e.g., force of gravity)
- Dynamic acceleration forces
  - (e.g., vibrations of the device)



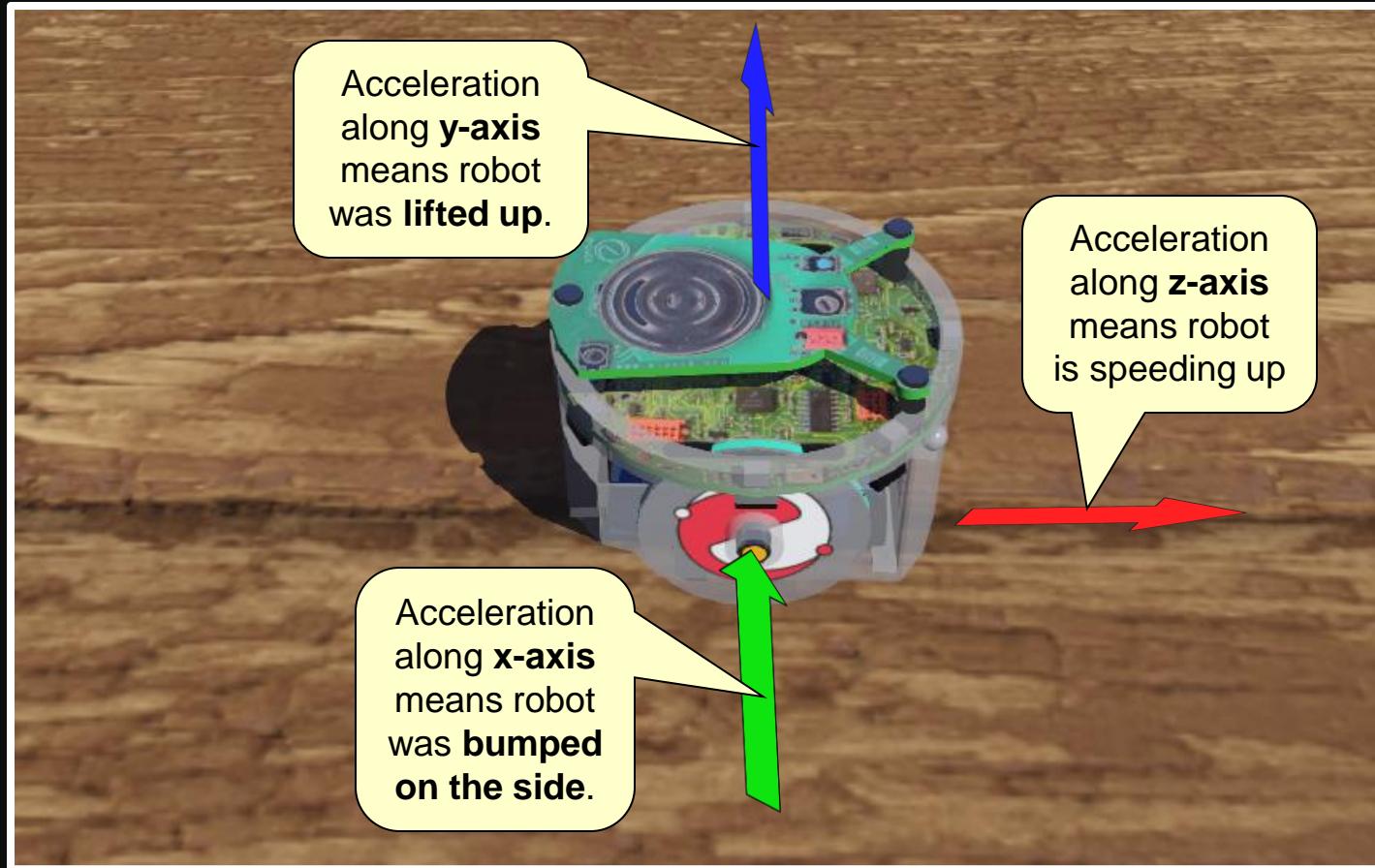
- Can determine:

- angle of incline
- if robot has flipped over
- vibrations



# E-Puck Acceleration Detection

- As sensor moves, it detects acceleration (i.e., change in speed) in one of the three axis directions.



# E-Puck Accelerometer Angles

(0.0, 0.0, 9.8)



(0.0, 0.0, 9.8)



(0.0, 0.0, 9.8)

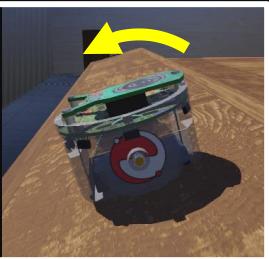
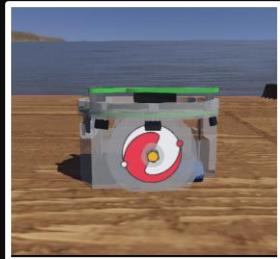


(0.0, 0.0, 9.8)

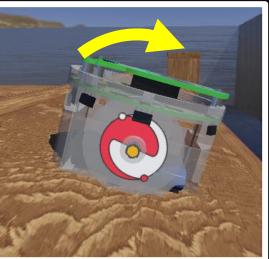


(0.0, 0.0, 9.8)

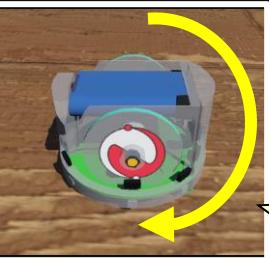
Tip Forward  
(-2.6, -0.0, 9.5)



Tip Backward  
(2.6, -0.0, 9.5)

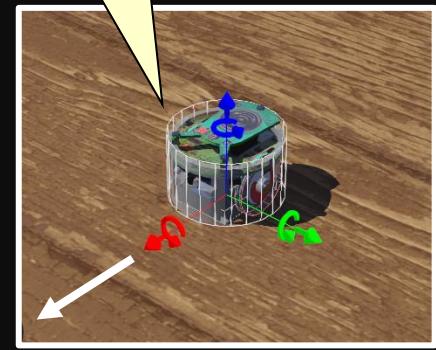


Upside Down  
(0.0, -0.0, -9.8)



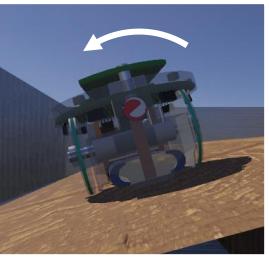
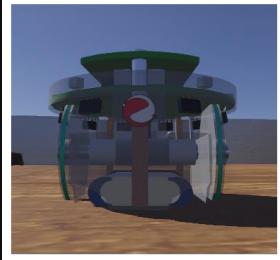
Spin about Y-axis

3-Axis accelerometer



(0.0, 0.0, 9.8)

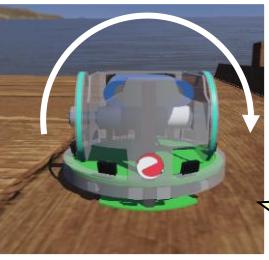
Tip Left  
(-0.0, -2.5, 9.5)



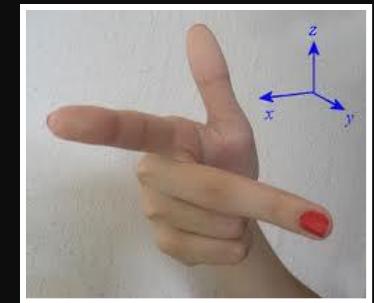
Tip Right  
(0.0, 2.5, 9.5)



Upside Down  
(0.0, -0.0, -9.8)



Spin about Z-axis



# E-Puck Accelerometer Code

```
import com.cyberbotics.webots.controller.Accelerometer;  
  
// Accelerometers are objects  
Accelerometer accelerometer;  
  
// Set up the accelerometer  
Accelerometer accelerometer = new Accelerometer("accelerometer");  
accelerometer.enable(timeStep);  
  
// Need to capture (x, y, z) values in a double array  
double[] accelValues = new double[3];  
  
// WHILE LOOP {  
    // Get all three values each time  
    accelValues = accelerometer.getValues();  
    String s = String.format("Accel (x=%2.1f, y=%2.1f, z=%2.1f) ",  
        accelValues[0], accelValues[1], accelValues[2]);  
    System.out.println(s);  
}
```



```
Accel (x=-0.01, y=0.01, z=9.81)  
Accel (x=-0.01, y=0.01, z=9.78)  
Accel (x=-0.01, y=0.02, z=9.78)  
Accel (x=-0.03, y=-0.61, z=9.82)  
Accel (x=-0.09, y=0.02, z=9.79)  
Accel (x=-0.09, y=0.02, z=9.79)  
Accel (x=-0.02, y=-2.46, z=9.75)  
Accel (x=0.20, y=1.83, z=9.82)  
Accel (x=-0.91, y=-1.98, z=6.91)  
Accel (x=-1.60, y=-1.31, z=14.02)  
Accel (x=0.86, y=-0.58, z=7.40)  
Accel (x=-0.32, y=-0.41, z=11.56)  
Accel (x=0.28, y=3.32, z=8.96)  
Accel (x=-0.20, y=-3.23, z=9.80)  
Accel (x=-0.03, y=0.37, z=10.01)  
Accel (x=-0.55, y=-1.59, z=9.29)  
Accel (x=0.03, y=0.66, z=6.68)  
Accel (x=0.16, y=-6.35, z=13.63)  
Accel (x=-1.48, y=-2.26, z=5.56)  
Accel (x=-1.00, y=-4.67, z=14.12)  
Accel (x=-1.18, y=-0.75, z=7.32)  
Accel (x=-1.50, y=-2.68, z=9.41)  
Accel (x=-1.23, y=-0.35, z=9.38)  
Accel (x=-1.36, y=-1.87, z=9.40)
```

As robot moves,  
values will  
fluctuate a lot.

# Accelerometer Data Smoothing

- With data bouncing up and down too much, we need to smooth it out by taking a ***running average***:

- Initialize an array of size 10 or so:

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

```
double[] readings;  
readings = new double[10];
```

- As readings come in, fill up the array:

0.01	0.01	0.02	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31
0	1	2	3	4	5	6	7	8	9

```
readings[i] = accelValues[1];
```

- When 11<sup>th</sup> reading comes in, wrap around to the start again, overwriting the oldest readings:

-0.58	0.01	0.02	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31
0	1	2	3	4	5	6	7	8	9

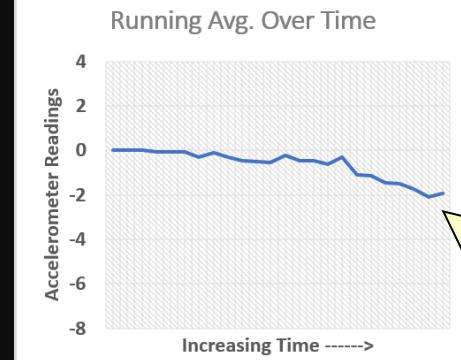
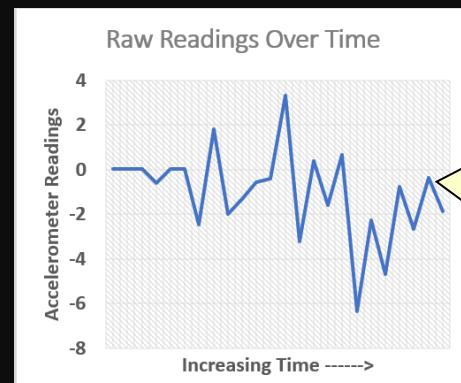
```
i = (i + 1) % 10;
```

# Accelerometer Data Smoothing

- When we take the average of the array, we get the average of the latest 10 readings:

0.01	0.01	0.02	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31	-0.445
-0.58	0.01	0.02	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31	-0.504
-0.58	-0.41	0.02	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31	-0.487
-0.58	-0.41	3.32	-0.61	0.02	0.02	-2.46	1.83	-1.98	-1.31	-0.114
-0.58	-0.41	3.32	-3.23	0.02	0.02	-2.46	1.83	-1.98	-1.31	-0.769
-0.58	-0.41	3.32	-3.23	0.37	0.02	-2.46	1.83	-1.98	-1.31	-0.409
-0.58	-0.41	3.32	-3.23	0.37	-1.59	-2.46	1.83	-1.98	-1.31	-0.605

This is the running average.



It is hard to tell exactly what is going on due to the shaking of the robot.

Noise is eliminated. Since this is the y-axis, we can detect that the robot is starting to tip forward.

Start the  
Lab ...

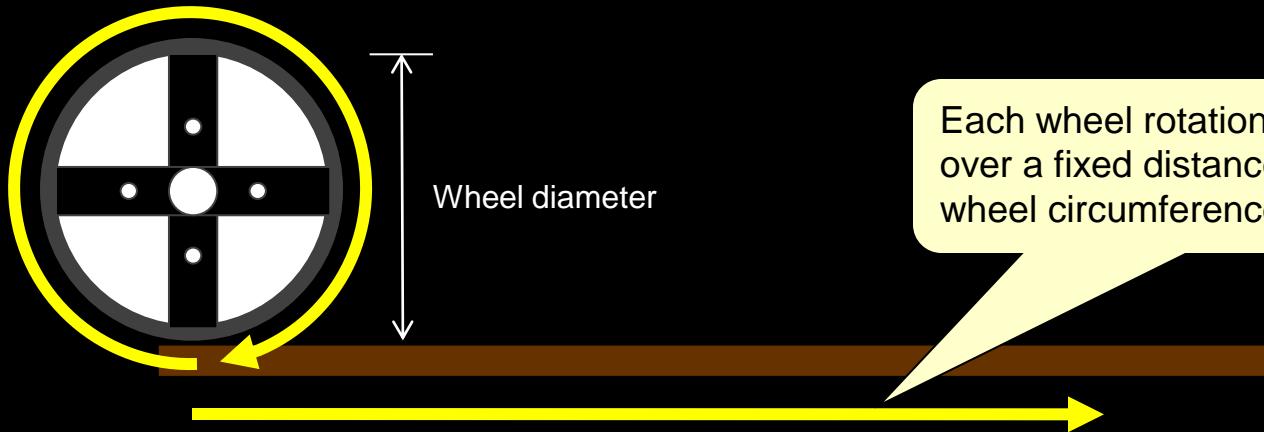
# Position Estimation

# Velocity

- A robot's velocity (i.e., speed) depends on two things:
  - RPM (Rotations Per Minute) of the wheels
  - Wheel diameter

## Velocity

$$\begin{aligned} &= \text{RPM} * \text{Circumference} \\ &= \text{RPM} * 2\pi * \text{wheel radius} \\ &= \text{RPM} * \pi * \text{wheel diameter} \end{aligned}$$



# Velocity (continued)

- Imagine that you have a motor spinning at 100RPMs and a wheel diameter of 6cm. How fast is it going ?

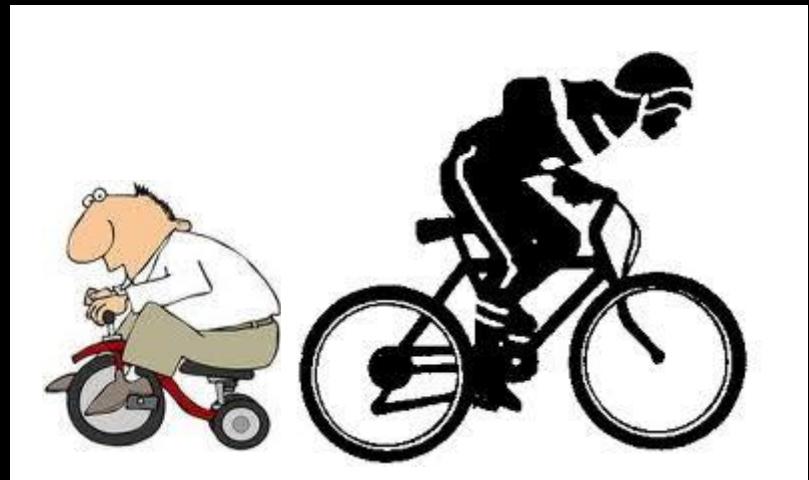
$$= \text{RPM} * \pi * \text{diameter}$$

$$= 100 \text{ rpm} * \pi * 6 \text{ cm}$$

$$= 1884.96 \text{ cm/minute}$$

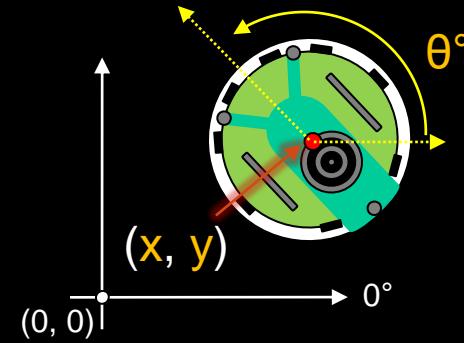
$$= 31.4 \text{ cm/sec}$$

- Larger wheels will increase the velocity, smaller wheels will decrease it.



# What is a Pose ?

- A robot's **position** is its location in the environment:
  - Represented as  $(x, y)$  coordinate in 2D or  $(x, y, z)$  in 3D
- A robot's **orientation** is its direction in the environment:
  - Represented as  $\theta^\circ$  in 2D or  $(\text{roll}^\circ, \text{pitch}^\circ, \text{yaw}^\circ)$  in 3D
  - Always with respect to some reference direction such as magnetic North or a horizontal line.
- A robot's **pose** is a combination of its location and orientation:
  - Represented as  $(x, y, \theta^\circ)$  in 2D or  $(x, y, z, \text{roll}^\circ, \text{pitch}^\circ, \text{yaw}^\circ)$  in 3D



# What is Position Estimation ?

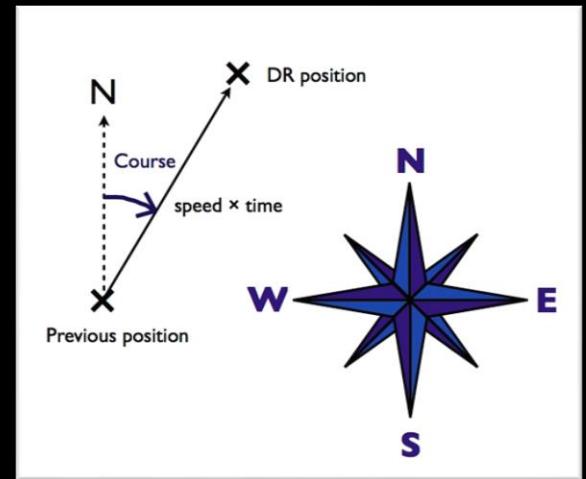
---

- The task of determining the pose of a robot is referred to as **pose estimation** (also known as **position estimation**).
- It is an “estimate” because no sensors are accurate enough to give you an exact position or orientation.
- Accuracy of estimate depends on various factors:
  - Quality and reliability of sensors
  - Accuracy of sensor readings
  - Presence of environmental noise and interference
  - Past readings (i.e., accumulative error)
  - Availability of reference data (e.g., maps)



# Odometry

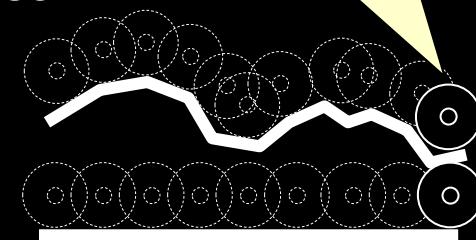
- Odometry is a way of determining a robot's position based on previous known position information given a specific course heading and velocity.
  - Used for years by boats and airplanes
  - Used on many mobile robots
- Errors accumulate over time as robot moves due to uncertainty in measurements.
- Periodically requires error measurement to be "fixed" or reset (usually from external sources) in order to be useful.
- Meant for short distance measurements.



# Odometry Errors

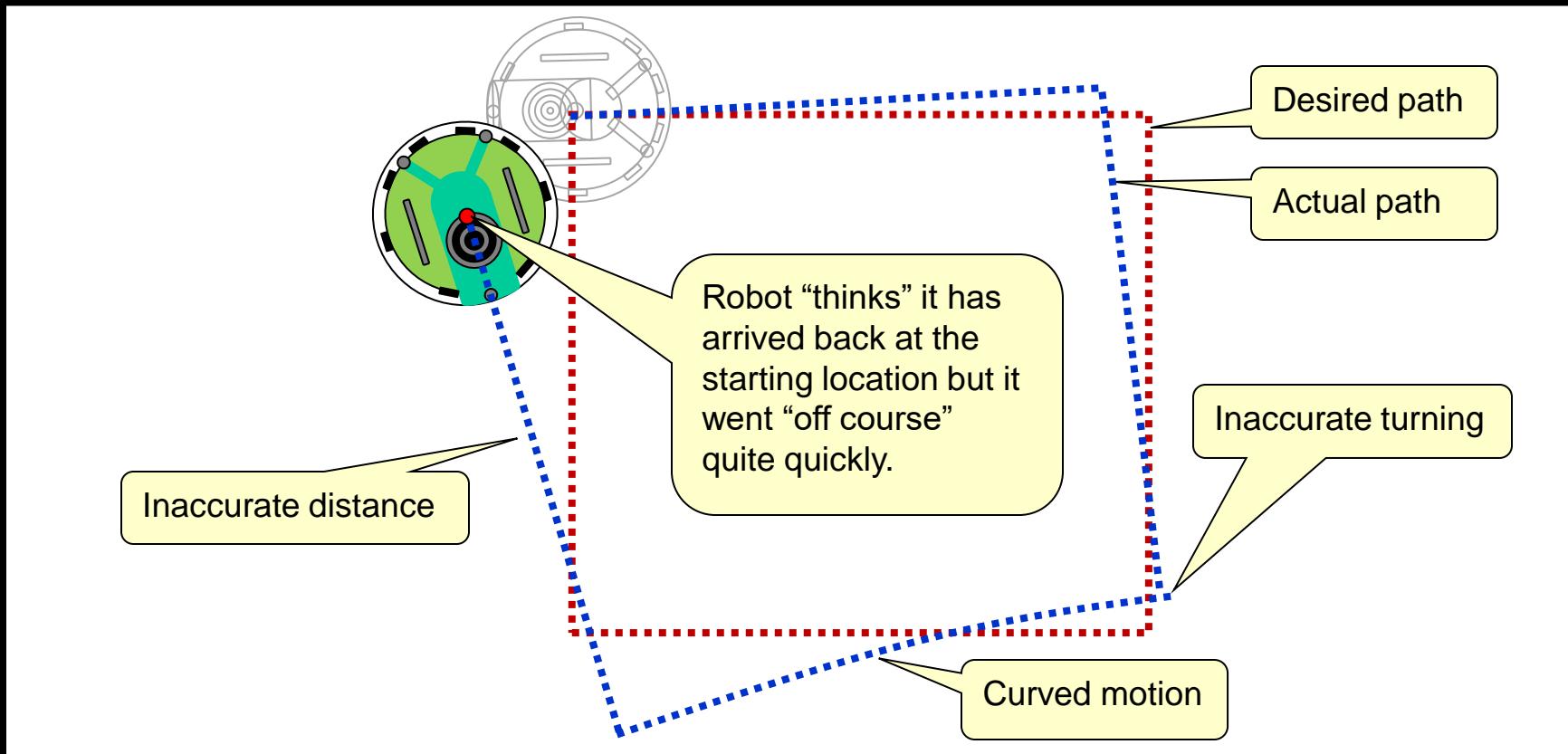
- Errors can creep-in due to:
  - Imprecise measurements
    - Actual speed and direction cannot be measured accurately
  - Inaccurate control model
    - Wheels are not infinitely thin and do not make contact with the ground surface at a single point
    - Wheels are not exactly the same size with axles aligned perfectly
  - Immeasurable physical characteristics
    - Friction is not infinite in rolling direction and zero otherwise
    - Wheels wobble slightly and skid during turns
    - Surface is not perfectly smooth and hard

Wheel travels further distance,  
but same (x,y) coordinate



# Odometry Errors – The Effect

- As a result of these error factors, a simple path cannot be traversed accurately.



# Odometry ... There is “Hope”

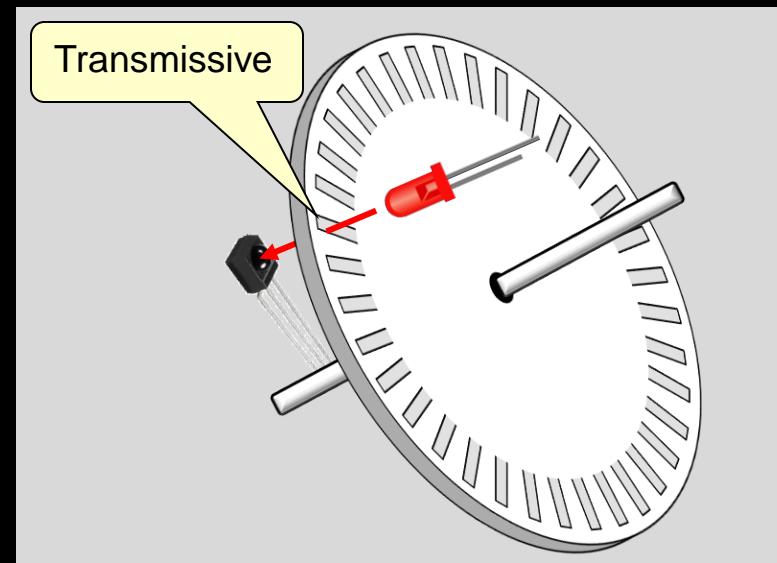
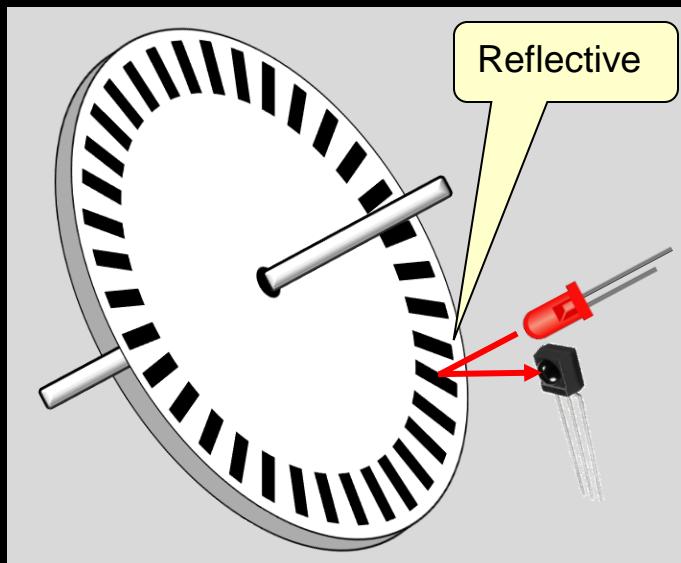
- Theoretically, we can calculate the actual robot's position as long as:
  1. the robot's structure is well known, and
  2. the robot's wheel acceleration/deceleration/velocity or amount of rotation can be accurately measured.
- Various sensors can be used to measure distance, velocity and/or acceleration:
  - Optical encoders (on per wheel)
  - Doppler sensors (usually ultrasonic)
  - Inertial Measurement Units (IMU)



Very popular for wheeled robots ... and inexpensive.

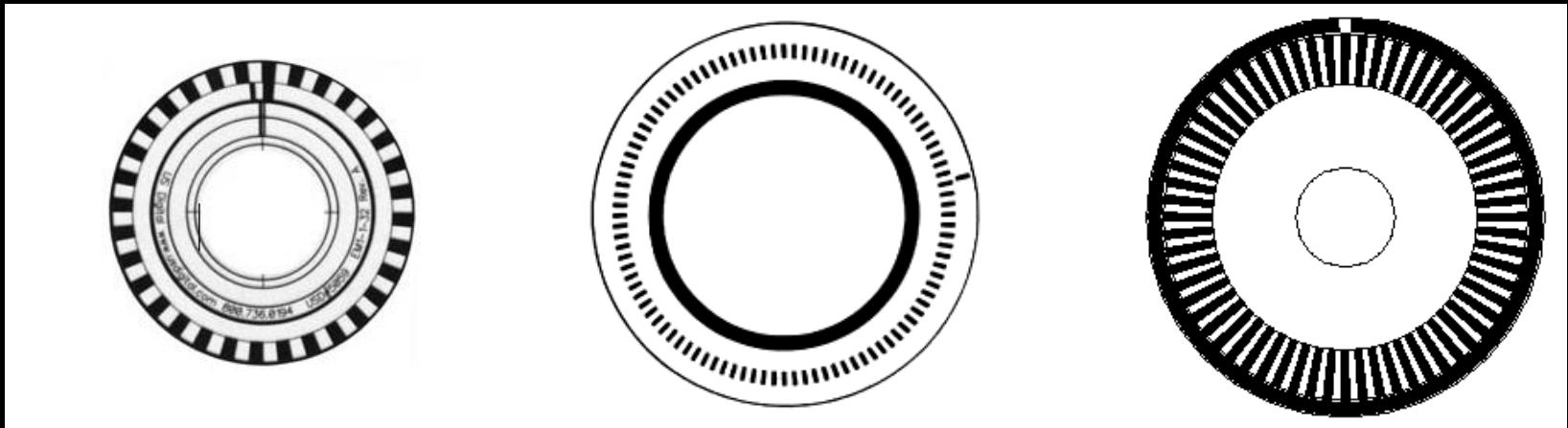
# Optical Encoders

- **Optical encoders** are devices used to measure angular position, velocity or amount of rotation.
  - A focused beam of light aimed at a photodetector which is periodically interrupted by an opaque or transparent pattern on a rotating disk attached to the shaft of the wheel.

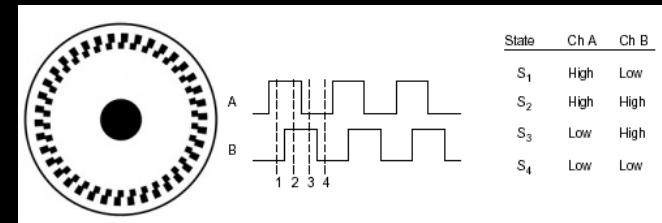


# Incremental Optical Encoders

- Disks have evenly spaced slots around border which indicate accuracy:



- Can measure **velocity** and infer **relative position**.
- Two subtypes: (1) single-channel (as shown above) or (2) phase-quadrature (allows double position and better accuracy)

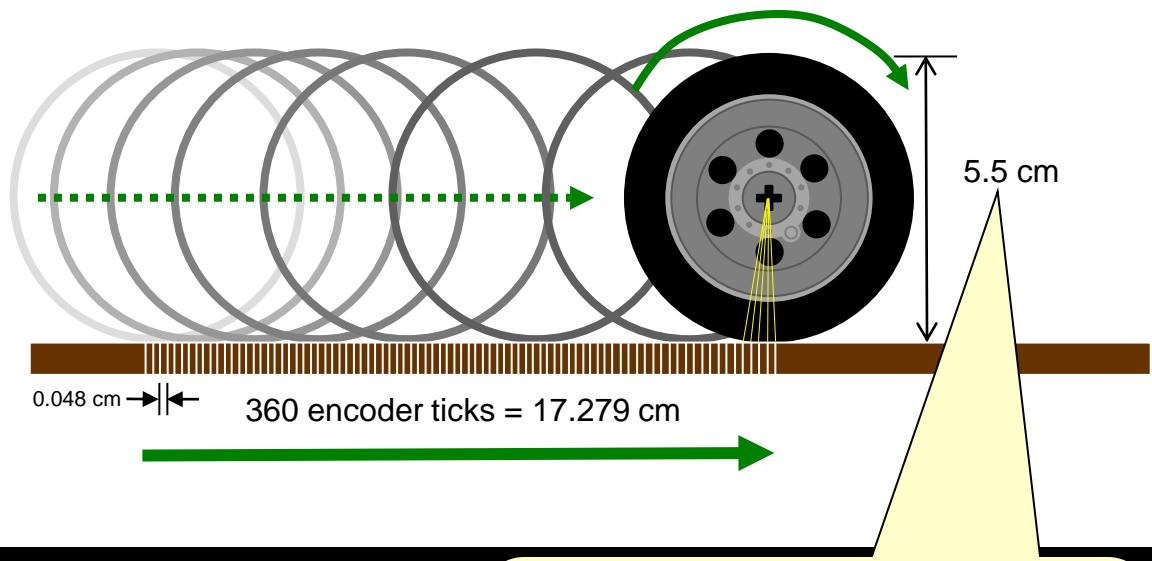


# Measuring Distance

- If robot is moving straight ahead, simply count encoder ticks to determine how far it travelled.

Distance traveled per tick:  
$$= \frac{(\text{Wheel Circumference})_{\text{cm}}}{360_{\text{ticks}}}$$
  
$$= 5.5_{\text{cm}} \pi / 360$$
  
$$= 0.048_{\text{cm}}$$

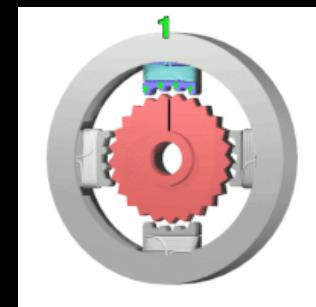
Calculation assumes an encoder with 360 ticks per revolution, but every robot may differ.



Actual wheel diameter can vary (e.g.,  $\pm 0.1\text{cm}$ ), depending on the particular wheel, its age, the weight of the robot, the placement of the wheel, etc...

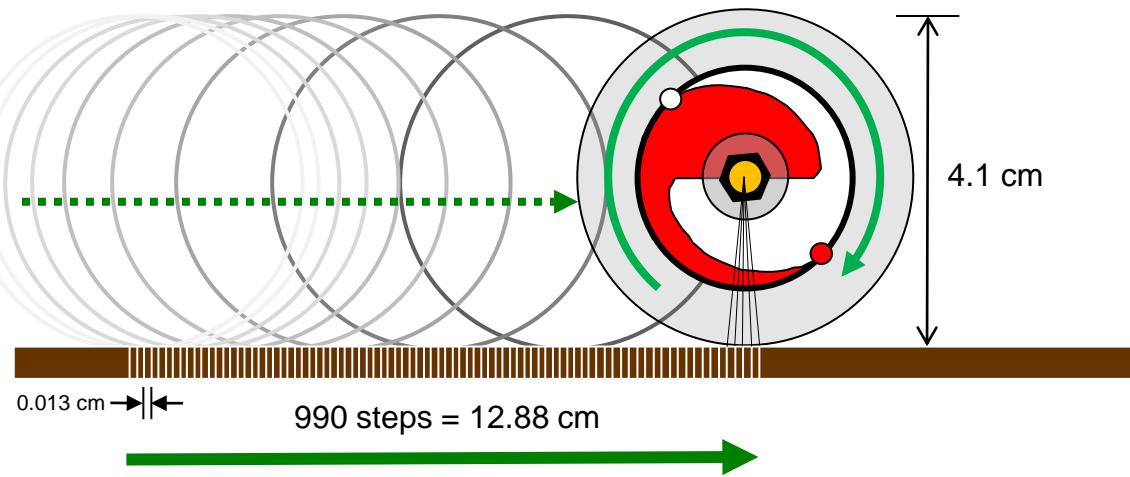
# E-Puck - Measuring Distance

- The GcTronic E-puck uses stepper motors instead of encoders.
  - Motor can move in small increments, so position does not need to be measured with an encoder.
- E-puck has 50:1 reduction gear, so provides accuracy of about 0.013cm per motor step.



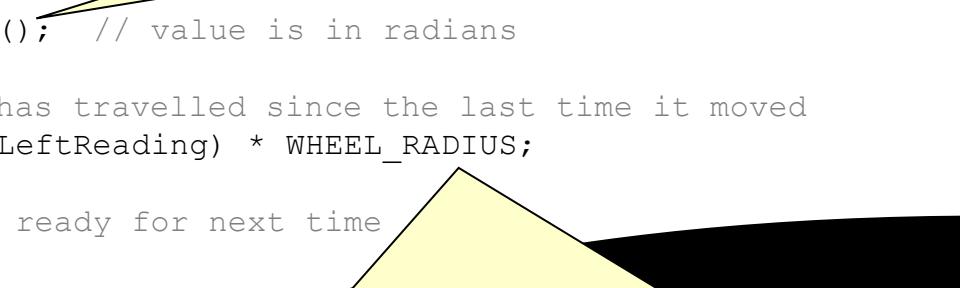
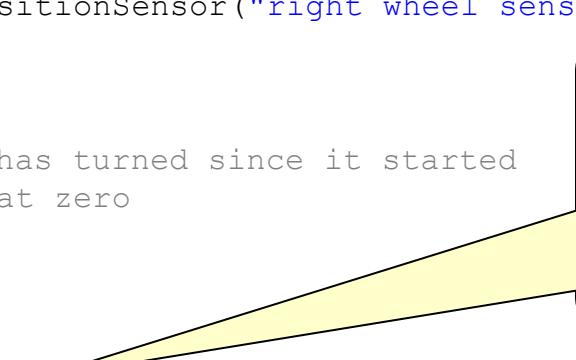
Distance traveled per step:  
$$= \frac{(\text{Wheel Circumference})_{\text{cm}}}{\text{steps/rotation}}$$
  
$$\approx \pi \frac{4.1 \text{ cm}}{990} / 990$$
  
$$\approx 0.013 \text{ cm}$$

Estimated geared-down steps per full wheel rotation.



# E-Puck – Forward Travel Calc.

```
import com.cyberbotics.webots.controller.PositionSensor;  
  
static final double WHEEL_RADIUS = 2.05; // cm  
  
// Get the wheel position sensors  
PositionSensor leftEncoder = robot.getPositionSensor("left wheel sensor");  
PositionSensor rightEncoder = robot.getPositionSensor("right wheel sensor");  
leftEncoder.enable(TimeStep);  
rightEncoder.enable(TimeStep);  
  
// Read number of radians that the wheel has turned since it started  
double previousLeftReading = 0; // start at zero  
  
// MOVE THE ROBOT A BIT  
// ...  
  
double leftReading = leftEncoder.getValue(); // value is in radians  
  
// Calculate the distance that the wheel has travelled since the last time it moved  
double distance = (leftReading - previousLeftReading) * WHEEL_RADIUS;  
  
previousLeftReading = leftReading; // Get ready for next time
```

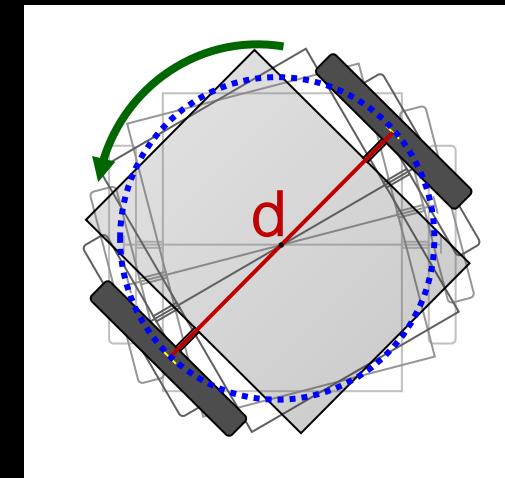


Sensor gives  
number of  
radians  
turned by  
wheel since  
it started.

$$\begin{aligned} \text{distance} &= (\text{radiansTurned}_{\text{rad}} / 2\pi) * \text{wheelCircumference} \\ &= ((\text{leftReading} - \text{previousLeftReading}) / 2\pi) * (2\pi \cdot \text{WHEEL\_RADIUS}) \\ &= (\text{leftReading} - \text{previousLeftReading}) * \text{WHEEL\_RADIUS} \end{aligned}$$

# Spin Angle Calculation

- If a 2-wheel differential drive robot **spins**, we can also count encoder ticks (or motor steps) to determine how many degrees it turned.
  - Each wheel follows the outline of a circle **C** centered at midpoint between wheels.
  - Circumference of **C** is defined by distance **d** between both wheels (i.e., axel length).
  - The number of degrees turned by the robot as it spins will depend on the portion (or %) of the circumference that is traced out.

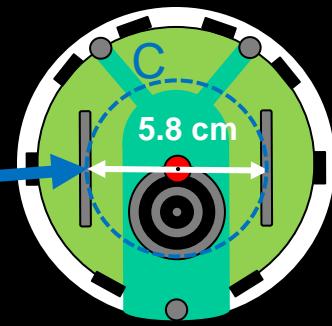


# E-Puck - Spin Angle Calculation

- Consider the GcTronic E-Puck robot with distance between wheels being  $5.8_{\text{cm}}$ .

- Spinning is centered around a circle **C** with diameter of  $5.8_{\text{cm}}$  and circumference of

$$\pi * 5.8_{\text{cm}} = 18.221_{\text{cm}}$$



- Use `getValue()` on a wheel position sensor to get # radians that wheel has turned and therefore know amount of travel (i.e., distance) along **C**'s circumference:  $\text{VALUE}_{\text{rad}} * 2.05_{\text{cm}}$

wheel radius

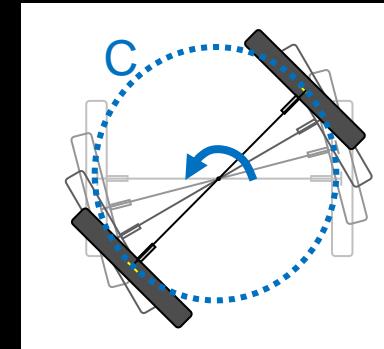
- Divide **C**'s circumference by this amount as follows:

$$\% \text{ of travel on } \mathbf{C}'\text{s perimeter} = 18.221_{\text{cm}} / (2.05_{\text{cm}} * \text{VALUE}_{\text{rad}})$$

$$\text{Angle turned} = (\% \text{ of travel on } \mathbf{C}'\text{s perimeter} * 2\pi)_{\text{rad}}$$

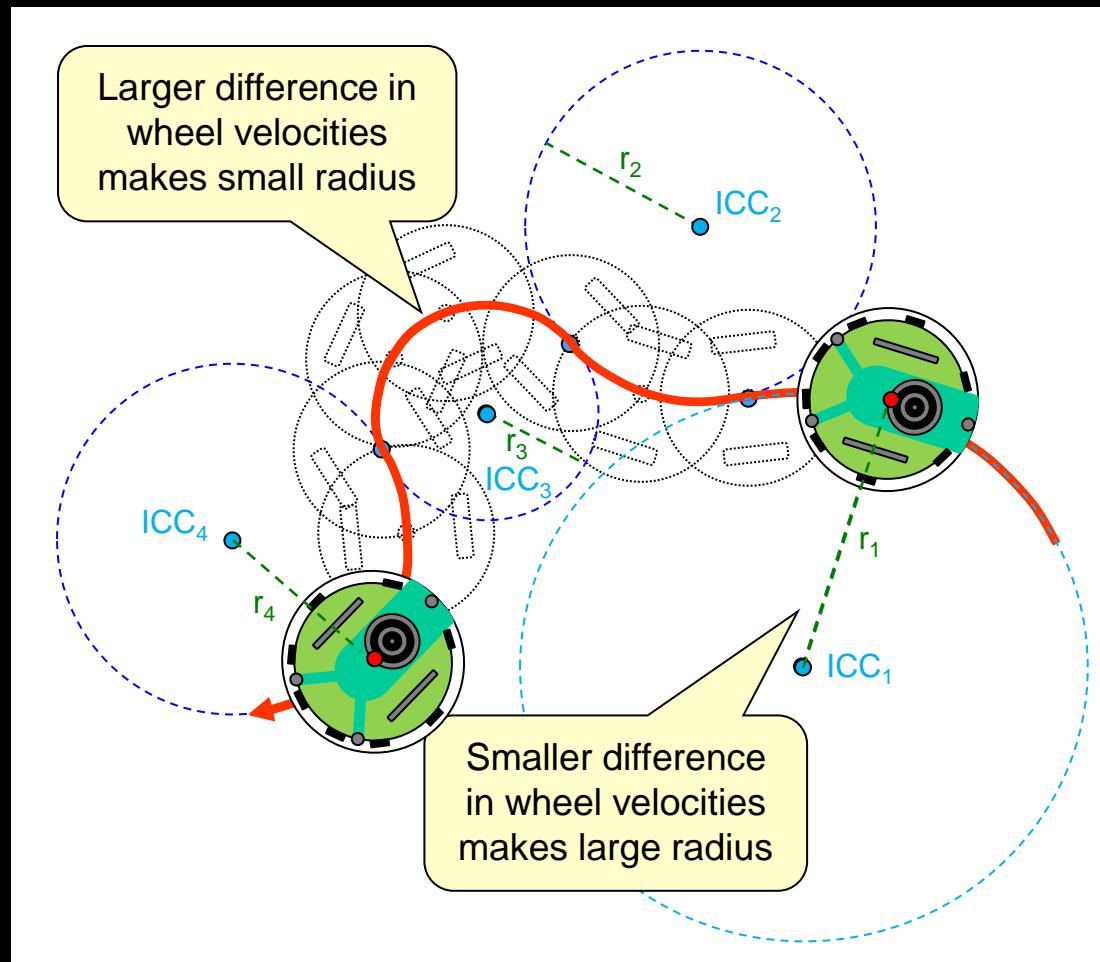
$$= (\% \text{ of travel on } \mathbf{C}'\text{s perimeter} * 360)^{\circ}$$

Amount that  
robot has turned



# Kinematics

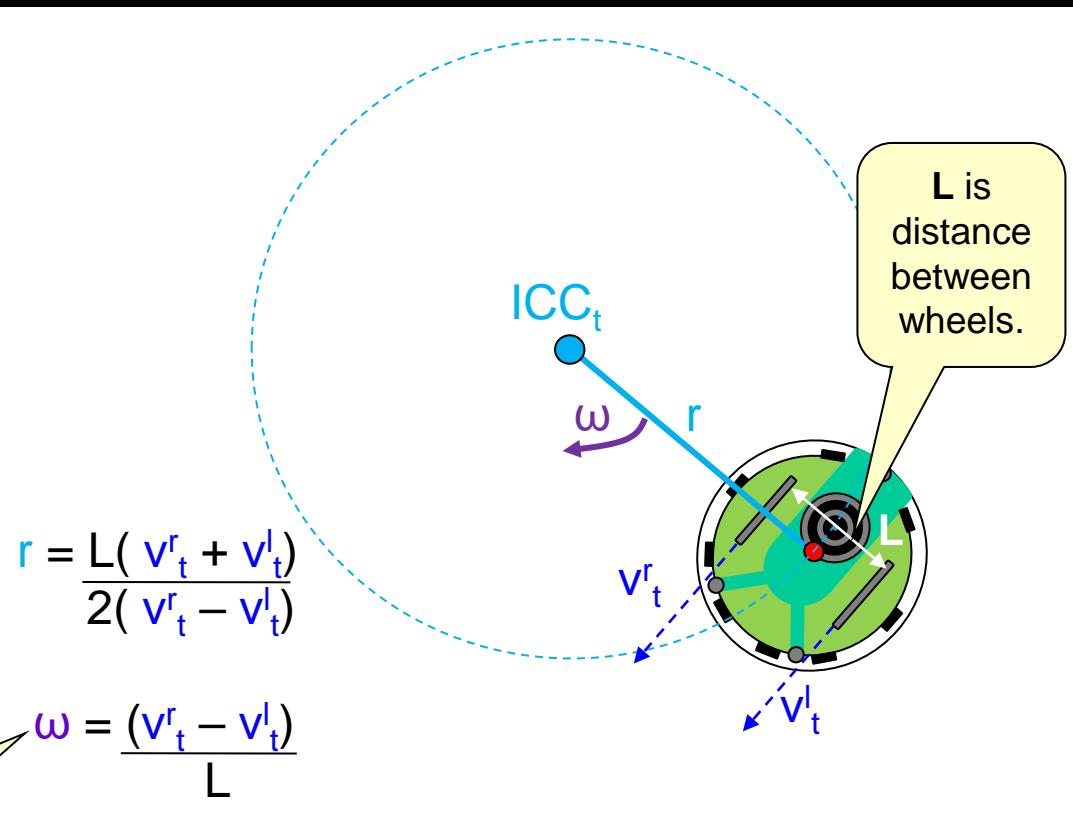
- Recall that the ***instantaneous center of curvature*** (ICC) is the point around which each wheel of the robot makes a circular course.
- ICC changes over time as a function of the individual wheel velocities



# Kinematics

- Assume that at each instance of time,  $t$ , the robot is following a curve around some  $\text{ICC}_t$  with radius  $r$  at angular rate  $\omega$  with left and right wheel velocities  $v_t^l$  and  $v_t^r$ , respectively.
- $r$  and  $\omega$  can both be calculated w.r.t. distance  $L$  between wheels and their velocities at time  $t$ .

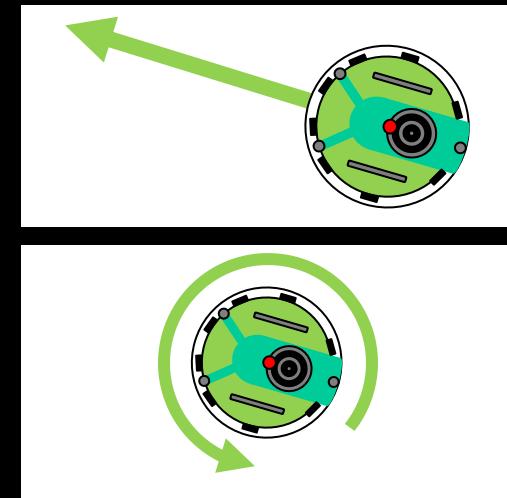
Unit is radians per sec



# Kinematics

- There are two special cases:

1. When  $v_t^l = v_t^r$ , then  $r$  is infinite and the robot moves in a straight line.
2. When  $v_t^l = -v_t^r$ , then  $r$  is zero and the robot spins (i.e., rotates in place).



- Differential drive robots are very sensitive to the velocity differences between the two wheels:
  - It is hard to move in a perfectly straight line.
  - It is hard to spin exactly in the same location.

# Forward Kinematics

- Consider the general ***forward kinematics*** problem:

Given:

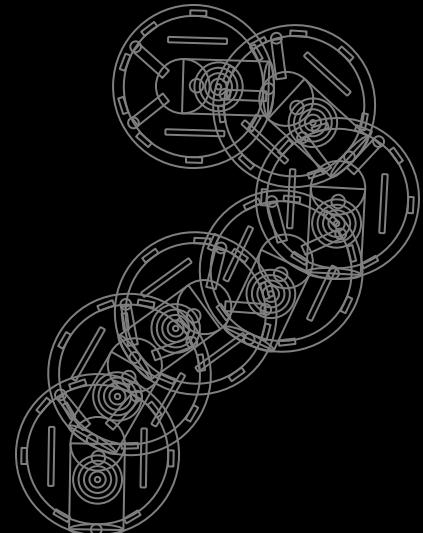
robot location ( $x_t$ ,  $y_t$ ) and orientation  $\theta_t$  at time  $t$

$\delta$  is an arbitrary number of seconds in the future.

Find:

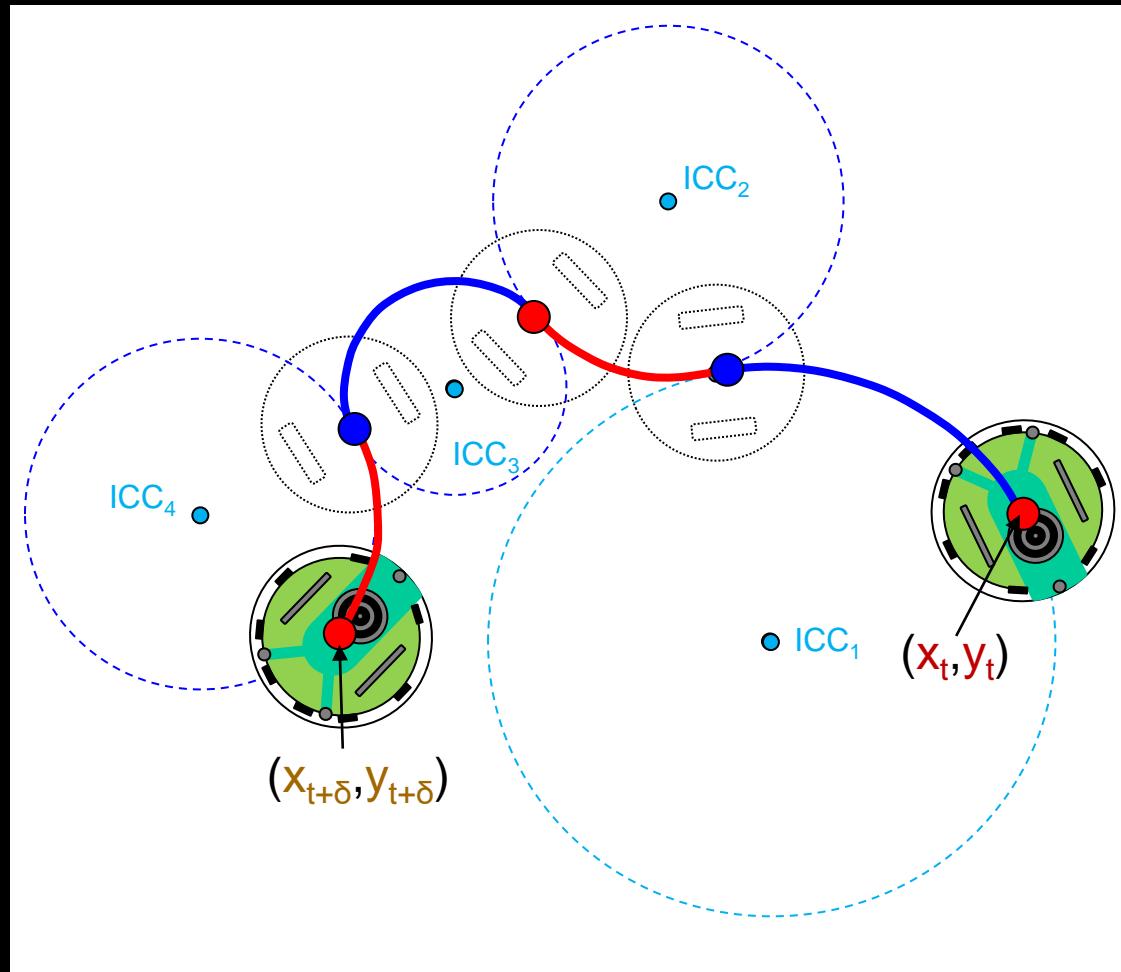
robot location ( $x_{t+\delta}$ ,  $y_{t+\delta}$ ) and orientation  $\theta_{t+\delta}$  at time  $t+\delta$

- This is not a straight-forward task
  - Robot may move, turn, spin arbitrarily
  - Robot may speed-up, slow-down and stop



# Forward Kinematics

- Problem is manageable if we break down path of robot into pieces that have constant motor speed and deal with each separately.
- Need to compute points each time the motor speeds change.



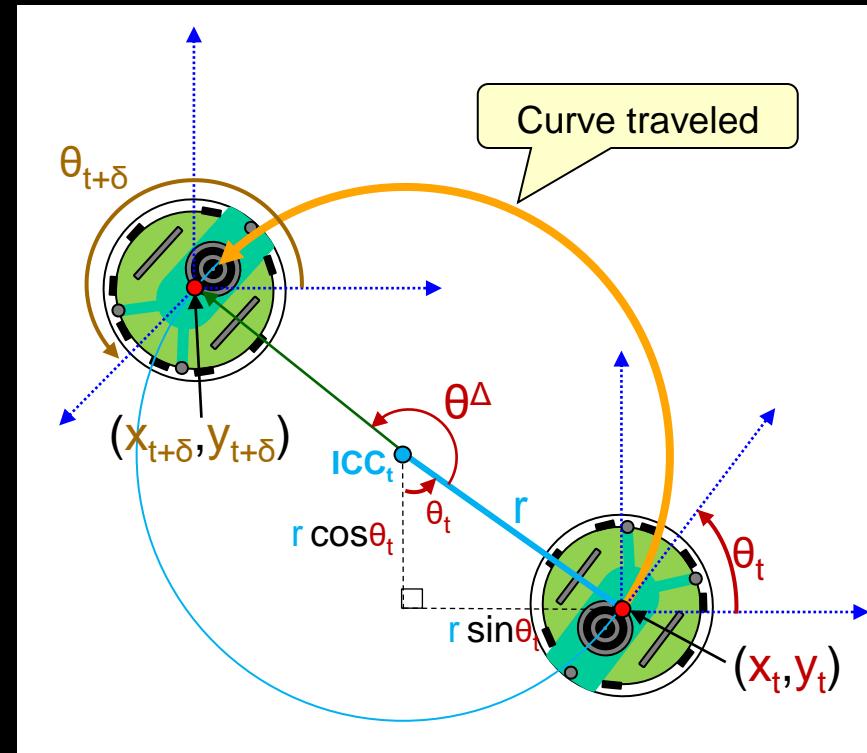
# Forward Kinematics

- Consider one such portion of the path.
- We must determine ...

$$ICC_t = (X_{icc}, Y_{icc})$$

$$= (x_t - r \cdot \sin \theta_t, y_t + r \cdot \cos \theta_t)$$

- But how do we determine radius  $r$  since we do not know the wheel velocities ?
- We can compute  $r$  in terms of encoder ticks ...



# Forward Kinematics

- Wheels trace out circles with different circumferences:

- Dist. traveled by left wheel:

$$D_L = |r_L * \theta^\Delta|$$

- Dist. traveled by right wheel:

$$D_R = |(r_L + L) * \theta^\Delta|$$

$$= |D_L + L * \theta^\Delta|$$

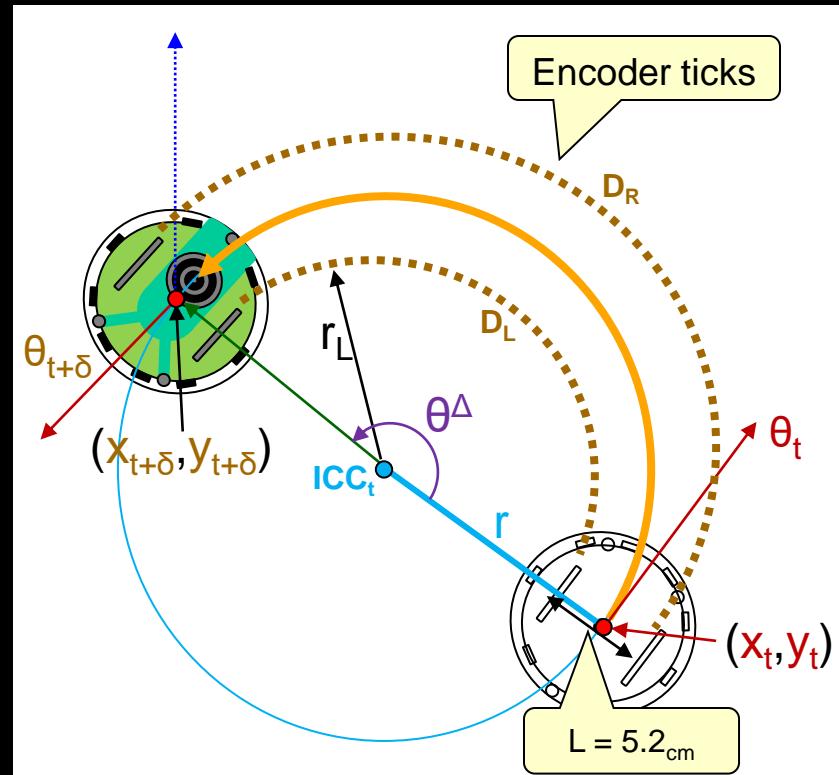
- We can re-arrange to get these:

$$\theta^\Delta = (D_R - D_L) / L$$

$$r_L = LD_L / (D_R - D_L)$$

- And so,

$$r = r_L + L/2 = LD_L / (D_R - D_L) + L/2$$



# Forward Kinematics

- Recall that `getValue()` gives radians travelled for an E-Puck wheel. Therefore,  $D_R$  and  $D_L$  can be determined by calling that function for each wheel:

$$D_R = \text{rightReading}_{\text{rad}} * \text{radius} = 2.05_{\text{cm}} * \text{rightReading}$$

$$D_L = \text{leftReading}_{\text{rad}} * \text{radius} = 2.05_{\text{cm}} * \text{leftReading}$$

- We can compute  $r$  now as follows:

$$\begin{aligned} r &= L * (D_L / (D_R - D_L) + \frac{1}{2}) \\ &= 5.8_{\text{cm}} * (2.05_{\text{cm}} * \text{leftReading} / (2.05_{\text{cm}} * \text{rightReading} - 2.05_{\text{cm}} * \text{leftReading}) + \frac{1}{2}) \\ &= [5.8 * (\text{leftReading} / (\text{rightReading} - \text{leftReading})) + 2.9]_{\text{cm}} \end{aligned}$$



# Forward Kinematics

---

- Recall that the amount of turning that happened is:

$$\begin{aligned}\theta^\Delta &= (D_R - D_L) / L \\ &= (2.05_{\text{cm}} * \text{rightReading} - 2.05_{\text{cm}} * \text{leftReading}) / L \\ &= (2.05_{\text{cm}} * (\text{rightReading} - \text{leftReading})) / 5.8_{\text{cm}} \\ &= (\text{rightReading} - \text{leftReading}) * 0.35344828_{\text{radians}} \\ &= (\text{rightReading} - \text{leftReading}) * 20.2510945^\circ\end{aligned}$$

- Now that we have the ICC radius  $r$  and the change in angle  $\theta^\Delta$ , we can determine the new location and angle by applying some “nifty” formulas.
- Assume that we know  $(x_t, y_t)$  and  $\theta_t$  at time  $t$  and that we want to compute  $(x_{t+\delta}, y_{t+\delta})$  and  $\theta_{t+\delta}$  at time  $t+\delta$  for some  $\delta$  number of seconds.

# Forward Kinematics – Curving

- At time  $t+\delta$ , the robot's pose is:

$$\begin{bmatrix} x_{t+\delta} \\ y_{t+\delta} \\ \theta_{t+\delta} \end{bmatrix} = \begin{bmatrix} \cos(\theta^\Delta) & -\sin(\theta^\Delta) & 0 \\ \sin(\theta^\Delta) & \cos(\theta^\Delta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t - X_{ICC} \\ y_t - Y_{ICC} \\ \theta_t \end{bmatrix} + \begin{bmatrix} X_{ICC} \\ Y_{ICC} \\ \theta^\Delta \end{bmatrix}$$

- Since  $X_{ICC} = x_t - r \cdot \sin \theta_t$  and  $Y_{ICC} = y_t + r \cdot \cos \theta_t$  ... then:

$$x_{t+\delta} = r \cdot \cos \theta^\Delta \cdot \sin \theta_t + r \cdot \cos \theta_t \cdot \sin \theta^\Delta + x_t - r \cdot \sin \theta_t$$

$$y_{t+\delta} = r \cdot \sin \theta^\Delta \cdot \sin \theta_t - r \cdot \cos \theta_t \cdot \cos \theta^\Delta + y_t + r \cdot \cos \theta_t$$

$$\theta_{t+\delta} = \theta_t + \theta^\Delta$$

where,

$$r = [5.8 * (\text{leftReading} / (\text{rightReading} - \text{leftReading})) + 2.9]_{cm}$$

$$\theta^\Delta = (\text{rightReading} - \text{leftReading}) * 20.2510945^\circ$$

So ... we just compute these equations each time the robot's wheels speed changes.

# Forward Kinematics – Straight

- Straight forward movement is a special case

- When (`rightReading == leftReading`) then  $r = \infty$

- Therefore, ICC equation cannot be used

- The math is even simpler:

$$x_{t+\delta} = x_t + d \cos \theta_t$$

$$y_{t+\delta} = y_t + d \sin \theta_t$$

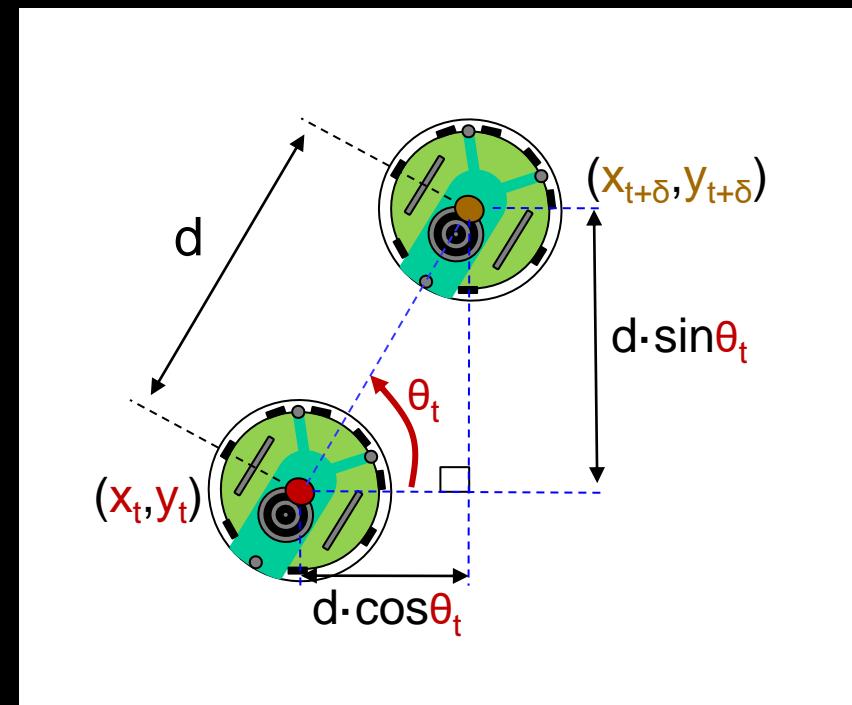
$$\theta_{t+\delta} = \theta_t$$

Angle does not change.

where,

$$d = \text{leftReading} * 2.05_{\text{cm}}$$

Can be either left or right sensor value.



# Forward Kinematics – Spinning

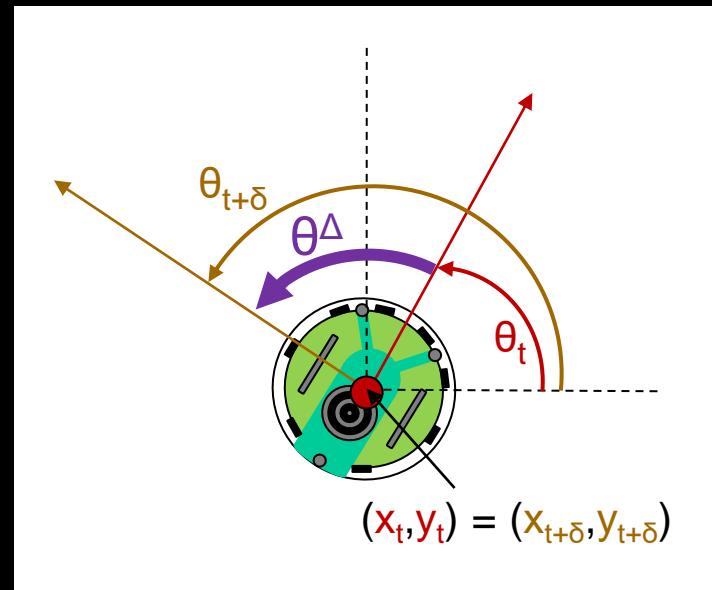
- Spinning is also a special case
  - when (`rightReading = - leftReading`) then  $r = 0$  ... ICC equation cannot be used
- The math is also simple:

$$\left. \begin{array}{l} x_{t+\delta} = x_t \\ y_{t+\delta} = y_t \\ \theta_{t+\delta} = \theta_t + \theta^\Delta \end{array} \right\} \text{Location does not change.}$$

where

$$\theta^\Delta = (\text{rightReading} - \text{leftReading}) * 20.2510945^\circ$$

Difference between right and left sensor values



Start the  
Lab ...

# Inverse Kinematics

# Inverse Kinematics

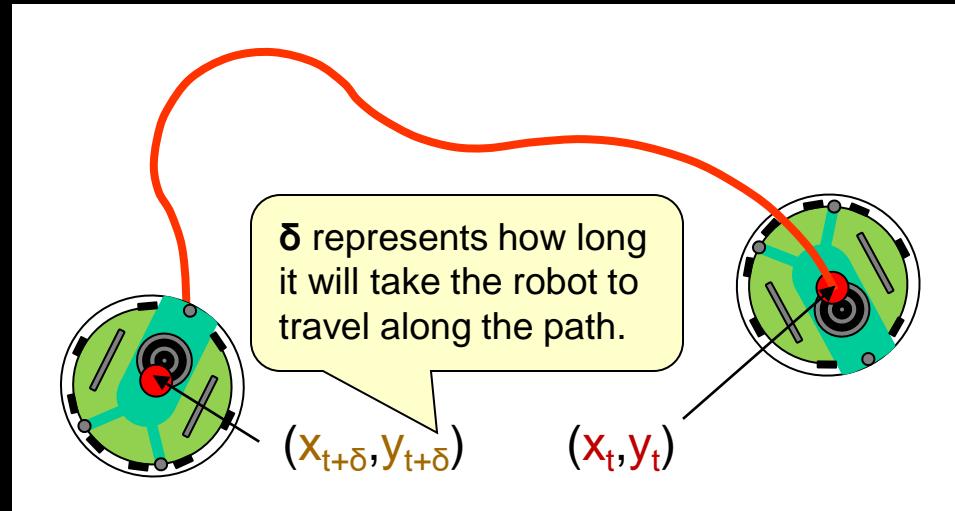
- Consider the ***inverse kinematics*** problem:

Given:

Sequence of robot positions (i.e. path) from  $(x_t, y_t, \theta_t)$  to  $(x_{t+\delta}, y_{t+\delta}, \theta_{t+\delta})$

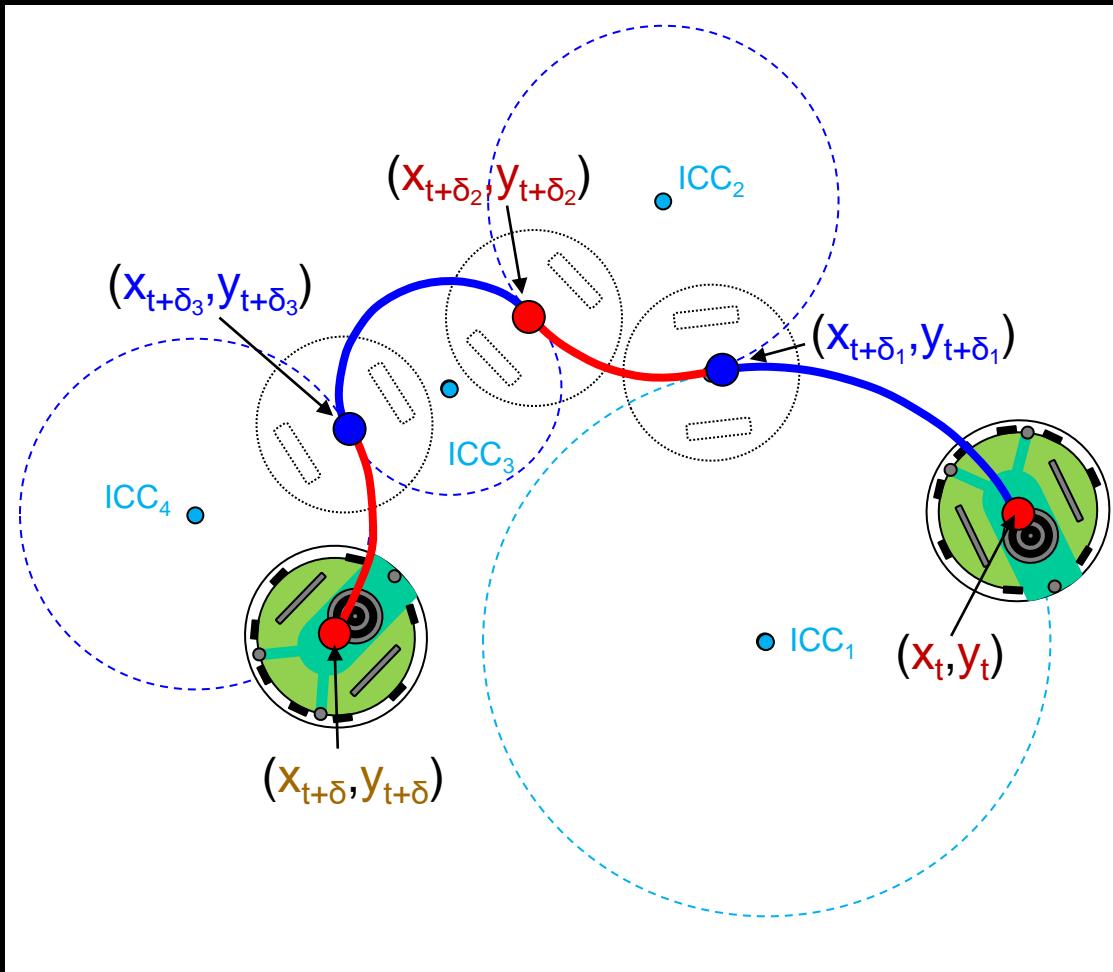
Find:

Speeds (over time) to set each motor to bring the robot along path from start position to end position over time  $\delta$  (also need to find  $\delta$ ).



# Inverse Kinematics

- It is a difficult problem!
  - complicated math
  - motors change speeds often along path
  - multiple solutions
- To simplify, we can try to fit ICC circles along the desired path along the way and compute the equations from one point to another.



# Inverse Kinematics

- Still involves solving for `leftReading` and `rightReading` in:

$$x_{t+\delta} = r \cdot \cos\theta^\Delta \cdot \sin\theta_t + r \cdot \cos\theta_t \cdot \sin\theta^\Delta + x_t - r \cdot \sin\theta_t$$

$$y_{t+\delta} = r \cdot \sin\theta^\Delta \cdot \sin\theta_t - r \cdot \cos\theta_t \cdot \cos\theta^\Delta + y_t + r \cdot \cos\theta_t$$

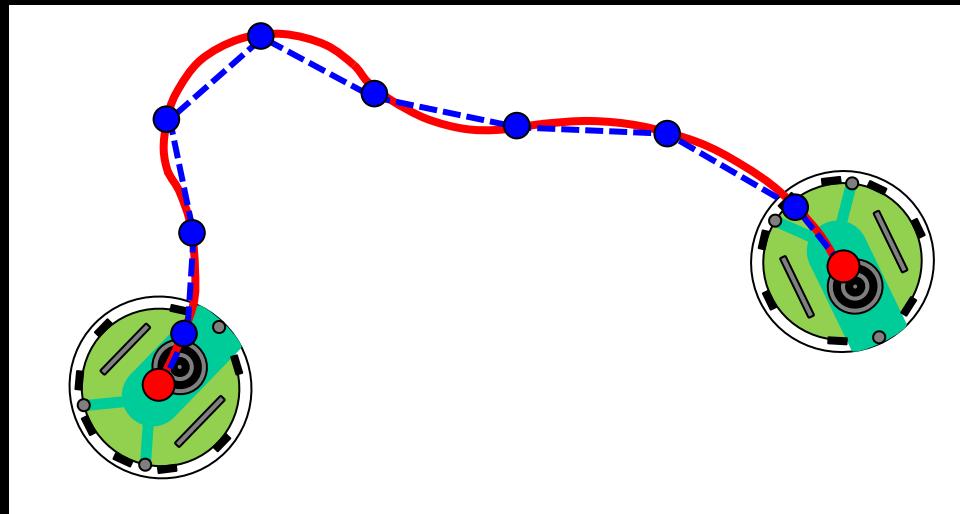
$$\theta_{t+\delta} = \theta_t + \theta^\Delta$$

$$r = [5.8 * (\text{leftReading} / (\text{rightReading} - \text{leftReading})) + 2.9]_{\text{cm}}$$

$$\theta^\Delta = (\text{rightReading} - \text{leftReading}) * 20.2510945^\circ$$

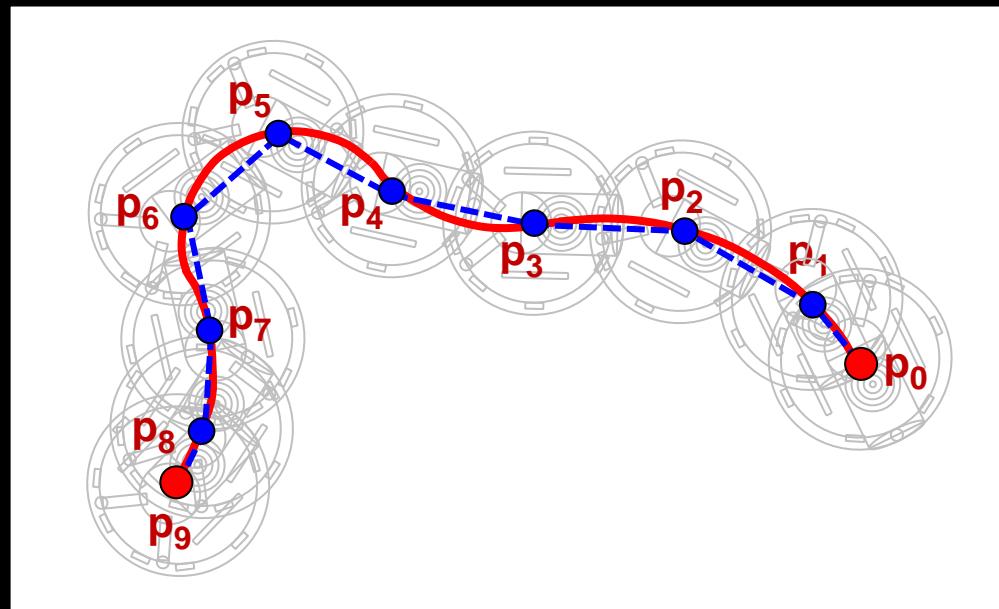


- Easier to determine piecewise-linear approximation to path by travelling to intermediate points along the way.



# Inverse Kinematics

- Can make robot travel to desired locations along approximated path by a series of spins and forward movements.
  - Spin at each vertex until facing desired angle
  - move forward until reaching next point



# Inverse Kinematics - Spin



- How do we spin from angle  $\theta_i$  to  $\theta_{i+1}$  ?
- We need to compute  $\theta_{i+1}$  from  $(x_i, y_i)$  to  $(x_{i+1}, y_{i+1})$ :

$$x^\Delta = x_{i+1} - x_i$$

$$y^\Delta = y_{i+1} - y_i$$

$$\begin{aligned}\theta_{i+1} &= \arctan(y^\Delta/x^\Delta) * 180 / \pi \\ &= \text{atan2}(y^\Delta, x^\Delta) * 180 / \pi\end{aligned}$$

Function that handles special cases (e.g.,  $x^\Delta = 0$ ).

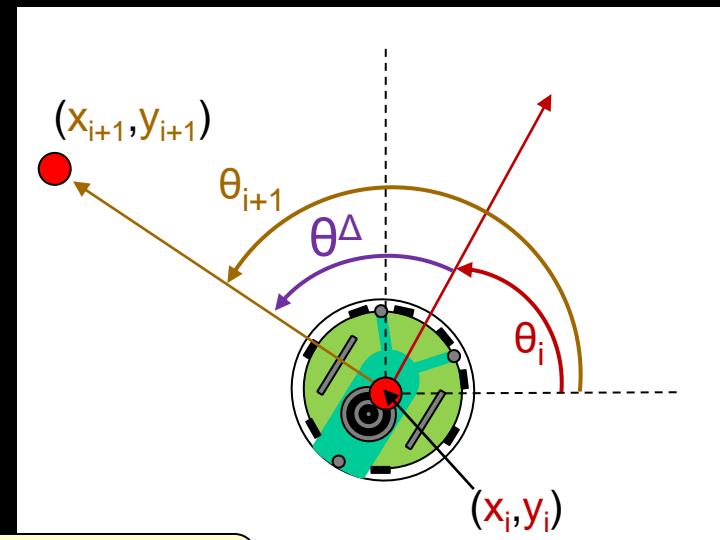
- Now compute amount of turn:

$$\theta^\Delta = (\theta_{i+1} - \theta_i) \% 360^\circ$$

Modulus handles wraparound case of turning  $> 360^\circ$ .

$$\text{IF } (\theta^\Delta < -180^\circ) \text{ THEN } \theta^\Delta = \theta^\Delta + 360^\circ$$

$$\text{ELSE IF } (\theta^\Delta > 180^\circ) \text{ THEN } \theta^\Delta = \theta^\Delta - 360^\circ$$



Do this to normalize so that all turning is within range of  $-180^\circ$  and  $+180^\circ$ .

# E-Puck - Inverse Kinematics

- If ( $\theta^\Delta > 0$ ) then right wheel should go forward and left backwards otherwise left should go forward and right backwards.
- Need to determine # encoder steps required to make the spin based on this formula (from before):

$$\begin{aligned}\theta^\Delta &= (D_R - D_L) / L \\ &= (R_{cm} * (\text{rightReading} - \text{leftReading})) / L_{cm}\end{aligned}$$

R = Wheel Radius

2.05 cm



L = Wheel Base

- Since  $\text{rightReading} = -\text{leftReading}$  when spinning:

$$\theta^\Delta = (R_{cm} * (2 * \text{rightReading})) / L_{cm} = (2R_{cm} / L_{cm}) * \text{rightReading}$$

And so...

$$\text{rightReading} = \theta^\Delta * (L_{cm} / R_{cm}) / 2$$

$\theta^\Delta$  is in radians here !

- Equations work for any speed.

# Inverse Kinematics - Forward

- How do we move forward from  $(x_i, y_i)$  to  $(x_{i+1}, y_{i+1})$  ?
- Length of time to move depends on wheel speed and distance to be travelled:

$$d = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

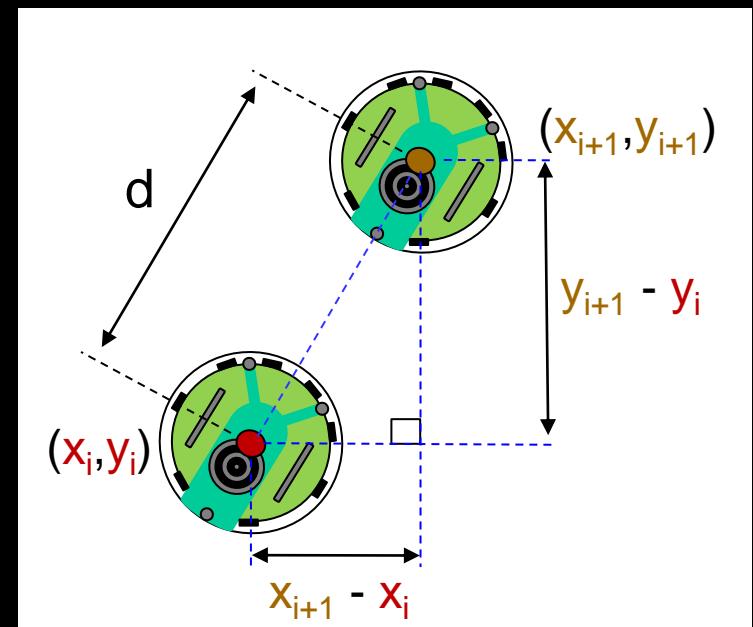
- Can express in terms of motor steps (where `rightReading = leftReading`)
- Solve for `rightReading` to move forward until:

Wheel circumference

$$\text{rightReading} = (d / (2\pi * R_{cm})) * 2\pi$$

$$= d / R_{cm} = (\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}) / R_{cm}$$

radians of rotation



# Summary

- To spin from  $(x_i, y_i)$  to  $(x_{i+1}, y_{i+1})$ , starting at angle  $\theta_i$ 
  - we determine the amount of turning  $\theta^\Delta$  to do:

$$\theta^\Delta = (\theta_{i+1} - \theta_i) \% 360^\circ$$

$$\theta_{i+1} = \text{atan2}((y_{i+1} - y_i), (x_{i+1} - x_i)) * 180^\circ / \pi$$



- We then spin left if  $\theta^\Delta > 0$  (or right  $\theta^\Delta < 0$ ) and wait until the right wheel encoder has moved this much:

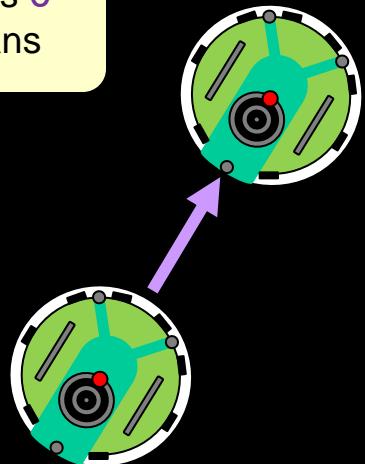
$$\text{rightReading} = \underbrace{\theta^\Delta * \pi/180^\circ}_\text{converts \theta^\Delta to radians} * (L_{cm} / R_{cm} / 2)$$

converts  $\theta^\Delta$  to radians

- To move forward from  $(x_i, y_i)$  to  $(x_{i+1}, y_{i+1})$

- we move straight until the right wheel encoder has moved this much:

$$\text{rightReading} = \left( \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2} \right) / R_{cm}$$



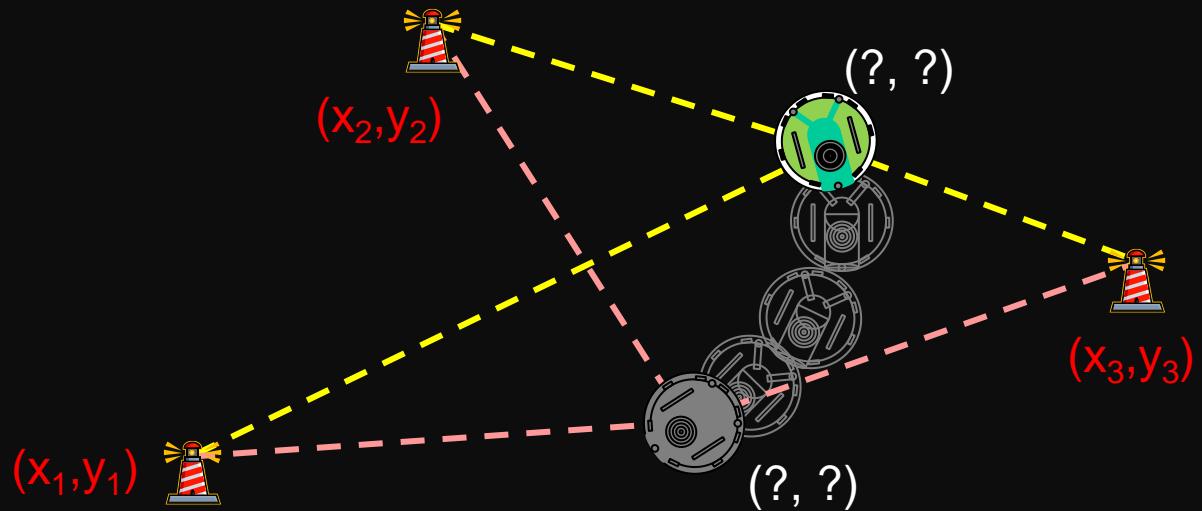
Start the  
Lab ...

# Beacon Positioning (Triangulation)

# Beacon Positioning



- A **beacon** is a detectable device that is placed at a fixed (i.e., known) position in the environment.
  - An **active beacon** transmits and/or receives signals
- A robot can estimate its “absolute” position and orientation by determining the **distance** and/or **angle** to three or more beacons.



# Beacon Systems

- Active Beacon systems ...

-  + can produce high accuracy in position estimation
-  - can be expensive to install and maintain
- require alterations to environment (i.e., installation)



- There are many commercially available indoor beacon systems:

- Active Badge
- Active Bat
- RADAR
- RICE project
- E911
- Cricket
- MotionStar Magnetic Tracker
- Easy Living
- Smart Floor



# Triangulating a Robot's Position

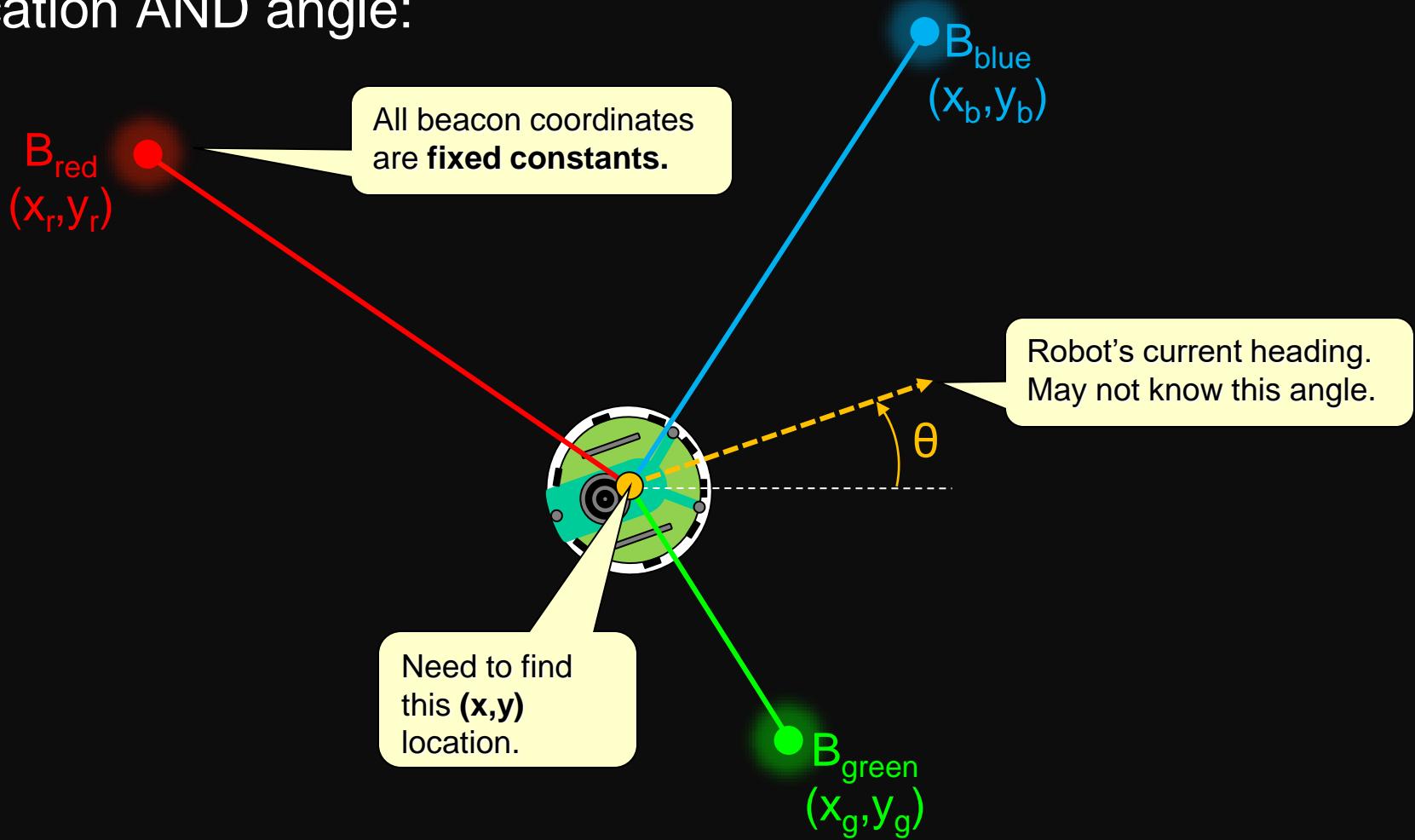
---

- Based on triangulation principle
  - uses geometric properties of triangles to compute position.
- Two types of triangulation techniques:
  - **Tri-Angulation**
    - determine robot position and angle based on **angle** to beacons
    - 2D requires 2 angles and 1 known distance, or 3 angles.
    - 3D requires 1 additional azimuth measurement
  - **Tri-Lateration**
    - determine robot position based on **distance** from beacons
    - 2D requires 3 **non-colinear** points
    - 3D requires 4 non-coplanar points



# Triangulation

- Problem: Given 3 beacons at fixed positions, compute robot location AND angle:



# Triangulation - Techniques

---

- There are many triangulation techniques (i.e., mathematical algorithms) for mobile robot positioning.

- some require a particular beacon ordering
  - some have “blind spots”
  - some only work within the triangle defined by the three beacons
  - more reliable methods exist but are more complex or require the handling of certain spatial arrangements separately.

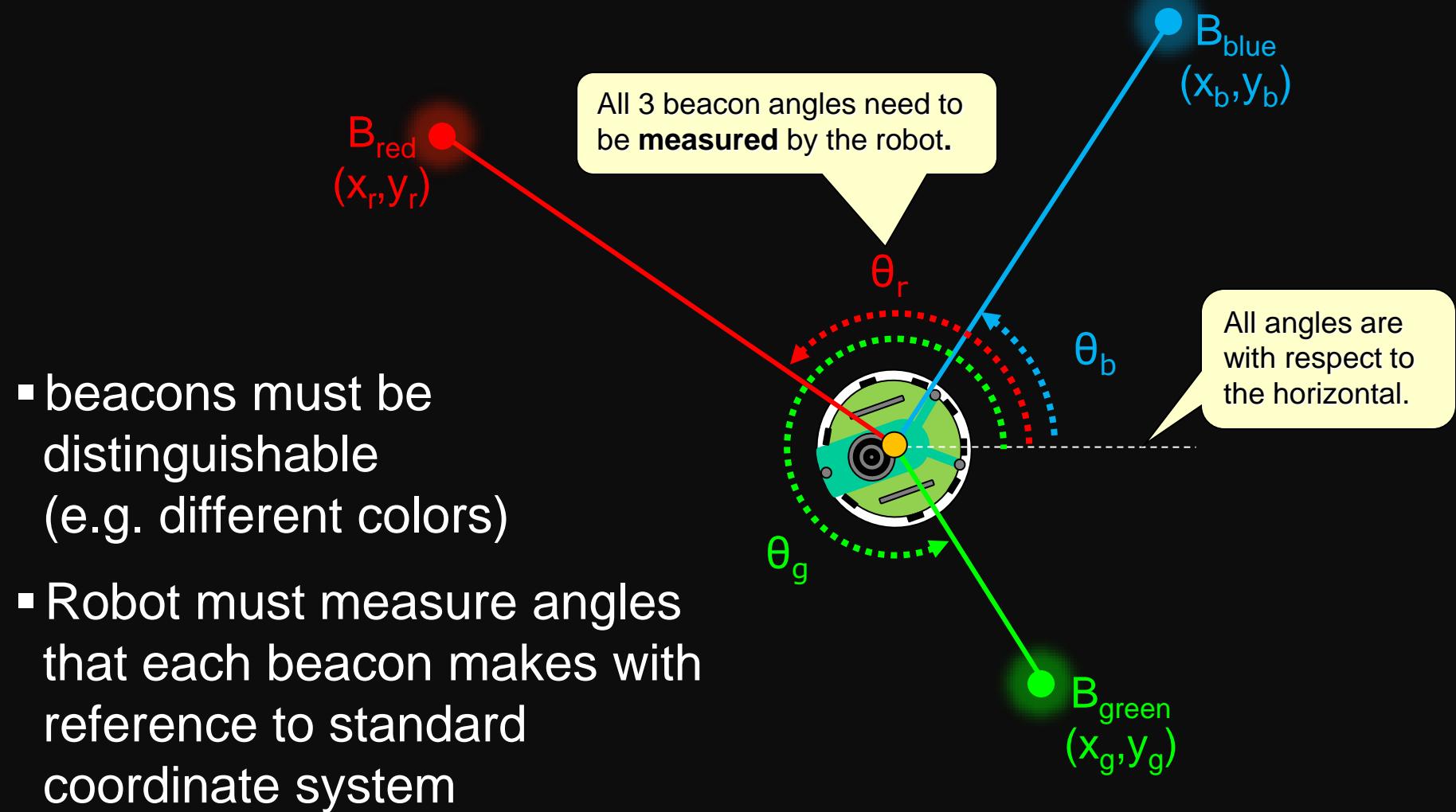


- We will use the “To Tal” algorithm:

- V. PIERLOT, M. VANDROOGENBROECK, and M. Urbin-Choffray. **A new three object triangulation algorithm based on the power center of three circles**. In Research and Education in Robotics (EUROBOT), volume 161 of Communications in Computer and Information Science, pages 248-262, 2011. Springer.

# Triangulation – The *Total* Algorithm

- Assumes 3 fixed beacon locations:



# Triangulation – The *Total* Algorithm

---

- Don't worry about the math. Just need to plug in formulas:
  - STEP 1: compute the modified beacon coordinates ...

$$x'_1 = x_r - x_g$$

$$y'_1 = y_r - y_g$$

$$x'_3 = x_b - x_g$$

$$y'_3 = y_b - y_g$$

- STEP 2: compute the three cotangents ...

$$T_{12} = 1 / \tan(\theta_g - \theta_r)$$

$$T_{23} = 1 / \tan(\theta_b - \theta_g)$$

$$T_{31} = \frac{1 - T_{12} * T_{23}}{T_{12} + T_{23}}$$

# Triangulation – The *Total* Algorithm

---

- STEP 3: compute the modified circle center coordinates...

$$x'_{12} = x'_1 + T_{12} * y'_1$$

$$y'_{12} = y'_1 - T_{12} * x'_1$$

$$x'_{23} = x'_3 - T_{23} * y'_3$$

$$y'_{23} = y'_3 + T_{23} * x'_3$$

$$x'_{31} = (x'_3 + x'_1) + T_{31} * (y'_3 - y'_1)$$

$$y'_{31} = (y'_3 + y'_1) - T_{31} * (x'_3 - x'_1)$$

- STEP 4: compute  $k'_{31}$ ...

$$k'_{31} = x'_1 * x'_3 + y'_1 * y'_3 + T_{31} * (x'_1 * y'_3 - x'_3 * y'_1)$$

- STEP 5: compute  $d$  ...

$$d = (x'_{12} - x'_{23}) * (y'_{23} - y'_{31}) - (y'_{12} - y'_{23}) * (x'_{23} - x'_{31})$$

# Triangulation – The *Total* Algorithm

- if  $d = 0$ , then there is no solution
  - corresponds to situation when robot is on perimeter of circle defined by beacons
  - no algorithm can handle this case
  - can detect when this happens by examining  $d$ .
  - As  $d$  gets smaller, error grows ... so if, for example,  $|d| < 100$  or so ... then the calculation may be inaccurate.

if ( $Math.abs(d) < 100$ )  
...

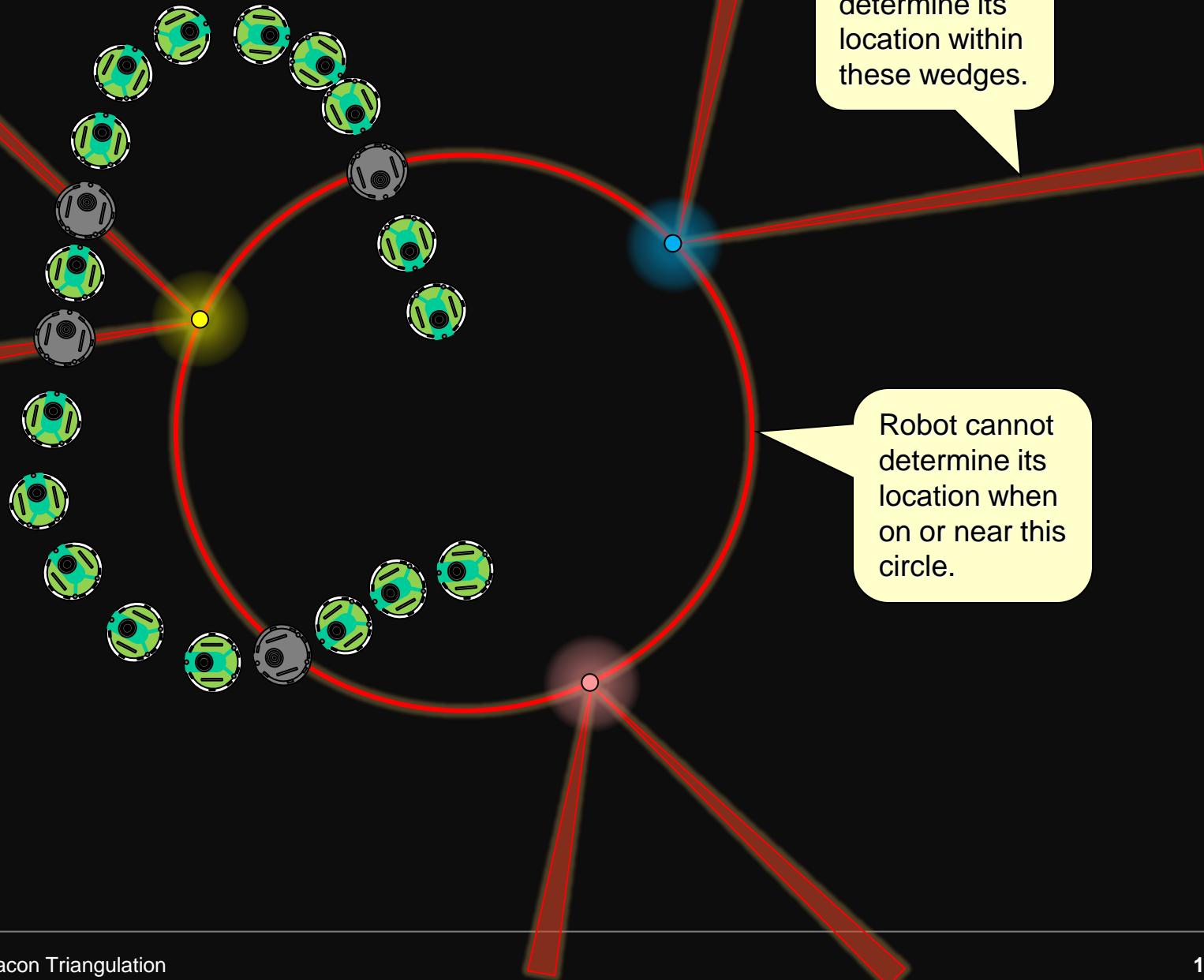
- STEP 6: compute the robot position ( $x, y$ ) ...

$$x = x_g + k'_{31} * (y'_{12} - y'_{23}) / d$$

$$y = y_g + k'_{31} * (x'_{23} - x'_{12}) / d$$



# Triangulation - Issues

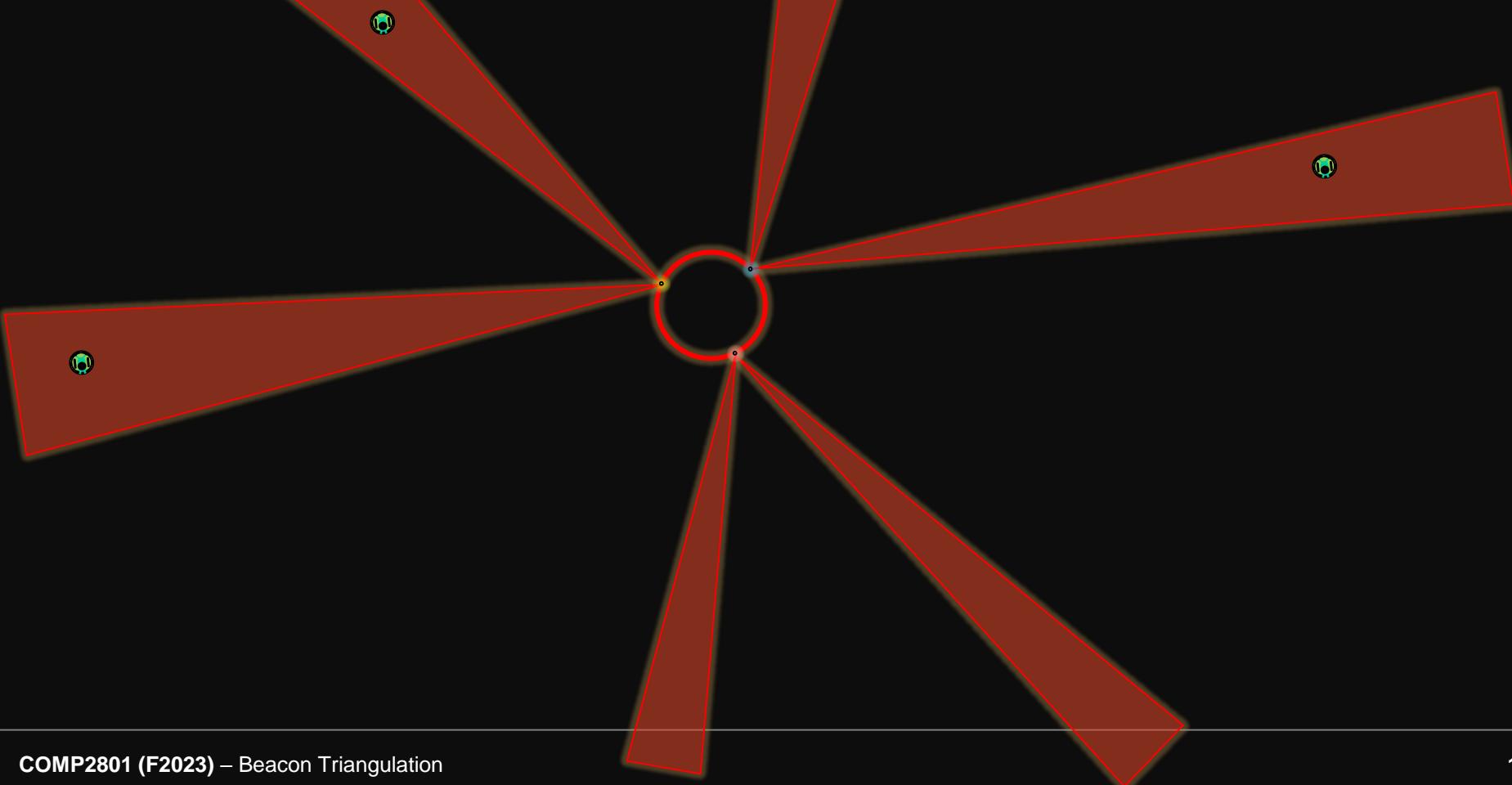


Robot cannot determine its location within these wedges.

Robot cannot determine its location when on or near this circle.

# Triangulation - Issues

- As robot gets further away, wedges get wide.
  - Cannot determine location accurately as robot gets far away from beacons or if beacons placed too close together.



# E-Puck - Measuring Angles

- How does robot measure  $\theta_r$ ,  $\theta_g$  and  $\theta_b$  ?



Initialize an array for the 3 angles:

```
double angles[] =  
{-999, -999, -999};
```

$B_{red}$

store red one  
at angles[0]

$a_{start} = 90^\circ$



- Spin robot around
- Look for beacons centered
- Keep track of rotation angle
- Counter-clockwise direction ensures positive angles

$B_{blue}$



store blue one  
at angles[2]

Rotate 360°  
looking for  
beacons  
using camera

$B_{green}$

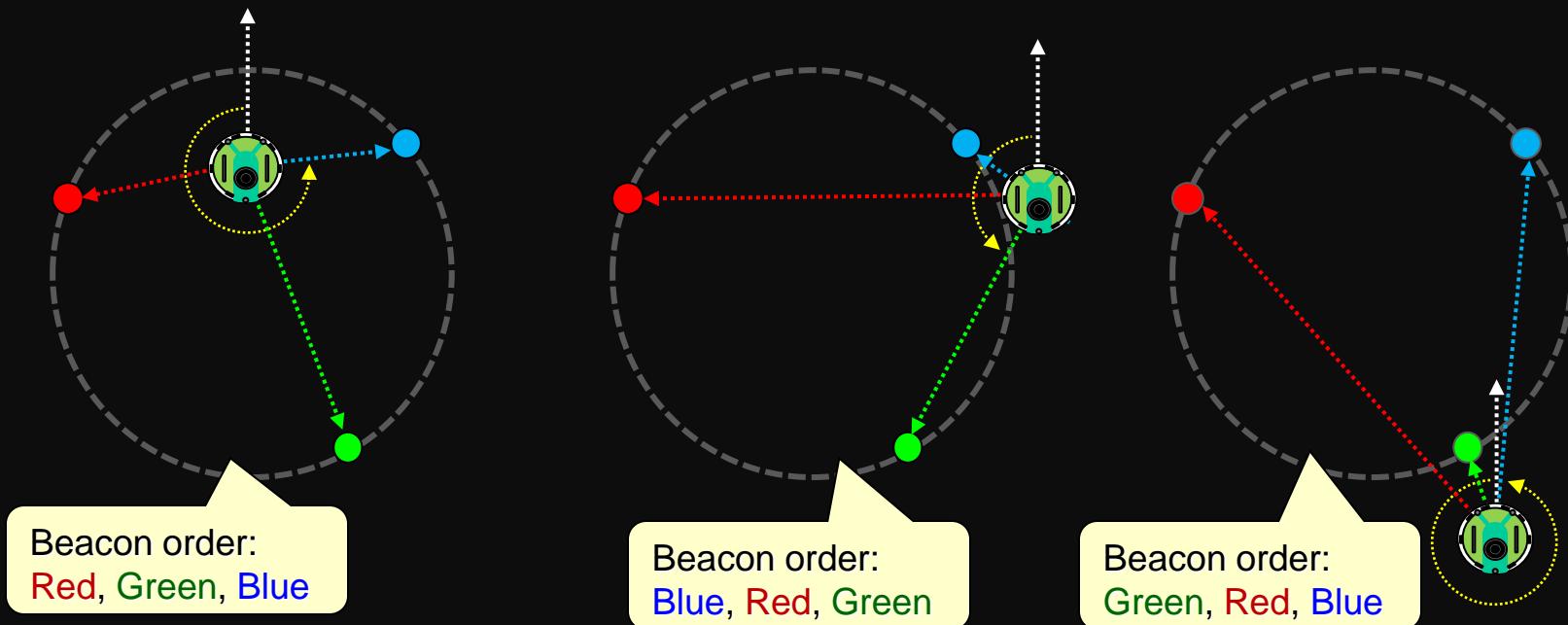
store green one  
at angles[1]



After a full  
rotation ...  
if any angle  
is still -999  
then a  
beacon was  
not found ...  
there is no  
solution.

# E-Puck – Beacon Ordering

- Keep in mind, the beacons may be encountered in a different order each time, depending on robot's location:



- In the array, just make sure that **red** goes at position 0, **green** at position 1 and **blue** at position 2.

# E-Puck - Measuring Angles

- Use wheel position sensors to measure angle while spinning
- Need to rotate 360°
- While rotating, look for the beacons

**spinRadians** is the number of radians that the wheels should turn in order for the robot to spin a full circle.

```
spinRadians = PI * WHEEL_BASE / WHEEL_RADIUS
Start rotating robot counter-clockwise
reading = 0;

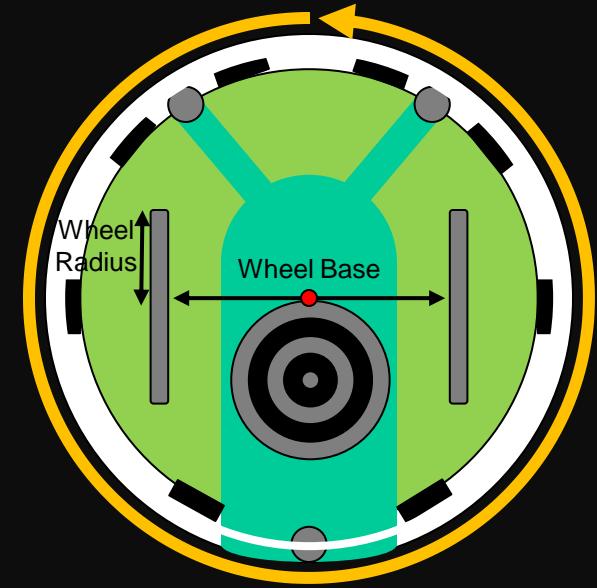
while (reading < spinRadians) {
    reading = readWheelSensor - previousReading;

    // CODE TO LOOK FOR BEACON GOES HERE
    // STORE THE THREE ANGLES θr, θg and θb

}

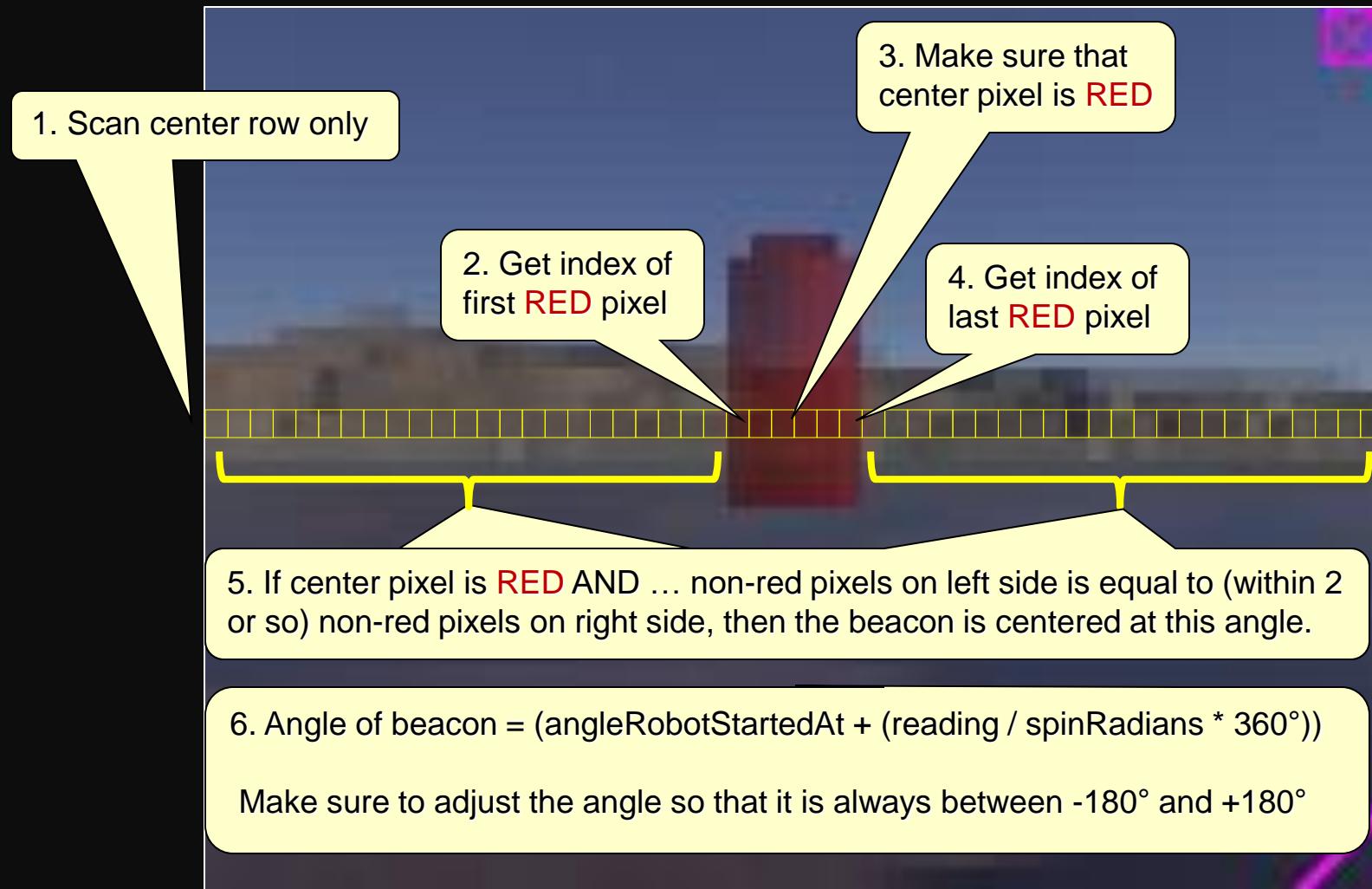
Stop rotating robot
previousReading = readWheelSensor;

Compute the position (x,y) using triangulation method
```



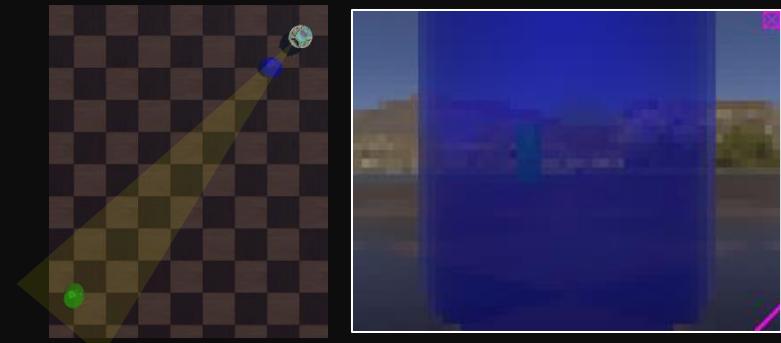
# E-Puck - Measuring Angles

- How to check if a beacon is centered at this angle:



# E-Puck - Measuring Angles

- You will need to check individually for the green and blue beacons in the same manner.
- It is possible that robot cannot see one of the beacons.
- It is also possible that the robot may see a beacon two or three times ... so just remember the first time it sees it centered (e.g., use a **boolean flag foundRed**)



When robot spins CCW, beacon appears to move right

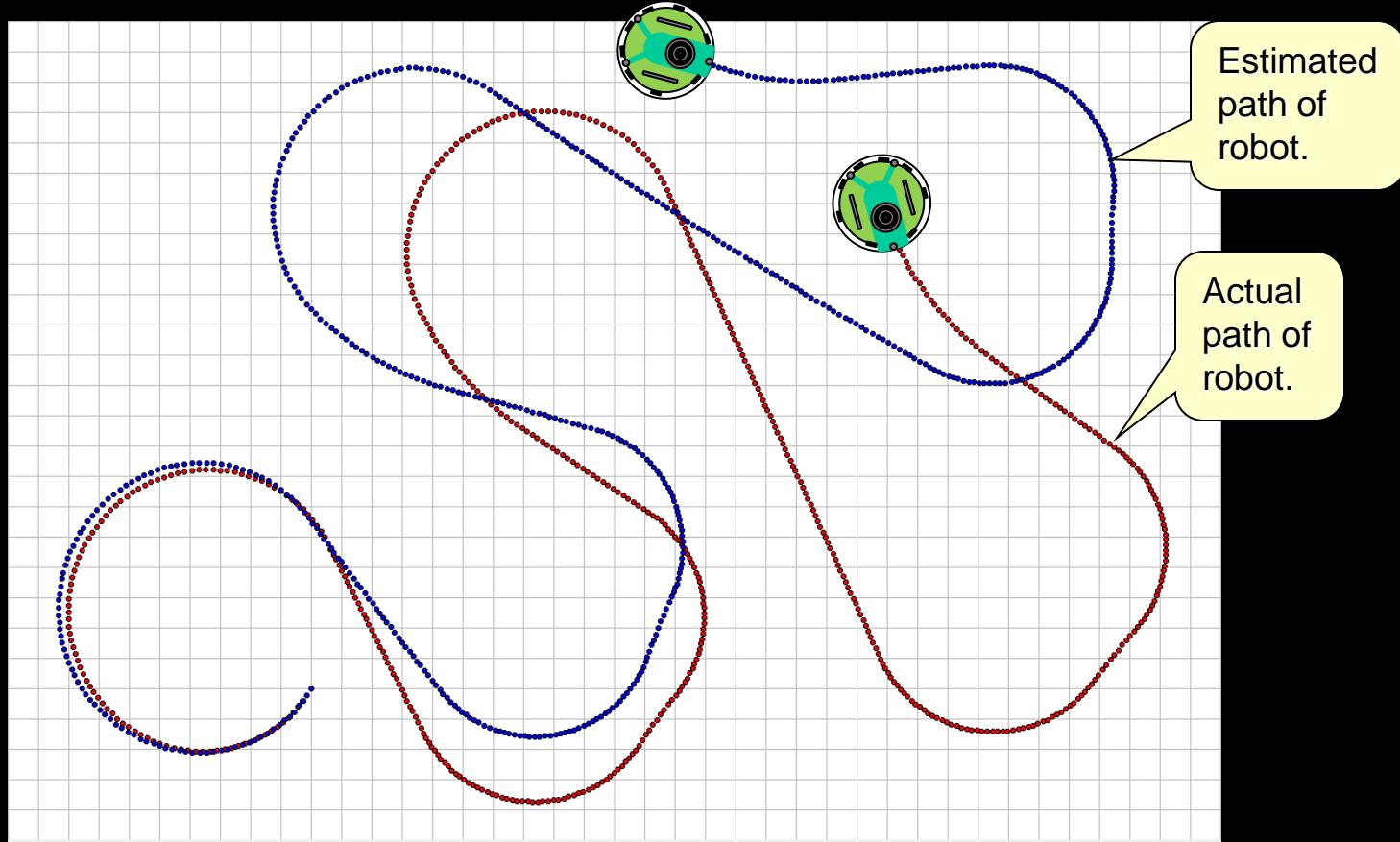


Start the  
Lab ...

# Grid-Based Estimation

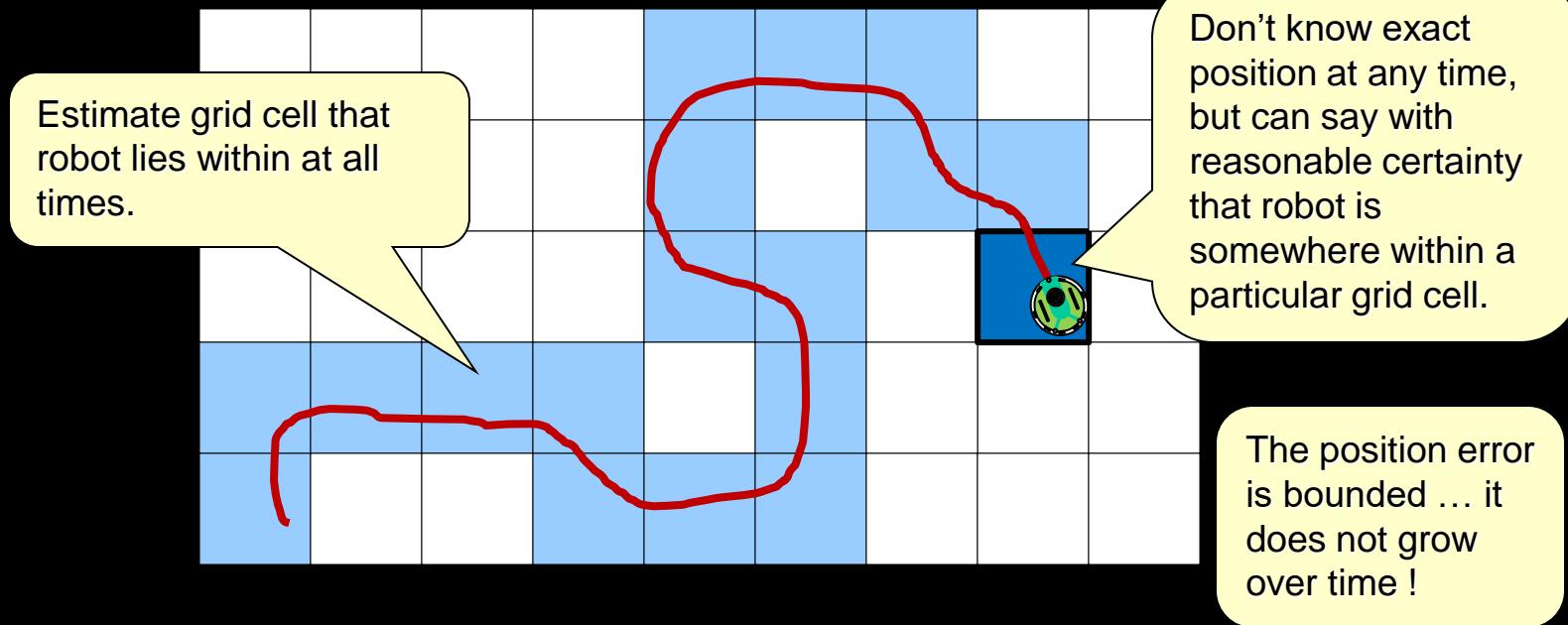
# Odometry Problems

- Using forward kinematics calculations, the odometry error is unbounded and will grow over time.



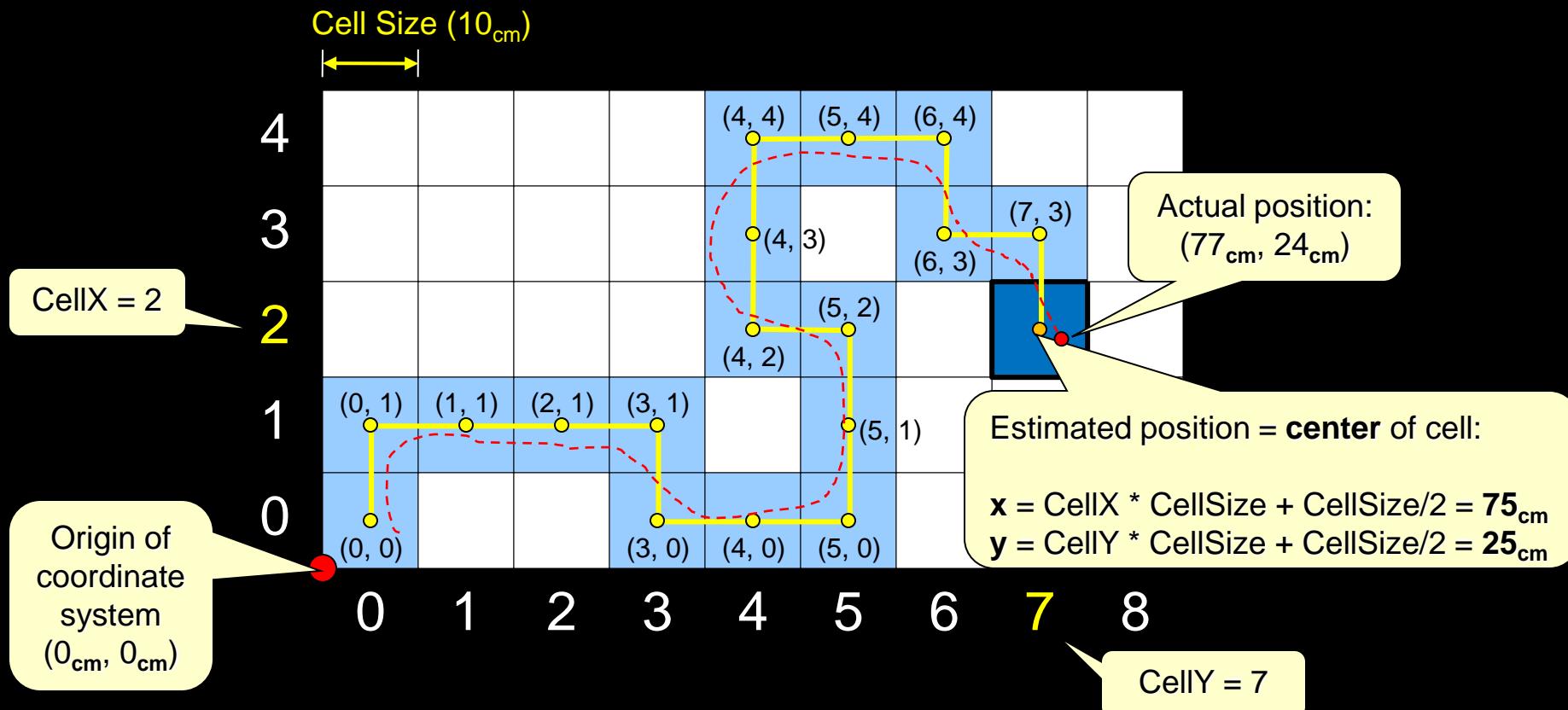
# Grid-Based Estimation

- Another position-estimation approach is to estimate the position of a robot according to its position within a fixed-size grid (just like a GPS grid cell).



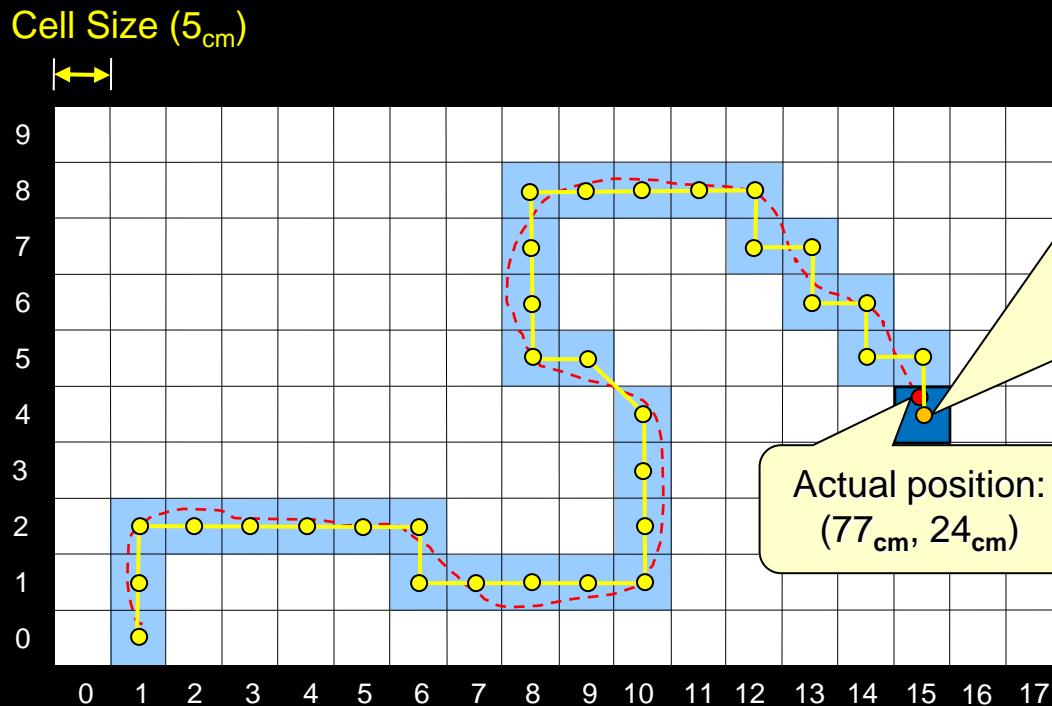
# Grid-Based Estimation

- Always have **rough idea** of robot location at any time.
  - Need to detect when robot crosses from one cell to another, then update cell number accordingly. Assume only local sensors available.



# Grid-Based Estimation

- A fine grid will produce a more accurate result.
- Path will be more “true” to actual robot path.



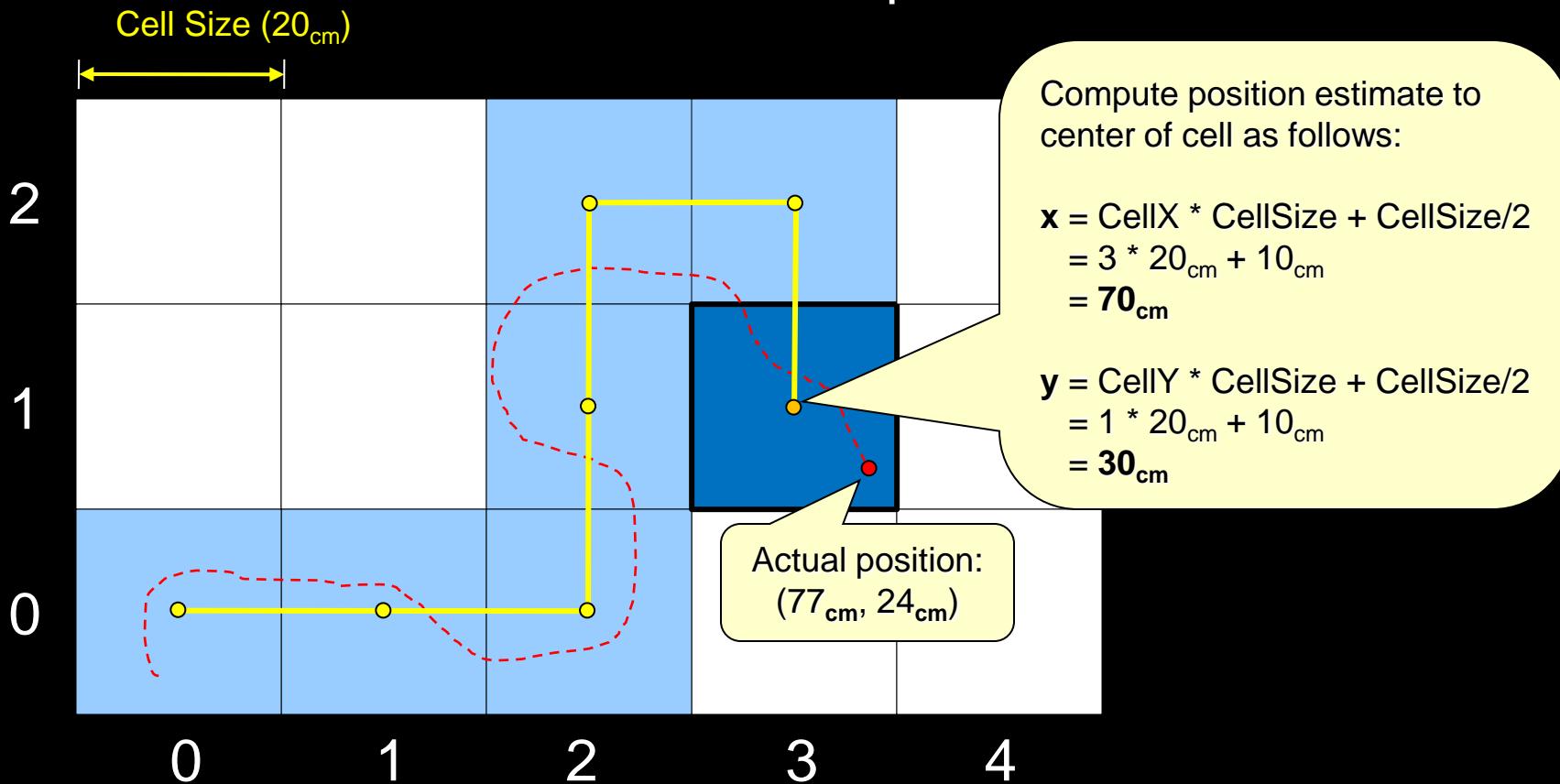
Compute position estimate to center of cell as follows:

$$\begin{aligned}x &= \text{CellX} * \text{CellSize} + \text{CellSize}/2 \\&= 15 * 5_{\text{cm}} + 2.5_{\text{cm}} \\&= 77.5_{\text{cm}}\end{aligned}$$

$$\begin{aligned}y &= \text{CellY} * \text{CellSize} + \text{CellSize}/2 \\&= 4 * 5_{\text{cm}} + 2.5_{\text{cm}} \\&= 22.5_{\text{cm}}\end{aligned}$$

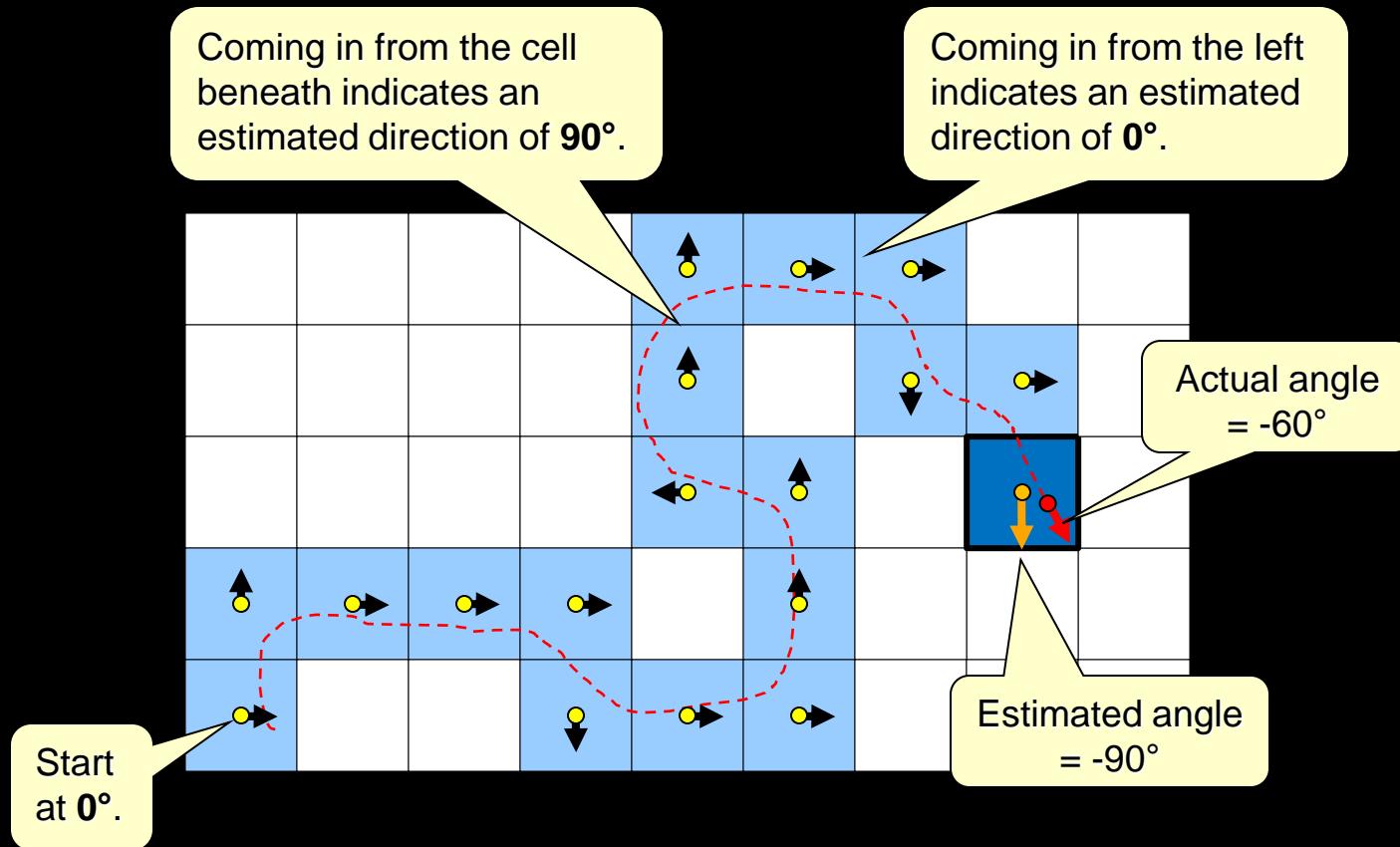
# Grid-Based Estimation

- A coarse grid will be much less accurate.
- Path will be far off from actual robot path.



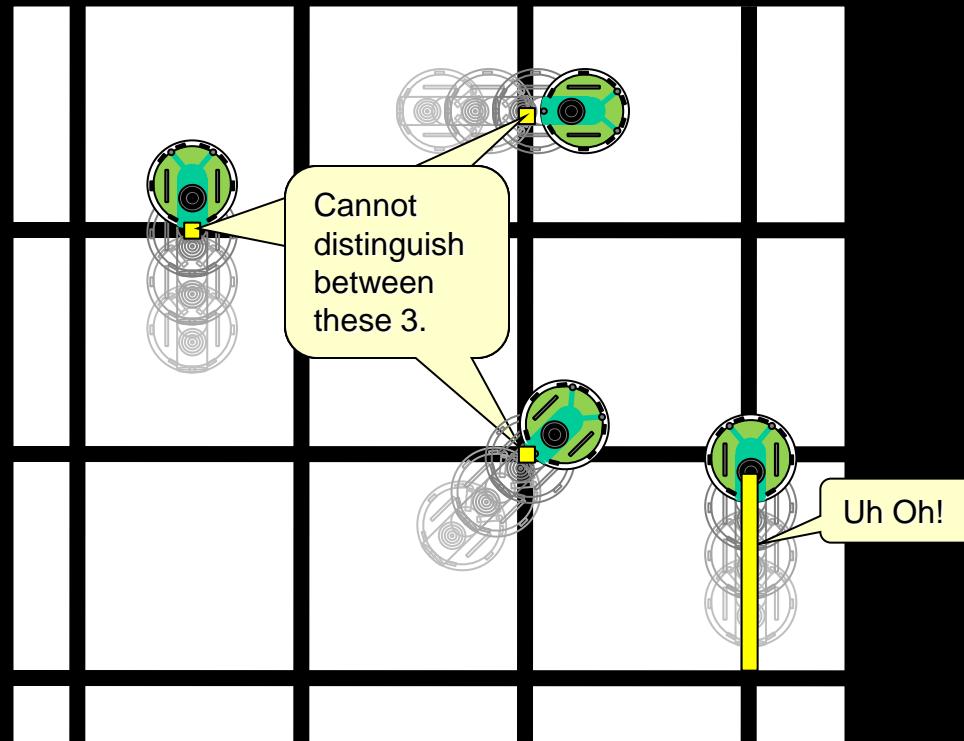
# Grid-Based Estimation

- Can also “estimate” robot’s direction when the robot moves from cell to cell:



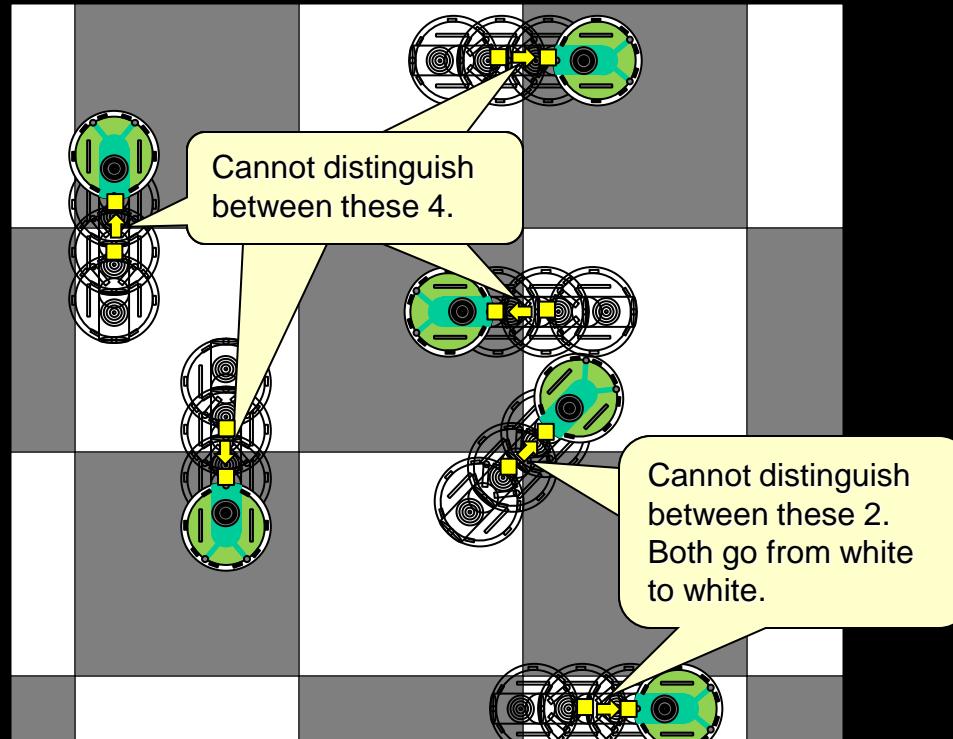
# Detecting Grid Cell-Changes

- How do we know when we cross from one cell to another ?
  - Consider crossing black lines on floor with a light sensor



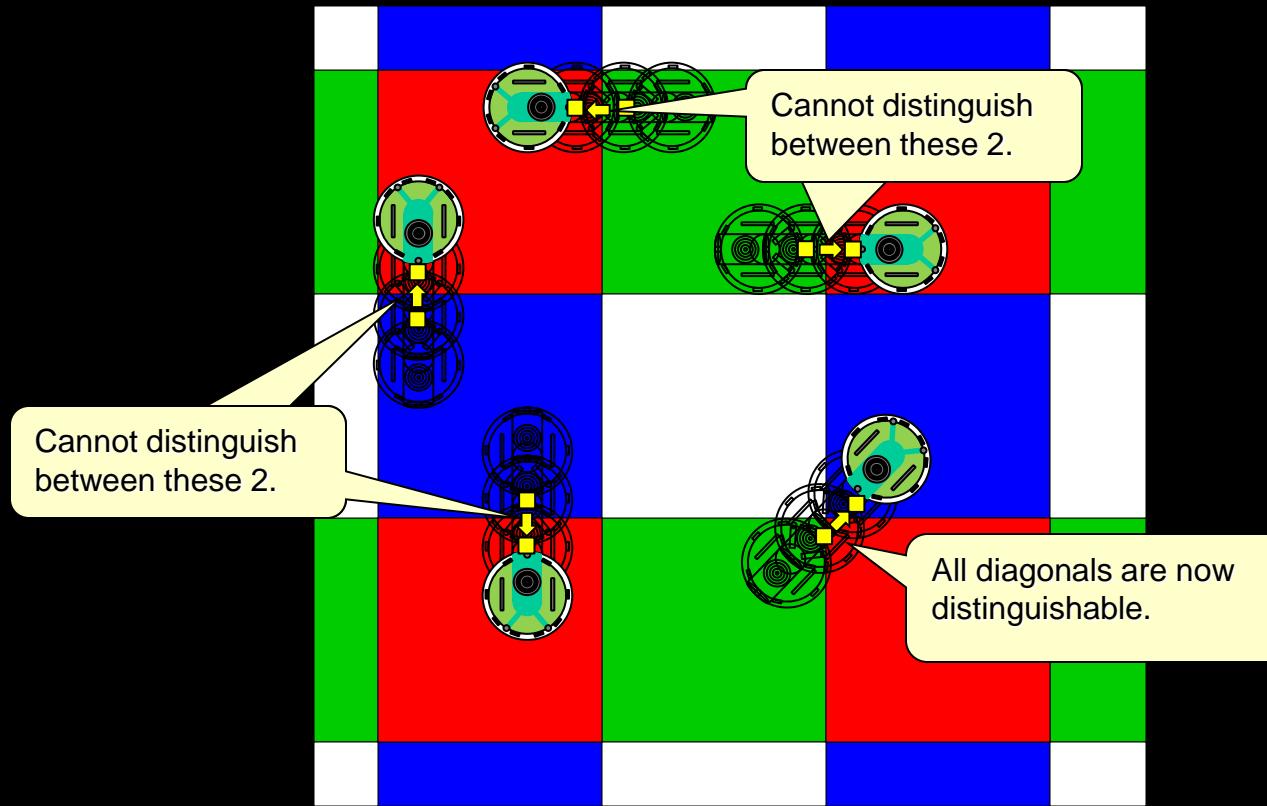
# Detecting Grid Cell-Changes

- Can color cells using checkerboard pattern:
  - Can tell when going from white to black



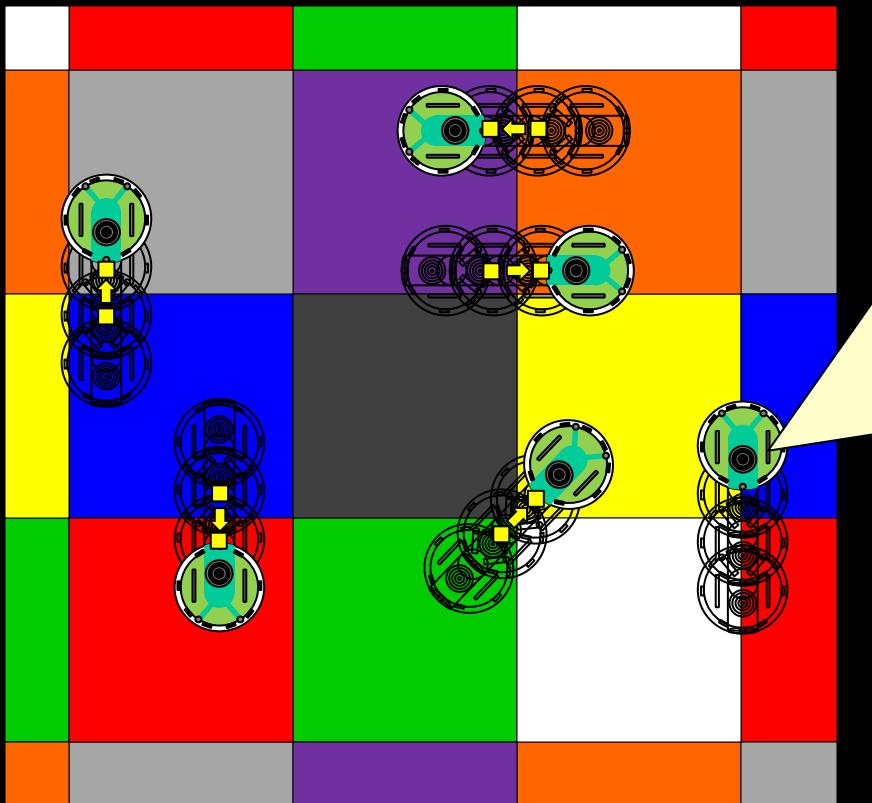
# Detecting Grid Cell-Changes

- Can use 4 colors:
  - Able to detect difference between vertical and horizontal



# Detecting Grid Cell-Changes

- Using 9 colors ensures unique cell-to-cell identification:

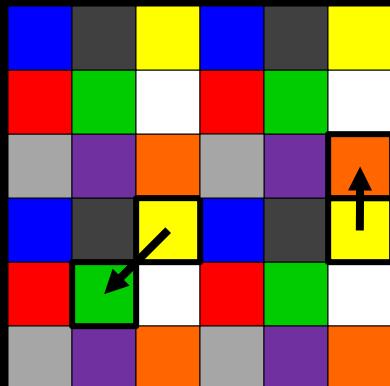


There will still be issues if the sensor lies evenly in two colors, as the reading will not match any one color.

This happens when travelling on the lines horizontally or vertically.

# Detecting Grid Cell-Changes

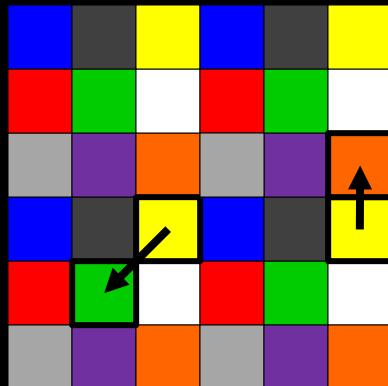
- Can define **OffsetTable** representing grid cell offsets when travelling from one color to each other color.
  - Elements in table indicate (x,y) cell offset when moving from cell to cell.



from\to	Red	Green	White	Blue	Black	Yellow	Gray	Purple	Orange
Red	(0,0)	(1,0)	(-1,0)	(0,1)	(1,1)	(-1,1)	(0,-1)	(1,-1)	(-1,-1)
Green	(-1,0)	(0,0)	(1,0)	(-1,1)	(0,1)	(1,1)	(-1,-1)	(0,-1)	(1,-1)
White	(1,0)	(-1,0)	(0,0)	(1,1)	(-1,1)	(0,1)	(1,-1)	(-1,-1)	(0,-1)
Blue	(0,-1)	(1,-1)	(-1,-1)	(0,0)	(1,0)	(-1,0)	(0,1)	(1,1)	(-1,1)
Black	(-1,-1)	(0,-1)	(1,-1)	(-1,0)	(0,0)	(1,0)	(-1,1)	(0,1)	(1,1)
Yellow	(1,-1)	(-1,-1)	(0,-1)	(1,0)	(-1,0)	(0,0)	(1,1)	(-1,1)	(0,1)
Gray	(0,1)	(1,1)	(-1,1)	(0,-1)	(1,-1)	(-1,-1)	(0,0)	(1,0)	(-1,0)
Purple	(-1,1)	(0,1)	(1,1)	(-1,-1)	(0,-1)	(1,-1)	(-1,0)	(0,0)	(1,0)
Orange	(1,1)	(-1,1)	(0,1)	(1,-1)	(-1,-1)	(0,-1)	(1,0)	(-1,0)	(0,0)

# Detecting Grid Cell-Changes

- Can define AngleTable representing estimated angle when travelling from one color to each other color.
  - Value in table is new angle estimate.



from\to	Red	Green	White	Blue	Black	Yellow	Gray	Purple	Orange
Red	---	0°	180°	90°	45°	135°	-90°	-45°	-135°
Green	180°	---	0°	135°	90°	45°	-135°	-90°	-45°
White	0°	180°	---	45°	135°	90°	-45°	-135°	-90°
Blue	-90°	-45°	-135°	---	0°	180°	90°	45°	135°
Black	-135°	-90°	-45°	180°	---	0°	135°	90°	45°
Yellow	-45°	-135°	-90°	0°	180°	---	45°	135°	90°
Gray	90°	45°	135°	-90°	-45°	-135°	---	0°	180°
Purple	135°	90°	45°	-135°	-90°	-45°	180°	---	0°
Orange	45°	135°	90°	-45°	-135°	-90°	0°	180°	---

# Computing Grid-Based Estimate

$x = 0_{\text{cm}}$ ,  $y = 0_{\text{cm}}$ ,  $\theta = 0^\circ$

Can be any starting location and angle.

$r$  = read ground sensor

$i_p$  = `getColorIndex(r)` as an int from **0-8**, or **9** if no match

**repeat** {

move robot in some way ...

$r$  = read ground sensor

$i_c$  = `getColorIndex(r)` as an int from **0-8**, or **9** if no match

$i_c$  = current index

**if** ( $(i_p \neq i_c) \&& (i_c \neq 9)$ ) **then** {

Only calculate new position if color changed and new color is good

$x = x + \text{OffsetTable}[i_p][i_c].x * \text{CELL\_WIDTH}_{\text{cm}}$

cell width & height are constants

$y = y + \text{OffsetTable}[i_p][i_c].y * \text{CELL\_HEIGHT}_{\text{cm}}$

$\theta = \text{AngleTable}[i_p][i_c]$

Angle does not depend on previous angle

**if** ( $\theta > 180^\circ$ ) **then**  $\theta = \theta - 360^\circ$

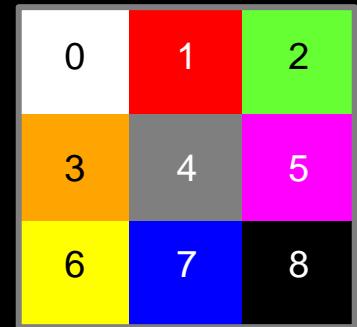
**if** ( $\theta < -180^\circ$ ) **then**  $\theta = \theta + 360^\circ$

Keep angle within  $-180^\circ$  to  $+180^\circ$  range

$i_p = i_c$

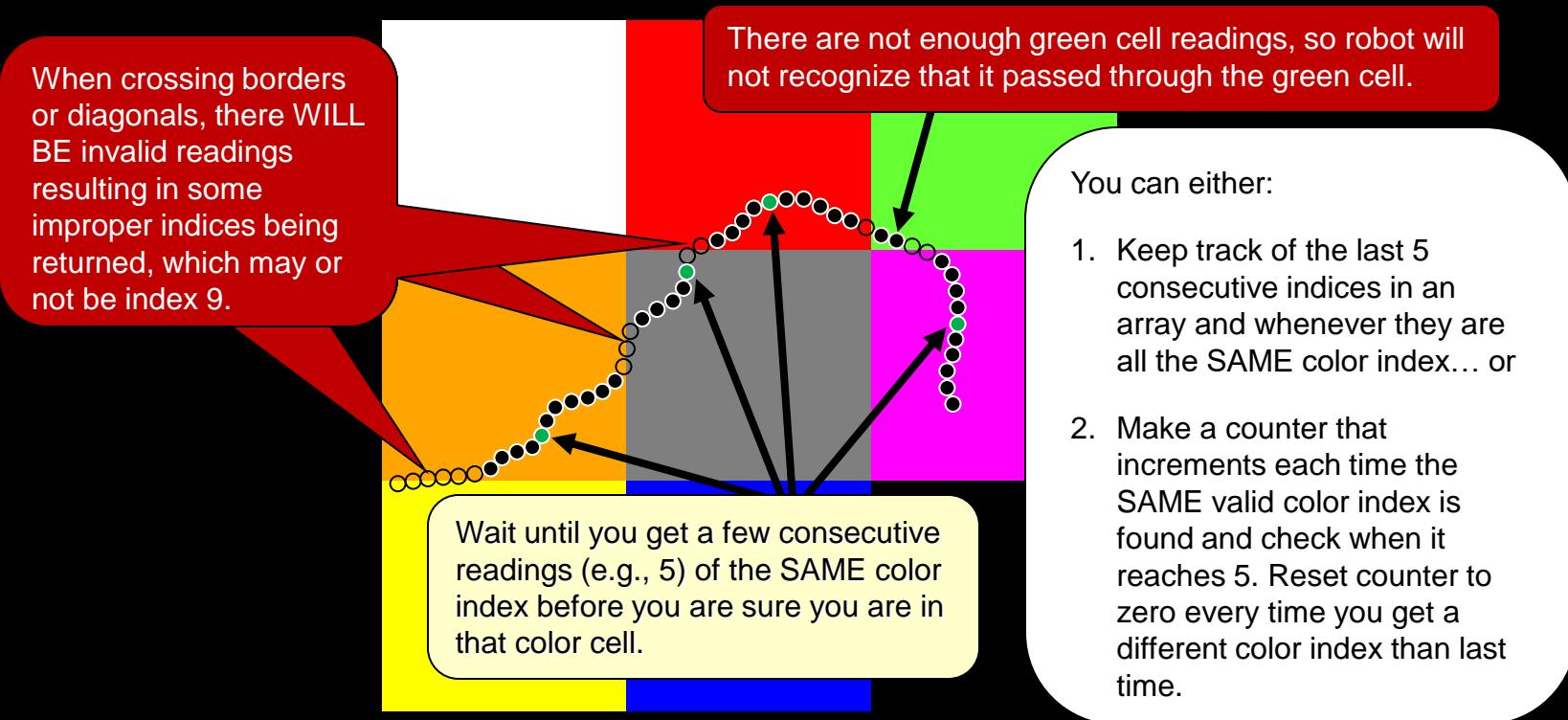
}

}



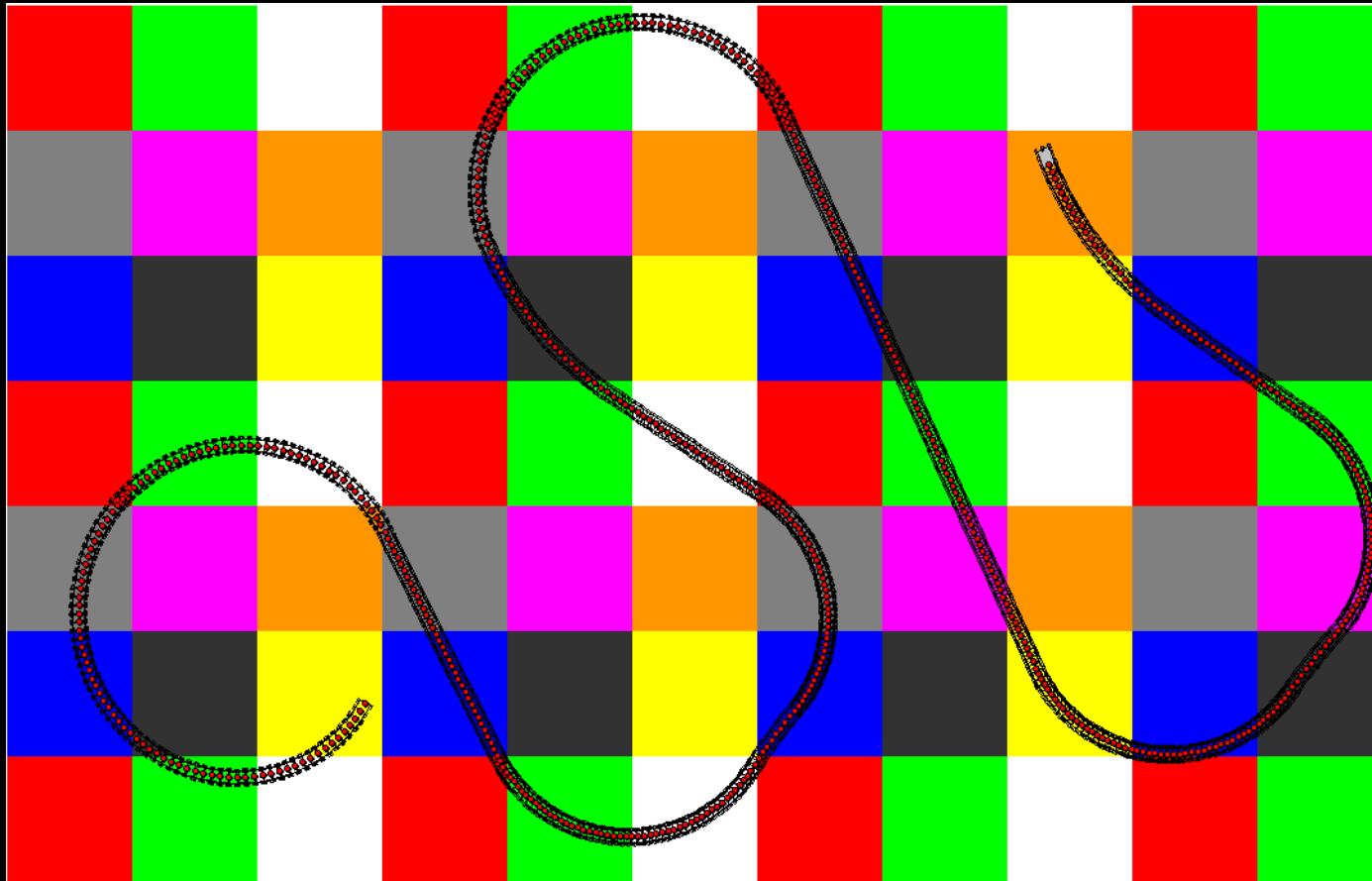
# Handling Consecutive Readings

- There will still be some problems because the robot will get a few spurious/fluctuating/invalid/wrong readings as it crosses diagonals and borders.
  - Just make sure that you have a few (e.g., 5) good readings before you are sure that you are in a cell with a certain color



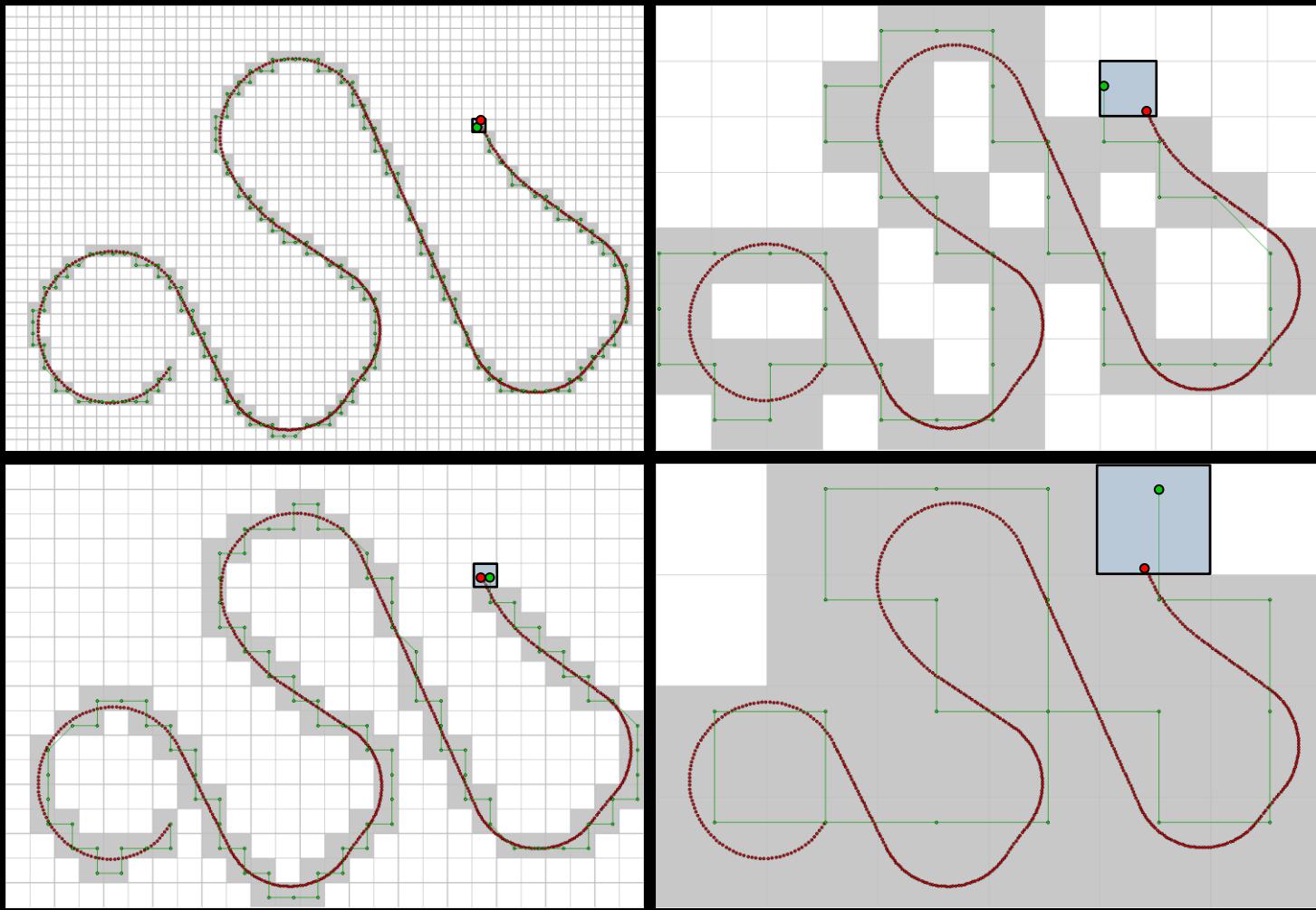
# Grid-Based Experiments

- Consider an entire floor with the colored tile pattern:



# Grid-Based Results

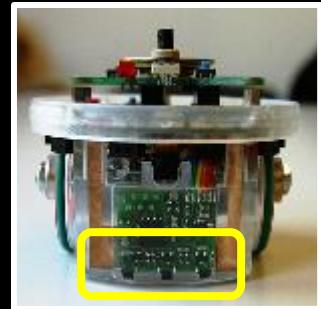
- Here are some results for various cell sizes:



# E-Puck Ground Sensor

- The e-puck has an expansion pack that allows 3 ground sensors to detect the amount of light under the front of the robot.

```
import com.cyberbotics.webots.controller.Device;  
  
// Ground sensor is a Device that behaves like a DistanceSensor  
DistanceSensor groundSensor;  
  
// Go through the devices and look for sensor named "gs1"  
// because it is not part of the standard e-puck robot  
int numDevices = robot.getNumberOfDevices();  
for (int i=0; i<numDevices; i++) {  
    Device d = robot.getDeviceByIndex(i);  
    if (d.getName().equals("gs1")) {  
        groundSensor = (DistanceSensor)d; // Typecast is needed  
        groundSensor.enable(timeStep);  
    }  
}  
  
// while (...) {  
//     Read the sensor like a distance sensor  
//     // returns value from 0 to 1000  
//     double reading = groundSensor.getValue();  
// }
```

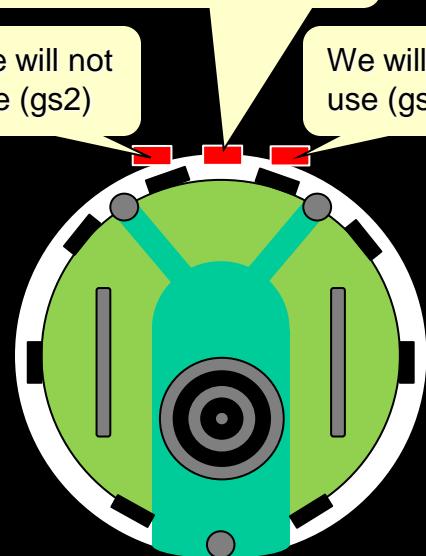


Sensor we will use (gs1)  
is at front center of robot.

We will not  
use (gs2)

We will not  
use (gs0)

You MUST read the  
sensor from INSIDE  
the WHILE loop

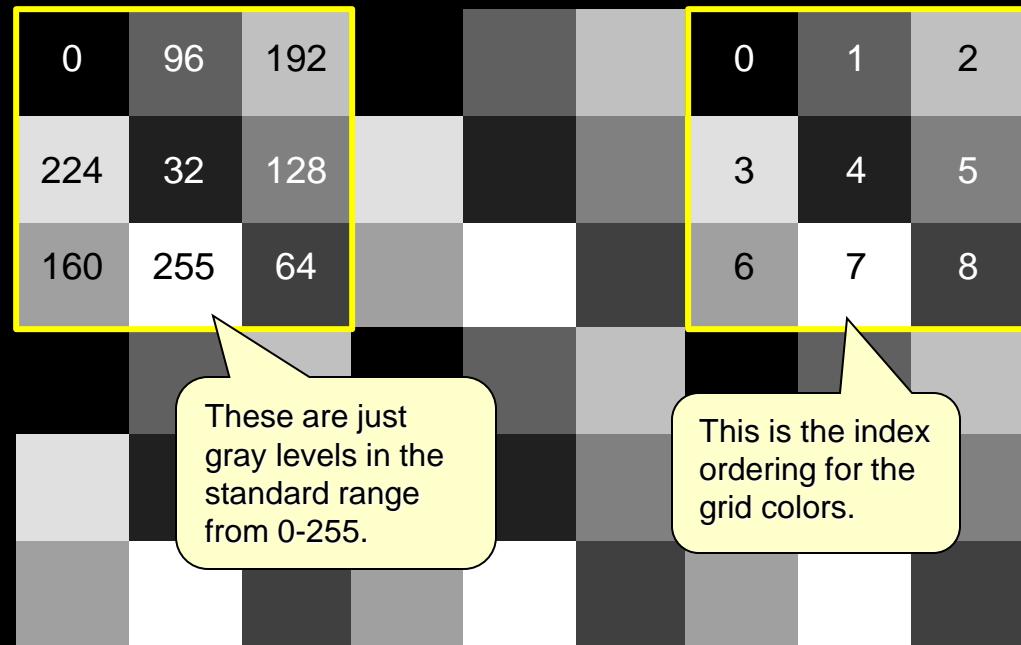


# Webots – Positioning Grid

- Cover the entire floor with a grayscale grid pattern (since ground sensors detect light intensity, not colors).
- 9 gray levels at furthest range apart is  $255/9 \approx 32$  gray shades apart from each other.
- Shades scattered to maximize gray level changes when moving horizontal and vertical

Here is a typical ground sensor reading for the given gray shades. However, the values will fluctuate quite a bit.

300	720	822
842	525	767
800	858	644

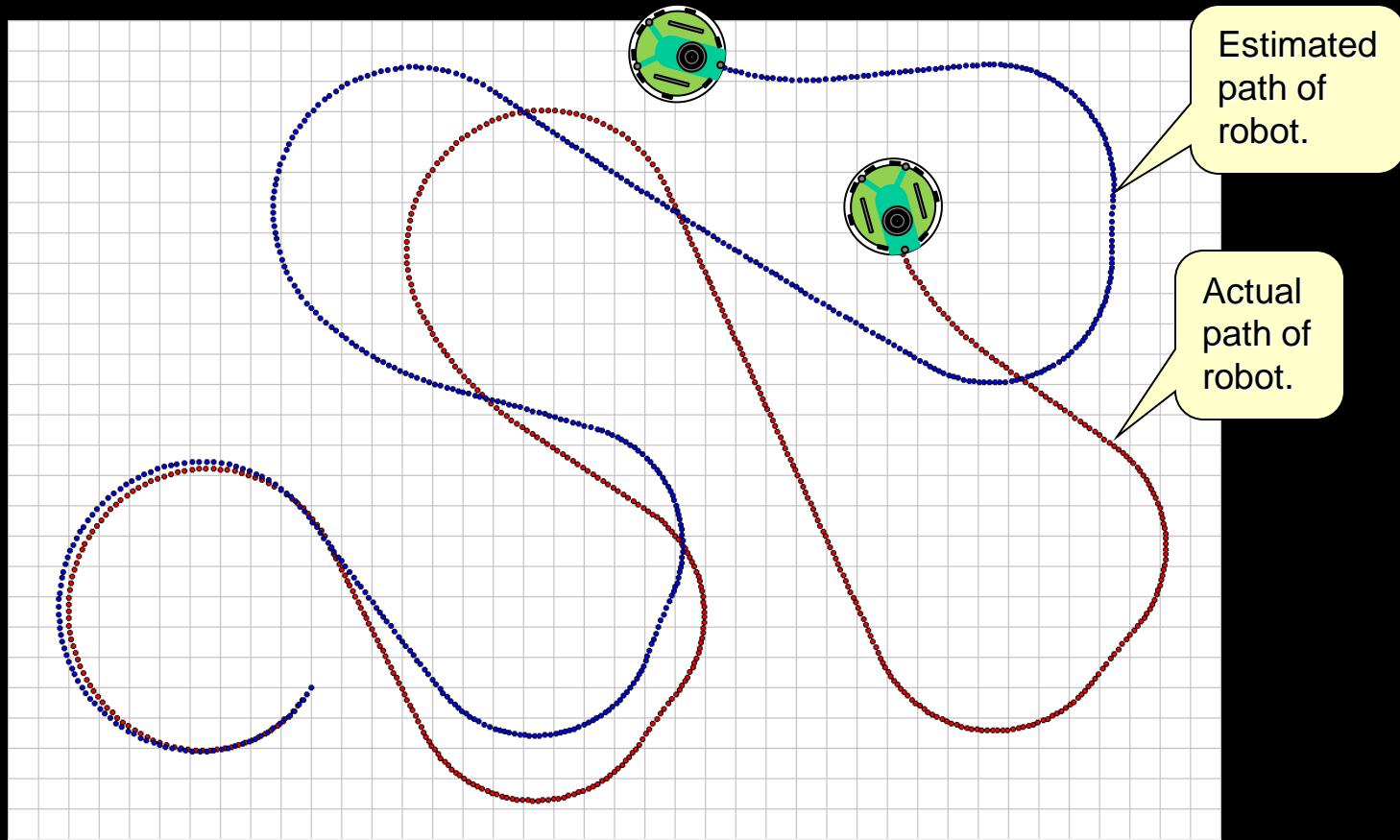


Start the  
Lab ...

# Odometry Correction

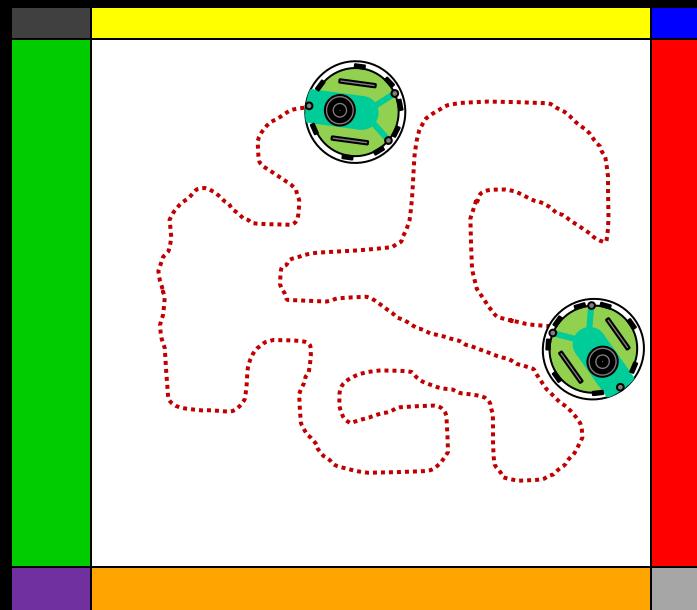
# Odometry Problems

- Recall the odometry error ... unpleasant indeed.



# Grid-Based Estimation Problems

- Grid-based estimation helps (i.e., error is bounded)
- But it requires modification of environment (i.e., must install colored tiles)
- And, if we travel too long on cell boundaries, we lose our spot!
- Also, we cannot determine any position changes within a cell.
  - Can be serious if we are mapping or trying to accomplish some task within the cell.



# Odometry Correction

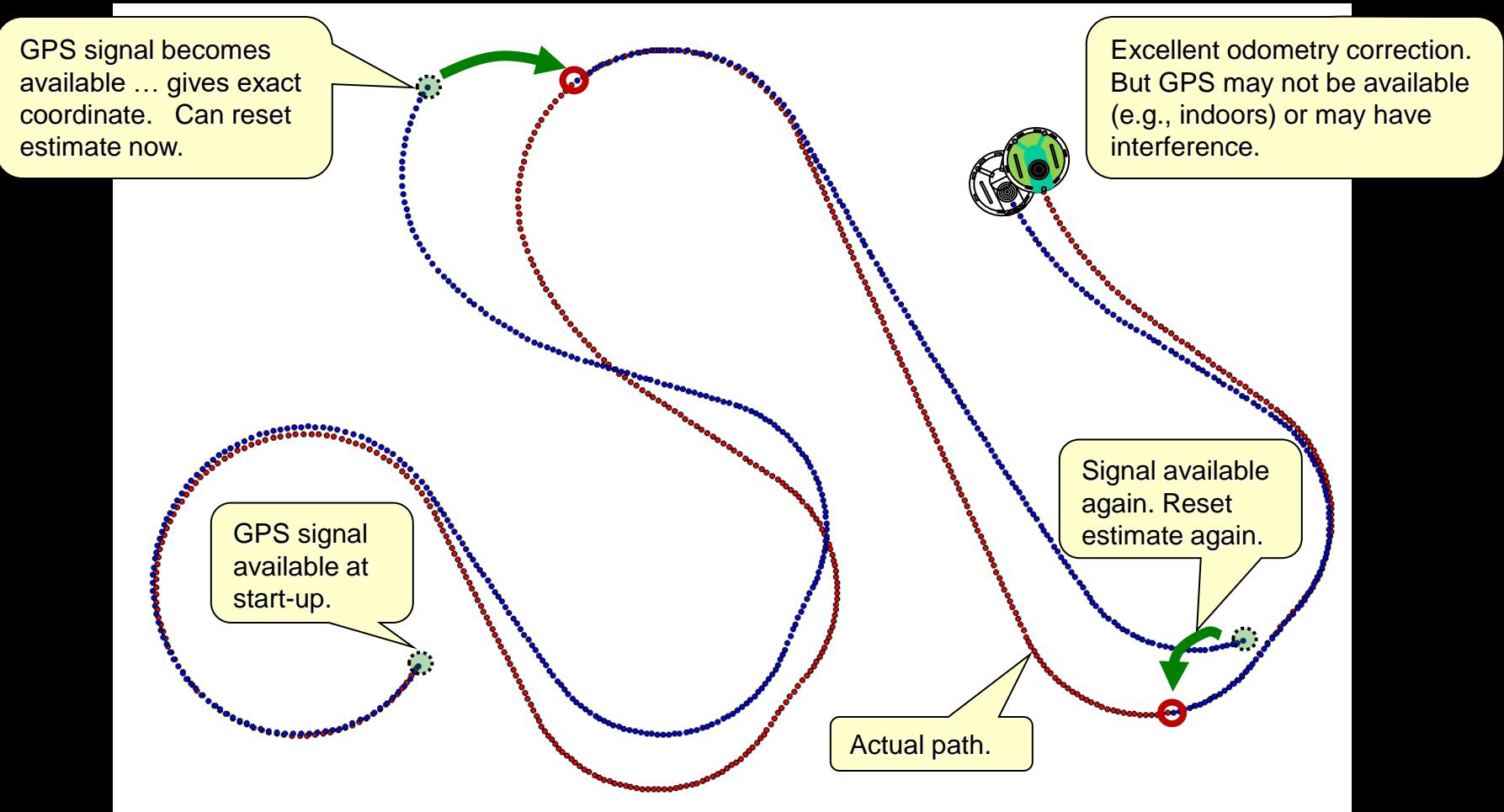
---

- Need to find a way to prevent error from growing too large over time by **re-adjusting estimates** when they become too far off.
- But how do we know **when** the estimate error has grown too much ?
  - Need to compare with some other known data (e.g., gps, compass reading, map, beacons, other estimates, etc...)
  - The more accurate this “other” data is ... the more accurately we will be able to reset the robot’s estimate.



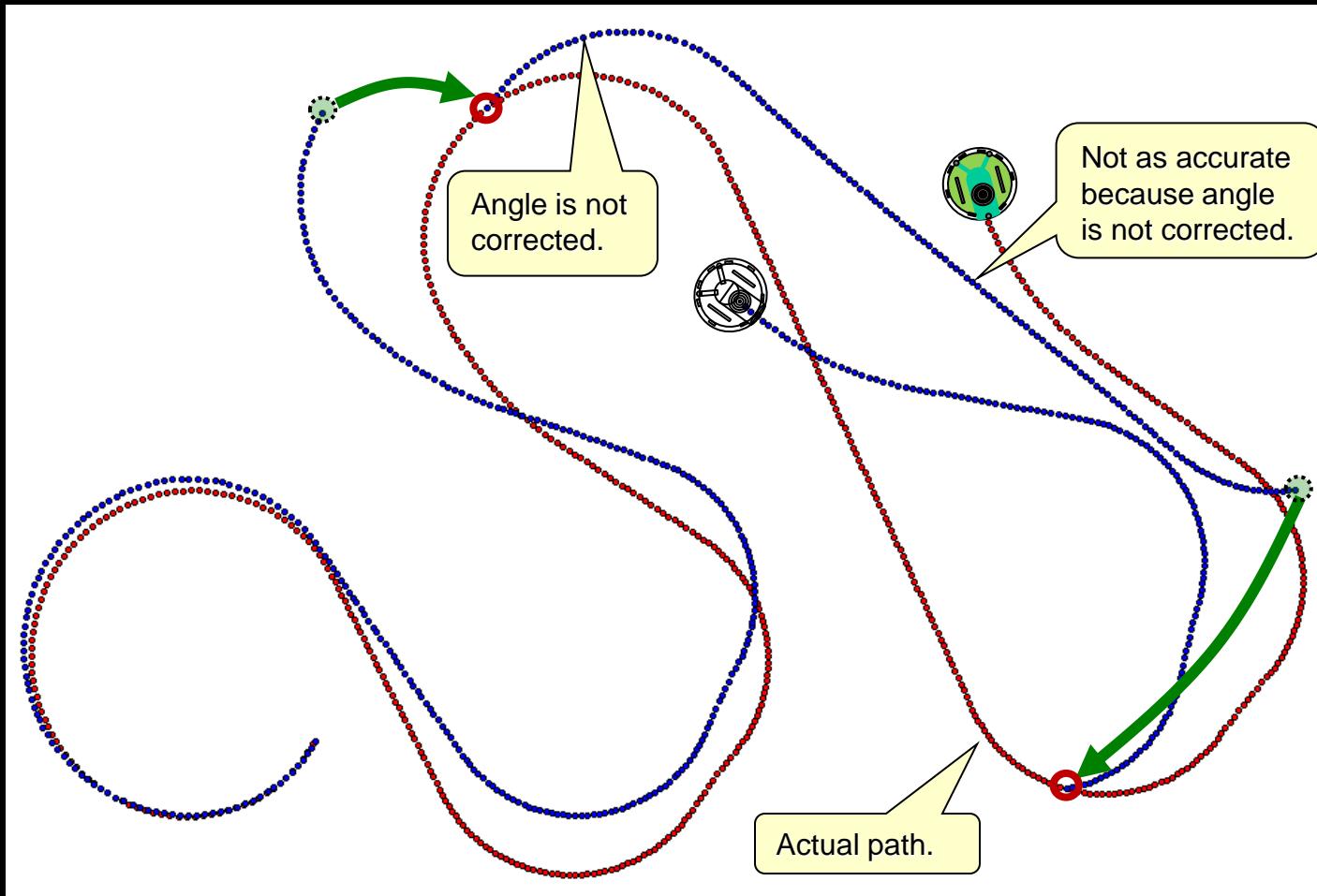
# Odometry Correction – Full GPS

- If exact GPS position is given, estimate can be reset:



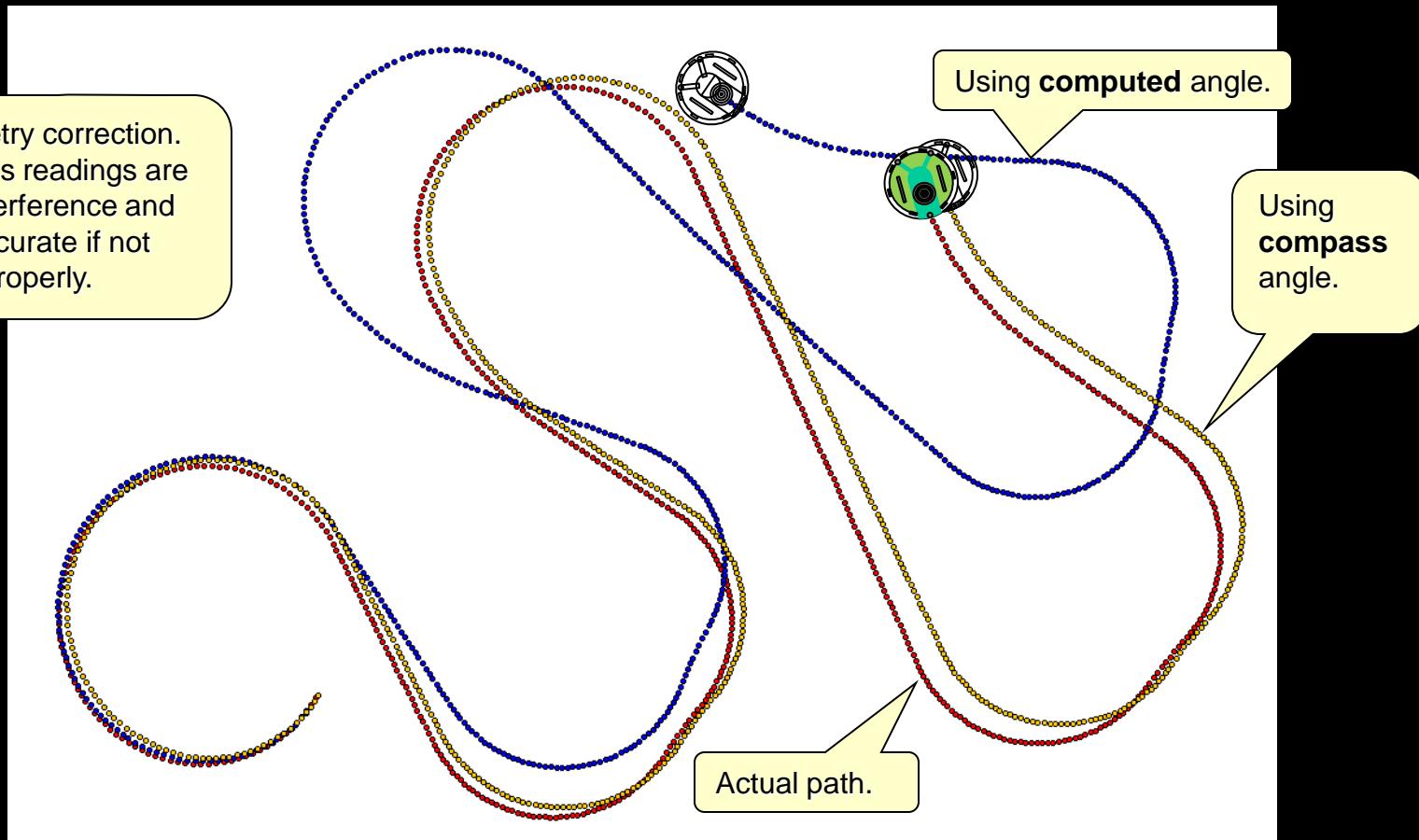
# Odometry Correction – Limited GPS

- If no direction given by GPS, just update (x,y):



# Odometry Correction - Compass

- If just a compass is available, errors in angles can be “fixed” if compass is accurate.



# Odometry Correction - GPS

- Updating the estimate is easy.

$(x_{fk}, y_{fk}, \theta_{fk}) = \text{get forward kinematics estimate}$

First, get an estimate.

**if** (GPS reading is available) **then** {

$x_{fk} = x_{gps}$

$y_{fk} = y_{gps}$

$\theta_{fk} = \theta_{gps}$

Replace kinematics estimate with GPS coordinate.

If direction not available from GPS, leave this line out.

}

**else if** (compass is available) **then** {

$\theta_{fk} = \theta_c$

Replace estimate with compass reading.

}



# Getting Webot's Robot Location

- Need a way of getting **actual position** in order to compare.
- Webots has a **Supervisor** class that replaces the **Robot** class so that the robot can be tracked/supervised.
  - Can get the exact (x, y) location of robot in simulated world.

```
import com.cyberbotics.webots.controller.Supervisor;
import com.cyberbotics.webots.controller.Field;
import com.cyberbotics.webots.controller.Node;

Supervisor robot = new Supervisor();

// Code required for being able to get the robot's location
Node    robotNode = robot.getSelf();
Field   translationField = robotNode.getField("translation");

// while (...) {
//   Get the wheel position sensors
double values[] = translationField.getSFRotation();
x = (values[0]*100);
y = -(values[2]*100); // Need to negate the Y value
// }
```

Convert the values into cm. Need to flip the y value so that origin is at bottom left.

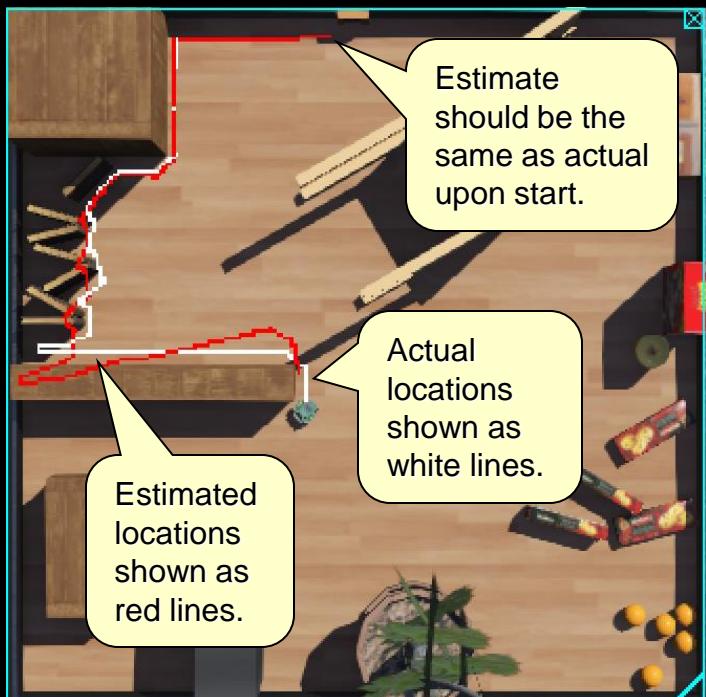
Using **Supervisor** instead of **Robot** now. Must set supervisor flag to TRUE in the scene tree.



Call this each time we want the robot's location.

# Trace Displaying

- A simple Webots **display** window allows you to display the actual robot locations as well as estimated locations as the robot moves.
- Locations must come in sequence, along a boundary.



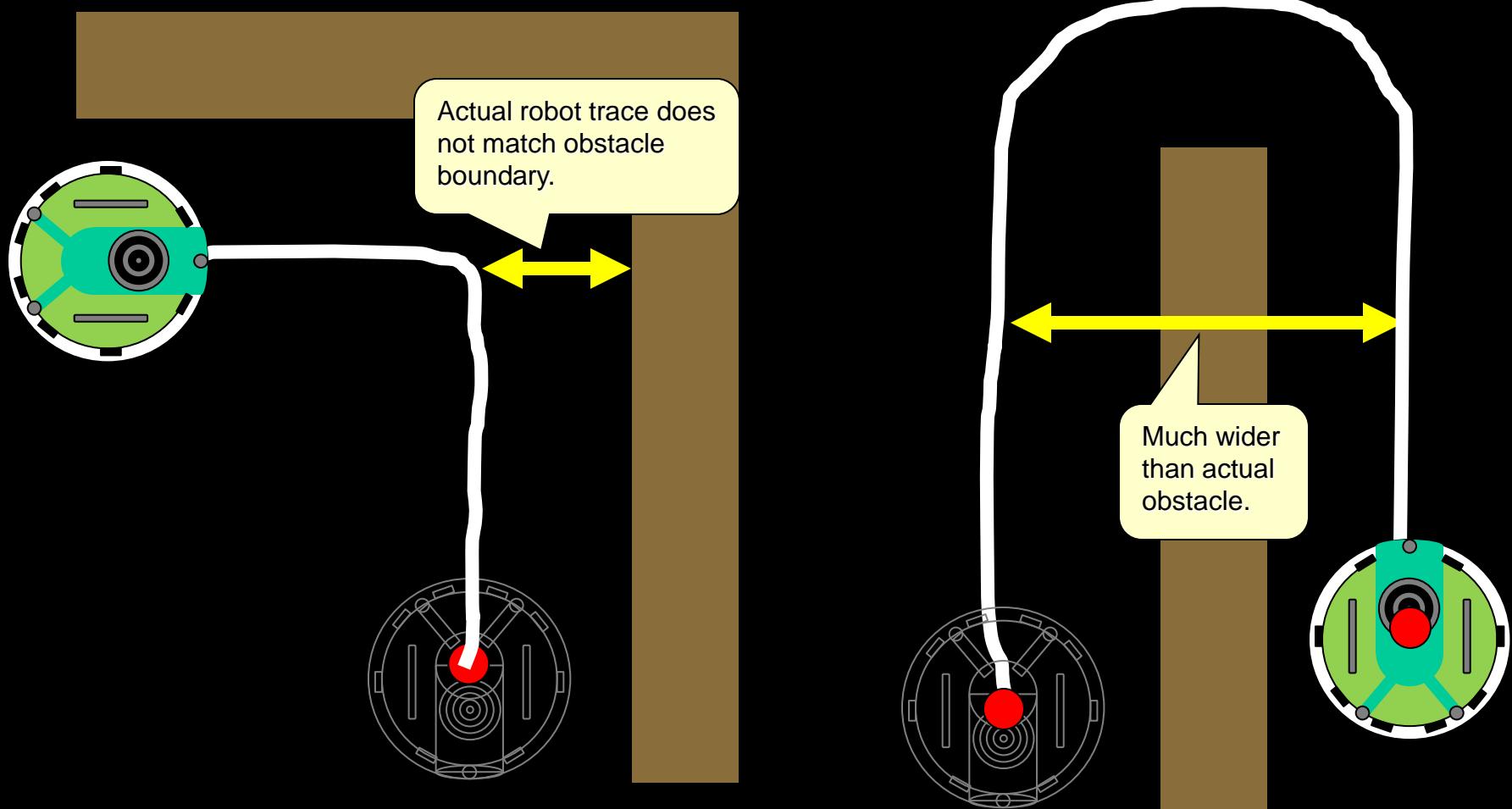
```
TrackerApp tracker;  
tracker = new TrackerApp(robot.getDisplay("display"));  
  
while (...) {  
    ...  
    tracker.addActualLocation(actualX, actualY);  
    tracker.addEstimatedLocation(estimateX, estimateY);  
    ...  
}
```

Do this once

Each time you want to display a new location, call these two functions... one to display the **actual location** of the robot, the other to display your **estimated location**. Both require integer coordinates. Don't forget to negate the y of the actual location (see slide 9)!

# Not a Map

- This is NOT an actual MAP. It only shows locations of the center of the robot.



# Compass

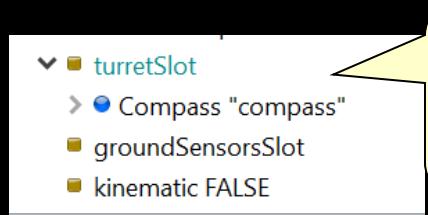
---

- A compass can be used to get an estimate of a robot's orientation with respect to a "North" heading of some sort.
- Compasses have a lot of issues though:
  - an improperly-calibrated compass is nearly useless
    - needs to be calibrated in the environment that the robot is placed in.
  - susceptible to interference from magnetic sources, metal objects, electronics, motors, etc..
- Cannot tell when a compass is giving wrong results.
- Many compasses are inaccurate in practice and so they are not often used due to the above reasons.



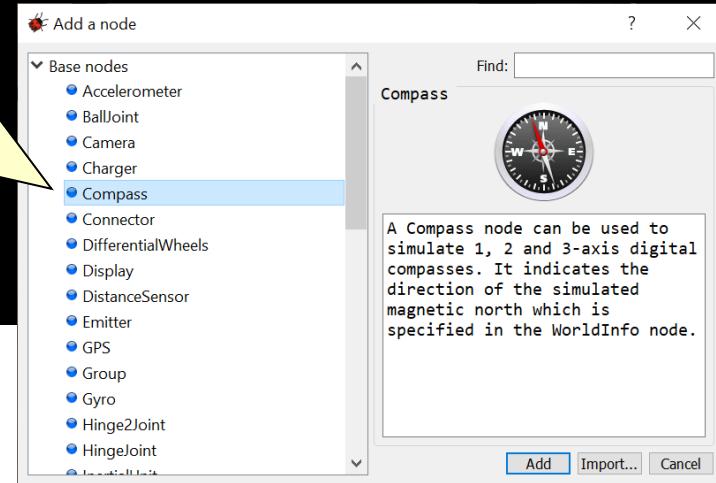
# Webots Compass

- The standard e-puck does not come with a compass. We need to add one to the **turretSlot**.



1. Double-click **turretSlot** under the E-puck in the scene tree.

2. Add the **Compass** from the **Base nodes** in dialog box that appears.



```
import com.cyberbotics.webots.controller.Compass;

// Get Compass sensor
Compass compass = robot.getCompass("compass");
compass.enable(timeStep);

while (...) {
    double compassReadings[] = compass.getValues();
    double rad = Math.atan2(compassReadings[0], compassReadings[1]);
    double bearing = (rad - Math.PI/2) / Math.PI * 180.0;

    if (bearing > 180)
        bearing = 360 - bearing;
    if (bearing < -180)
        bearing = 360 + bearing;
}
```

Do this once.

Adjust the angle so that it is always within the range of **-180°** to **180°**.

It is a little bit of work to get the compass reading in degrees.

Start the  
Lab ...

# Navigation in Unknown Environments

# Navigation

---

- In robotics, **navigation** is the act of moving a robot from one place to another in a collision-free path.
- When navigating, robots either:
  - I. head towards goal location(s) based on sensor input
  - II. follow a **fixed path** (known in advance)
- Robot usually relies on local sensor information and updates its location/direction according to the *perceived* “best” choice that will lead to the goal.
- Assumes there is no map of the environment, otherwise the problem becomes that of path-planning.



# Path Planning

---

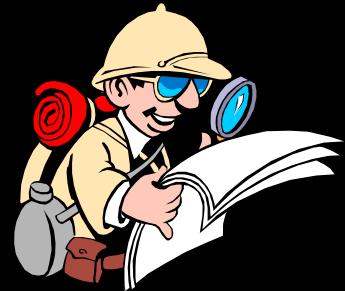
- When a fixed path is provided on which to navigate, the path is usually computed (i.e., planned) beforehand.
- ***Path planning*** is the act of examining known information about the environment and computing a path that satisfies one or more conditions.
  - e.g., avoids obstacles, shortest, least turns, safest etc...
- Key to “good” path planning is efficiency
  - in real robots, optimal solution is not always practical
  - approximate solutions are often sufficient and desired.



# Path Planning

---

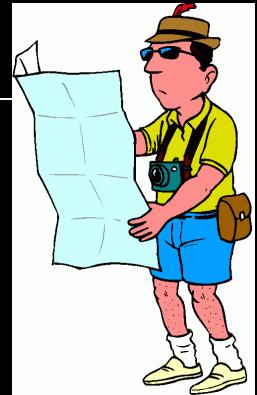
- To accomplish complicated tasks, a mobile robot usually MUST pre-plan its paths.
- Many interesting problems are solved that make use of planned motion of the robot:
  - Efficient collision-free travel (e.g., shortest paths)
  - Environment coverage (e.g., painting, cleaning)
  - Guarding and routing (e.g., security monitoring)
  - Completion of various tasks etc...
- We will look first at **goal-directed** navigation in which the robot is trying to reach a goal location based on sensor readings.



# Goal-Directed Navigation

- Approaches to goal-directed navigation vary depending on two important questions:

- Are robot & goal locations (i.e., coordinates) known ?
  - if available, goal position given as a coordinate, otherwise the problem becomes one of searching.
  - robot would either maintain its own location as it moves (e.g., forward kinematics) or have this information provided externally (e.g., GPS system).
- Are obstacles (i.e., locations and shape) known ?
  - if available, coordinates of all polygonal obstacle vertices would be given and known to the robot.
  - if unavailable, robot must be able to sense obstacles (sensing is prone to error and inaccuracies).



# Goal-Directed Navigation

- Here is a summary of categories for navigating towards a goal location under various conditions:

- Goal Location “Unknown”

- Obstacles Unknown
      - 1. Behaviors (wandering, ball-seeking, wall following etc...)
    - Obstacles Known
      - 1. Search algorithms } Won't be discussed
      - 2. Coverage algorithms

} Already discussed

} Discussed later

- Goal Location “Known”

- Obstacles Unknown (i.e., local sensing information only)
      - 1. Reactive Navigation
    - Obstacles Known (i.e., global information available)
      - 1. Feature-Based Navigation }
      - 2. Potential Field Navigation }
      - 3. Roadmap-Based Planning } Won't be discussed

} Discussed here

} Discussed later

# “Bug” Algorithms

---

- Simple navigation when goal location is **known** but obstacles locations are **unknown** (i.e., no map)
  - robot must know its **location** within the environment at all times (i.e., using forward kinematics, beacon location, grid est., gps etc..)
  - robot must have **sensors** to detect and follow obstacle boundaries
- There are three simple algorithms for this scenario:
  - Bug1
  - Bug2
  - Tangent Bug

Not discussed here



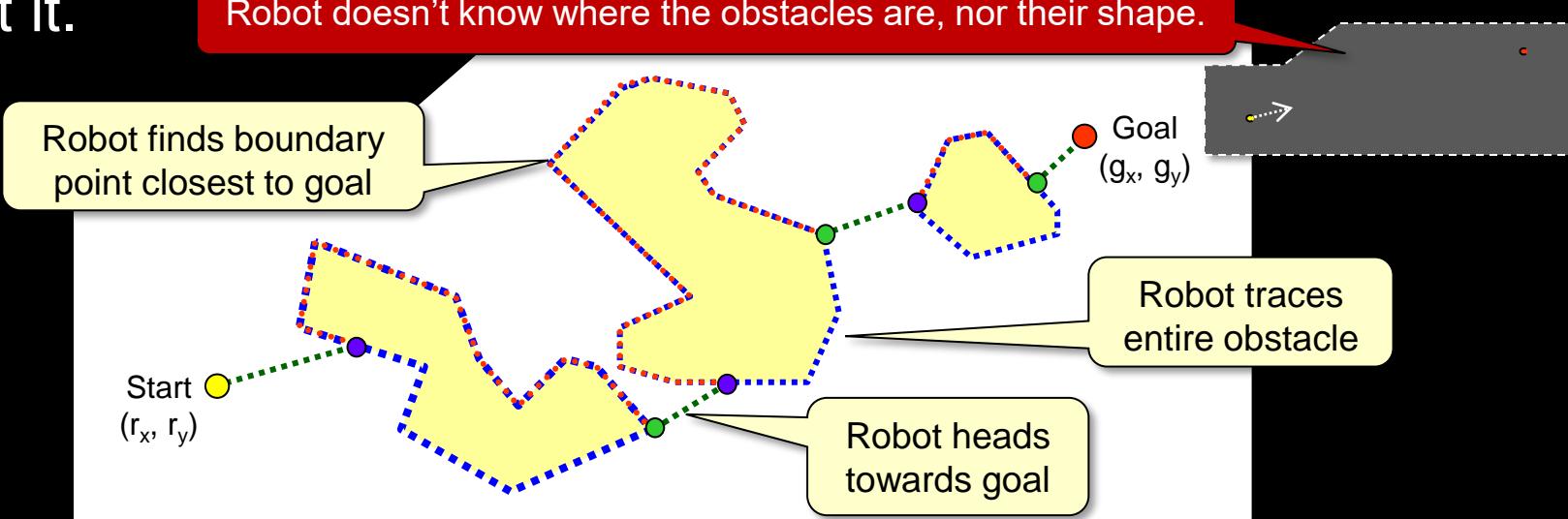
# The Bug1 Algorithm

- Bug1 Strategy:

- Move toward goal unless obstacle encountered, then go around obstacle and find its closest point to the goal.
- Travel back to that closest point and move towards goal.

- Assumes robot knows goal location but is unable to see or detect it.

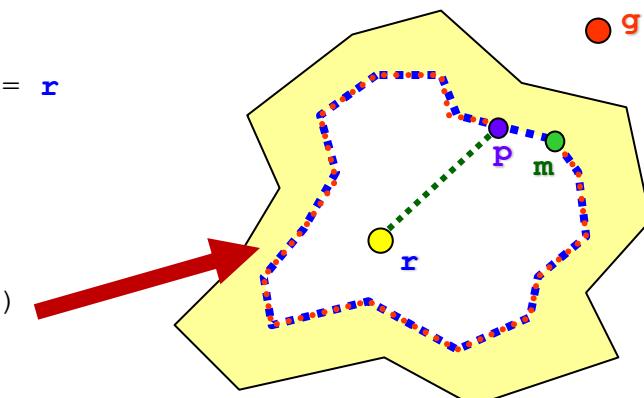
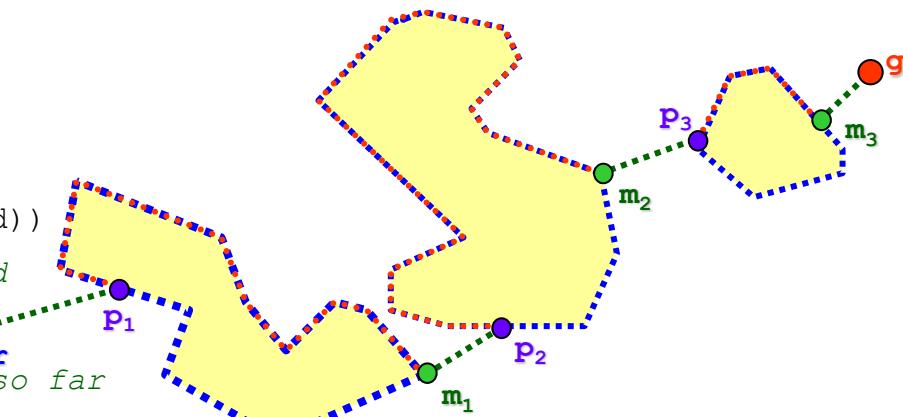
Robot doesn't know where the obstacles are, nor their shape.



# The Bug1 Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
    REPEAT
        Move from r towards g
        r = robot's current location
    UNTIL ((r == g) OR (obstacleIsEncountered))
    IF (r == g) THEN quit // goal reached
    LET p = r // contact location
    LET m = r // location closest to g so far
    REPEAT
        Follow obstacle boundary
        r = robot's current location
        IF ((distance(r,g) < distance(m,g)) THEN m = r
    UNTIL ((r == g) OR (r == p))
    IF (r == g) THEN quit // goal reached
    Move to m along obstacle boundary
    IF (obstacleIsEncountered at m in direction of g)
        THEN quit // goal not reachable
ENDWHILE
```



# The Bug1 Algorithm

- This algorithm:
  - always finds goal location (if it is reachable).
  - performs an exhaustive search for the “best” point to leave the obstacle and head towards the goal.
- If we denote the perimeter of an obstacle  $\text{Obj}_i$  as  $\text{perimeter}(\text{Obj}_i)$ , then the robot may travel a distance of:

$$|\overline{sg}| + 1.5 * [\text{perimeter}(\text{Obj}_1) + \text{perimeter}(\text{Obj}_2) + \dots + \text{perimeter}(\text{Obj}_n)]$$

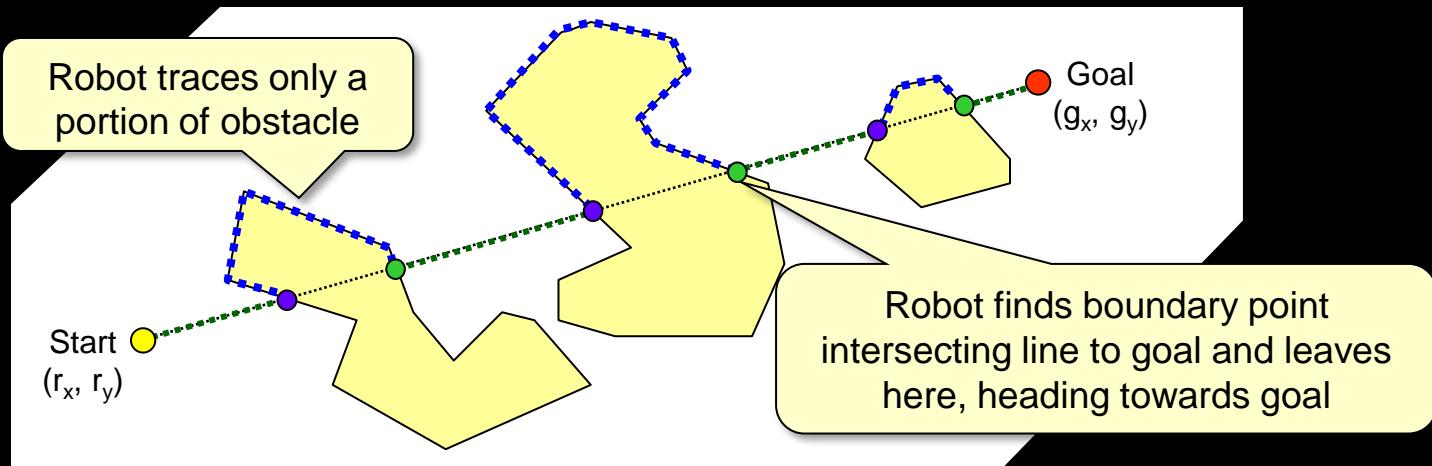


**s** = start location

Once around the obstacle to determine best position to leave from and up to  $\frac{1}{2}$  times around to get back to that position (since we can take the shorter of the two choices).

# The Bug2 Algorithm

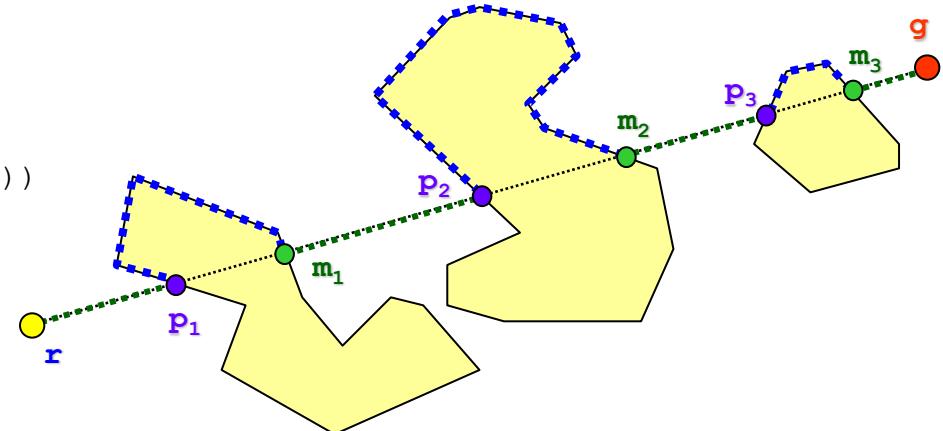
- A variation to this algorithm will allow the robot to avoid traveling ALL the way around the obstacles.
- Bug2 Strategy:
  - Move toward goal unless obstacle encountered, then go around obstacle. Remember the line from where the robot encountered the obstacle to the goal and stop following when that line is encountered again.



# The Bug2 Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
    LET L = line from r to g
    REPEAT
        Move from r towards g
        r = robot's current location
    UNTIL ((r == g) OR (obstacleIsEncountered))
    IF (r == g) THEN quit // goal reached
    LET p = r           // contact location
    REPEAT
        Follow obstacle boundary
        r = robot's current location
        LET m = intersection of r and L
    UNTIL (((m is not null) AND (dist(m,g) < dist(p,g))) OR (r == g) OR (r == p))
    IF (r == g) THEN quit // goal reached
    IF (r == p) THEN quit // goal not reachable
ENDWHILE
```

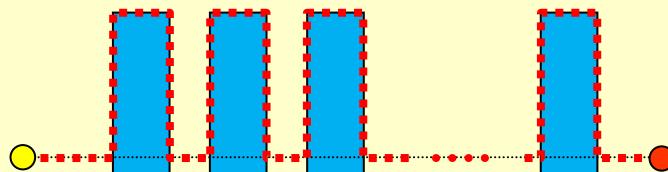


# The Bug2 Algorithm

- This algorithm:
  - also always finds goal location (if it is reachable).
  - performs a “greedy” search for the “best” point to leave the obstacle and head towards the goal.
- The robot may travel a distance of:

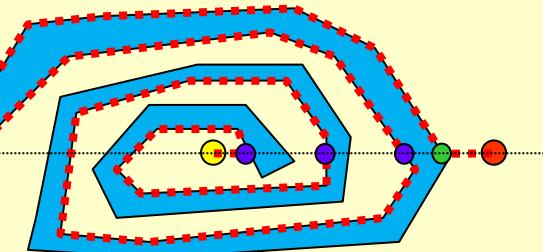
$$|\overline{sg}| + O(\text{perimeter}(\text{obj}_1) + \text{perimeter}(\text{obj}_2) + \dots + \text{perimeter}(\text{obj}_n))$$

In worst case, we choose the “wrong way to go around the obstacle, leading to almost a full perimeter traversal:



Amount of perimeter travel will depend on choice of left/right edge following:

What happens if robot follows on its left side instead of right?



# The Bug2 Algorithm

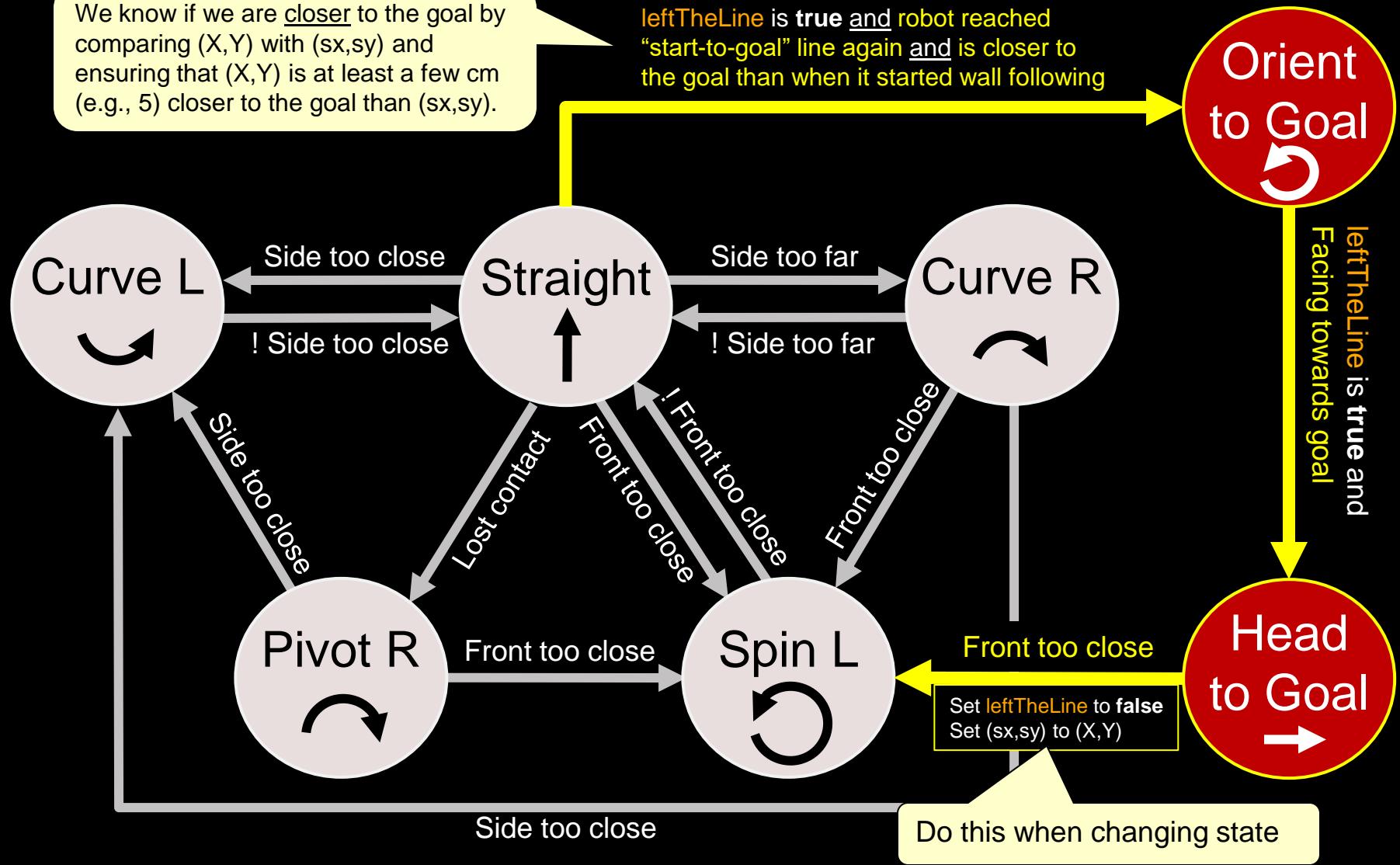
---

- Bug2 algorithm is quicker than Bug1.
- The algorithms do have practical problems:
  - Assumes perfect positioning (not really possible)
  - Assumes error-free sensing (not ever possible)
  - Real robots have limited angular resolution
- Algorithms assume that robot could only detect obstacle upon contact or close proximity.
  - Can improve algorithm when robot is equipped with a 360° range sensor that determines distances to obstacles around it.
  - If you have time, look up the Tangent Bug algorithm.

# Coding With State Machine

We know if we are closer to the goal by comparing (X,Y) with (sx,sy) and ensuring that (X,Y) is at least a few cm (e.g., 5) closer to the goal than (sx,sy).

leftTheLine is **true** and robot reached “start-to-goal” line again and is closer to the goal than when it started wall following



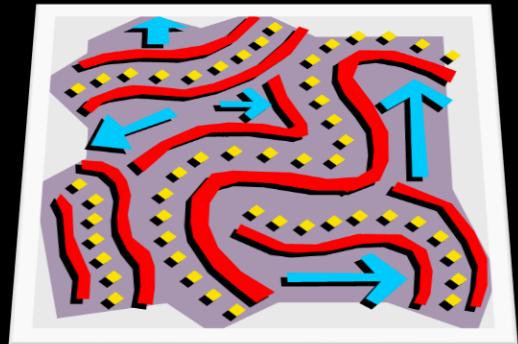
Start the  
Lab ...

# Mapping

# Maps

---

- A robot's environment may change over time.
- A *map* is a stored representation of an environment at some particular time.
- Allows robot to:
  - plan navigation strategies
  - avoid obstacle collisions during travel
  - identify changes in the environment
  - identify accessible/inaccessible areas
  - verify its own position in the environment
- We will only consider 2-D maps in this course



# Maps

---

- Realistically, maps are only **estimates**
  - often imprecise
- When navigating using a map, a robot must also rely on its sensors to avoid collisions since maps may be **inaccurate** or simply **wrong**.
- The goal when making a map is to make it as accurate as possible, although this is not easy.
- Before creating a map, we must decide on how it will be stored (i.e., represented) in memory.



# Map Representation

---

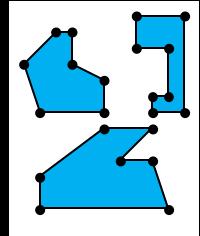
- Maps can be represented as various types:
  - **Topological maps**
    - Keeps relations between obstacles (or free space) within env.
  - **Obstacle maps**
    - Keeps locations of obstacles and inaccessible locations in env.
  - **Free-space maps**
    - Keeps locations that robot is able to safely move to within env.
  - **Path/Road maps**
    - Keeps set of paths that robot can travel along safely in env.
    - Usually used in industrial applications to move robots along known safe paths.

# Map Representation

- Maps are stored in one of two main ways:

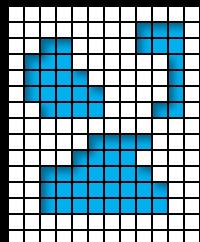
- **Vector**

- stored as collection of line segments and polygons
    - usually represents obstacle boundaries



- **Raster**

- storage in terms of fixed 2D grid of cells
    - each cell stores probability of occupancy (i.e., obstacle)

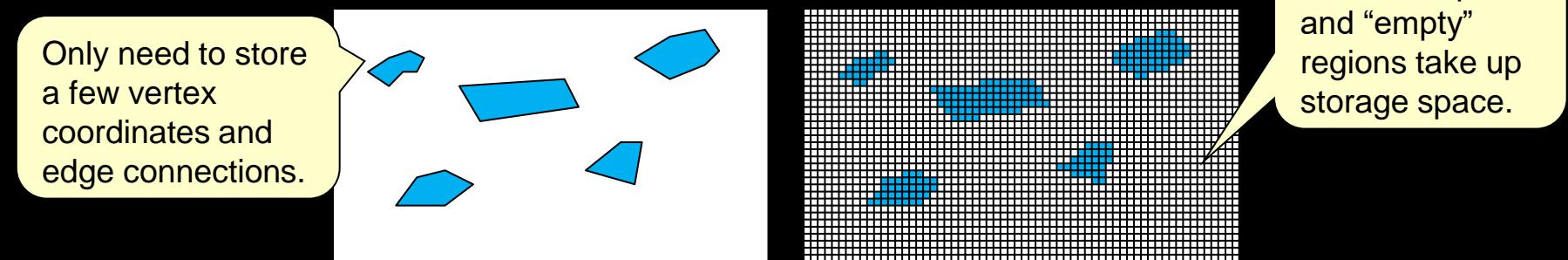


- Main differences lie in:

- storage space requirements
  - algorithm complexity and runtime

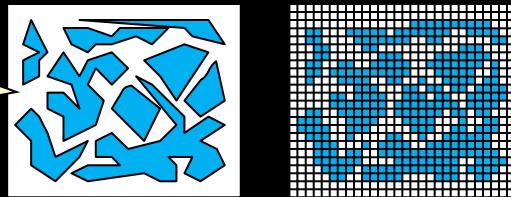
# Map Storage Space

- Large environments with few and simple obstacles take less space to store as vector:



- Smaller, obstacle-dense environments may be better stored as raster/grid:

Storing many vertices and edges may require more space than storing a small course grid.



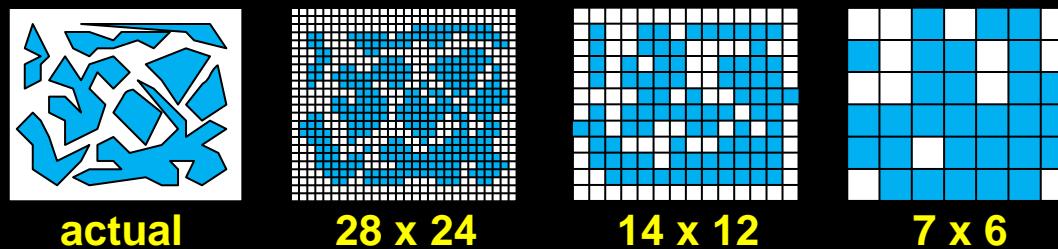
# Map Storage Space

- Vector maps require the following storage space:
  - $m$  obstacles with  $n$  vertices each requires storage of (x,y) vertex coordinates as well as edges (e.g., stored as linked list pointers)
  - $\underline{\text{Storage}} = (m * n) * 2_{\text{integers}} + 2_{\text{indices}} * (m * n) = O(mn)$ 
- Raster maps require storage space that varies according to grid size (i.e., according to desired resolution):
  - a grid of size  $M \times N$  takes  $O(MN)$
- If  $m,n \ll M,N$  then vector maps are more efficient

Optional, edges don't have to be stored explicitly, but same time complexity.

# Map Storage Space

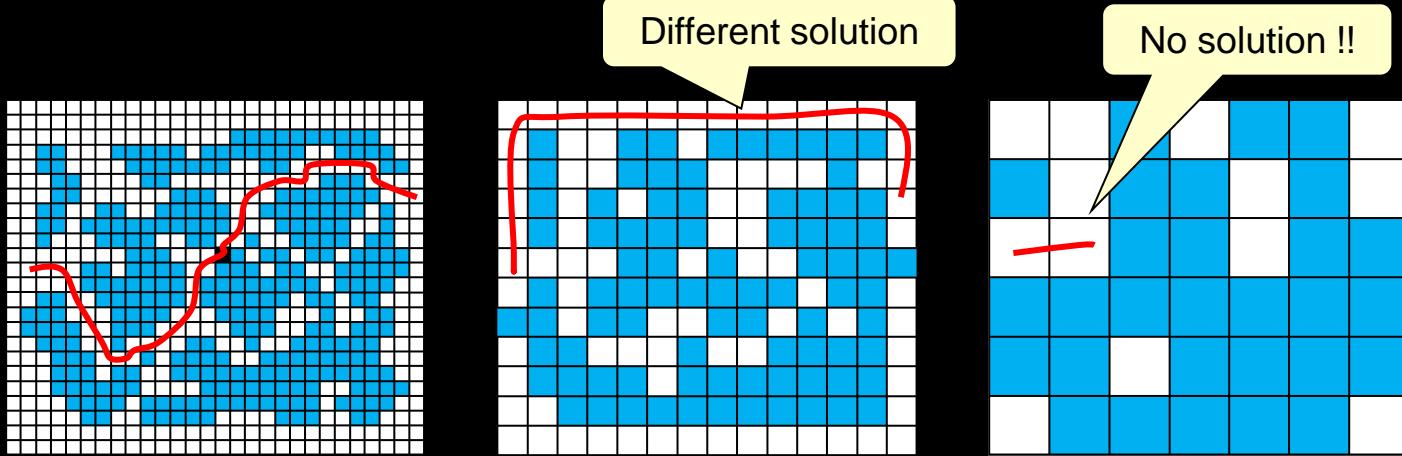
- Of course, much varies according to the resolution of the raster maps (i.e., depends on **M** & **N**).
- Resolution depends on desired accuracy. Notice the difference that it can make on the map:



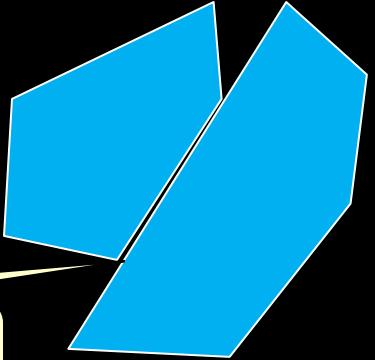
- As resolution decreases:
  - storage requirements are reduced
  - representation of “true” environment is compromised

# Map Accuracy

- This decrease in accuracy can affect solutions to problems:



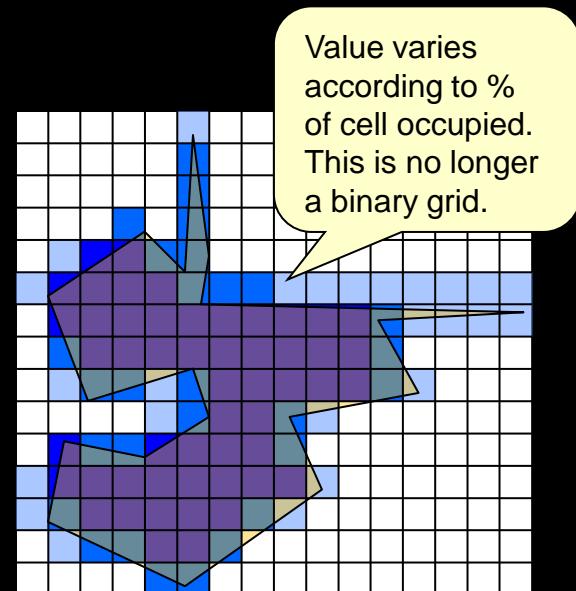
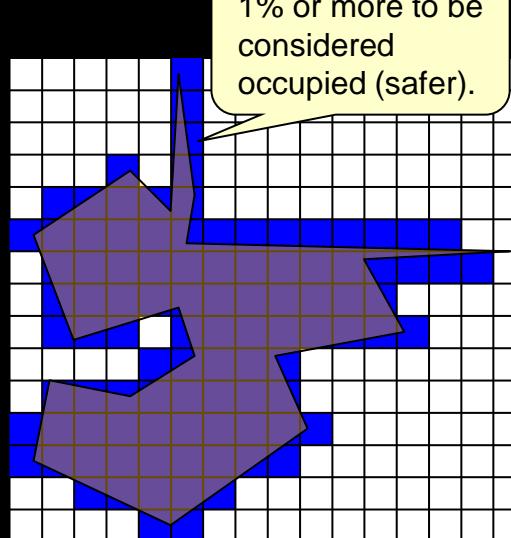
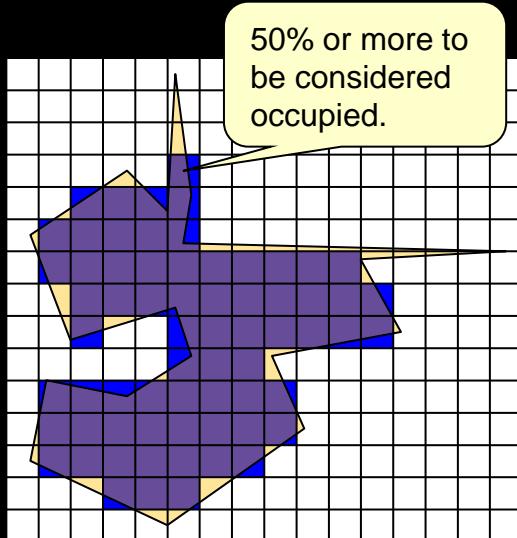
- With vector maps, solution does not depend on storage resolution, but instead on numerical precision:



Close polygons may compute as intersecting, depending on numerical precision.

# Map Accuracy

- Robot safety may be another issue with raster maps.
  - Assumptions about obstacle locations may lead to collisions
- Occupancy of grid cells can depend on some threshold indicating “*certainty*” that obstacle is at this location:



# Map Accuracy

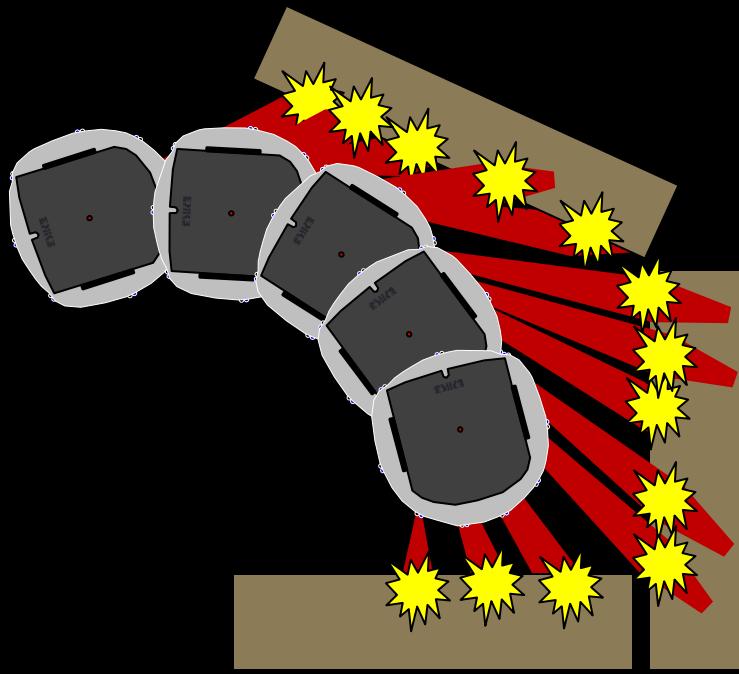
---

- In practice, many robots use such raster maps because they allow for “fuzziness” in terms of obstacle position.
  - They are commonly called ***occupancy grids*** (or ***certainty grids*** or ***evidence grids*** ).
- Still useful since most maps are constructed based on sensor data (which is already uncertain).
- The cell’s occupancy value indicates the probability that an obstacle is at that location.

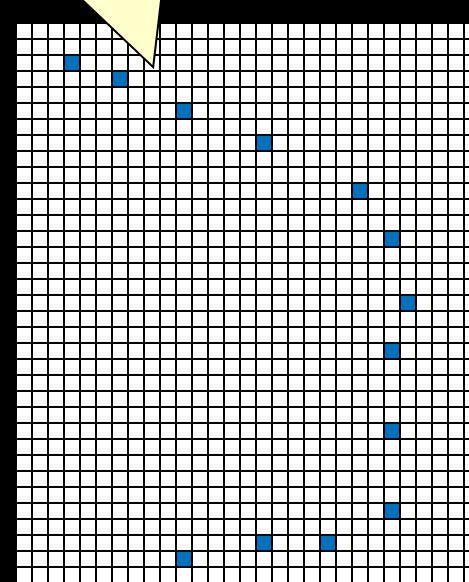


# Multiple Readings

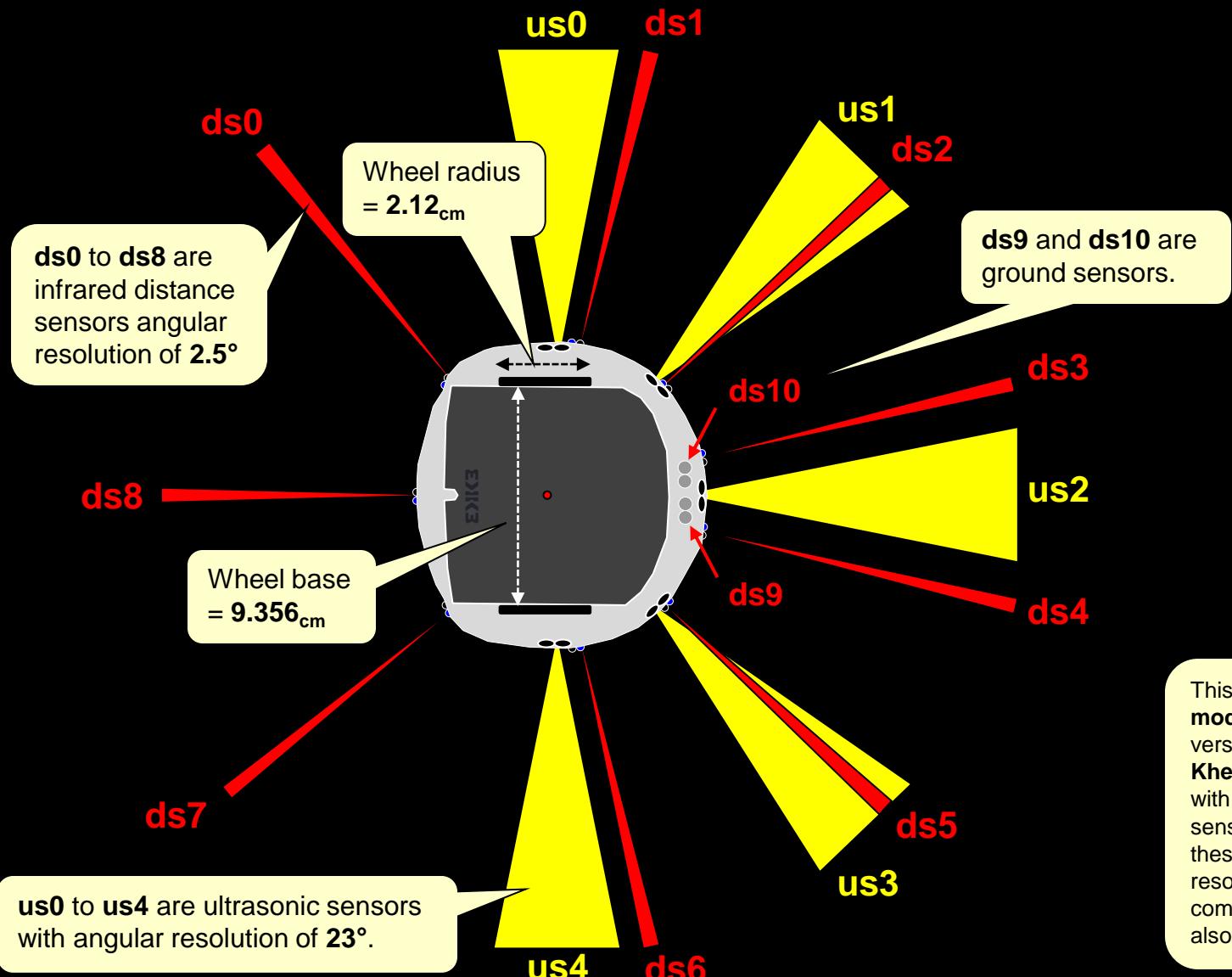
- A map is formed by merging together the results from multiple readings from various locations in the environment.



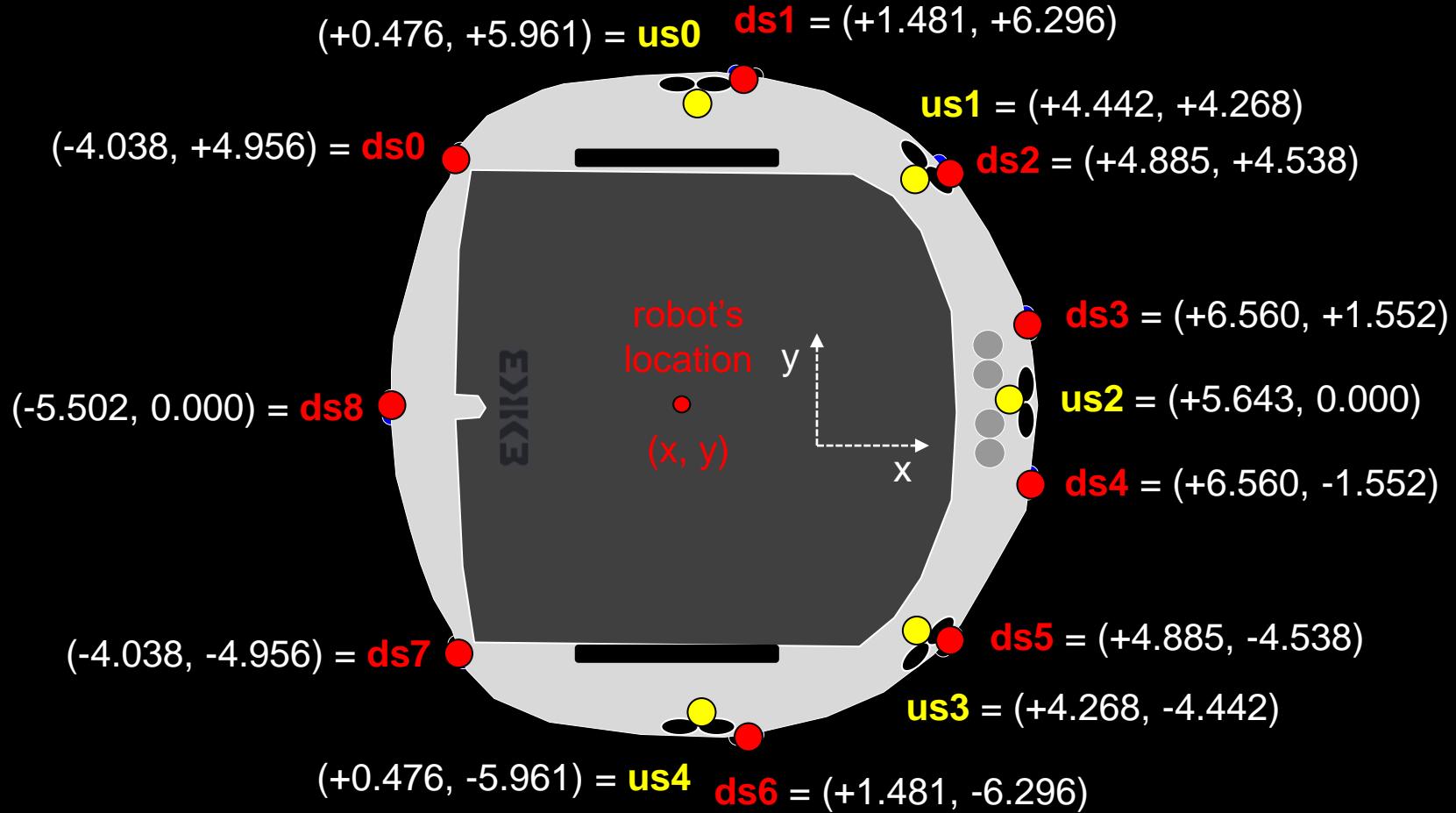
Occupancy Grid. Can easily be represented using a 2D array.



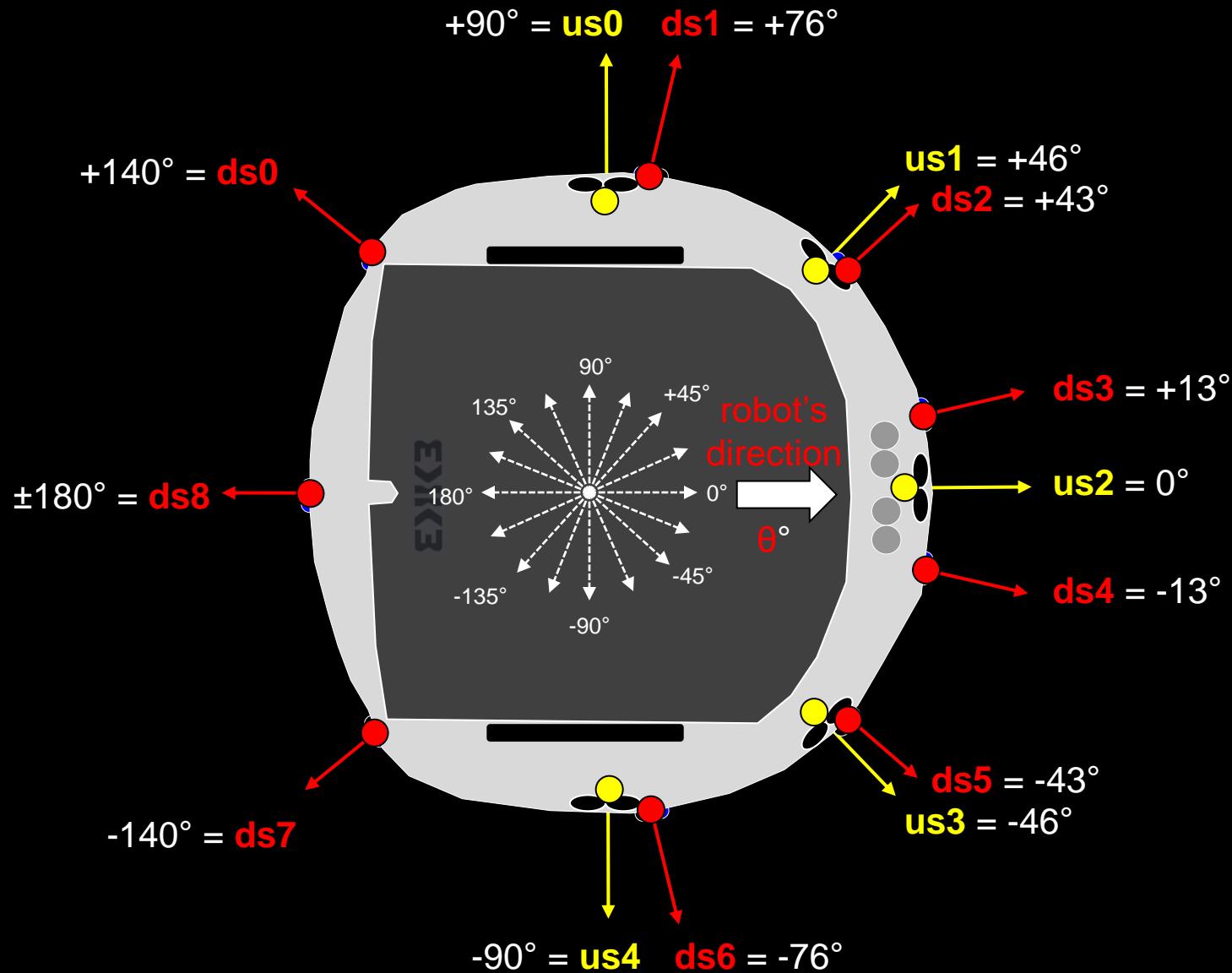
# The Khepera III Robot



# Sensors – Location Offsets



# Sensors – Angle Offsets

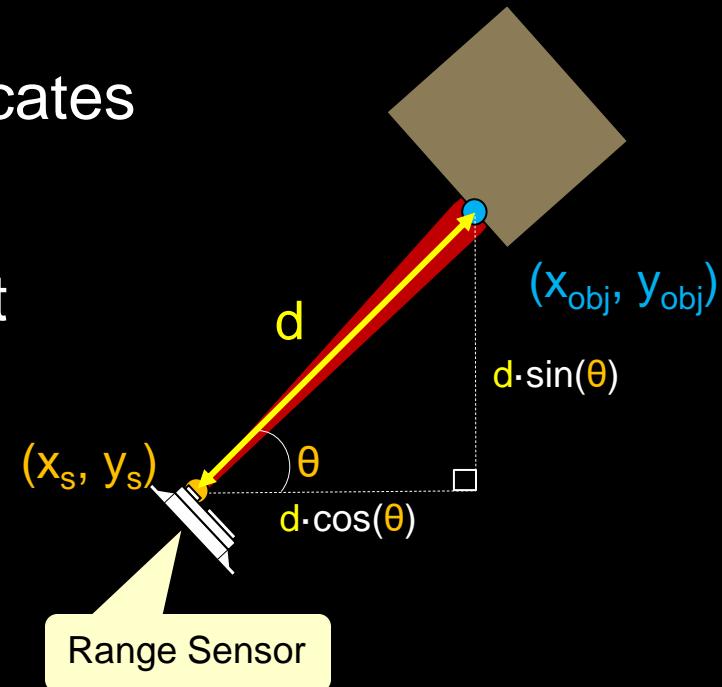


# Mapping Raw Sensor Data

- Mapping involves taking lots of sensor readings.
- Assume that a range sensor is at location  $(x_s, y_s)$  in the environment and is tilted at angle  $\theta$  with respect to the horizontal in the coordinate system.
- Assume that the sensor reading indicates an object at  $d_{cm}$  from the sensor.
- We now know that there is an object at location  $(x_{obj}, y_{obj})$  where:

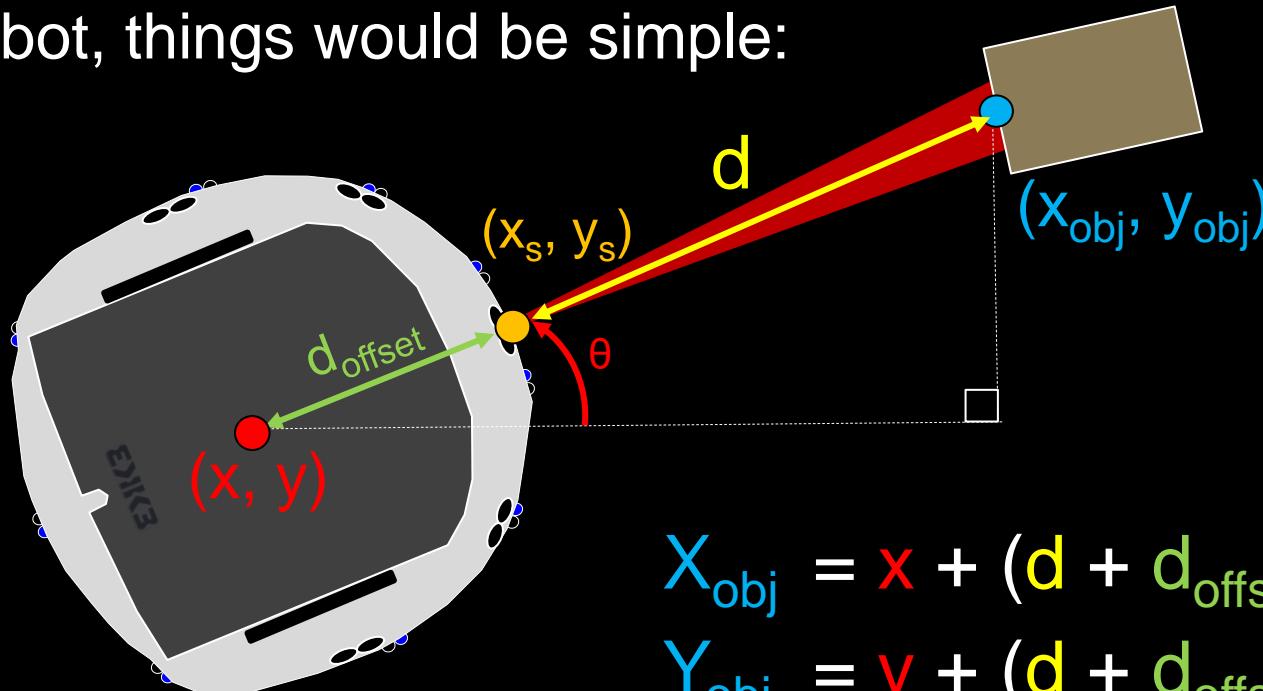
$$x_{obj} = x_s + d \cdot \cos(\theta)$$

$$y_{obj} = y_s + d \cdot \sin(\theta)$$



# Computing Object Location

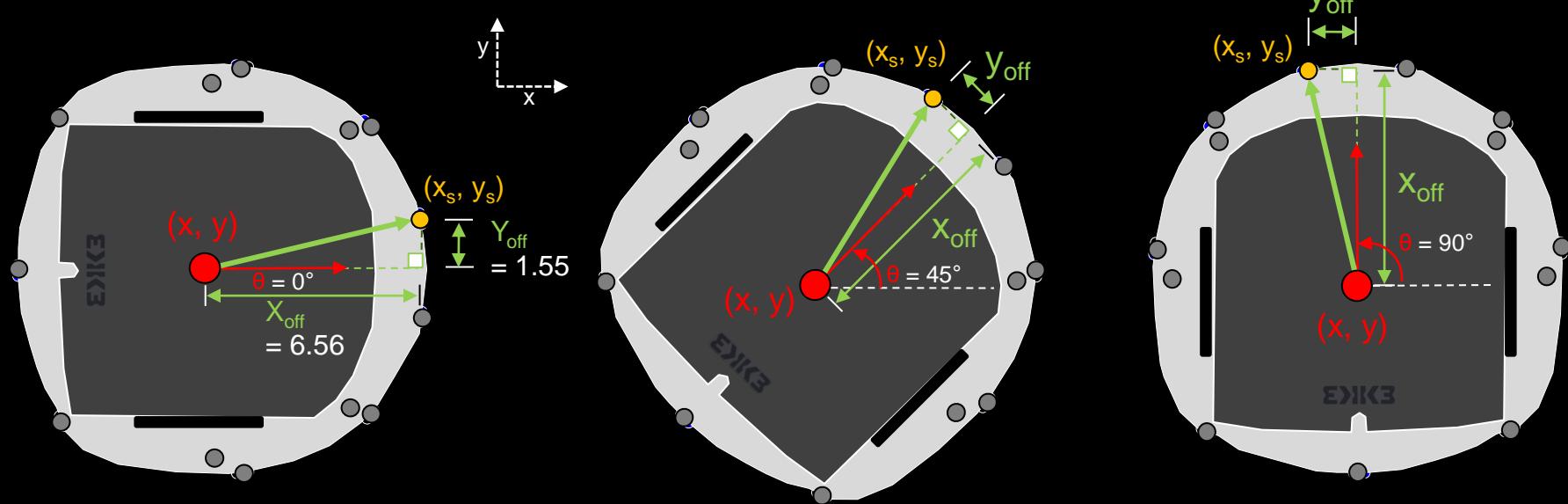
- The location  $(x_{\text{obj}}, y_{\text{obj}})$  of an object needs to be computed with respect the robot's pose  $(x, y, \theta)$  because that is the only reference point that we have within the environment.
- If the sensor was pointing straight ahead at the front of the robot, things would be simple:



$$X_{\text{obj}} = x + (d + d_{\text{offset}}) \cdot \cos(\theta)$$
$$Y_{\text{obj}} = y + (d + d_{\text{offset}}) \cdot \sin(\theta)$$

# Rotating Sensor Offsets

- To get reading with respect to the robot's position (i.e., center of robot), we need to *transform* (i.e., *rotate*) the sensor offsets according to direction the robot is currently facing.



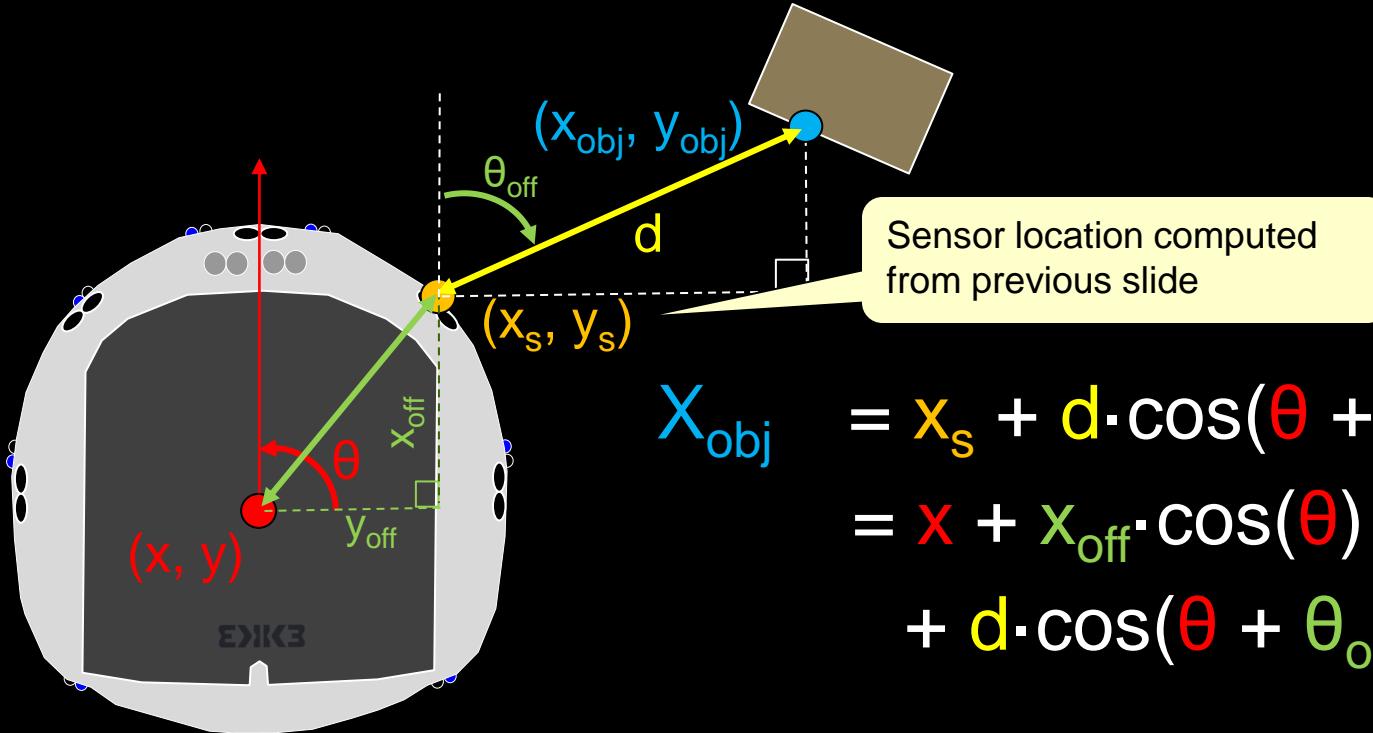
$$x_s = x + x_{\text{off}} \cdot \cos(\theta) - y_{\text{off}} \cdot \sin(\theta)$$

$$y_s = y + y_{\text{off}} \cdot \cos(\theta) + x_{\text{off}} \cdot \sin(\theta)$$

Sensor location computed as rotation around the robot's location  $(x, y)$ .

# Accounting for Sensor Offsets

- For each sensor, you must take into account its offsets:



$$\begin{aligned} X_{obj} &= x_s + d \cdot \cos(\theta + \theta_{off}) \\ &= x + x_{off} \cdot \cos(\theta) - y_{off} \cdot \sin(\theta) \\ &\quad + d \cdot \cos(\theta + \theta_{off}) \end{aligned}$$

$$\begin{aligned} Y_{obj} &= y_s + d \cdot \sin(\theta + \theta_{off}) \\ &= y + y_{off} \cdot \cos(\theta) + x_{off} \cdot \sin(\theta) \\ &\quad + d \cdot \sin(\theta + \theta_{off}) \end{aligned}$$

# Mapper App

- An embedded **MapperApp** class (included with your controller) allows you to display an occupancy grid map.
- Map points are filled in for each  $(x_{obj}, y_{obj})$  computed.

Black dots indicate all the  $(x_{obj}, y_{obj})$  computed over time.



Create the display for mapping based on the loaded world size (in cm). This code is given to you.

```
MapperApp mapper;  
mapper = new MapperApp(worldX, worldY, display);  
  
mapper.addObjectPoint(Xobj, Yobj);
```

You do this each time you want to add a new object point to the map. Pass in the coordinate and it will appear on the map.

There will be lots  
of invalid/noisy  
readings

# Reading the Sensor

- Here is how to set up the distance sensors on the Kheperra III robot:

```
import com.cyberbotics.webots.controller.DistanceSensor;  
  
// Sensors are objects  
static DistanceSensor    rangeSensors[];  
  
// Set up the range sensors to be used  
rangeSensors = new DistanceSensor[9];  
for (int i=0; i<9; i++) {  
    rangeSensors[i] = robot.getDistanceSensor("ds"+i);  
    rangeSensors[i].enable(timeStep);  
}
```

- Here is how to read distance **d** from the sensor:

```
// Read a distance reading d for sensor i in the array of sensors  
double d = rangeSensors[i].getValue() * 100;
```

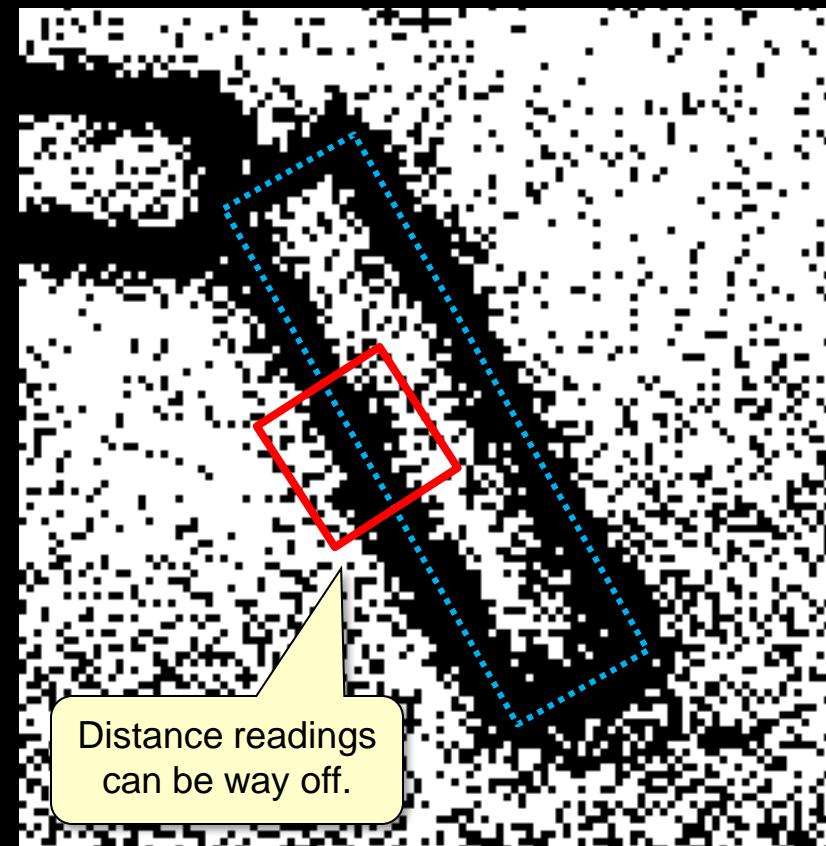
Do this to read a single distance reading from one sensor in the array. Multiply by 100 to convert to centimeters.

Start the  
Lab ...

# Sensor Models

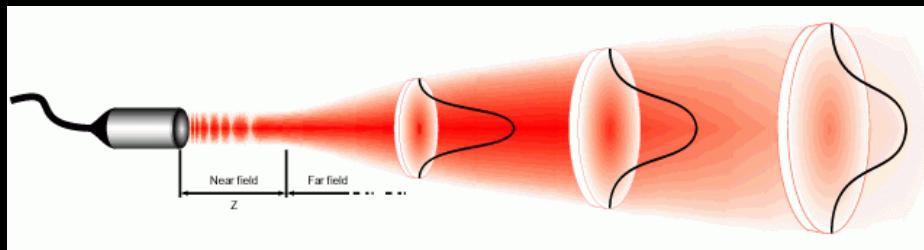
# Problems With Sensor Data

- Raw maps we made assumed that sensor data was accurate
  - Each reading indicated a part of the obstacle
- Result is that objects appear on map as a fuzzy set of points with a lot of noise
- Problem is due to inaccurate sensors and inability to be precise due to:
  1. Beam-width Error
  2. Distance Error
  3. Reflection Error

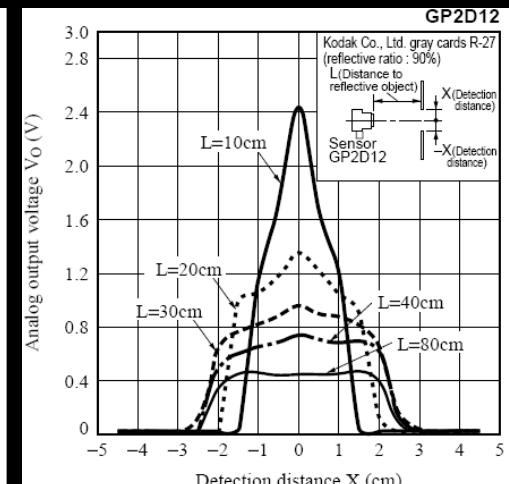
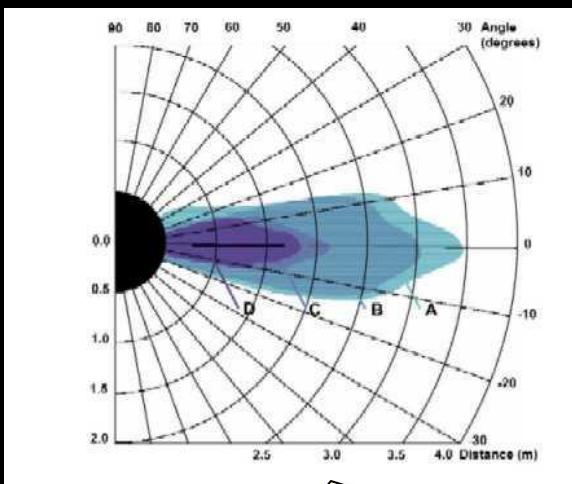
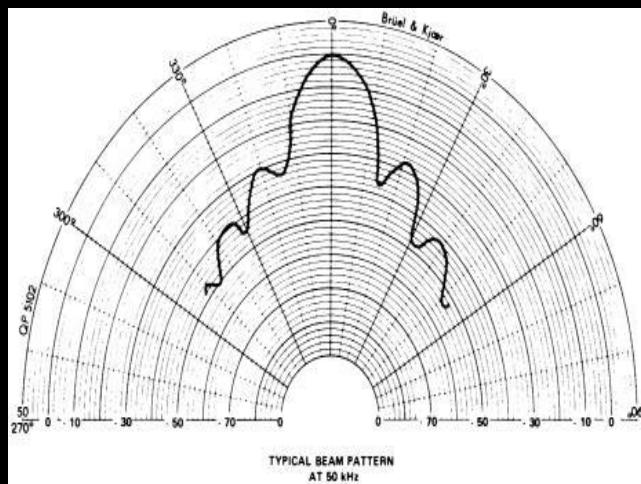


# 1. Beam-width Error

- Shape of beam is not a simple wedge:



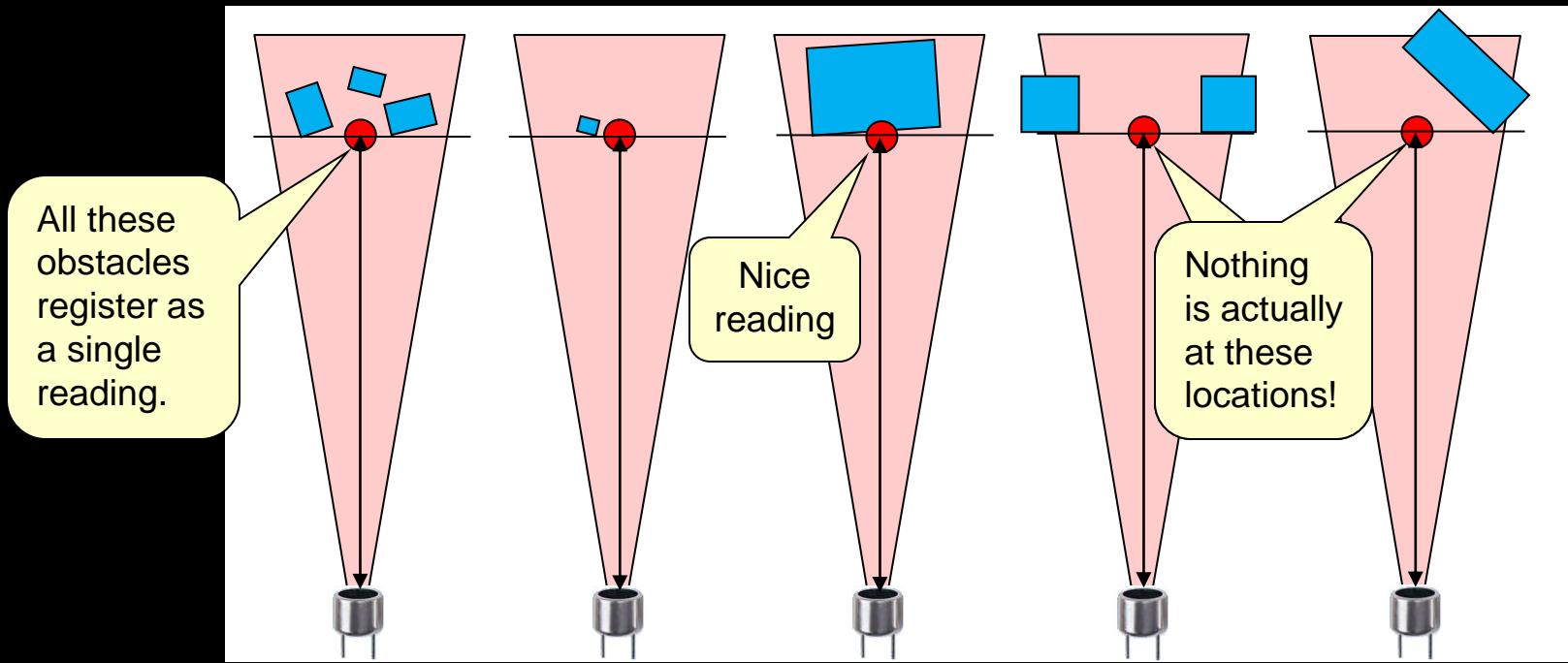
- wider objects near center of beam result in better accuracy



These are sensor specifications showing the sensor beam's shape.

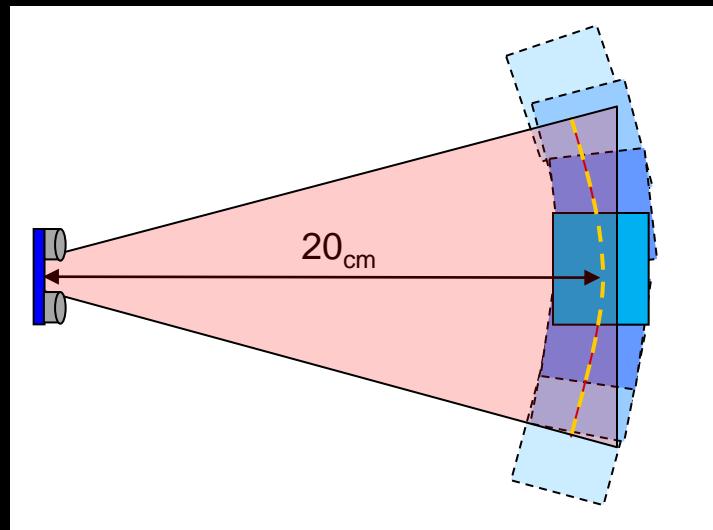
# 1. Beam-width Error

- So, when we receive a range reading, we are actually unsure as to whether or not the reading accurately represents what is directly ahead:



# 1. Beam-width Error

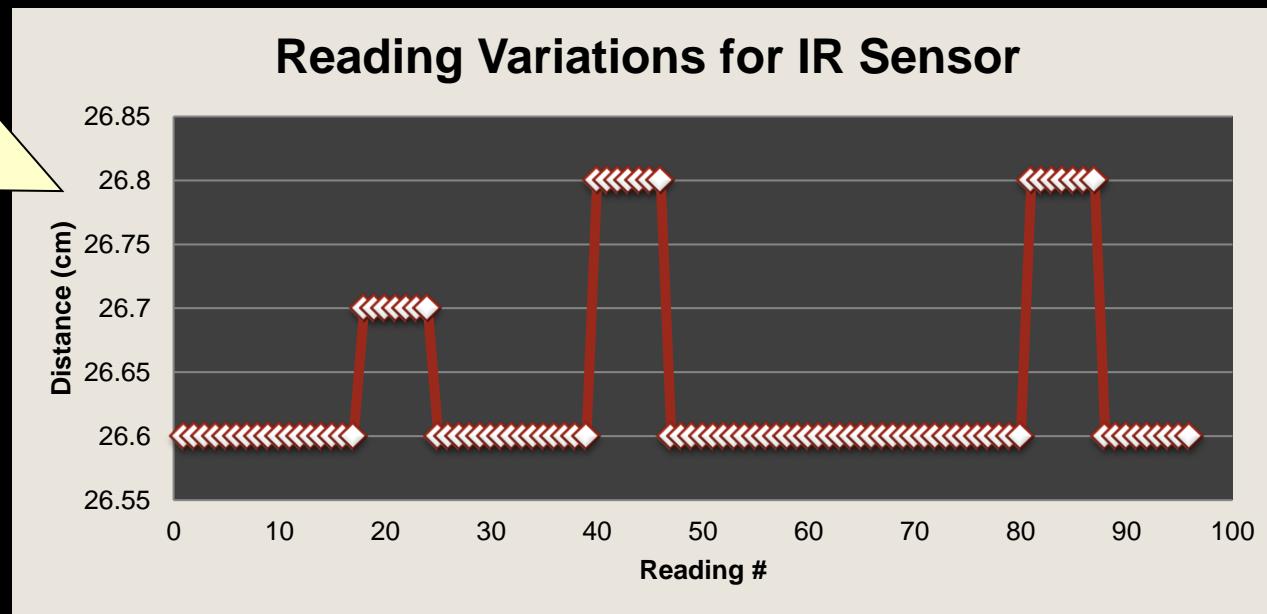
- When a single reading is obtained, object may actually be anywhere along the width of the beam, not necessarily at the center of the beam.
- When an object is detected at, say  $20_{\text{cm}}$ , it can actually be anywhere within the beam arc defined by the  $20_{\text{cm}}$  radius:



## 2. Distance Error

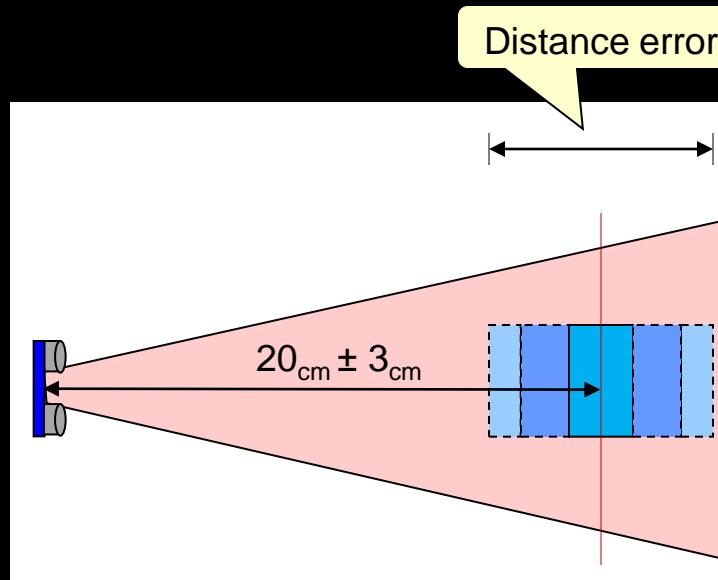
- In addition to the location of the obstacle on the beam, sensors themselves give inaccurate distances.

Readings vary even though obstacle is stationary



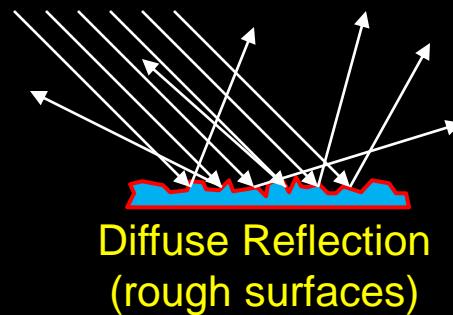
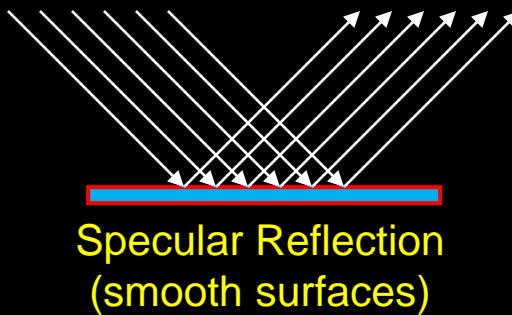
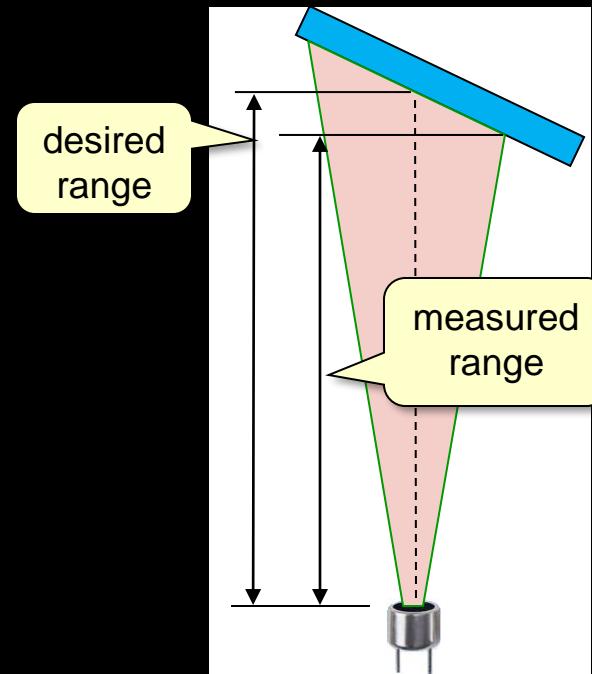
## 2. Distance Error

- When a single reading is obtained, object may actually be closer or further than what is expected.
- When object is detected at, say  $20_{\text{cm}}$ , it can actually be closer or further within some tolerance:



# 3. Reflection Errors

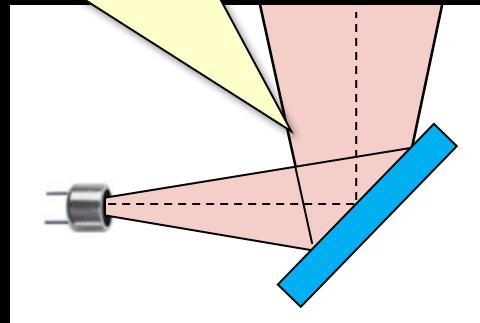
- Sensitivity to obstacle angle can result in improper range readings.
- When beam's angle of incidence falls below a certain critical angle ***specular reflection*** errors occur.



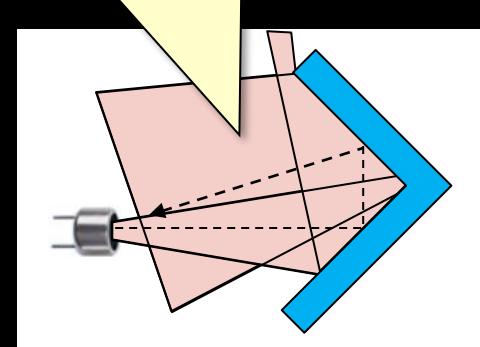
# 3. Reflection Errors

- Specular reflection can cause reflected ultrasound to:
  - never return to the sensor
  - return to the sensor too late
- In either case, the result is that the distance measurement is too large and inaccurate

Echo never returns, resulting in maximum distance reading.

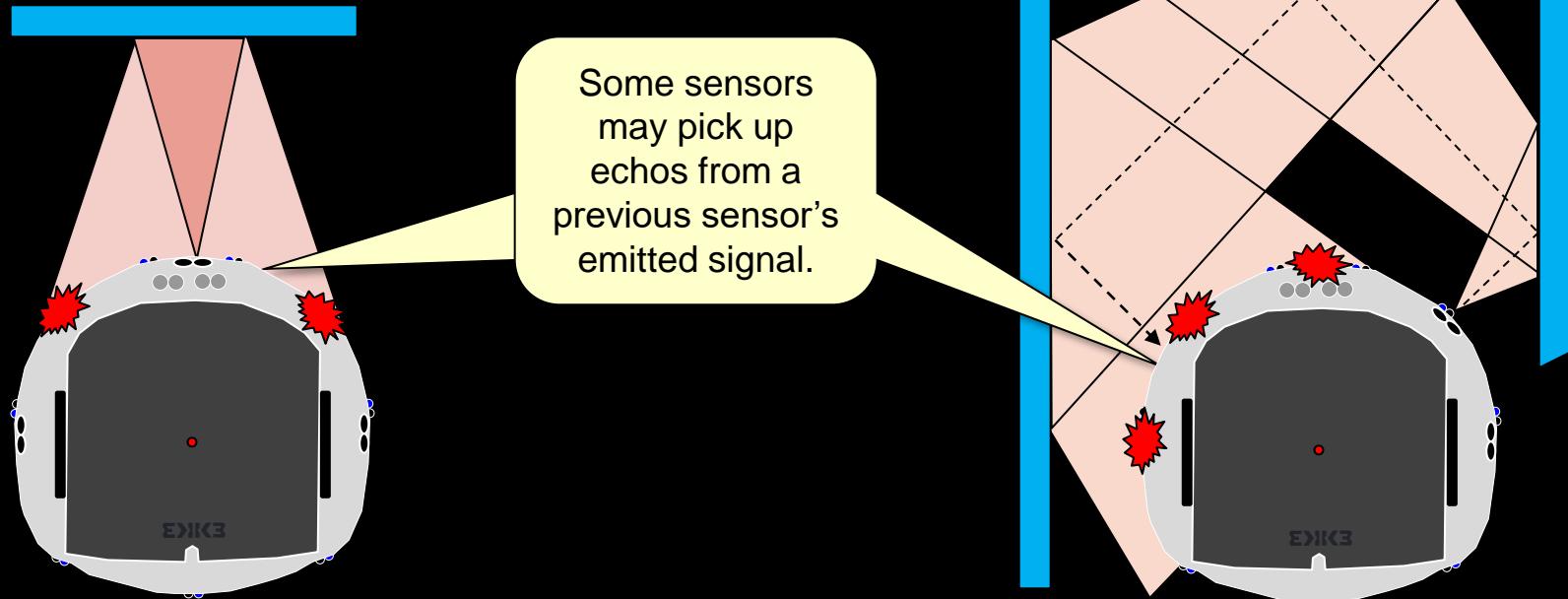


Distance returned represents total round-trip delay.



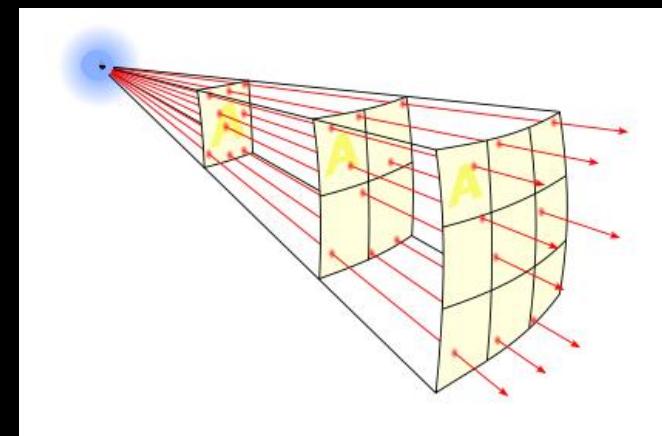
# 3. Reflection Errors

- Using multiple fixed position sensors can lead to another problem called **crosstalk**:
  - A form of interference in which echoes emitted from one sensor are detected by others



# Sensor Models

- Before using a sensor for mapping, a **sensor model** should be developed:
  - specifies how the sensor readings are to be interpreted
  - depends on physical parameters of sensor (e.g., beam width, accuracy, precision etc...)
  - must be able to deal reasonably with noisy data
- For range sensors, they all have similar common characteristics that must be dealt with:
  - distance errors (distance accuracy)
  - angular resolution (beam width)
  - noise (invalid data)



# Sensor Models

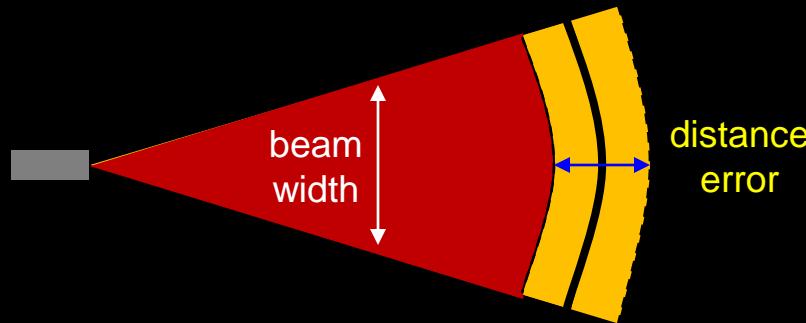
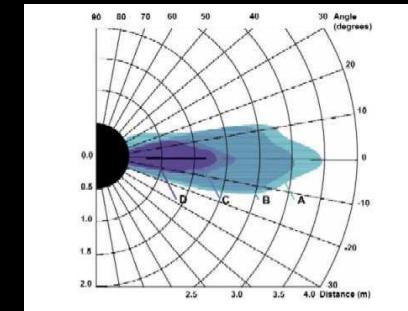
---

- Various ways to come up with a sensor model:
  - Empirical: Through testing
  - Subjective: Through Experience
  - Analytical: Through analysis of physical properties
- Once sensor model is determined, it is applied to each sensor reading so as to determine how it affects the map being built.
- We will consider our sensor models in terms of how they are used in generating occupancy grid maps.



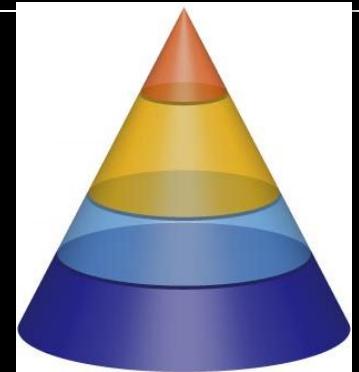
# Sensor Models

- Our sensor model will consider distance accuracy and beam width.
- Sensor beam width is not easy to model
  - has different width at different distances
  - different obstacles have different reflective effects
- Most models keep things simple by assuming that the beam is a cone-shaped wedge:



# Sensor Models

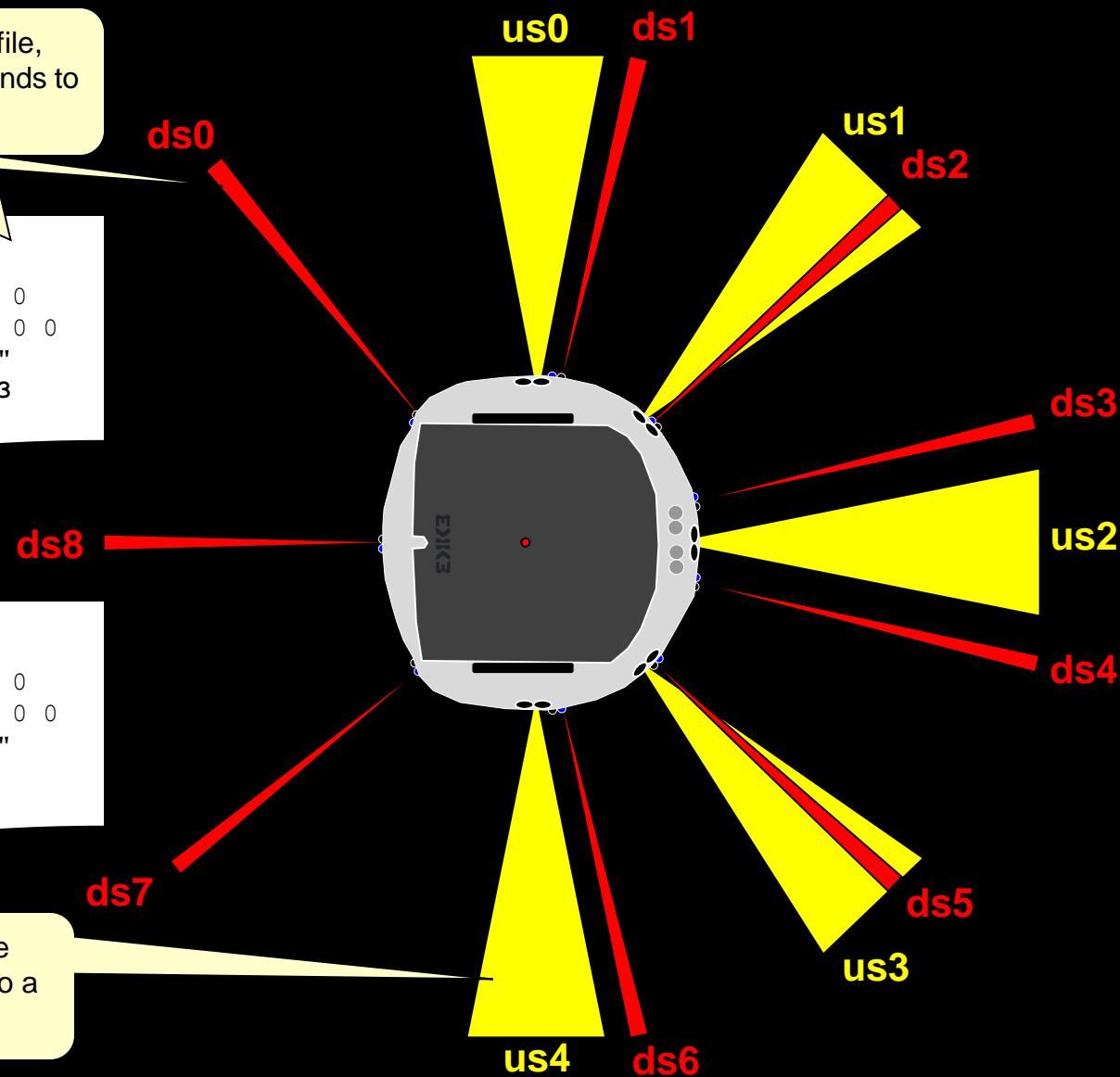
- We will assume that our sensors have beams with this simple cone shape:
  - actually an **approximation** of the true shape
  - simplifies calculations
  - will vary beam width and distance error with each sensor
- How do we pick beam width and distance error ?
  - beam width and distance error may vary between individual sensors of the same type
  - beam width and distance error usually obtained through experimentation
  - take average of many readings at certain distances



# Khepera III Sensor Beam Width

In the **Khepera3\_DistanceSensor.proto** file, the **aperture** is set to **0.03** which corresponds to a **1.72°** beam width.

```
PROTO Khepera3_DistanceSensor [
    field SFVec3f    translation  0 0 0
    field SFRotation rotation    0 1 0 0
    field SFString   name        "ds"
field SFFloat    aperture    0.03
    field SFInt32   numberOfRays 3
]
```

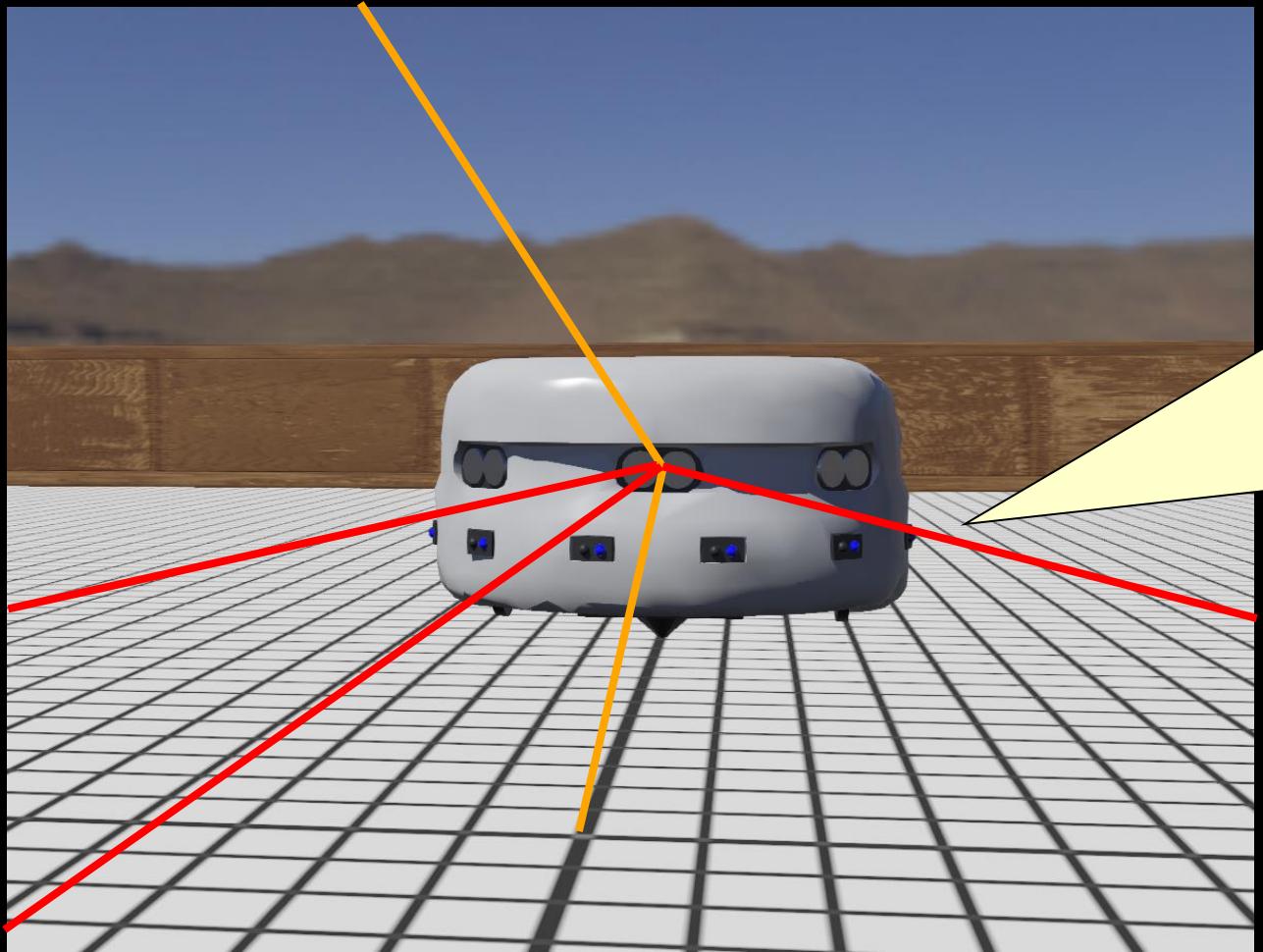


```
PROTO Khepera3_USSensor [
    field SFVec3f    translation  0 0 0
    field SFRotation rotation    0 1 0 0
    field SFString   name        "ds"
field SFFloat    aperture    0.4
    field SFInt32   numberOfRays 5
]
```

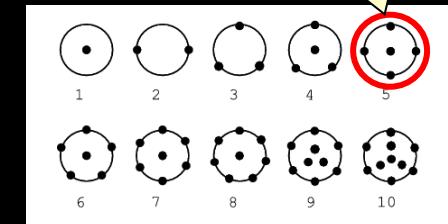
In the **Khepera3\_USSensor.proto** file, the **aperture** is set to **0.4** which corresponds to a **22.9°** beam width.

# Ultrasonic Sensor Simulation

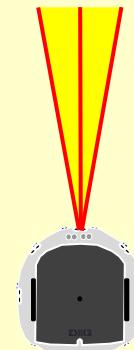
- Webots simulates the ultrasonic sensor by using up to 10 individual rays:



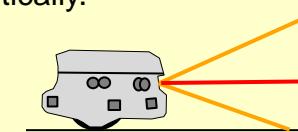
We will use this one.



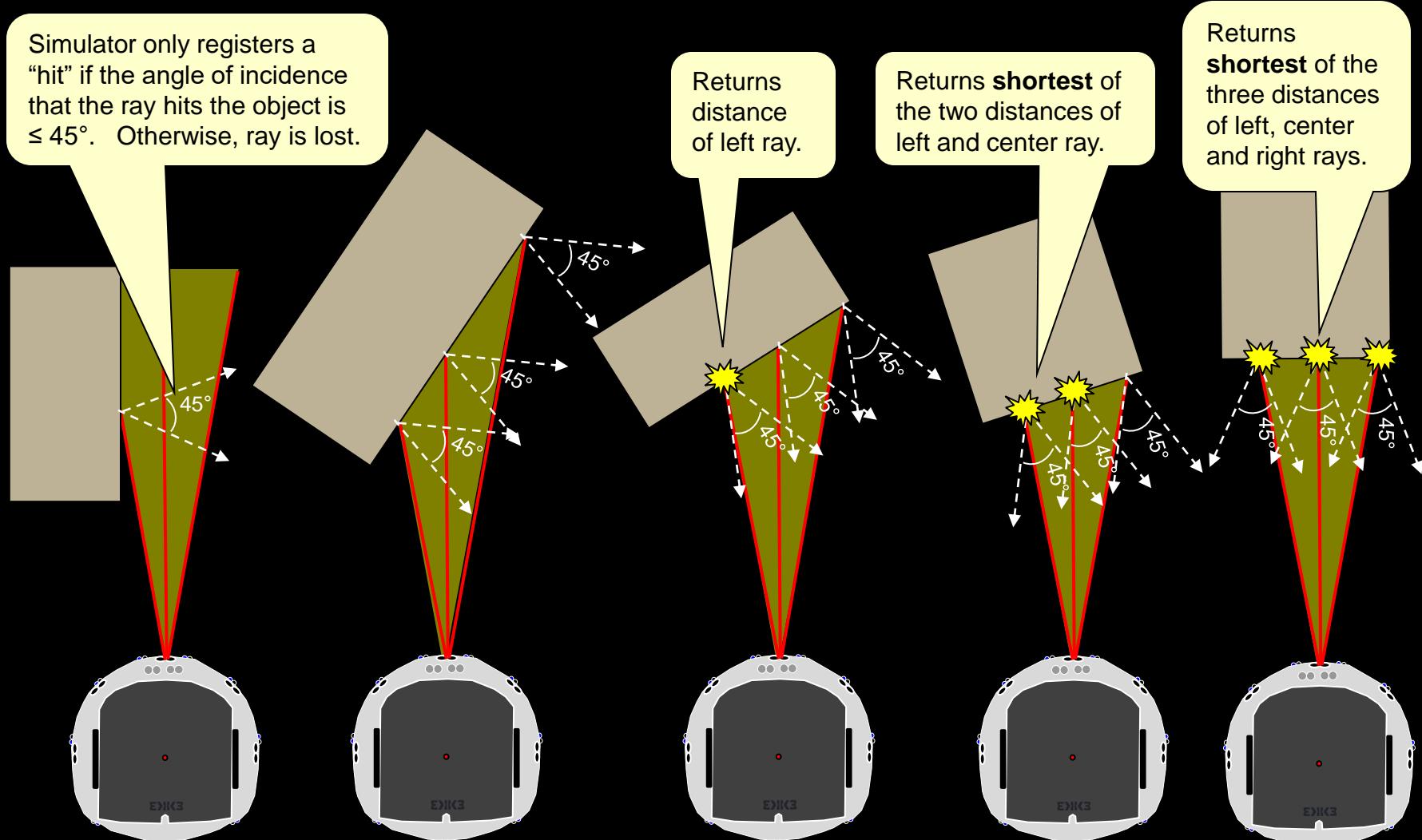
The three red beams are all horizontal at the height of the sensor.



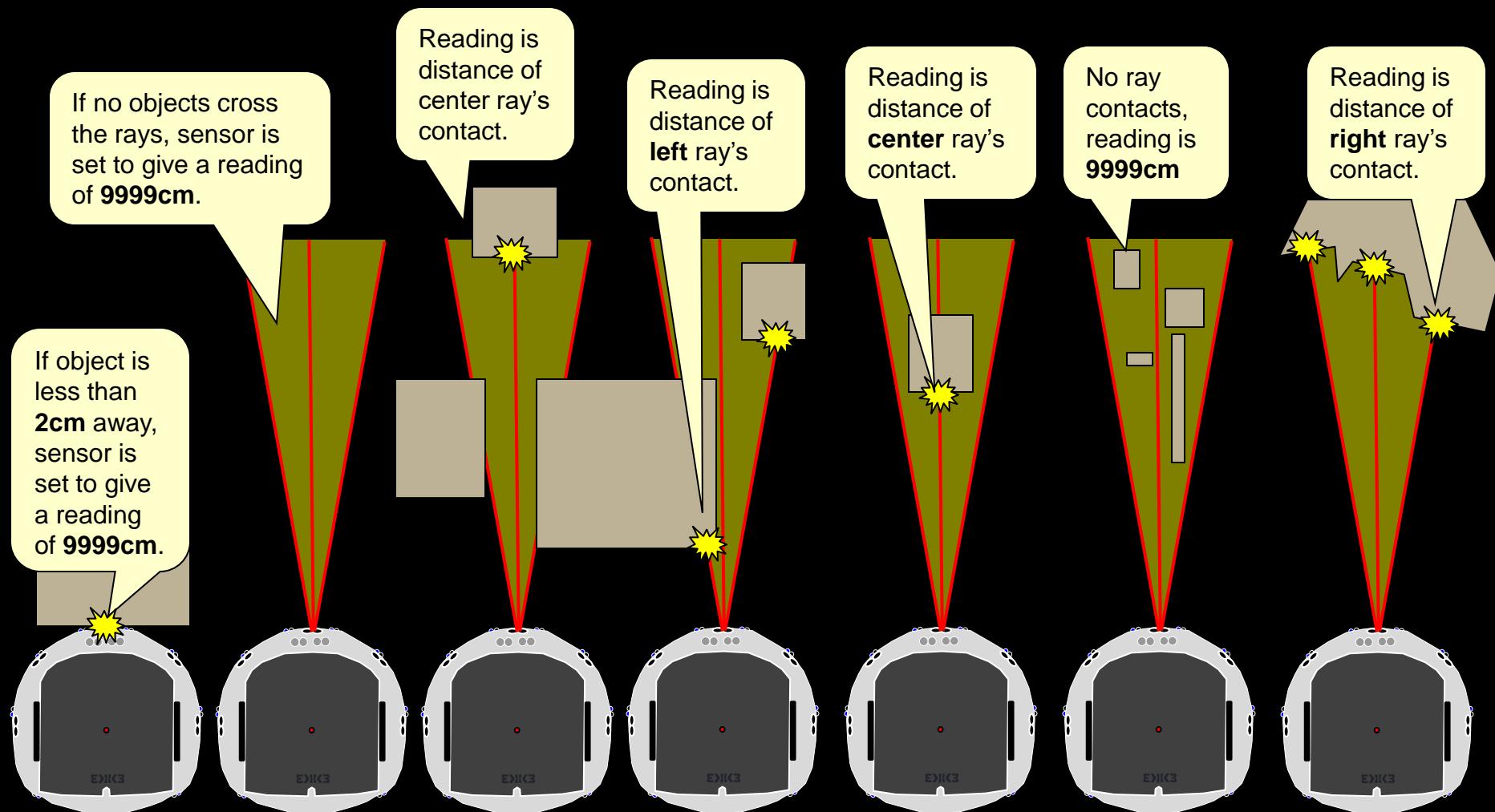
The two orange beams are centered horizontally but go up and down vertically.



# Webots Sonar Sensor Simulation

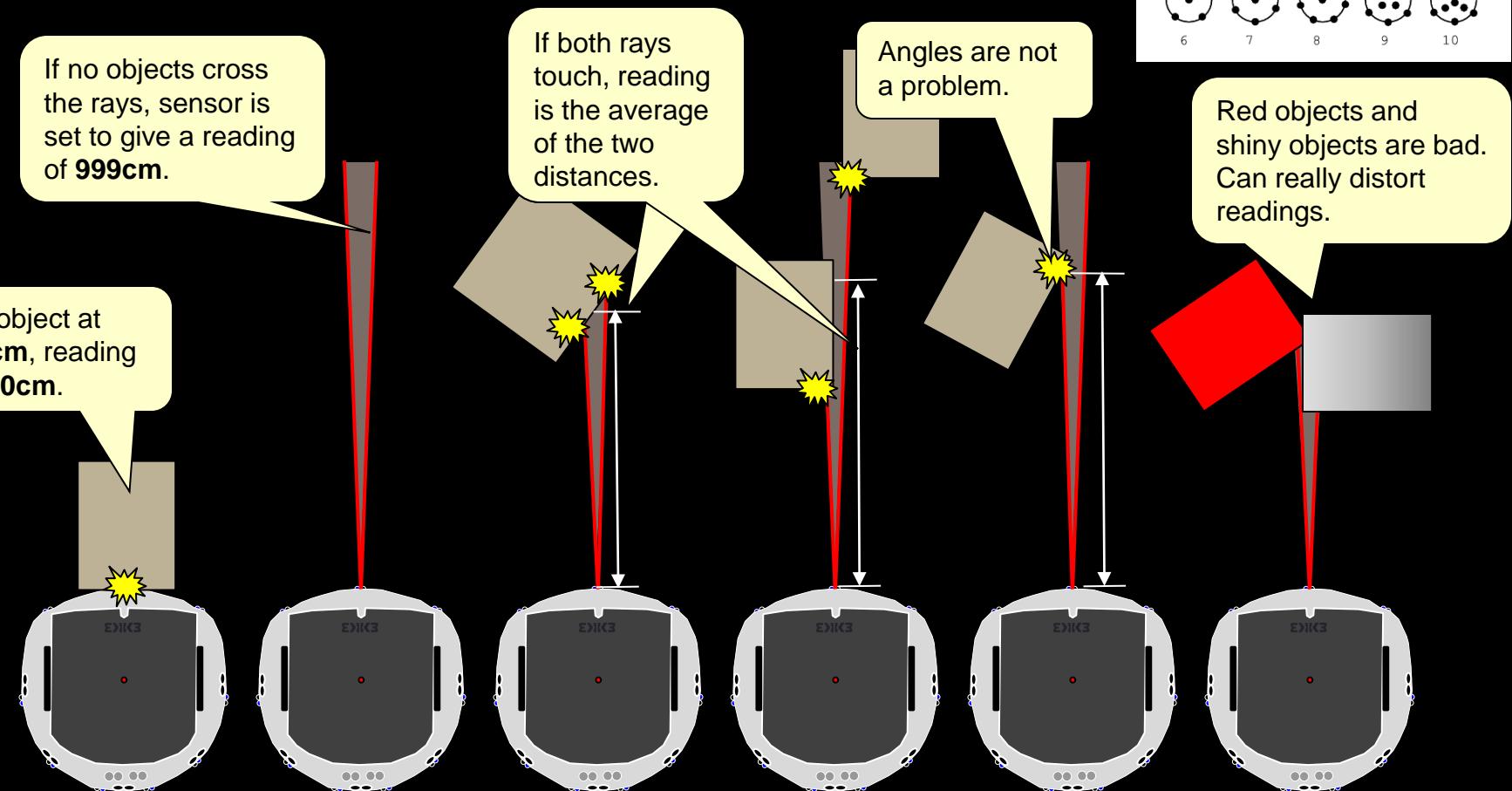


# Webots Sonar Sensor Simulation



# Webots IR Sensor Simulation

- Our IR sensors are set to use just 3 rays:



# Webots Sensor Distance Error

- Our sensors are set to give an accurate reading  $\pm$  some error each time.
  - These are defined in the look-up tables of the **Khepera3\_DistanceSensor.proto** and **Khepera3\_USSensor.proto** files.

**IR Sensors** have a distance error of  $\pm 3.0\%$

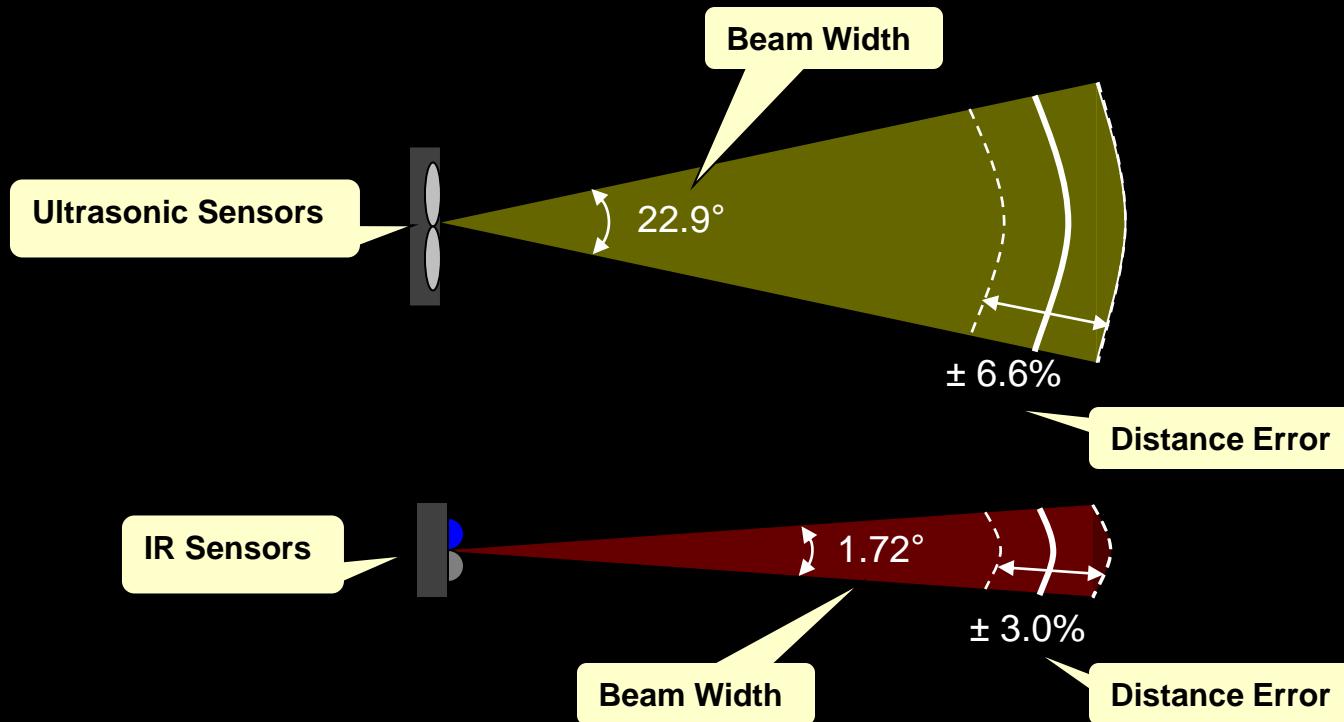
Range	Error
0cm – 5cm	$\pm 0.0\text{cm}$
5cm	$\pm 0.01\text{cm}$
10cm	$\pm 0.04\text{cm}$
30cm	$\pm 0.3\text{cm}$
60cm	$\pm 1.25\text{cm}$
100cm	$\pm 3\text{cm}$

**Ultrasonic Sensors** have a distance error of  $\pm 6.6\%$

Range	Error
0cm – 5cm	$\pm 0.0\text{cm}$
5cm	$\pm 0.03\text{cm}$
10cm	$\pm 0.1\text{cm}$
30cm	$\pm 0.7\text{cm}$
60cm	$\pm 2.5\text{cm}$
100cm	$\pm 6.6\text{cm}$

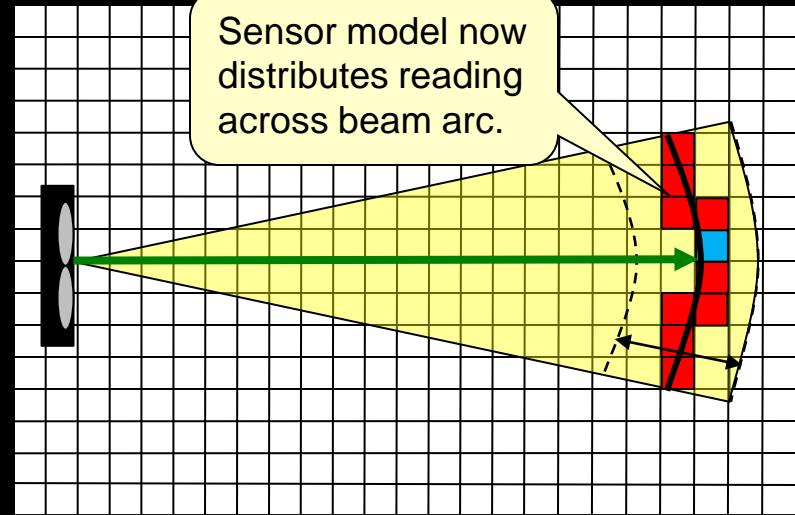
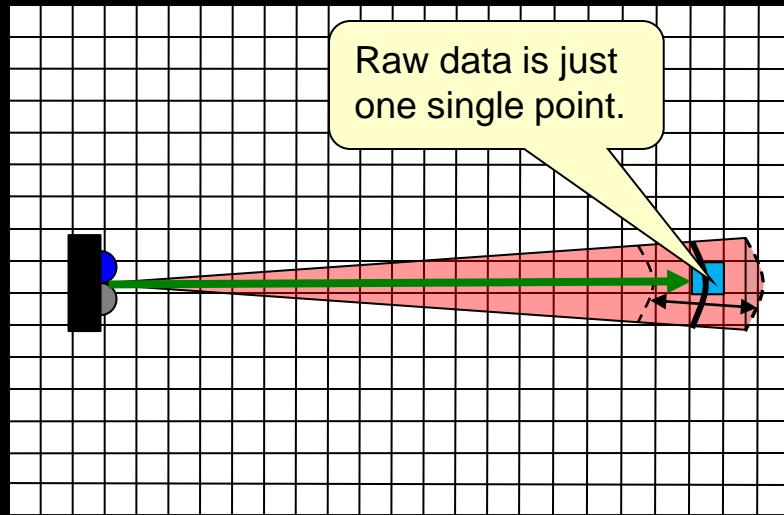
# Our Sensor Models

- Here are the two sensor models that we will use:



# Applying A Sensor Model

- When we detect an obstacle, we will project the sensor model onto the grid by spreading the reading across the beam arc, since the obstacle is not necessarily at the center.



# Sensor Model Implementation

- How do we compute which cells are affected by our sensor model ?
  - Need to determine the cells covered by each arc.
- First, compute arc endpoints:

$(x_s, y_s)$  is the sensor's location, not robot's

$\varphi$  is the sensor's direction, not robot's

$$x_a = x_s + d * \cos(\varphi + \sigma/2)$$

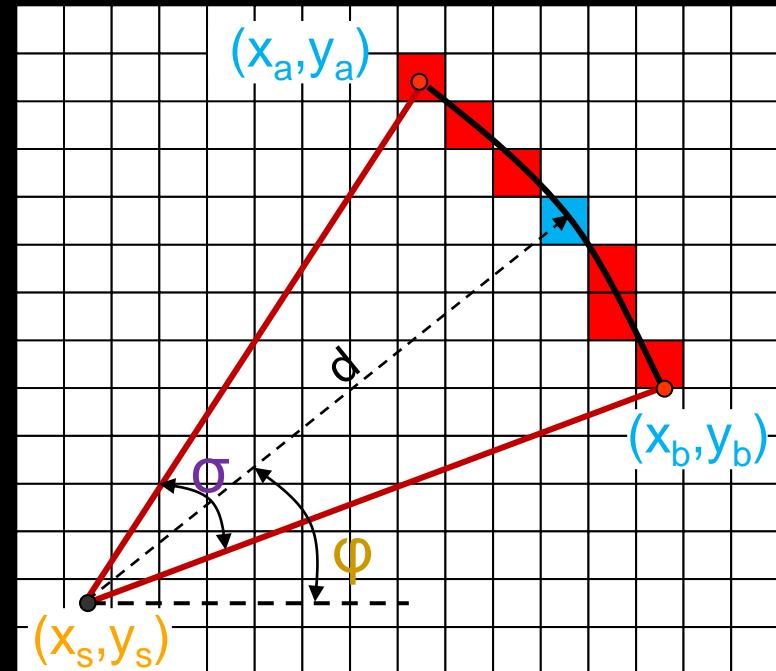
$$y_a = y_s + d * \sin(\varphi + \sigma/2)$$

$$x_b = x_s + d * \cos(\varphi - \sigma/2)$$

$$y_b = y_s + d * \sin(\varphi - \sigma/2)$$

$d$  is the sensor's range reading

$\sigma$  is the sensor's beam width



# Sensor Model Implementation

- Then compute the angular interval so that we cover each cell along the arc once (roughly):

This will  
be a float.

$$\omega = \sigma / \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

- Now go to each grid cell along the arc and increment it accordingly:

If you find that some cells are being skipped, then you can set  $\omega = \omega / 2$  to double-up on the number of cells to fill in.

This will  
be a double

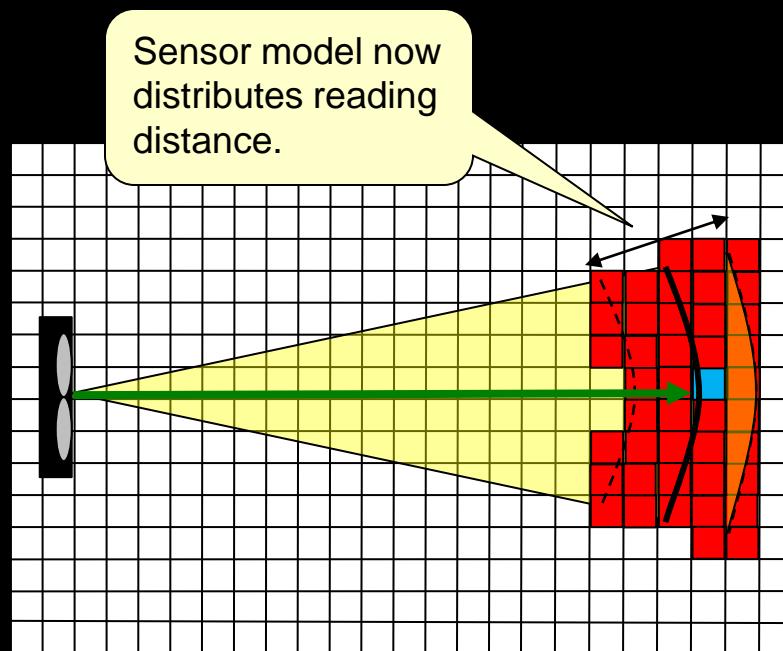
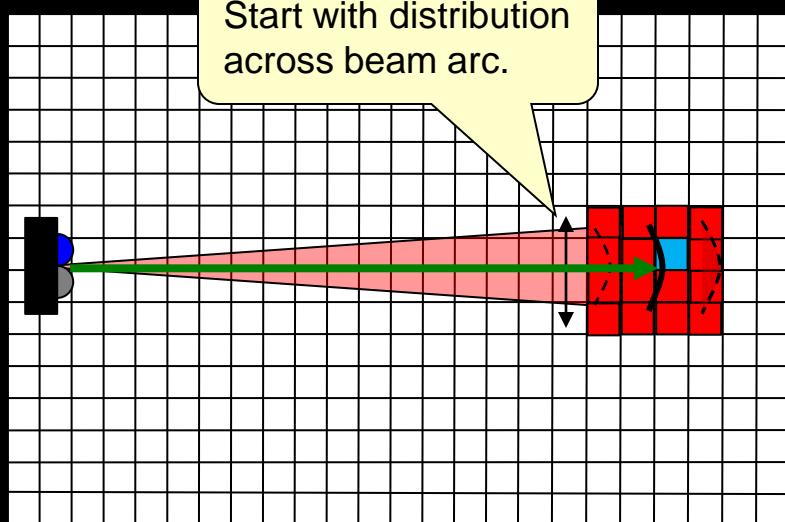
```
FOR a = -σ/2 TO σ/2 BY ω DO {
    objX = xs + (d * cos(φ + a))
    objY = ys + (d * sin(φ + a))
    set grid at (objX, objY) = 1
}
```

φ is the sensor's direction, which is the robot's angle plus the sensor's angle offset.

But first make sure that (objX, objY) is within the grid range!!

# Applying A Sensor Model

- We will also distribute the reading according to the distance error to accommodate inaccurate range readings from the sensor.



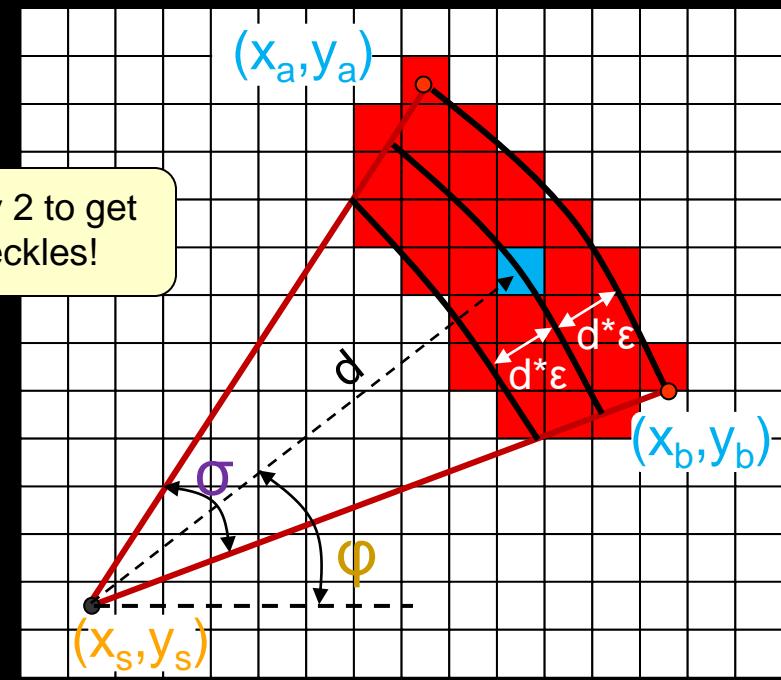
# Sensor Model Implementation

- How do we do this ?
  - we iterate through ranges corresponding to the range  $\varepsilon$  defined by the % distance error (e.g., 0.03 for 3%).

Choose an INC which is less than 1 to make sure that no cells are skipped. Perhaps INC = 0.25.

```
FOR r = d*(1-ε) TO d*(1+ε) BY INC DO {  
    xa = xs + (r * COS(φ + σ/2))  
    ya = ys + (r * SIN(φ + σ/2))  
    xb = xs + (r * COS(φ - σ/2))  
    yb = ys + (r * SIN(φ - σ/2))  
    ω = (σ / √((xa-xb)2 + (ya-yb)2)) / 2  
  
    FOR a = -σ/2 TO σ/2 BY ω DO {  
        objX = xs + (r * cos(φ + a))  
        objY = ys + (r * sin(φ + a))  
        set grid at (objX, objY) = 1  
    }  
}
```

Divide by 2 to get rid of speckles!

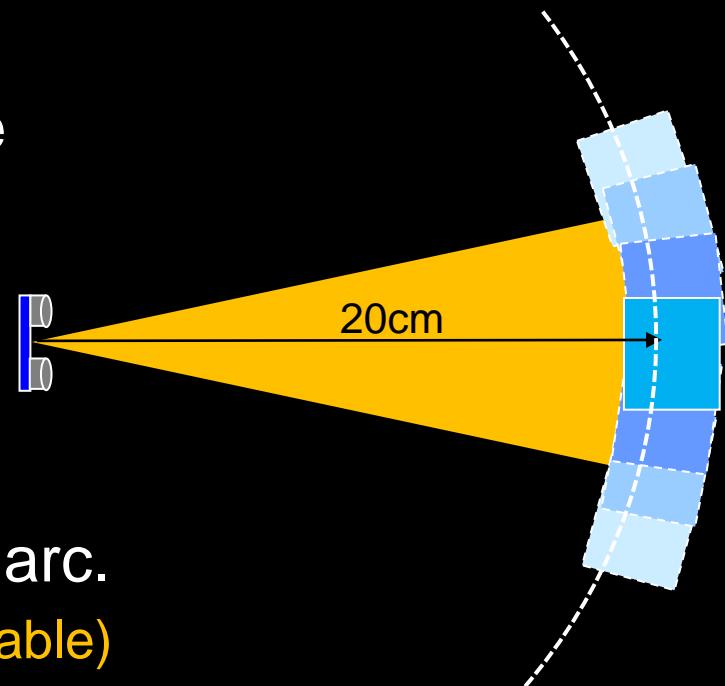


Start the  
Lab ...

# Improved Sensor Model Mapping

# Error Distribution

- When object is detected at, say  $20_{\text{cm}}$ , it can actually be anywhere within the beam arc defined by the  $20_{\text{cm}}$  radius.
- The likelihood (or probability) that the object is *centered* across the arc is greater than if the object was off to the side of the arc.  
(if location is considered to be a random variable)
- We can thus express the sensor reading itself as a set of **probabilities** across the grid.

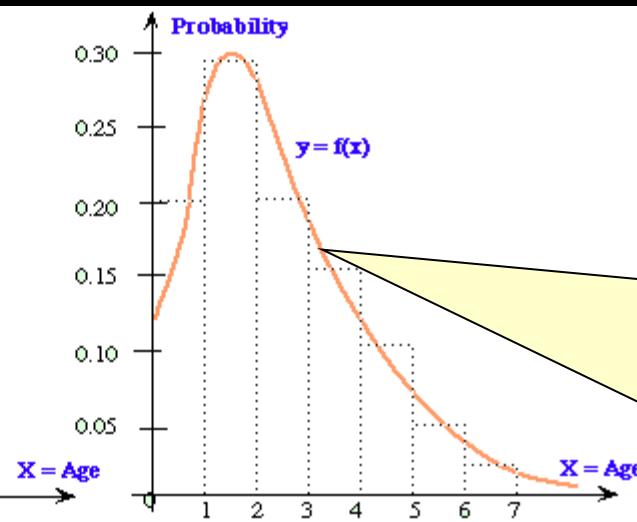
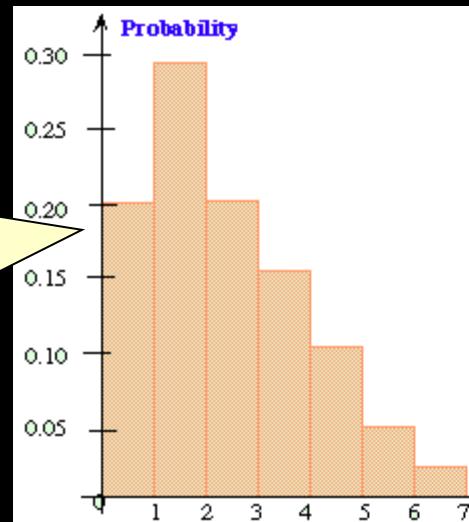


# Probability Density Functions

- Random variables operating in continuous spaces are called *continuous random variables*.
- Assume that all continuous random variables posses a **Probability Density Function** (PDF).

E.g.,

Probability distribution of a car on the road being a certain age.



**Probability Density Function**  
for this distribution.

(Also known as the **Probability Distribution Function**).

# Probability Density Functions

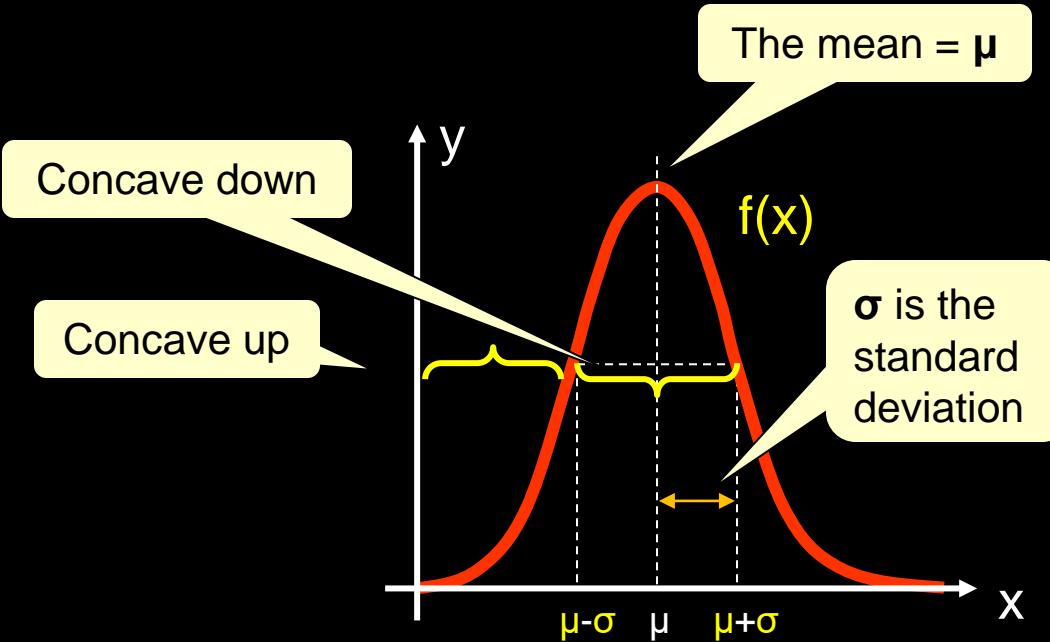
- Common PDF is the **normal distribution**:

- given mean  $\mu$  and variance  $\sigma^2$  the normal distribution is ...

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x - \mu)^2}{2\sigma^2}}$$

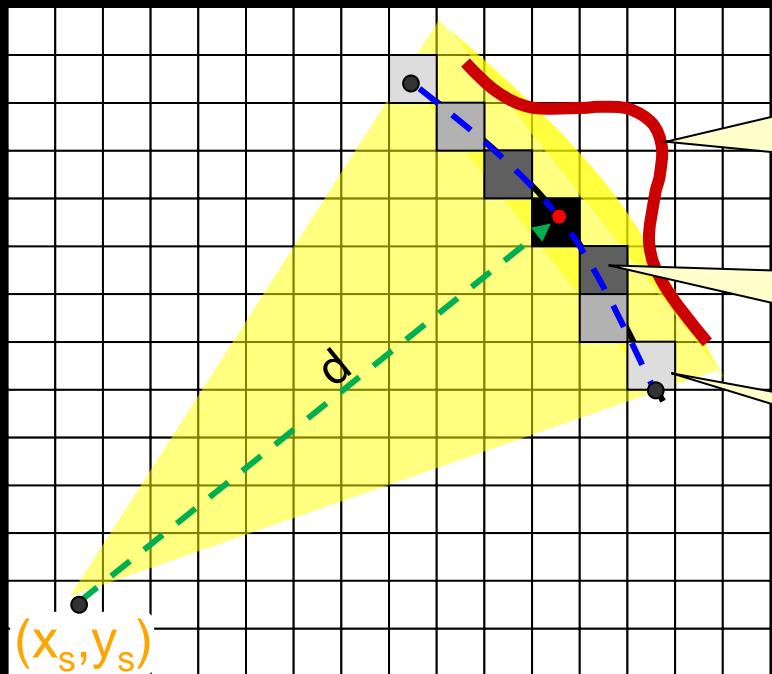


Don't worry, you do not need to understand this.



# Gaussian Distribution

- A more realistic sensor model assigns probabilities to the cells according to some error distribution such as this **Gaussian** (or **Normal**) distribution.



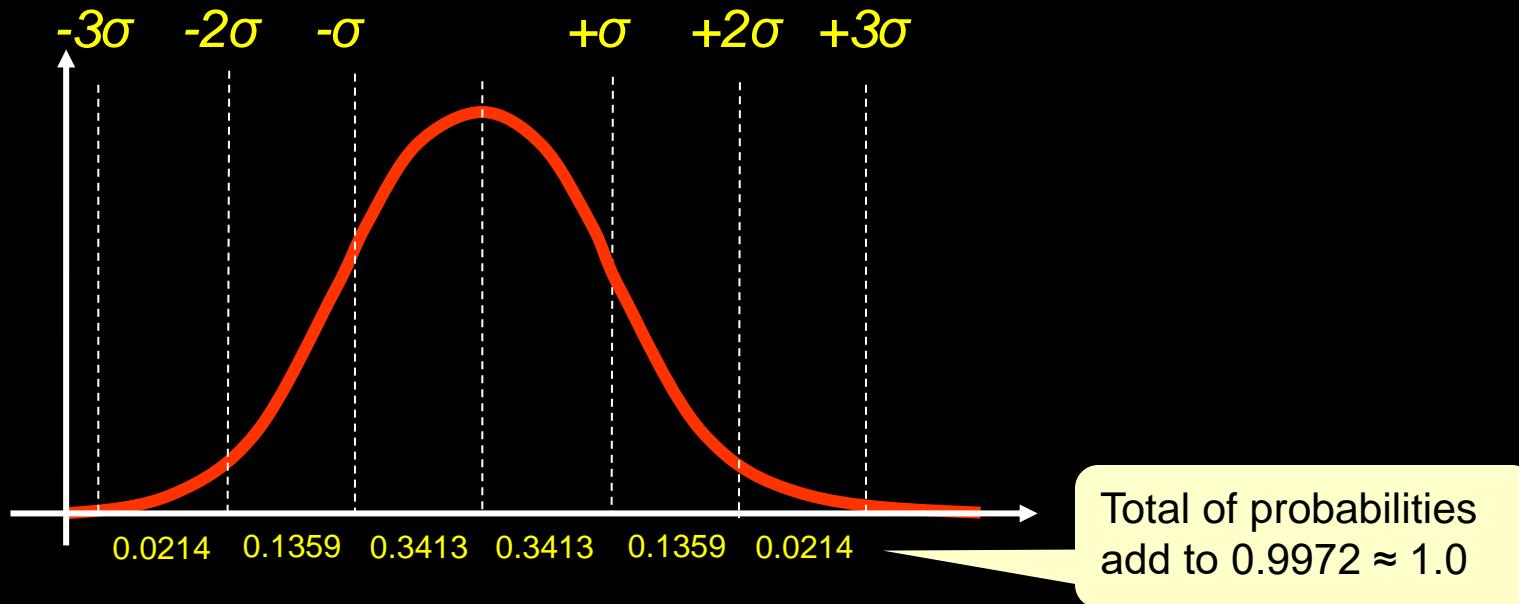
We will now apply this distribution across our arc to assign a “**probability that it is occupied**” value to each cell.

Darker means “**more likely**” that object was at this cell.

Lighter means “**less likely**” that object was at this cell.

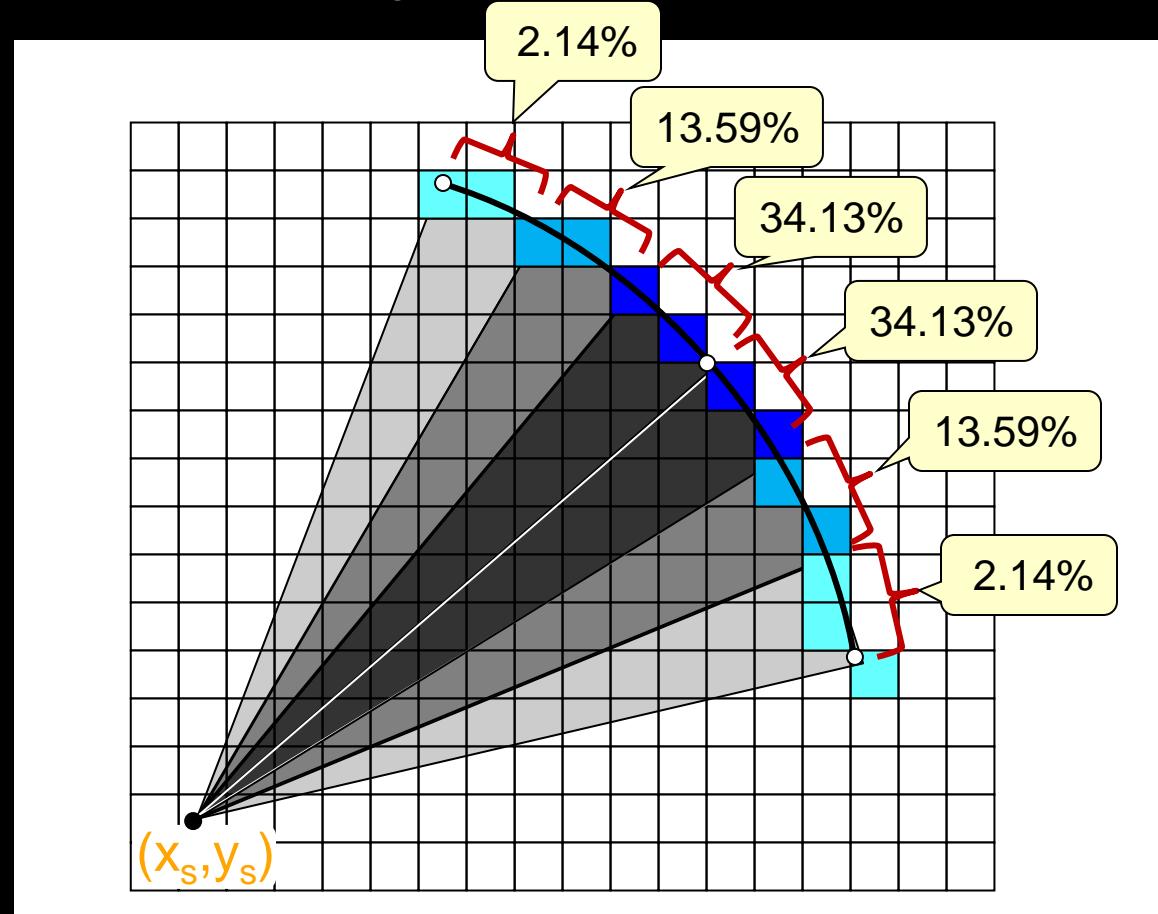
# Gaussian Distribution

- How do we implement this on our occupancy grid ?
- Often the probabilities are approximated using what is known as the *six-sigma* rule. Which essentially divides the probabilities into 6 probability regions.



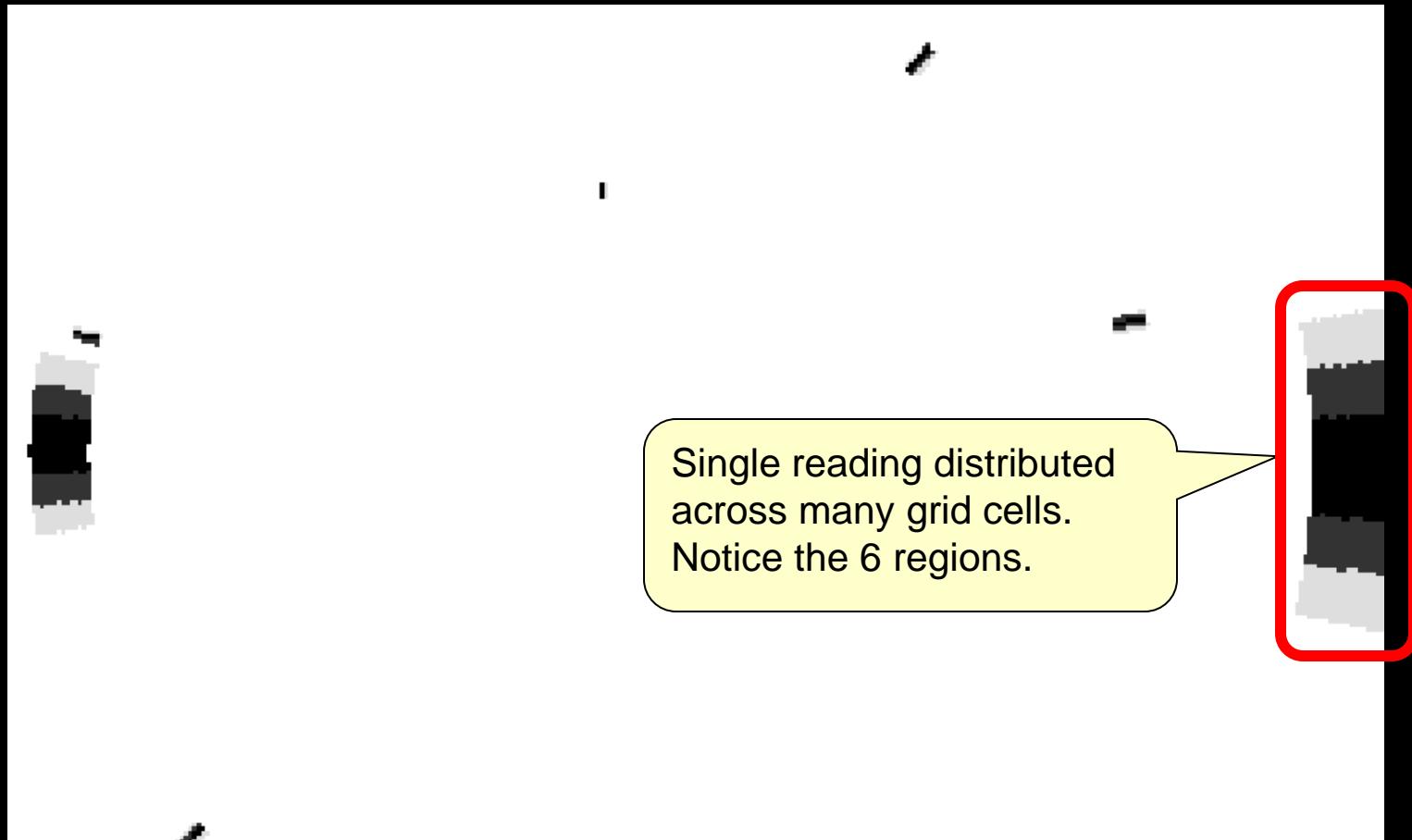
# Applying Gaussian Distribution

- Divide arc into 6 “wedges” and apply the specific probabilities to the cells in each wedge.



# Applying Gaussian Distribution

- Here is the result of applying the Gaussian distribution across the angle:



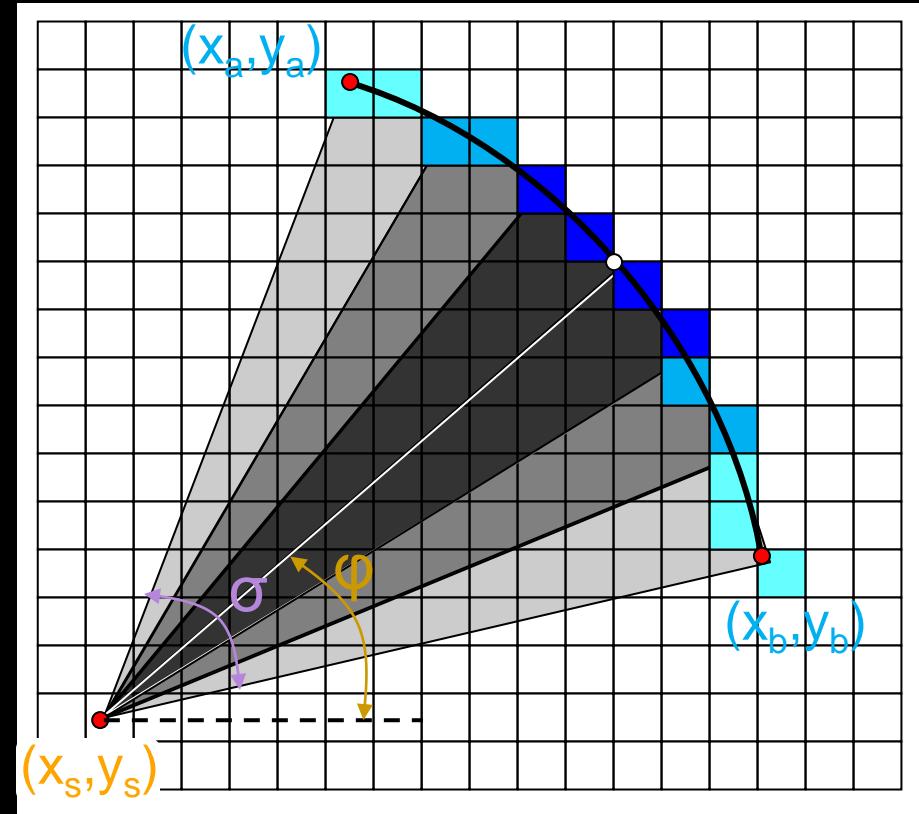
# The code

- Recall the code for filling in grid cells along the arc:

```
 $\omega = \sigma / \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$ 
FOR a = - $\sigma/2$  TO  $\sigma/2$  BY  $\omega$  DO {
    objX =  $x_s + (d * \cos(\phi + a))$ 
    objY =  $y_s + (d * \sin(\phi + a))$ 
    grid[objX][objY] = 1
}
```

- We need to set the **probability** now instead of setting to 1.
- Can grab the probability from a hard-coded array:

```
static final float[] SIGMA_PROB =
{0.0214f, 0.1359f, 0.3413f, 0.3413f, 0.1359f, 0.0214f};
```



# The Code

- Just need to find the index  $i$  to look up into array:

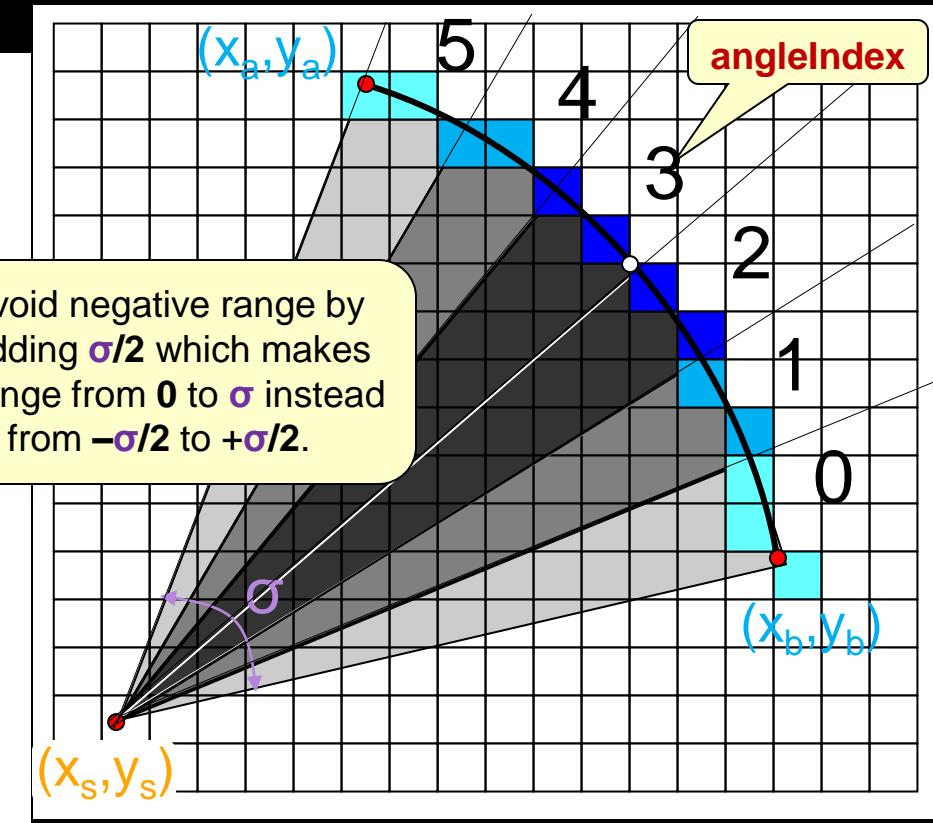
```
SIGMA_PROB = {0.0214, 0.1359, 0.3413,  
               0.3413, 0.1359, 0.0214}
```

$$\omega = \sigma / \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

```
FOR a = -σ/2 TO σ/2 BY ω DO {  
    objX = xs + (d * cos(φ + a))  
    objY = ys + (d * sin(φ + a))  
  
    percentArc = (a + σ/2) / σ
```

This is the amount of processing so far that we reached during the FOR loop (i.e., 0% to 100%)

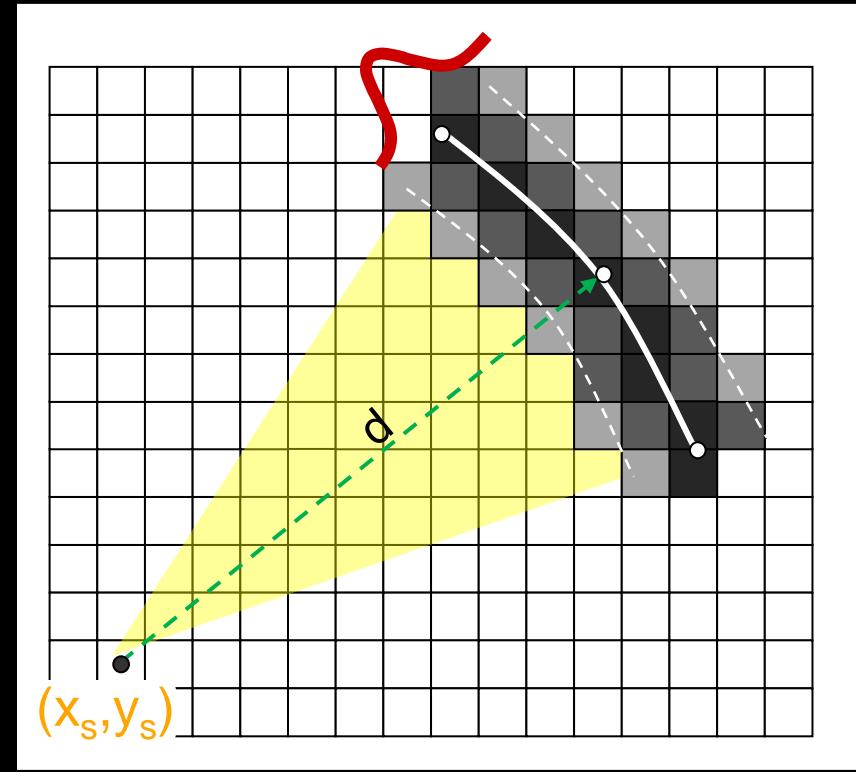
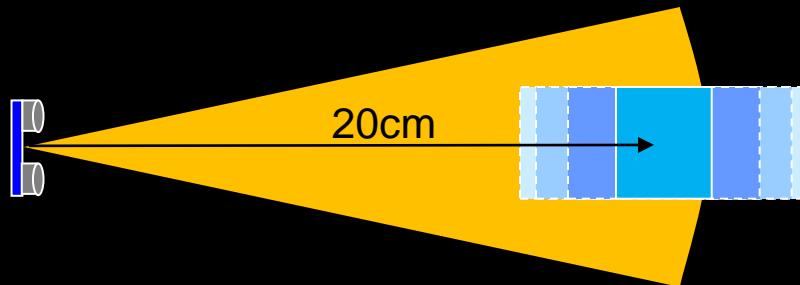
```
angleIndex = (int) (percentArc * 5.99)  
  
grid[objX][objY] =  
    SIGMA_PROB[angleIndex]  
}
```



Multiplying by 5.99 and then truncating to an integer, will ensure indices in the 0 to 5 range).

# Applying Gaussian Distribution

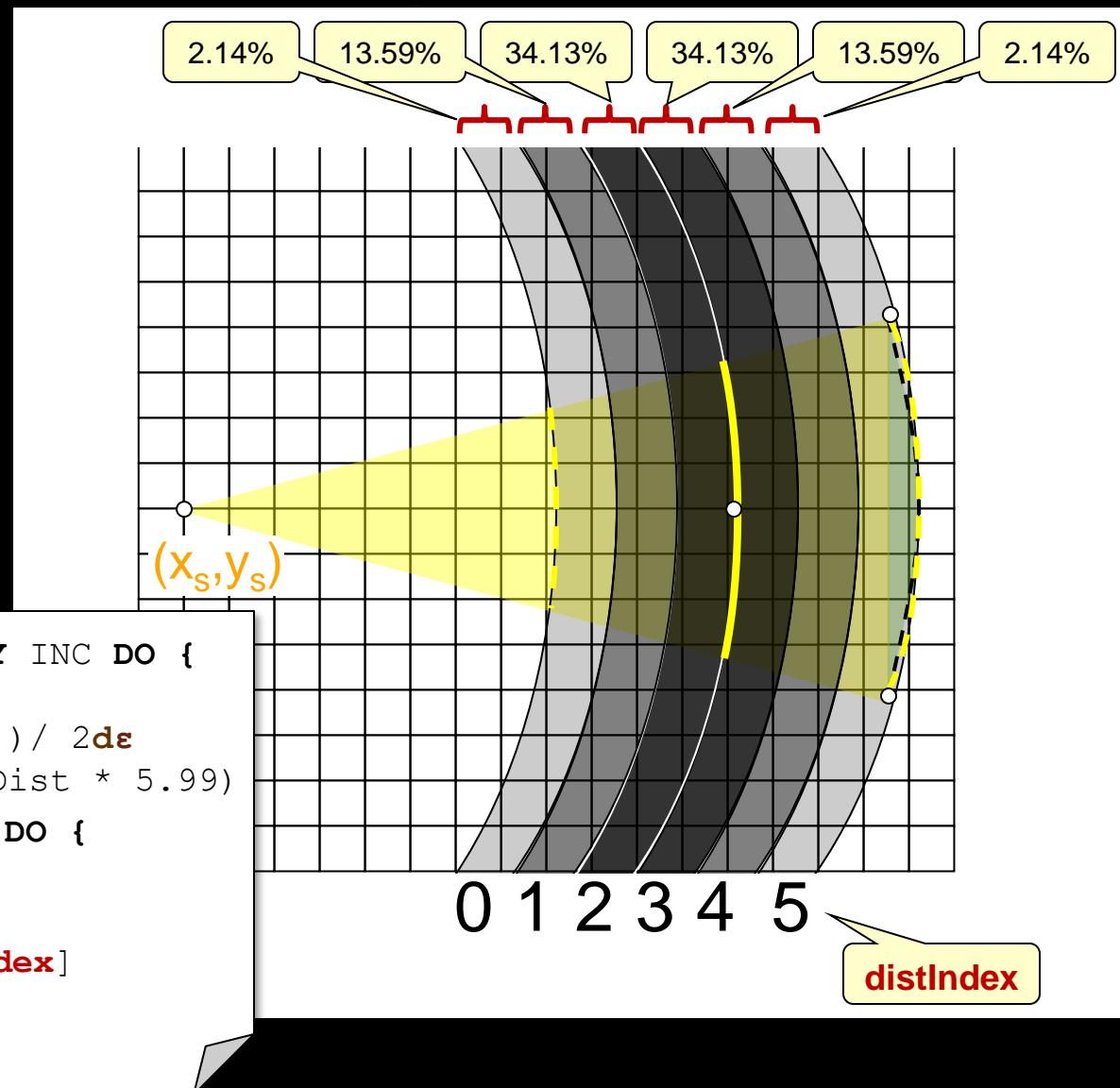
- Should also apply the distribution to **distance** since object is more likely at the distance range measured than closer or further.



# Applying Gaussian Distribution

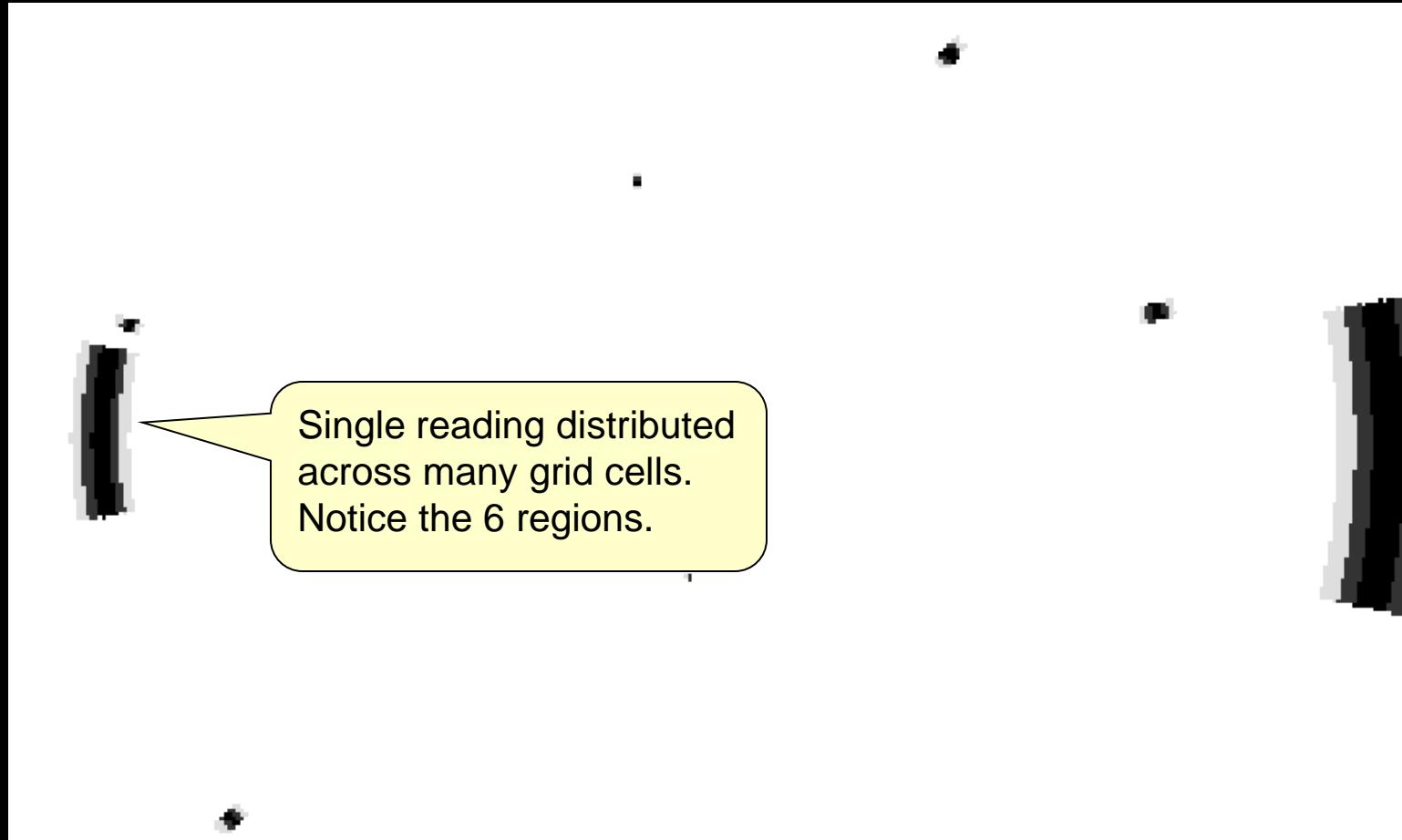
- Divide distance range into 6 “bands” and apply the specific probabilities to the cells in each band.

```
FOR r = d*(1-ε) TO d*(1+ε) BY INC DO {  
    ...  
    percentDist = (r - d*(1-ε)) / 2dε  
    distIndex = (int)(percentDist * 5.99)  
    FOR a = -σ/2 TO σ/2 BY ω DO {  
        ...  
        grid[objX][objY] =  
            SIGMA_PROB[distIndex]  
    }  
}
```



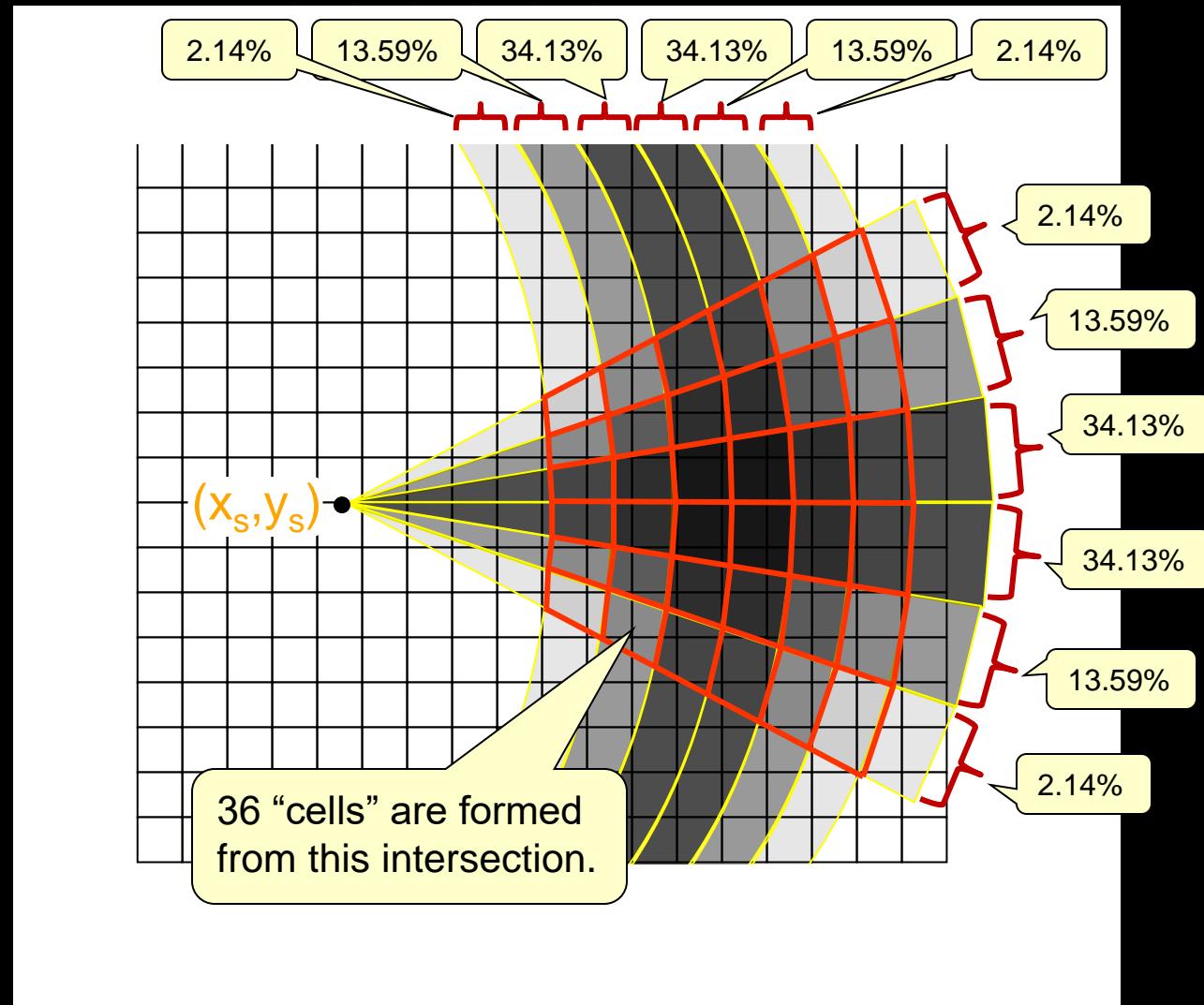
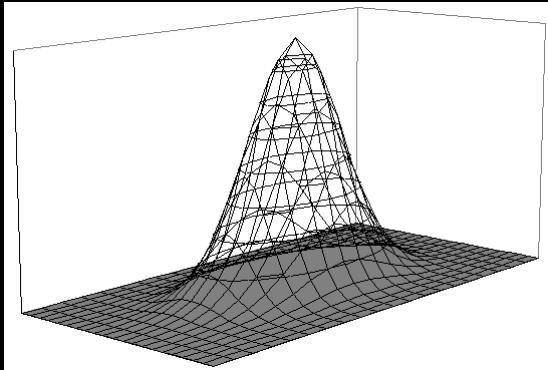
# Applying Gaussian Distribution

- Here is the result of applying the Gaussian distribution to the distance:



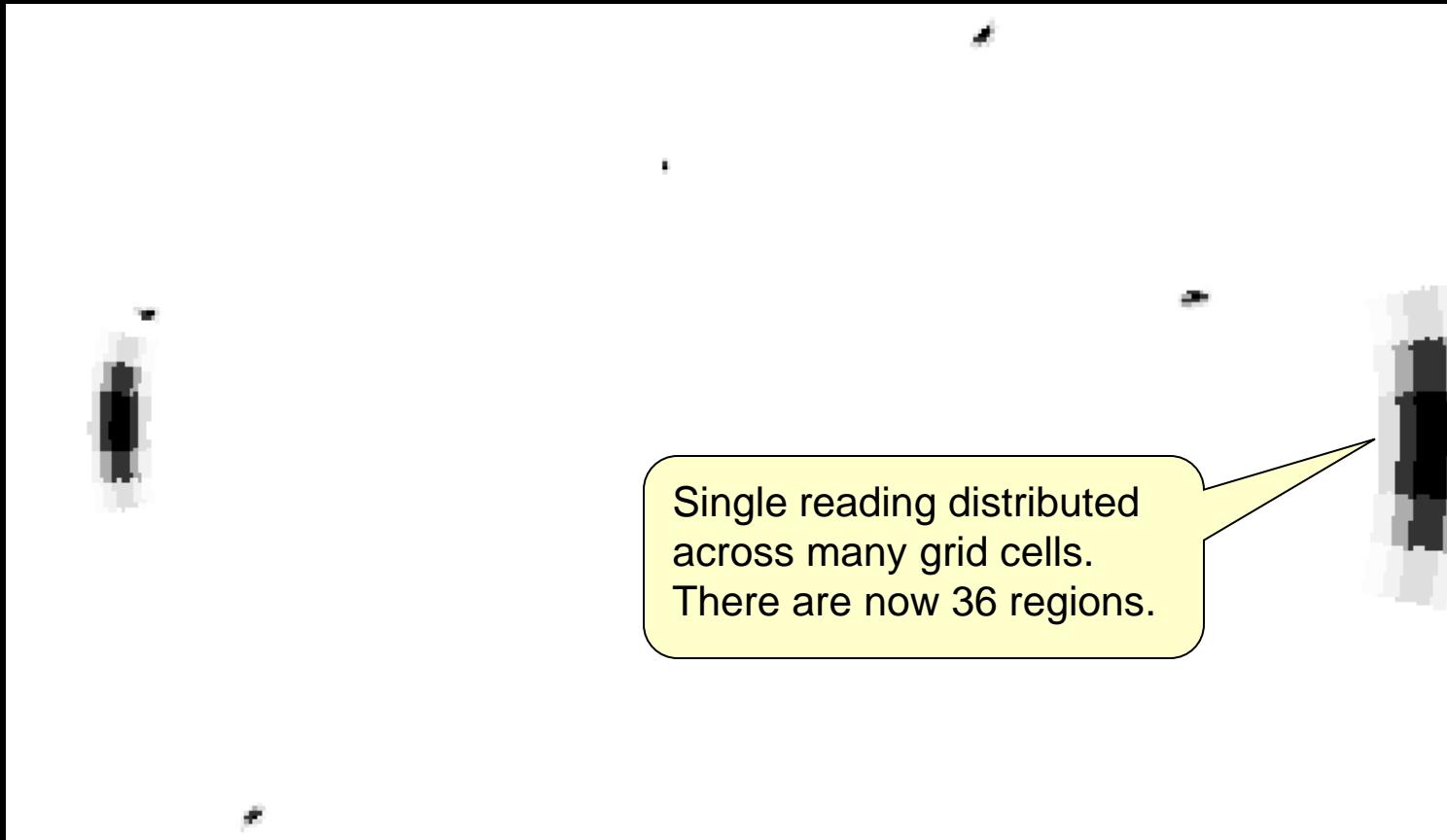
# Applying Gaussian Distribution

- Finally, apply probabilities along both angle as well as distance:



# Applying Gaussian Distribution

- Here is the result of applying the Gaussian distribution to both angle and distance.



# Applying Gaussian Distribution

- Here are the probabilities that are to be assigned to each of the 36 cells:

0.05%	0.29%	0.73%	0.73%	0.29%	0.05%
0.29%	1.85%	4.64%	4.64%	1.85%	0.29%
0.73%	4.64%	11.65%	11.65%	4.64%	0.73%
0.73%	4.64%	11.65%	11.65%	4.64%	0.73%
0.29%	1.85%	4.64%	4.64%	1.85%	0.29%
0.05%	0.29%	0.73%	0.73%	0.29%	0.05%

- Can just store the probabilities in a 1D array:

```
static final float[] SIGMA_PROB =  
    {0.0214f, 0.1359f, 0.3413f, 0.3413f, 0.1359f, 0.0214f};
```

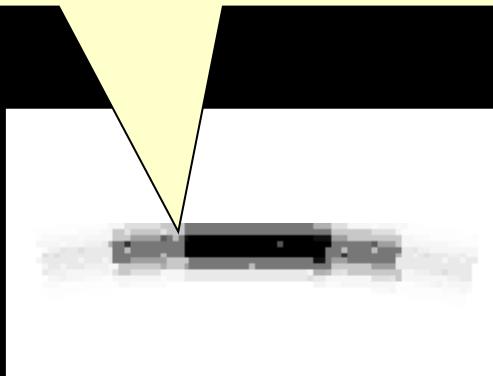
- Then combine both directions through multiplication:

```
probability = SIGMA_PROB [angIndex] * SIGMA_PROB [distIndex];
```

# Ensuring Consistency

- The technique just shown, if not careful, does not properly assign probabilities across the wedge for a single sensor reading.

Due to round-off inaccuracies, there will likely be some grid cells counted twice and some not counted during a single update. This may lead to a **speckled** pattern.



This can also occur if the increment on the **FOR** loop for the distance is not small enough. It should be smaller than the grid's precision to ensure that no grid cells are missed:

```
for (double r=0; r<limit; r+=INC) {  
    ...  
}  
e.g., INC = 1  
      INC = 0.75  
      INC = 0.5  
      INC = 0.25
```



0.25 avoids speckled pattern.

# Ensuring Consistency

---

- To avoid speckled pattern, create a temporary grid

- Create to be same size as entire grid

```
temp = new float[width][height];
```

- Initialize all values to 0

```
temp[i][j] = 0;
```

- Apply all readings to the temporary grid by *setting* the cell values (i.e., not adding them)

```
temp[i][j] = SIGMA_PROB[angIndex] * SIGMA_PROB[distIndex];
```

- Merge temporary grid with complete map once reading probabilities have been completed

```
grid[i][j] += temp[i][j]
```

# Non-Obstacle Certainty

- Another way to refine the grid is to say something about the certainty that an obstacle is NOT there.

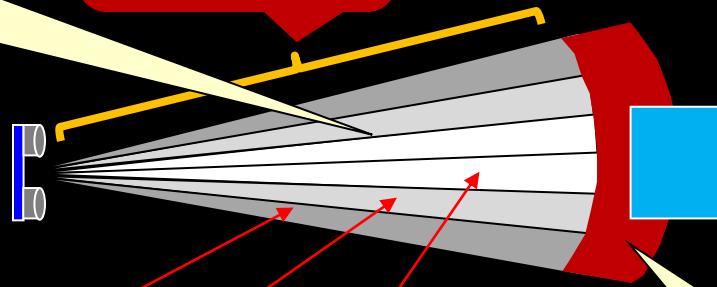
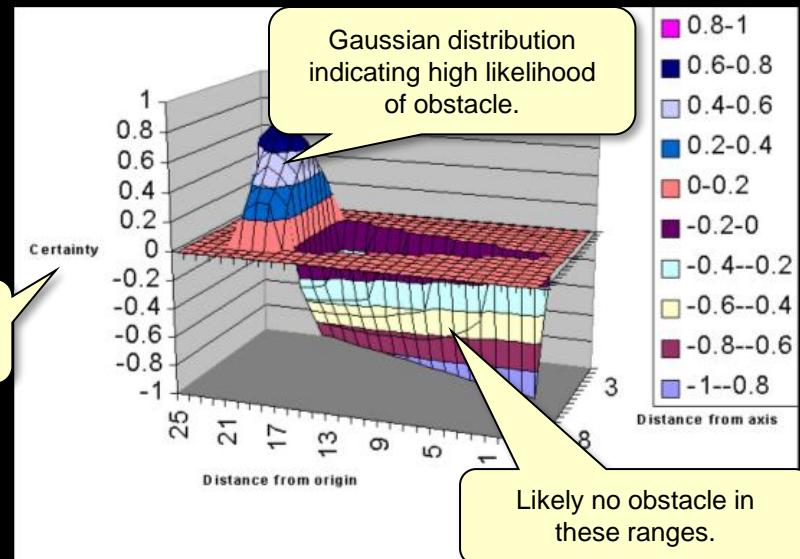
Our scale goes from 0 to 1, not -1 to 1 as shown here.

Obstacle CANNOT lie in here otherwise distance reading would have been smaller.

We won't apply any distance distribution.

We can **decrease** occupancy grid values here according to the angular distribution.

{ 0.0214f, 0.1359f, 0.3413f, 0.3413f, 0.1359f, 0.0214f };



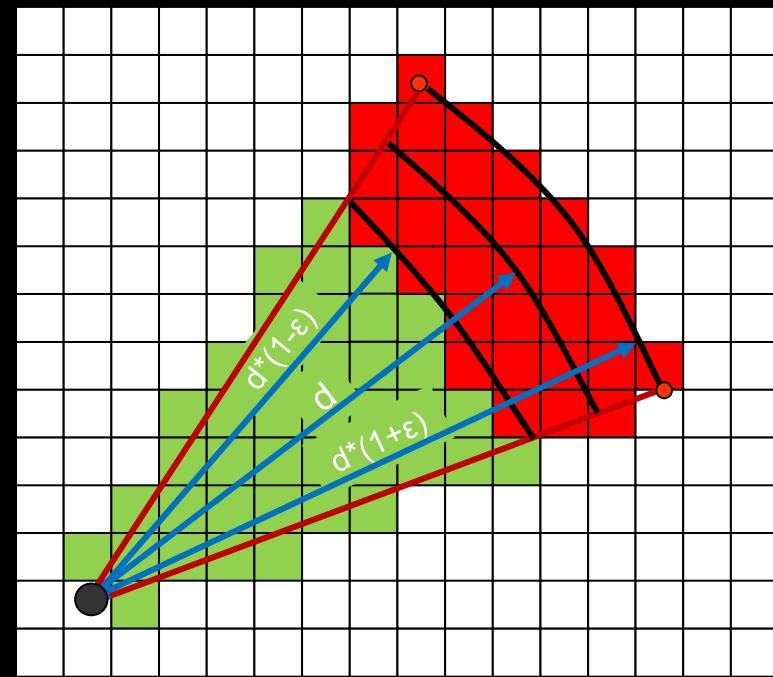
Obstacle lies in here somewhere according to a Gaussian distribution.

# Non-Obstacle Certainty Range

- Currently, the current FOR loop code only updates radius values that are between  $d^*(1-\varepsilon)$  and  $d^*(1+\varepsilon)$  ... red cells.
- But now we need to update cells with radius values from 0 up to  $d^*(1-\varepsilon)$  as well ... green cells:

Start at 0 now instead of  $d^*(1-\varepsilon)$

```
FOR r = 0 TO d*(1+ε) BY INC DO {  
    ...  
    FOR a = -σ/2 TO σ/2 BY ω DO {  
        ...  
        IF (r < d*(1-ε)) THEN  
            lighten the cell using angular  
            distribution only (not distance)  
        ELSE  
            darken the cell as before using  
            both angular and distance  
    }  
}
```

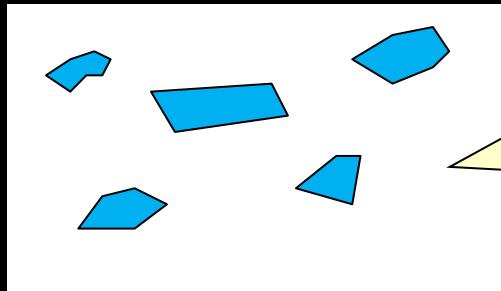


Start the  
Lab ...

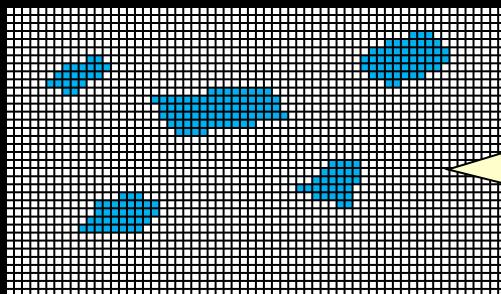
# Finding Obstacle Borders

# Raster Vs. Vector Maps

- Recall that large environments with few and simple obstacles take less space to store as vector than as occupancy grid:



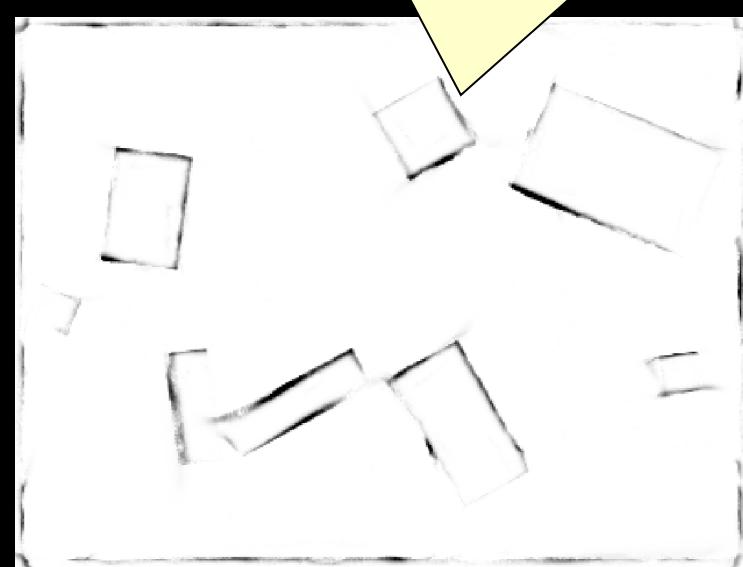
Only need to store a few vertex coordinates and edge connections.



Both “occupied” and “empty” regions take up storage space.

Our occupancy grids represent **400 x 300** grid cells (i.e., 120K).

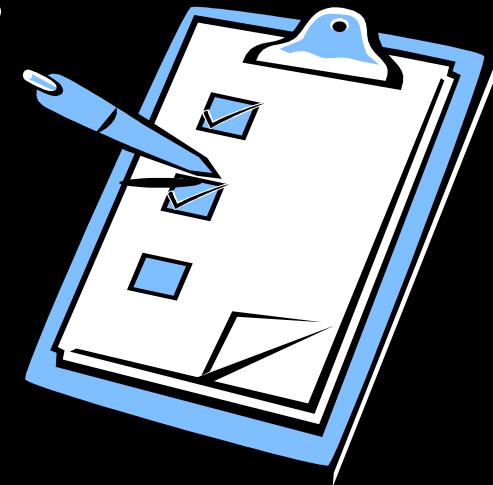
Since each cell stores a float, it requires  $120K * 4 = \mathbf{480K}$  of storage to represent the map.



# Raster Vs. Vector Maps

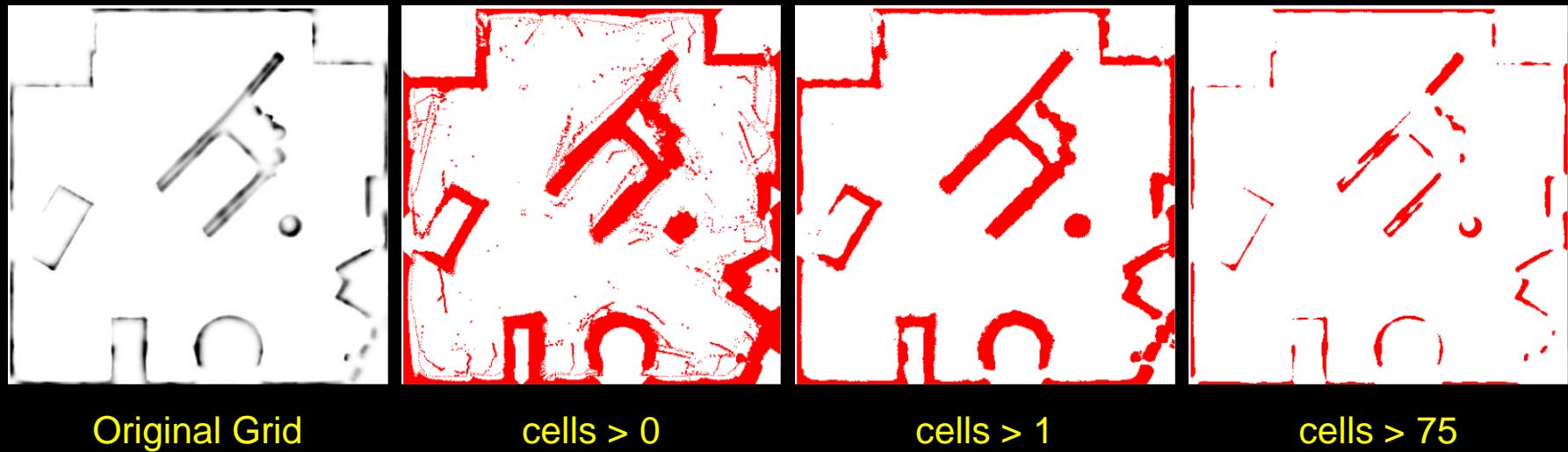
---

- It would be better to store the map as a vector
  - will take up less space
  - can represent objects as polygonal models
- Doing this will require 3 steps:
  1. Decide which grid cells are actually considered occupied
  2. Determine the **borders** of the obstacles
  3. Convert the borders into **polygons**
- In this lab period, we will concentrate only on the first two steps.



# Step 1: Thresholding

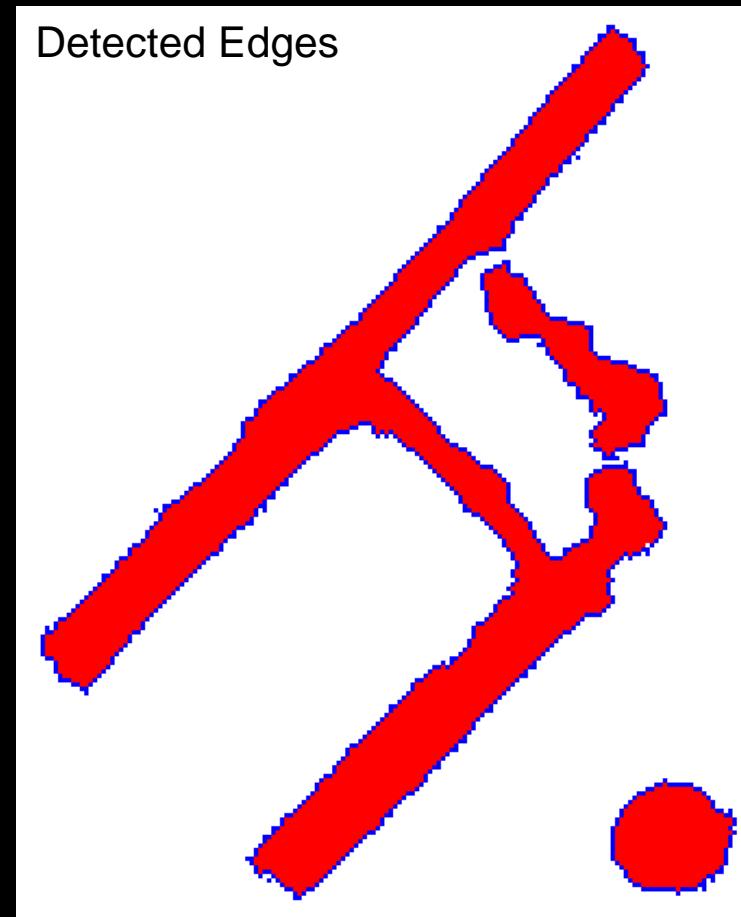
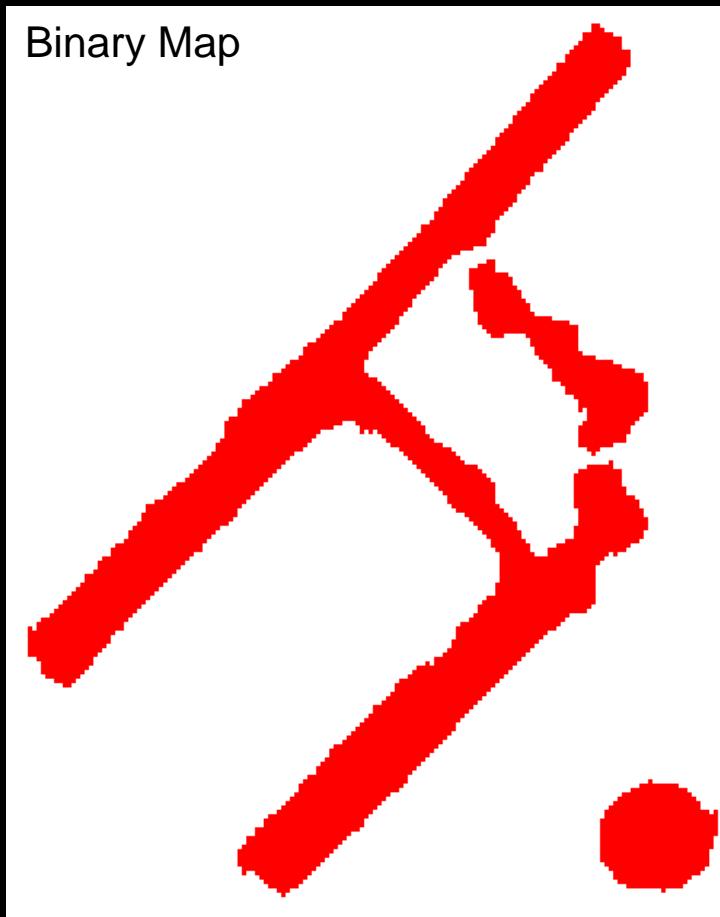
- Since map cells represent the “**probability**” that the cell is part of an obstacle, we must decide on whether or not a particular grid cell is occupied.
- Need to have a binary grid by applying a threshold:



- We will now work with the binary grid instead of the original.
  - Grid cell values are: **0** = if no obstacle or **1** = if obstacle

# Step 2: Border Detection

- We now need to process the grid to determine the borders of all obstacles.

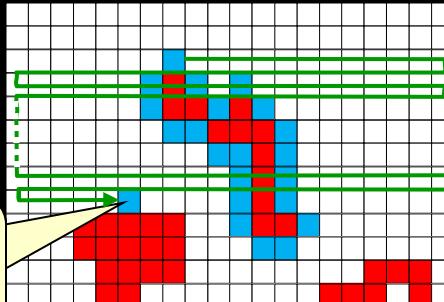


# Finding the Start of Each Border

- Begin border detection by finding any point on a border. Easiest way is to check all cells starting from top to bottom going left to right until a non-zero cell is found:

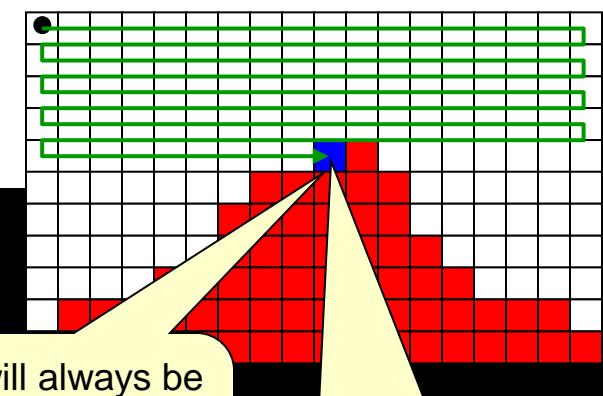
This loop goes backwards!

```
FOR each Y from height-1 to 1 {  
    FOR each X from 0 to width-1 {  
        IF ((grid[X][Y] == 1) AND ((X == 0) OR grid[X-1][Y] == 0))  
            grid[X][Y] = 2; // i.e. BORDER  
            TraceWholeBorder(grid, X, Y)  
    }  
}
```



After one obstacle done, go back up to look for the start of the next obstacle to be traced.

(X,Y) will always be the top left corner of a new obstacle to be traced.

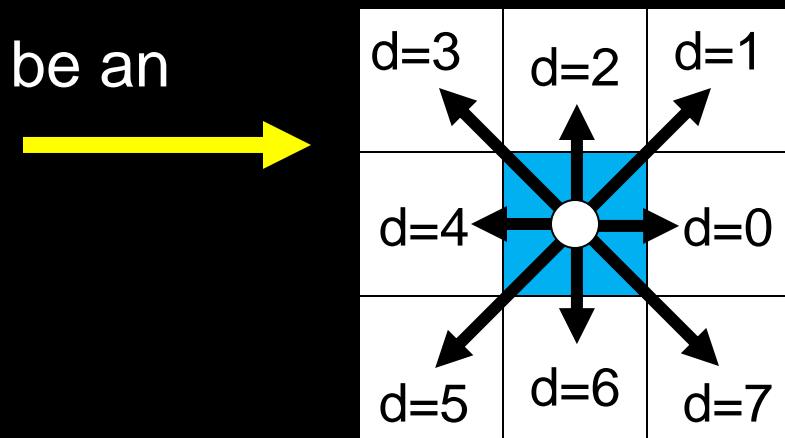


As we trace, we will set border cells to have a value of 2.

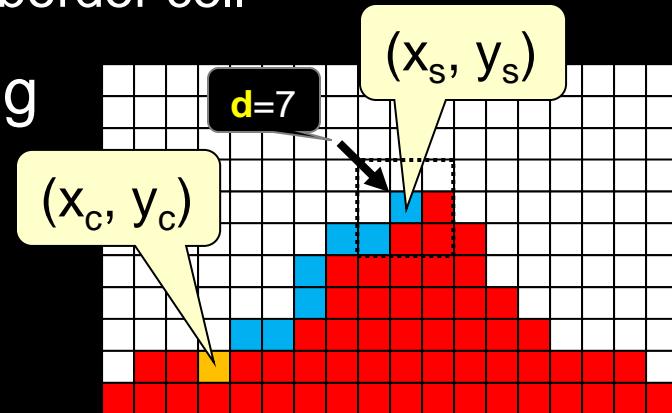
# TraceWholeBorder() Algorithm

- Define direction **d** around a cell to be an integer from **0** to **7** as shown here and then define **d'** as follows:

```
IF (d is even) THEN  
    d' = (d+7) modulus 8  
OTHERWISE  
    d' = (d+6) modulus 8
```

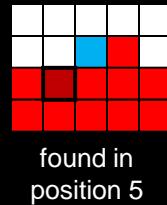
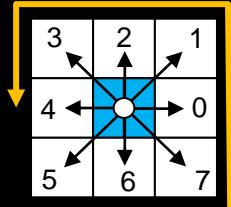


- Border will be traced counter-clockwise (CCW)
  - d** is direction we came in from for border cell just added
  - d'** is direction to start looking at to find next border cell
- Begin trace with start cell  $(x_s, y_s)$  coming from direction **d = 7**, and maintain a current point  $(x_c, y_c)$  which is the last point that we added to the border.

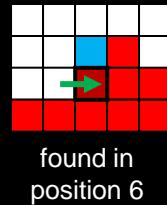


# TraceWholeBorder() Algorithm

- To find the next border cell do this:
  - Starting with direction  $d'$ , check pixels around  $(x_c, y_c)$  in CCW order (i.e., increment  $d'$ ) until either:
    - a non-zero (i.e., obstacle) cell is found,
    - or all 8 directions were checked and no non-zero cell was found.
- Here are some scenarios starting with  $d' = 5$ :



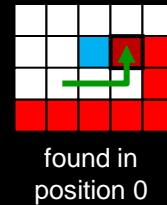
found in position 5



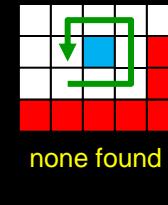
found in position 6



found in position 7

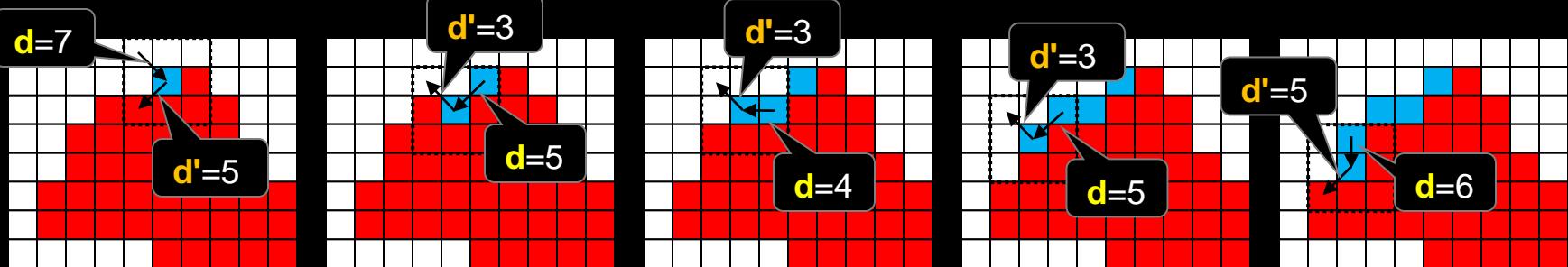


found in position 0



none found

- Notice how  $d$  and  $d'$  change as the trace proceeds:

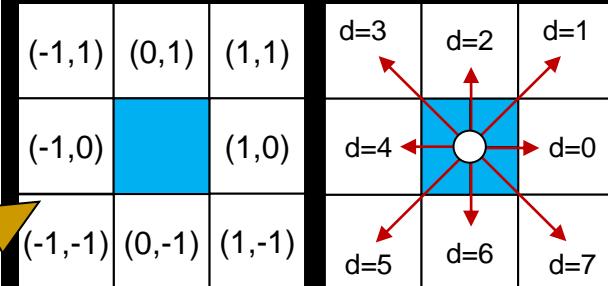


# Algorithm Pseudocode

```

traceWholeBorder(xs, ys) {
    xc = xs
    yc = ys
    d = 7
    REPEAT {
        IF (d is even) THEN
            d' = (d+7) modulus 8
        OTHERWISE
            d' = (d+6) modulus 8
        found = false
        REPEAT 8 TIMES {
            tempX = xc + the xOffset in direction of d'
            tempY = yc + the yOffset in direction of d'
            IF tempX and tempY are NOT beyond the grid boundaries THEN {
                IF grid at cell (tempX, tempY) is non-zero THEN {
                    xc = tempX
                    yc = tempY
                    set grid cell at (xc, yc) to be a border (i.e., 2)
                    d = d'
                    found = true
                    break out of the inner repeat loop
                }
            }
            d' = (d'+1) modulus 8;
        }
        IF not found THEN {
            set grid cell at (xc, yc) to 0
            done tracing ... quit the function
        }
        IF (xc, yc) is the same as (xs, ys) THEN
            done tracing ... quit the function
    }
}

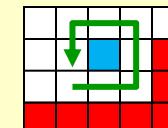
```



(xOffset,yOffset)  
values for each  
direction

Happens when (xc, yc) is on the grid boundaries. Need to check for this otherwise we can get an **ArrayIndexOutOfBoundsException**  
DO NOT use the **isInsideGrid()** function.

Must be a single point since tracing all 8 directions did not find a non-white cell. So, erase it.



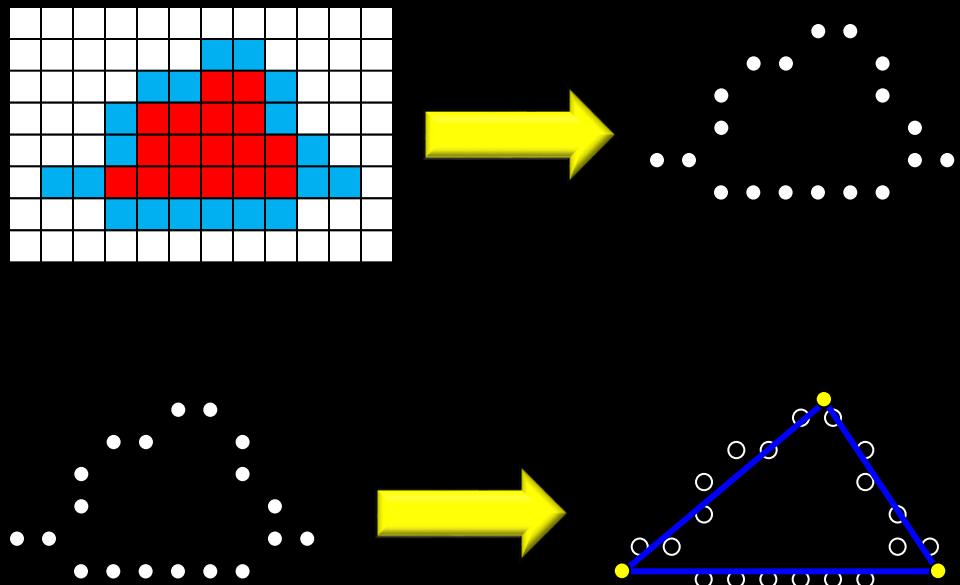
We finished tracing when we reach the start again.

Start the  
Lab ...

# Converting to Vector Maps

# Converting To Vector

- Once we have detected object borders within the occupancy grid, we can convert to polygons.
  - Consider each obstacle separately.
  - When tracing a border, build up a list of the border points for that obstacle in sequential order ... assume each grid cell is a point.
- Then run a line-fitting algorithm



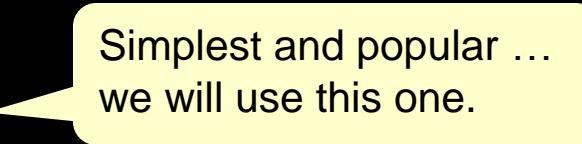
# Converting To Vector

---

- Key issues:

- How do we group points into line segments ?
- How can we detect and eliminate noisy data ?

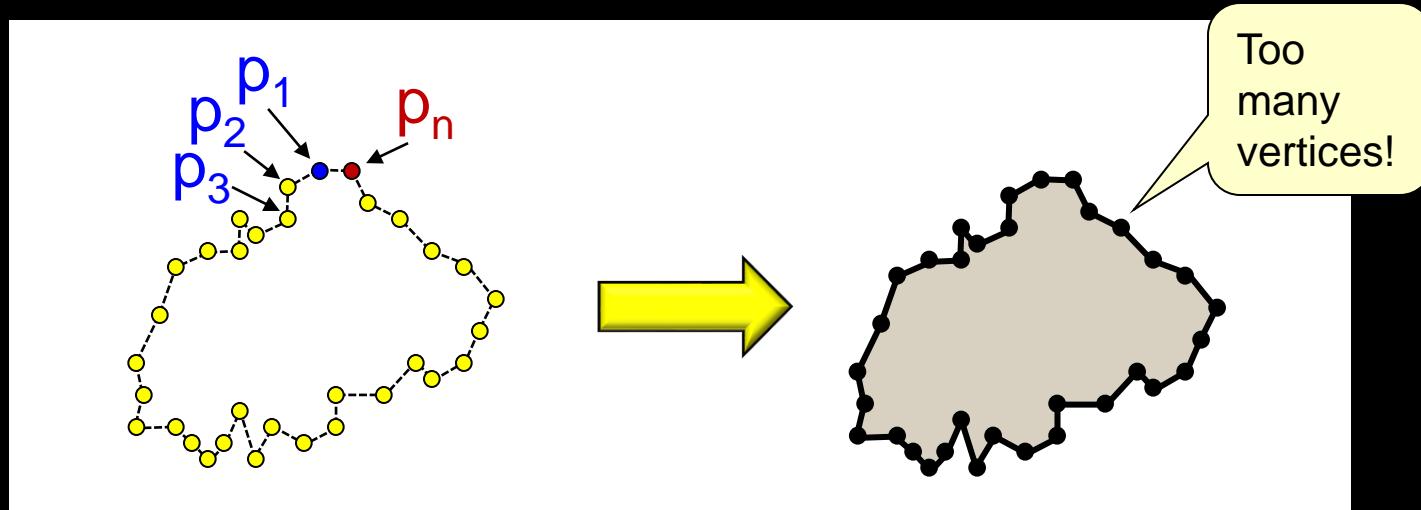
- There are a variety of common techniques:

- Split & Merge
- Incremental      }     

Simplest and popular ...  
we will use this one.
- Line Regression
- RANSAC (Random Sample Consensus)
- Hough-Transform
- EM (Expectation-Maximization)

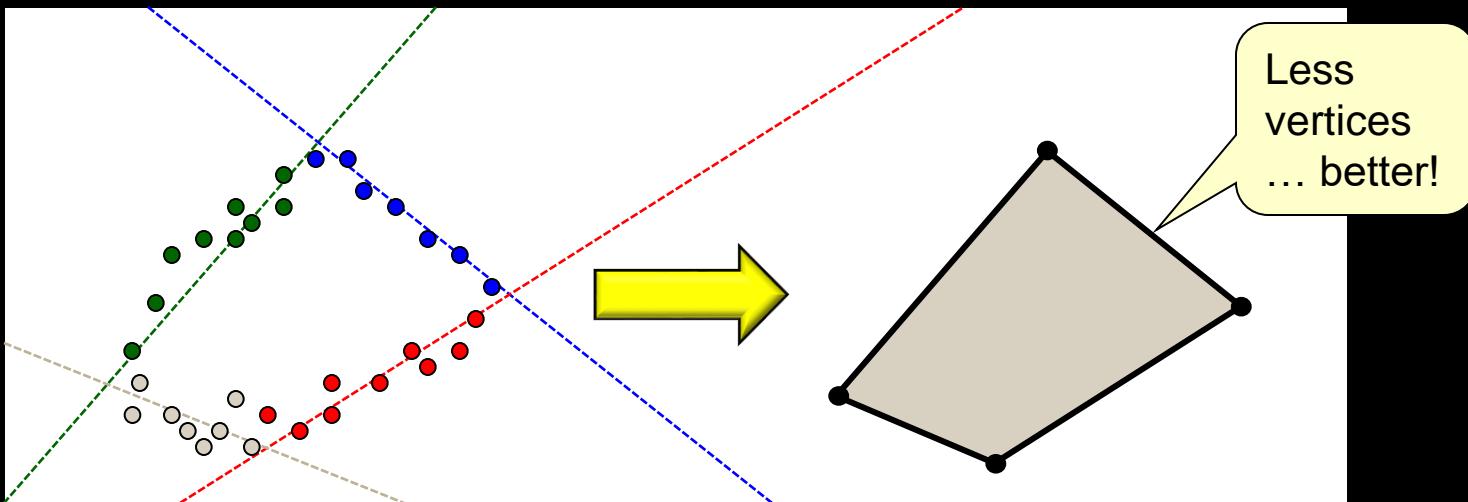
# The Main Idea

- Consider a set of border points:
  - $P = \{p_1, p_2, p_3, \dots, p_n\}$  where  $p_i = (x_i, y_i)$ ,  $1 \leq i \leq n$
  - As we do a border-tracing algorithm, these points will be coming in a counter-clockwise order
- We can simply connect all points in order, but this is too many points and assumes that all points are valid.



# The Main Idea

- It is better to try and “fit” lines to the data:



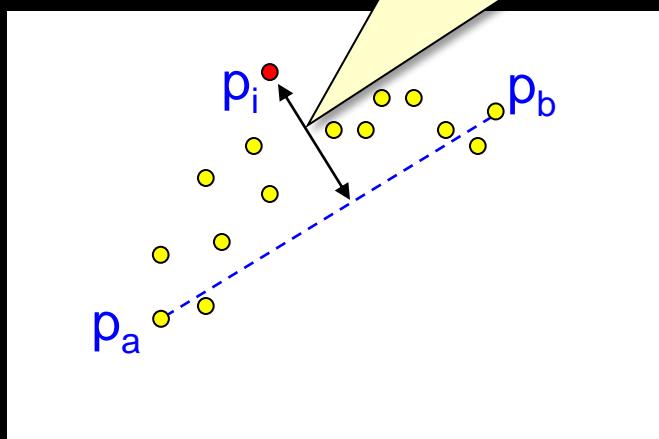
- But how do we know which points fit to a line ?
  - Assume consecutive points lie on the same line unless they are too far away from the line.

# The Main Idea

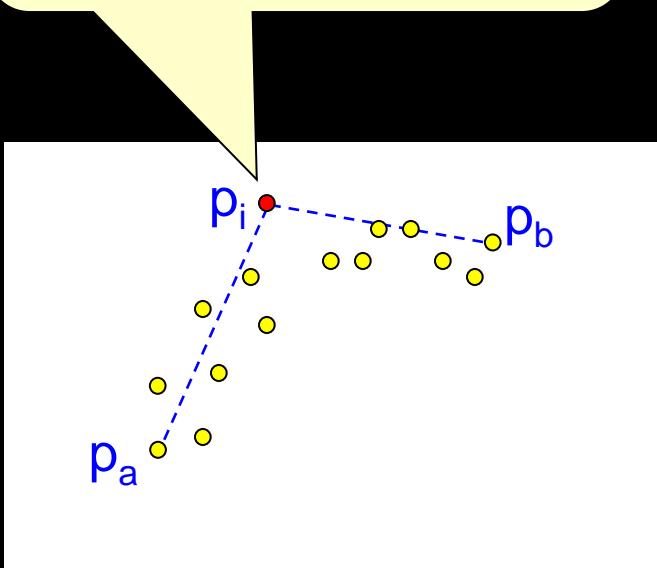
- Let  $p_i$  be point of maximum distance from line  $L = \overline{p_a p_b}$

Distance from  $p_i$  to  $L$  is:

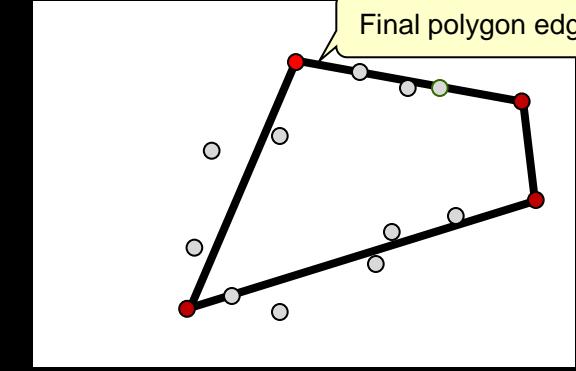
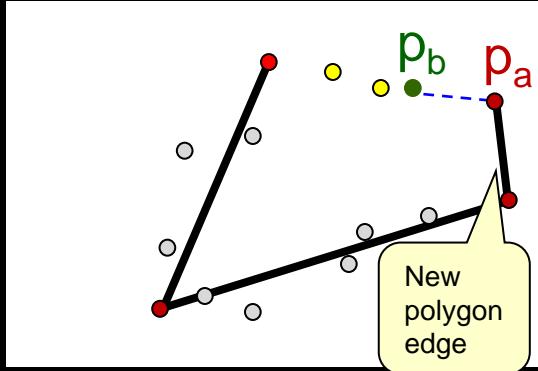
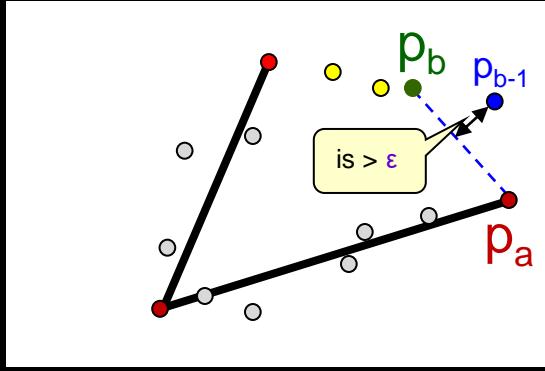
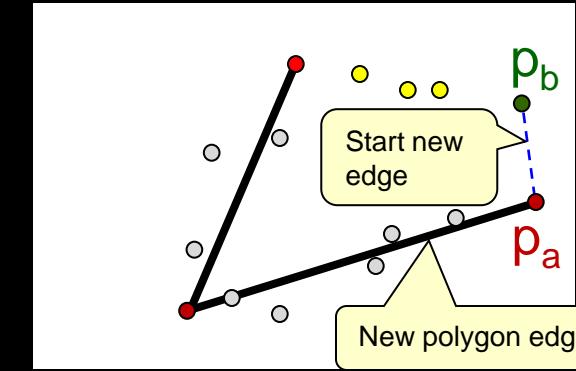
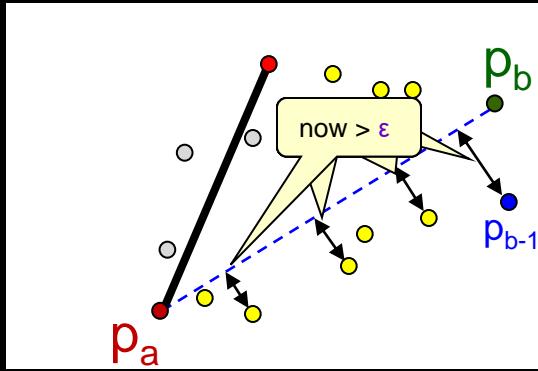
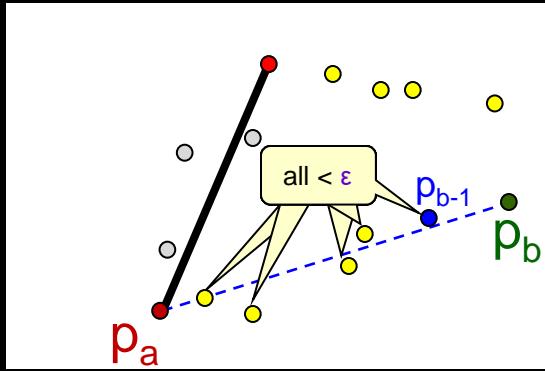
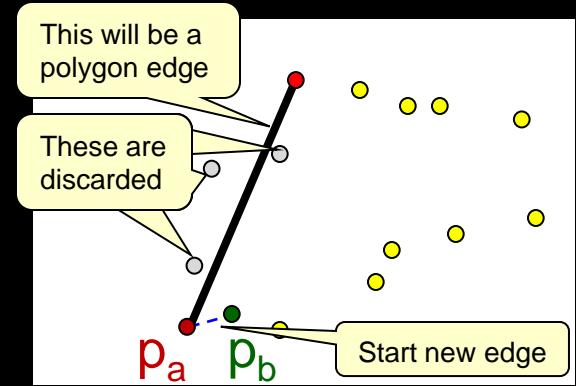
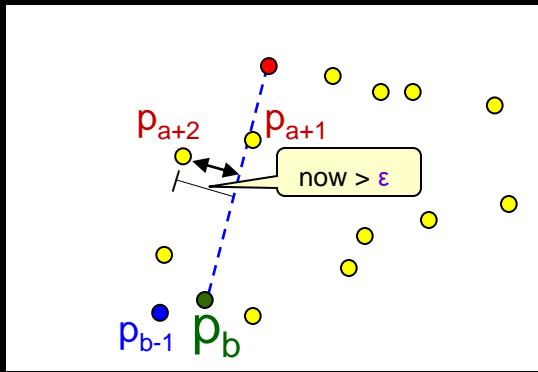
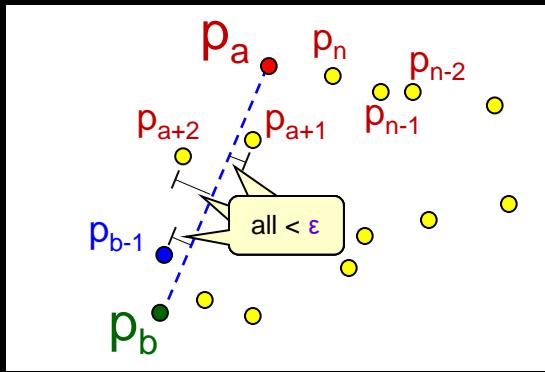
$$\frac{|(x_b - x_a)(y_a - y_i) - (x_a - x_i)(y_b - y_a)|}{\sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}}$$



If  $p_i$  is too far away, then this data must represent more than one line (i.e., two distinct polygon edges).



# The Incremental Algorithm



# The Pseudocode

- Here is the algorithm for point set  $p_1, p_2, p_3, \dots, p_n$ :

```
1. Set polygon to be a new obstacle with no vertices
2. First, make sure the point set has at least 3 points, otherwise quit
3. Set a = 1, and set pa to be the polygon's first vertex
4. FOR index b = 2 to n DO {
5.   FOR index i = a+1 to b-1 DO {
6.     IF point pi is too far from line papb THEN {
7.       Add pb-1 as the next vertex of the polygon unless it is the same as the last vertex added.
8.       Set a = b-1.
9.     }
10.   }
11. }
```



Note that the pseudocode has points with indices 1 to **n** but our **Obstacle** indices go from 0 to **n-1**.

**Too far** means greater than some **LINE\_TOLERANCE** (i.e., denoted as  $\epsilon$  in the previous slide)

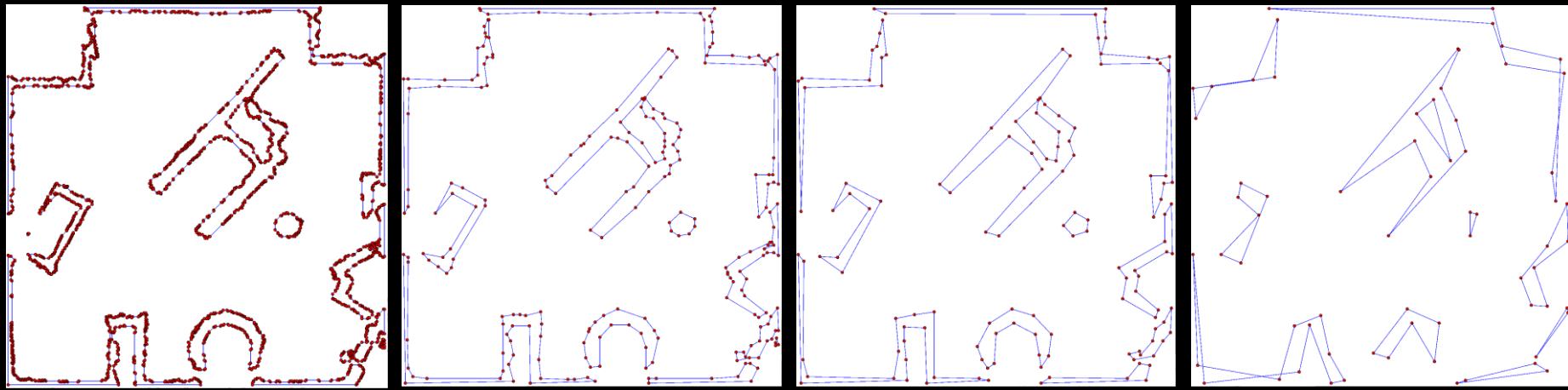


Since variable **a** has been changed, **p<sub>a</sub>** must also be updated.

- Once the polygon is created, we should check to make sure it has at least three vertices. If not, we discard it since it is not a proper polygon.

# The Incremental Algorithm

- By increasing the line-fit tolerance, the number of polygon vertices decreases:



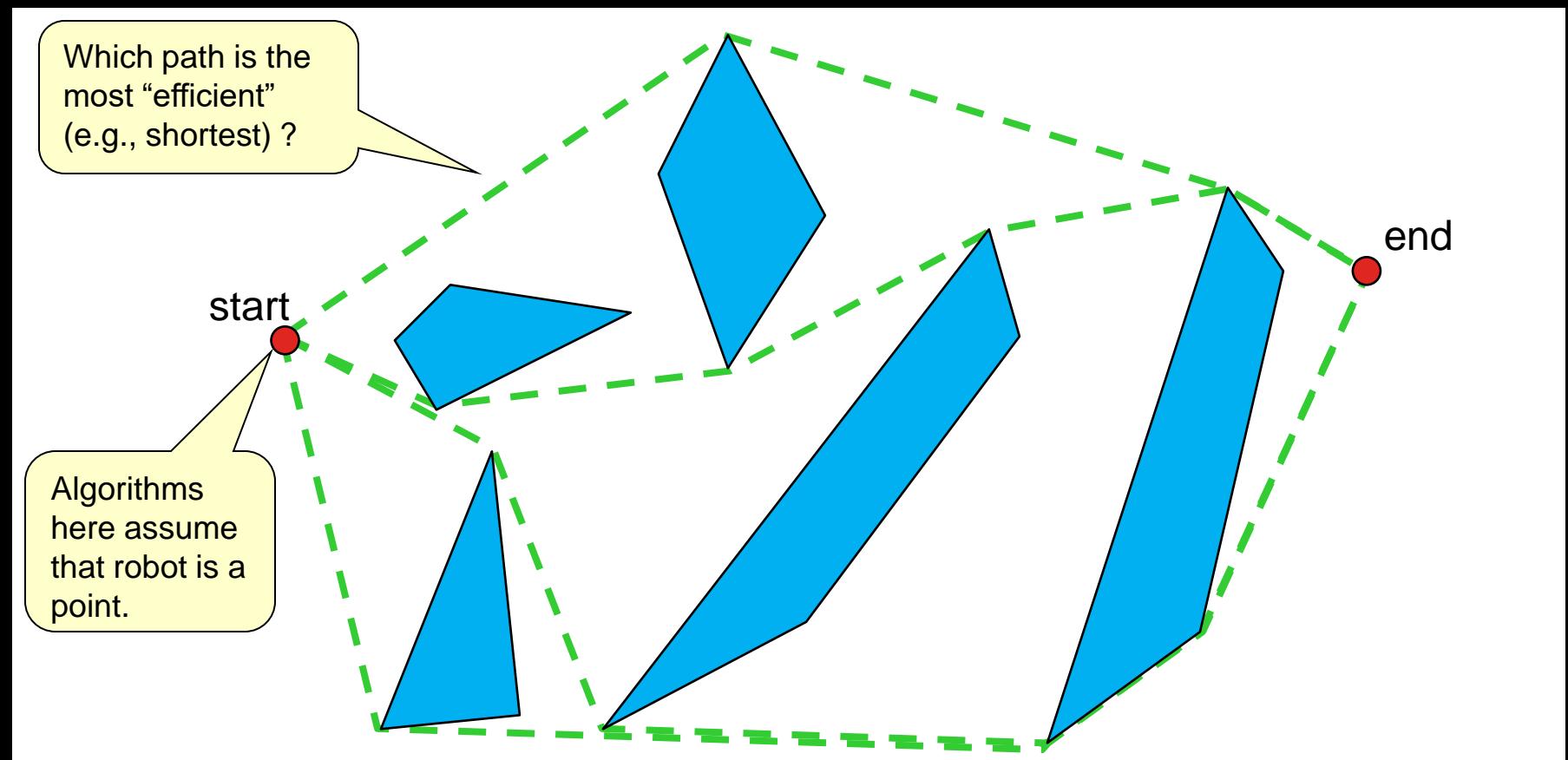
Increased Line-Fit Tolerance (i.e., Error Threshold)

Start the  
Lab ...

# Path Planning

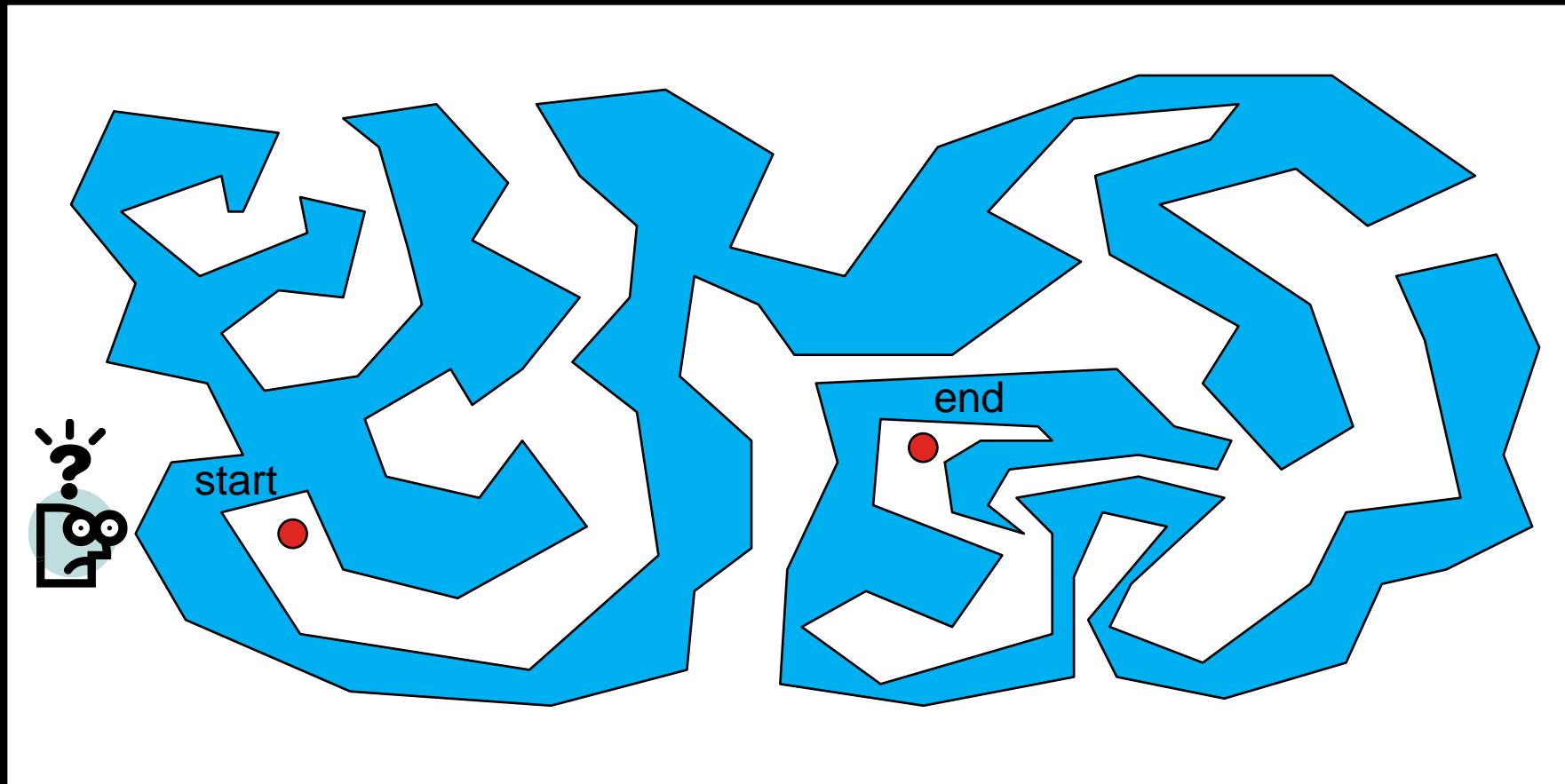
# Path Planning – Convex Obstacles

- How do we get a robot to plan a path around objects efficiently from one location to another ?



# Path Planning – Non-Convex Obst.

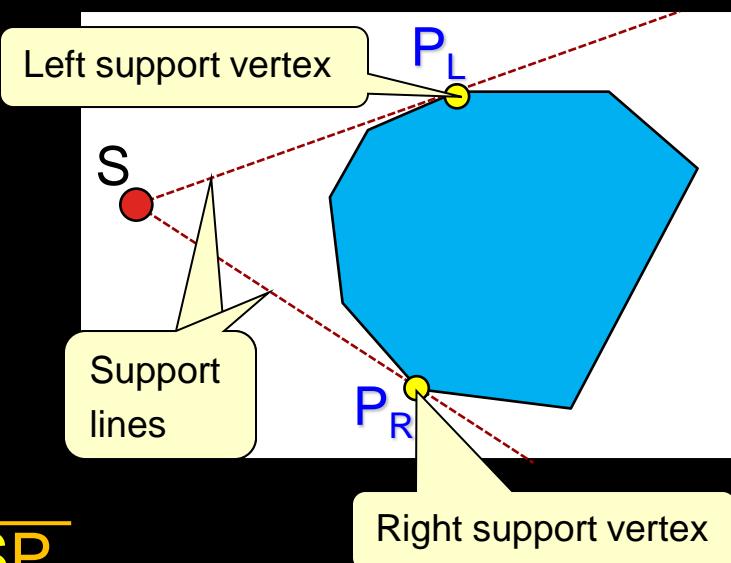
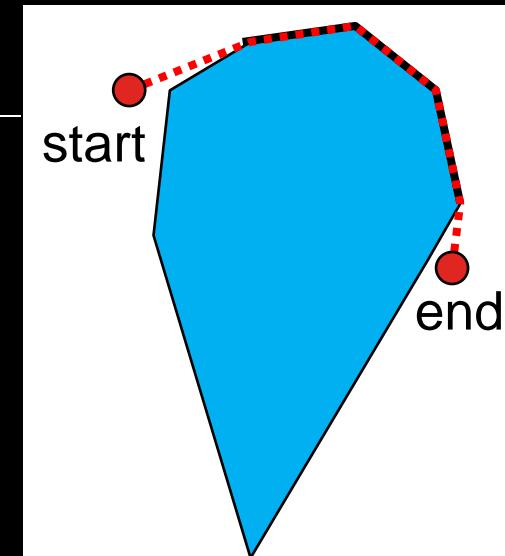
- Solution is not as obvious with non-convex obstacles. We will consider this in another lab.



# Shortest Paths

- Shortest path actually travels around obstacles, “hugging” the boundary.
- If an obstacle is in the way, robot will go around it by heading towards the left or right *support vertices*:

- most “extreme” vertices of obstacle with respect to some point,  $S$ .
- like “grab points” for picking up obstacle with two arms.
- obstacle always lies completely on one side of support lines  $\overline{SP_L}$  and  $\overline{SP_R}$ .



# Shortest Path Properties

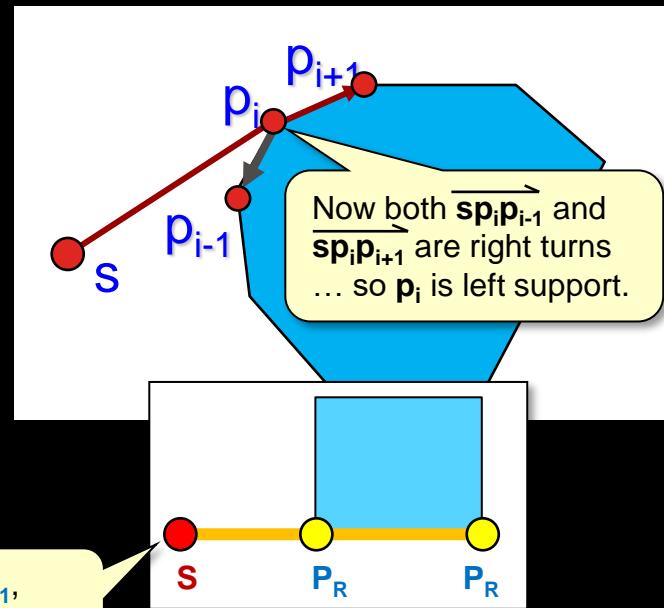
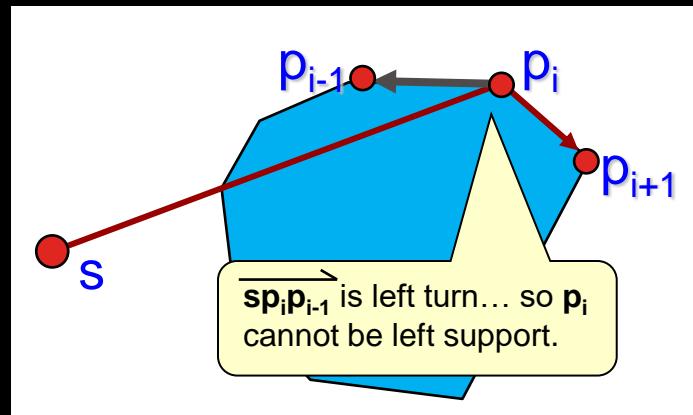
- Can find  $P_L$  and  $P_R$  by checking each vertex using a “left/right turn test” ... for convex polygons:

$P_L = p_i$  if and only if both  $\overrightarrow{sp_ip_{i-1}}$  and  $\overrightarrow{sp_ip_{i+1}}$  are right turns.

$P_R = p_i$  if and only if both  $\overrightarrow{sp_ip_{i-1}}$  and  $\overrightarrow{sp_ip_{i+1}}$  are left turns.

- Check all polygon vertices:

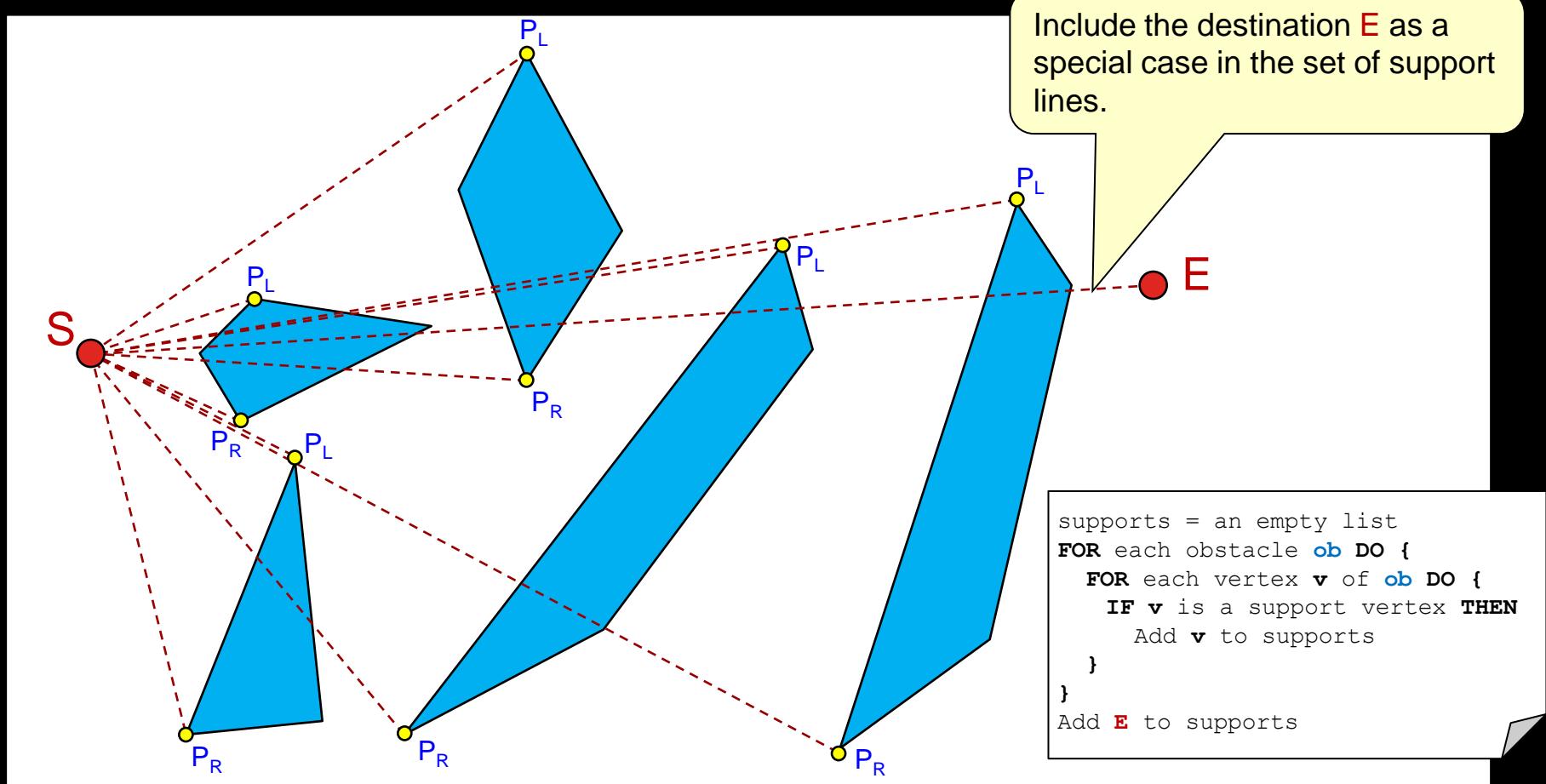
```
s = (xs, ys) is the source point
FOR (each vertex pi = (xi, yi) of the polygon) {
    pi+1 = (xi+1, yi+1) // polygon vertex after pi
    pi-1 = (xi-1, yi-1) // polygon vertex before pi
    t1 = (xi-xs) * (yi+1-ys) - (yi-ys) * (xi+1-xs)
    t2 = (xi-xs) * (yi-1-ys) - (yi-ys) * (xi-1-xs)
    IF ((t1 ≤ 0) AND (t2 ≤ 0)) THEN
        pi is the left support vertex, so add it
    IF ((t1 ≥ 0) AND (t2 ≥ 0)) THEN
        pi is the right support vertex, so add it
}
```



When  $S$  is collinear to  $p_i$  and one of  $p_{i+1}$  or  $p_{i-1}$ , there could be two supports on the same side !!

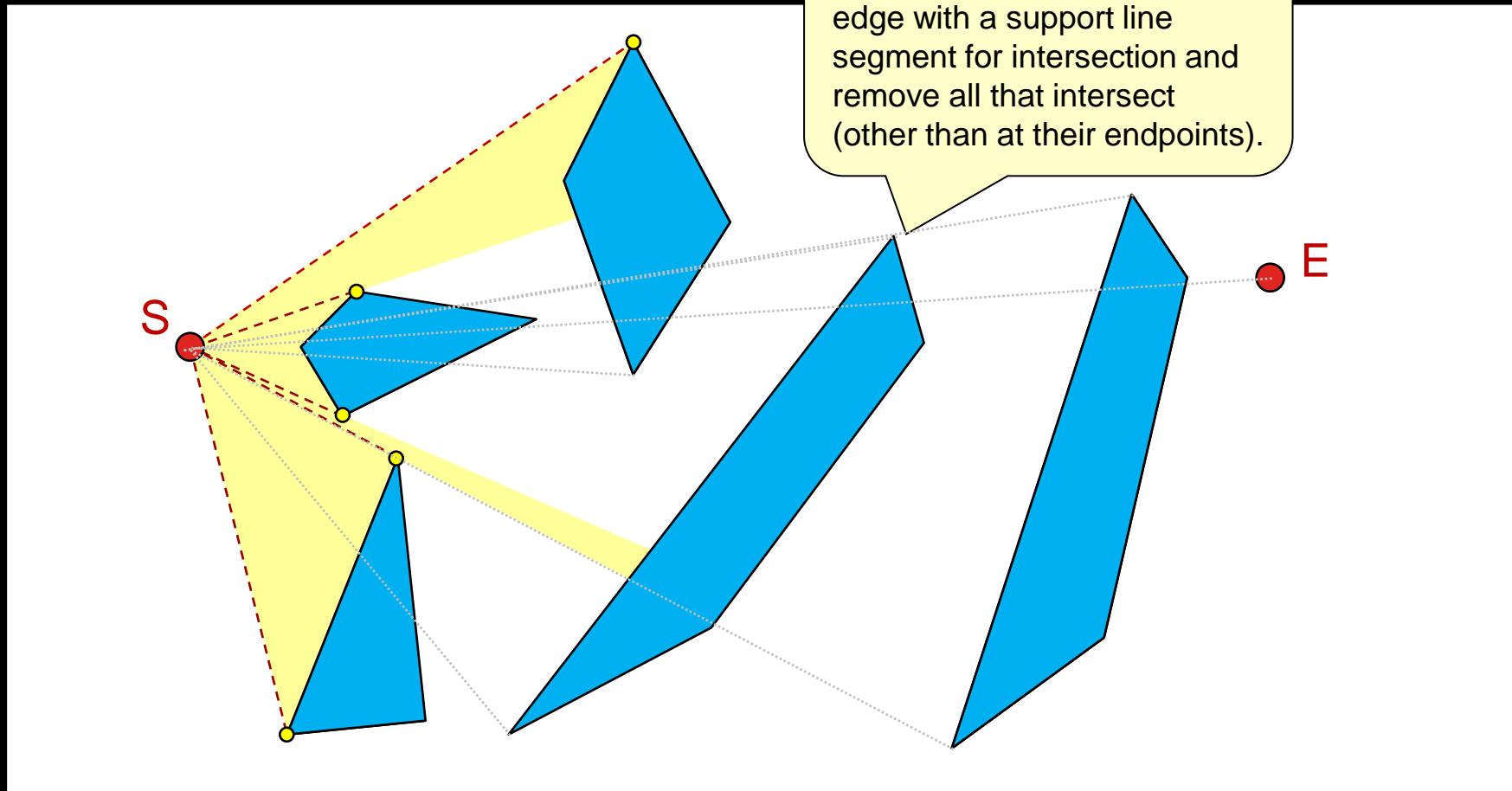
# Finding All Support Vertices

- The first step towards computing a shortest path is to find all obstacle support vertices:



# Visible Support Vertices

- Then eliminate any support vertices that are not visible from S:



# Eliminating Support Vertices

- Just need to add an IF statement before adding:

```
supports = an empty list
FOR each obstacle ob DO {
    FOR each vertex v of ob DO {
        IF v is a support vertex THEN {
            IF SupportLineIntersectsObstacle(s, v, obstacles) is true THEN
                Add v to supports
        }
    }
}
IF SupportLineIntersectsObstacle(s, E, obstacles) is false THEN
    Add E to supports
```

**S → v** is a support line

**S → E** is a support line

Add this function call

Add this function call

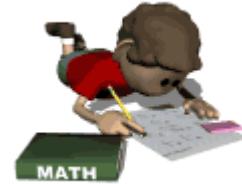
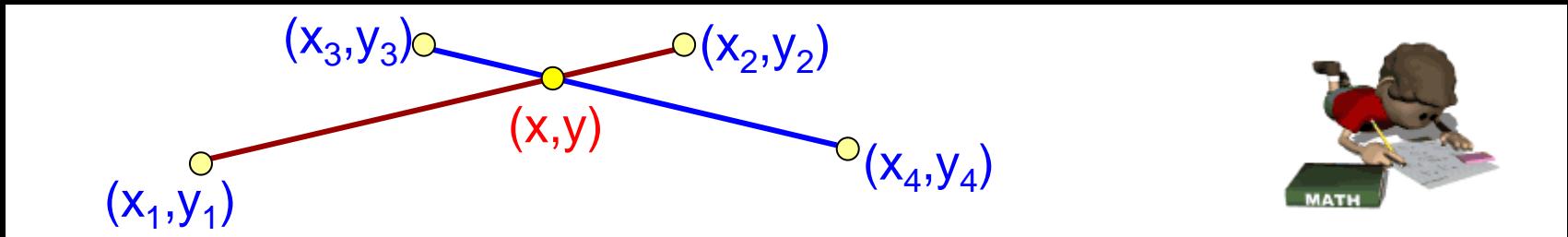
- Function checks each support line with all obstacles:

```
SupportLineIntersectsObstacle(s, supportPoint, obstacles) {

    FOR each obstacle ob of obstacles DO {
        FOR each vertex v of ob DO {
            va = vertex of ob after v
            IF support line from s to supportPoint intersects obstacle edge v → va THEN
                RETURN true
        }
    }
    RETURN false
}
```

# Line Intersection test

- How do we check for line-segment intersection ?



- Can use well-known equation of a line:

$$y = m_a x + b_a$$

$$y = m_b x + b_b$$

where

$$m_a = (y_2 - y_1) / (x_2 - x_1)$$

$$m_b = (y_4 - y_3) / (x_4 - x_3)$$

$$b_a = y_1 - x_1 m_a$$

$$b_b = y_3 - x_3 m_b$$

Must handle special case where lines are vertical.  
(i.e.,  $x_1 == x_2$  or  $x_3 == x_4$ )

- Intersection occurs when these are equal:

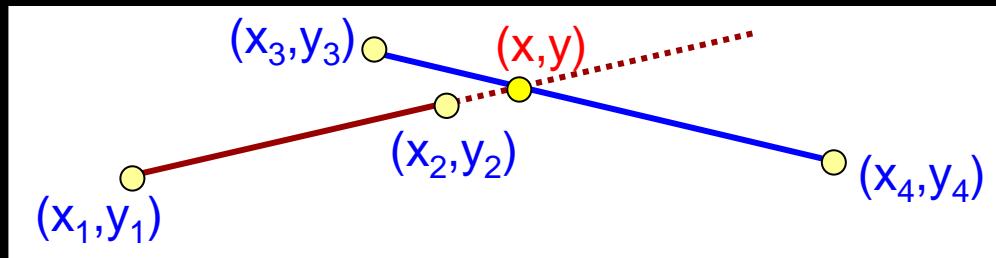
$$m_a x + b_a = m_b x + b_b \rightarrow x = (b_b - b_a) / (m_a - m_b)$$

If  $(m_a == m_b)$  the lines are parallel and there is no intersection

# Line Intersection test

- Final test is to ensure that intersection ( $x, y$ ) lies on line segment ... just make sure that each of these is true:

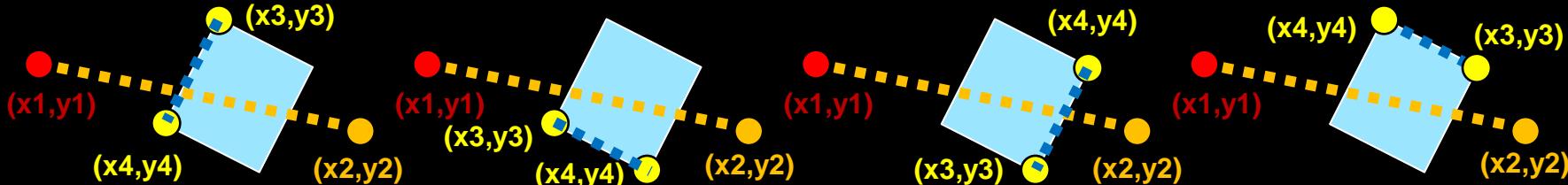
- $\max(x_1, x_2) \geq x \geq \min(x_1, x_2)$
- $\max(x_3, x_4) \geq x \geq \min(x_3, x_4)$



- In java, we have a nice function to do all this for us:

```
java.awt.geom.Line2D.Double.linesIntersect(x1,y1,x2,y2,x3,y3,x4,y4)
```

- You will be checking intersection of a support line with each edge of an obstacle:

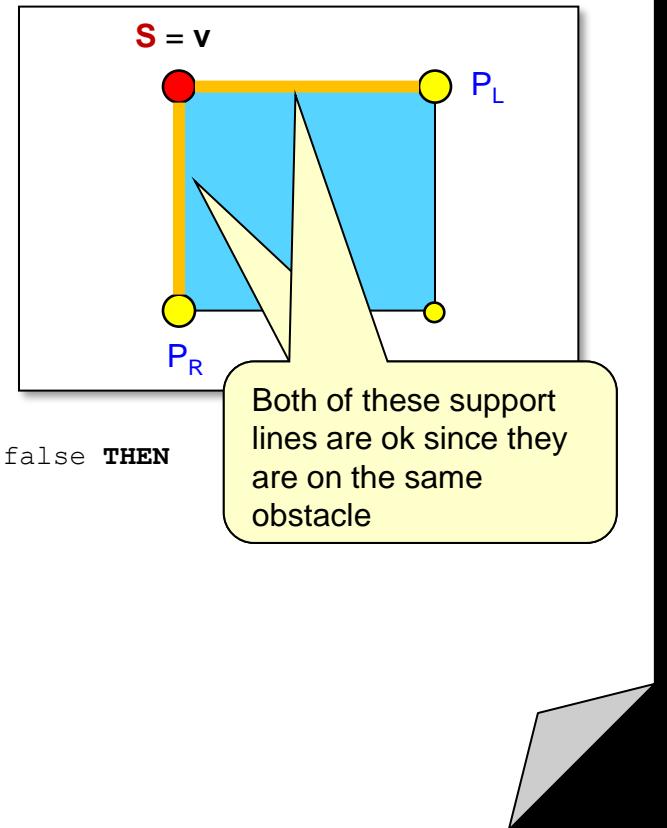


# Handling Special Cases: 1

```
supports = an empty list

FOR each obstacle ob DO {
    FOR each vertex v of ob DO {
        IF S has the same coordinates as v THEN {
            Add vertex of ob before v to supports
            Add vertex of ob after v to supports
        }
    }
    OTHERWISE {
        IF v is a support vertex THEN {
            IF SupportLineIntersectsObstacle(S, v, obstacles) is false THEN
                Add v to supports
            }
        }
    }
}

IF SupportLineIntersectsObstacle(S, E, obstacles) is false THEN
    Add E to supports
```



# Handling Special Cases: 2

```
SupportLineIntersectsObstacle(S, supportPoint, obstacles) {
```

```
    FOR each obstacle ob of obstacles DO {
```

```
        FOR each vertex v of ob DO {
```

```
            va = vertex of ob after v
```

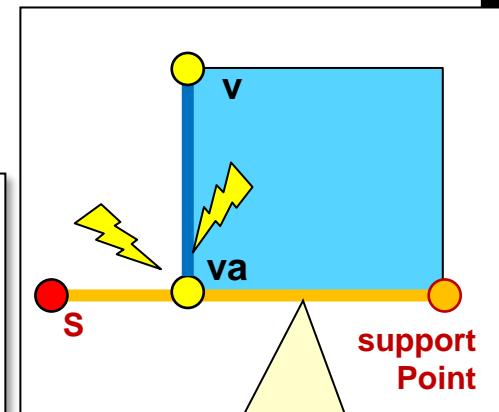
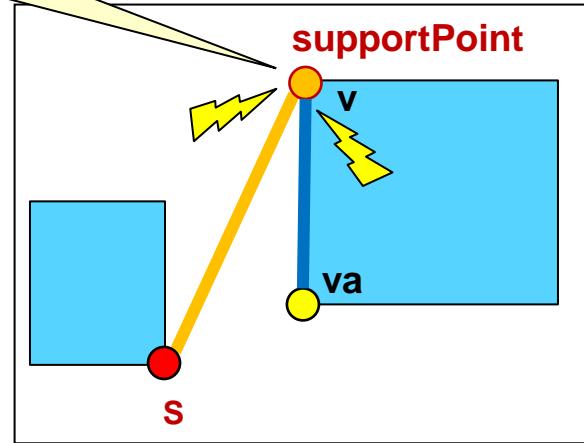
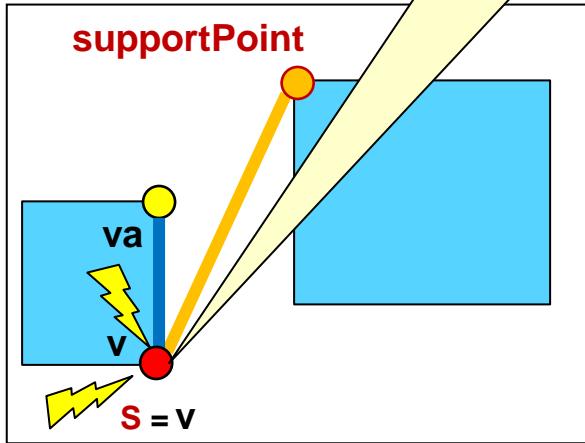
```
            IF [(support line from S to supportPoint intersects obstacle edge v → va) AND  
                  (S is not the same coordinate as v or va) AND  
                  (supportPoint is not the same coordinate as v or va)] THEN {
```

```
                RETURN true
```

```
}
```

```
}  
RETURN false
```

By definition, the support line intersects this obstacle at the vertex. But this is ok.

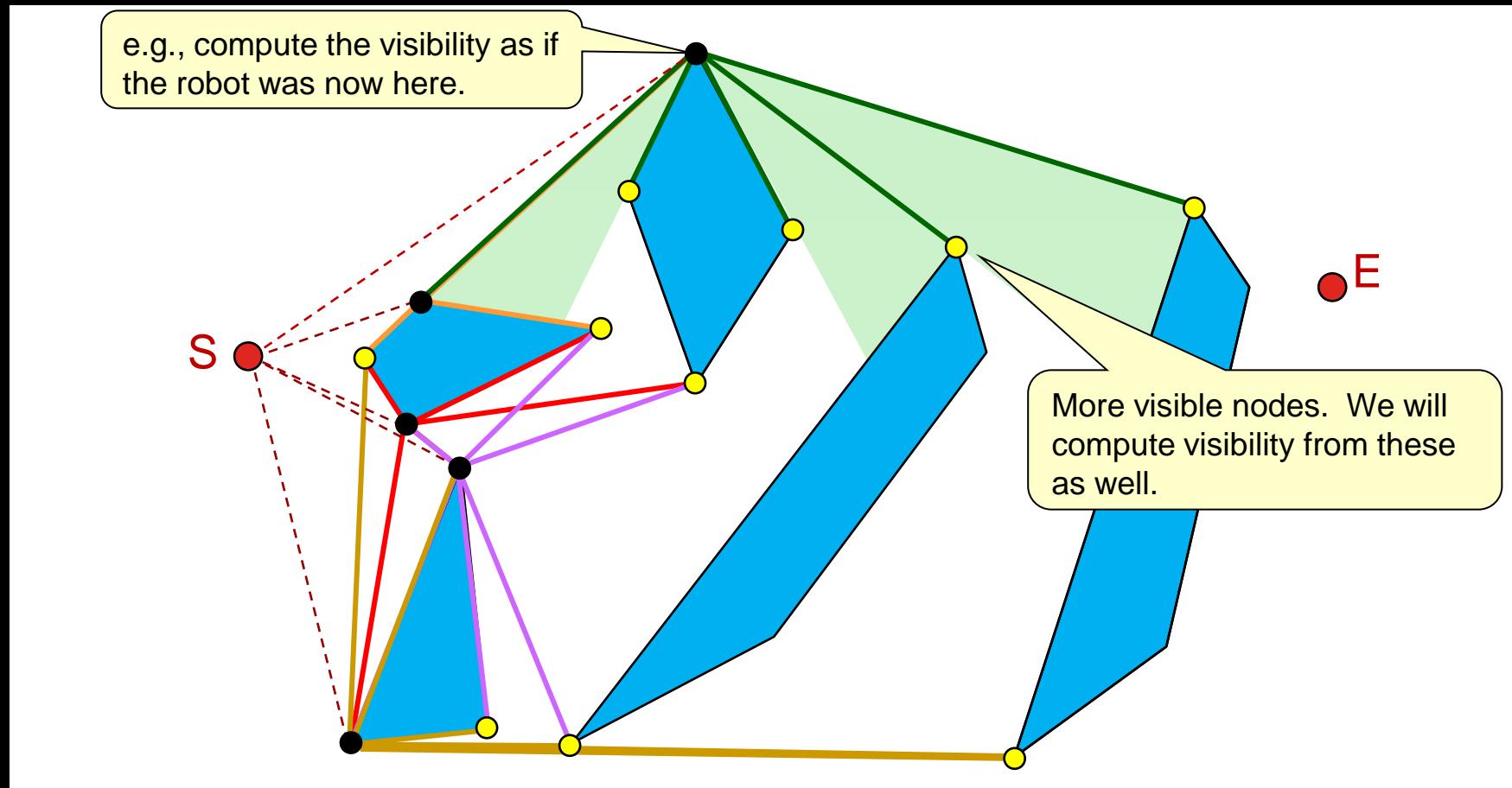


This support line running parallel to the obstacle edge will intersect and will be removed.

Be careful with your logic here. Many students get this wrong.

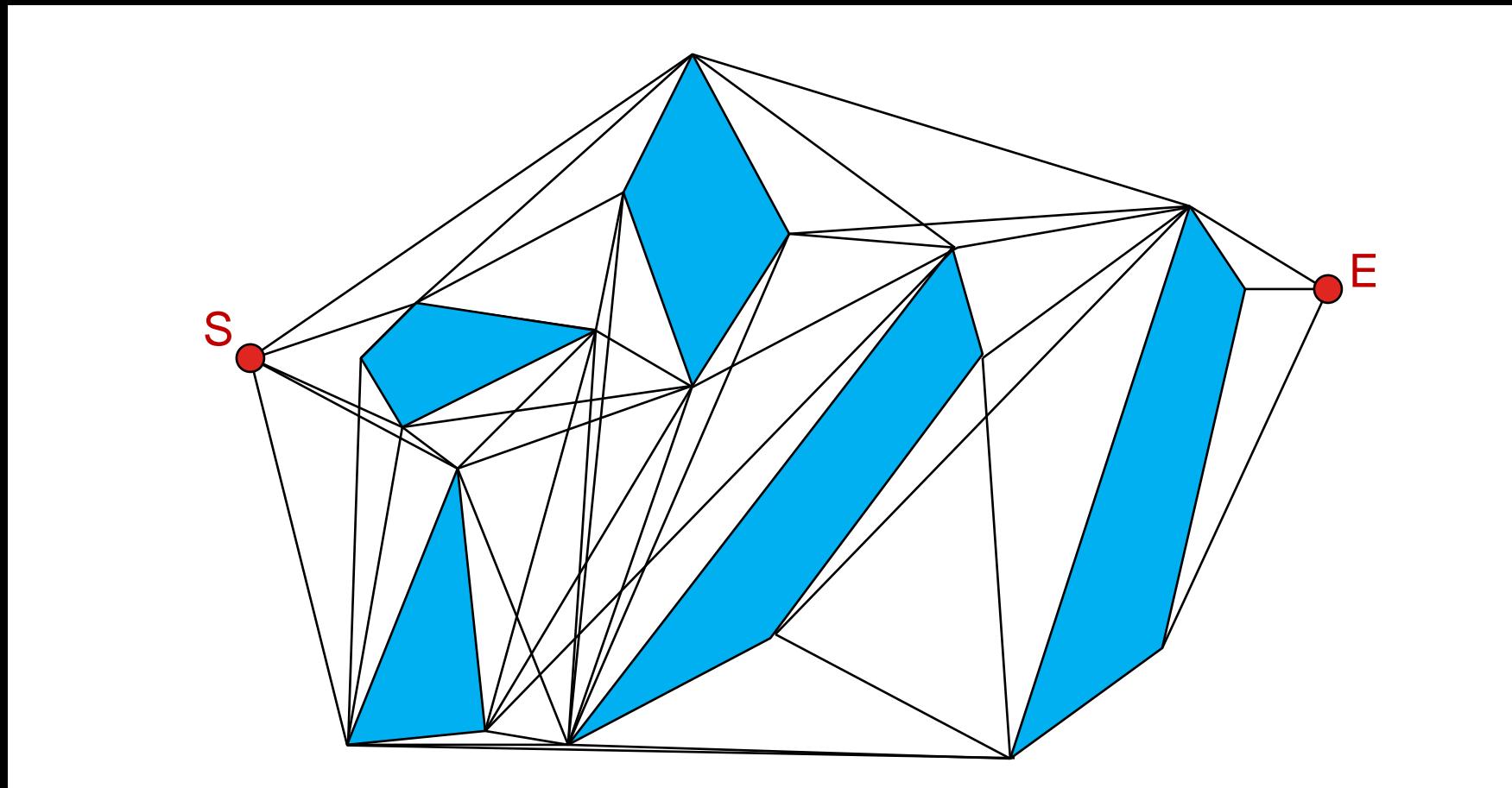
# Iterating Through Support Vertices

- We will now repeat this process from each obstacle vertex (as if robot traveled to those vertices):



# The Visibility Graph

- By appending all these visible segments together, a **visibility graph** is obtained:



# The Pseudocode

```
computeVisibilityGraph() {  
    graph = an empty graph
```

Add **S** as a Node of the graph  
Add **E** as a Node of the graph

**S** and **E** are the start and end points of our environment

These are the obstacles of our environment

```
    FOR each obstacle ob of obstacles DO {  
        FOR each vertex v of ob DO {  
            IF v is not already a Node in the graph THEN  
                Add v as a Node in the graph  
        }  
    }
```

```
    FOR each Node n of the graph DO {
```

Find all visible support points from **n**

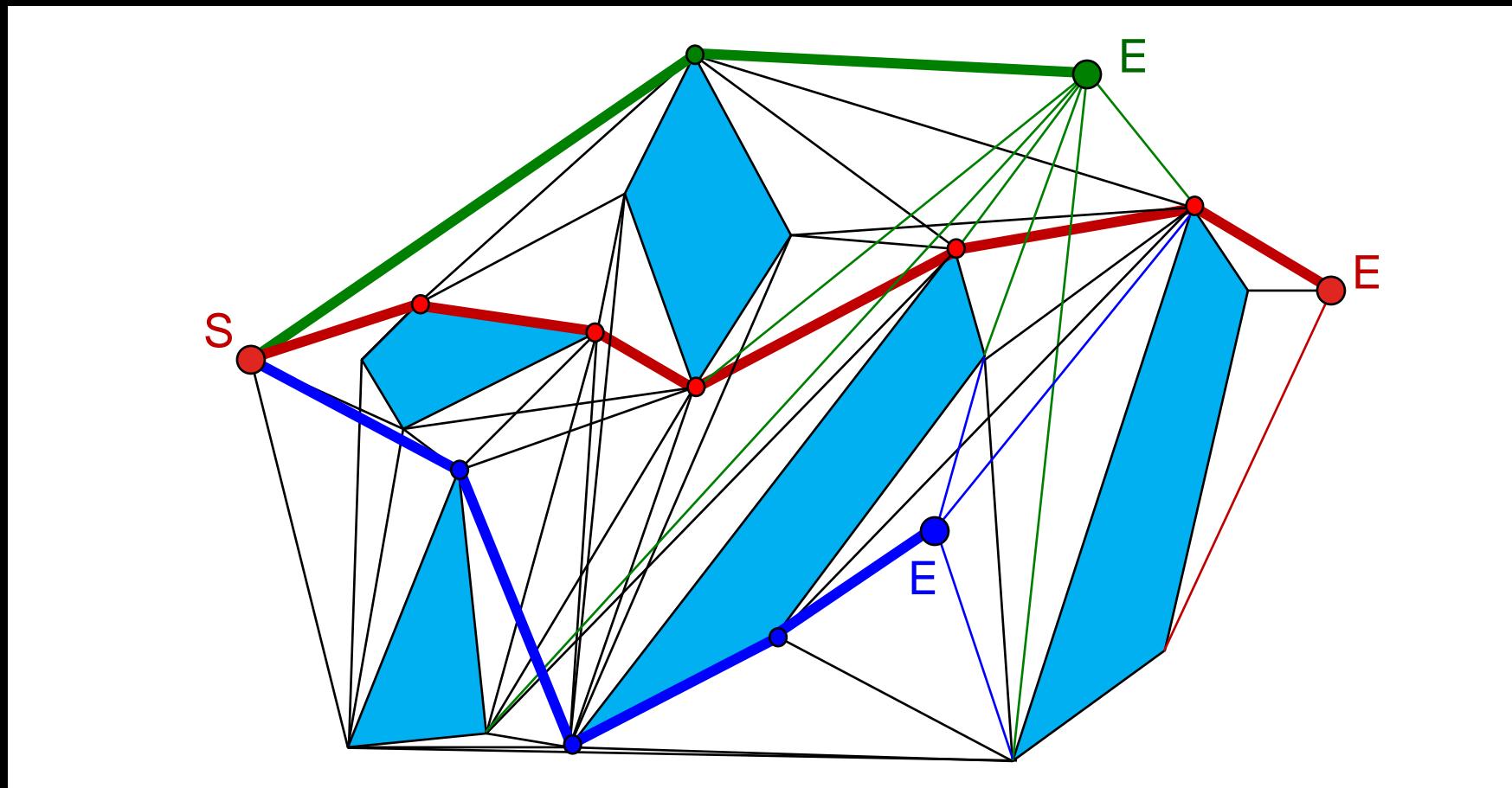
This is all our hard work from before

```
        FOR each visible support point p that we found DO {  
            m = find the node at point p in the graph  
  
            IF ((m was found) AND (n!=m)) THEN  
                Add an Edge in the graph from Node n to Node m  
        }  
    }
```

Don't check coordinate values here  
... just make sure that **n** is not the  
same identical node as **m**.

# Visibility Graph Paths

- Shortest paths from the **start** to the **end** location will always travel along visibility graph edges:

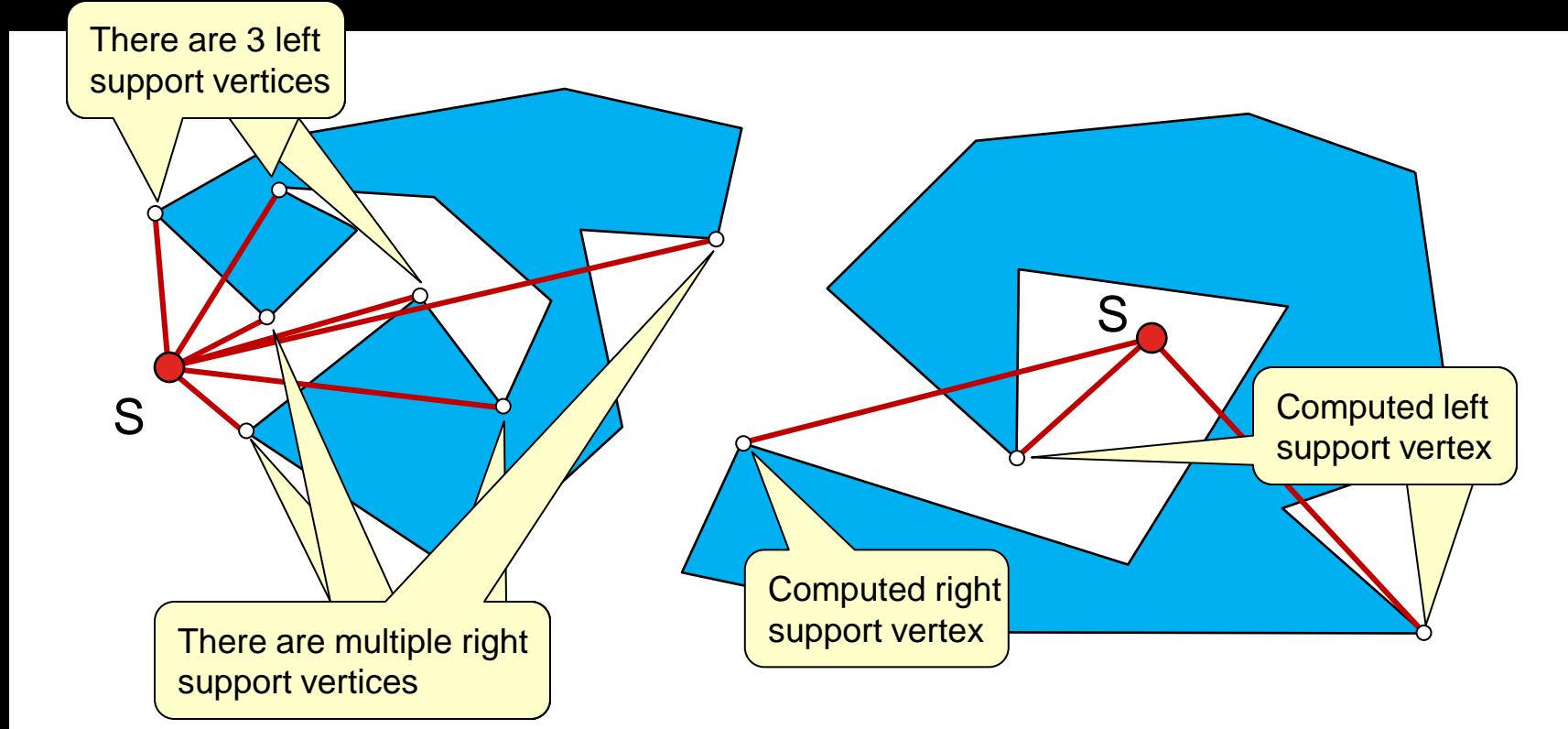


Start the  
Lab ...

# Path Planning (Non-Convex Obstacles)

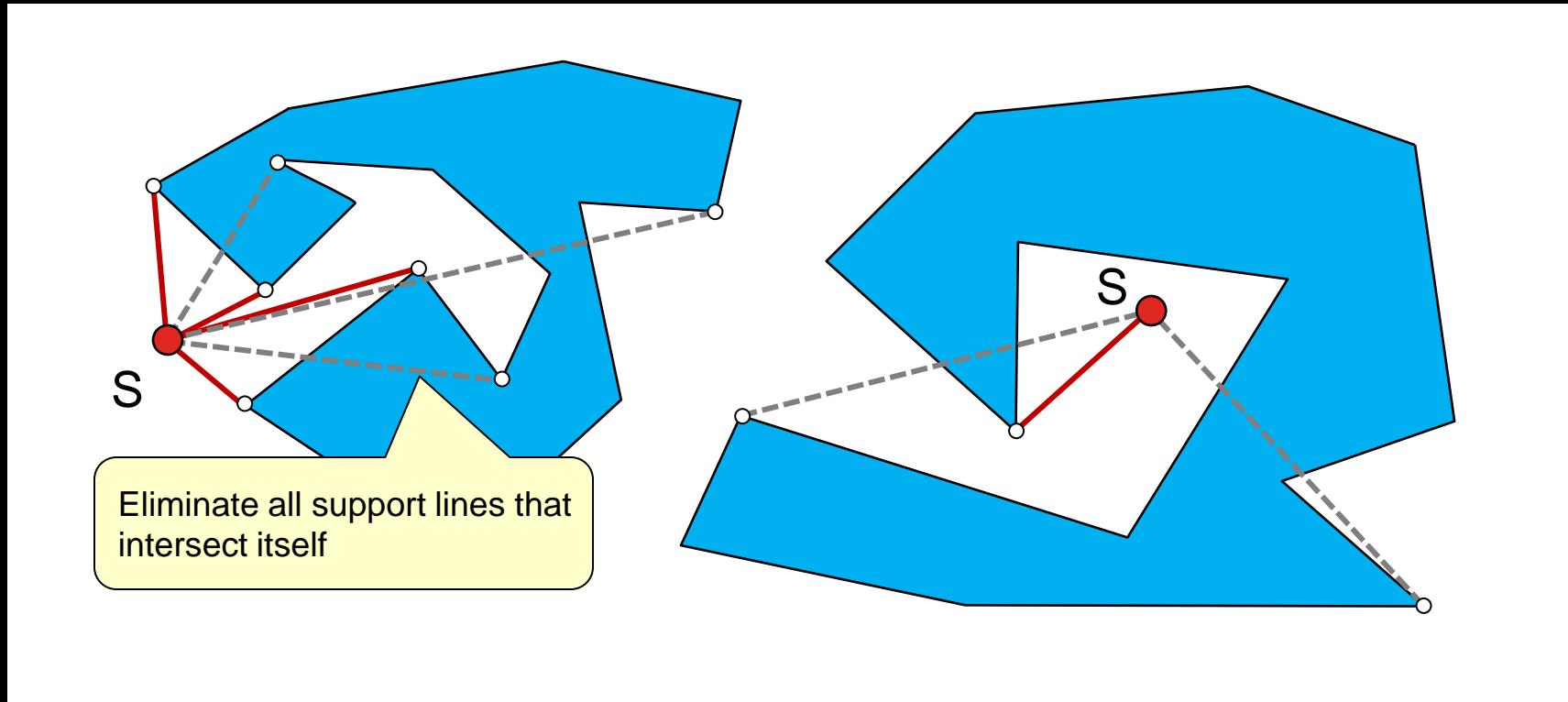
# Non-Convex Obstacle Supports

- Our support-line algorithm can produce multiple support lines per polygon if it is non-convex:



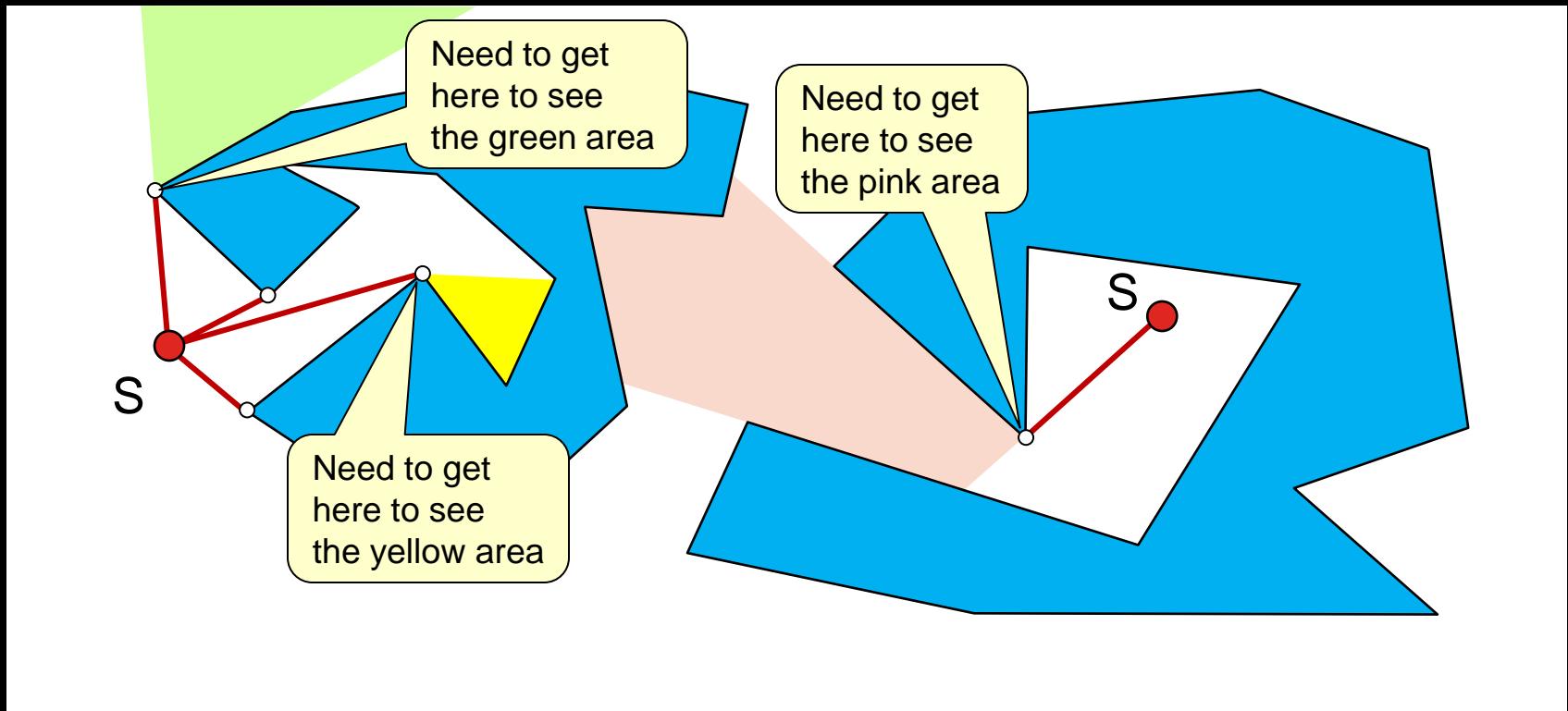
# Non-Convex Obstacle Supports

- We can eliminate those that intersect itself as well as those that intersect other polygons.



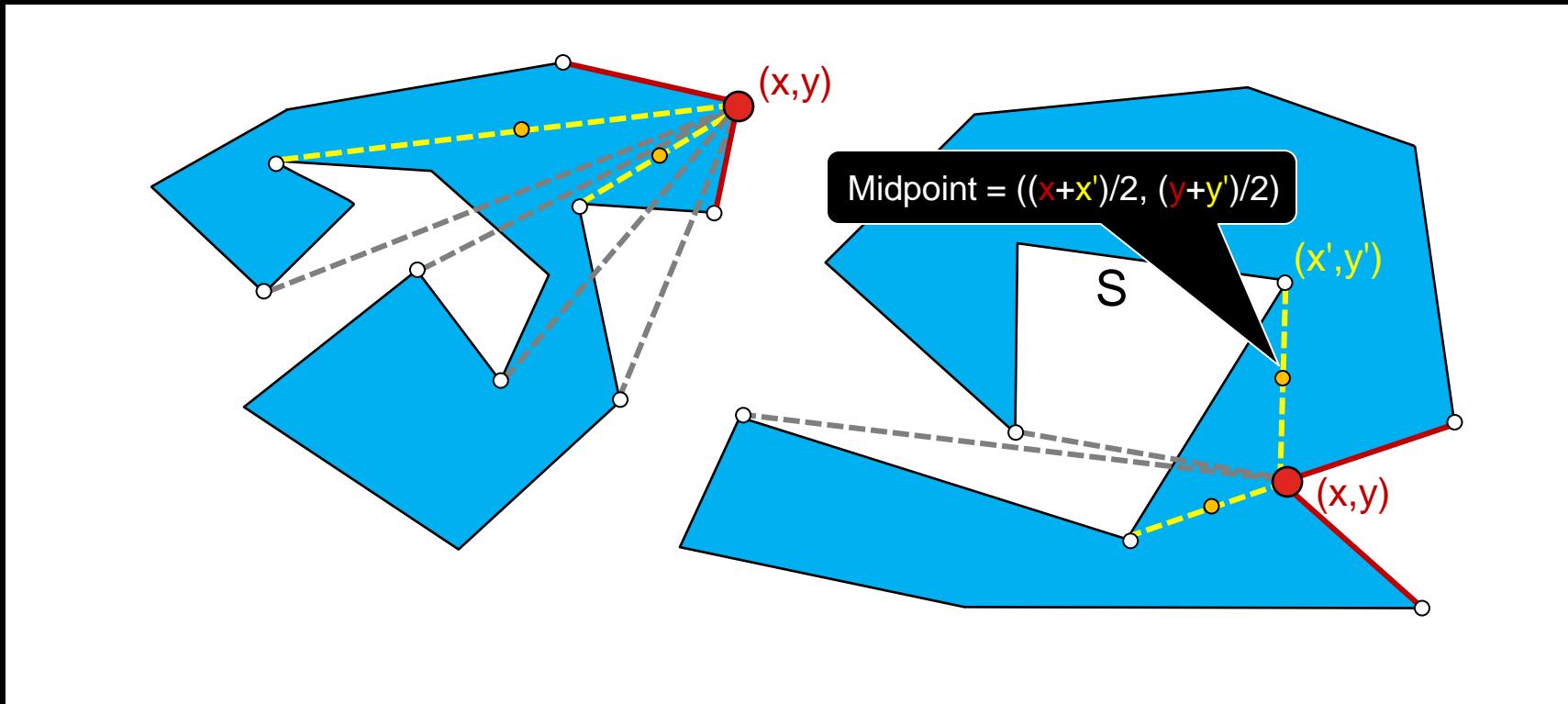
# Non-Convex Visibility

- A support vertex represents an “observation point” that robot must be at in order to “see” (i.e., have visibility) around a corner. They are all necessary.



# Non-Convex Problems

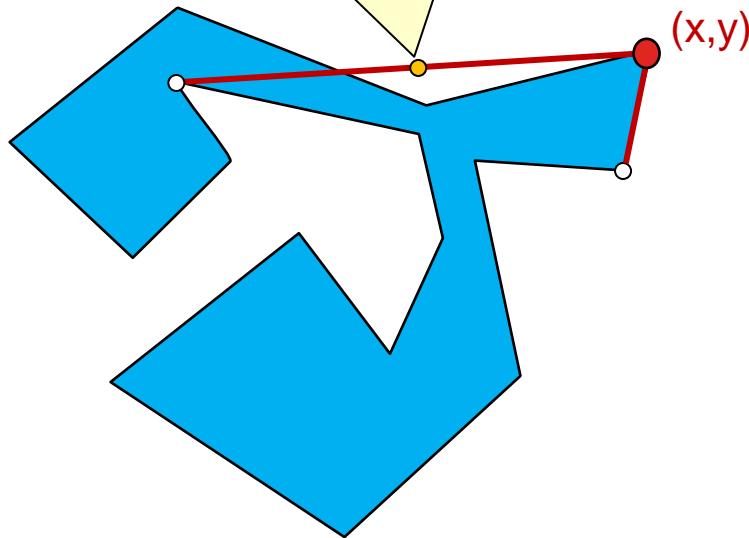
- Some “inner” support lines intersect edges only at vertices (see yellow below) and these are invalid.
- Perhaps we could check if midpoint of line lies within obstacle



# Non-Convex Problems

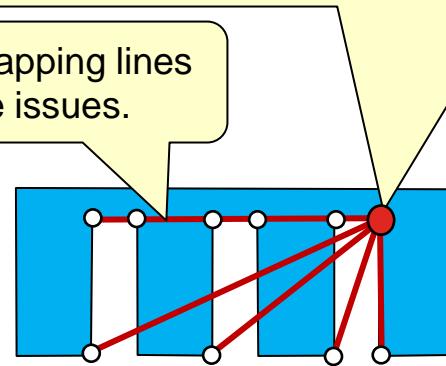
- But there are cases where this will not work:

Midpoint of this inner support line lies outside of the polygon, so it cannot be detected as invalid.



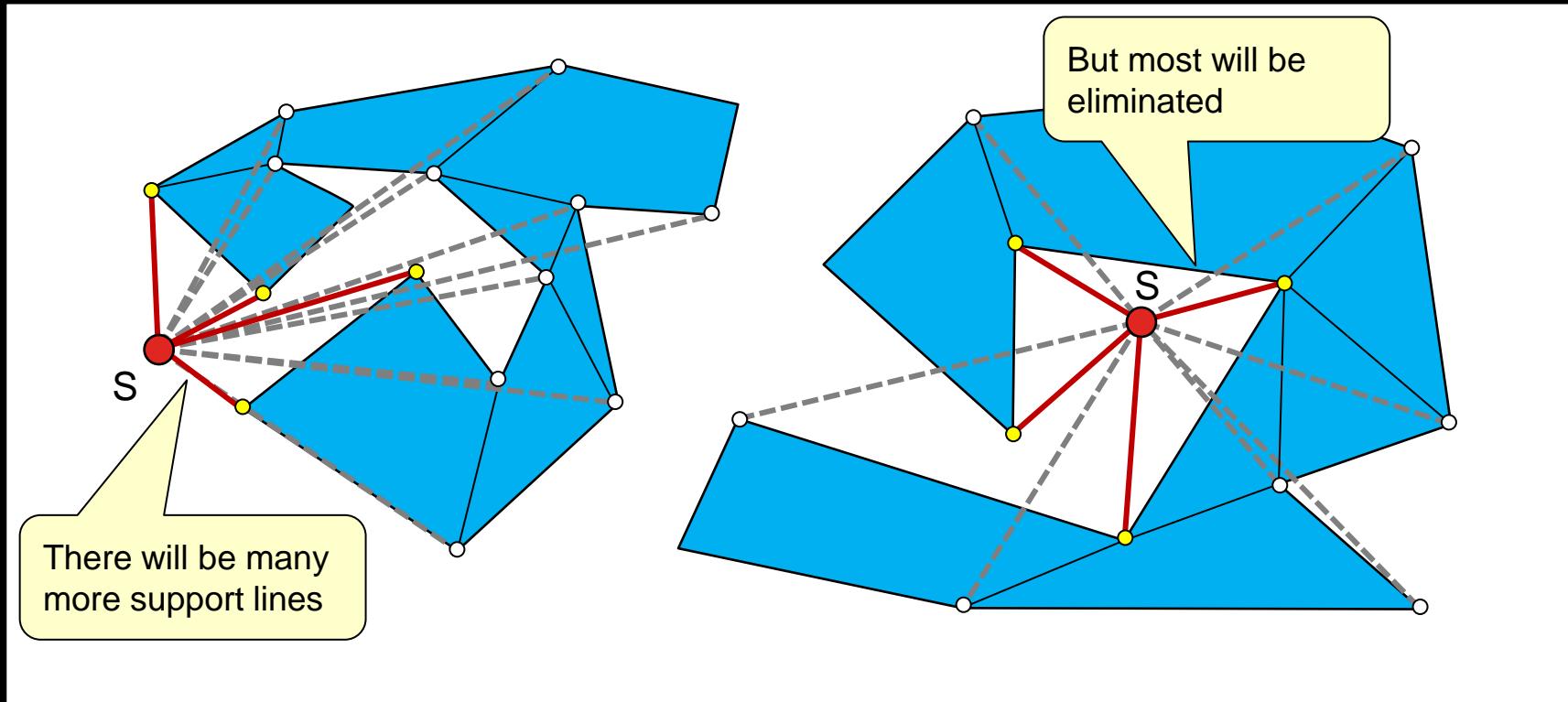
From this vantage point, there are multiple support lines ... many of them overlap due to the collinearity.

Overlapping lines cause issues.



# Non-Convex Easy Solution

- Easiest solution is to break up the non-convex polygons into convex pieces. Then the algorithm from before should work.

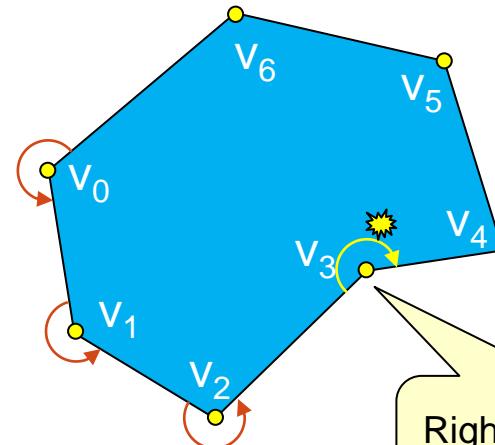
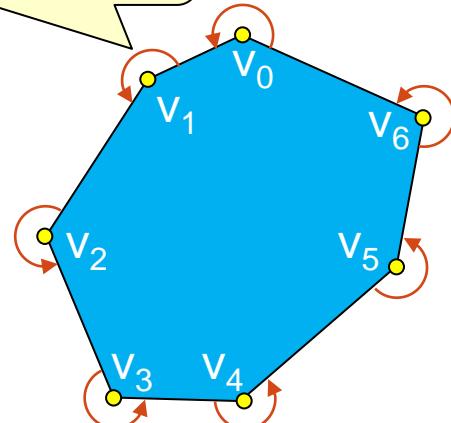


# Convexity Checking

- How do we know if a polygon is convex or not ?
  - traverse vertices CCW ... all must make **left** turns:

All **left** turns indicates that polygon is convex.

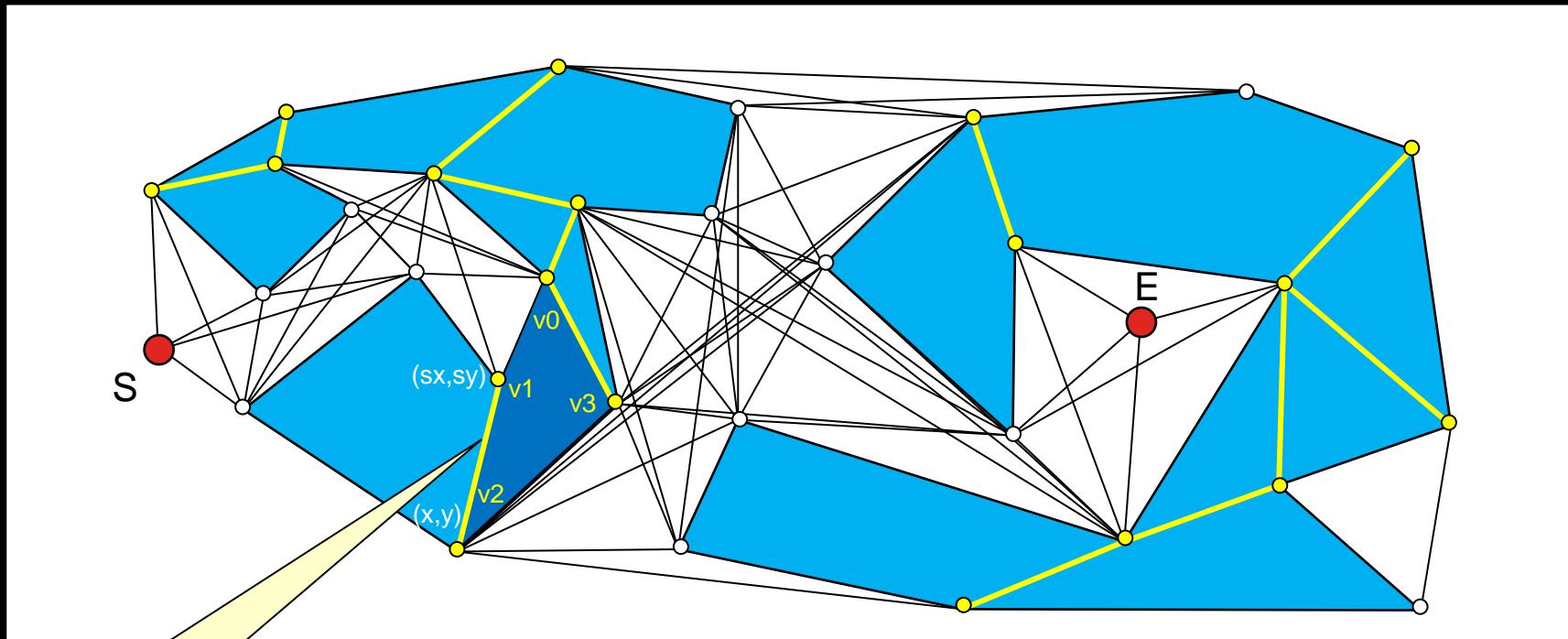
$v_{i-1} \rightarrow v_i \rightarrow v_{i+1}$  is a left turn if  
 $((x_i - x_{i-1}) * (y_{i+1} - y_{i-1}) - (y_i - y_{i-1}) * (x_{i+1} - x_{i-1})) > 0$



Right Turn! Polygon is not convex.

# Non-Convex Easy Solution

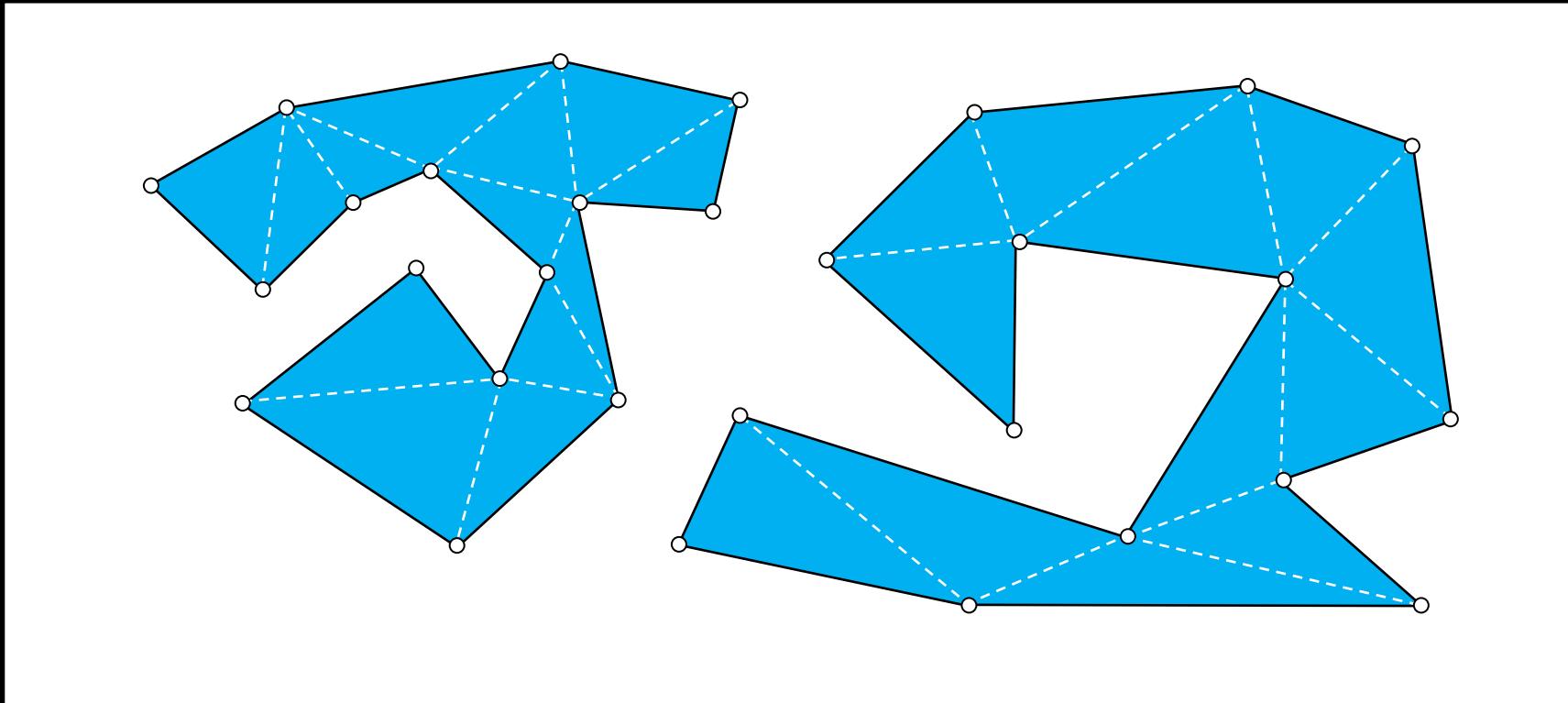
- There will be special cases for shared edges between convex pieces that need to be eliminated when computing the visibility graph:



Must make sure that none of these (yellow) *interior bordering support lines* are in the graph. Just need to check if support line  $(x,y) \rightarrow (sx,sy)$  has same vertex coordinates as edge  $(v_1, v_2) \dots$  or edge  $(v_2, v_1)$ .

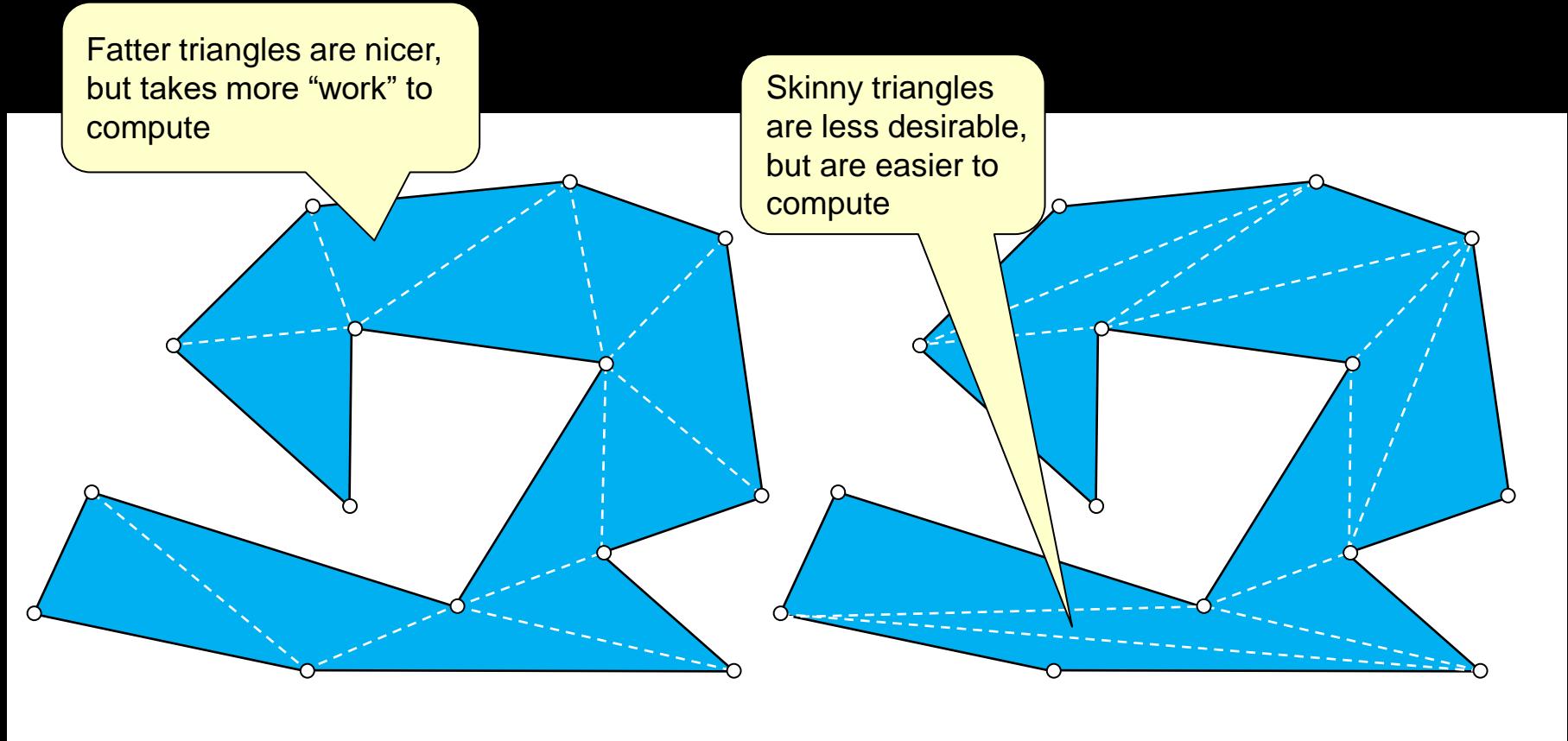
# Convert Non-Convex To Convex

- How do we break a polygon into convex pieces ?
  - There are many algorithms. The most popular is to break into triangles. This is known as *triangulation*.



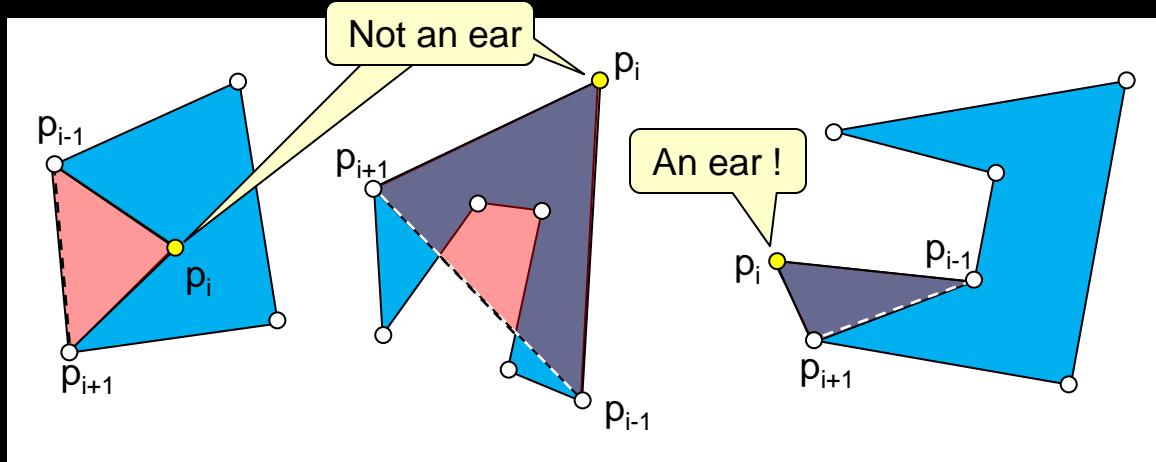
# Triangulation

- How do we triangulate a polygon ?
  - There are many ways/algorithms to do this as well.



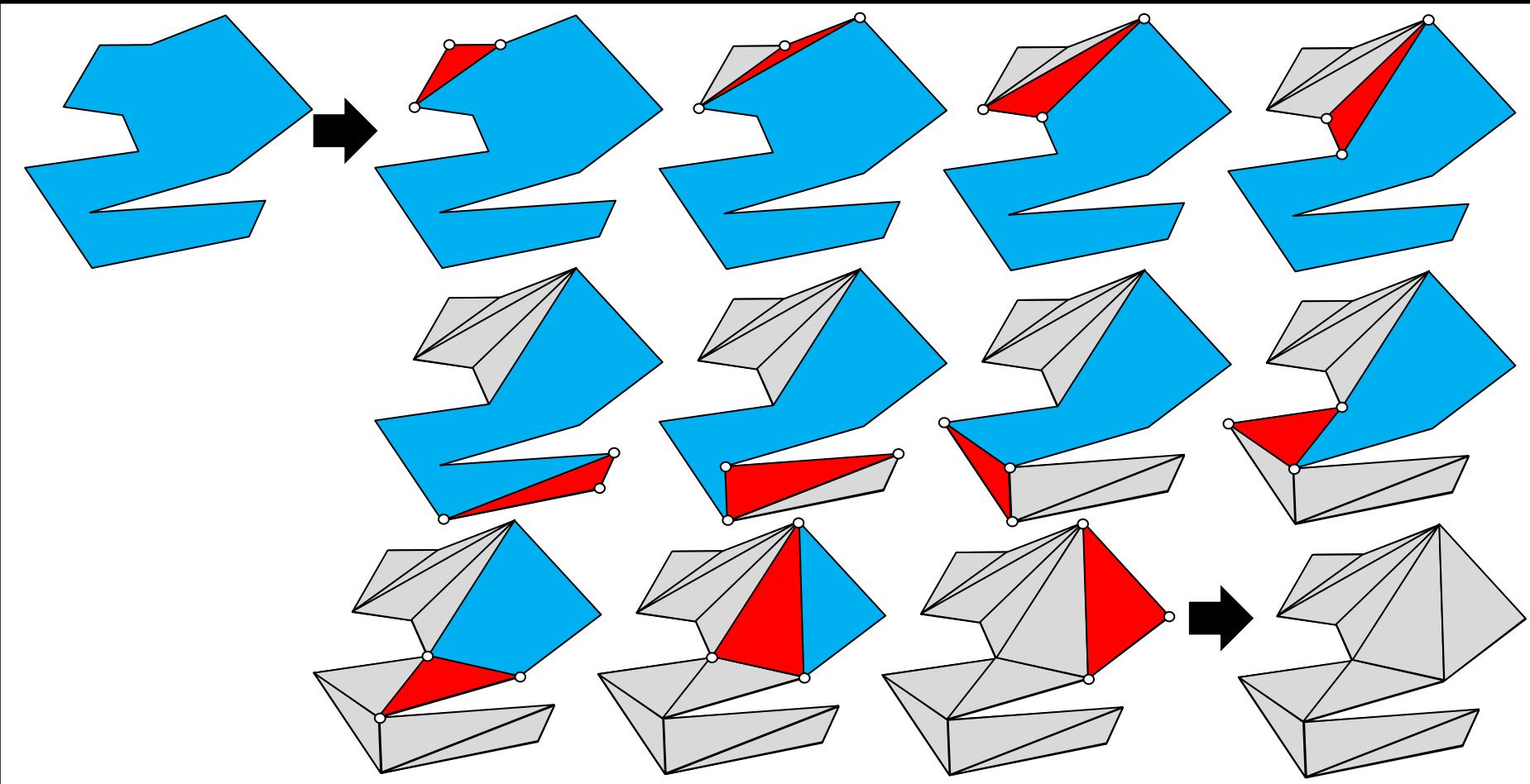
# Ear-Cutting Triangulation

- We will use an “ear-cutting” algorithm since it is easy.
  - Idea is to repeatedly cut off “ears” of the polygon, making a set of triangles along the way.
  - An “ear” is a vertex  $p_i$  such that Line segment  $\overline{p_{i-1}p_{i+1}} \dots$ 
    1. intersects the polygon boundary only at points  $p_{i-1}$  and  $p_{i+1}$  AND...
    2. it lies entirely inside the polygon.



# Ear-Cutting Triangulation

- Repeatedly cut off “ears” of the polygon, making a set of triangles along the way.



# Ear-Cutting Algorithm

```
1 function EarCut(Obstacle P)
2     triangles = an empty list
3     ear = null;
4     if (P has only 3 vertices) then
5         Add P to triangles and return triangles
6     Copy all vertices of P into a list of points Q
7     while (Q has more than 3 points) do
8         earIndex = -1
9         for each point pi in Q do
10            if (pi-1pipi+1 is a left turn) then
11                ear = a new obstacle with vertices pi-1pi and pi+1
12                earIndex = i
13                for each point pk in Q (such that k≠i-1, k≠i, k≠i+1) do
14                    if (point pk lies inside or on boundary of the ear) then
15                        earIndex = -1
16                    if (earIndex != -1) then
17                        Break out of the FOR loop at line 8 ... we found an ear
18                    if (earIndex == -1) quit and return triangles, since no more ears were found
19                    Remove pearIndex from Q
20                    Add ear to triangles
21                    Add (a new obstacle with vertices p0, p1 and p2) to triangles
22    return triangles
```

Nothing to do if only 3 vertices

Copy obstacle vertices into a new list  
that we can add/remove from without  
destroying the original obstacle.

Find  
an  
ear

Left turn if CCW ordering and  
 $((x_i - x_{i-1})(y_{i+1} - y_{i-1}) - (y_i - y_{i-1})(x_{i+1} - x_{i-1})) > 0$

reject p<sub>i</sub> as an ear since it is invalid

Cut the ear off and add to the solution

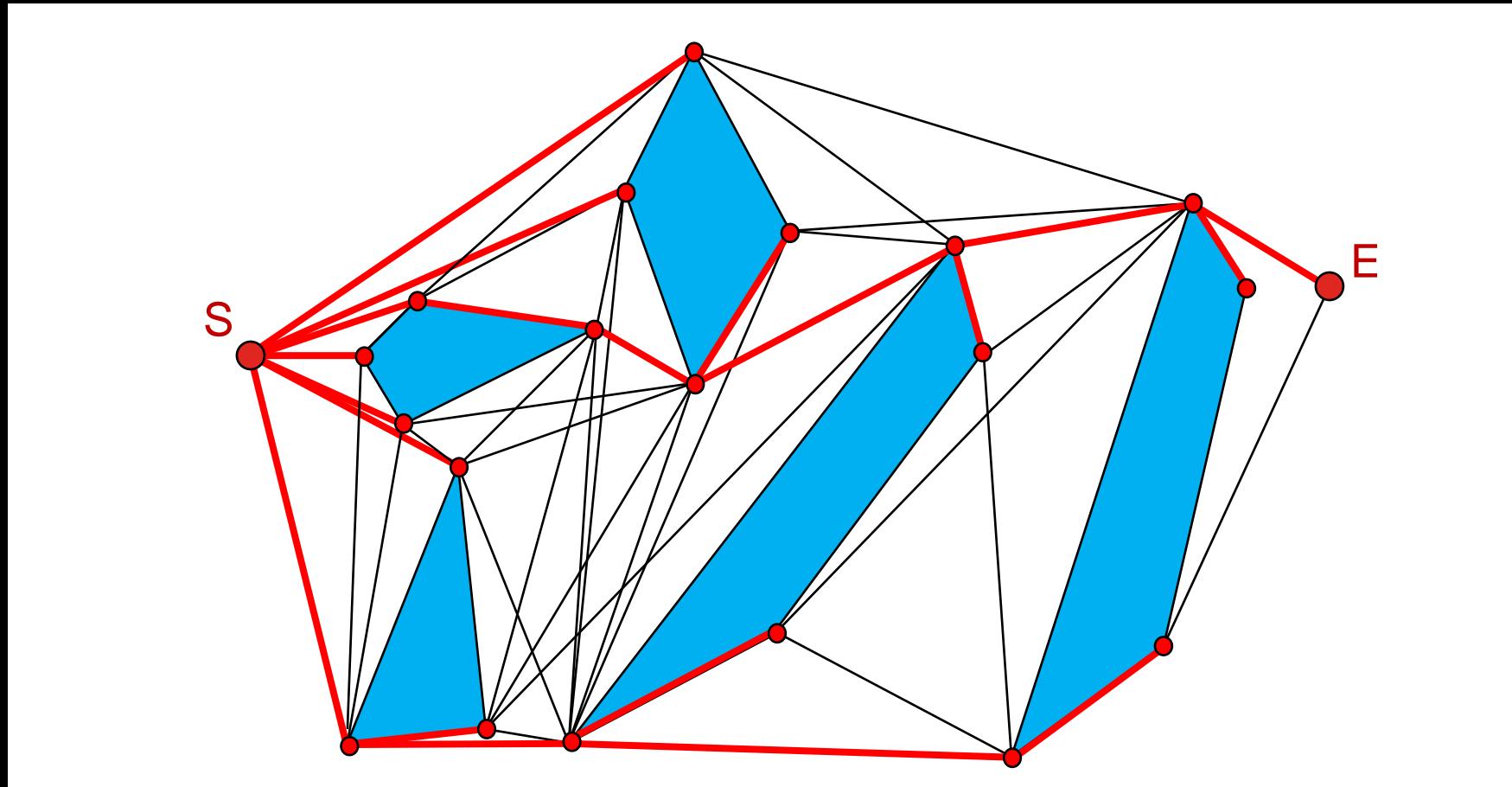
The last 3 points form an ear, so add it too

Start the  
Lab ...

# Computing Shortest Paths

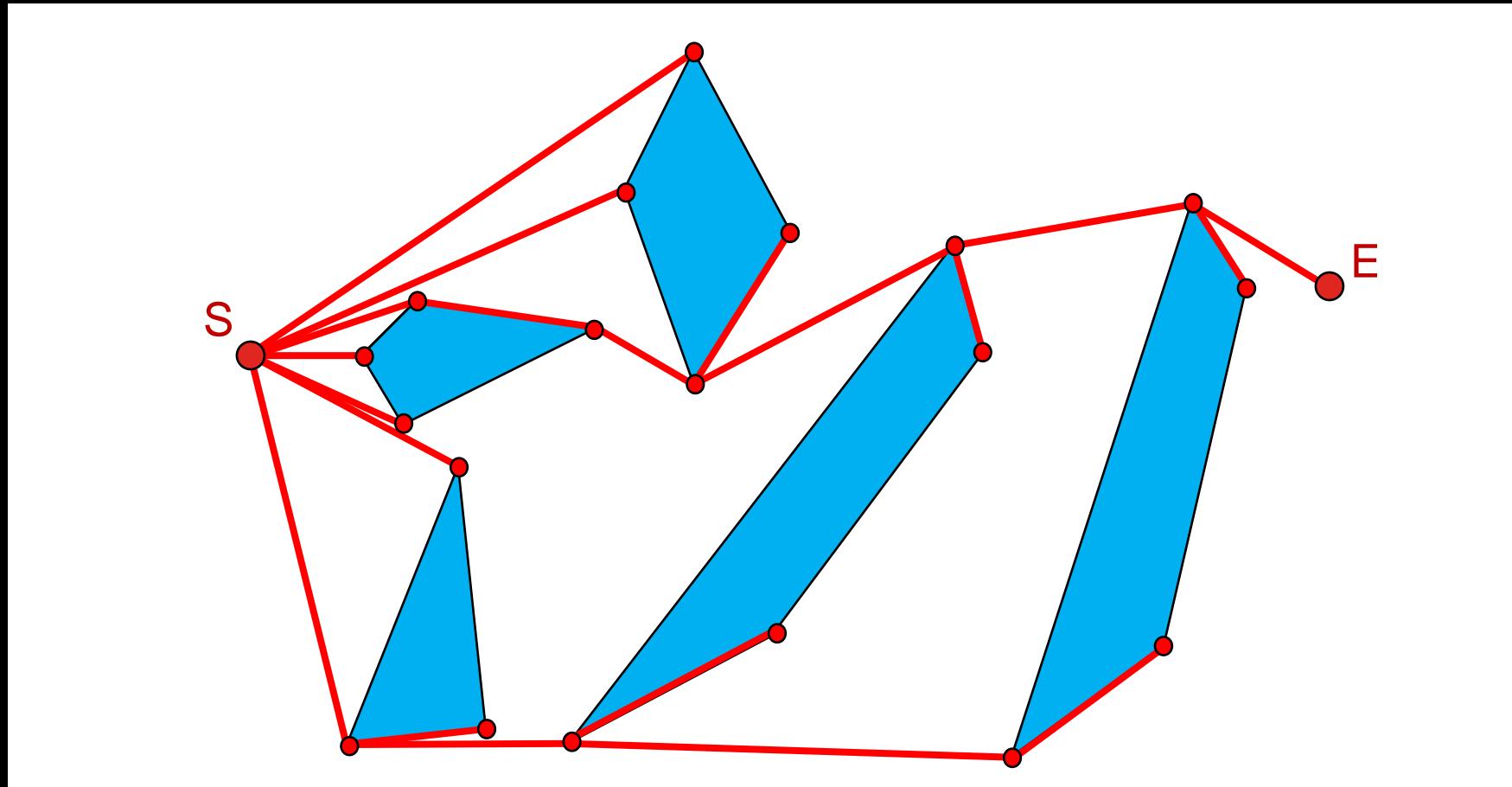
# Shortest Paths

- The shortest path from the start to each vertex of the visibility graph will consist of edges of the graph:



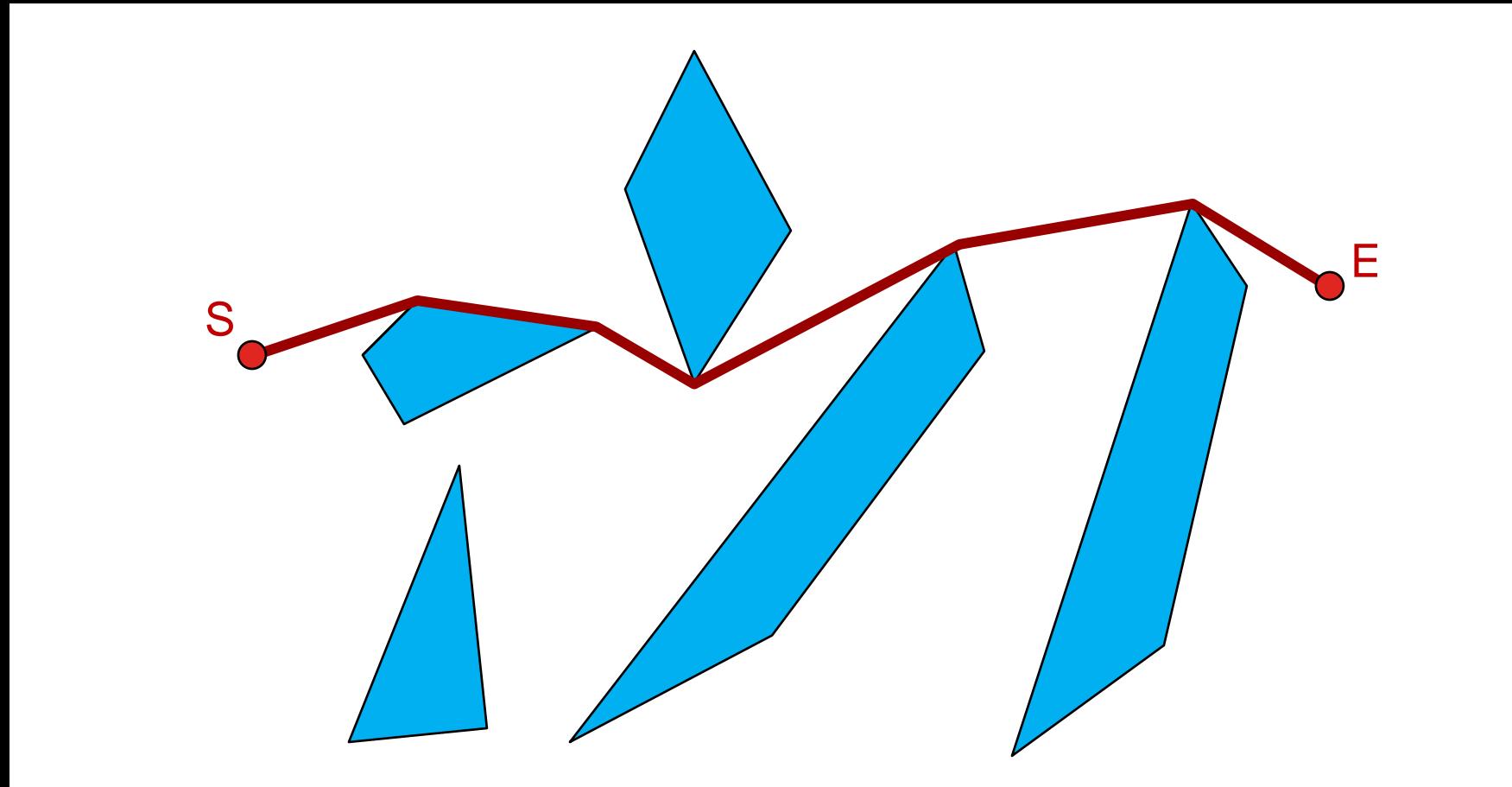
# Shortest Path Algorithm

- Result is called the *Shortest Path Tree*:



# Shortest Path Algorithm

- The shortest path to the goal is one of these paths:



# Dijkstra's Shortest Path Algorithm

- A popular algorithm for computing shortest paths in a graph is known as **Dijkstra's Algorithm**:
  - Starts with a *weight* of ZERO at the start node
  - Propagates outwards from the source (like a waveform) to all graph edges.
    - Nodes “closer to” the source are visited before those further away.
  - Each time an edge is travelled along, the robot incurs a cost according to some metric (e.g., distance, time, battery usage, etc..) which is usually represented by a *weight* on the edge.
  - Once all nodes have been reached by the “wavefront”, the algorithm is done, and each node will have a weight corresponding to the cost to get there from the source.

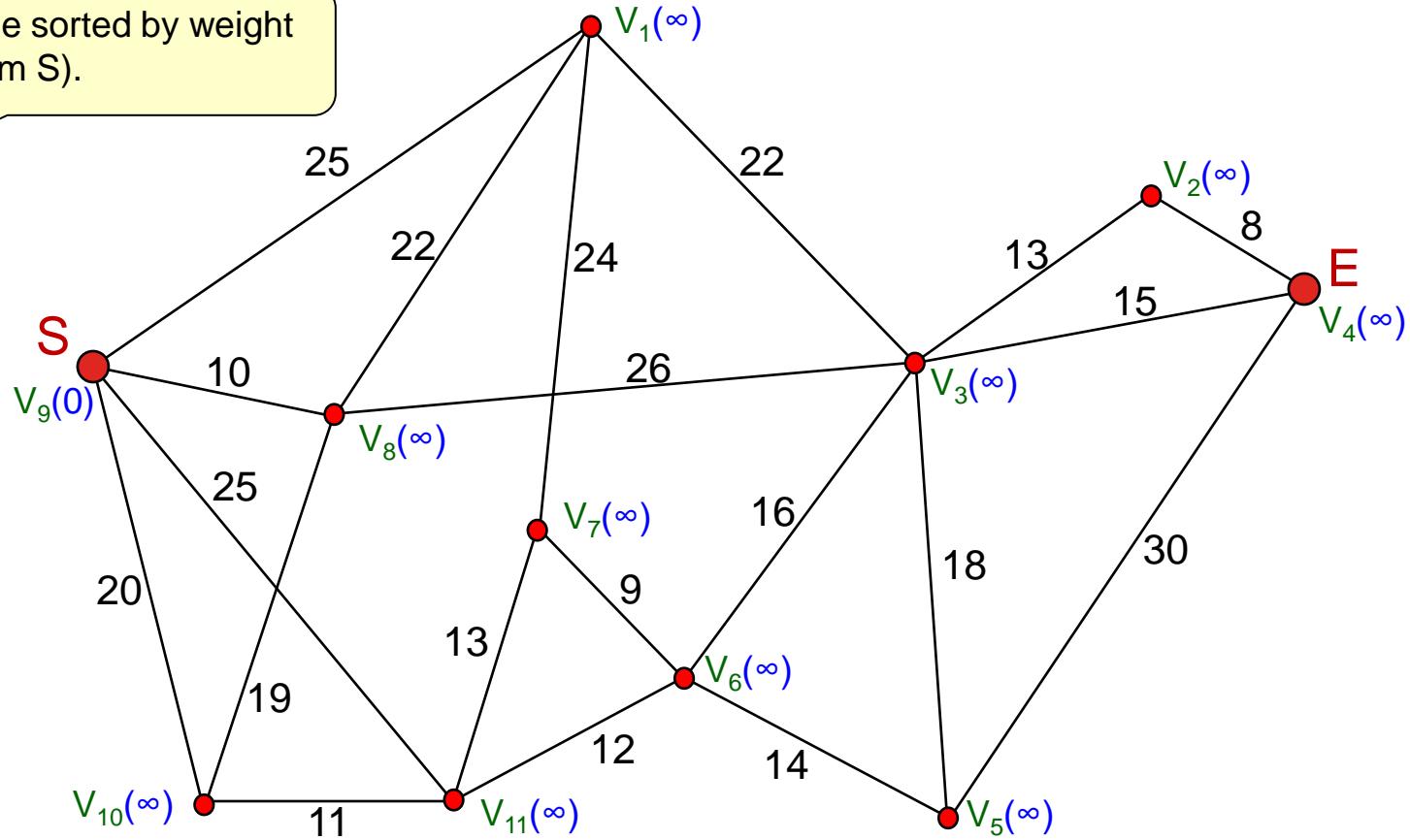


# Dijkstra's Shortest Path Algorithm

- To start, source is given weight of 0, all other nodes a weight of  $\infty$ . Nodes are stored in priority queue.

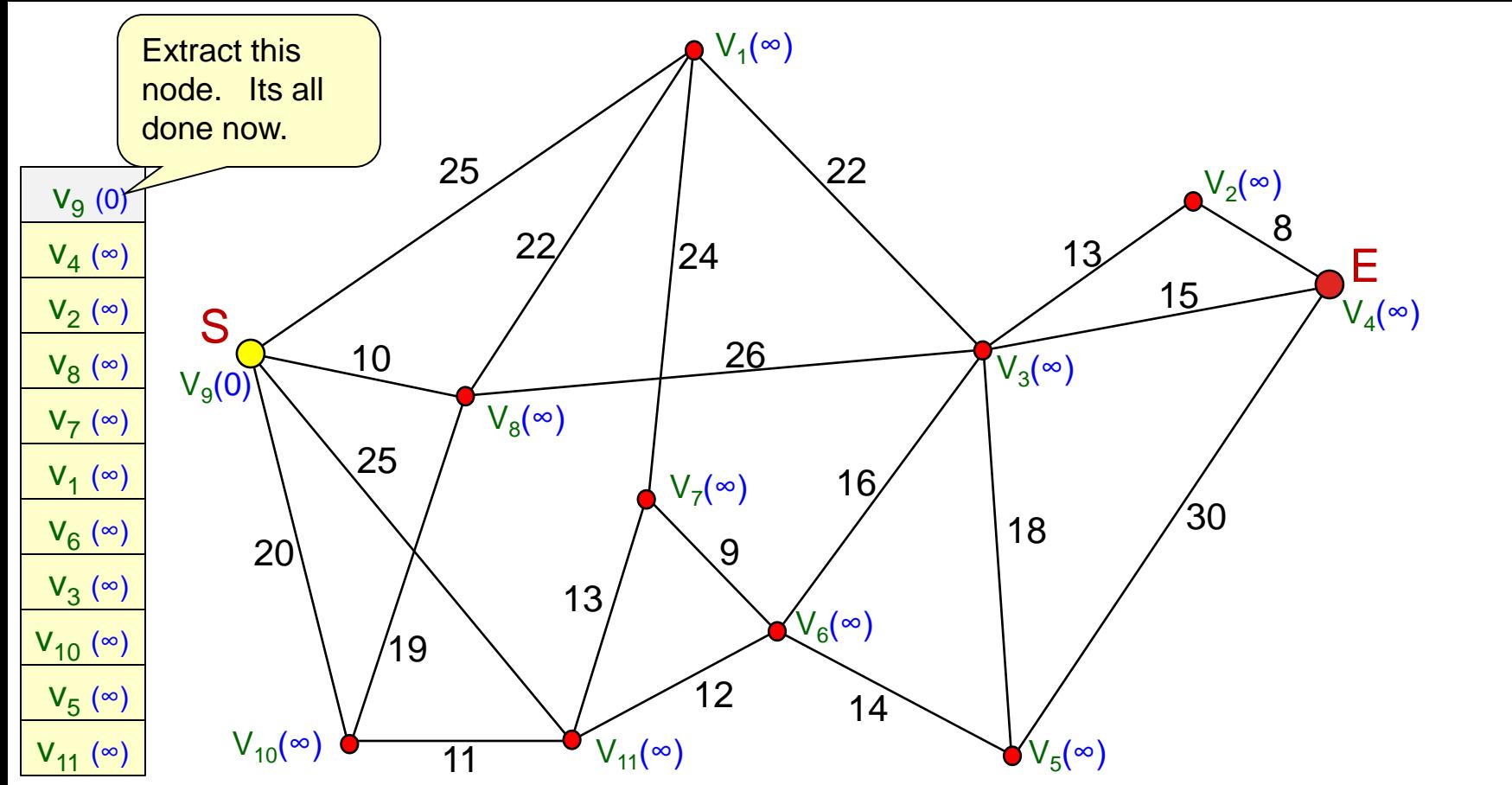
Priority queue sorted by weight  
(i.e., cost from S).

V <sub>9</sub> (0)
V <sub>4</sub> ( $\infty$ )
V <sub>2</sub> ( $\infty$ )
V <sub>8</sub> ( $\infty$ )
V <sub>7</sub> ( $\infty$ )
V <sub>1</sub> ( $\infty$ )
V <sub>6</sub> ( $\infty$ )
V <sub>3</sub> ( $\infty$ )
V <sub>10</sub> ( $\infty$ )
V <sub>5</sub> ( $\infty$ )
V <sub>11</sub> ( $\infty$ )



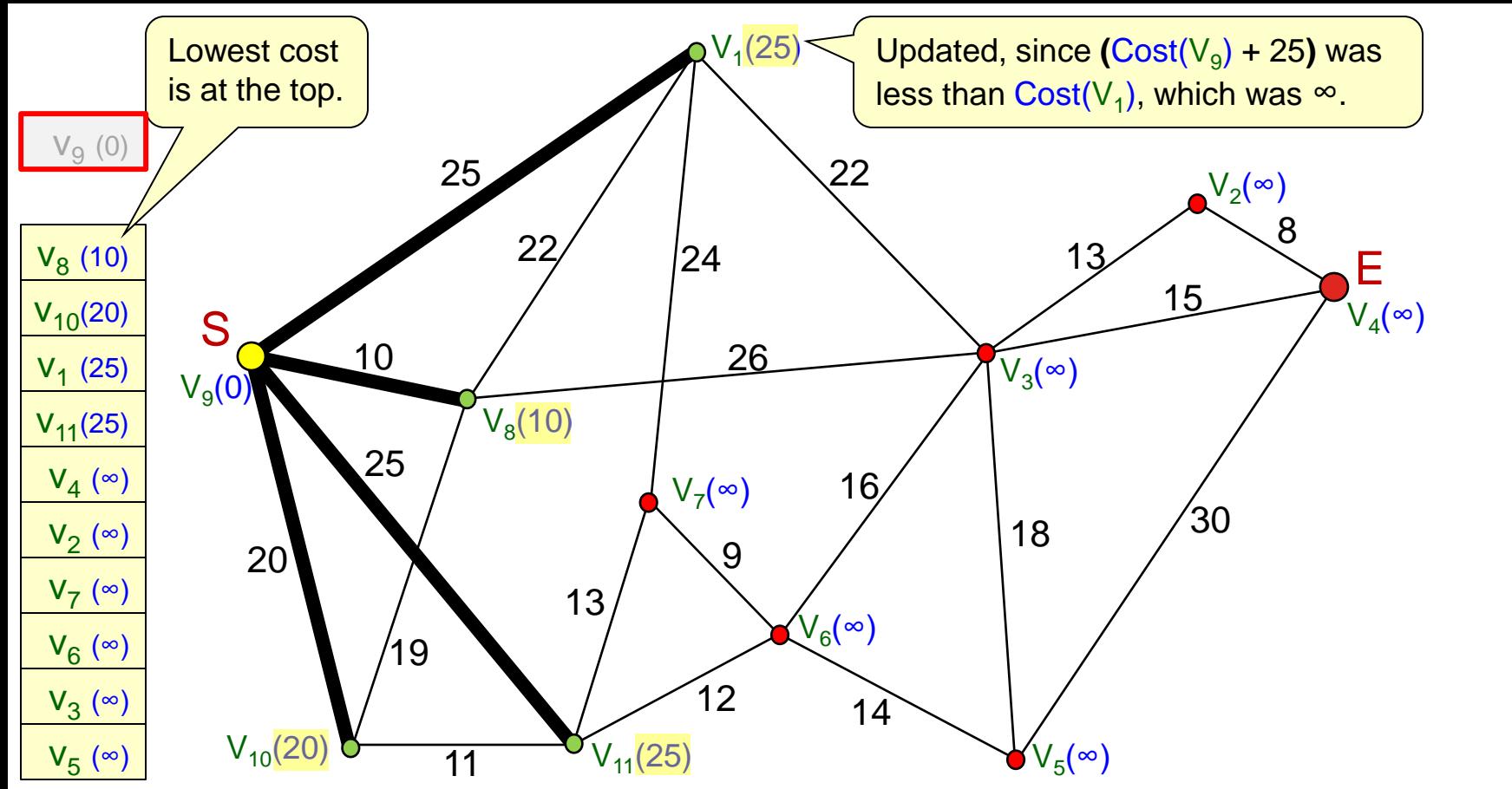
# Dijkstra's Shortest Path Algorithm

- Algorithm repeatedly extracts top node from queue.
  - An extracted node is done being processed, has its final cost



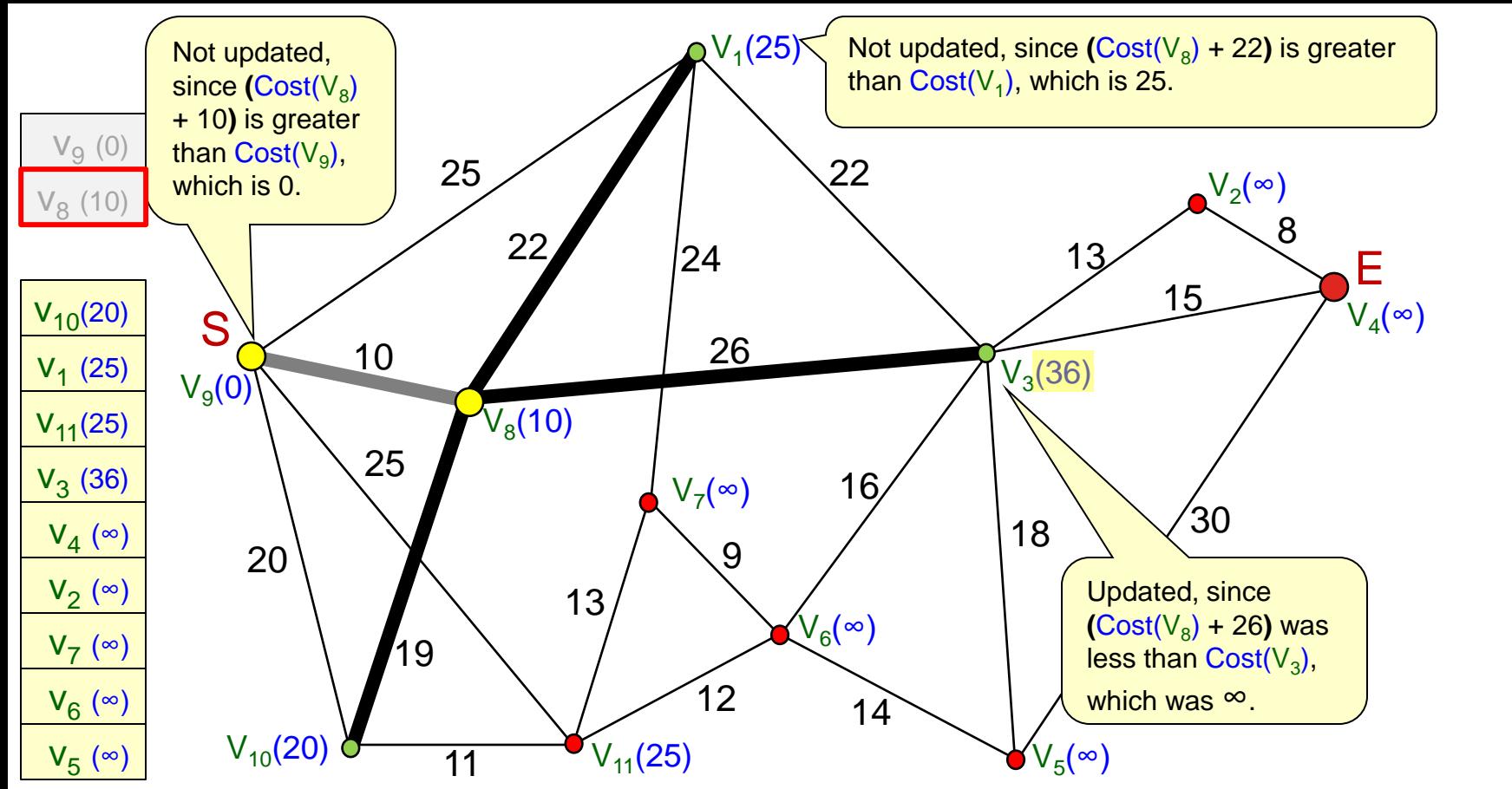
# Dijkstra's Shortest Path Algorithm

- Visit all nodes connected to the extracted node
  - Update their cost if it is less (use e.g., edge length)



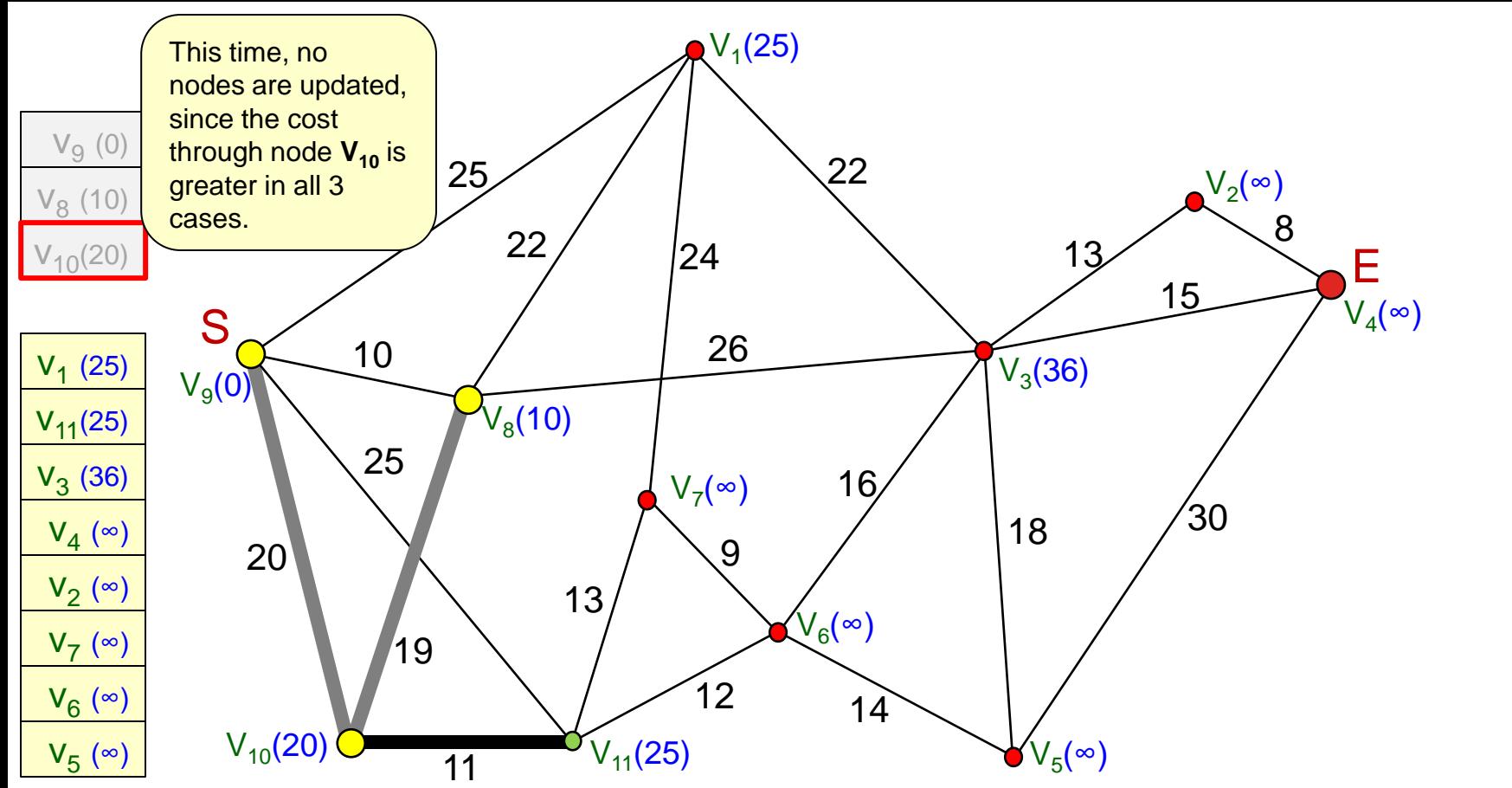
# Dijkstra's Shortest Path Algorithm

- Repeat again, taking off the next closest node and check its neighbours.



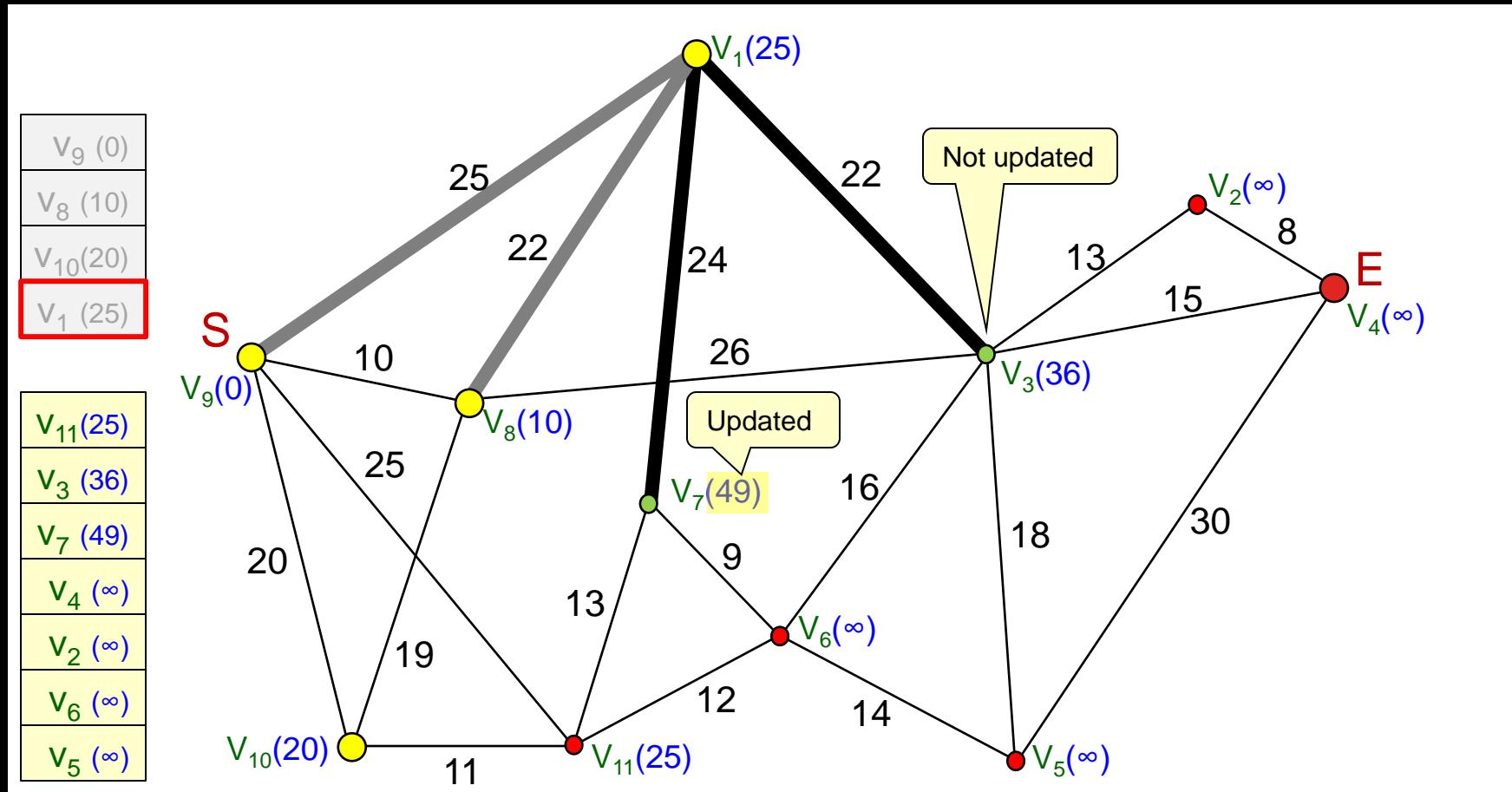
# Dijkstra's Shortest Path Algorithm

- Repeat again, always updating nodes if the cost through this extracted node is lower.



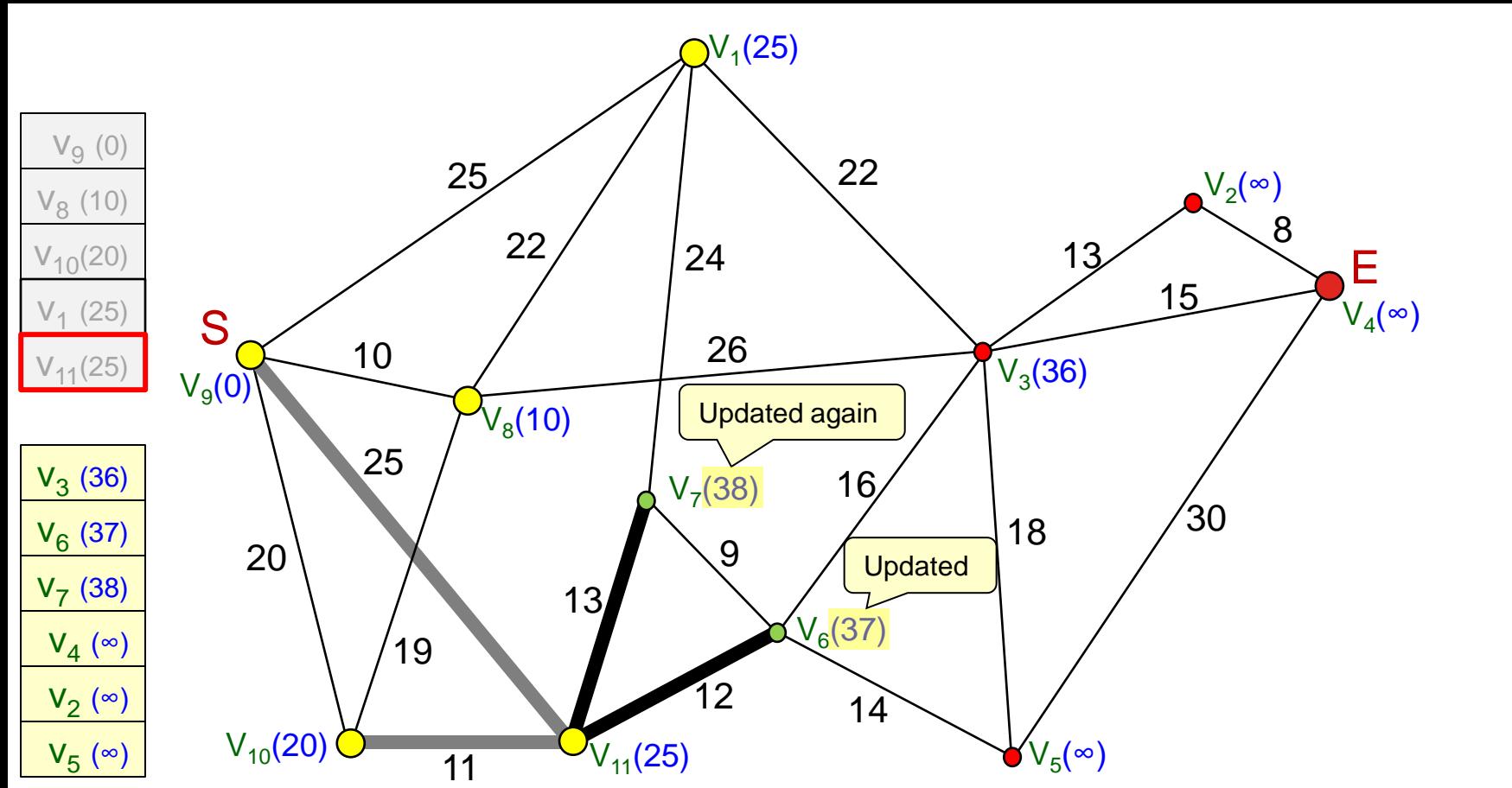
# Dijkstra's Shortest Path Algorithm

- Repeat again ...



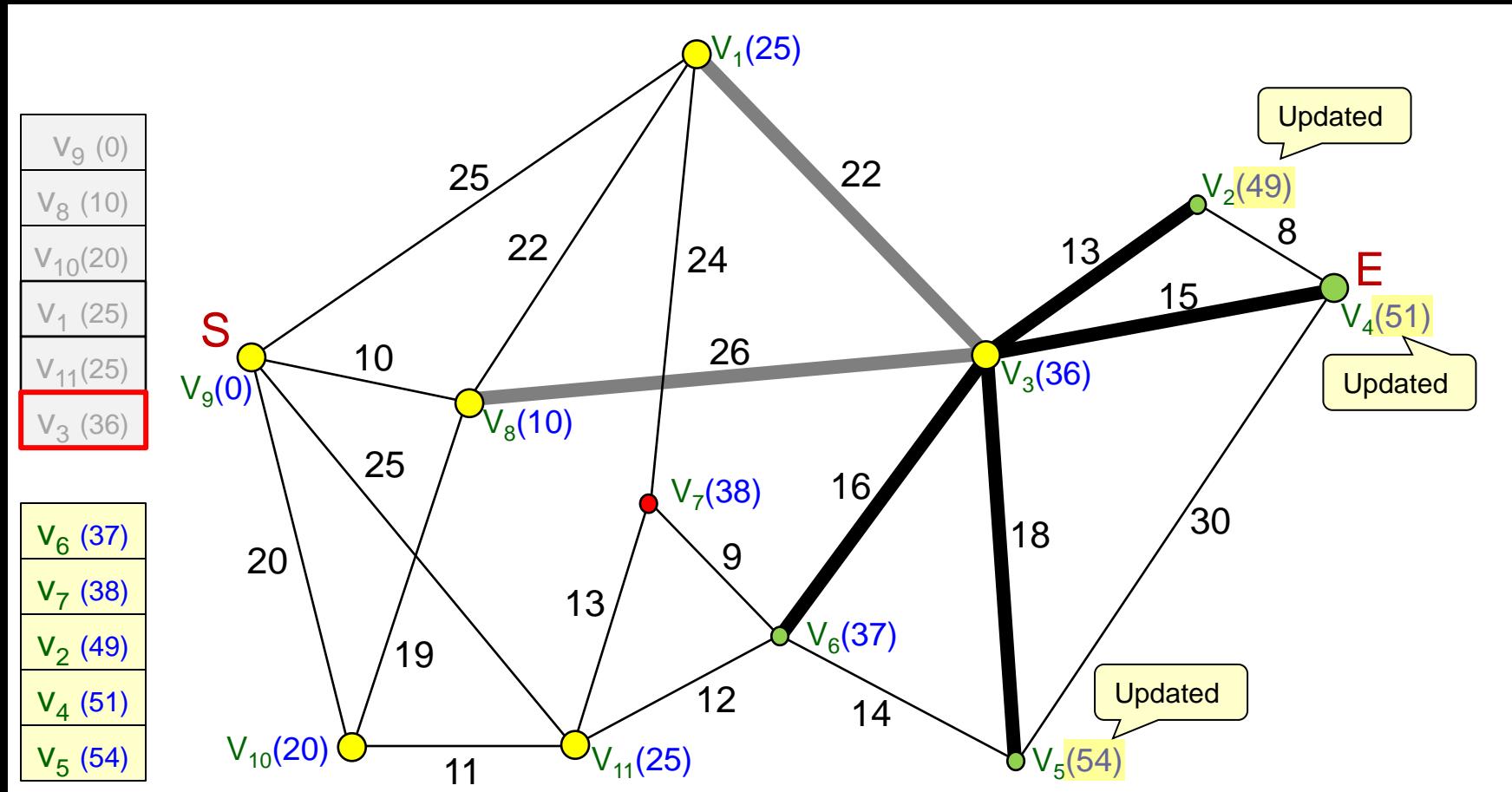
# Dijkstra's Shortest Path Algorithm

- Repeat again ...



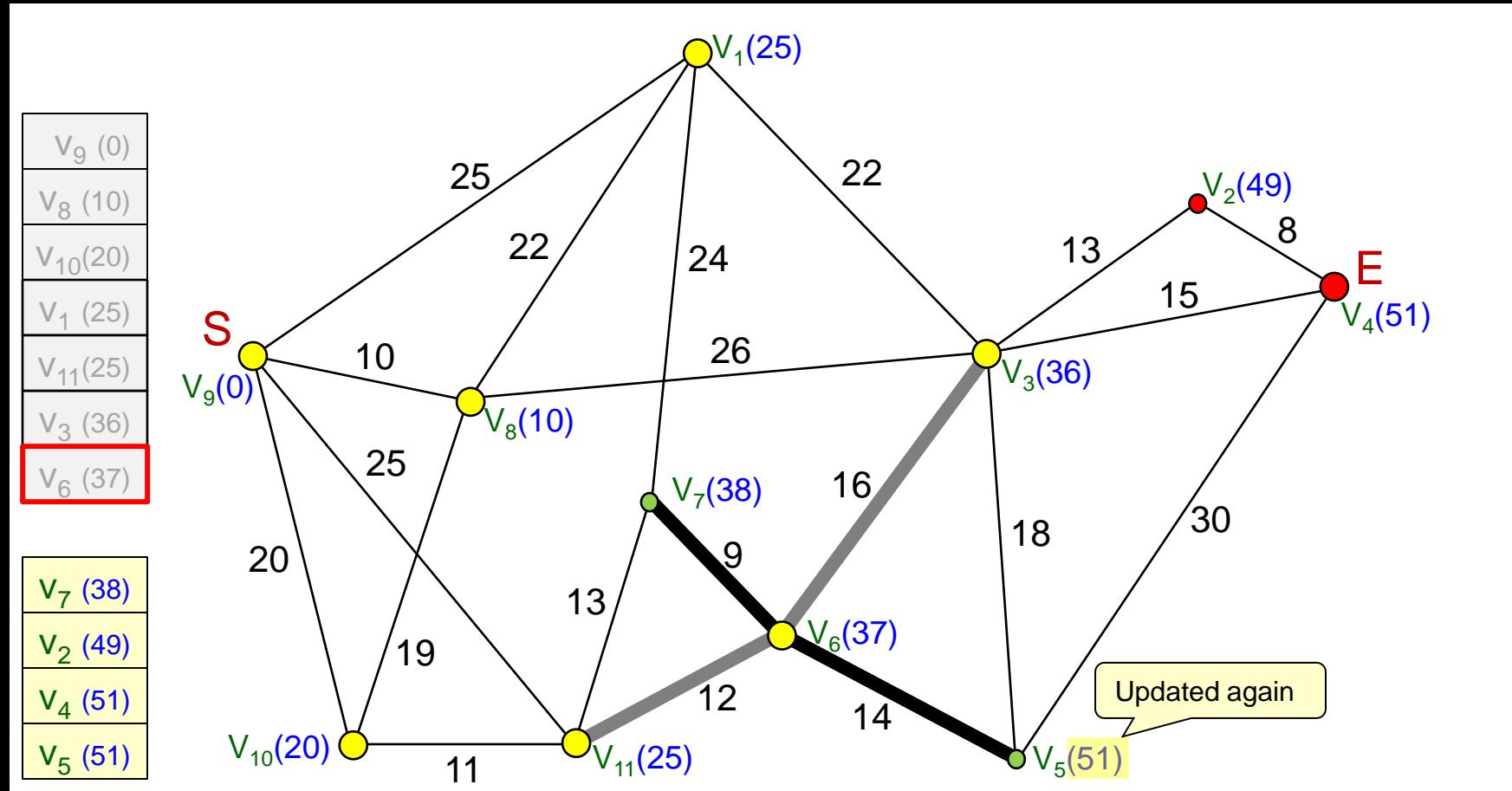
# Dijkstra's Shortest Path Algorithm

- Keep going ...



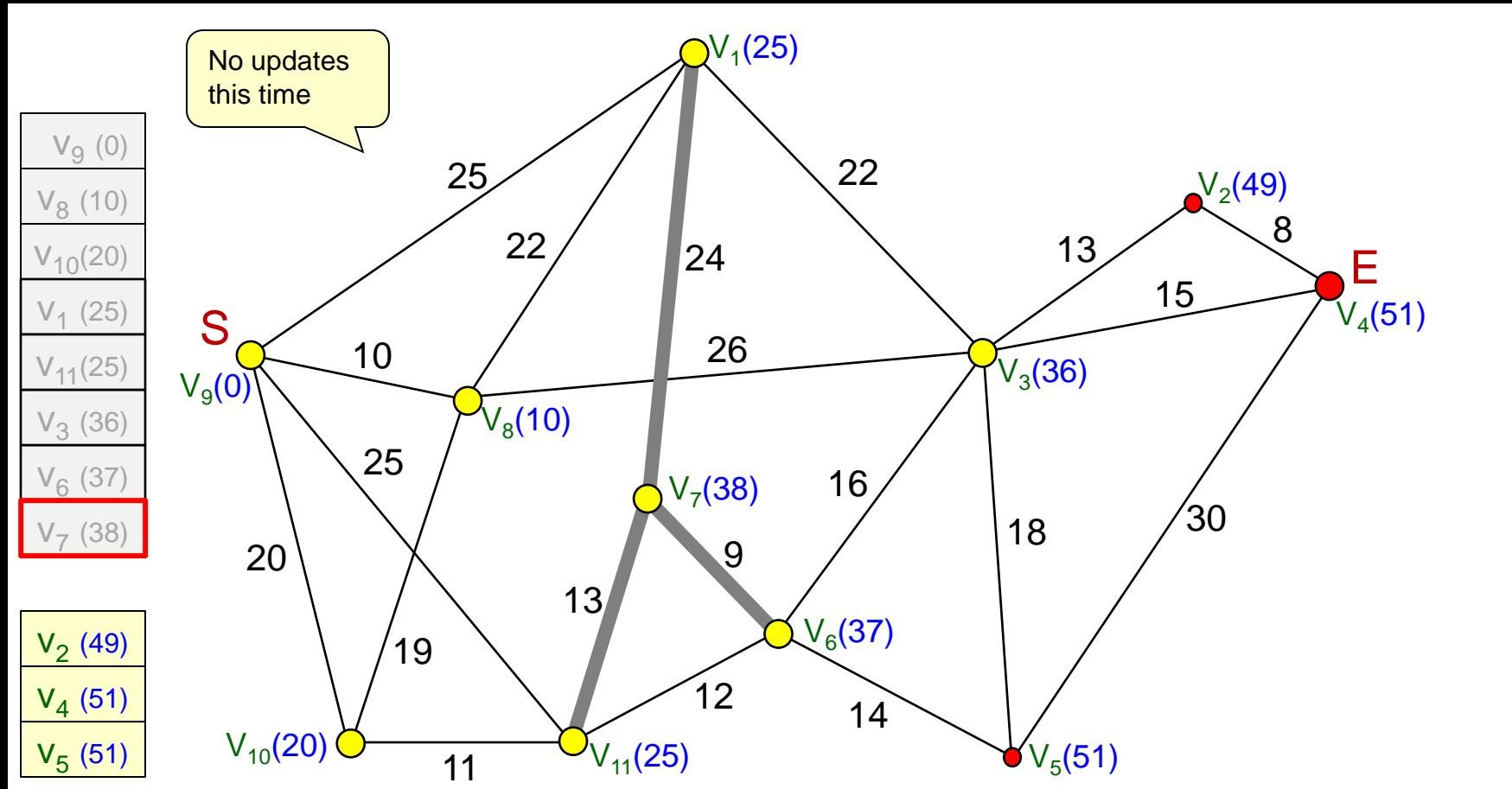
# Dijkstra's Shortest Path Algorithm

- Keep going ...



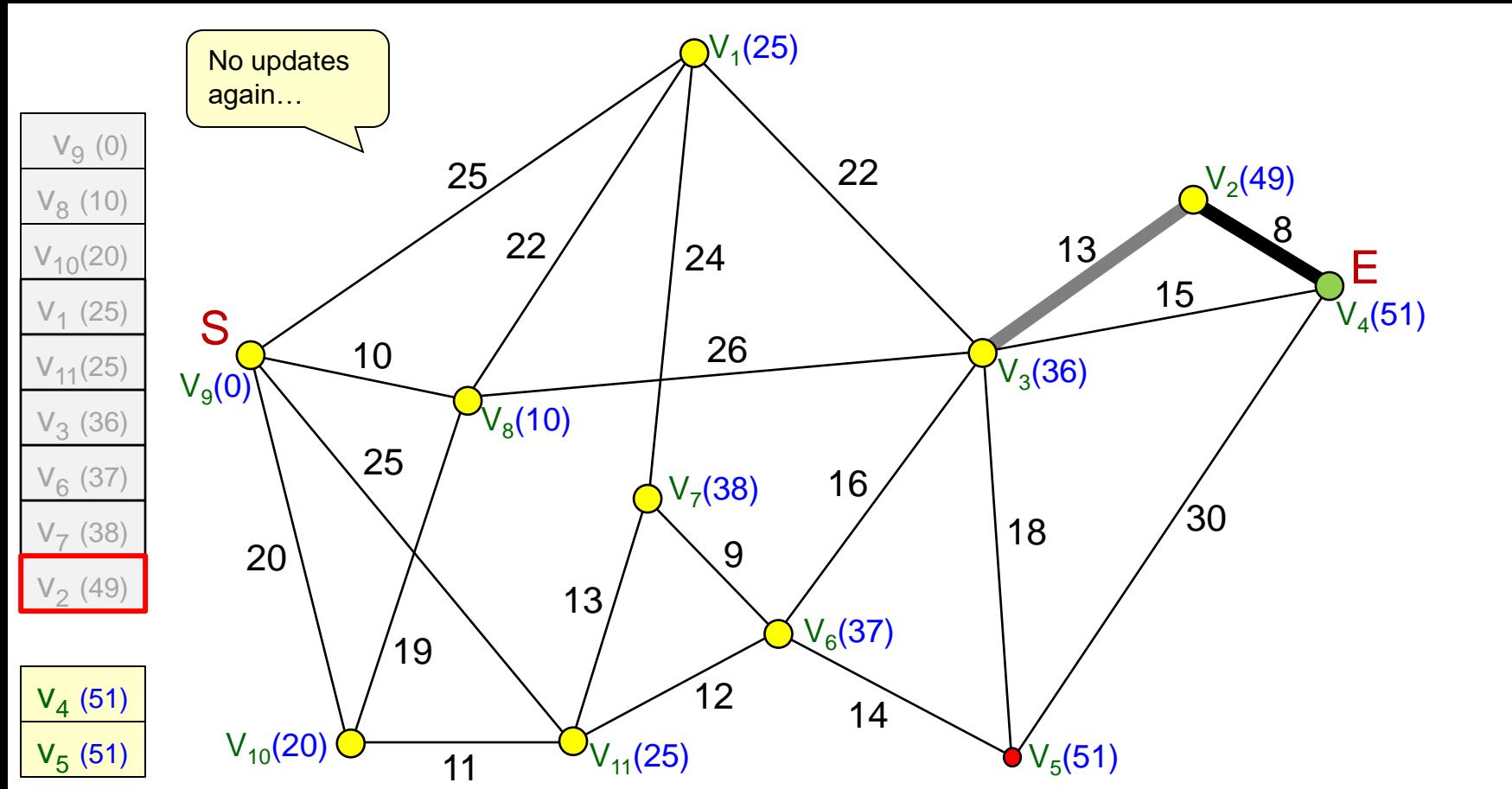
# Dijkstra's Shortest Path Algorithm

- Almost done ...



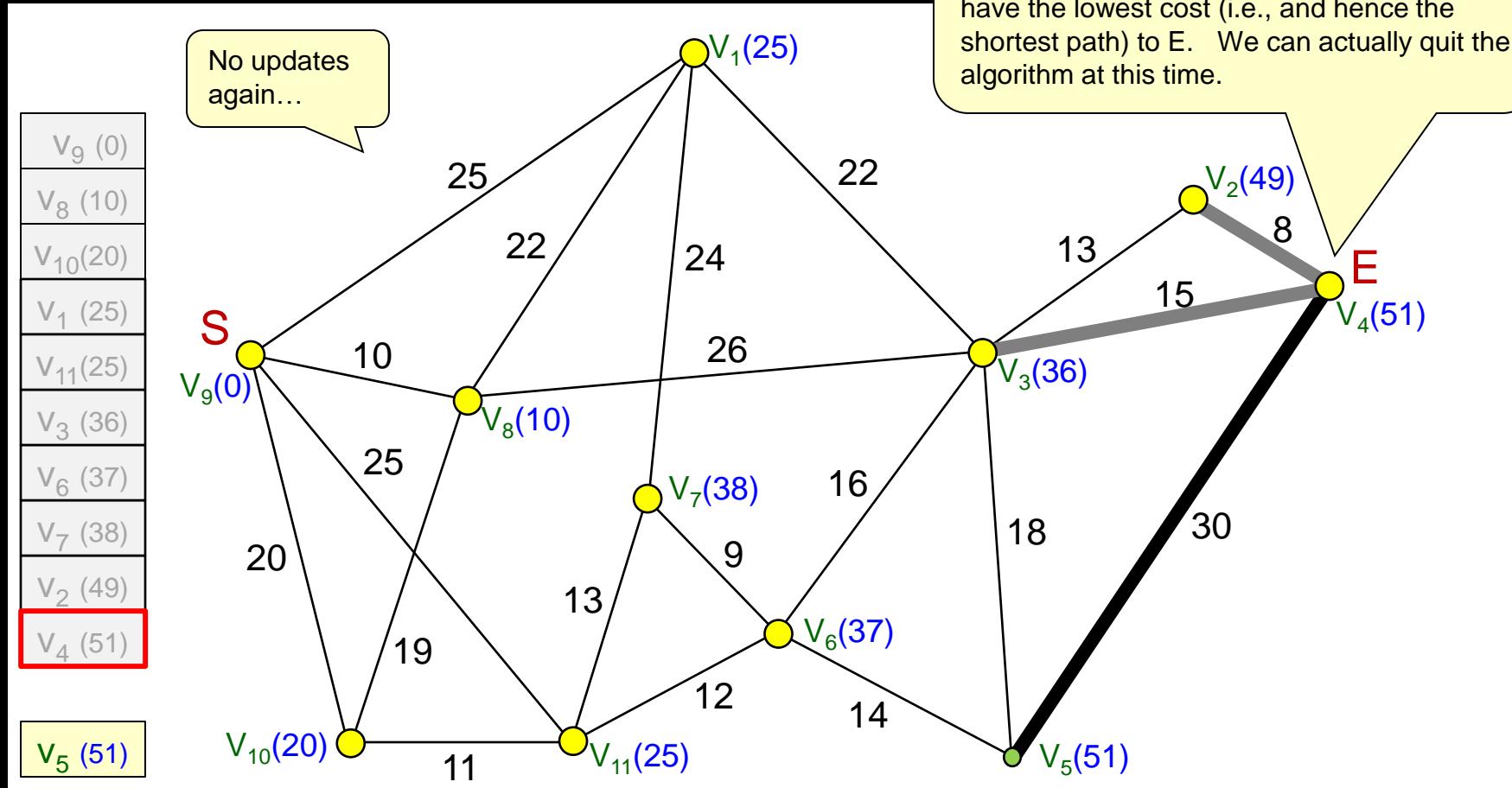
# Dijkstra's Shortest Path Algorithm

- Almost done ...



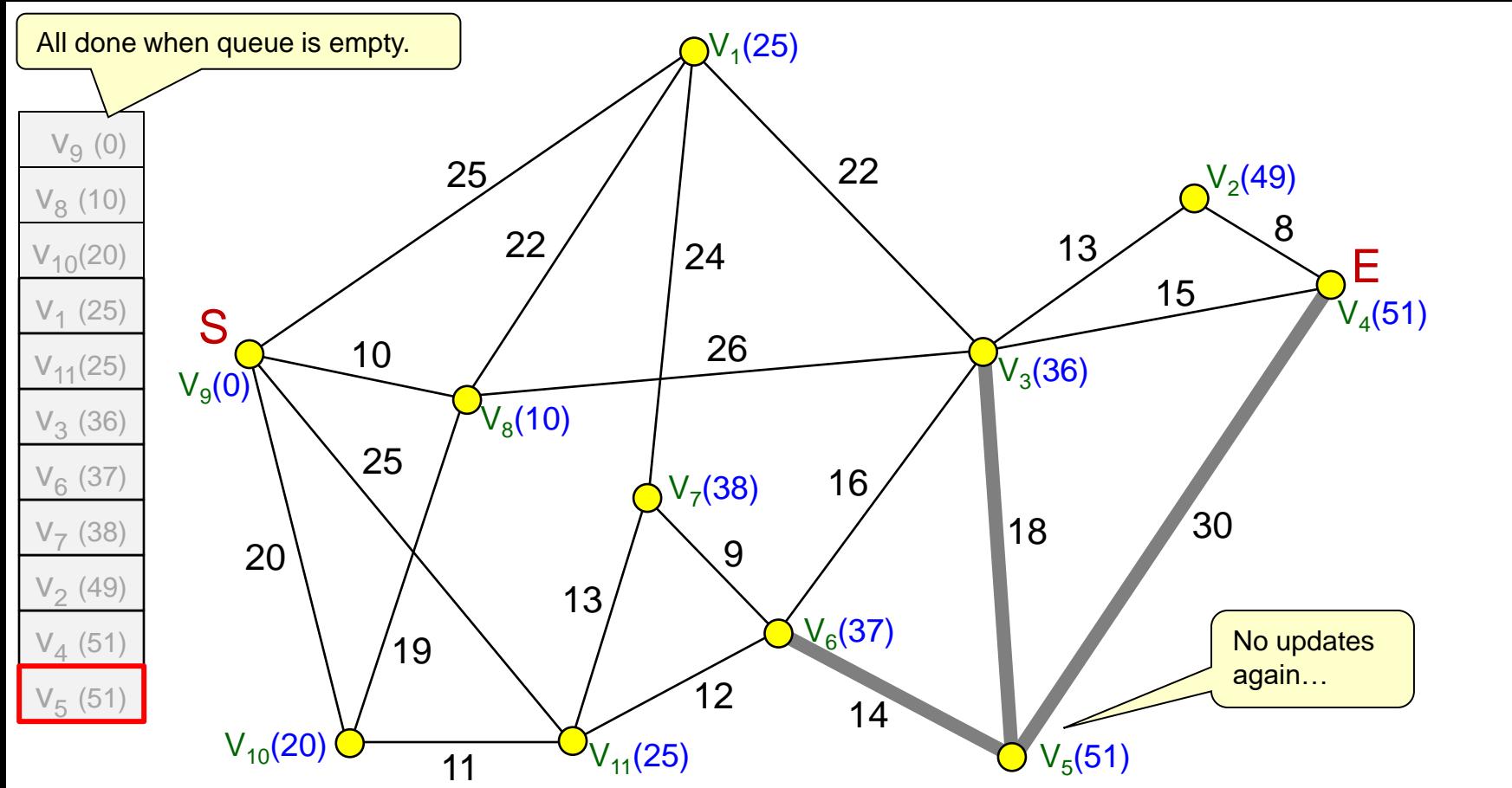
# Dijkstra's Shortest Path Algorithm

- Almost done ...



# Dijkstra's Shortest Path Algorithm

- And this completes it. We now have the shortest path cost to each node, with respect to the source S.



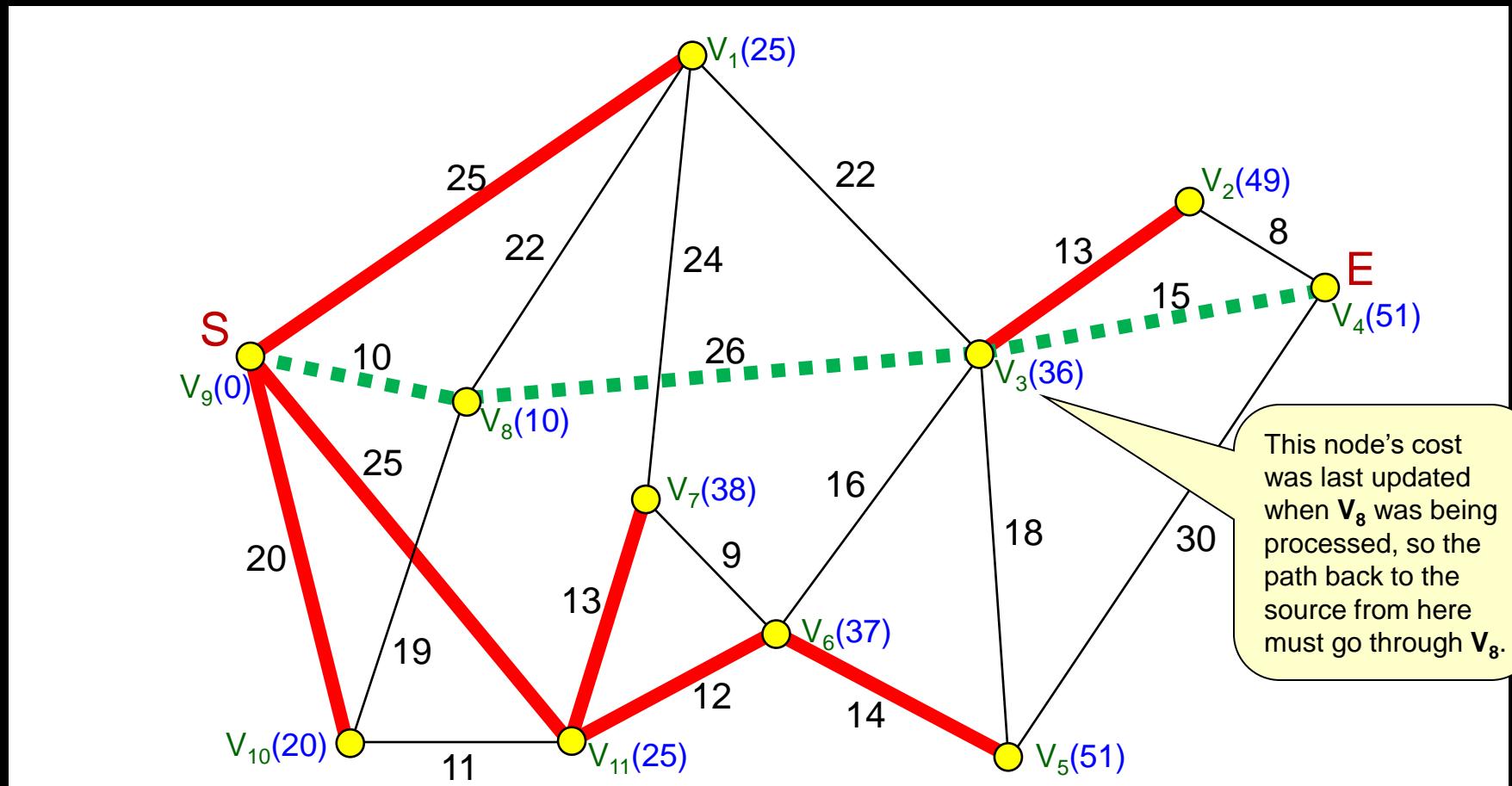
# Computing the Shortest Path Tree

- We can use Dijkstra's shortest path algorithm to compute the shortest path tree from **s** in this graph.
  - Takes  $O(V \log V + E)$  time for a  $V$ -vertex /  $E$ -edge graph

```
1 FUNCTION DijkstraShortestPathTree(G, s)
2     Initialize weight(v) of each vertex v to  $\infty$  but initialize weight(s) of s to 0
3     Q = a queue containing all vertices sorted by weights (lowest weight is at front)
4     WHILE (Q is not empty) DO
5         v = get and remove the vertex from Q with minimal weight
6         FOR each edge  $\overrightarrow{vu}$  outgoing from v DO
7             IF (weight(u) > weight(v) +  $|\overrightarrow{vu}|$ ) THEN
8                 weight(u) = weight(v) +  $|\overrightarrow{vu}|$ 
9                 Re-sort node u in Q (because a weight has changed now)
```

# Finding a Shortest Path

- Trace path from any node back to source by remembering node that updated the cost to it:



# Remembering How We Got There

- When updating a node's cost to a better one, just add a line to remember which vertex led to that node in the path from  $s$

1 **FUNCTION** DijkstraShortestPathTree( $G, s, e$ )

Need to add parameter  $e$  if we don't want to compute the whole tree (e.g., if we just want to find the path to  $e$ )

2 Initialize **weight( $v$ )** of each vertex  $v$  to  $\infty$  but initialize **weight( $s$ )** of  $s$  to **0**

3  $Q$  = a queue containing all vertices sorted by weights (lowest weight is at front)

4 **WHILE** ( $Q$  is not empty) **DO**

5      $v$  = get and remove the vertex from  $Q$  with minimal weight

6       **// if ( $v$  is the destination  $e$ ) then break out of loop**

7       **FOR** each edge  $\overline{vu}$  outgoing from  $v$  **DO**

8           **IF** ( $\text{weight}(u) > \text{weight}(v) + |\overline{vu}|$ ) **THEN**

Only add this if we don't want to compute the whole tree (e.g., if we just want to find the path to  $e$ )

9               **Set parent of  $u$  to  $v$**

Store  $v$  as the node that let to  $u$  in the shortest path from  $s$  to  $u$ . So  $v$  is the parent of  $u$  in the shortest path tree from  $s$ .

10                $\text{weight}(u) = \text{weight}(v) + |\overline{vu}|$

11               Re-sort node  $u$  in  $Q$  (because a weight has changed now)

# Tracing the Path Back

- Finding the shortest path from **s** to **e** involves tracing the path back from **e** to **s**:

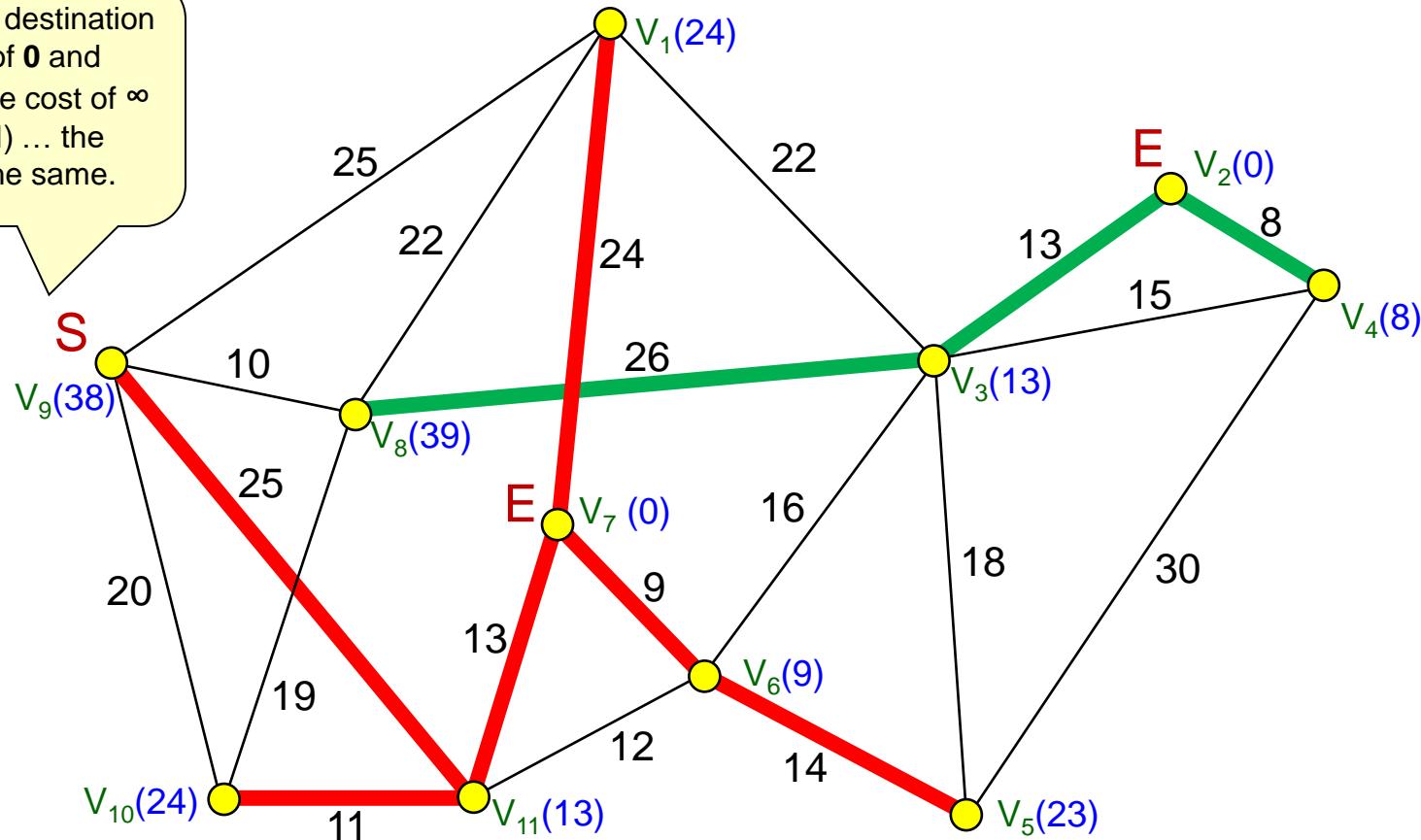
```
1 FUNCTION TraceBackPath(G, s, e)
2   currentNode = e
3   path = an empty list
4   WHILE (currentNode is not s) DO
5     add currentNode to front of path list
6     currentNode = parent of currentNode
7   Add s to front of path
```

Just get parent, then the parent of that parent, then that node's parent ... etc ... until we reach **s**.

# Multiple Sources

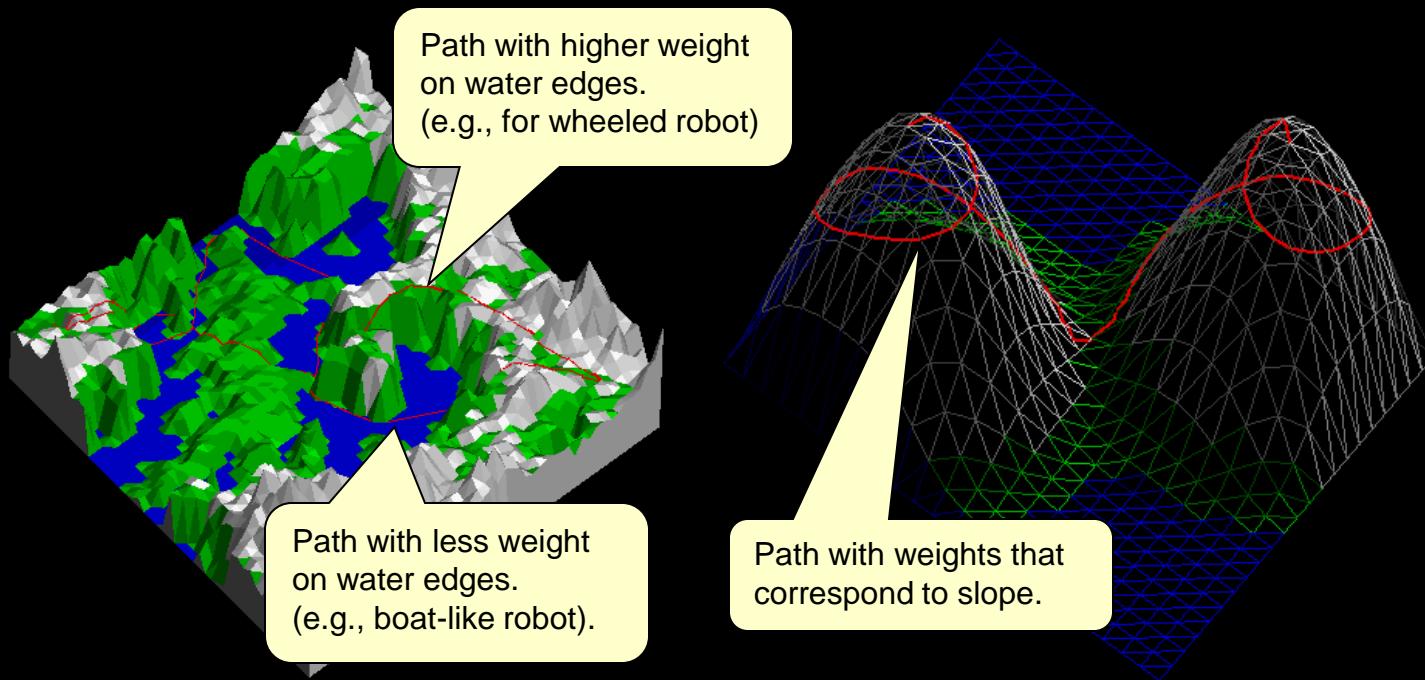
- Interestingly, the algorithm also works for multiple destinations.
  - Robot may wish to go to the closest of a set of destinations.

Just set each destination to have cost of **0** and source to have cost of  $\infty$  (i.e., reversed) ... the algorithm is the same.



# Other Metrics

- Algorithm allows arbitrary weights on edges (as long as they are positive).
  - Allows some edges to be more “costly” than others
  - Can result in a kind of “weighted shortest path” that can go, for example, around obstacles (e.g., water).

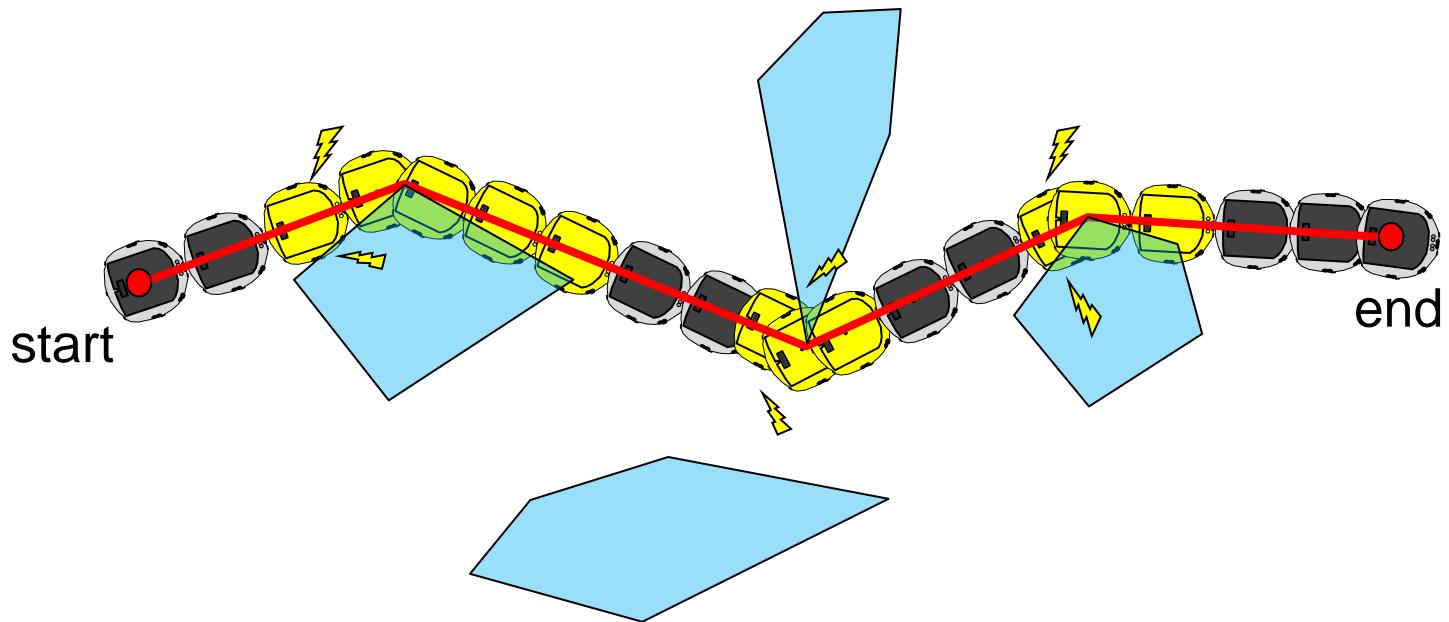


Start the  
Lab ...

# Grown Obstacle Space

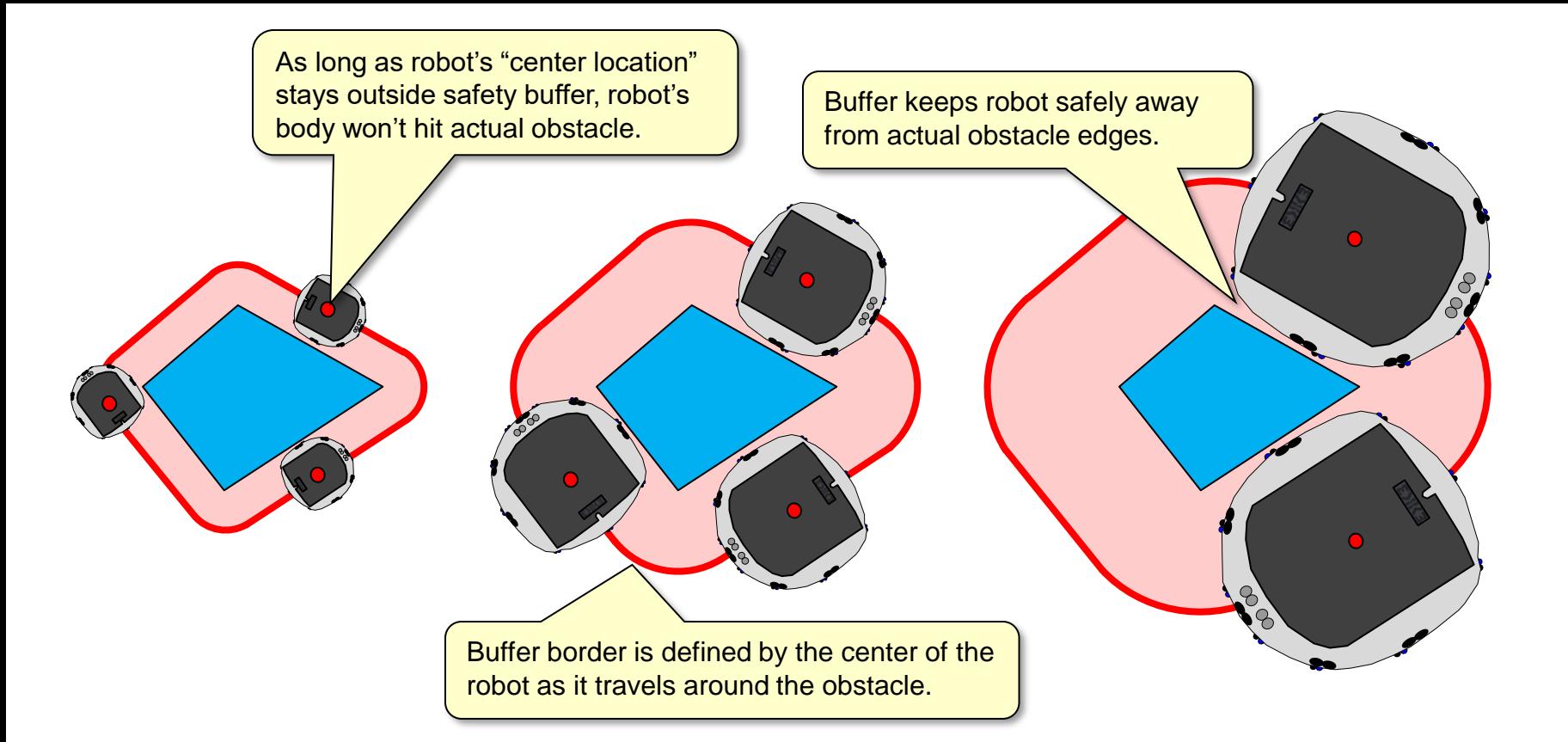
# Shortest Path Problems

- Any real robot will collide with obstacles if it travels along our computed shortest path because we assumed that the robot was just a point, but a robot has a shape:



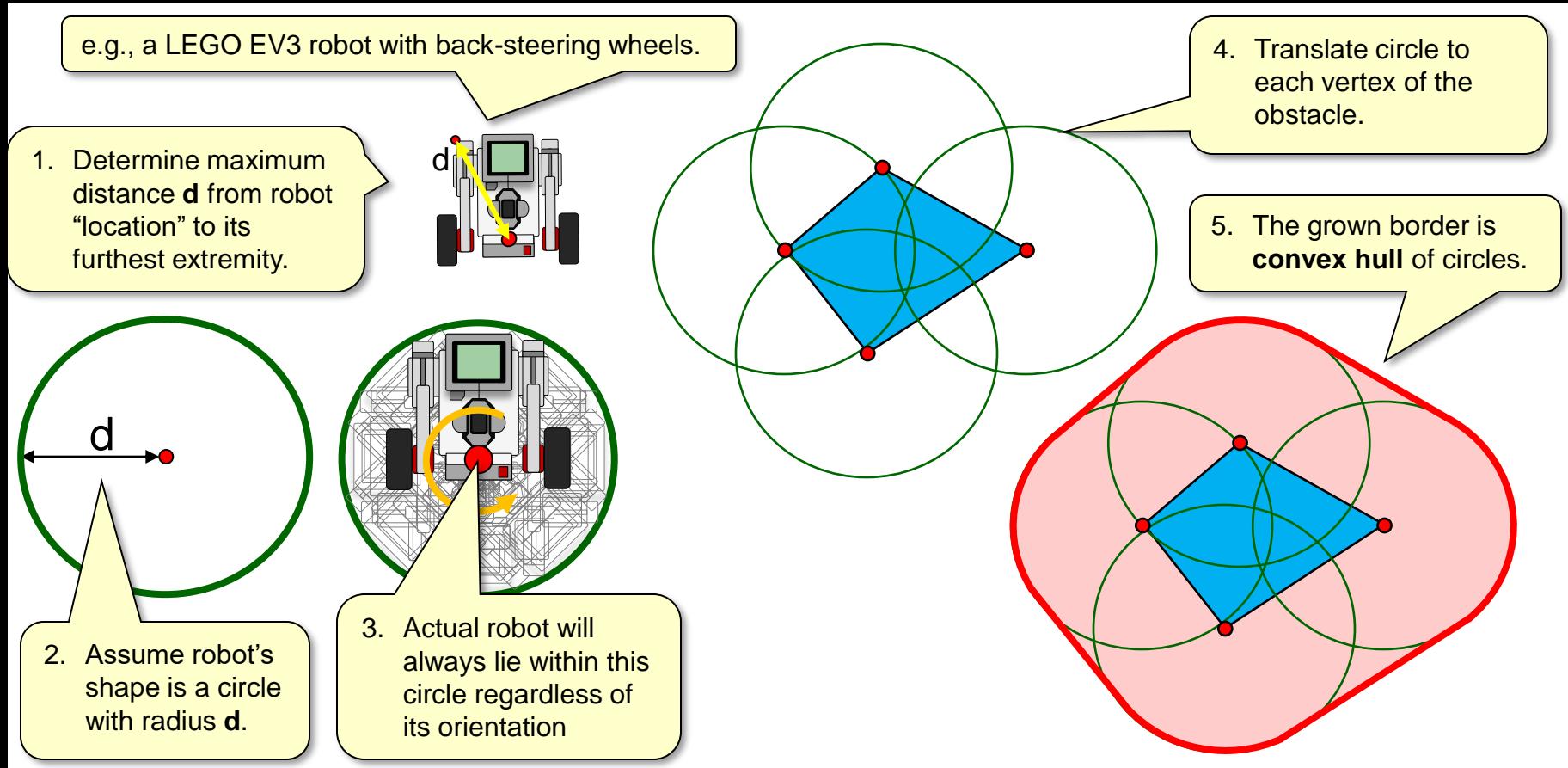
# Safety Buffers

- Need to have a “safety buffer” around the obstacles which takes into account robot’s size and shape.



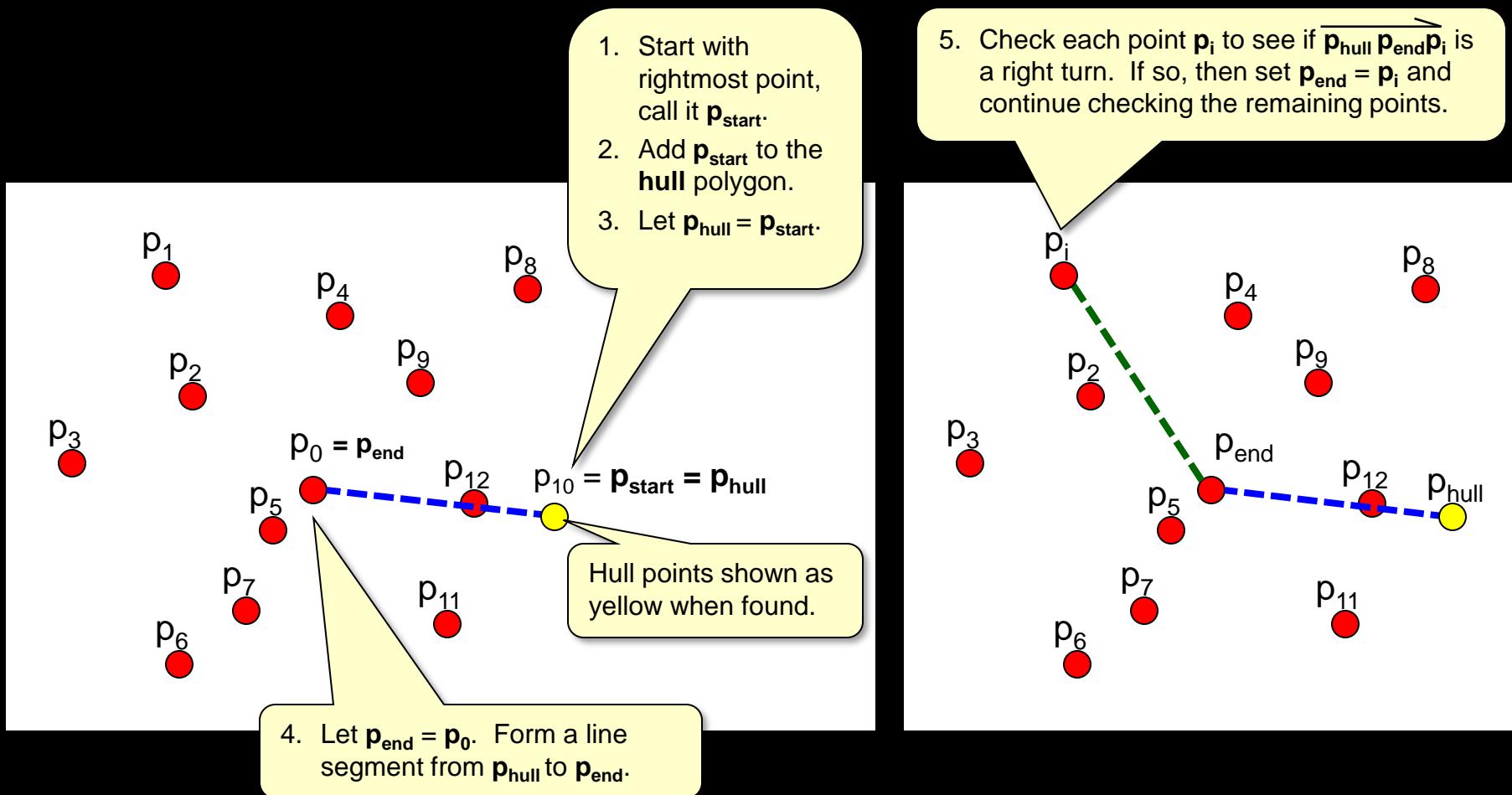
# Grown Obstacle Space

- Must “grow” obstacles by amount that considers **any orientation of robot** at any point on obstacle border:



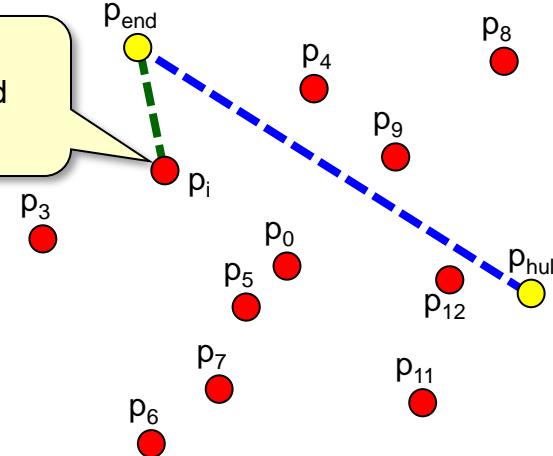
# Computing the Convex Hull

- Algorithms exist to find convex hull of a set of points
  - Graham Scan (a.k.a. “Gift-Wrapping” method) is a simple one:

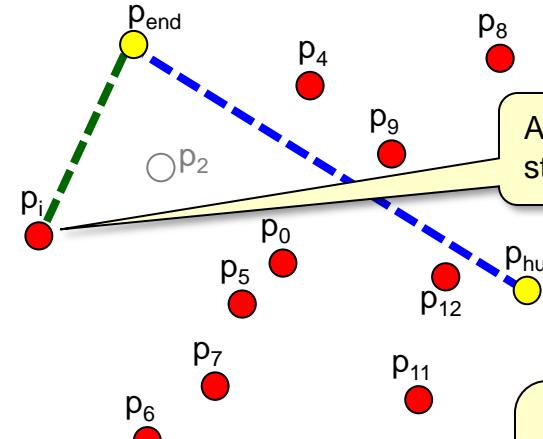


# Computing the Convex Hull

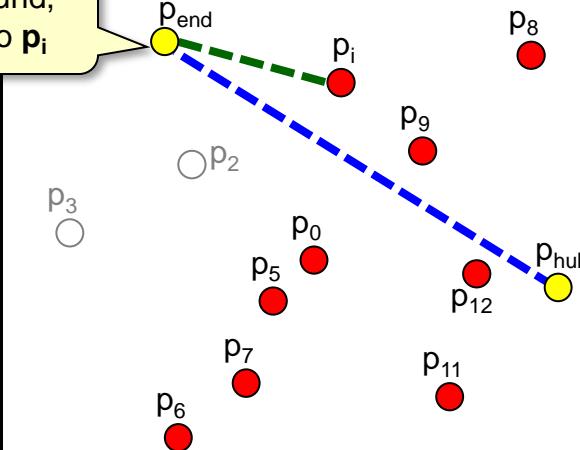
A left turn, so  $p_{end}$  stays the same and we continue



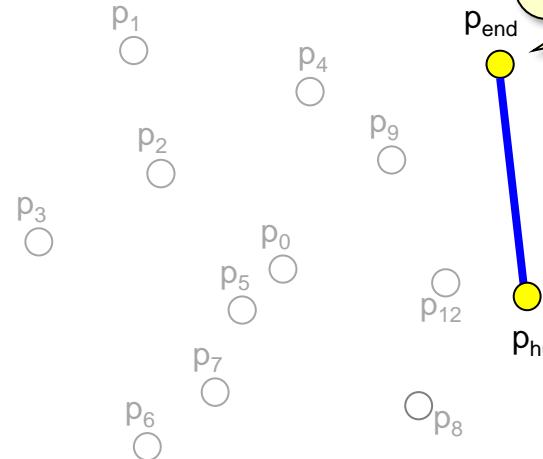
Another left turn, so  $p_{end}$  stays the same



When right turn found, now change  $p_{end}$  to  $p_i$

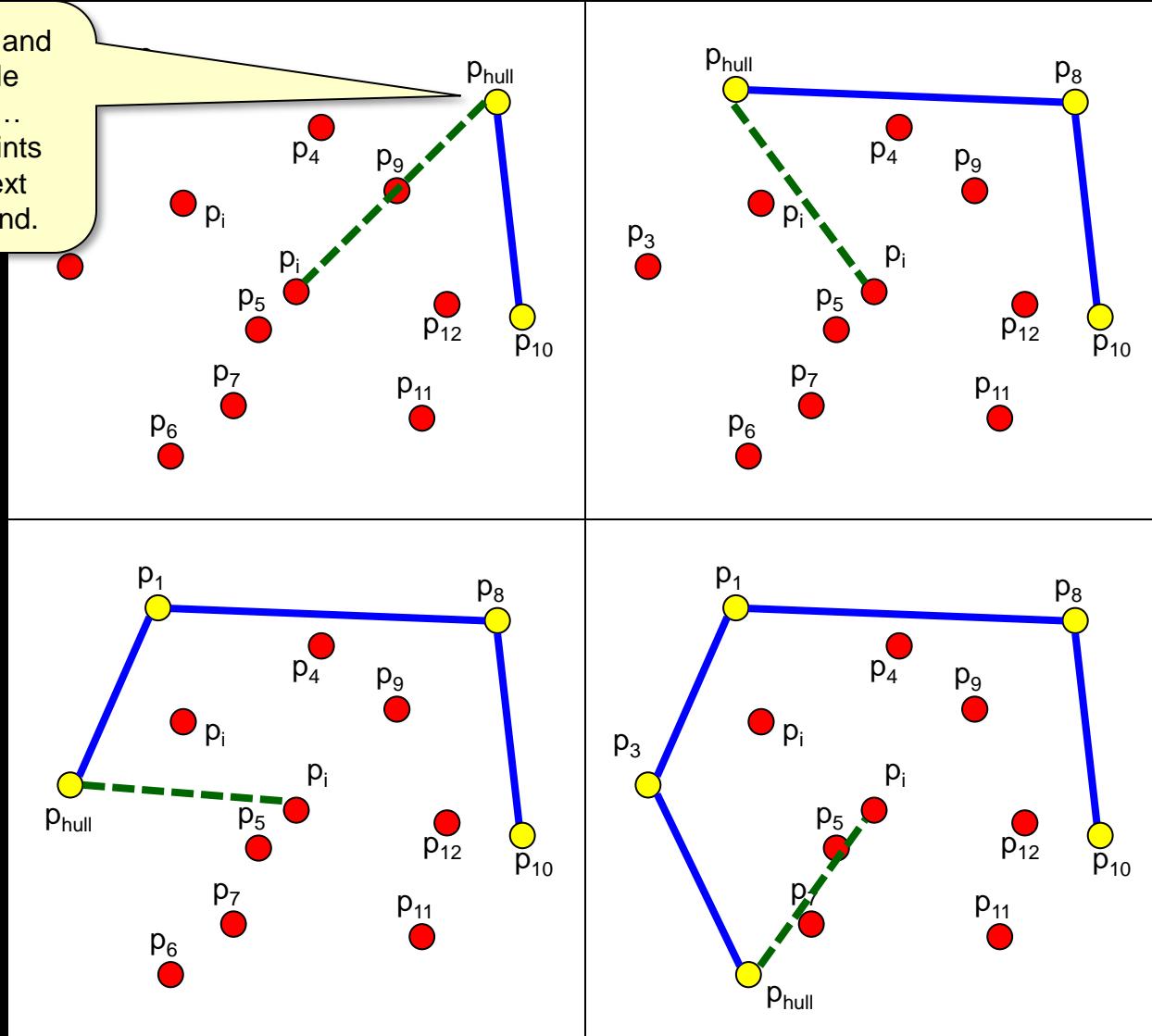


After all are checked,  $p_{end}$  will be the next point on the hull, so add it to the **hull** polygon.

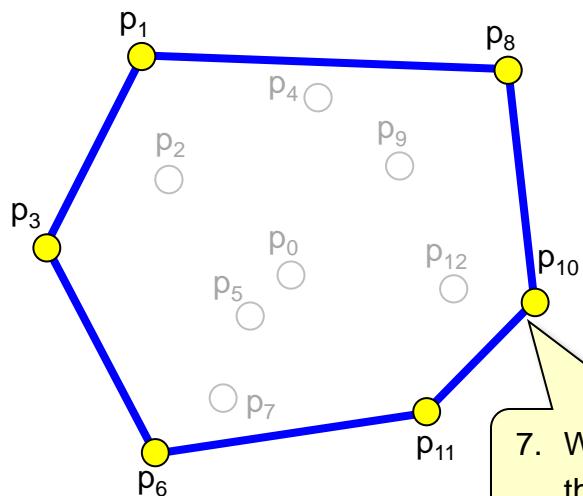
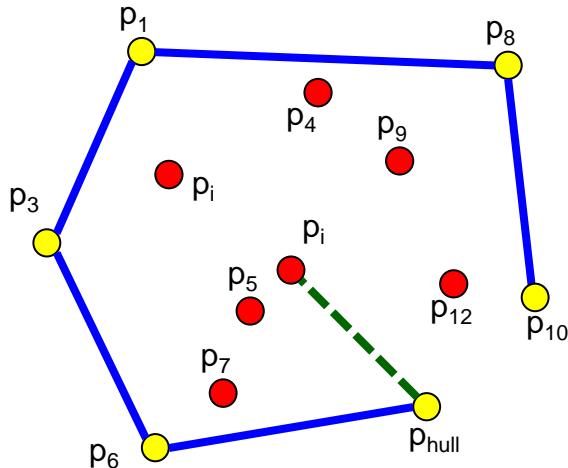


# Computing the Convex Hull

6. Set  $p_{\text{hull}} = p_{\text{end}}$  and repeat the whole process again ... checking all points in order until next hull point is found.



# Computing the Convex Hull



```

1   FUNCTION ConvexHull(PointList points)
2       hull = an empty list
3       pHull = the rightmost point in points
4       pStart = pHull
5       pEnd = NULL
6       WHILE (pEnd is not equal to pStart) DO
7           Add pHull to hull
8           pEnd = the first point in points
9           FOR (each point pi in points) DO
10              IF (pHull is the same as pEnd) THEN
11                  pEnd = pi
12              IF (angle pHull to pEnd to pi is a right turn) THEN
13                  pEnd = pi
14              pHull = pEnd
15      RETURN hull

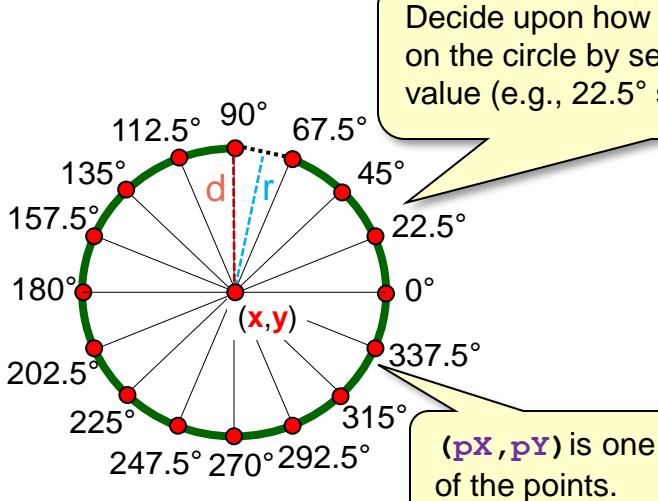
```

it is a right turn only if  
 $((p_{End_x} - pHull_x) * (p_i_y - pHull_y) - (p_{End_y} - pHull_y) * (p_i_x - pHull_x)) < 0$

7. When **p\_end** = **p\_start**  
then we are done.

# Convex Hulls Around Obstacles

- Find Convex Hull of “points circles” around obstacle vertices:



Decide upon how many points you want on the circle by setting a **DEGREE\_UNIT** value (e.g., 22.5° shown here).

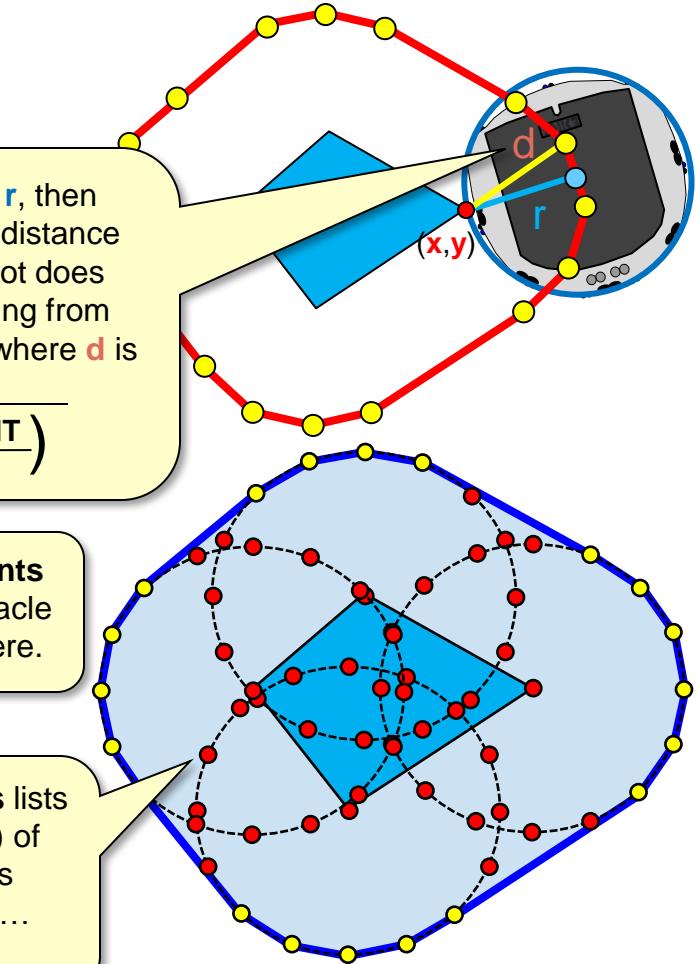
Given a robot radius of  $r$ , then points must be at least distance  $d$  from  $(x, y)$  so that robot does not collide when travelling from one vertex to another, where  $d$  is

$$\frac{r}{\cos\left(\frac{\text{DEGREE\_UNIT}}{2}\right)}$$

```
circlePoints = empty list  
  
FOR (a=0; a<360; a+=DEGREE_UNIT) DO {  
    xoff = d*cos(a)  
    yoff = d*sin(a)  
    if (xoff > 0) px = x + ceil(xoff)  
    otherwise px = x + floor(xoff)  
    if (yoff > 0) py = y + ceil(yoff)  
    otherwise py = y + floor(yoff)  
    add point (px, py) to circlePoints  
}
```

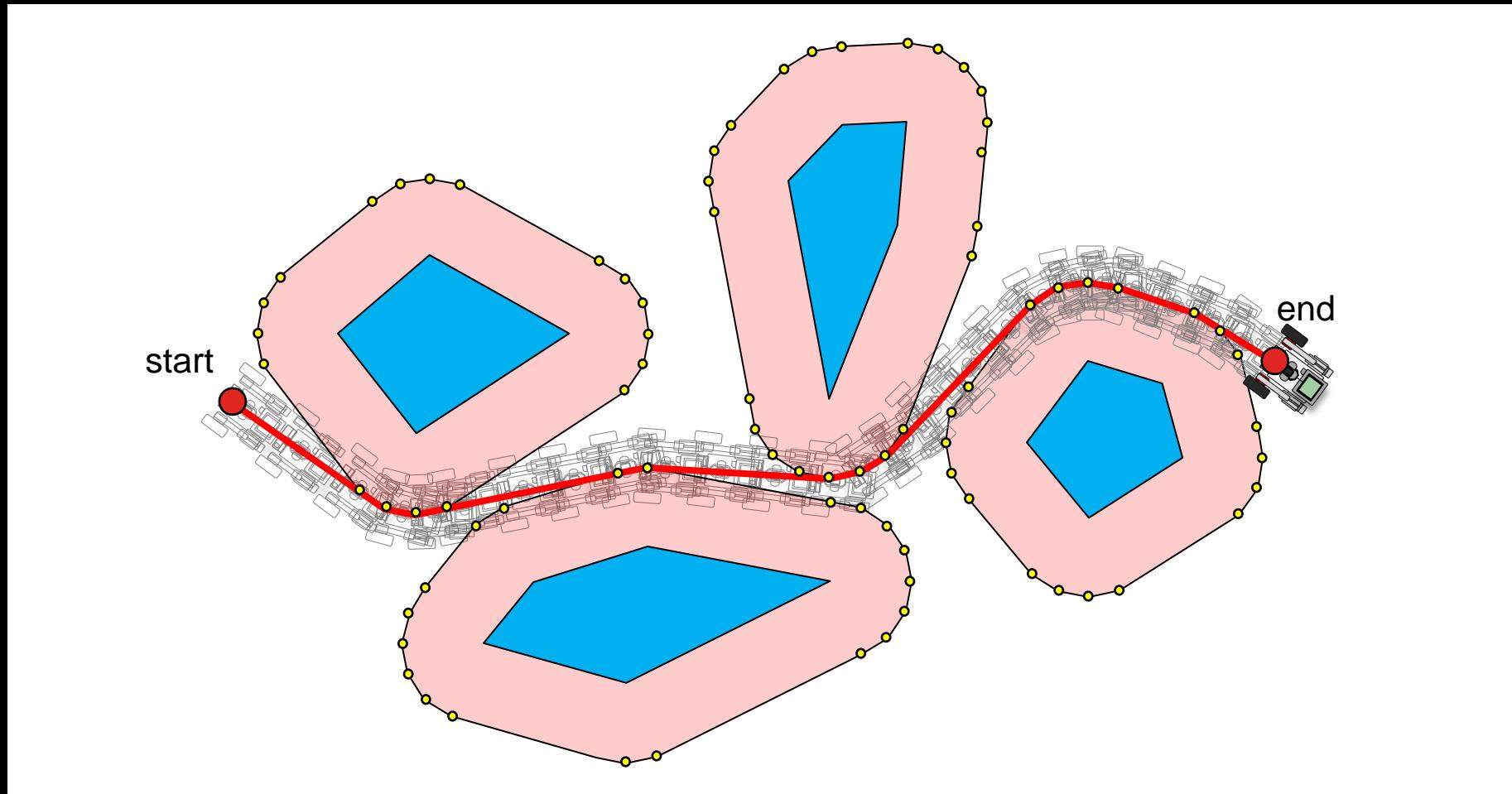
Create a list of **circlePoints** centered around an obstacle vertex  $(x, y)$  as shown here.

Merge all **circlePoints** lists from each vertex  $(x, y)$  of the obstacle. Using this “merged” list of points ... find the convex hull.



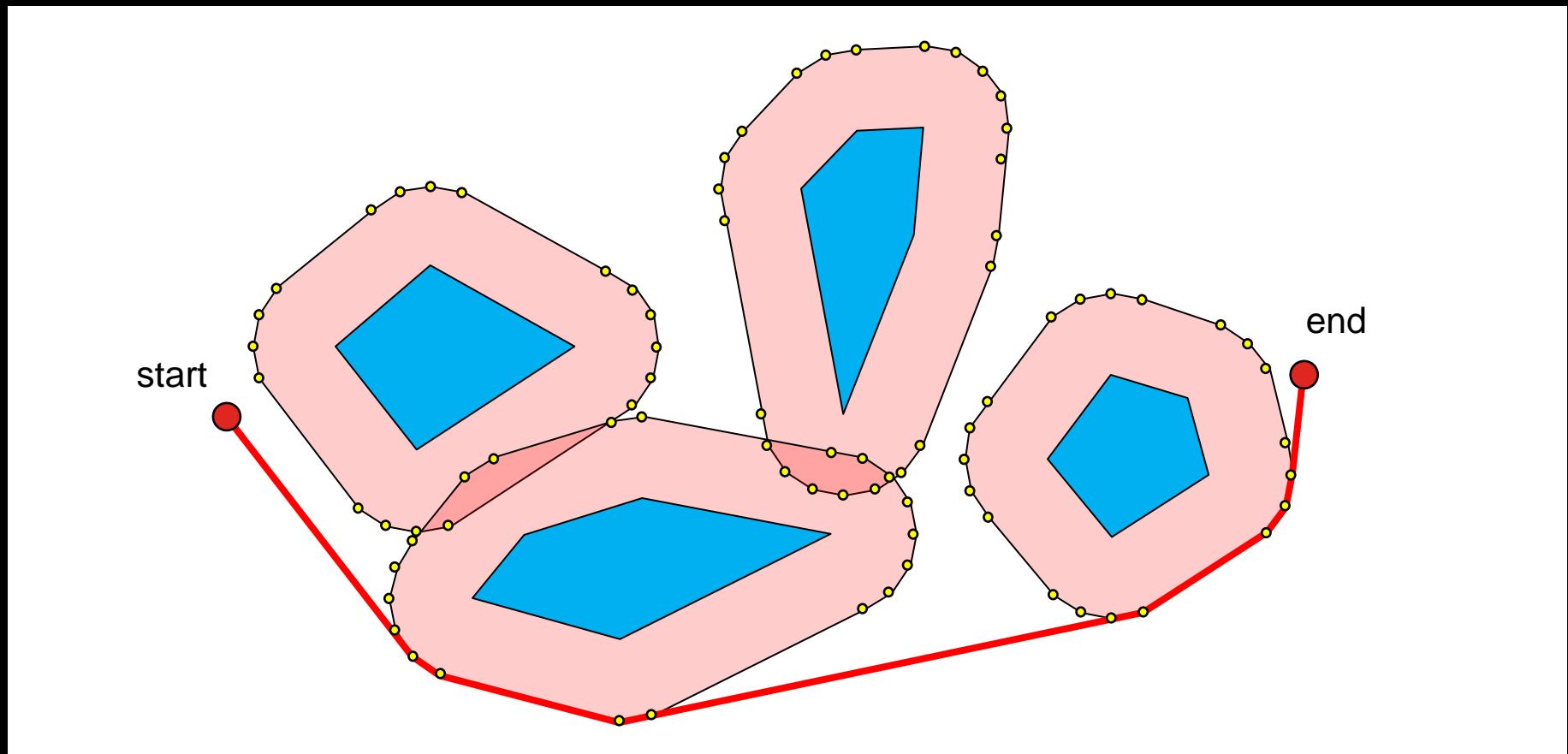
# Real Robot Shortest Path

- Now the robot has collision-free travel



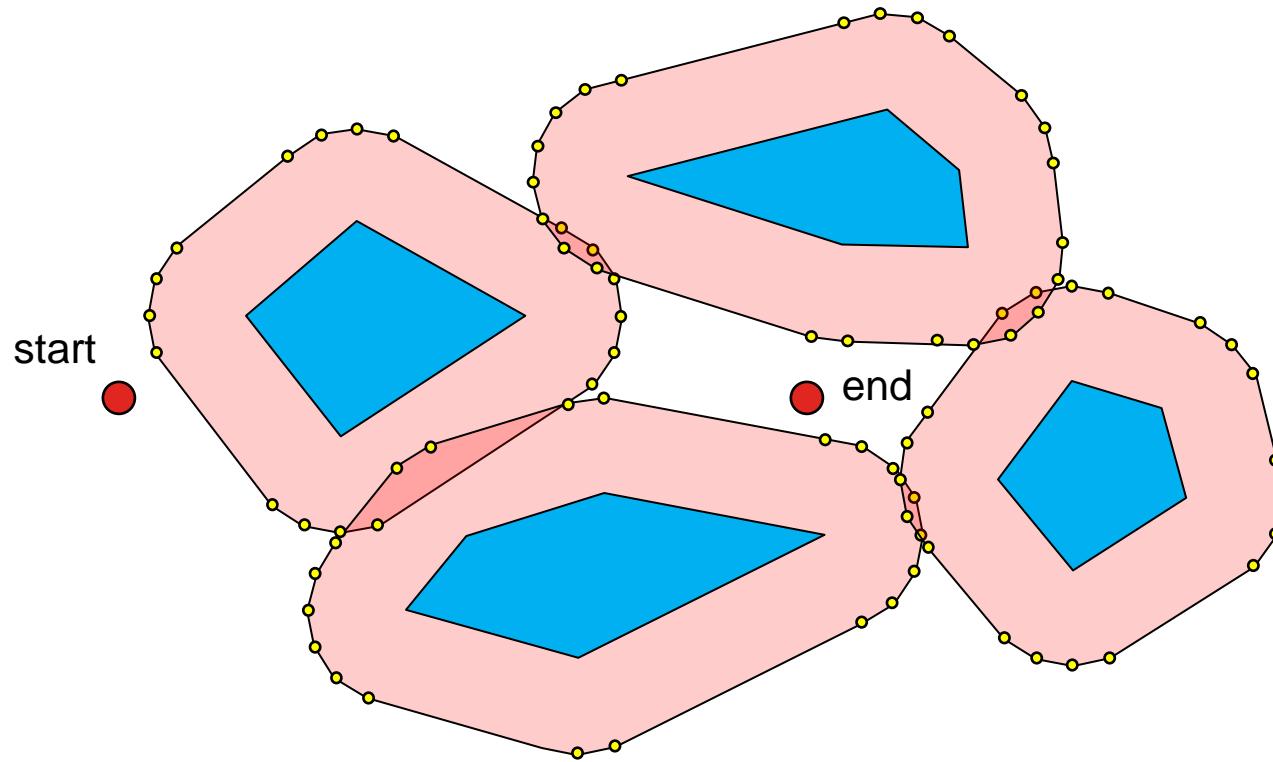
# Real Robot Shortest Path

- In some cases, the grown obstacles will intersect, resulting in a different solution path.



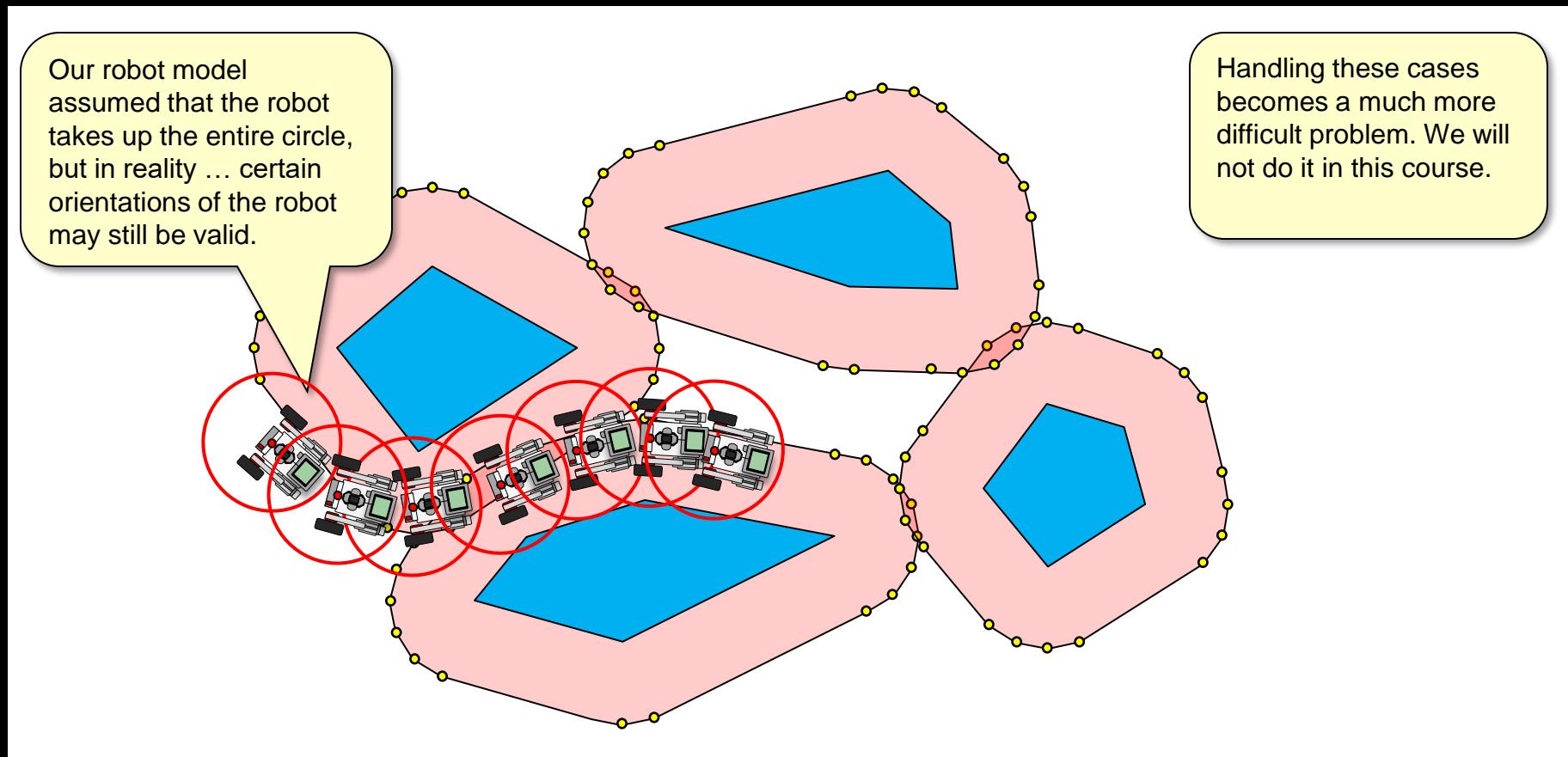
# Real Robot Shortest Path

- In some cases, there may even be no solution !



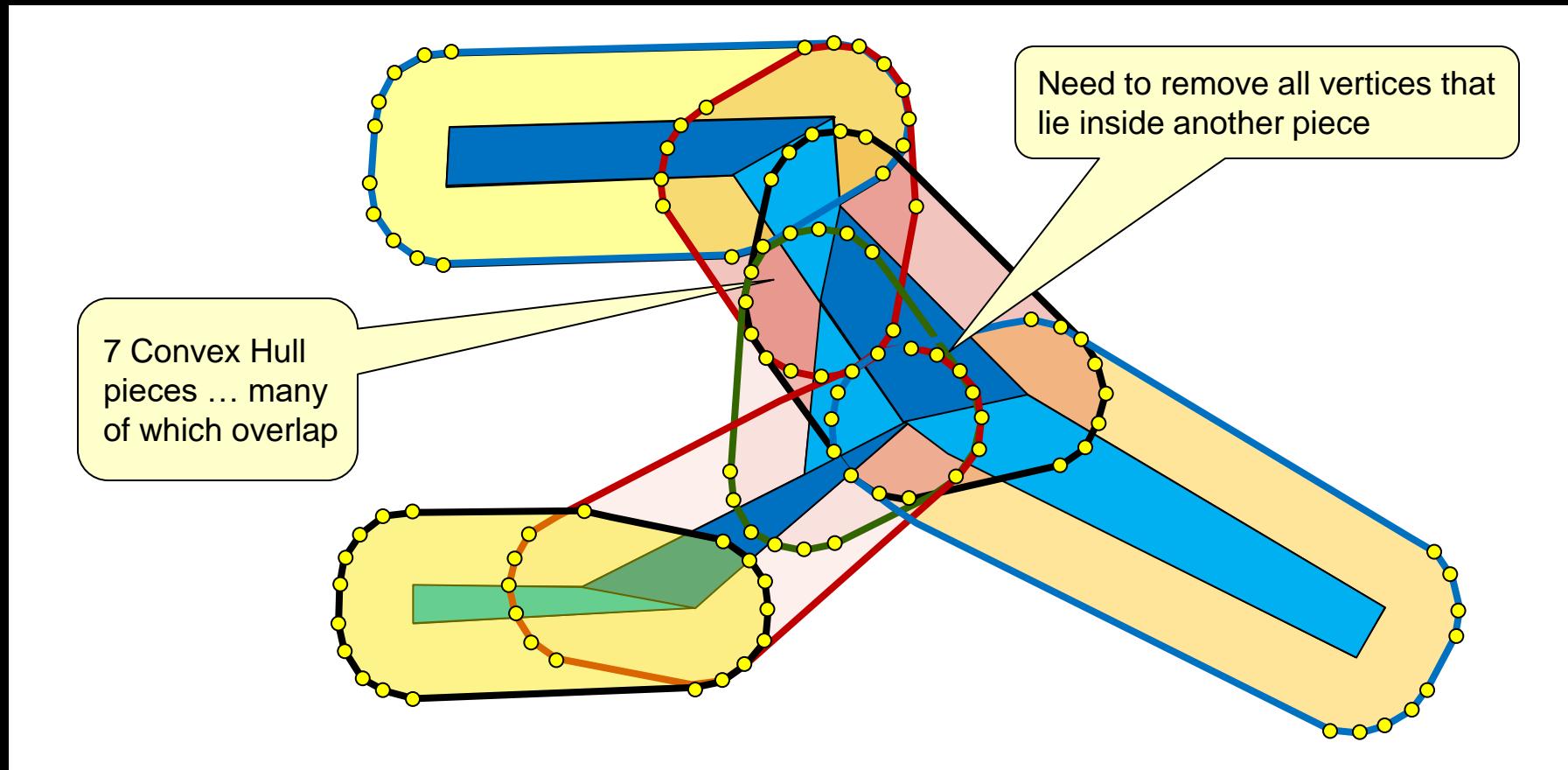
# Real Robot Shortest Path

- However, this is a restriction on our robot model and algorithm, as actual robot can “fit”:



# Non-Convex Solution

- Compute Convex Hull for each piece:



# Non-Convex Challenges

- Compute visibility graph ... but it is tricky:

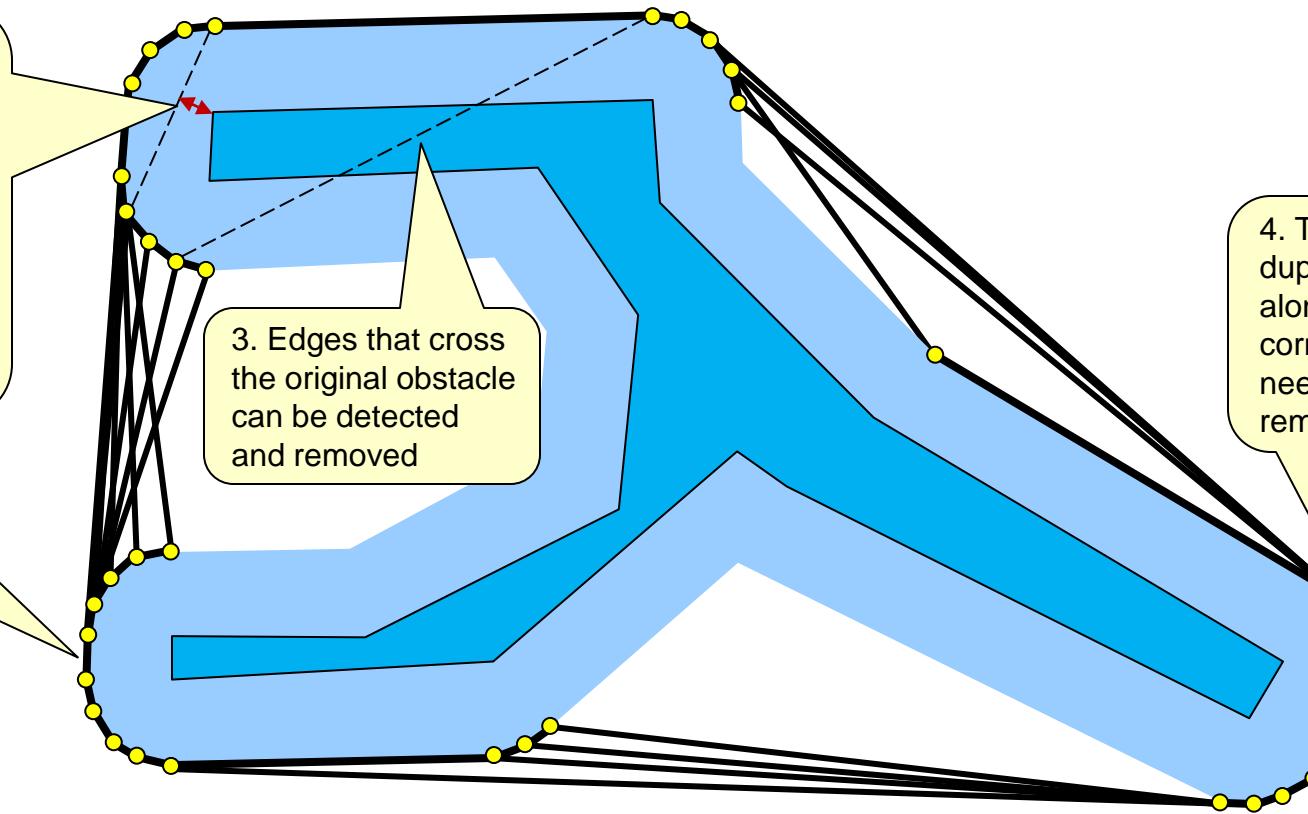
Unfortunately, there are a lot of intersection-related problems that pop up due to the overlapping of the various convex pieces. This will require the coding of special cases.

1. Edges that cut through the grown obstacle like this. They can be found and eliminated by removing all edges that are within the robot's radius from any vertex of the original obstacle.

2. Need to ensure that all edges along the corners are in the graph.

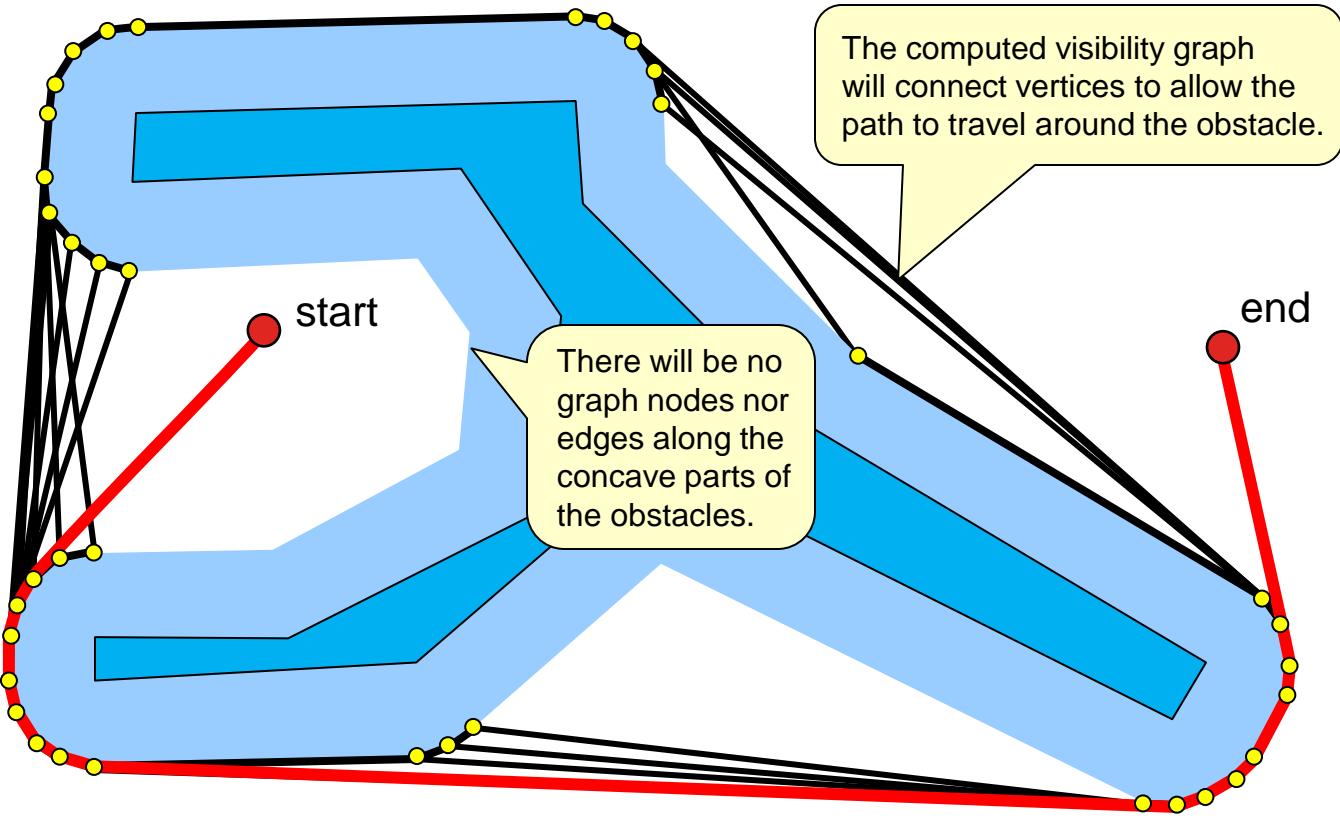
3. Edges that cross the original obstacle can be detected and removed

4. There may be duplicate nodes along the corners that need to be removed.



# Non-Convex Shortest Path

- If done correctly, the shortest path can be found:

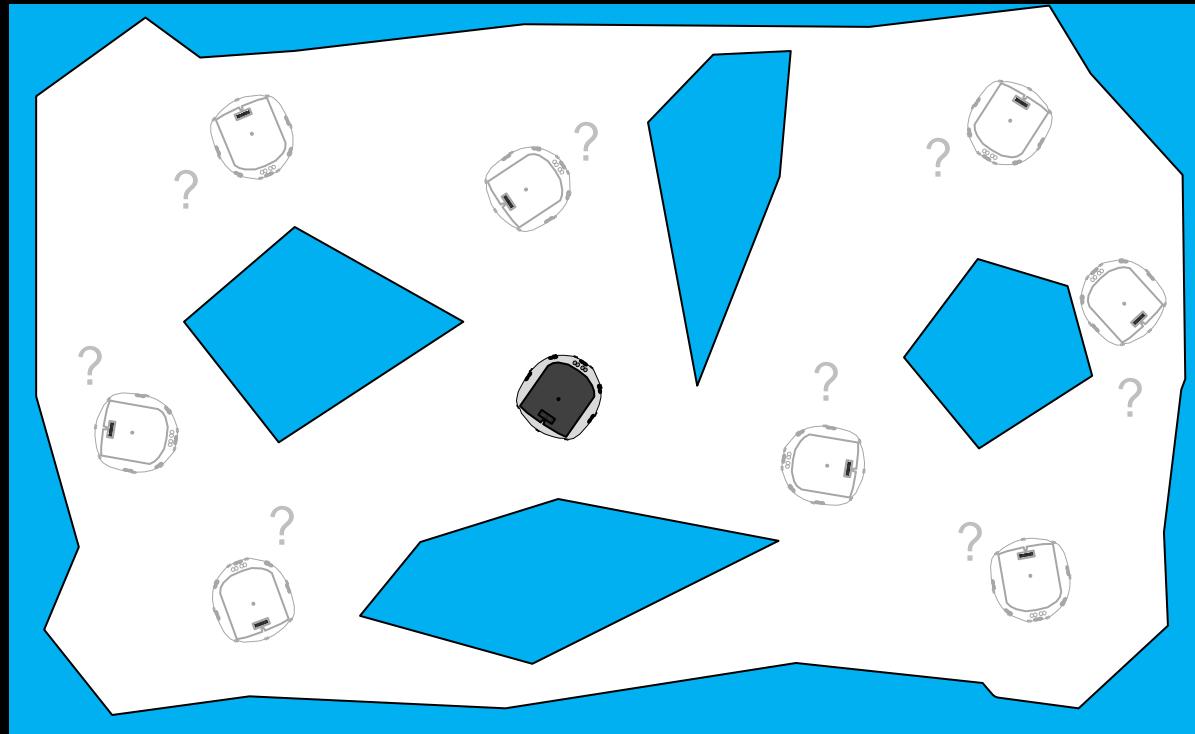


Start the  
Lab ...

# Localization

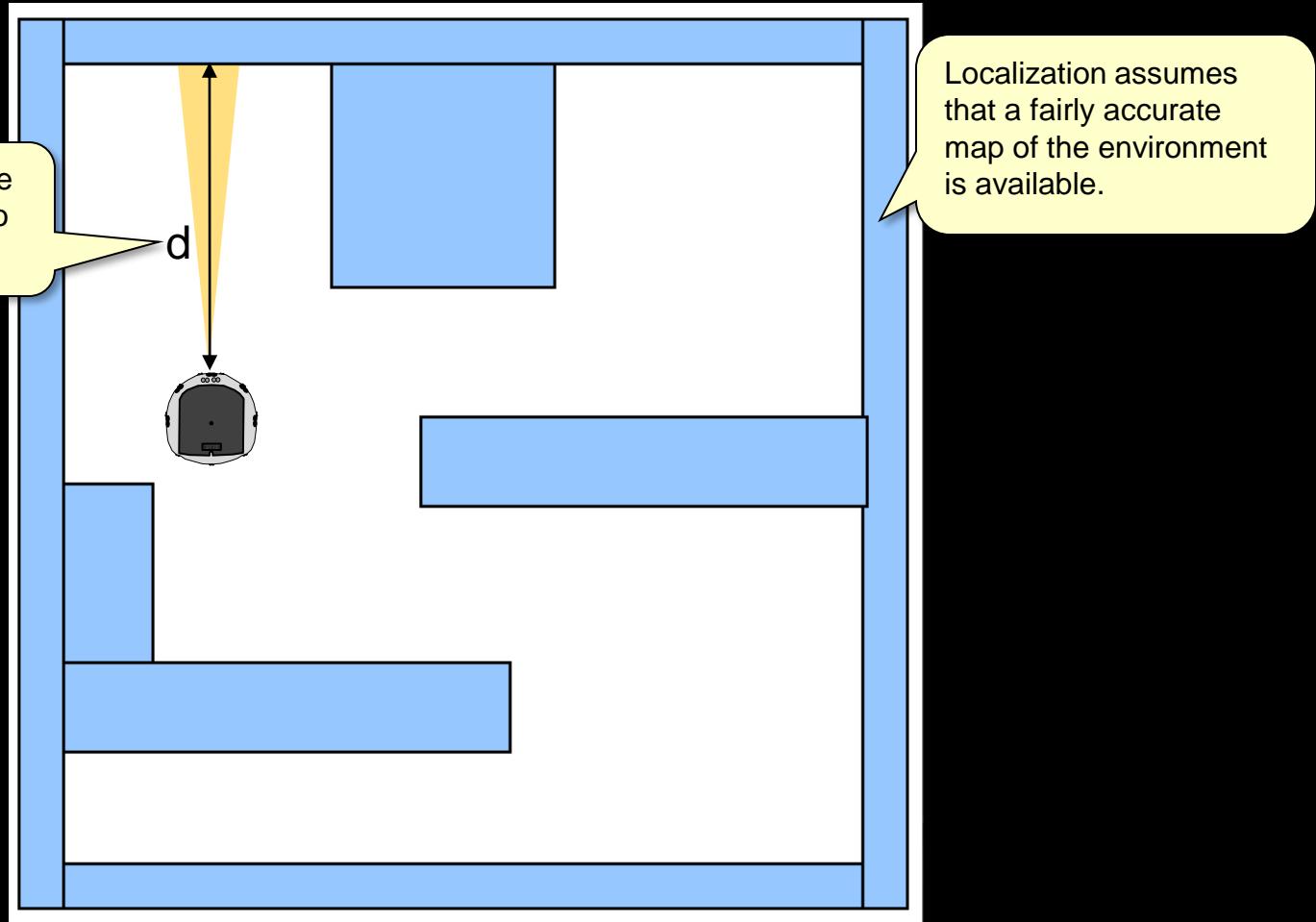
# Localization

- Given a robot and a map, find the robot's pose (i.e., where the robot is and which direction it faces)
  - There are many algorithms ... we will look at Monte Carlo Estimation (a.k.a. Particle Filtering)



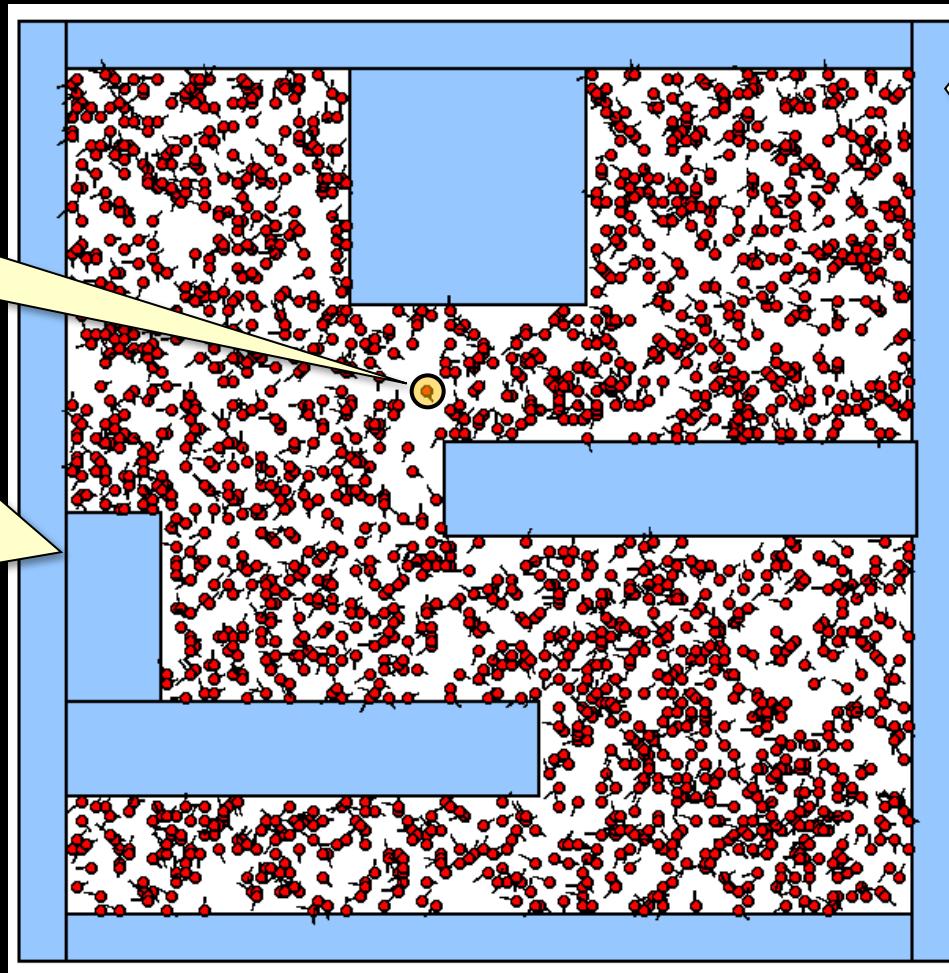
# Monte Carlo Localization

- Assume robot is entirely enclosed within its environment at all times.



# Monte Carlo Localization

- Start by computing a LOT (i.e., 2000 used here) of random poses as *initial estimates* of where the robot is.



For explanation purposes, assume that the real robot is actually here.

Each red circle indicates the “potential location” of the robot.



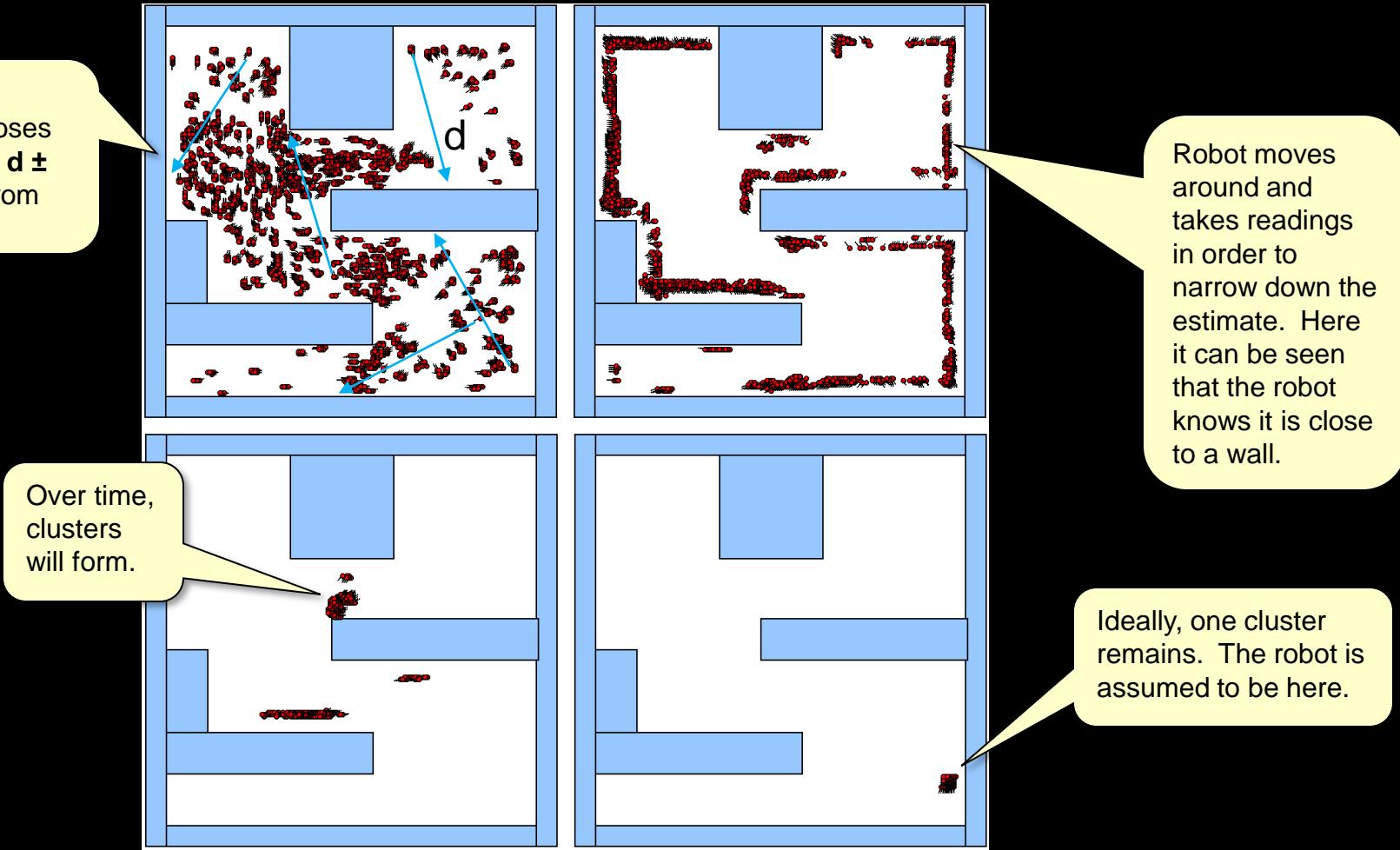
The black line indicates the orientation of the robot for that potential pose.

Need to check to ensure that no pose estimate lies **inside** of an obstacle, otherwise discard it.

Need to also check to ensure that no pose estimate lies outside of the boundary (just check against **minX**, **minY**, **maxX** and **maxY** of all obstacles).

# Monte Carlo Localization

- As time goes on, many of these poses will be discarded and replaced by “better” estimates.



# Monte Carlo Localization

- Algorithm is as follows:

```
1  FUNCTION Localize(ArrayList<Obstacle> obstacles, NUM_SAMPLES, %TOL)
2      currentEstimates = a list of NUM_SAMPLES randomly chosen poses
3      REPEAT {
4          message = getMessageFromRobot()
5          IF (message is a new distance reading) THEN
6              d = getDistanceReading()
7              estimateFromReading(currentEstimates, d, obstacles, NUM_SAMPLES, %TOL)
8          IF (message is a motion update from forward movement) THEN
9              distance = getDistanceMovedForward()
10             updateLocation(currentEstimates, distance, %TOL)
11         IF (message is a motion update from turning) THEN
12             angle = getAngleTurned()
13             updateOrientation(currentEstimates, angle, %TOL)
14     }
```

Wait until the robot sends some new information ...

Now re-compute the **currentEstimates** based on this latest distance reading.

If robot moved forward, move all estimates forward by that amount.

If robot turned, turn all estimates by that amount.

# Monte Carlo Localization

- When a distance reading is obtained ... update estimates:

```
1  FUNCTION estimateFromReading(currentEstimates, d, obstacles, NUM_SAMPLES, %TOL)
2      goodEstimates = an empty list
3      FOR (each pose p in currentEstimates) DO
4          IF (isGoodEstimate(p, d, obstacles, %TOL)) THEN
5              Add p to goodEstimates if it is a valid pose
6      IF (goodEstimates is empty)
7          currentEstimates = reset to NUM_SAMPLES random poses
8      OTHERWISE
9          Call resetEstimates()
10         c = NUM_SAMPLES / number of poses in goodEstimates
11         Clear the currentEstimates list
12         Add c copies of each pose in goodEstimates to currentEstimates
```

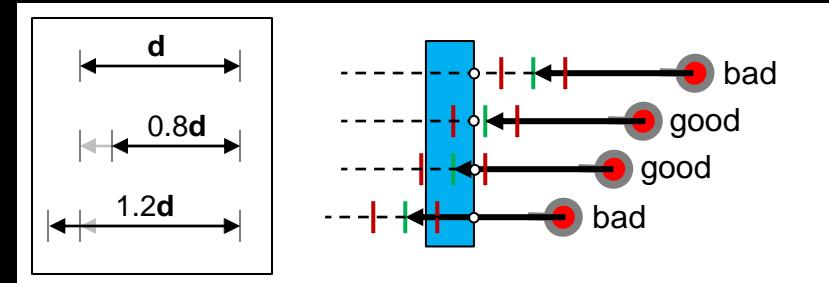
Go through the poses and keep only the “good” ones. Check also that it is valid (i.e., it does not lie within an obstacle).

Duplicate the good ones so that we always have a constant number of samples.

To make a copy, you need to call the constructor: `new Pose(p.x, p.y, p.angle)`

- How do we know if an estimate is “good” ?

- Check intersection with closest obstacle. If within reasonable error tolerance (e.g., 20%), then it is a good estimate.



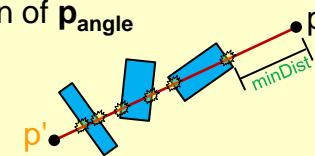
# Is the Estimate Good?

```

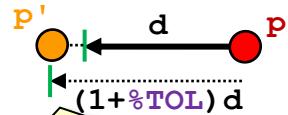
1   FUNCTION isGoodEstimate( $p_x, p_y, d, \text{obstacles}, \%TOL$ )
2       Compute point  $p'$  to be  $(p_x + (1+\%TOL) d \cdot \cos(p_{\text{angle}}), p_y + (1+\%TOL) d \cdot \sin(p_{\text{angle}}))$ 
3        $\text{minDist} = \text{infinity}$ ;
4       FOR (each obstacle  $\text{obj}$  in  $\text{obstacles}$ ) DO
5           FOR (each edge  $e$  of  $\text{obj}$ ) DO
6               IF ( $\overline{pp'}$  intersects  $e$ ) THEN
7                    $q = \text{intersection of } \overline{pp'} \text{ with } e$ 
8                   IF ( $q$  is not  $\text{null}$ )
9                        $qDist = \text{distance from } p \text{ to } q$ 
10                      IF ( $qDist < \text{minDist}$ ) THEN
11                           $\text{minDist} = qDist$ 
12
13      IF ( $\text{minDist}$  is still infinity) THEN
14          RETURN false
15
16      IF ( $\text{minDist} > (1-\%TOL)d \text{ AND } \text{minDist} < (1+\%TOL)d$ ) THEN
17          RETURN true
18      RETURN false

```

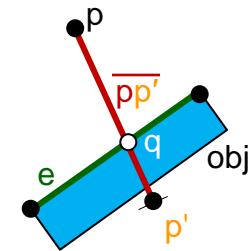
Minimum distance from  $(p_x, p_y)$  to an obstacle in the direction of  $p_{\text{angle}}$



Find distance to closest obstacle intersection point  $q$  among all obstacle edges.



$p'$  is  $\%TOL$  further than  $d$  from  $(p_x, p_y)$  in direction of  $p_{\text{angle}}$



If no intersection, estimate is bad.

If reading is within  $\%TOL$  range, it is a good estimate.

# Monte Carlo Localization

- Updating estimates when a robot moves forward ...

```
1  FUNCTION updateLocation(currentEstimates, d, %TOL)
2      FOR (each pose p in currentEstimates) DO
3          randomPercent = %TOL · (2R - 1)
4          px = px + d · (1 + randomPercent) · cos(pangle)
5          py = py + d · (1 + randomPercent) · sin(pangle)
```

Move forward by distance **d** cm plus a random amount from  $-\%TOL$  to  $+\%TOL$

**R** is a random number such that  $0 \leq R < 1$ .

The randomness is needed so as to allow the poses to vary slightly over time. This is crucial for the algorithm to work, otherwise it may never converge to a proper pose or may lose a good pose due to inaccurate robot movements.

**Math.random()** gives a random # in range from 0 to 0.9999999

- Updating estimates when a robot spins ...

```
1  FUNCTION updateOrientation(currentEstimates, angle, %TOL)
2      FOR (each pose p in currentEstimates) DO
3          randomPercent = %TOL · (2R - 1)
4          pangle = pangle + angle · (1 + randomPercent)
```

Turn by **angle** degrees plus a random amount from  $-\%TOL$  to  $+\%TOL$

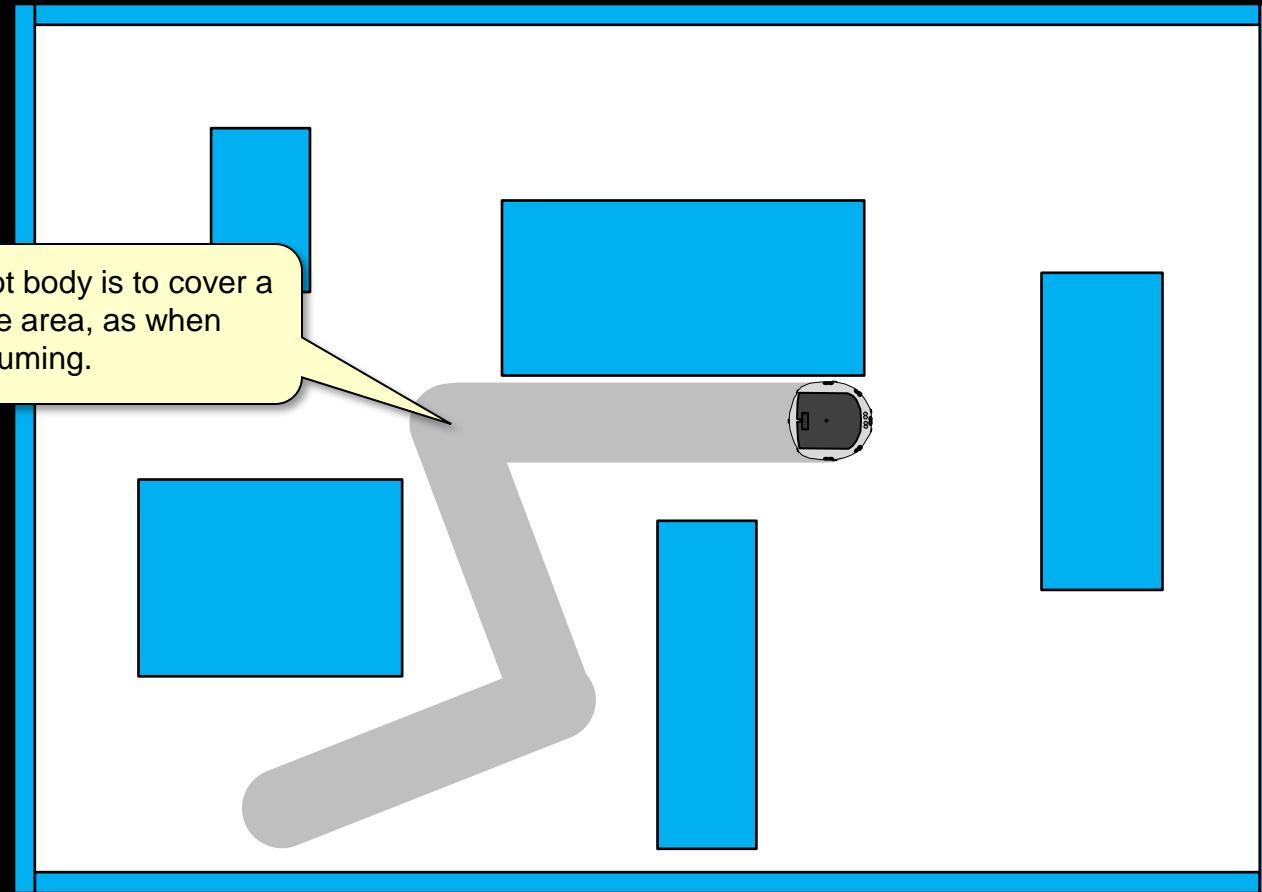
Start the  
Lab ...

# Area Coverage Paths

# Area Coverage



- How do we get a robot to cover the whole area of an environment ?

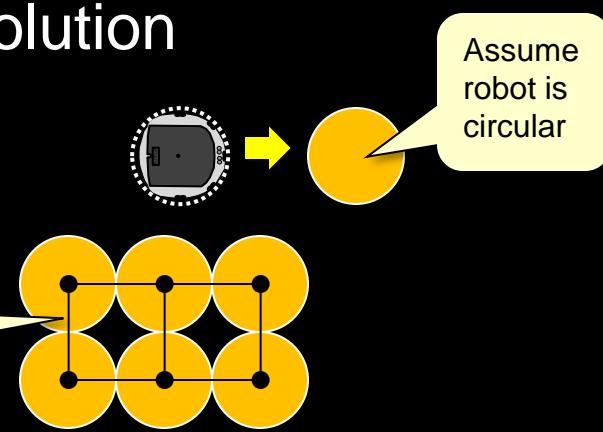


# Area Coverage

- No simple perfect solution
  - Many algorithms of various complexities
- Main **goals** are:
  - cover all reachable areas of environment
  - minimize amount of overlapping
- Robots are not perfectly accurate, approximate solution is ok
- We will discuss a rectangular grid-based solution
  - will provide an approximation only
  - will not attempt to minimize overlap

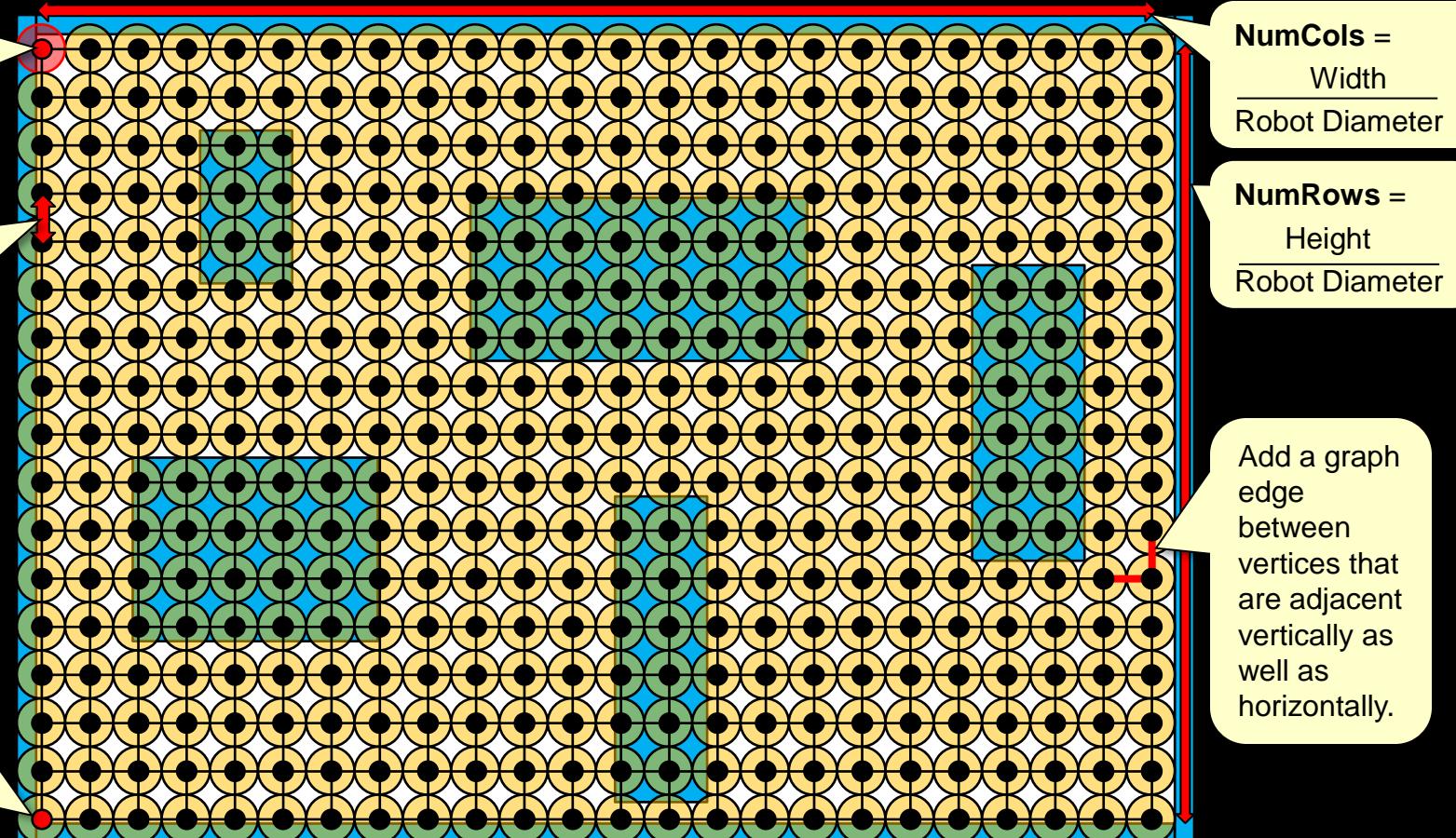


We will lay a grid of potential robot locations across the environment and then form a graph to travel along.



# Grid Creation

- Overlay a 2D grid (i.e., graph of nodes & edges) of “robot-sized” circles touching adjacently vertically and horizontally:



# Grid Creation Pseudocode

- The code for creating the grid is basic:

```
g = an empty graph

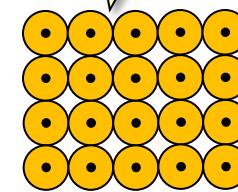
numRows = EnvironmentHeight / ROBOT_DIAMETER
numCols = EnvironmentWidth / ROBOT_DIAMETER

nodes = an empty 2D array that is numRows x numCols in size
FOR each row r DO
    FOR each column c DO
        nodes[r][c] = new node centered at that r and c
        add nodes[r][c] to g

FOR each row r DO
    FOR each column c (except the last one) DO
        add edge in g from add nodes[r][c] to nodes[r][c+1]

FOR each row r (except the last one) DO
    FOR each column c DO
        add edge in g from add nodes[r][c] to nodes[r+1][c]
```

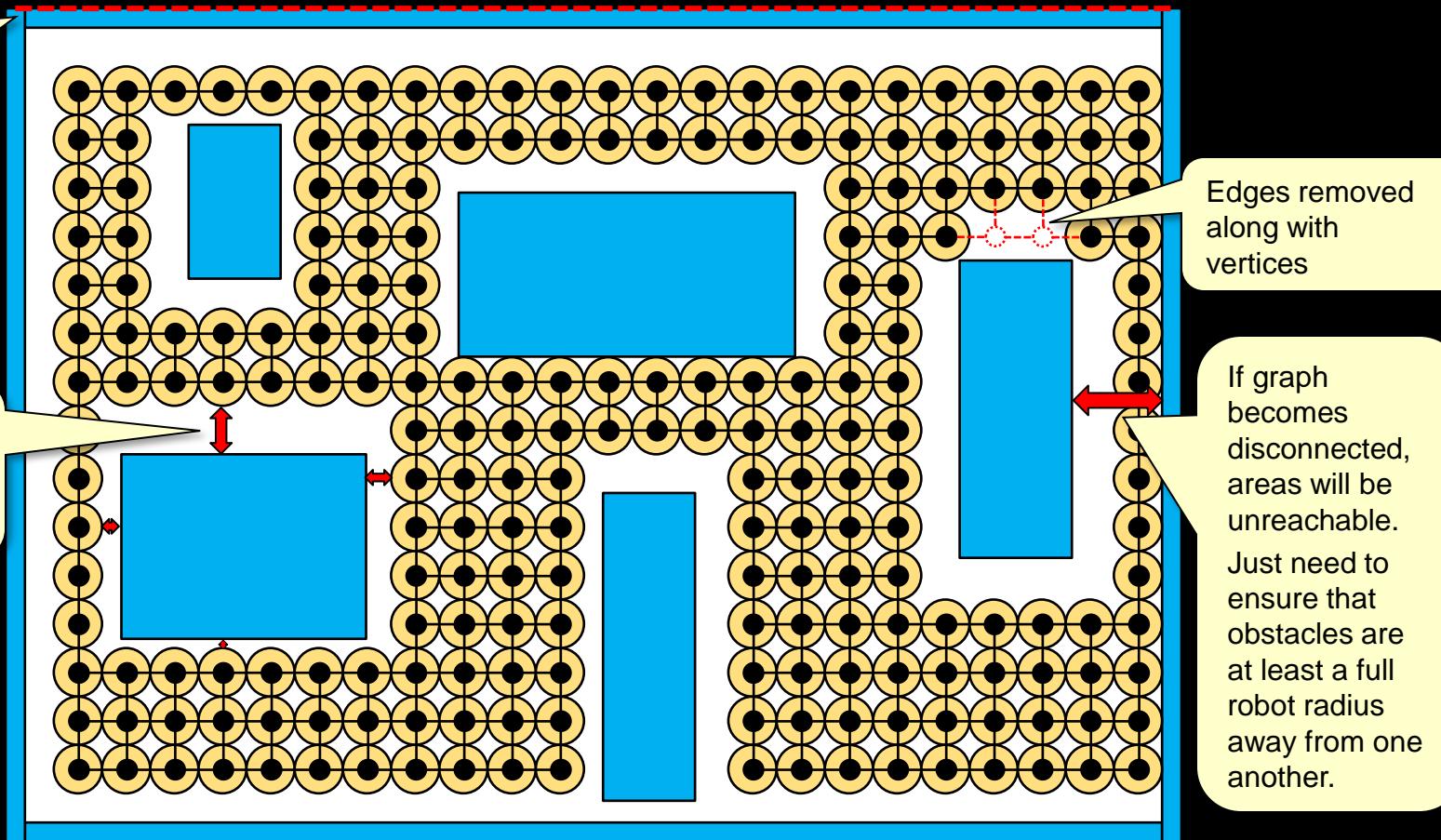
Distance between vertices is robot diameter.



# Grid Reduction

- Remove all vertices that represent robot positions that intersect with any obstacles:

No vertices past the boundaries now.



# Grid Reduction Pseudocode

- The reduced graph simply requires you to check if a node's circle intersects an obstacle:

```
FOR each node n of the graph DO {
    FOR each obstacle obj in the environment DO {
        IF the n's center lies within the obstacle THEN
            mark n as invalid
        FOR each edge e of obj DO {
            IF distance from n's center to e is <= ROBOT_RADIUS THEN
                mark n as invalid
        }
    }
    FOR each invalid node n DO
        remove n from the graph
```

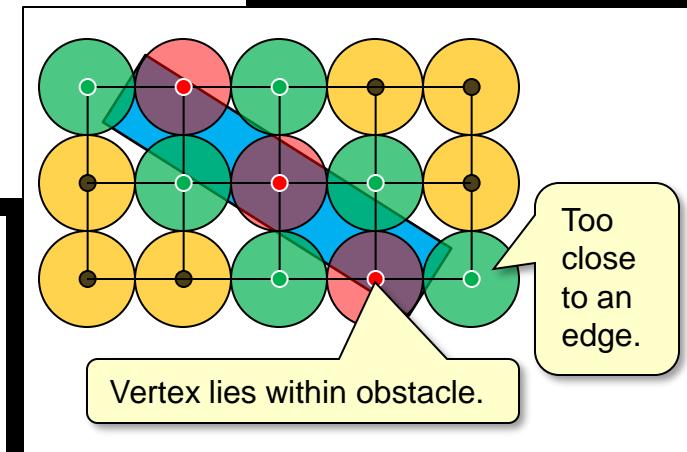
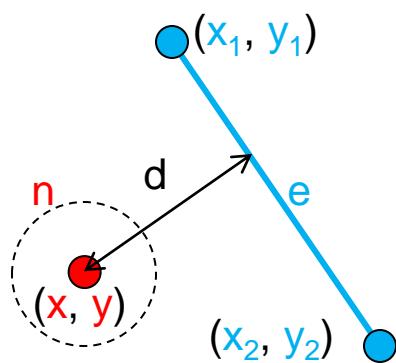
$$\text{calculate } t = - \frac{(x_1 - x)(x_2 - x_1) + (y_1 - y)(y_2 - y_1)}{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

IF  $(0 \leq t \leq 1)$  THEN

$$d = \frac{|(x_2 - x_1)(y_1 - y) - (y_2 - y_1)(x_1 - x)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

OTHERWISE  $d$  is the smallest of these two:

$$\sqrt{(x_1 - x)^2 + (y_1 - y)^2}$$
$$\sqrt{(x_2 - x)^2 + (y_2 - y)^2}$$



# Spanning Tree

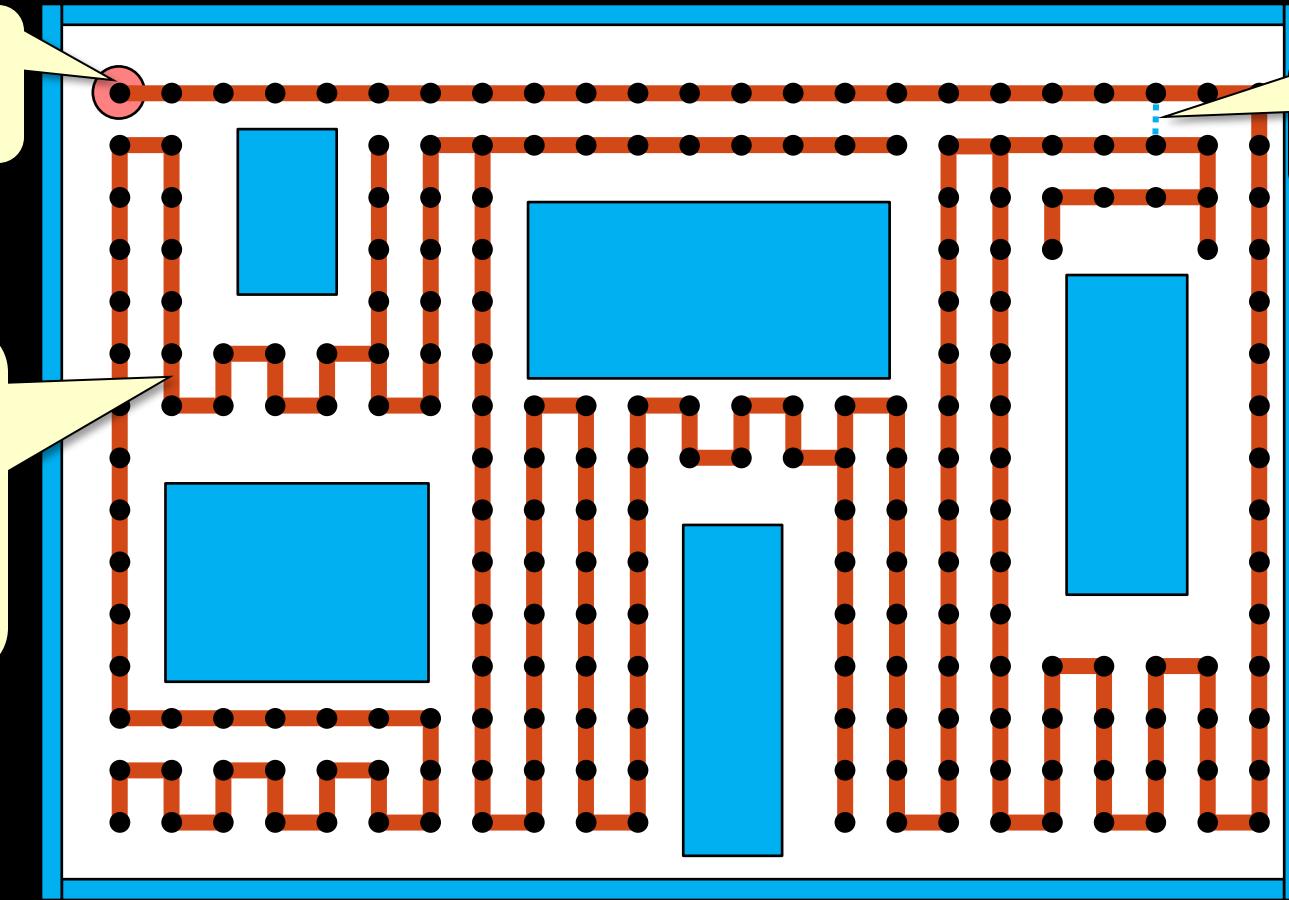
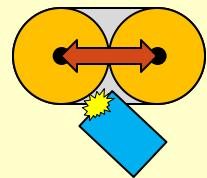
- Compute a *spanning tree* in the graph (shown in red):
  - represents a path that covers all the vertices.

In this example,  
this is root of the  
spanning tree.

There are  
many possible  
spanning trees.  
This one  
follows an up,  
right, down, left  
ordering  
traversal.

Many graph  
edges have been  
removed

This algorithm  
assumes that no object lies  
between  
adjacent vertex  
circles so that  
robot can travel  
between them  
without collision:



# Spanning Tree Pseudocode

- Spanning tree can be any traversal of the graph that reaches all nodes. It will depend on which edges are traversed first at each vertex.

```
computeSpanningTree(G) {  
    FOR each node n of graph G DO  
        mark n as "not visited"  
    startNode = any node in G  
    dummyEdge = an edge from startNode to itself  
    computeSpanningTreeFrom(startNode, dummyEdge)  
  
    edges = all edges that are not marked as part of tree  
    FOR each edge e in edges DO  
        remove e from G  
}
```

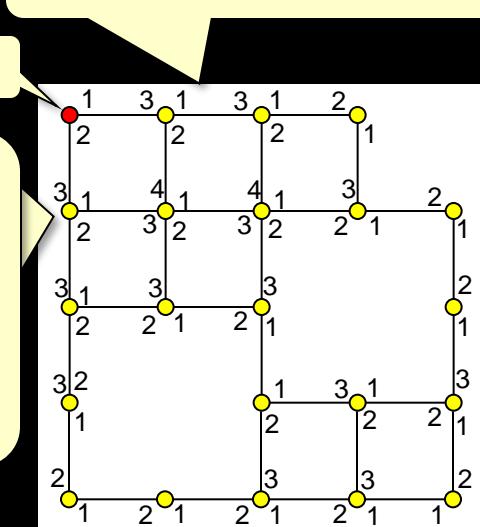
```
computeSpanningTreeFrom(aNode, incomingEdge) {  
    IF aNode was already visited THEN  
        RETURN  
  
    mark aNode as "visited"  
    mark incomingEdge as part of the spanning tree  
  
    FOR each edge e connected to aNode DO  
        otherNode = node of edge e that is not aNode  
        computeSpanningTreeFrom(otherNode, e)  
}
```

By marking a node as “visited”, we can avoid processing that node again and this is essential to stop the recursion.

startNode

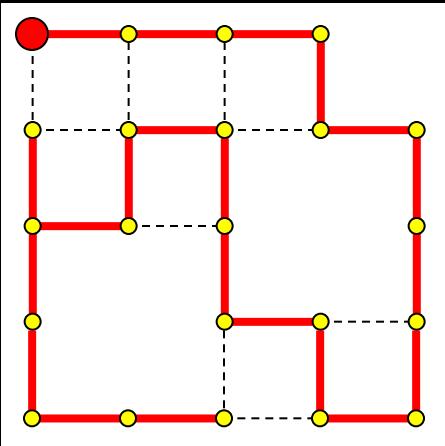
Each vertex has its edges in some order. Neighbours are visited in that order.

The ordering of edges in this example is right, down, left, up.

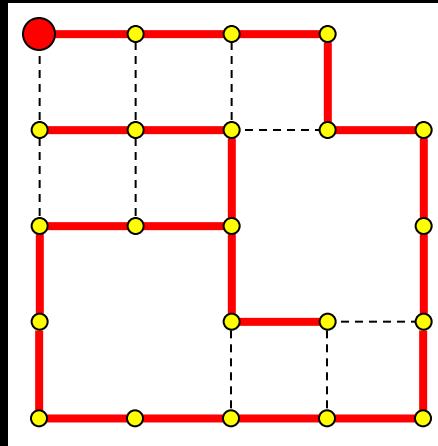


# Various Spanning Trees

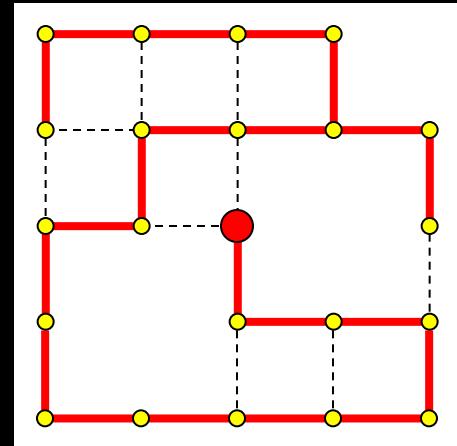
Up/Right/Down/Left



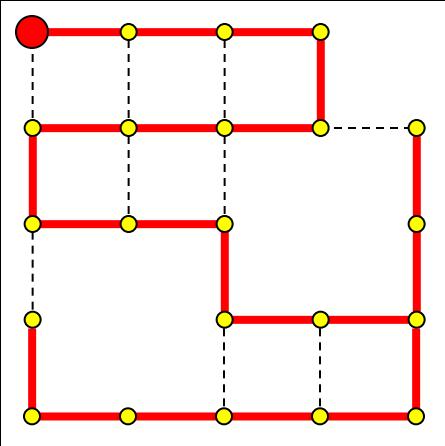
Right/Down/Left/Up



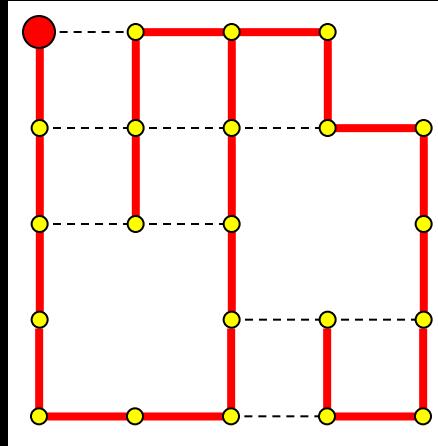
Right/Down/Left/Up



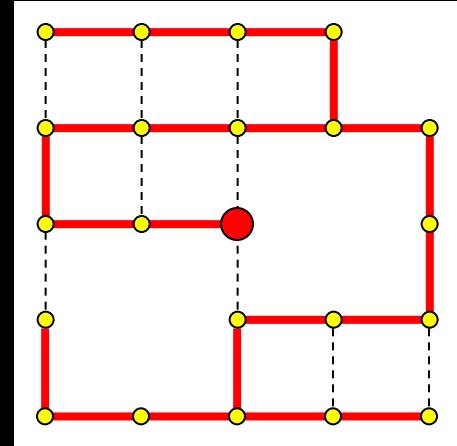
Left/Right/Up/Down



Up/Down/Left/Right

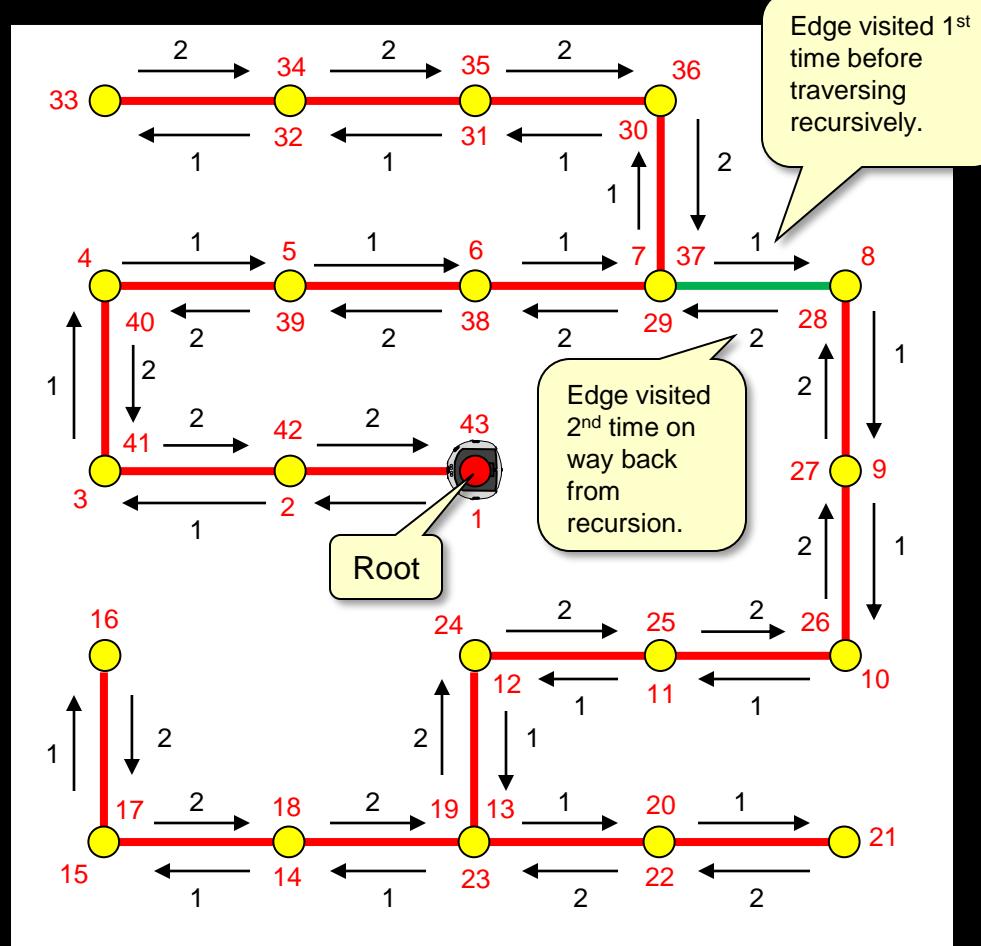


Left/Right/Up/Down



# Spanning Tree Traversal

- Need to compute a path in the spanning tree, recursively.
- Each edge needs to be travelled on twice, which means that each vertex needs to be added to the path as many times as it has edges.
- Path in this example will have 43 points on it.
- Robot travels from point to point.



# Spanning Tree Traversal

- To travel along spanning tree, we need to compute a path:

We build up a **path** in the tree which will represent the ordering of nodes to visit by the robot.

```
computeSpanningTree(G) {  
    ... Compute spanning as before  
  
    FOR each node n of graph G DO  
        set previous of n to NULL  
        set path to be an empty list  
        dummyEdge = an edge from startNode to itself  
        computeCoveragePathFrom(startNode, dummyEdge)  
    }  
}
```

Instead of "visited" boolean, keep the previous node that is the node's parent in the spanning tree.

Each time we arrive at a **aNode** that has no previous value set, it is a new node in the path, so add it to the **path**.

```
computeCoveragePathFrom(aNode, incomingEdge) {  
    IF previous of aNode is not NULL THEN  
        RETURN  
  
    add aNode's location to the path  
  
    set previous of aNode to node at other end of incomingEdge  
  
    FOR each neighbour neigh of aNode DO {  
        edge = the edge that connects aNode and neigh  
        computeCoveragePathFrom(neigh, edge)  
    }  
    add location of aNode's previous to the path  
}  
}
```

**path** is a global variable

We will assume that the robot can travel from its start location to the start node without collision.

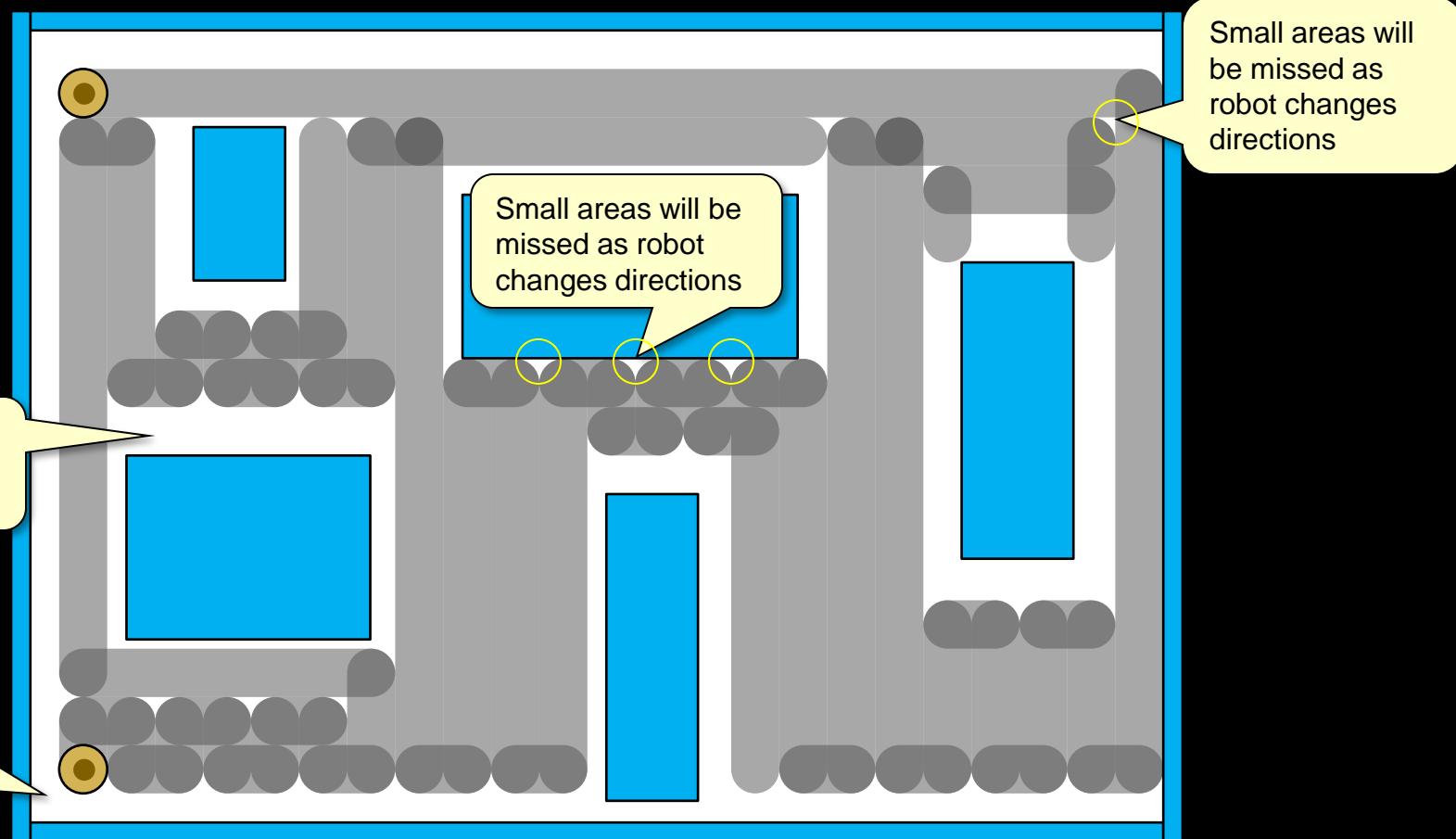
Set **aNode**'s previous to be the other end of the edge that we came in on.

Make sure that the path goes back to the previous node.

Ensures that we are following spanning tree on way back from recursion.

# Spanning Tree Coverage

- Traveling along the spanning tree will cover most of the environment (except around borders of obstacles):

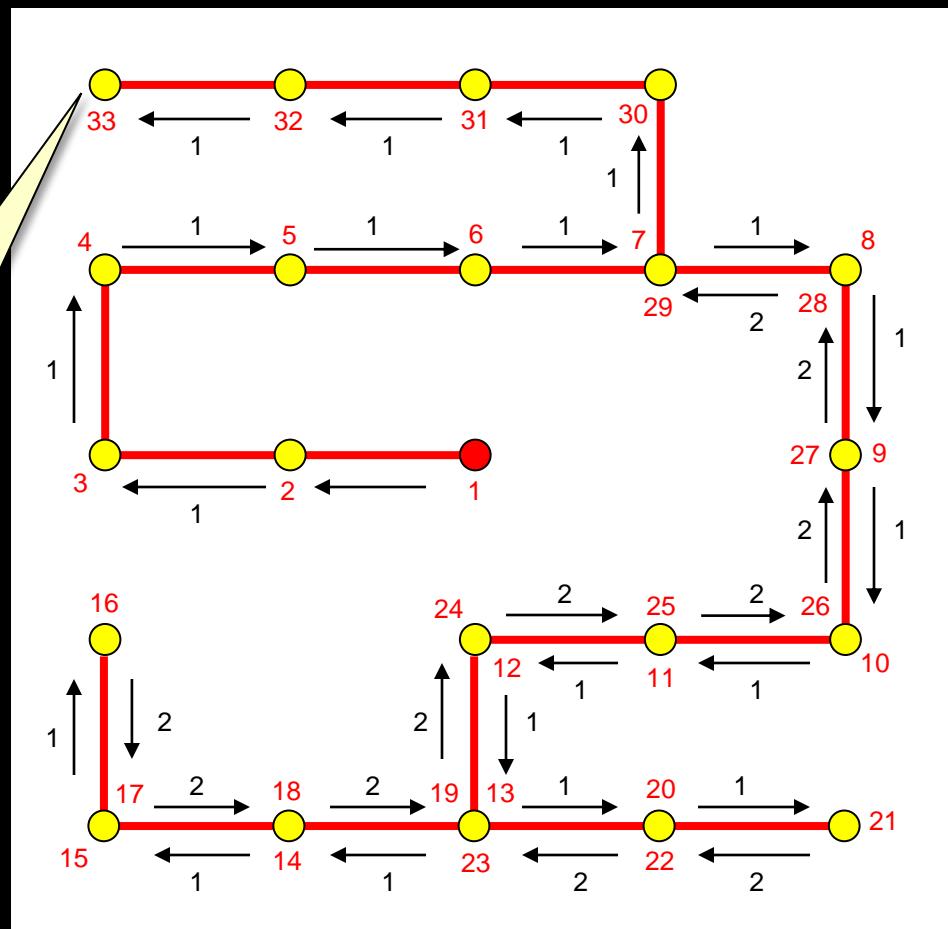


# More Efficient Travelling

- Currently, robot travels back to root, but this is not necessary.
- Better if we stop when travelled on each edge once.
- Result is that path is shorter with less points.

Robot stops here now. It does not need to go back to the root node.

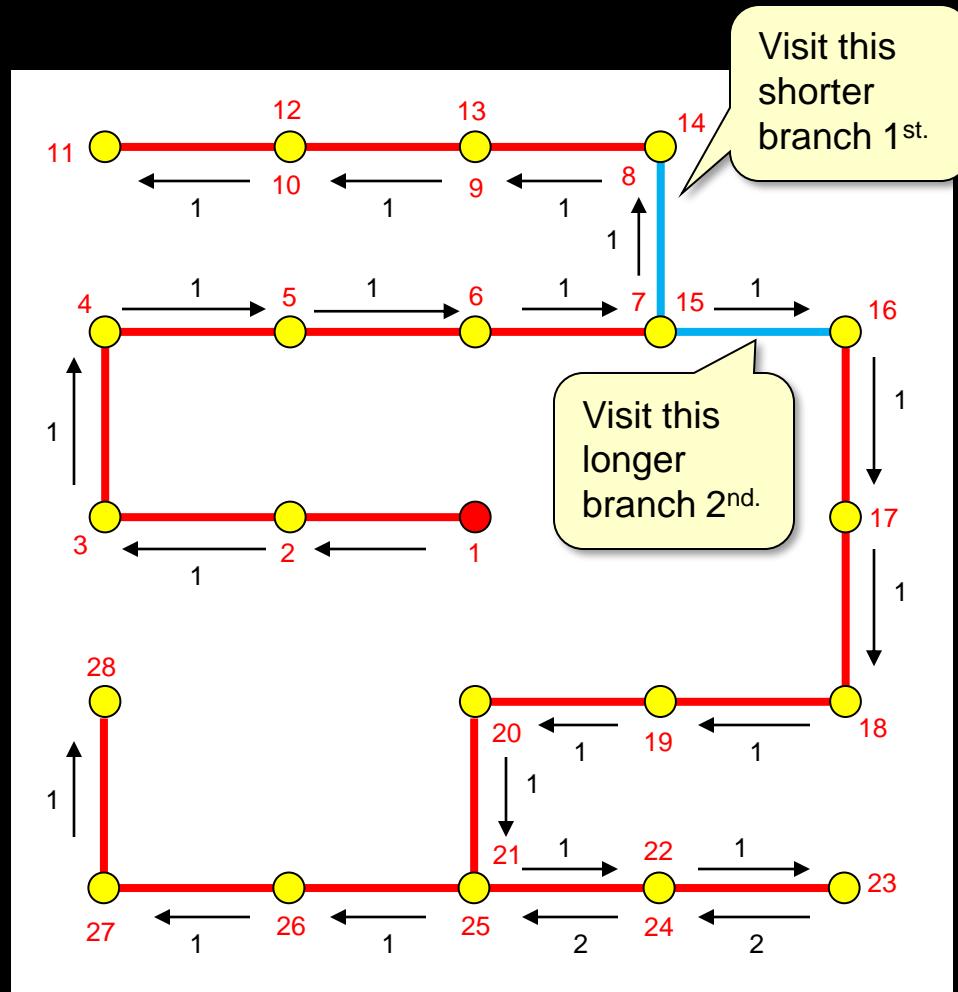
Path has only 33 points ... which is 23% shorter!



# More Efficient Travelling

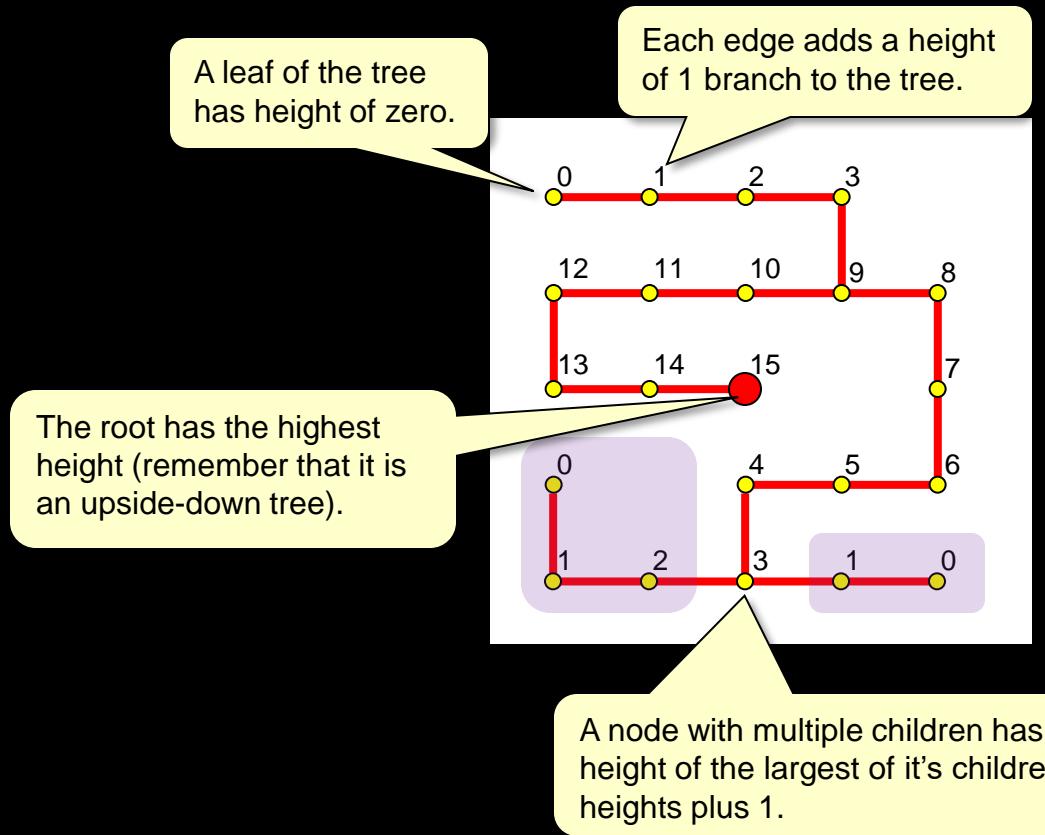
- Can shorten even more if we visit the smaller branches before the longer ones.
- Results in less travel since we don't have to travel a second time on the longer branches
- Requires us to visit the shorter branches of the tree first

Path has only 28 points now ... which is 35% shorter than original!



# Tree Height

- If we want to visit certain branches of the spanning tree in some specific order (e.g., smallest first), then we need to compute the height of the tree at various nodes.



# Computing Tree Height

- Start by assigning heights of 0 to all nodes.
- Then simply traverse all nodes recursively, setting their height to be the height of their maximum child's height

```
computeNodeHeightsFrom(aNode) {  
    mark aNode as visited  
  
    IF aNode has just 1 edge THEN  
        set aNode's height to 0  
    ELSE  
        set aNode's height to 1  
  
    max = 0  
    FOR each edge e of aNode DO {  
        otherNode = node of edge e that is not aNode  
  
        IF otherNode was not yet visited THEN {  
            computeNodeHeightsFrom(otherNode)  
            IF height of otherNode > max THEN  
                max = height of otherNode  
        }  
    }  
    set height of aNode to its current height + max  
}
```

A leaf of the tree has height of zero.

All other nodes have at least a height of 1.

Add the height of the maximum child to aNode's current height in the tree already.

Get all children's heights recursively and keep the maximum.

# Traversal By Tree Height

- To traverse according to the branch sizes, we need to sort the neighbouring nodes by height and visit in that order:

Same code as before, but now we get the neighbours and sort them by increasing order of height so that we can visit the smaller branches first.

```
computeCoveragePathFrom(aNode, incomingEdge) {
    IF previous of aNode is not NULL THEN
        return

    add aNode's location to the path

    set previous of aNode to node at other end of incomingEdge

    [
        neighbours = a list of aNode's neighbours
        sort neighbours by increasing order of their precomputed height

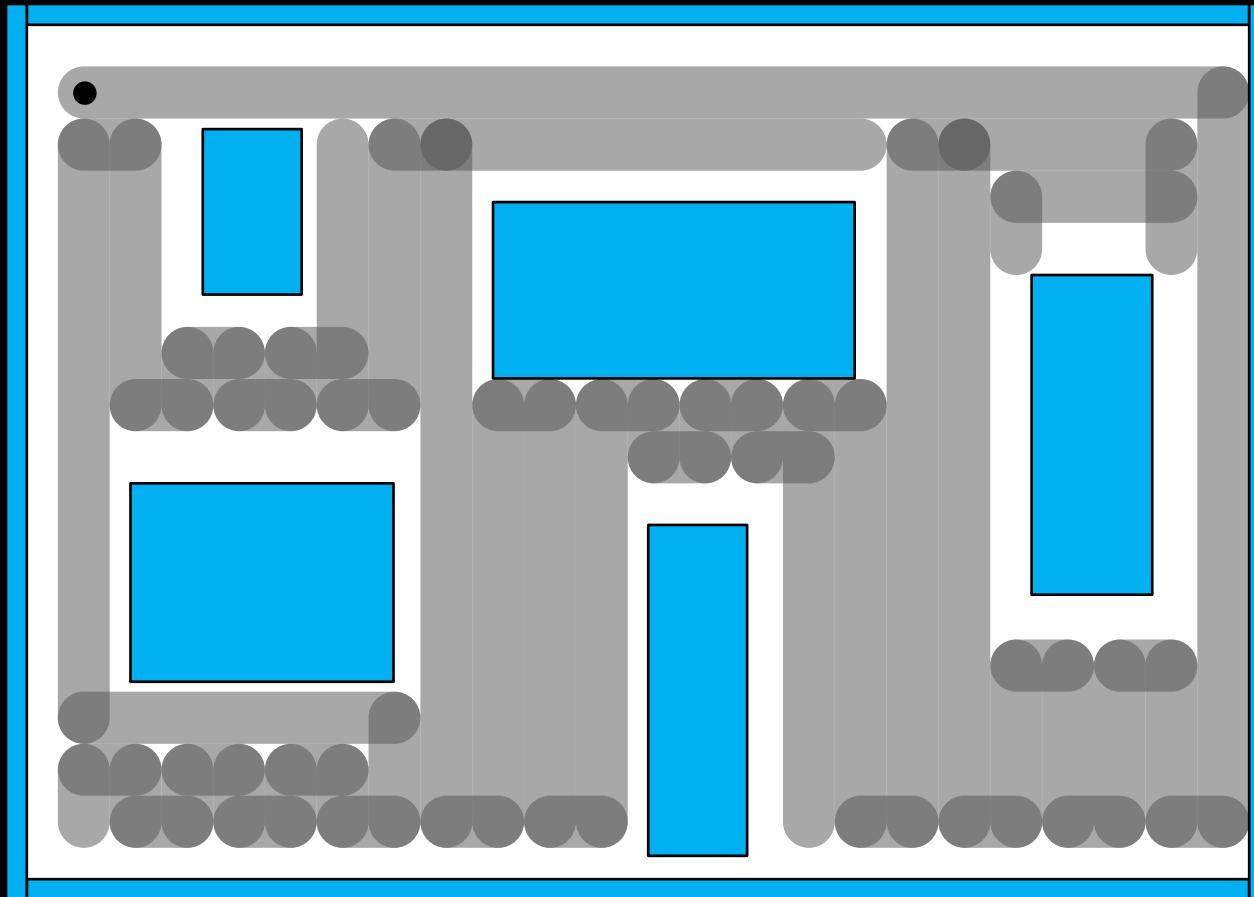
        FOR each neighbour neigh in neighbours DO {
            edge = the edge that connects aNode and neigh
            computeCoveragePathFrom(neigh, edge)
        }
        add location of aNode's previous to the path
    ]
}
```

Start the  
Lab ...

# Improving Coverage

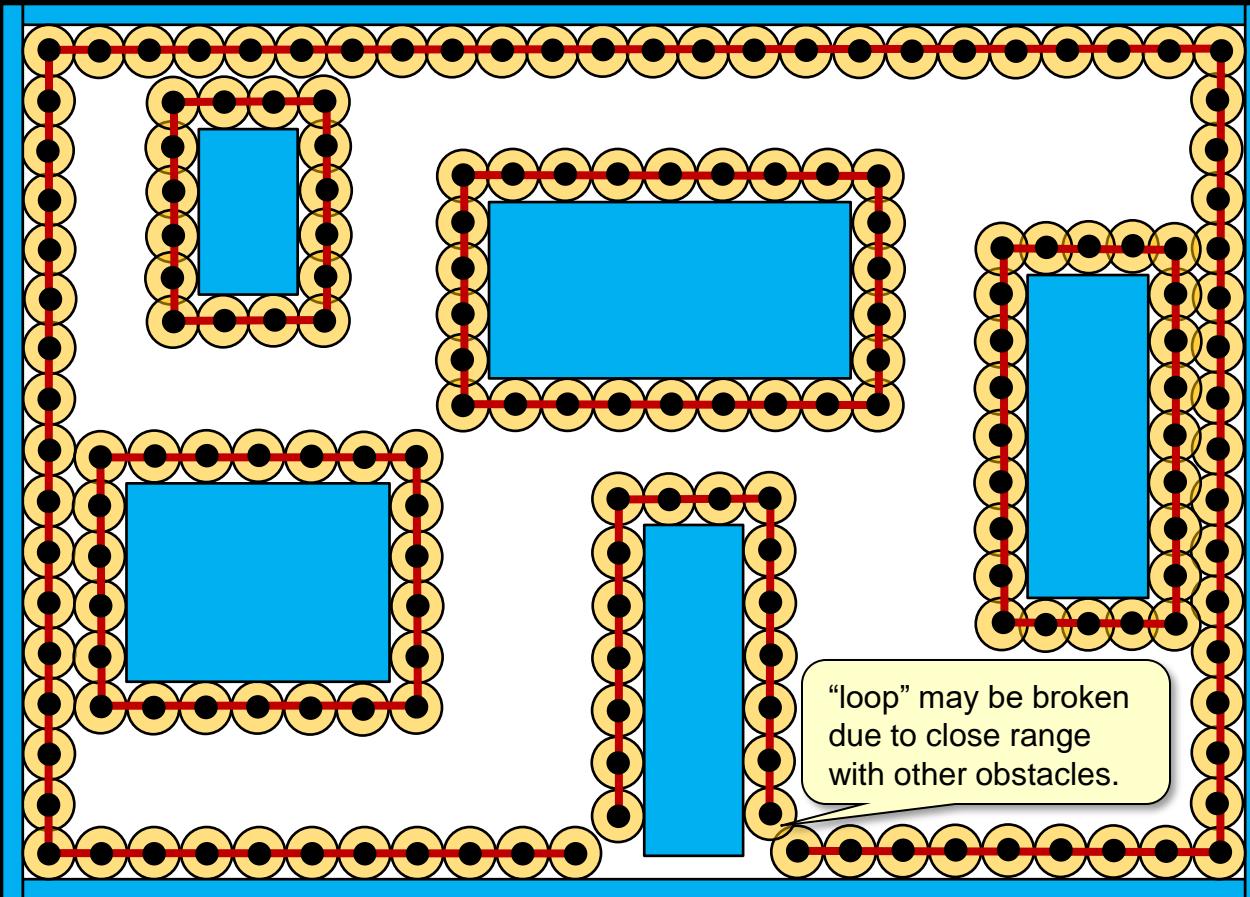
# Spanning Tree Coverage

- Recall that spanning tree coverage left areas untouched around the obstacles and border:



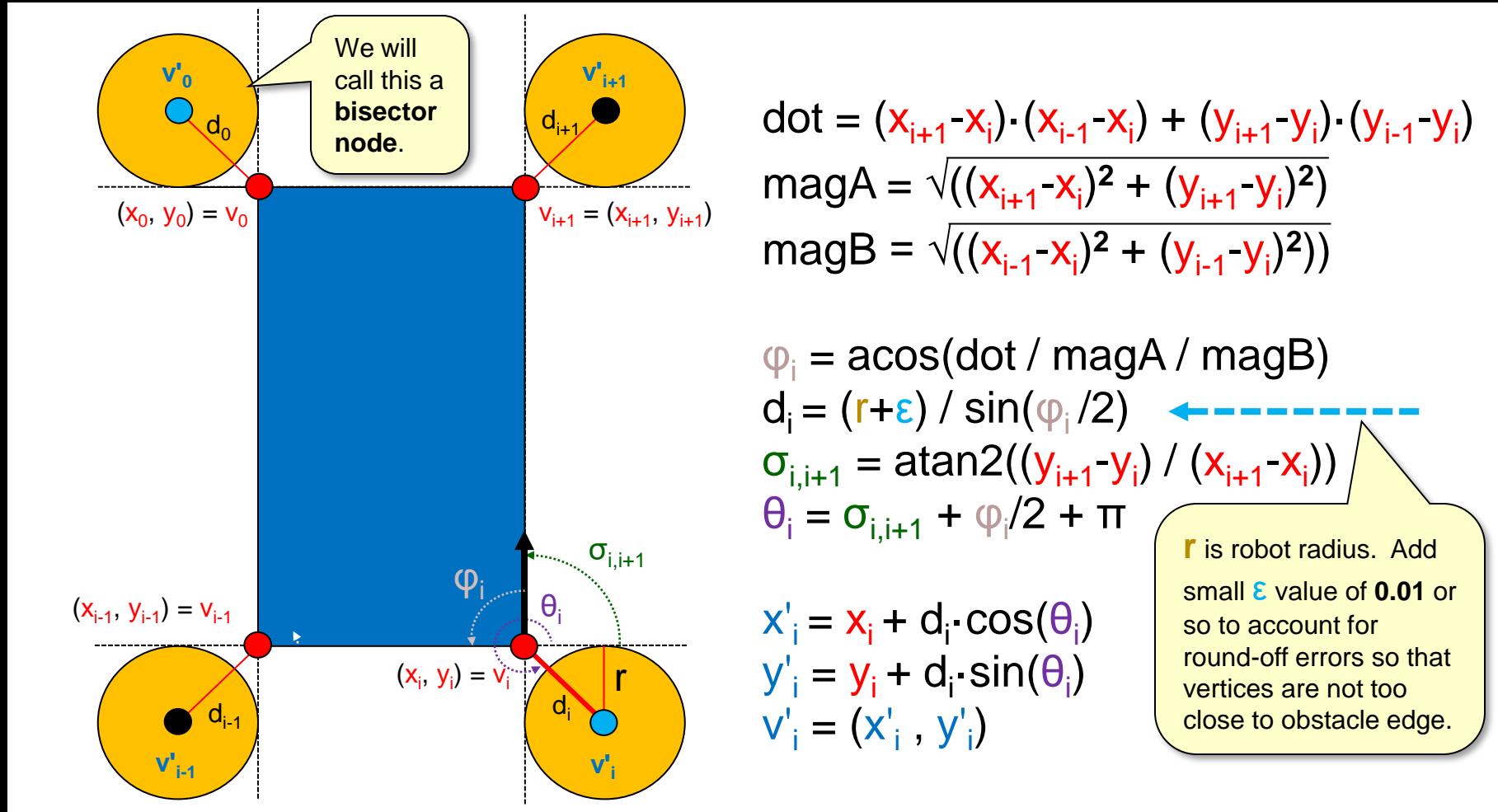
# Border/Obstacle Coverage

- How do we get better coverage around the obstacles?
- Place additional graph “loops” around the obstacles:



# Bisector Vertices

- Add graph nodes around obstacles by considering the robot radius and the bisectors of pairs of adjacent obstacle edges:



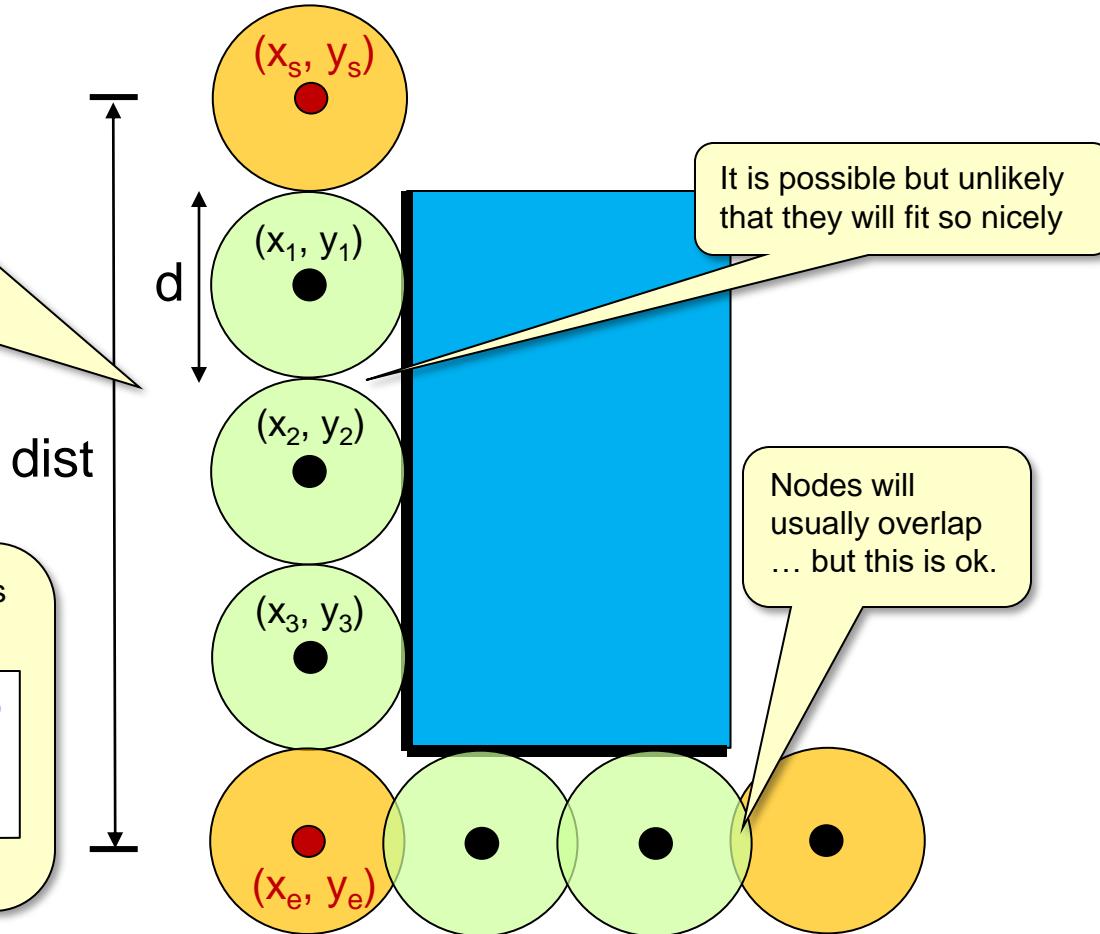
# Creating the Graph Nodes

- For each edge of the obstacle, add nodes along each edge from its two endpoints.

Number of nodes to add in between the bisector nodes for an edge is  
**nCount = ceil[dist/d - 1]**  
with a gap of  
**gap = dist / (nCount + 1)**  
between each adjacent node.

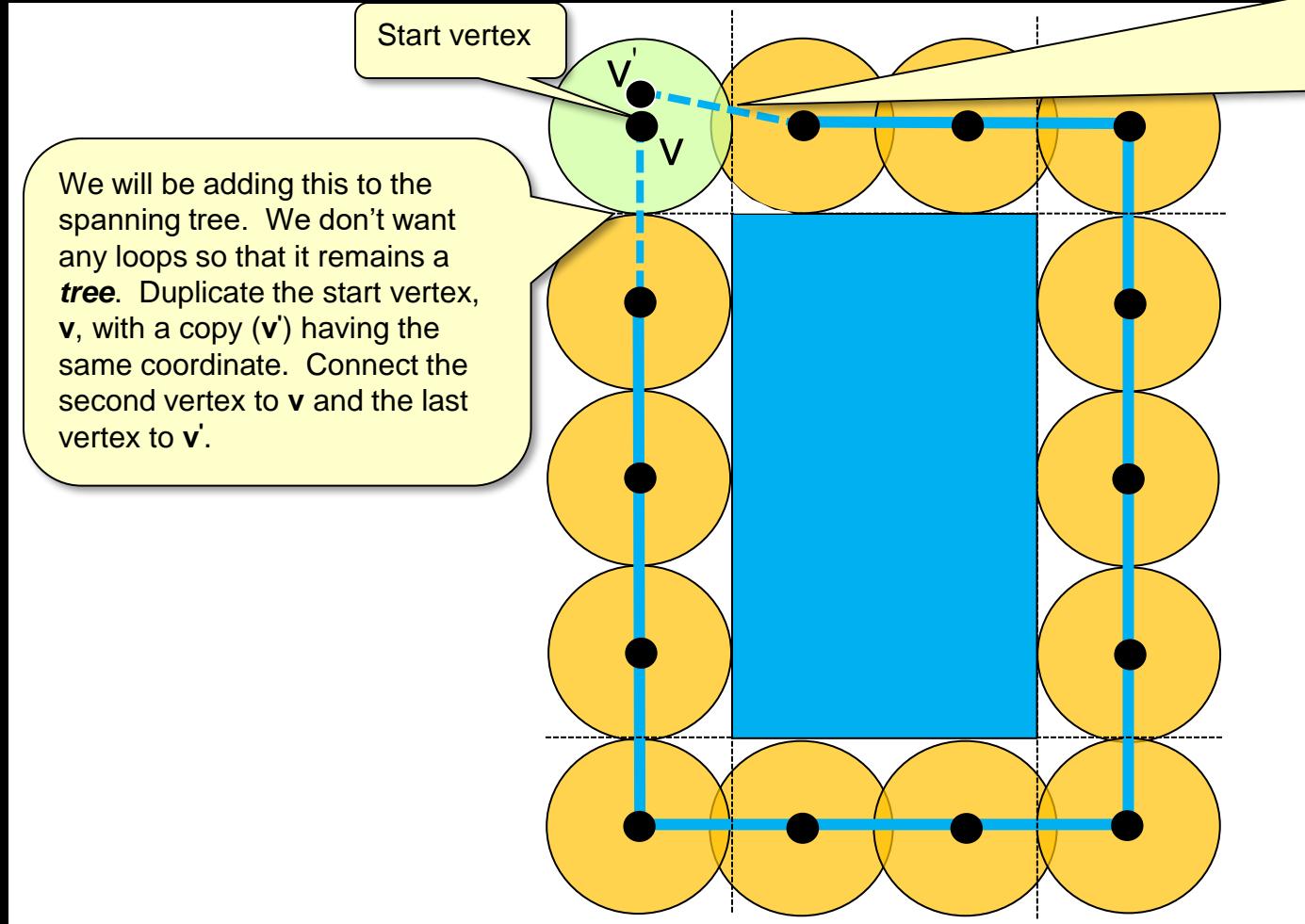
Location  $(X_k, Y_k)$  of each node is computed as follows:

```
FOR k FROM 1 to nCount DO
   $X_k = x_s + (x_e - x_s) * gap / dist * k$ 
   $Y_k = y_s + (y_e - y_s) * gap / dist * k$ 
```



# Connecting the Nodes

- Connect adjacent nodes along each edge:



Assuming a counter-clockwise travel around the obstacle, this edge is needed so that we clean fully around the obstacle, otherwise the white area will not be covered.

# Iterating Through Node Loop

- When looking for invalid nodes, we need to iterate through the Node “loop”:

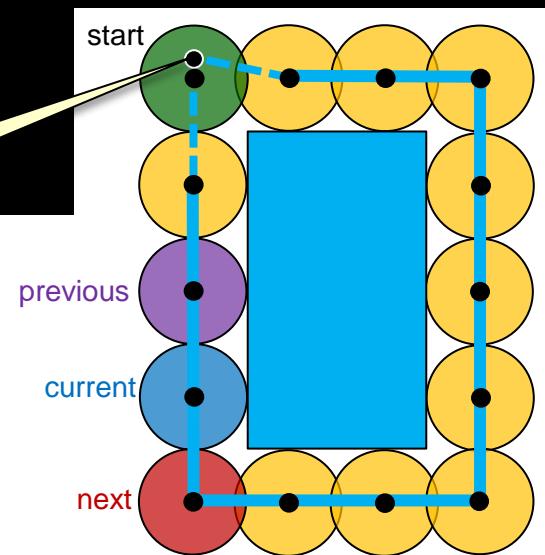
```
start = the starting node of the loop  
  
bad = an empty list  
previous = NULL  
current = start  
next = node at other end of start's first and only edge  
  
WHILE (previous is NULL) or (current does not have the same coordinates as start) THEN {
```

This happens only first time in loop.

```
    IF current is invalid THEN  
        add current to bad  
    previous = current  
    current = next  
    next = node at other end of next's first edge  
    IF (next == previous) AND (there is at least one more edge connected to current) THEN  
        next = node at other end of current's 2nd edge  
    }  
    IF current is invalid THEN  
        add current to bad
```

This handles the very last **current** node in case it is invalid.

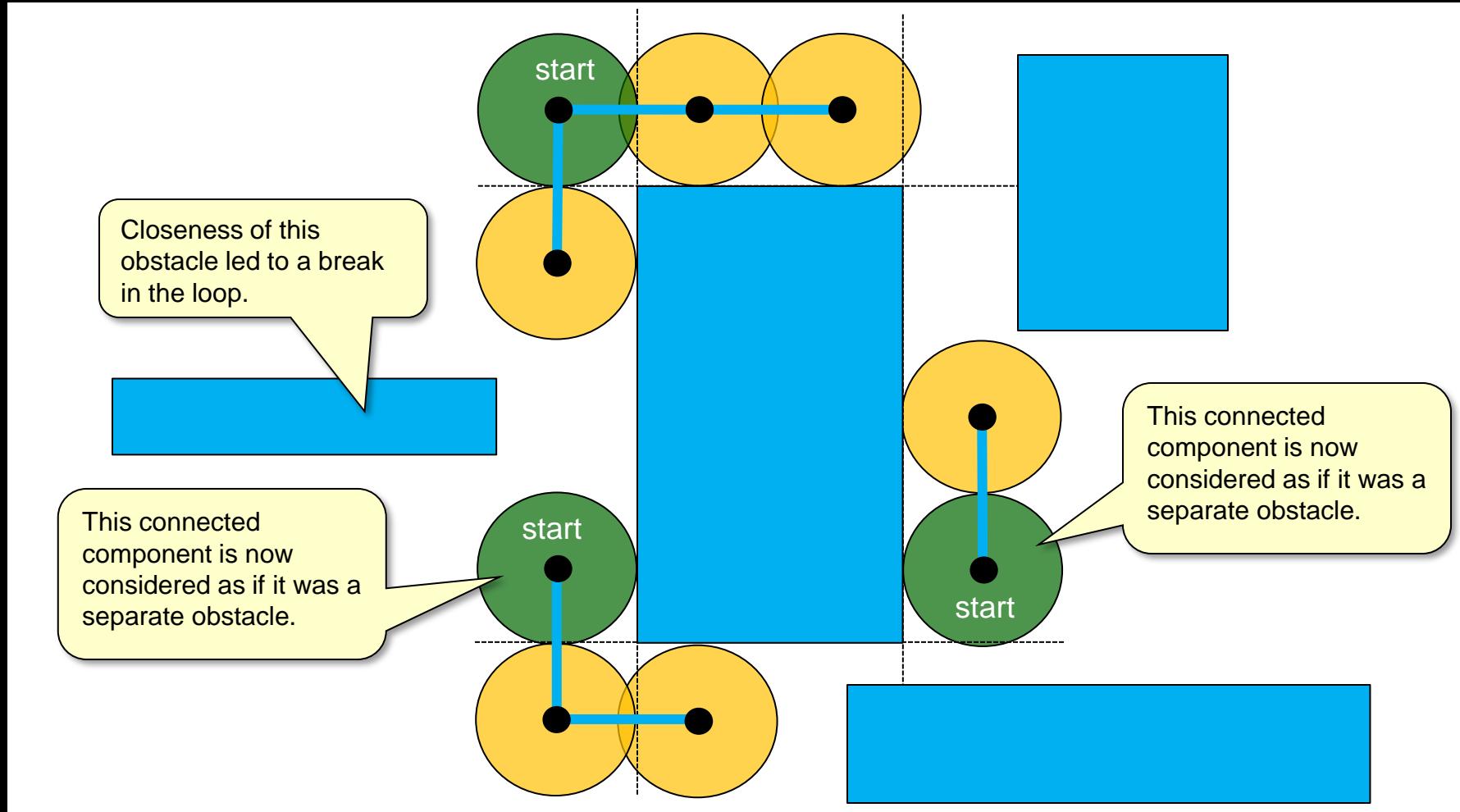
This happens when **current** is here



This case is needed in case the next edge is not the first one in **next** node's list. This will happen later when we are disconnecting the duplicate start vertex (slide 9).

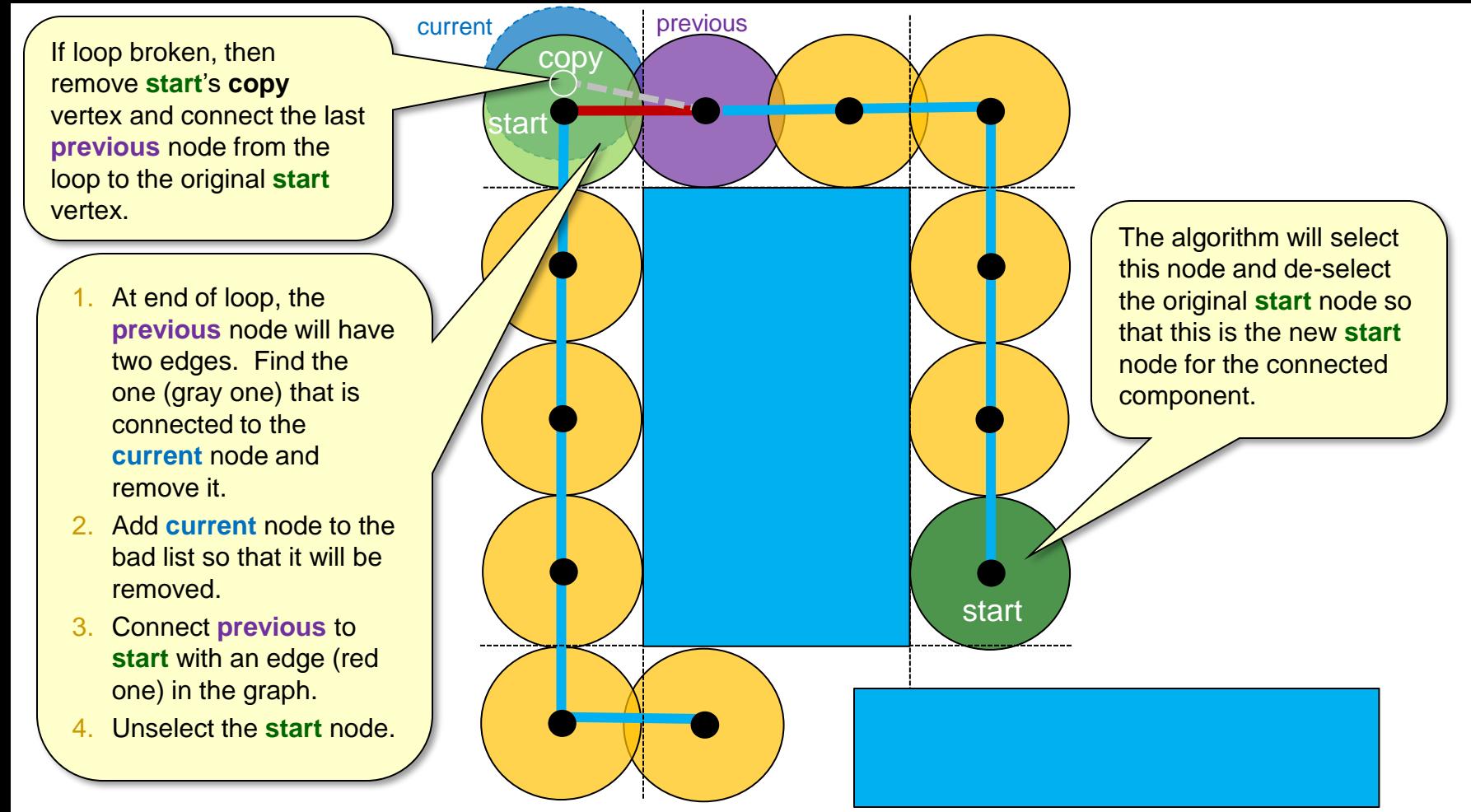
# Handling “Really” Broken Loops

- If loop gets broken into multiple components, we must determine the **start** of each connected component:



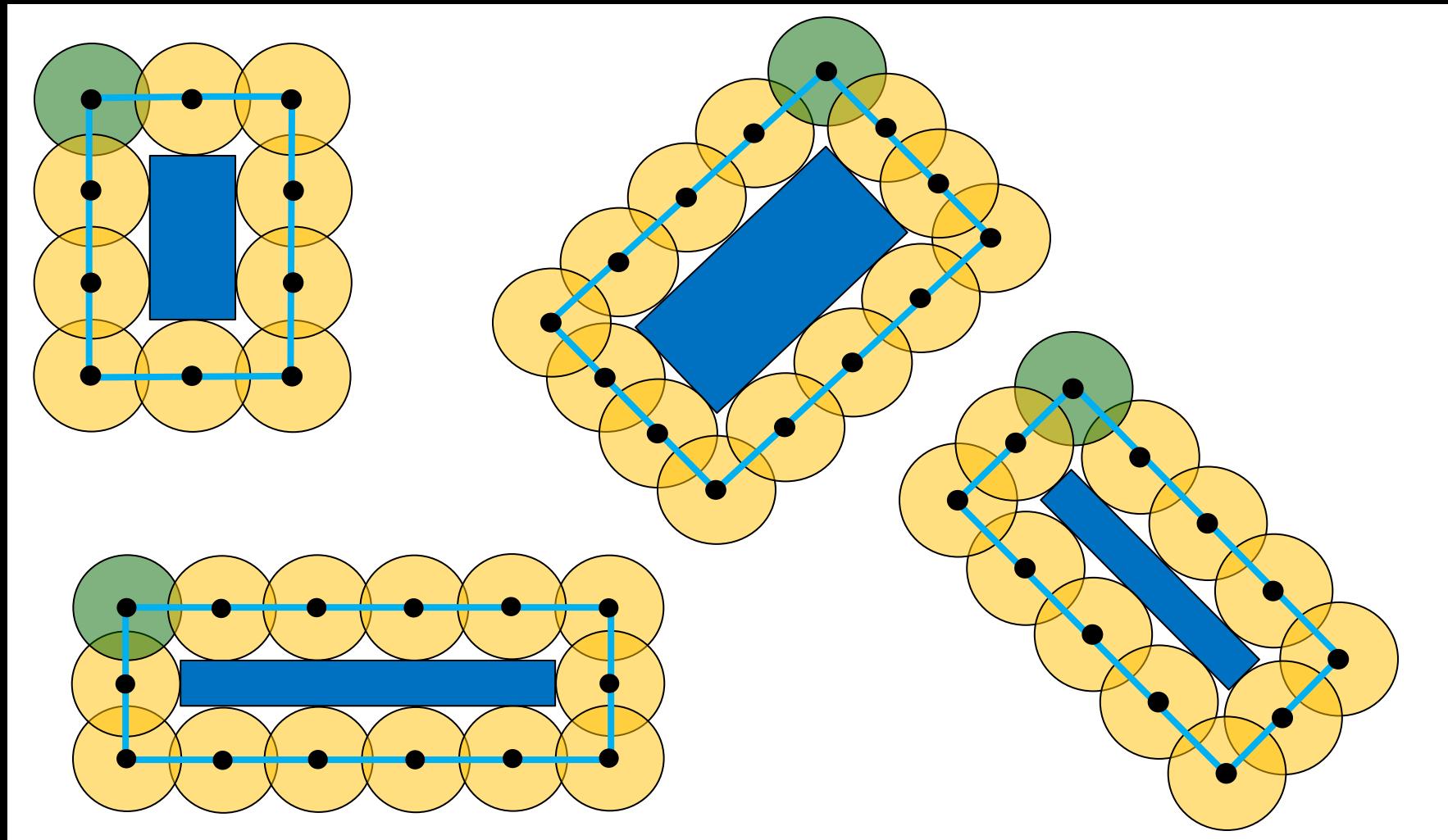
# Handling Broken Loops

- If loop gets broken at all, it is still a single connected component and there is no special case on the first vertex:



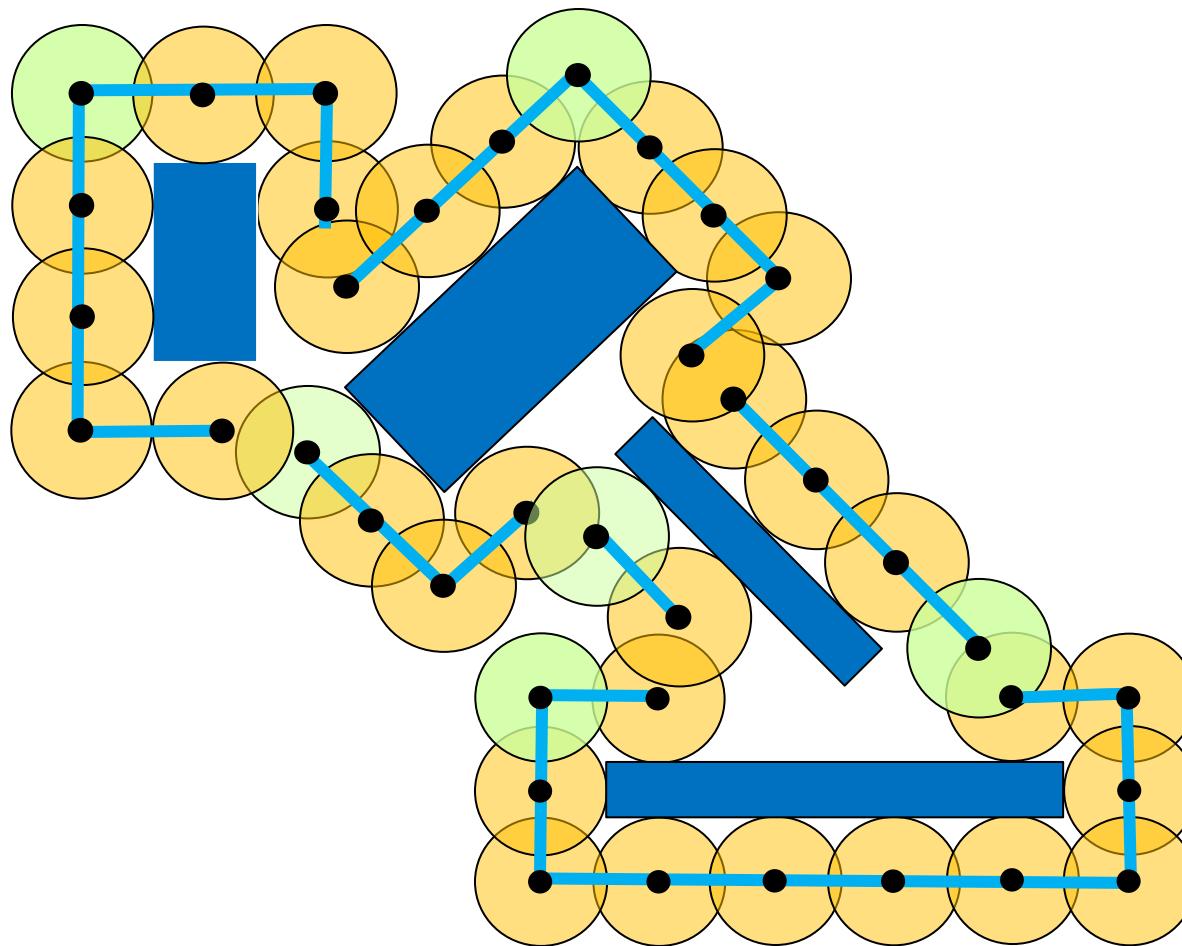
# Obstacle-Coverage Vertices

- Do each connected-component separately



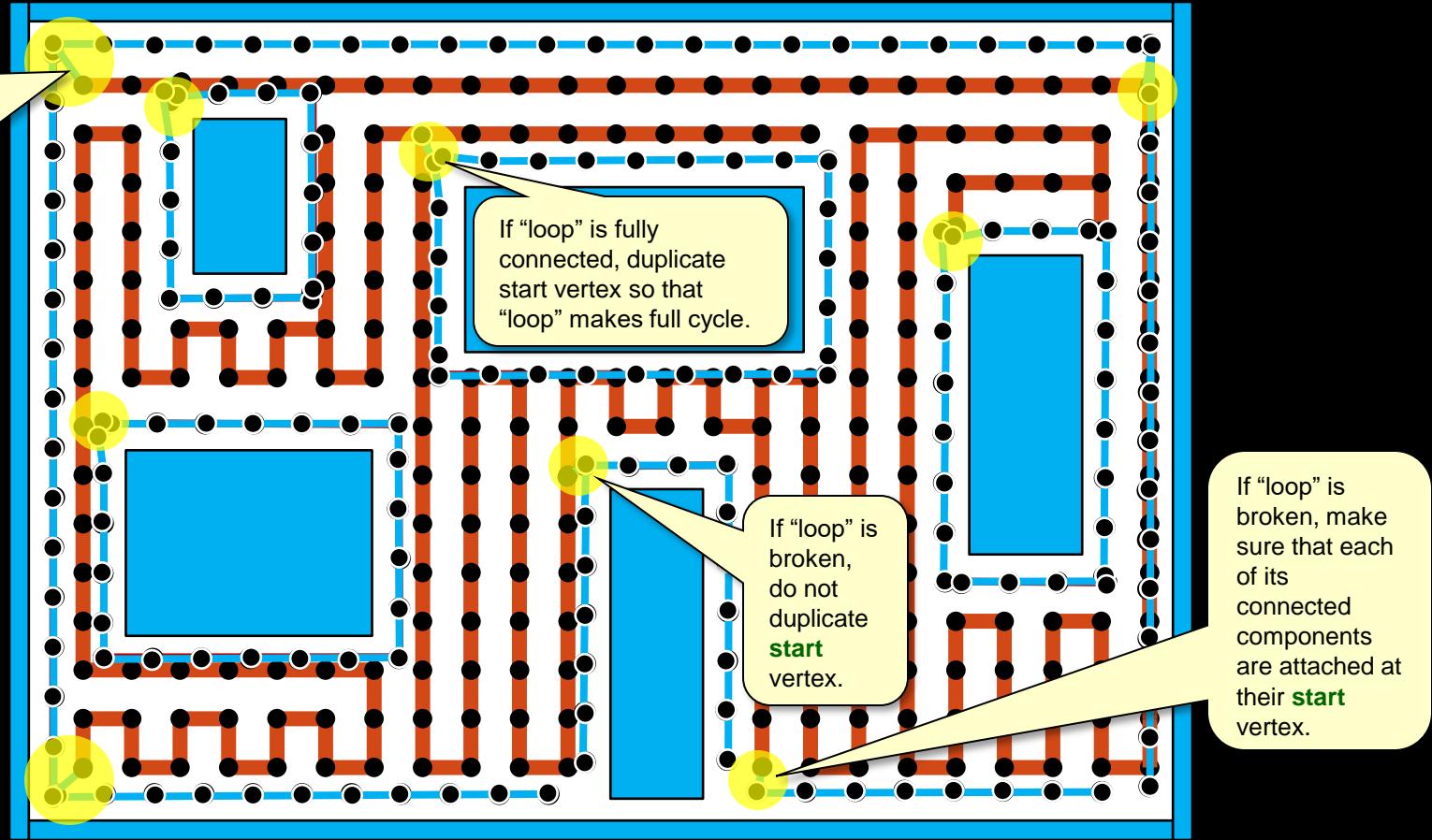
# Connected Components

- Eliminate invalid vertices that intersect other obstacles ... resulting in connected components:



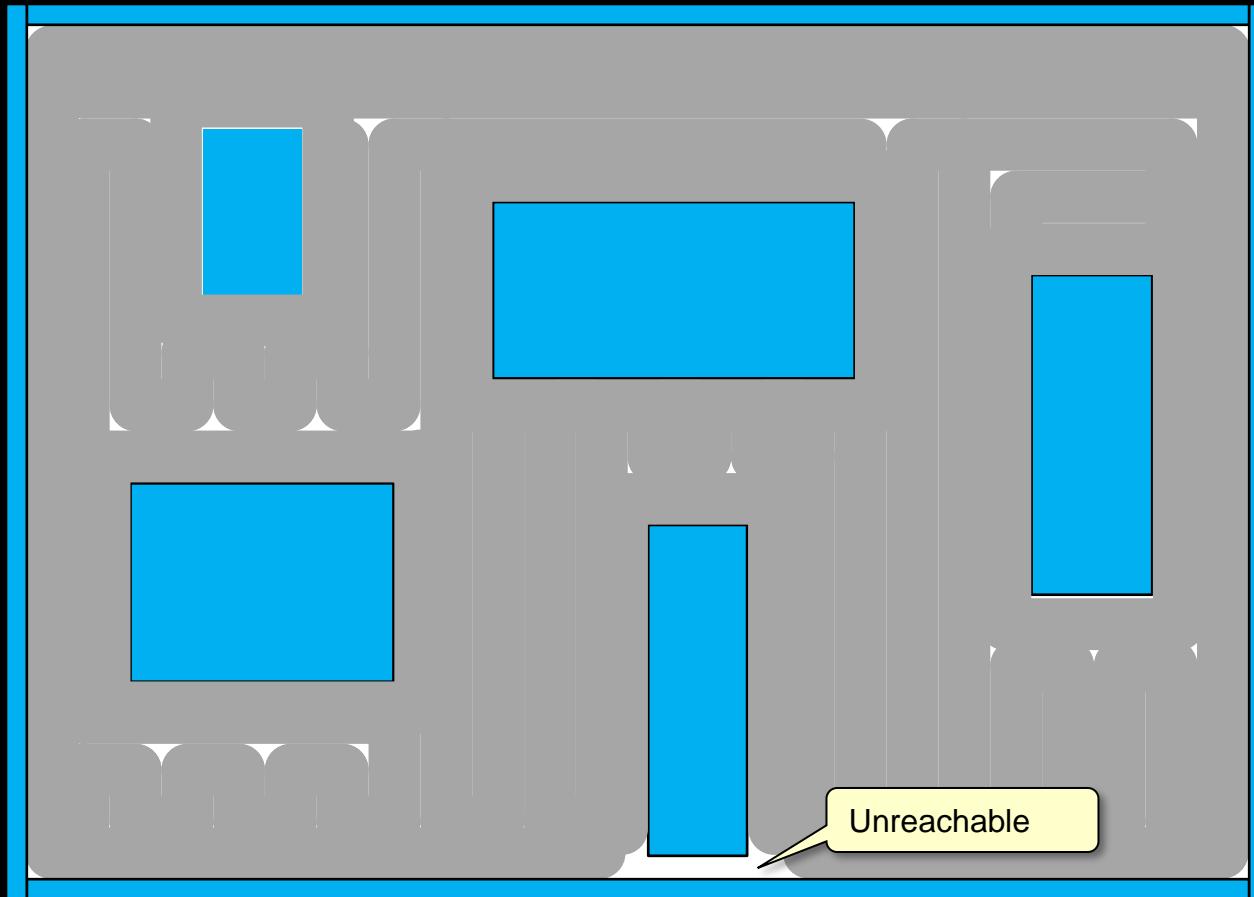
# Merged Spanning Tree

- Attach obstacle and border “loops” to spanning tree
  - Result is still a tree since each obstacle added only branches:



# Final Area Coverage

- Environment is reasonably well-covered in the end:



Start the  
Lab ...

# Laser Range Finders

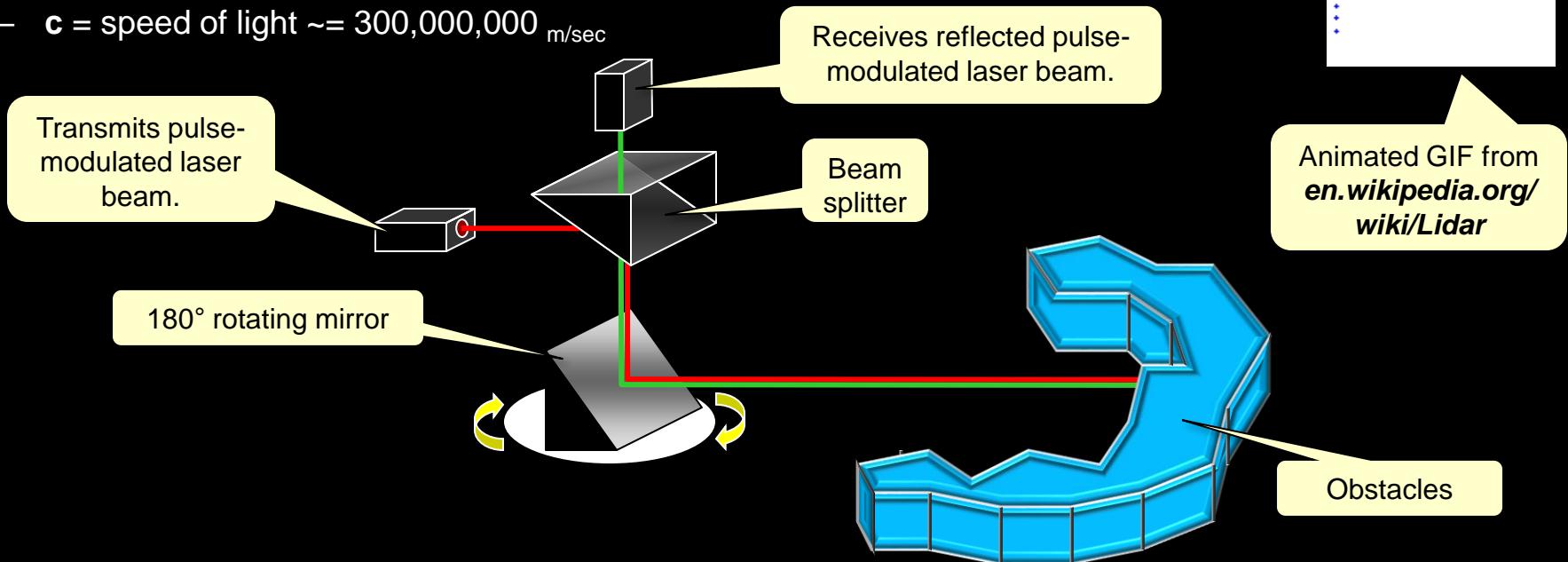
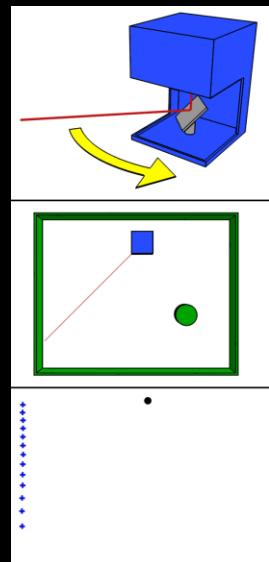
# Laser Range Finders

- Laser Range Finders are perhaps the most accurate sensors for measuring distances.
- Similar concept to IR distances sensors in that IR light is emitted and detected.
- These sensors are ***LiDAR*** (Light Detection and Ranging) systems
- Lidar systems use one of three techniques:
  - *Pulsed Modulation*
  - *Amplitude Modulation Continuous Wave (AMCW)*
  - *Frequency Modulation Continuous Wave (FMCW)*



# Pulsed Modulation Lidar

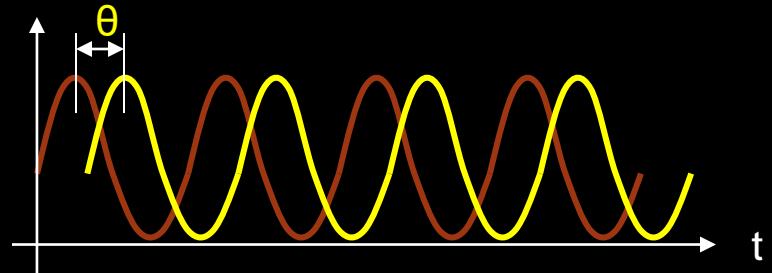
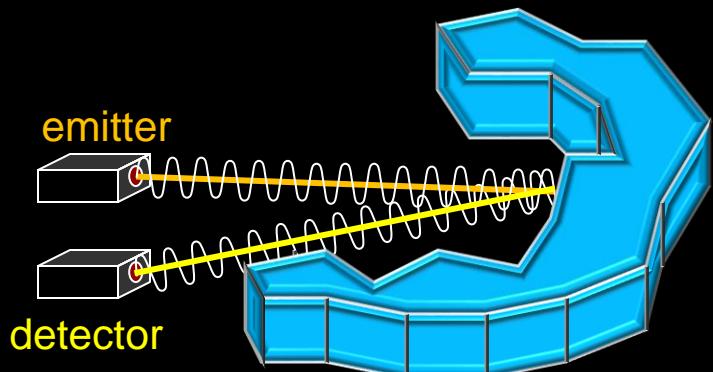
- Emits a pulsed laser light beam
  - Reflected light is returned to detector
  - Rotating mirrors used to direct outgoing and incoming light to perform up to 240° scan
- Range calculated as  $r = t \times c / 2$ 
  - $t$  = time taken for light to return from when it was sent
  - $c$  = speed of light  $\approx 300,000,000$  m/sec



# Amp. Mod. Cont. Wave Lidar

## ■ AMCW sensors

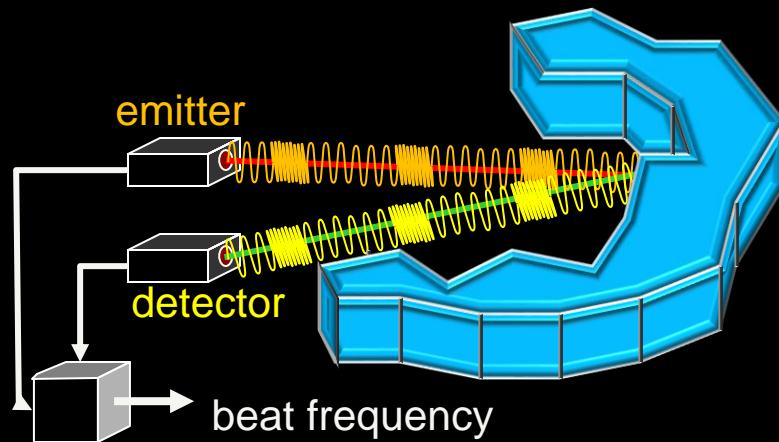
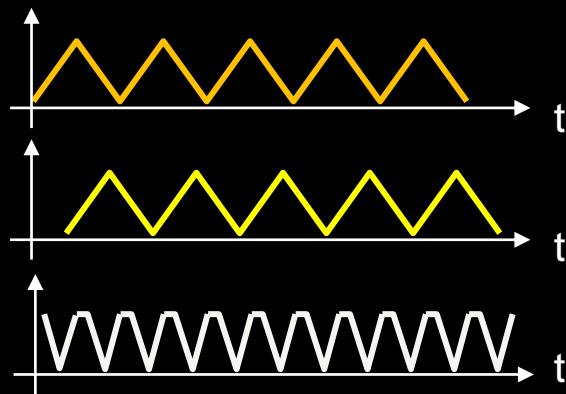
- emitter sends out a continuous modulated laser signal (i.e., intensity of beam is modulated using some wave pattern (e.g., sin wave)).
- detected light has same amplitude but is phase-shifted
- difference in phase shift indicates range



Range calculated as  $r = \theta c / 4\pi f$  where  
 $\theta$  = phase shift  
 $f$  = frequency of modulated signal

# *Freq. Mod. Cont. Wave Lidar*

- FMCW technique is simpler and hence lower cost
- Resolution is limited by modulating frequency
- FMCW sensors similarly emit a continuous laser beam, but modulated now by frequency.
  - emitted signal is mixed with reflected signal.
  - result is difference in frequency



# Many Scanners Available ... e.g. ...

## ▪ Sick LMS-291

- 180° field of view with 0.5° resolution
- accuracy  $\pm 1.5_{\text{cm}}$  in short range ( $1_{\text{m}} - 8_{\text{m}}$ )
- and  $\pm 4_{\text{cm}}$  in long range ( $8_{\text{m}} - 20_{\text{m}}$ )



## ▪ Hokuyo URG-04LX

- 240° field of view with 0.36° resolution
- accuracy  $\pm 1_{\text{cm}}$  in range ( $6_{\text{cm}} - 4_{\text{m}}$ )



## ▪ Velodyne HDL 32E

- 360° field of view with 0.33° resolution
- accuracy  $\pm 2_{\text{cm}}$



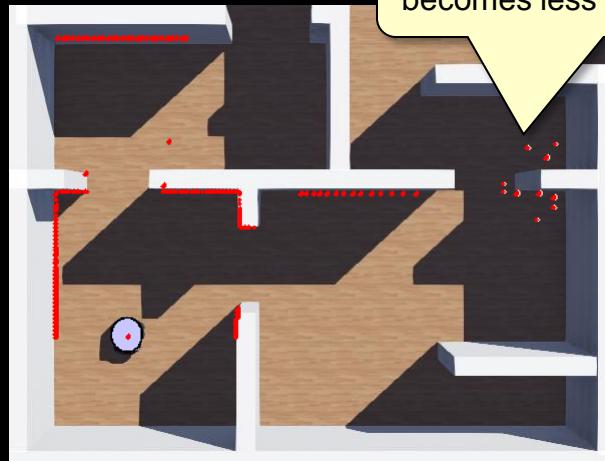
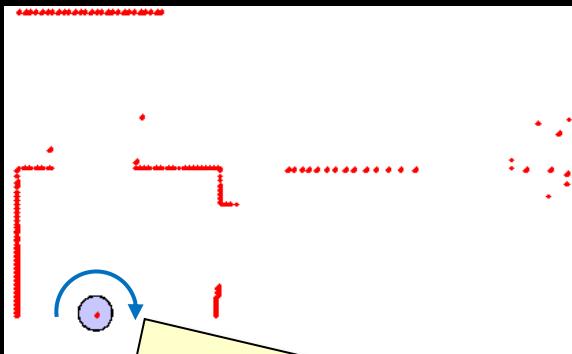
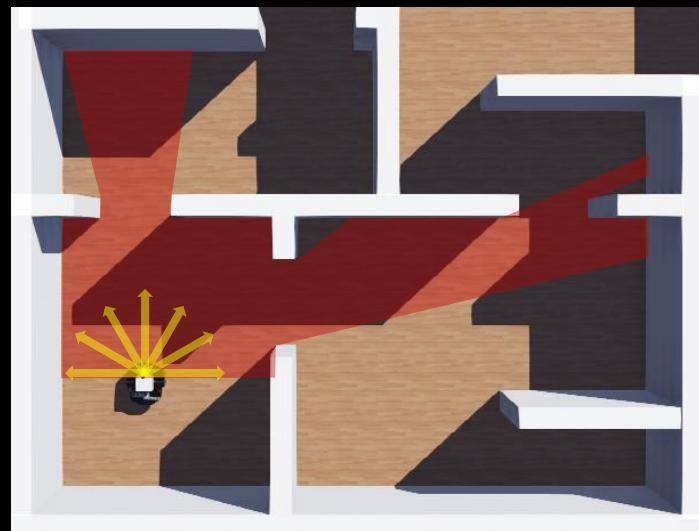
## ▪ Velodyne VLP-16

- 360° field of view with 0.4° resolution
- accuracy  $\pm 3_{\text{cm}}$

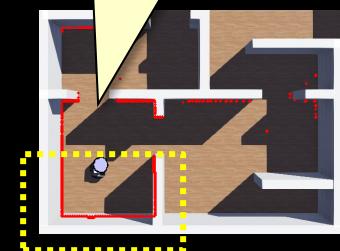


# Laser Range Finder 2D Scan

- Consider a robot with a 180° sick LMS-291 lidar sensor in a 2D environment:
- Scanning a single set of data produces a set of points in 2D:



Need just two sets of scans to cover a full 360°.



# The Webots Lidar Class

- We use the **Lidar** class in Webots to represent a Lidar sensor:

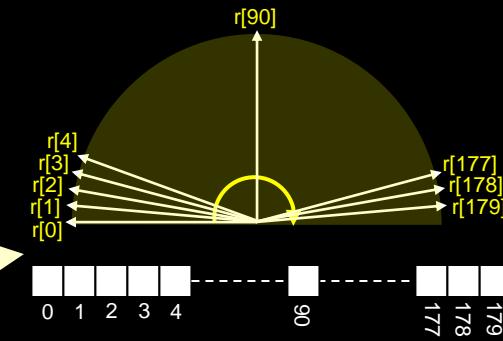
```
import com.cyberbotics.webots.controller.Lidar;  
  
Lidar lidar = new Lidar("Sick LMS 291");  
lidar.enable(TimeStep);  
  
int fieldOfView = lidar.getFov()*180/Math.PI // 180 degrees  
int lidarWidth = lidar.getHorizontalResolution(); // 180 degrees  
int lidarLayers = lidar.getNumberOfLayers(); // 1 layer  
double maxRange = lidar.getMaxRange(); // 80 meters  
  
float r[] = null;  
  
while (robot.step(TimeStep) != -1) {  
    r = lidar.getRangeImage();  
    ...  
}
```

This function reads the sensor and returns an array of the distances for each of the angles. In our case, 1 reading for each degree angle ... so 180 readings that cover the 180° range.

Each range reading is stored in an array at a specific index corresponding to the angle it was obtained at (i.e., from 0 to 179)

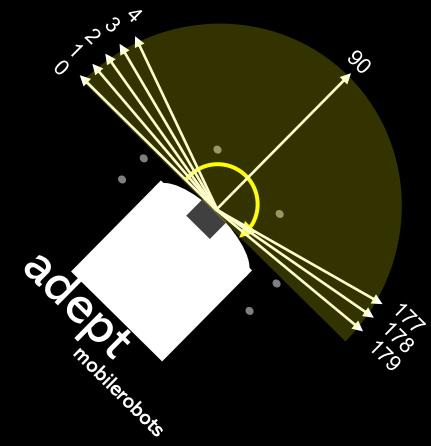
- Can be one of these:
- "Ibeo Lux"
  - "Hokuyo URG-04LX"
  - "Hokuyo URG-04LX-UG01"
  - "Hokuyo UTM-30LX"
  - "LDS-01"
  - "Sick LMS 291"
  - "Sick LD-MRS"
  - "Velodyne VLP-16"
  - "Velodyne HDL-32E"
  - "Velodyne HDL-32E"

Various methods are available to get sensor specifications.

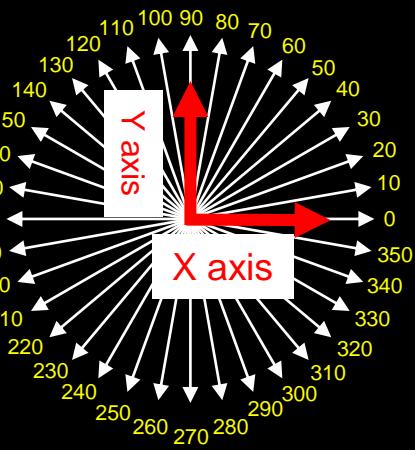
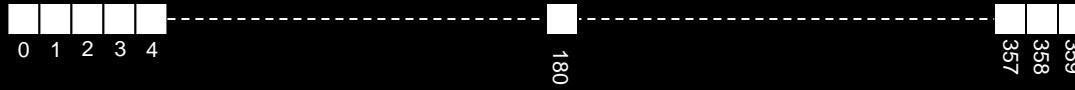


# Converting Coordinate Systems

- As mentioned, robot takes range readings which are stored in an array with indices from 0 - 179 representing each of the degrees from  $0^\circ$  to  $179^\circ$  where  $90^\circ$  is directly in front of the robot.

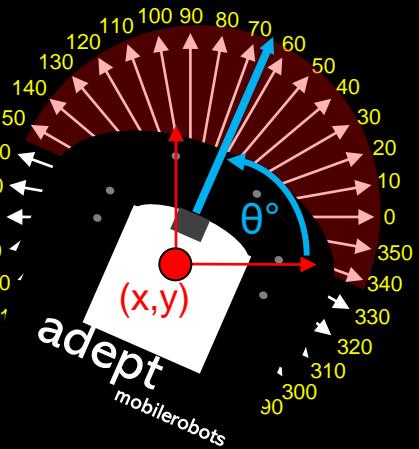
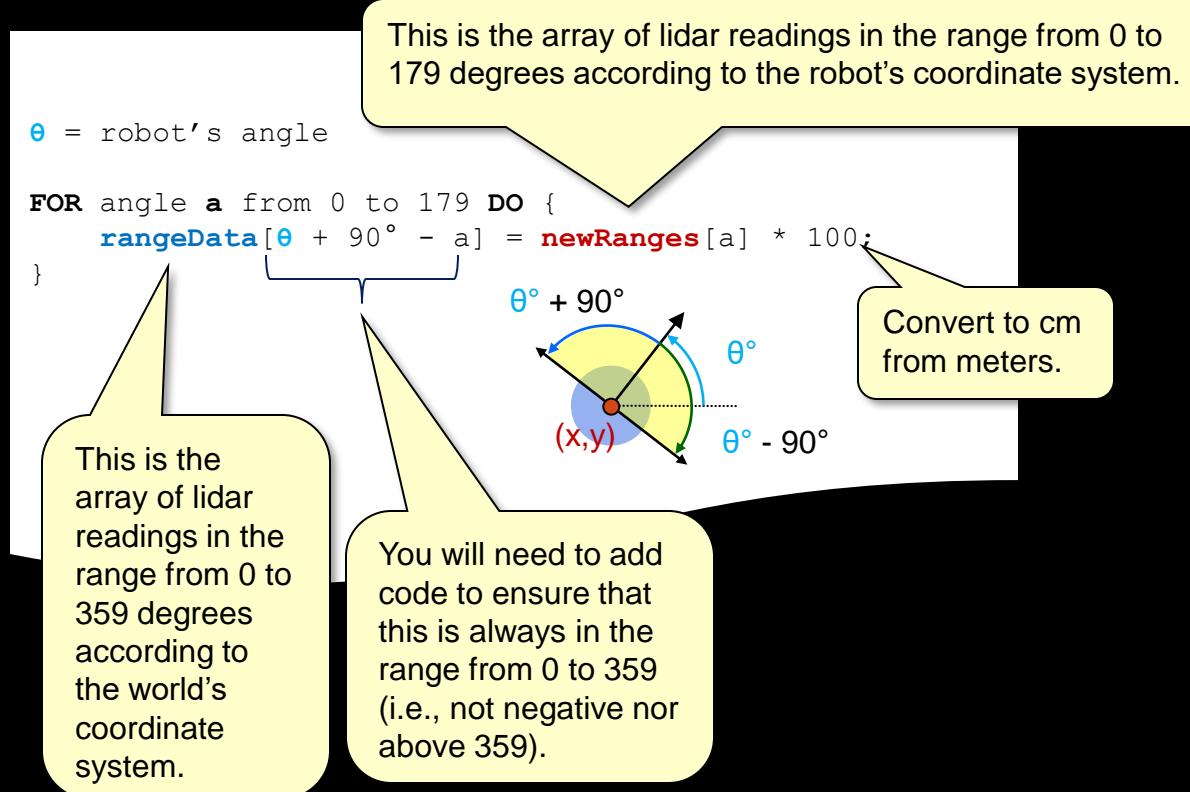


- But we need to apply the readings to the world coordinate system by storing them in the world coordinate array that has indices from 0 - 359 representing each of the degrees from  $0^\circ$  to  $359^\circ$  where  $0^\circ$  is the horizontal in the world coordinate system.



# Converting to World Coordinates

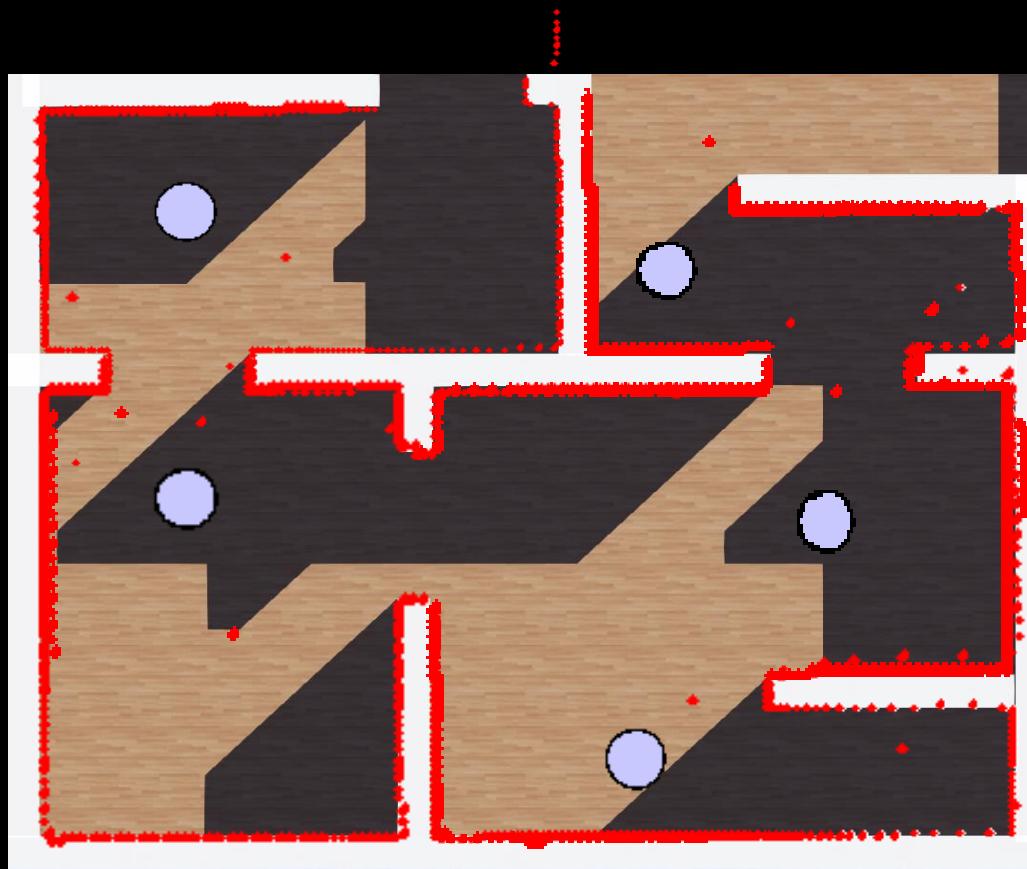
- Must transfer current readings from robot array to world array by “rotating the readings”.
  - Just need to flip the readings around and place them at different indices in the world array.



# More Accurate Mapping

---

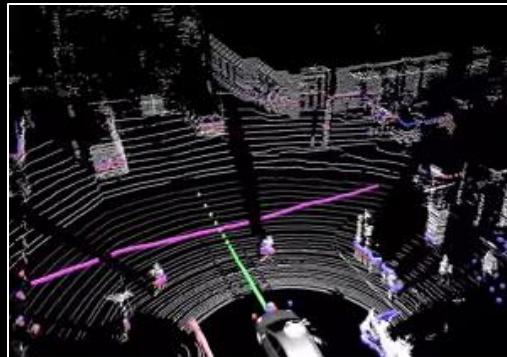
- Can make accurate maps by moving robot to just a few positions and doing a 360° scan:



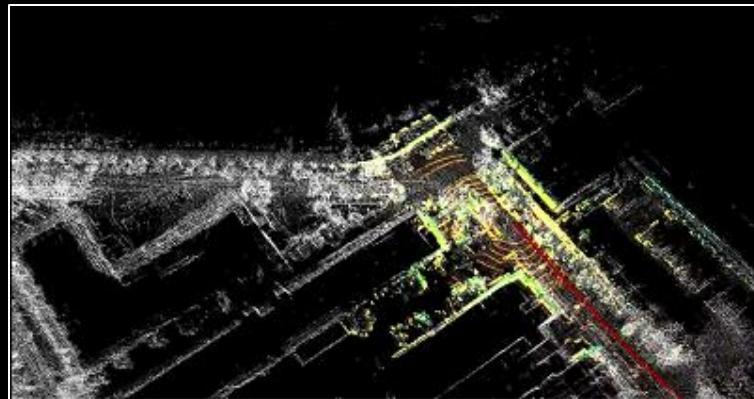
# 3D Scene Representation

---

- Taking scans at multiple vertical layers produces 3D scene:
- [https://www.youtube.com/watch?v=nXIqv\\_k4P8Q](https://www.youtube.com/watch?v=nXIqv_k4P8Q)

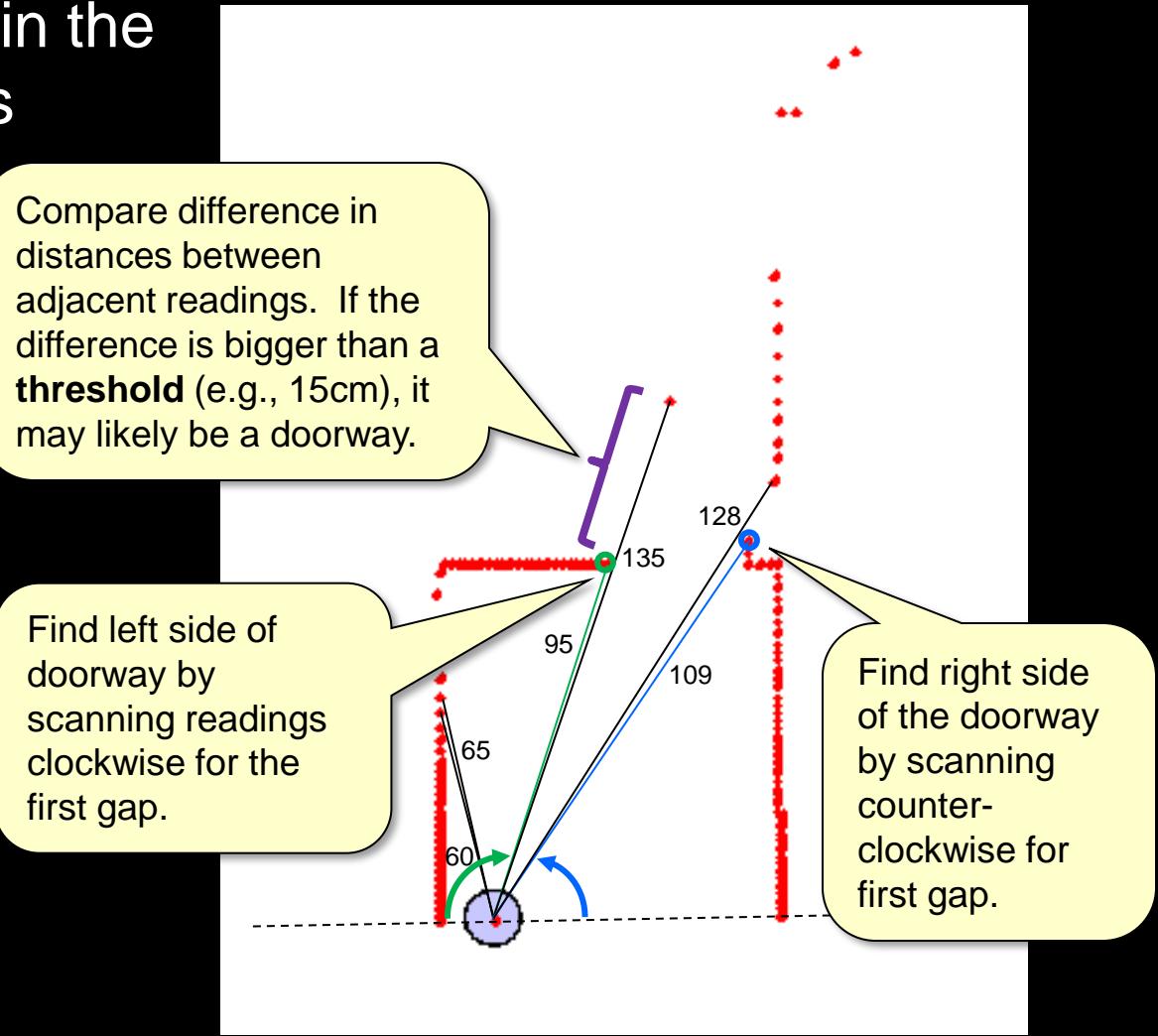


- <https://www.youtube.com/watch?v=KmulCcnbQ1U&t=25s>



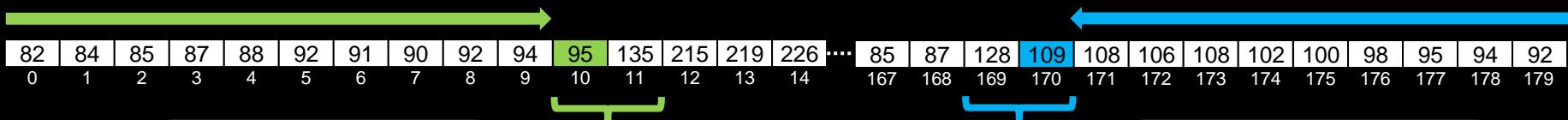
# Doorway Identification

- Since there is a lot of data available from a single scan, we can identify features in the environment, such as doorways:



# Doorway Identification

- Just go through the **rangeReadings** array from the left side and then from the right side, looking for the first “big” gap:



First “big” gap from left side is from index 10 to 11. So, the **left doorway** point is at index **10**.

First “big” gap from right side is from index 170 to 169. So, the **right doorway** point is at index **170**.

- Then calculate the coordinates represented by the doorway point. Here is how to calculate the left one:

Angle that robot is currently facing

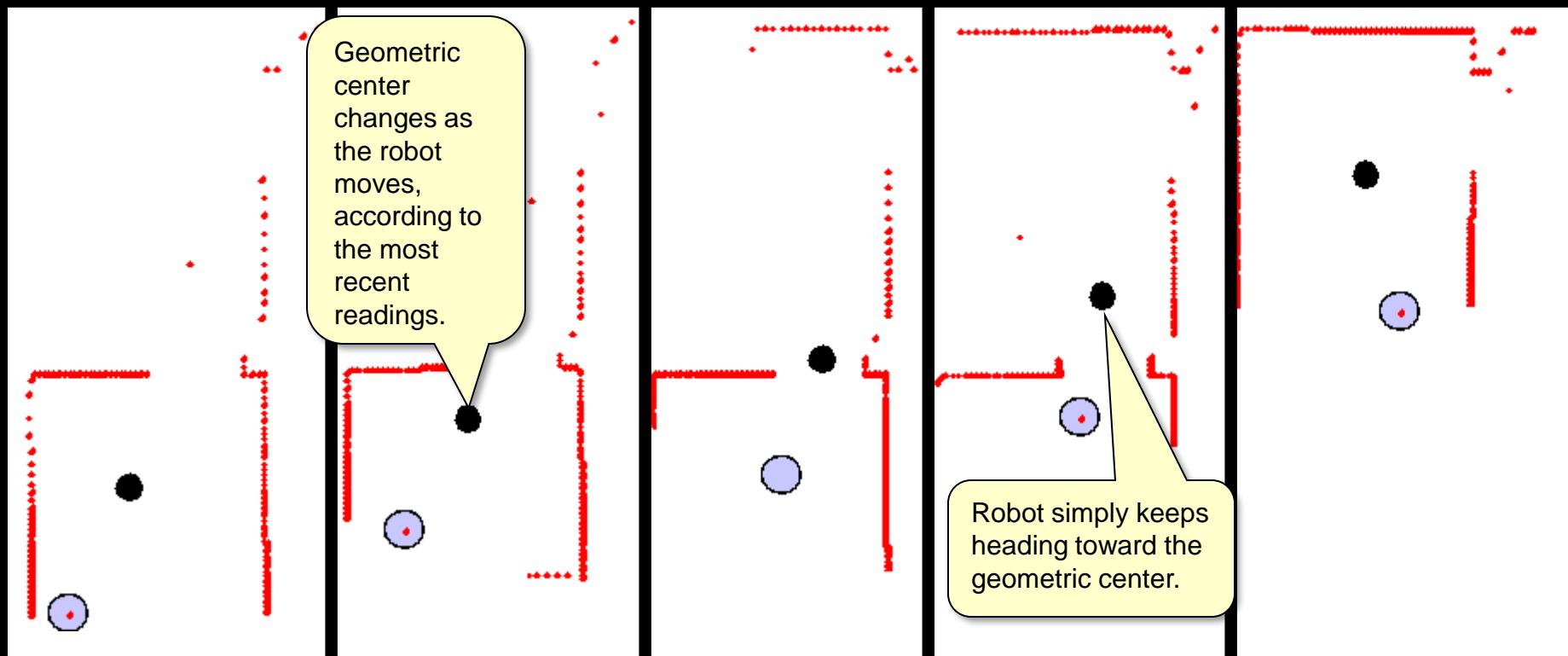
Found from array as shown above

```
leftDoorwayAngle = θ + 90° - leftDoorwayIndex  
leftDoorwayPointX = rangeReadings[leftDoorwayIndex] * cos(leftDoorwayAngle)  
leftDoorwayPointY = rangeReadings[leftDoorwayIndex] * sin(leftDoorwayAngle)
```

You will also need to convert from meters to cm here.

# Open-Area-Directed Navigation

- We can even navigate through a doorway by travelling towards the open areas.
  - In an empty room, we can find the **geometric center** (a.k.a. centroid) of the room by averaging all the coordinates:



# Computing Geometric Center

- Compute geometric center by averaging all computed lidar points based on the robot's current location and orientation:

```
(x, y) = get robot's position  
angle = get robot's angle
```

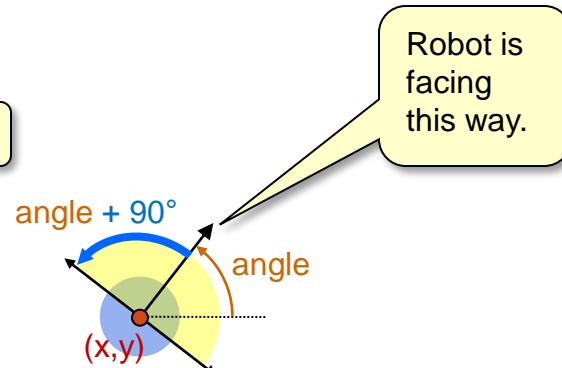
```
// This will be the geometric center point  
centerX = 0  
centerY = 0
```

Convert to cm from meters.

```
FOR lidarAngle FROM 0 TO 179 DO {  
    d = lidar reading at lidarAngle * 100  
    px = d * cos(angle + 90 - lidarAngle)  
    py = d * sin(angle + 90 - lidarAngle)  
    add px to centerX and py to centerY  
}
```

Divide both **centerX** and **centerY** by 180 to get the average

Add **x** to **centerX** and **y** to **centerY** to translate it to the robot's location



Robot is facing this way.

This will translate the computed point to be relative to the robot's (and lidar sensor's) location.

Start the  
Lab ...