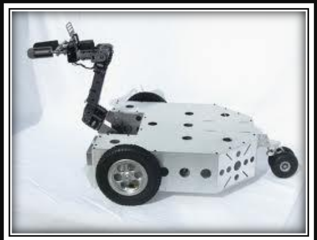
Basic Movement and Sensing

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.



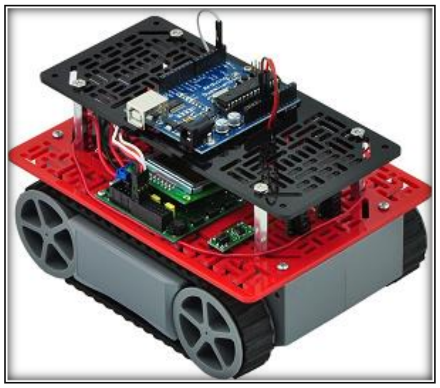


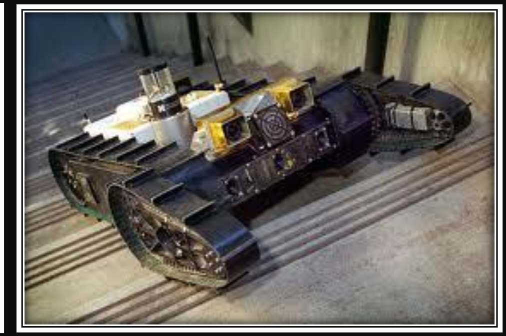
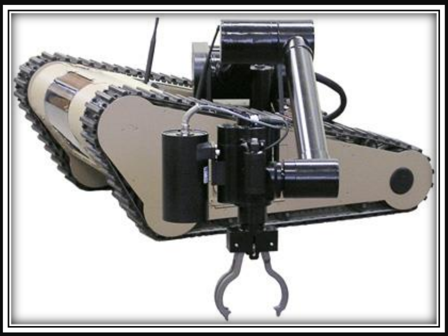




Treaded Robots

- 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.

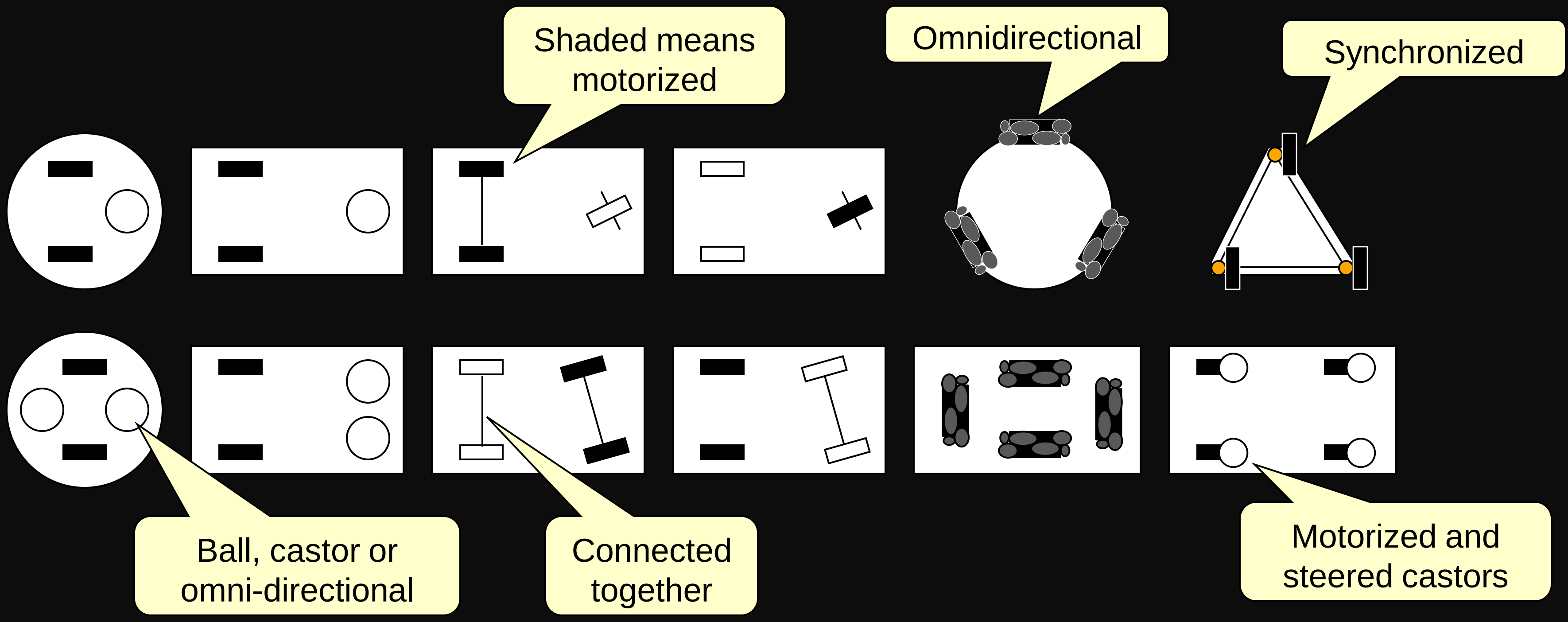
Wheel Designs

- 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

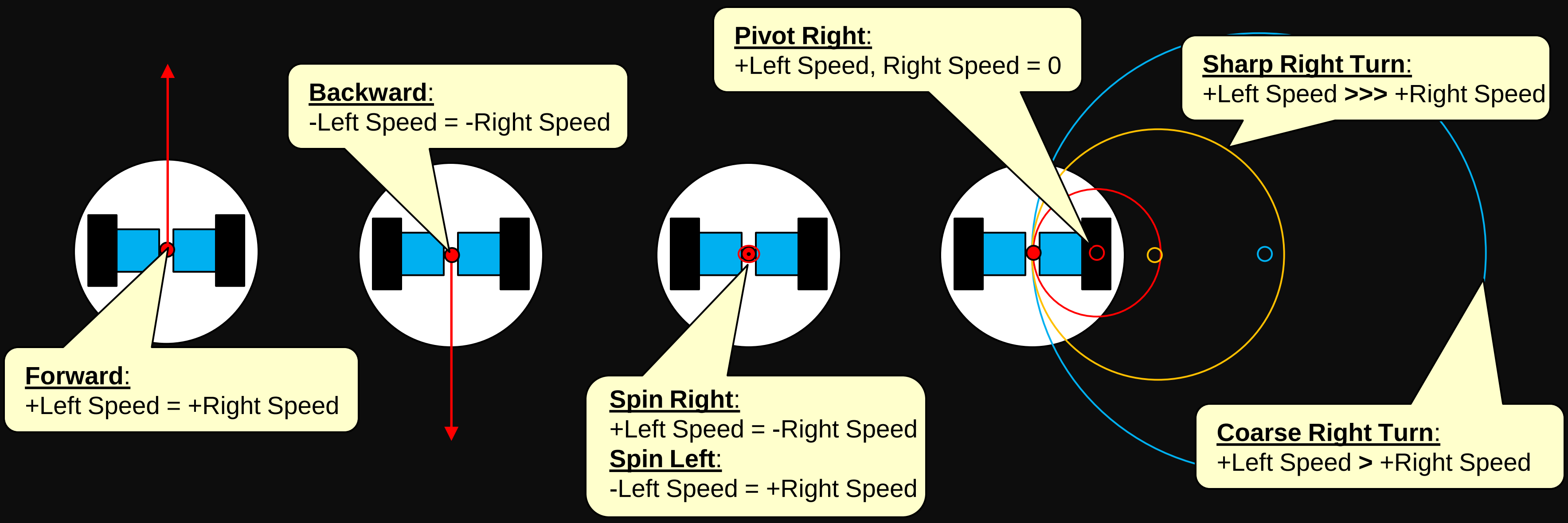


Differential Steering Robots

- Robots with two-wheels use *differential steering*

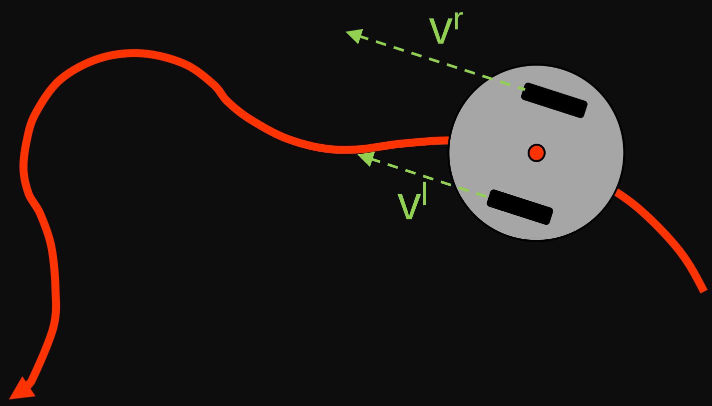
  (i.e., more drive torque is applied to one side of the vehicle than the other side).

- They are simple and easy to maneuver:



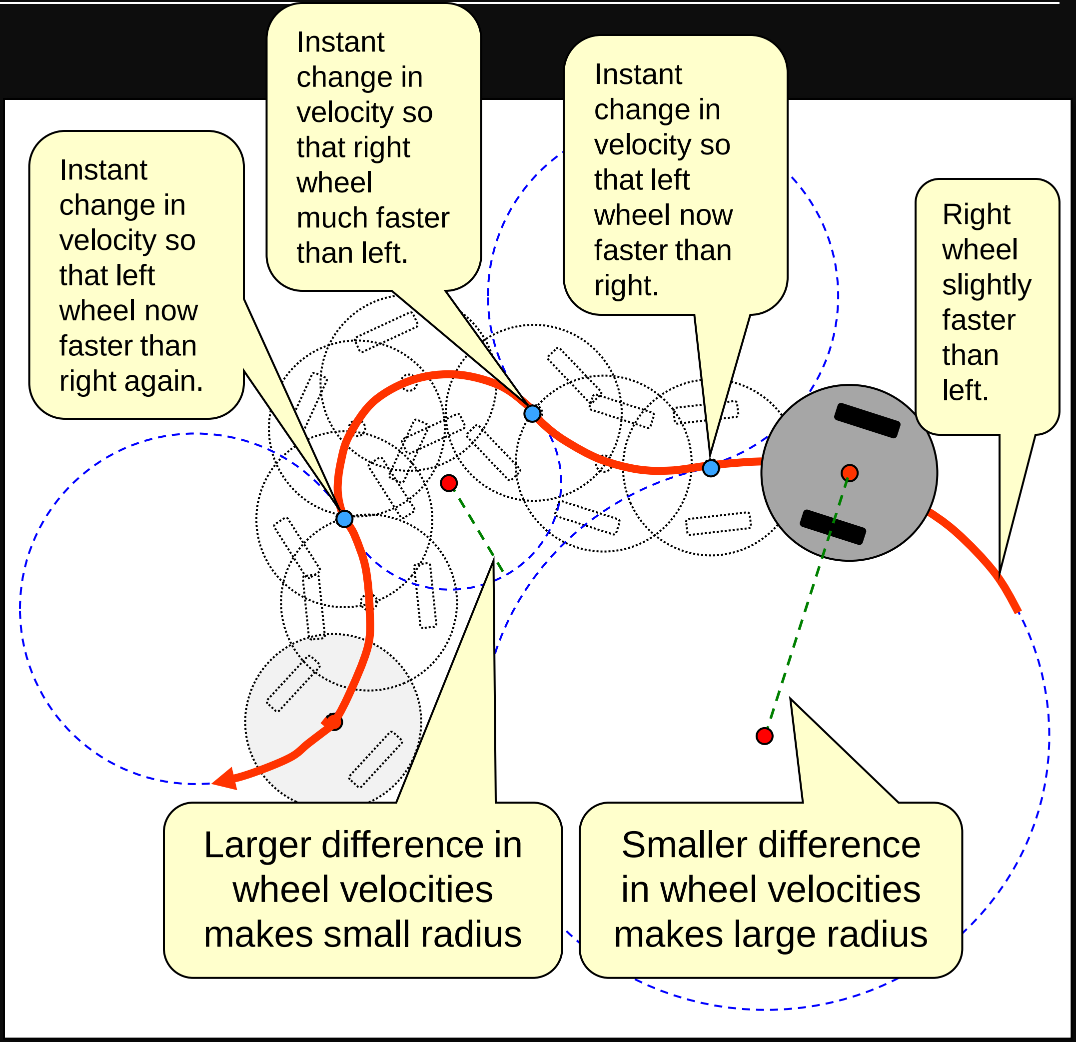
Differential Steering Robots

- 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.



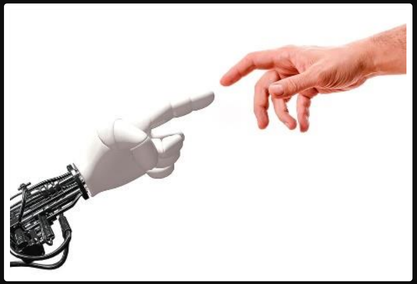
GCtronic e-puck: Motors

- Robot uses two-motor differential steering:

```java
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.

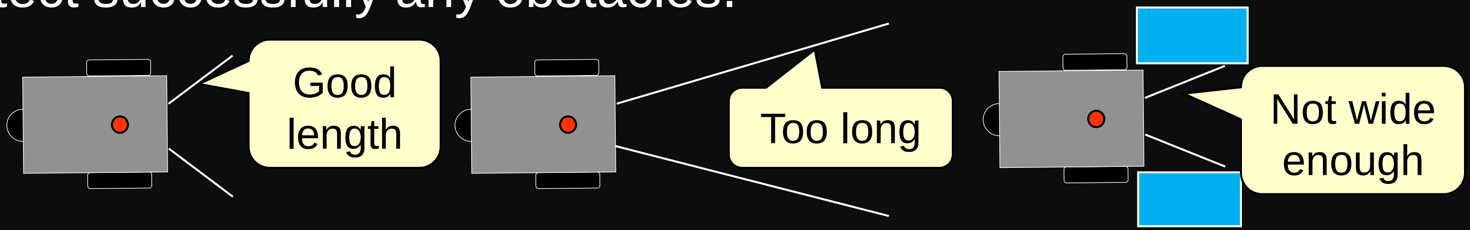
Tactile: Bumpers

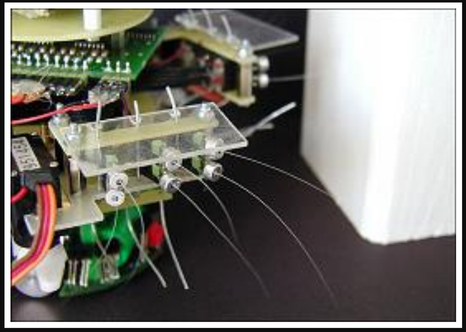
- Bumpers are simple, but they have a short detection range from from 1_{mm} to 2_{cm}.
- Unfortunately, they require physical contact with the object to detect it:
  - Can cause damage to robot depending on speed
  - Bumpers can break over time
  - Objects can be pushed or damaged

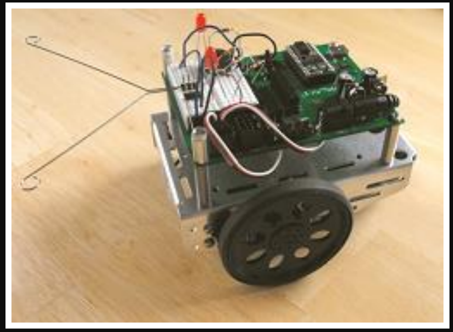
Tactile: Whiskers

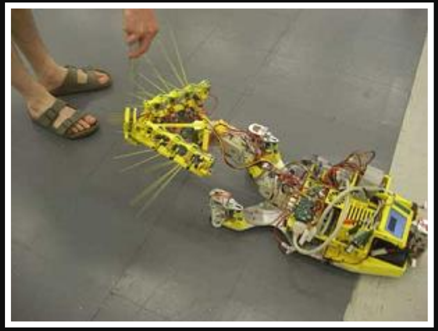
- Whiskers are like flexible bumpers
  - Usually placed at front and extend long enough to ensure safe stopping distance.
  - Should extend the entire body width so as to detect successfully any obstacles.
-

A rat constantly whisks its whisker hairs backward and forward, about 7or 8 times per second, hitting objects that bend the whiskers to various degrees. By whisker-tapping an object several times at different places, the rat can put together a 3D image in its brain.







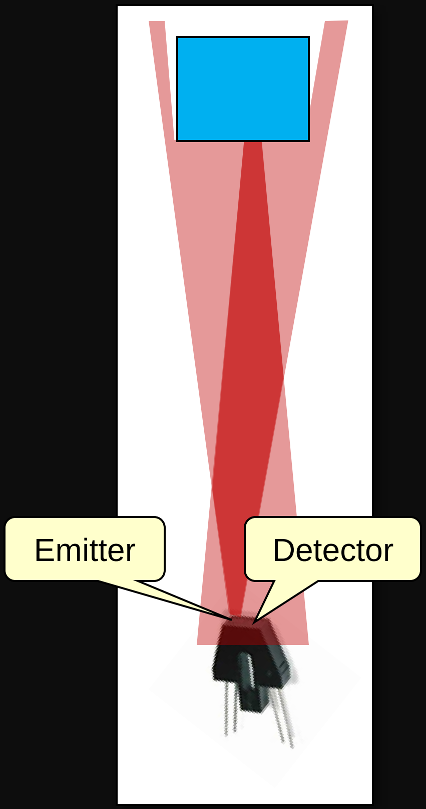


(0:17) http://www.youtube.com/watch?v=HwC2tZcOKM8

(0:47) 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

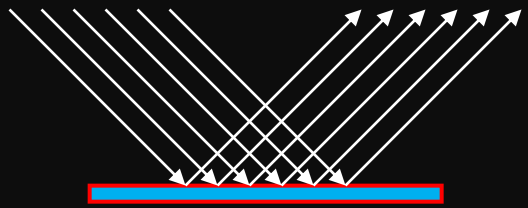


Non-Tactile: Reflective Issues

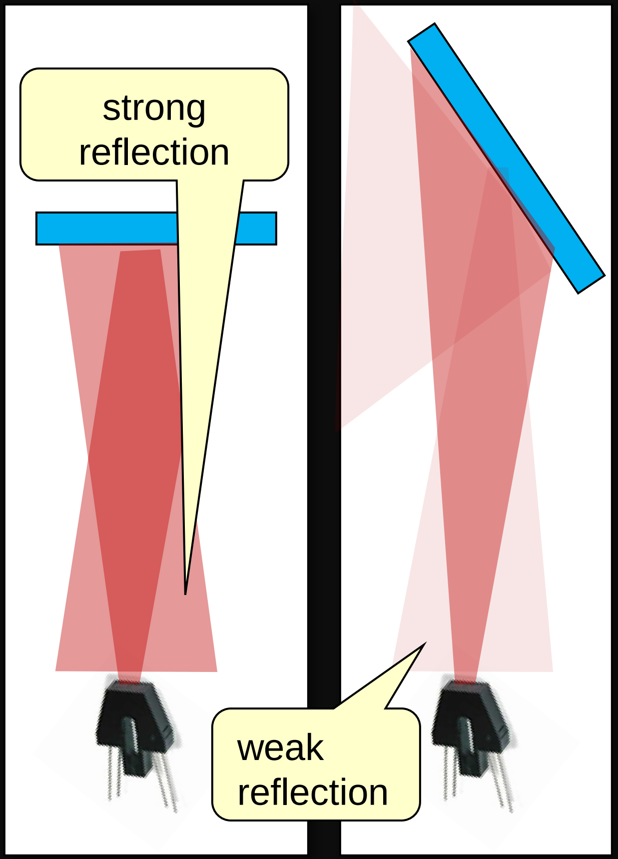
- IR is sensitive to obstacle angle
  – can result in improper detection.

- When beam's angle of incidence falls below a certain critical angle ***specular reflection*** occurs.

  – Object may not be detected

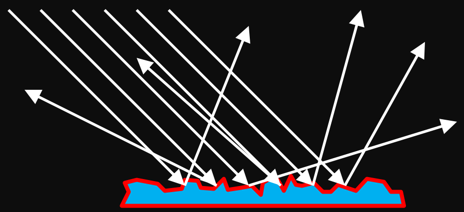


Specular Reflection
(smooth surfaces)

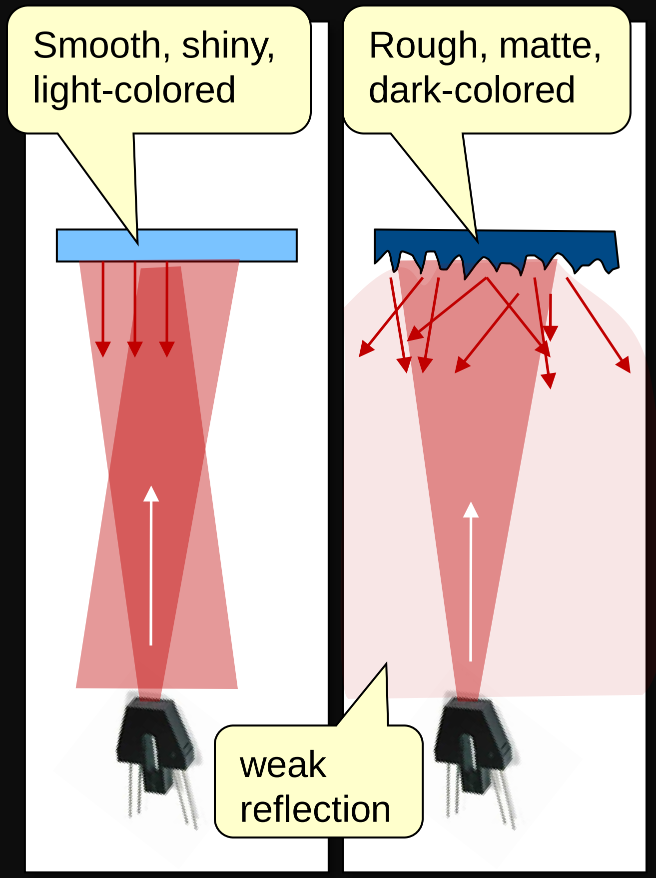


Non-Tactile: Reflective Issues

- 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.



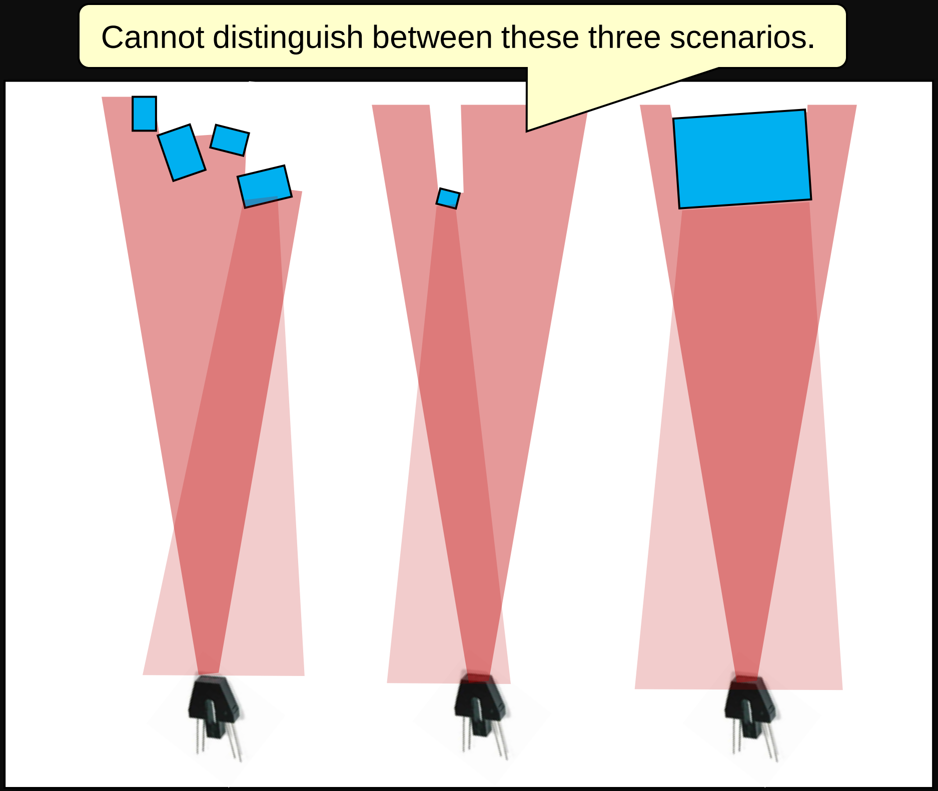
Diffuse Reflection (rough surfaces)

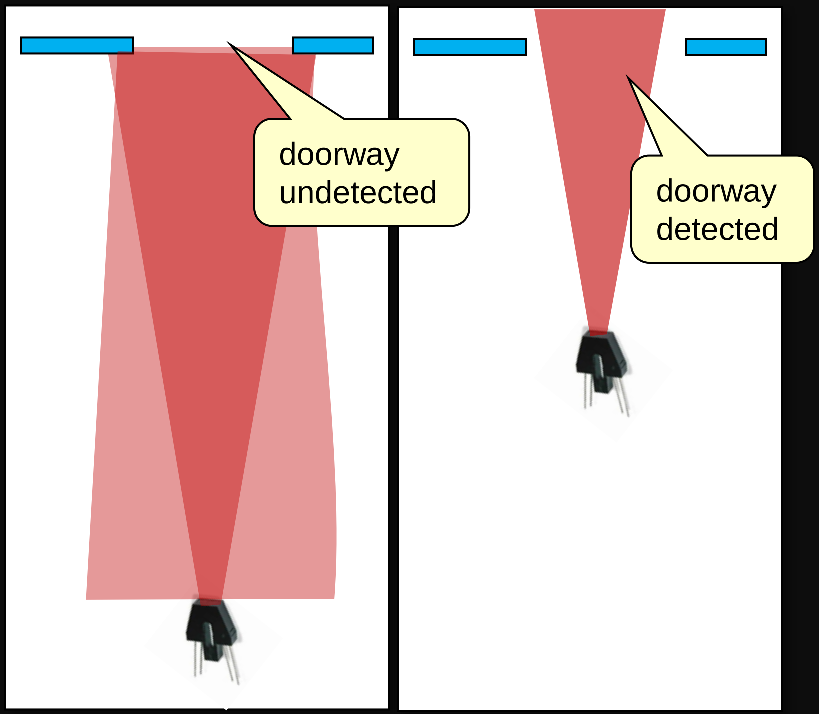


- Color also affects amount of light reflected (e.g., white reflects more than black).

Non-Tactile: Reflective Issues

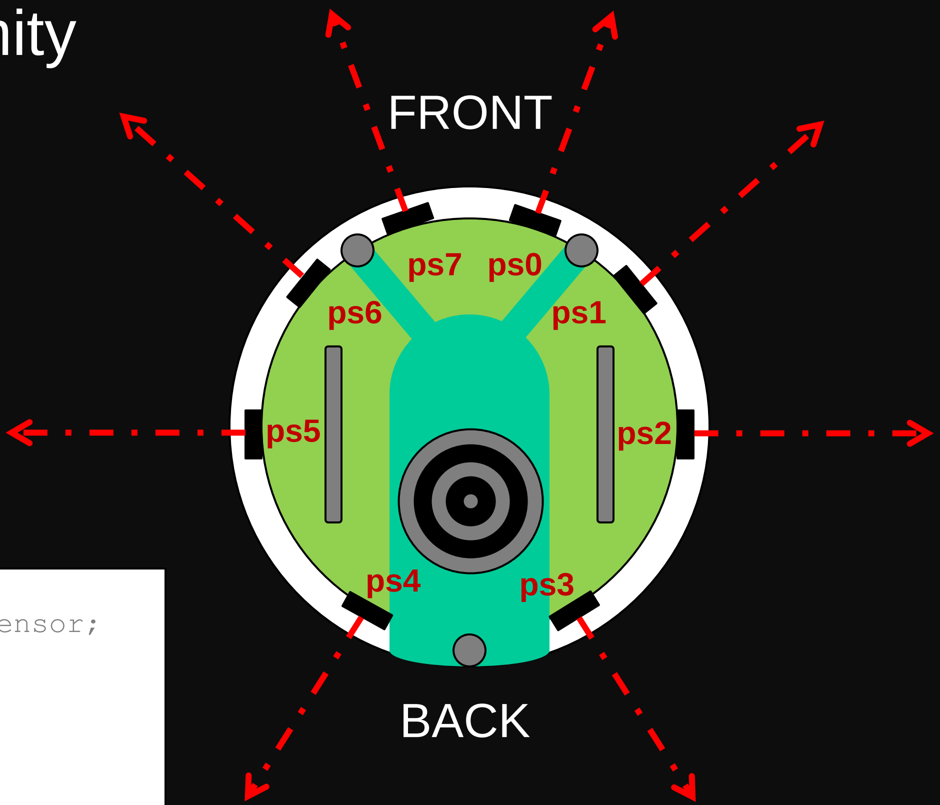
- Beam width can also play a role in obstacle detection:
  - multiple close obstacles cannot be distinguished
  - gaps cannot be detected (e.g., doorways)





GCtronic e-puck: IR sensors

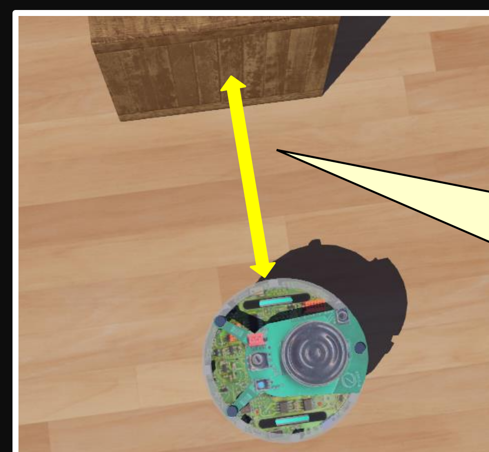
- The E-Puck robot has 8 proximity sensors around the outer ring.




```java
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

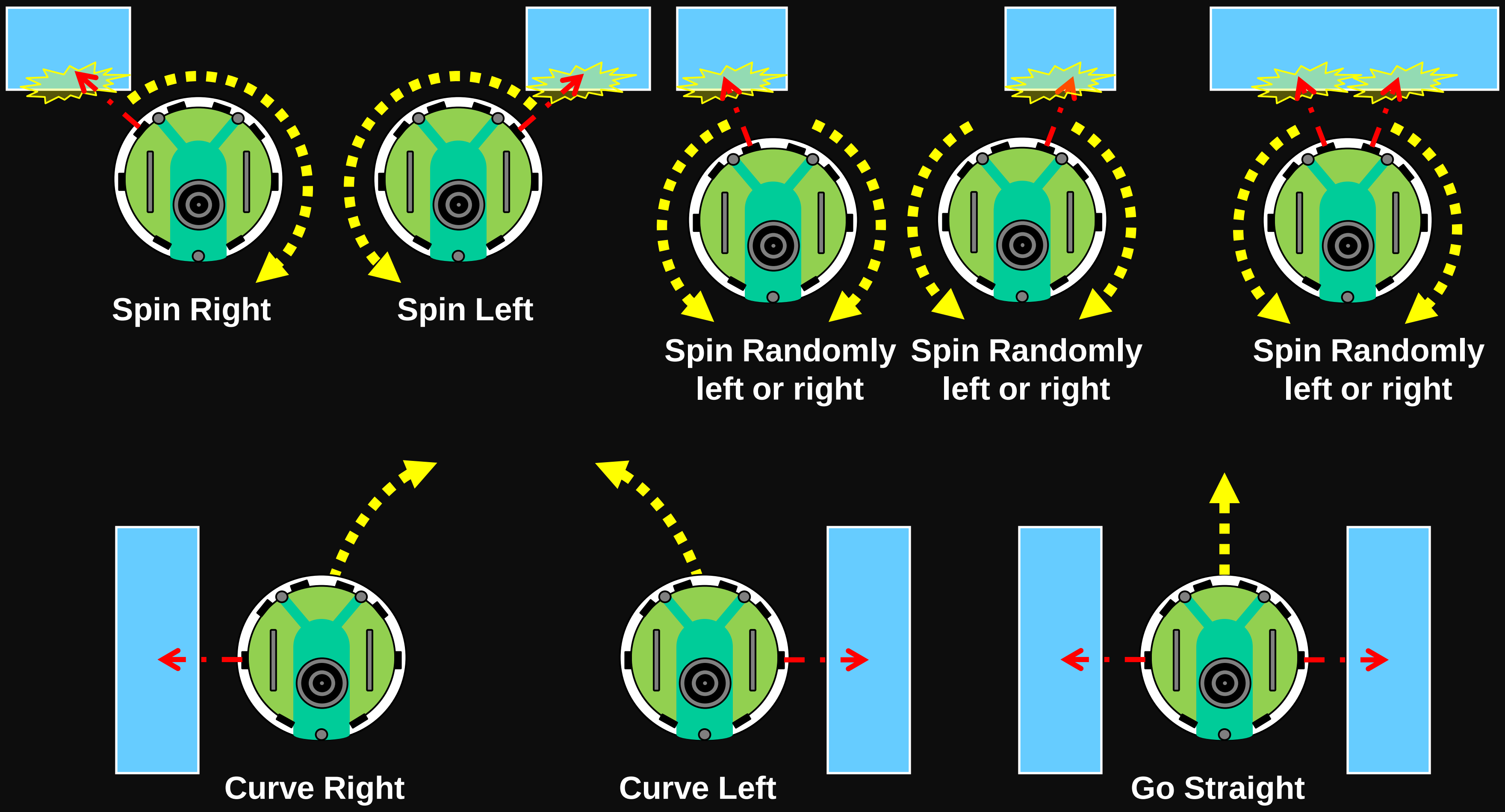
Color and texture of objects result in different values!!



When too far away, sensor still gives readings in range ~59-72.

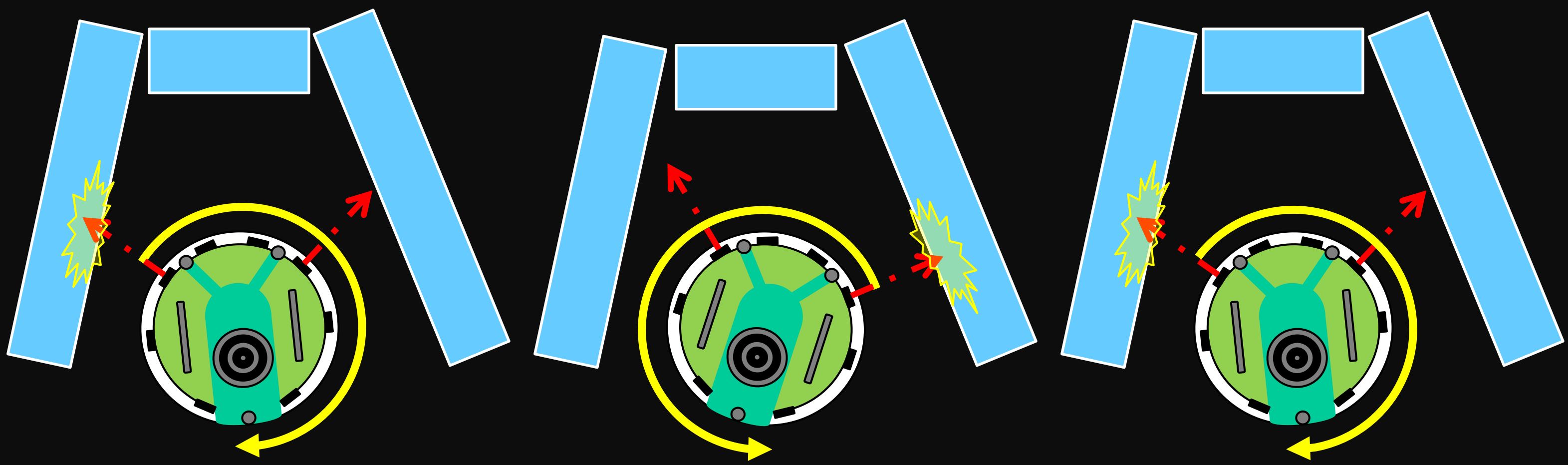
Anything less that 80 seems to indicate that “no object is detected”

Collision Detection



Collision Avoidance: Problem

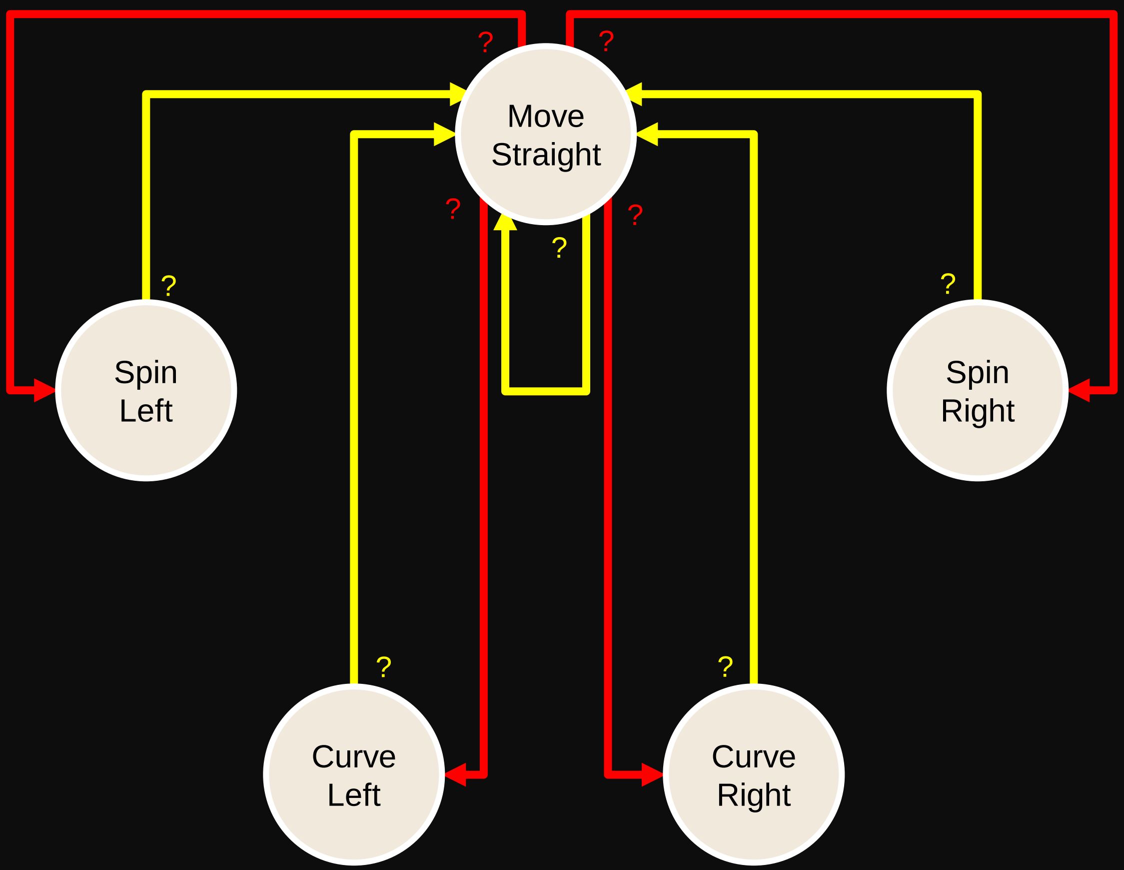
- In corners and tight spaces, the robot may end up oscillating back and forth.

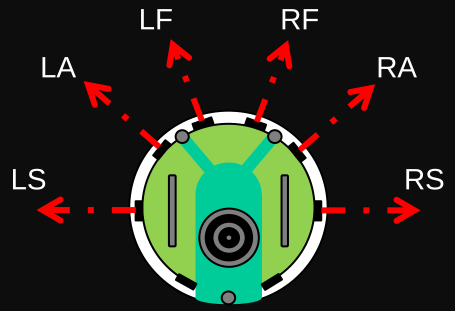


- Solution: Commit to a direction and “stick with it”!!

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 …