

Human-Robot Interaction and Autonomous Systems

AI 616

Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

Lecture 7



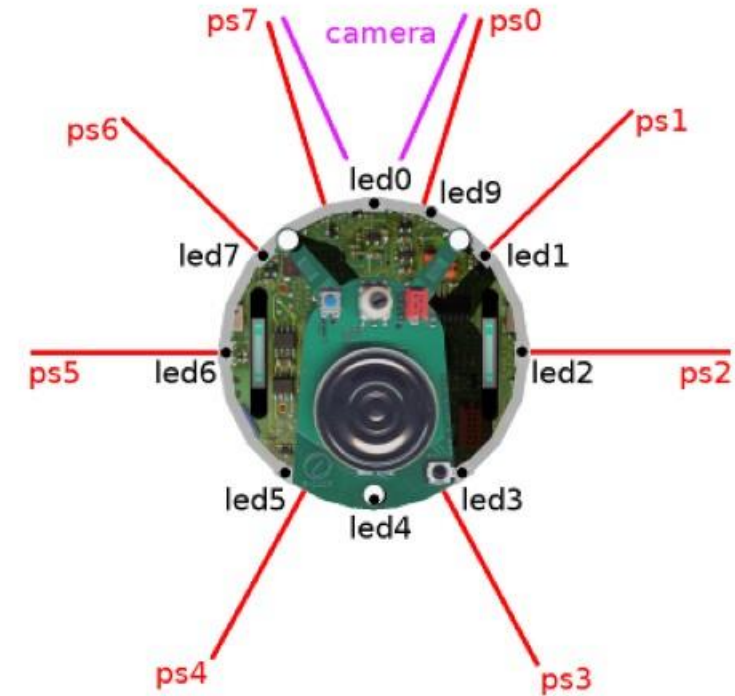
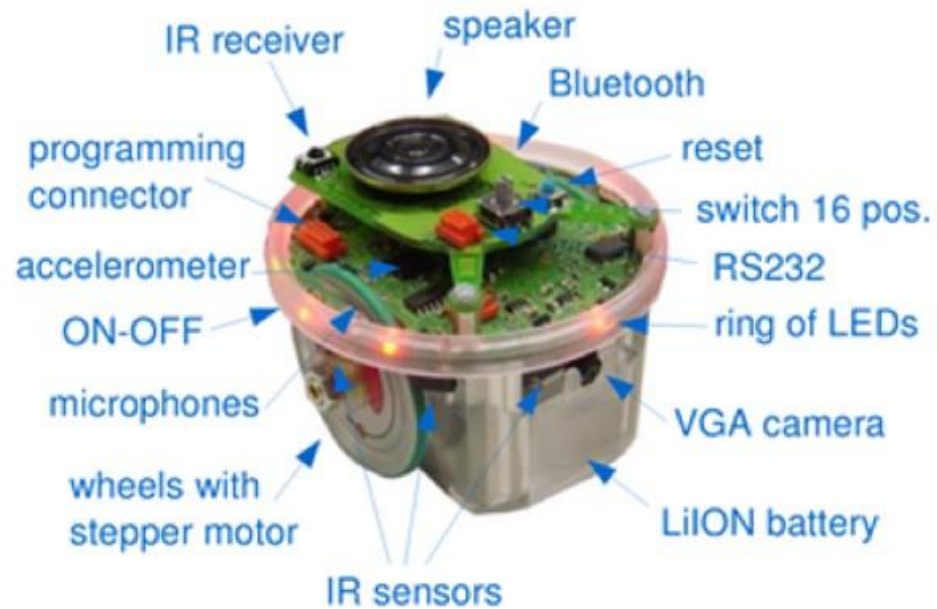


Course Outlines

Module	Content	Simulation Lab
1. Introduction to Robotics	- Fundamentals of Robot Control - Classification of Robotics - Exploration of Robot Types	- Basic Robot Control - Modeling Robot Classes - Simulating Various Robot Types
2. Advanced HRI Studies	- Introductory to Advanced Concepts in HRI	- Basic to Complex HRI Scenarios
3. Social Dynamics in Robotics	- Architectures for Social Interaction - Case Studies and History of HRI	- Social Interaction Simulations - Historical Evolution in HRI
4. Strategic HRI System Design	- Core Principles and Communication Strategies in HRI	- Designing Interfaces - Communication Protocol Simulations
5. HRI Efficiency and Effectiveness	- Metrics for Mission Effectiveness and Human Factors	- Implementing and Analyzing HRI Metrics
6. Innovative HRI System Design	- Exploring Innovative Designs and Real-World Applications	- Testing Innovative Designs - Simulating Diverse Scenarios
7. HRI System Classification	- Developing Comprehensive Classifications	- Classification Application and Case Studies
8. Human Factors in HRI	- Analyzing Ergonomics and Human Factors	- Ergonomic Simulations - Scenario-Based Human Factor Analysis
9. Robotics and AI Integration	- Theoretical Concepts and Practical AI Applications	- AI Algorithm Implementation - AI-Driven HRI Modeling
10. HRI Case Studies	- Mobile Assistive Robot Studies	- Designing and Testing Mobile Assistive Robots





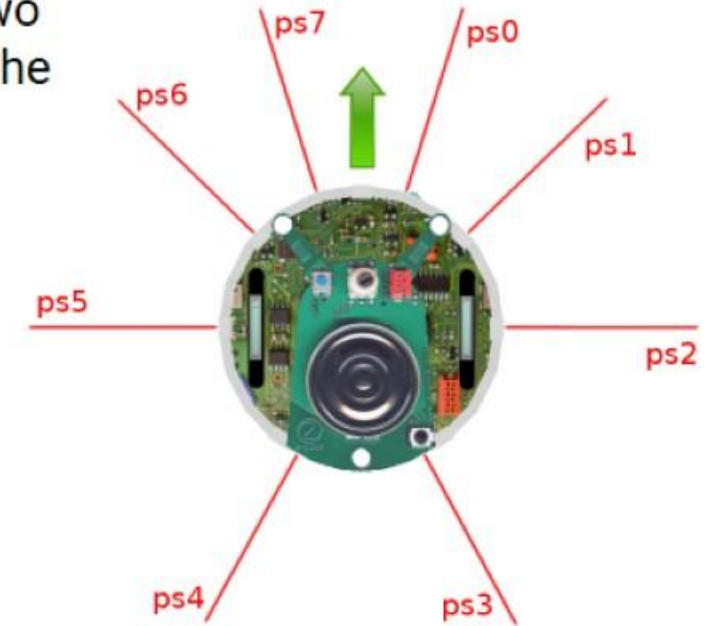


Objective: Expand obstacle detection to utilize multiple infrared sensors for more nuanced navigation.

Step 1: Controller Expansion for Multiple Sensors

Enhance the controller to read from multiple sensors and react accordingly.

Controller programming requires some information related to the e-puck model. In order to create the collision avoidance algorithm, we need to read the values of its 8 infra-red distance sensors located around its turret, and we need to actuate its two wheels. The way that the distance sensors are distributed around the turret and the e-puck direction are depicted in [this figure](#).



1. **Start:** Begin the robot control program.
2. **Initialize Robot and Motors:** Set up the robot, left and right motors, including setting positions to infinite and initial velocities to 0.
3. **Define Constants:** Specify the time step and forward speed.
4. **Initialize Sensors:** For each sensor ('ps0', 'ps1', 'ps2', 'ps7'), enable it with the defined time step and add it to the sensor objects list.
5. **Control Loop Begins:** Enter the main control loop that runs at each time step.
 - **Read Sensor Values:** Get distance readings from all initialized sensors.
 - **Obstacle Detection:** Check if there's an obstacle in front or to the sides based on sensor values.
 - If an obstacle is detected in front or front-right, **Turn Left**.
 - If an obstacle is detected in front-left, **Turn Right**.
 - If no obstacles are detected, **Move Forward**.
6. **Repeat Control Loop:** Continue the loop until the program ends.
7. **End:** The program concludes when manually stopped or when the robot's task is completed.

```
# Import necessary classes for the robot control
from controller import Robot, Motor, DistanceSensor
```

```
# Initialize the Robot instance
robot = Robot()
```

```
# Define the time step of the robot's control loop
TIME_STEP = 64
```

```
# Initialize motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set to infinite position mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
```

```
# Initial velocity
leftMotor.setVelocity(0.0)
rightMotor.setVelocity(0.0)
```

```
# Speed constants
FORWARD_SPEED = 5.0
```

```
# Initialize multiple distance sensors by their names
sensors = ['ps0', 'ps1', 'ps2', 'ps7']
sensorObjects = []
for sensorName in sensors:
```

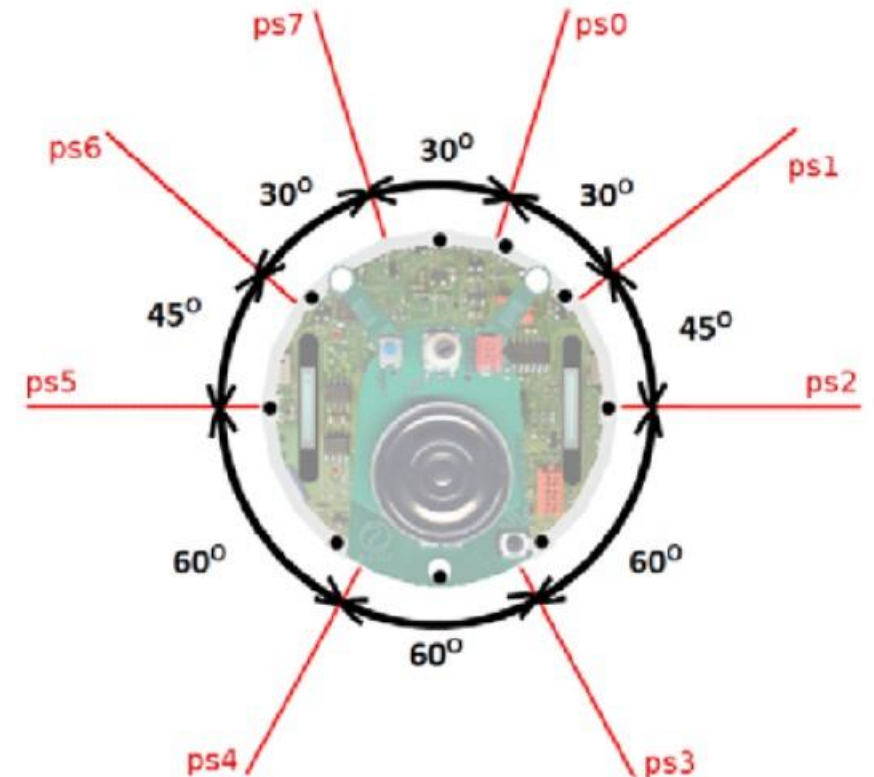
```
    sensor = robot.getDevice(sensorName) # Get the sensor device
    sensor.enable(TIME_STEP) # Enable the sensor with the defined time step
    sensorObjects.append(sensor) # Append the sensor object to the list
```

```
# Control loop: execute at each simulation step
while robot.step(TIME_STEP) != -1:
    # Read sensor values into a list
    sensorValues = [sensor.getValue() for sensor in
                     sensorObjects]
```

```
# Basic obstacle detection and navigation strategy
if sensorValues[0] < 70 or sensorValues[1] < 70:
    # Obstacle detected in front or front-right: turn left
    leftMotor.setVelocity(-FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
elif sensorValues[3] < 800:
    # Obstacle detected in front-left: turn right
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(-FORWARD_SPEED)
else:
    # No obstacle detected: move forward
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
```

Initializing multiple distance sensors on a robot

1. **Define Sensor Names:** Start by listing the names of the sensors you're going to use. For instance, ``sensors = ['ps0', 'ps1', 'ps2', 'ps7']`` includes four sensors located at different positions on the robot.
2. **Create a List for Sensor Objects:** Prepare a list to store the sensor objects. This is done with ``sensorObjects = []``.
3. **Loop Through Each Sensor Name:**
 - Use a ``for`` loop to iterate over each sensor name in the ``sensors`` list.
4. **Get the Sensor Device:**
 - Inside the loop, use ``robot.getDevice(sensorName)`` to get each sensor based on its name. This function connects your code to the specific hardware sensor on the robot.
5. **Enable the Sensor:**
 - Call ``sensor.enable(TIME_STEP)`` to activate the sensor, allowing it to start collecting data. The ``TIME_STEP`` specifies how often (in milliseconds) the sensor should update its data.
6. **Store Sensor Objects:**
 - Add the initialized and enabled sensor to the ``sensorObjects`` list with ``sensorObjects.append(sensor)``. This step is crucial for easily accessing each sensor's data later in your code.



Explain the process of reading sensor values into a list

```
sensorValues = [sensor.getValue() for sensor in sensorObjects]
```

1. **List Comprehension:** This line uses a Python feature called "list comprehension" to create a new list. It's a concise way to apply an operation (in this case, `sensor.getValue()`) to each item in an existing list (`sensorObjects`).
2. **`sensorObjects` List:** `sensorObjects` is a list that contains all the distance sensor objects you previously initialized. Each `sensor` in this list is an individual distance sensor object.
3. **`sensor.getValue()`:** For each `sensor` in the `sensorObjects` list, `.getValue()` is called. This method retrieves the current reading from the sensor, which is a numerical value representing the distance to the nearest object.
4. **Creating `sensorValues` List:** The numerical values retrieved by `sensor.getValue()` for each sensor are collected into a new list named `sensorValues`. This list now contains the current distance reading from each of the sensors in `sensorObjects`.
5. **Why This Matters:** By gathering sensor readings into `sensorValues`, you can easily check and react to the robot's surroundings. For example, you can use these values to decide if the robot should continue moving forward, turn left, or turn right to avoid obstacles.

This approach simplifies accessing and using sensor data, making your robot's control logic clearer and easier to manage, especially for tasks like obstacle avoidance.

The obstacle detection and navigation strategy uses sensor readings to control the robot's movement. When sensors 0 or 1 detect an obstacle (values below 800), indicating something in front or to the front-right, the robot turns left by setting a negative velocity for the left motor and a positive velocity for the right motor. If sensor 3 detects an obstacle, implying an obstruction is to the front-left, it turns right by reversing these velocities. If no obstacles are detected, both motors propel the robot forward at the same speed. This basic logic helps the robot navigate by avoiding collisions.

```
# Basic obstacle detection and navigation strategy
if sensorValues[0] < 800 or sensorValues[1] < 800:
    # Obstacle detected in front or front-right: turn left
    leftMotor.setVelocity(-FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
elif sensorValues[3] < 800:
    # Obstacle detected in front-left: turn right
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(-FORWARD_SPEED)
else:
    # No obstacle detected: move forward
    leftMotor.setVelocity(FORWARD_SPEED)
    rightMotor.setVelocity(FORWARD_SPEED)
```

```

# Import necessary classes for the robot control
from controller import Robot, Motor, DistanceSensor

# Initialize the Robot instance
robot = Robot()

# Define the time step of the robot's control loop
TIME_STEP = 64

# Initialize motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf')) # Set to infinite position mode
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0.0) # Initial velocity
rightMotor.setVelocity(0.0)

# Speed constants
FORWARD_SPEED = 5.0

# Initialize multiple distance sensors by their names
sensors = ['ps0', 'ps1', 'ps2', 'ps7']
sensorObjects = []
for sensorName in sensors:
    sensor = robot.getDevice(sensorName) # Get the sensor device
    sensor.enable(TIME_STEP) # Enable the sensor with the defined time step
    sensorObjects.append(sensor) # Append the sensor object to the list

```

```

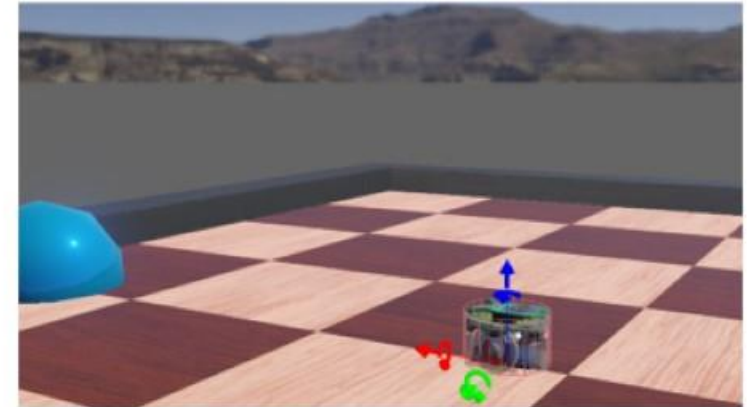
# Control loop: execute at each simulation step
while robot.step(TIME_STEP) != -1:
    # Read sensor values into a list
    sensorValues = [sensor.getValue() for sensor in sensorObjects]

    # Basic obstacle detection and navigation strategy
    if sensorValues[0] < 800 or sensorValues[1] < 800:
        # Obstacle detected in front or front-right: turn left
        leftMotor.setVelocity(-FORWARD_SPEED)
        rightMotor.setVelocity(FORWARD_SPEED)
    elif sensorValues[3] < 800:
        # Obstacle detected in front-left: turn right
        leftMotor.setVelocity(FORWARD_SPEED)
        rightMotor.setVelocity(-FORWARD_SPEED)
    else:
        # No obstacle detected: move forward
        leftMotor.setVelocity(FORWARD_SPEED)
        rightMotor.setVelocity(FORWARD_SPEED)

```

Robot that flashes its LED red and white three times before turning right when it detects an obstacle.

1. **Start:** Begin the robot program.
2. **Initialize the Robot:** Set up the robot's motors to an infinite position and velocity to 0.
3. **Define Time Step:** Set the simulation time step (64 ms in this case).
4. **Initialize Sensors:** Enable distance sensors (ps0 and ps7) for obstacle detection.
5. **Initialize LED:** Set up an LED for visual feedback.
6. **Enter Control Loop:** Start the main loop that runs every simulation step.
 - **Read Sensor Values:** Check the distance values from sensors ps0 and ps7.
 - **Detect Obstacle:** If both sensors detect an obstacle (values less than 70),
 - **Flash LED:** Turn the LED on and off three times to signal obstacle detection.
 - **Turn Right:** Adjust motor velocities to turn the robot right.
 - **Else,**
 - **Move Forward:** Set both motors to move the robot forward.
7. **Repeat Control Loop:** Continue the loop until the program ends.
8. **End:** Terminate the program.



```
from controller import Robot, LED, DistanceSensor
```

```
# Initialize the robot  
robot = Robot()
```

```
# Define the time step  
TIME_STEP = 64
```

```
# Initialize the LEDs on the robot  
led = robot.getDevice('led0') # Assuming the robot has an LED named  
'led0'
```

```
# Initialize the distance sensors  
sensor0 = robot.getDevice('ps0')  
sensor7 = robot.getDevice('ps7')  
sensor0.enable(TIME_STEP)  
sensor7.enable(TIME_STEP)
```

```
# Initialize motors, assuming the robot has motors named 'left wheel  
motor' and 'right wheel motor'  
leftMotor = robot.getDevice('left wheel motor')  
rightMotor = robot.getDevice('right wheel motor')  
leftMotor.setPosition(float('inf'))  
rightMotor.setPosition(float('inf'))  
leftMotor.setVelocity(0)  
rightMotor.setVelocity(0)
```

```
# Function to flash LED three times
```

```
def flashLED(led):  
    for _ in range(3):  
        led.set(1) # Turn LED red  
        robot.step(TIME_STEP)  
        led.set(0) # Turn LED off (or white if your robot supports it)  
        robot.step(TIME_STEP)
```

```
# Main control loop
```

```
while robot.step(TIME_STEP) != -1:
```

```
    # Read sensor values  
    value0 = sensor0.getValue()  
    value7 = sensor7.getValue()
```

```
    # Check if there is an obstacle in front  
    if value0 < 70 and value7 < 70:
```

```
        flashLED(led) # Flash LED if obstacle is detected
```

```
        # Turn right
```

```
        leftMotor.setVelocity(5) # Adjust velocity to turn the robot
```

```
        rightMotor.setVelocity(-5)
```

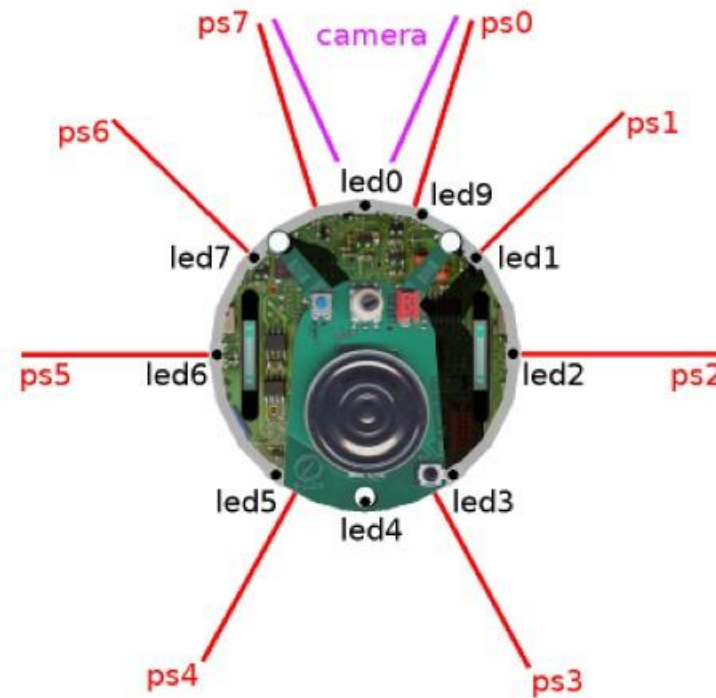
```
    else:
```

```
        # Move forward if no obstacle
```

```
        leftMotor.setVelocity(5)
```

```
        rightMotor.setVelocity(5)
```

E-puck robot can use its camera to capture an image and analyze the color of the central pixel in the Webots simulation environment.



```
# Import necessary modules from the Webots library.
# Robot is the main class for robot control.
# Camera is for accessing the robot's camera.
from controller import Robot, Camera

# Define the simulation time step in milliseconds.
# This is how often the simulation updates the robot's sensors and actuators.
TIME_STEP = 64

# Create an instance of the Robot class.
# This will be our main handle to interact with the robot.
robot = Robot()

# Get a handle to the camera device on the robot.
# 'camera' is the name of the camera device specified in the robot's model.
# If the camera has a different name in your project, change 'camera' accordingly.
camera = robot.getDevice('camera')

# Enable the camera to start capturing images.
# The camera will update its image every TIME_STEP milliseconds.
camera.enable(TIME_STEP)

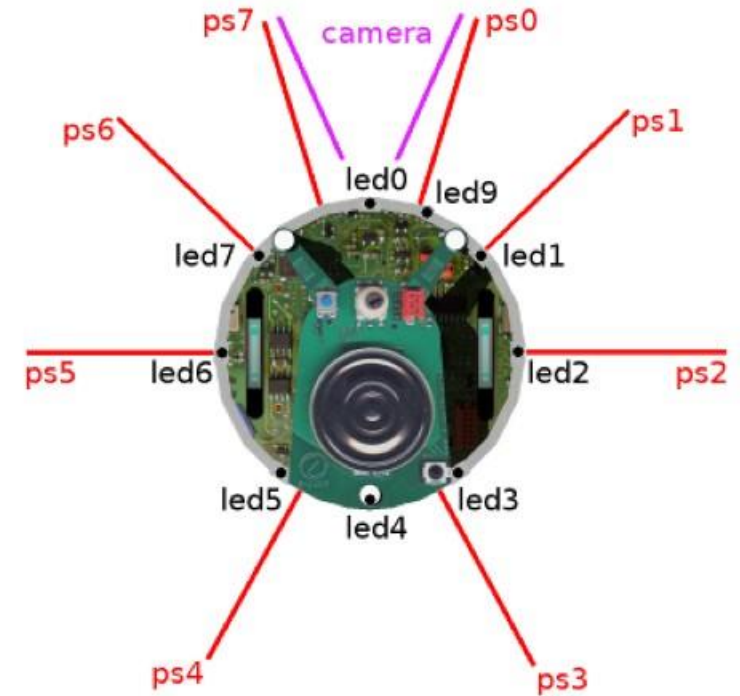
# Main loop: Execute actions repeatedly until the simulation is stopped.
while robot.step(TIME_STEP) != -1:
    # Capture an image from the camera.
    # The getImage() method returns the latest image captured by the camera.
    image = camera.getImage()
```

```
# Retrieve the width and height of the camera image.
# These values are set in the Webots world file and can be used to process the
image.
cameraWidth = camera.getWidth()
cameraHeight = camera.getHeight()
print(f'Captured an image of size: {cameraWidth}x{cameraHeight}')

# Calculate the position of the central pixel in the image.
# We divide the width and height by 2 to find the center.
centerX = cameraWidth // 2
centerY = cameraHeight // 2

# Get the RGB color values of the central pixel.
# imageGetRed, imageGetGreen, and imageGetBlue methods are used to
extract
# the color components of a pixel at a specified position.
red = camera.imageGetRed(image, cameraWidth, centerX, centerY)
green = camera.imageGetGreen(image, cameraWidth, centerX, centerY)
blue = camera.imageGetBlue(image, cameraWidth, centerX, centerY)

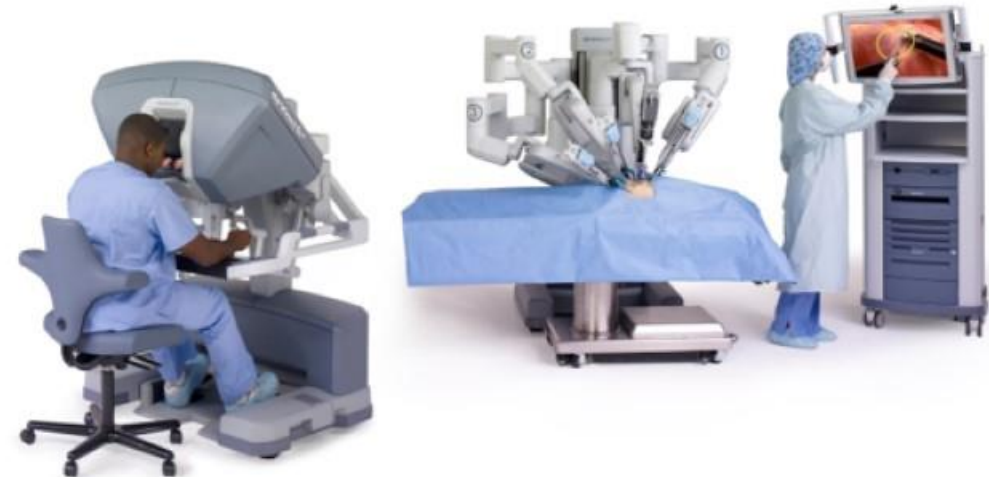
# Print the RGB color values of the central pixel.
# This can be useful for tasks like color detection or analysis.
print(f'Color of the central pixel: R={red}, G={green}, B={blue}')
```



"Innovative Designs and Real-World Applications of Robotics and AI"



1. Robotic Surgery: Surgical robots, like the da Vinci Surgical System, allow surgeons to perform complex procedures with more precision, flexibility, and control than traditional techniques. These robots offer minimally invasive options, reducing recovery time and improving patient outcomes.



2.Care Robots: Robots designed for care settings assist with tasks ranging from patient monitoring to delivering therapy, helping to alleviate the workload on human caregivers and enhancing patient care.



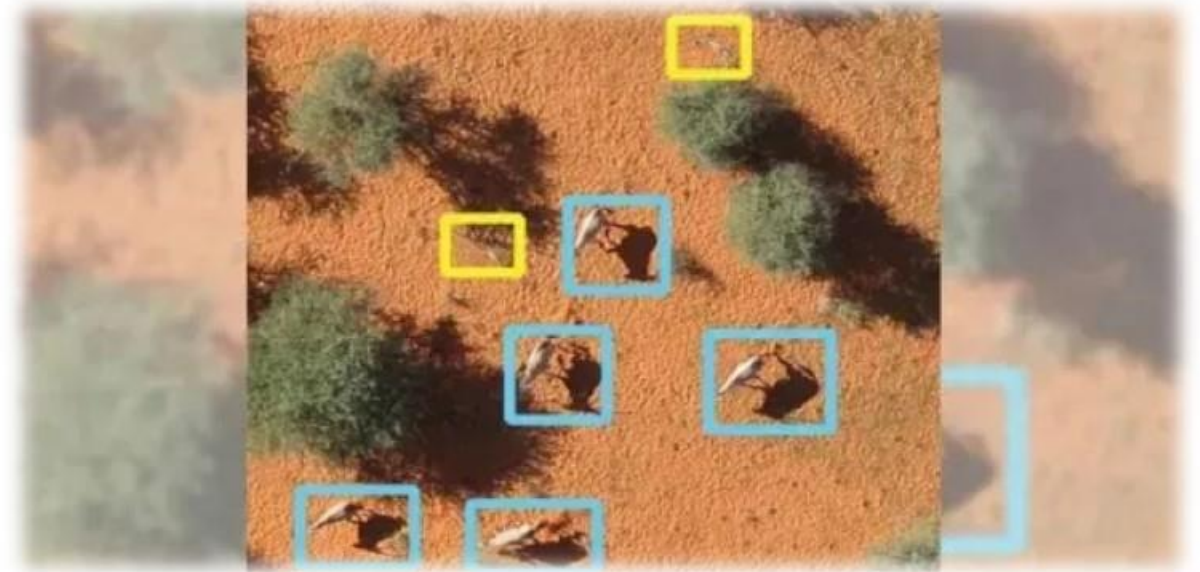
Agricultural Drones and Robots: These technologies revolutionize farming practices by providing detailed aerial views that help in crop monitoring, health assessment, and targeted pesticide spraying. Ground-based robots automate tasks like weeding, planting, and harvesting.



Precision Agriculture: AI-driven analysis of data collected from sensors and drones enables farmers to make informed decisions about water management, crop rotation, and optimal planting strategies, leading to increased crop yield and sustainability.



Wildlife Conservation: AI algorithms analyze images and data from camera traps to monitor wildlife, track poaching activities, and study biodiversity, contributing to conservation efforts.



Ocean Cleanup: Innovative robotic systems are deployed to collect plastic and other waste from oceans and rivers, addressing the critical issue of water pollution.



Educational Robots: Robots serve as teaching assistants or learning platforms, offering interactive and personalized learning experiences. They help in teaching coding, STEM subjects, and languages, making education more engaging.

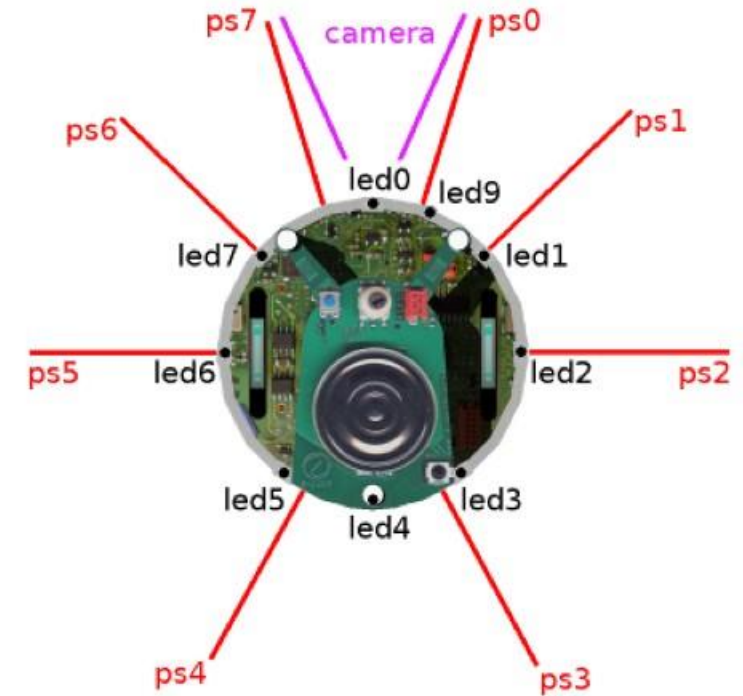


Traffic Management: AI-driven traffic control systems analyze real-time data to optimize traffic flow, reduce congestion, and improve road safety.

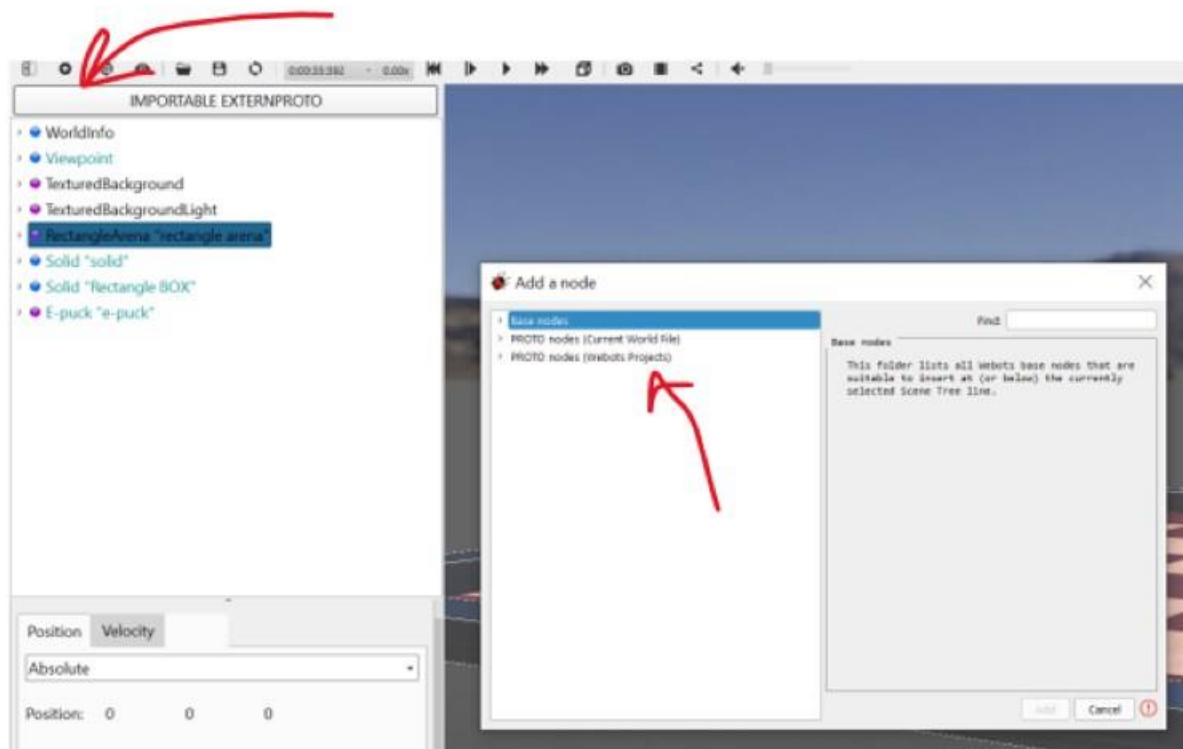


Waste Management: Smart waste collection systems use sensors to monitor trash levels and optimize collection routes, improving efficiency and cleanliness.





ADD Obstacle (ROCK) in Webots



Add a node

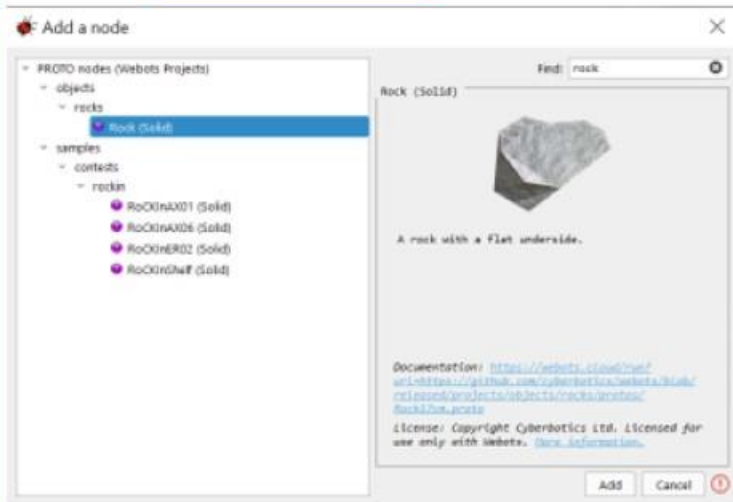
- > Base nodes
- > PROTO nodes (Current World File)
- > PROTO nodes (Webots Projects)
 - > bounding_objects
 - > default
 - > devices
 - > humans
 - > objects
 - > advertising_board
 - > animals
 - > apartment_structure
 - > backgrounds
 - > balls
 - > bathroom
 - > bedroom
 - > buildings
 - > cabinet
 - > chairs
 - > computers
 - > create_wall
 - > drinks
 - > factory
 - > floors



Add a node

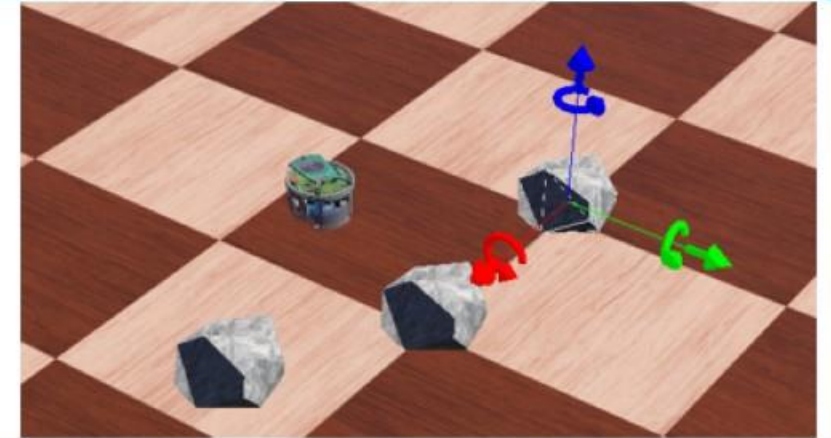
- > garden
- > gear
- > hospital
- > kitchen
- > lego
- > lights
- > living_room_furniture
- > mirror
- > obstacles
 - > OilBarrel (Solid)
 - > Ramp30deg (Solid)
- > paintings
- > panels
- > plants
- > road
- > robotstadium

+ > Add a node > PROTO nodes>Objects > search rocks >

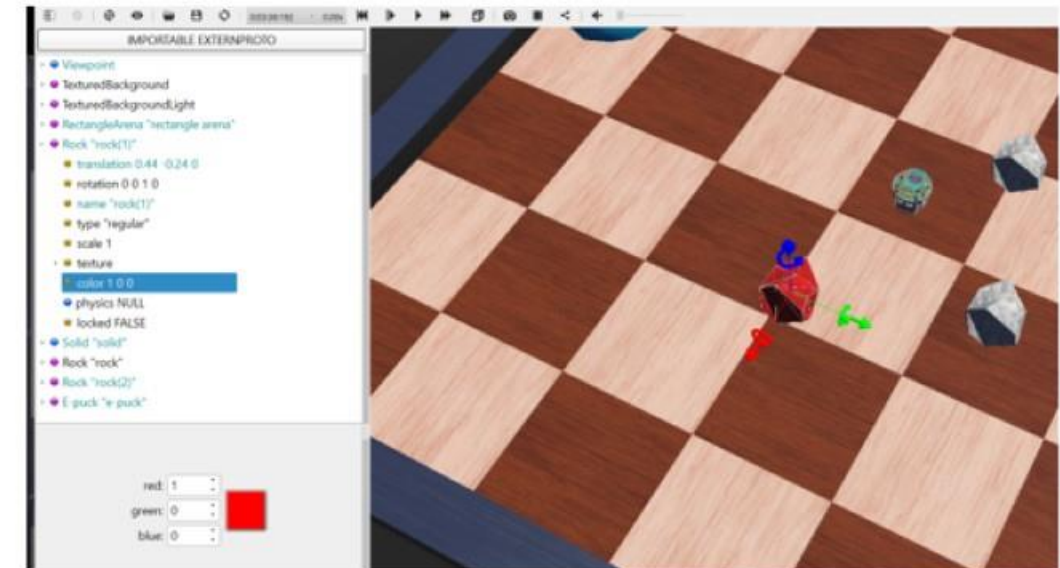


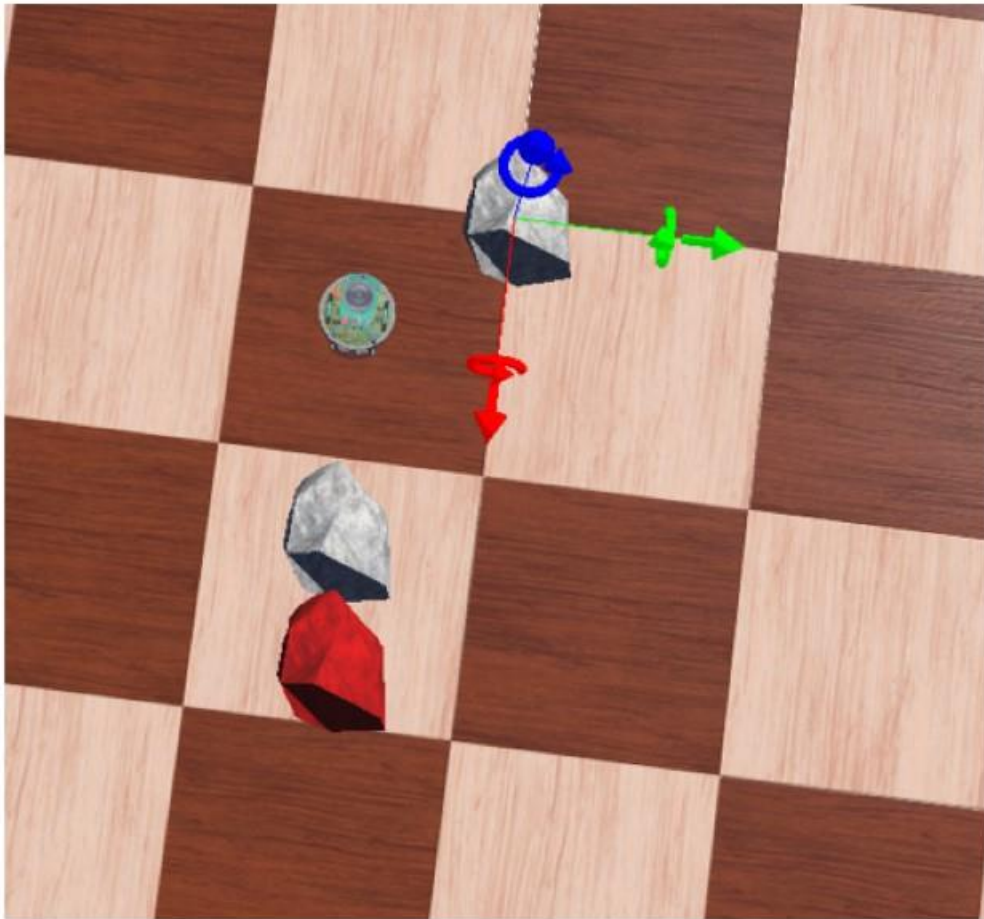
Click on the Arrows to move the object

Ctrl + V to copy paste the object



Change the color of Rock





1- Change the Location of the Gray Object to be in front of the Epuck Robot.

2- Read the RGB Values of the center of Camera

```
from controller import Robot, Camera
```

```
# Time step definition
```

```
TIME_STEP = 64
```

```
# Create the Robot instance
```

```
robot = Robot()
```

```
# Initialize the camera and enable it  
camera = robot.getDevice('camera')  
camera.enable(TIME_STEP)
```

```
# Main control loop
```

```
while robot.step(TIME_STEP) != -1:
```

```
    # Capture an image from the camera
```

```
    image = camera.getImage()
```

```
# Assuming you've set the resolution in the Webots world file
```

```
cameraWidth = camera.getWidth()
```

```
cameraHeight = camera.getHeight()
```

```
print(f"Captured an image of size: {cameraWidth}x{cameraHeight}")
```

```
# Get the color of the central pixel
```

```
centerX = cameraWidth // 2
```

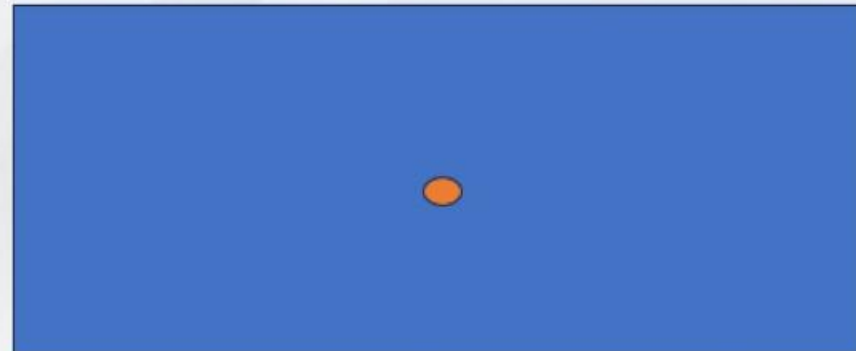
```
centerY = cameraHeight // 2
```

```
red = camera.imageGetRed(image, cameraWidth, centerX, centerY)
```

```
green = camera.imageGetGreen(image, cameraWidth, centerX, centerY)
```

```
blue = camera.imageGetBlue(image, cameraWidth, centerX, centerY)
```

```
print(f"Color of the central pixel: R={red}, G={green}, B={blue}")
```



- ▶ TexturedBackgroundLight
- ▶ RectangleArena "rectangle arena"
- ▶ Rock "rock(1)"
- ▶ Solid "solid"
- ▶ Rock "rock"
- ▶ Rock "rock(2)"
- ▶ E-puck "e-puck"
 - translation 0.219 -0.136 -4.41e-05
 - rotation 0.0166 -0.911 -0.413 0.00373
 - name "e-puck"
 - controller "my_camera"
- ▶ controllerArgs
 - window "e-puck"
 - customData ""
 - supervisor FALSE
 - synchronization TRUE

Node	Position	Velocity
Absolute		
Position:	0	0
Rotation:	1	0

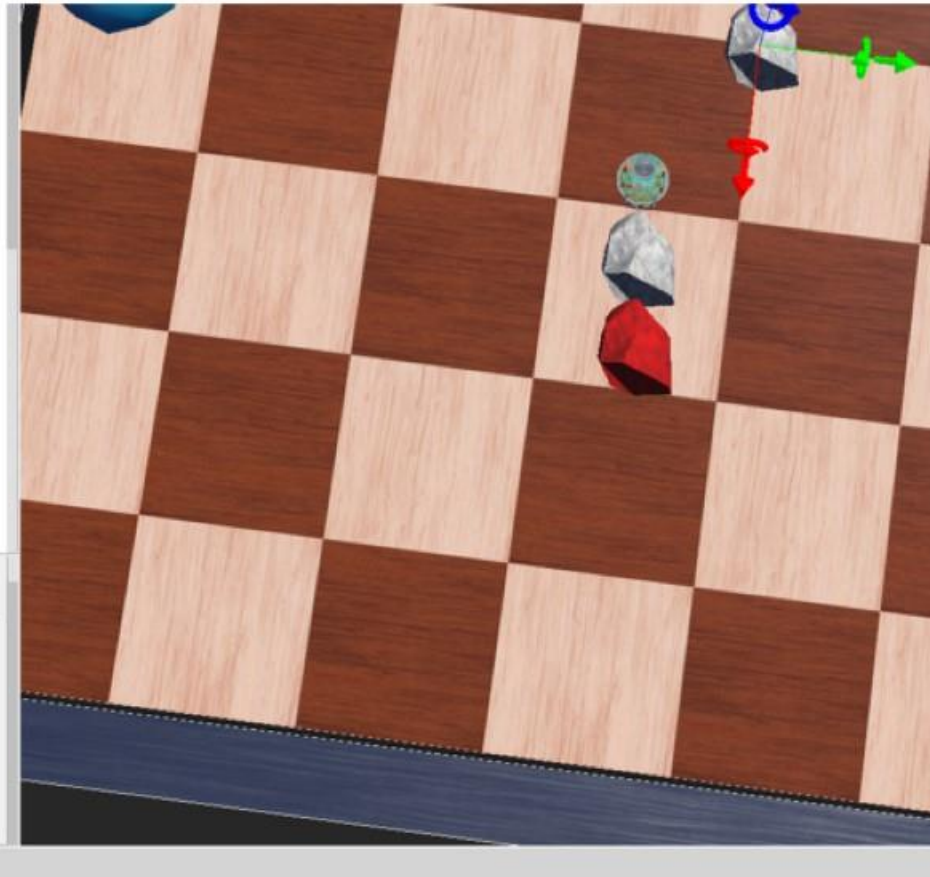
Console - All

Color of the central pixel: R=176, G=173, B=174

Captured an image of size: 52x39

Color of the central pixel: R=176, G=173, B=174

Captured an image of size: 52x39



The output value of RGB:

R=176 , G=173, B=174

Use any website to convert RGB values to Color:

https://www.w3schools.com/colors/colors_rgb.asp

RGB Calculator

`rgb(176, 173, 174)``#b0adae``hsl(340, 2%, 68%)`

R:

176

176

G:

173

173

B:

174

174

Now Remove the gray object and let Epuck robot camera capture the red object

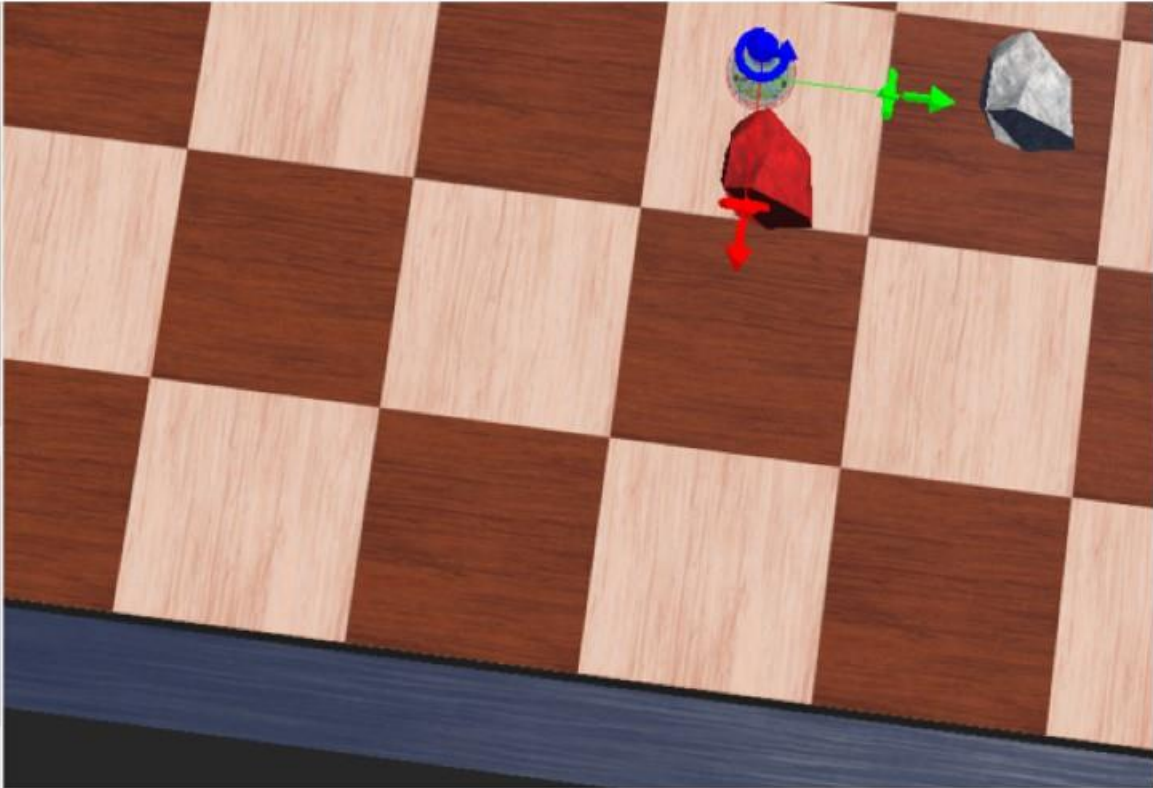
E-puck "e-puck"

- translation 0.342 -0.138 -4.46e-05
- rotation 0.0407 -0.938 -0.345 0.00363
- name "e-puck"
- controller "my_camera"
- controllerArgs
 - window "e-puck"
 - customData ""
 - supervisor FALSE
 - synchronization TRUE

Node	Position	Mass	Velocity
Absolute			
Position: 0.341646 -0.138202 -4.46378e-05			
Rotation: 0.040719 -0.937809 -0.344755 0.00362851			

Console - All

```
color of the central pixel: R=138, G=30, B=33  
captured an image of size: 52x39  
color of the central pixel: R=138, G=30, B=33
```



RGB Calculator

`rgb(138, 30, 33)``#8a1e21``hsl(358, 64%, 33%)`

R: 138

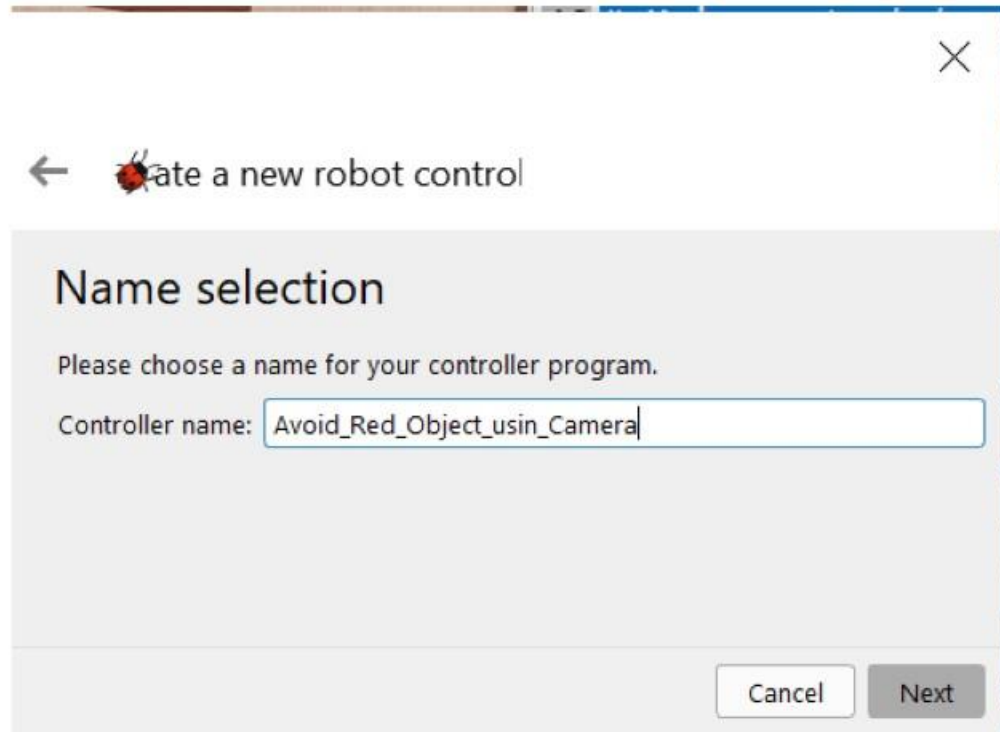
138

G: 30

30

B: 33

33



← Create a new robot control

Name selection

Please choose a name for your controller program.

Controller name:

Cancel Next

```
# Import the necessary modules from the Webots API
from controller import Robot, Camera, Motor
```

```
# Set the time step of the simulation
TIME_STEP = 64
```

```
# Create a robot instance to interact with the robot
robot = Robot()
```

```
# Get the camera device, enable it, and set the timestep
camera = robot.getDevice('camera')
camera.enable(TIME_STEP)
```

```
# Get the motor devices
left_motor = robot.getDevice('left wheel motor')
right_motor = robot.getDevice('right wheel motor')
```

```
# Set the target position of the motors to infinity for velocity
control
left_motor.setPosition(float('inf'))
right_motor.setPosition(float('inf'))
```

```
# Set the initial velocity of the motors to 0 (robot is stationary)
left_motor.setVelocity(0)
right_motor.setVelocity(0)
```

```
# Main loop: this is where the robot's behavior is defined
while robot.step(TIME_STEP) != -1:
    # Retrieve the image captured by the camera
    image = camera.getImage()

    # Get the width and height of the image from the camera's
    properties
    width = camera.getWidth()
    height = camera.getHeight()

    # Calculate the central pixel's coordinates
    centerX = width // 2
    centerY = height // 2

    # Get the red, green, and blue color components of the
    central pixel
    red = camera.imageGetRed(image, width, centerX,
    centerY)
    green = camera.imageGetGreen(image, width, centerX,
    centerY)
    blue = camera.imageGetBlue(image, width, centerX,
    centerY)
    print(f"Color of the central pixel: R={red}, G={green},
    B={blue}")
```

```
# Define what we mean by 'red' (This can be adjusted as
needed)
```

```
RED_THRESHOLD = 100
```

```
GREEN_THRESHOLD = 45
```

```
BLUE_THRESHOLD = 45
```

```
# Detect whether the central pixel is red enough
if red > RED_THRESHOLD and green < GREEN_THRESHOLD
and blue < BLUE_THRESHOLD:
```

```
    # If red enough, it means there's a red obstacle ahead, so
    turn right
```

```
        left_motor.setVelocity(2) # positive speed turns the wheel
        forward
```

```
        right_motor.setVelocity(-2) # negative speed turns the
        wheel backward
```

```
    else:
```

```
        # If not red enough, it means the path is clear, so move
        forward
```

```
        left_motor.setVelocity(5)
```

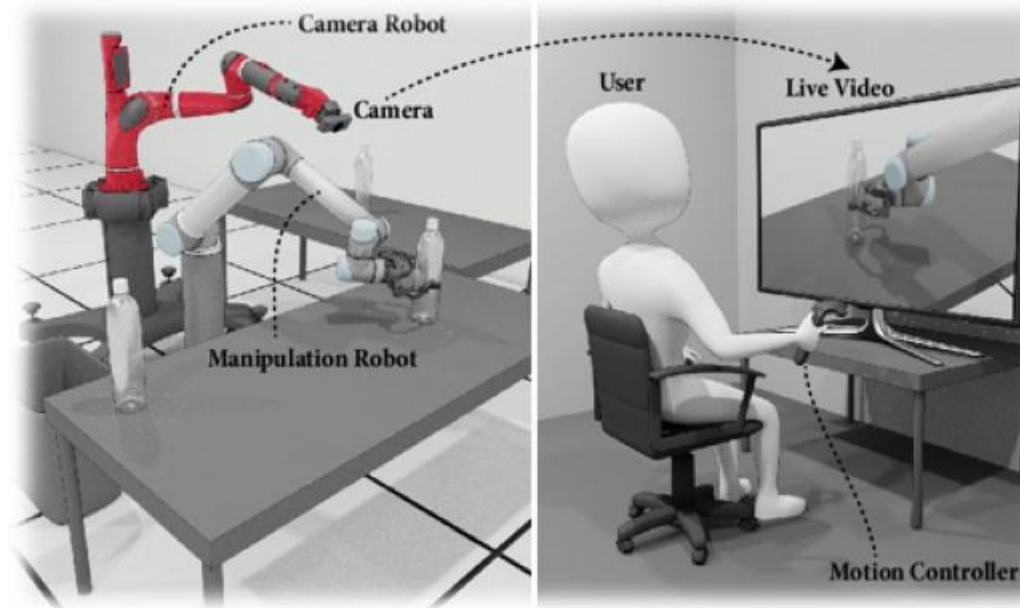
```
        right_motor.setVelocity(5)
```

The ROBOT will move forward If the RED obstacle (ROCK) the Robot Turn Right, If the Obstacle BLUE the robot will move Backward.

Human-Robot Interaction (HRI) system classification can be approached from several dimensions, reflecting the multifaceted nature of how humans and robots interact. The classification helps in understanding, designing, and evaluating HRI systems across various domains. Here's an overview of common classification dimensions in HRI:

Objective: Classifying HRI systems provides a framework for understanding the diverse ways in which robots and humans can interact.

Teleoperation: The human operator controls the robot from a distance, often using a remote control interface. Applications include drone flying, remote surgery, and exploration in hazardous environments.



Autonomous Robots: Robots operate independently based on pre-programmed behaviors or AI-driven decisions, with minimal human intervention. Examples include autonomous household cleaning robots and self-driving vehicles.

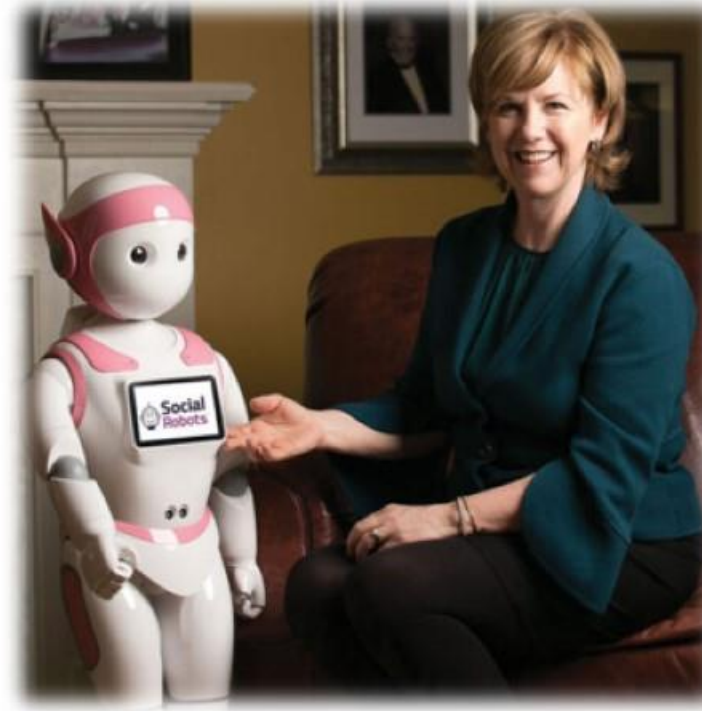


Collaborative Robots (Cobots): Robots working alongside humans in shared spaces, directly assisting human workers.

Common in manufacturing and assembly lines.



Social Robots: Robots designed to engage with humans on a social or emotional level, such as companion robots for the elderly or educational robots for children.



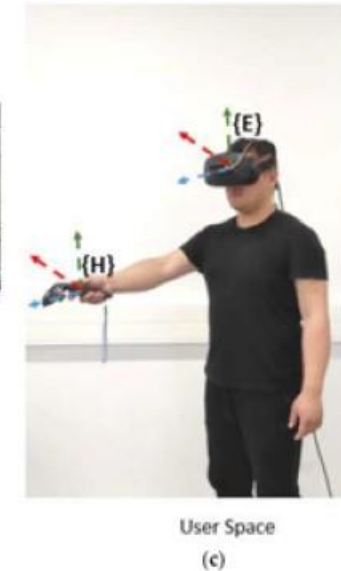
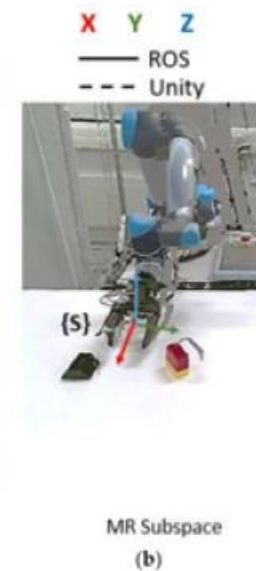
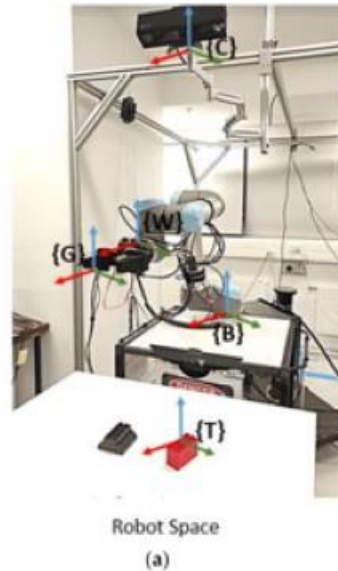
Utility Robots: Focus on functional tasks without the need for social or emotional interaction, like warehouse robots or agricultural drones.



- **Fully Autonomous:** Robots capable of performing tasks without human intervention, based on sensor data and pre-defined algorithms or learning.



- **Semi-Autonomous:** Robots that require occasional human guidance or decision-making to perform tasks.
- **Manual/Teleoperated:** Robots fully controlled by humans, without autonomous capabilities.



4. Physical vs. Non-Physical Interaction

Physical Interaction (pHRI): Involves direct physical contact between the human and the robot. This includes collaborative manipulation of objects, physical assistance, and touch-based communication.



Non-Physical Interaction: Involves interaction through visual, auditory, or other non-tactile means, such as command and control through voice or graphical interfaces.



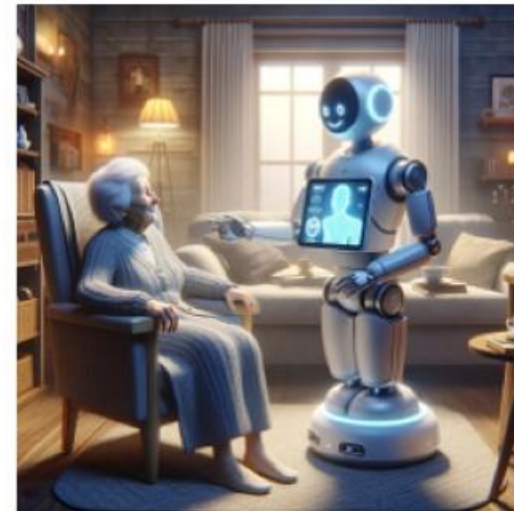
- **Industrial HRI:** Interaction focused on manufacturing, assembly, and other industrial applications. Efficiency and safety are key concerns.
- **Medical HRI:** Involves surgical robots, rehabilitation robots, and other healthcare applications. Precision and reliability are crucial.



- **Service HRI:** Encompasses service and assistance robots in retail, hospitality, and personal care. Usability and user experience are focal points.
- **Educational HRI:** Robots used as teaching aids or learning platforms. Engagement and adaptability to different learning styles are important.



- **Short-term Interaction:** Brief encounters between humans and robots, typical in public spaces or for specific tasks.
- **Long-term Interaction:** Prolonged engagement over time, common in companionship robots or when robots are integrated into daily routines.





Thank You