

Human-Robot Interaction and Autonomous Systems

AI 616

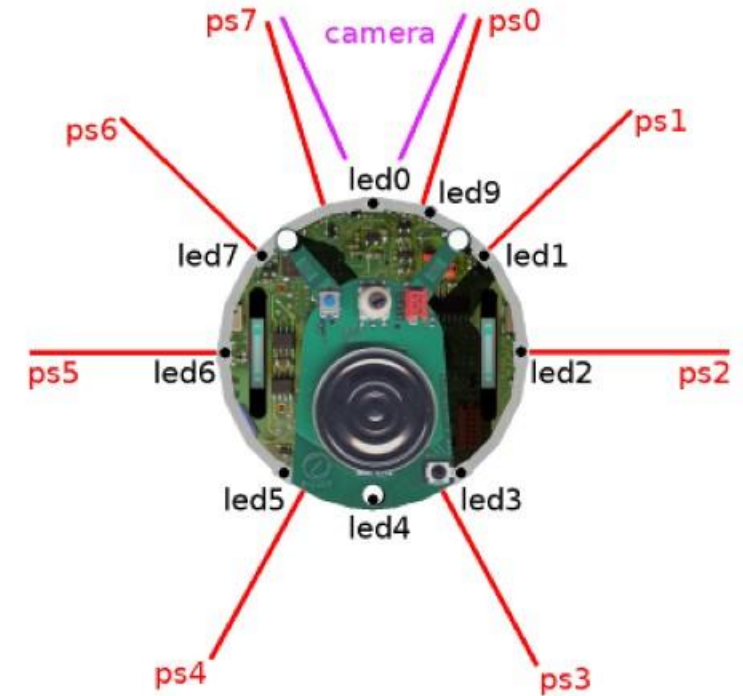
Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

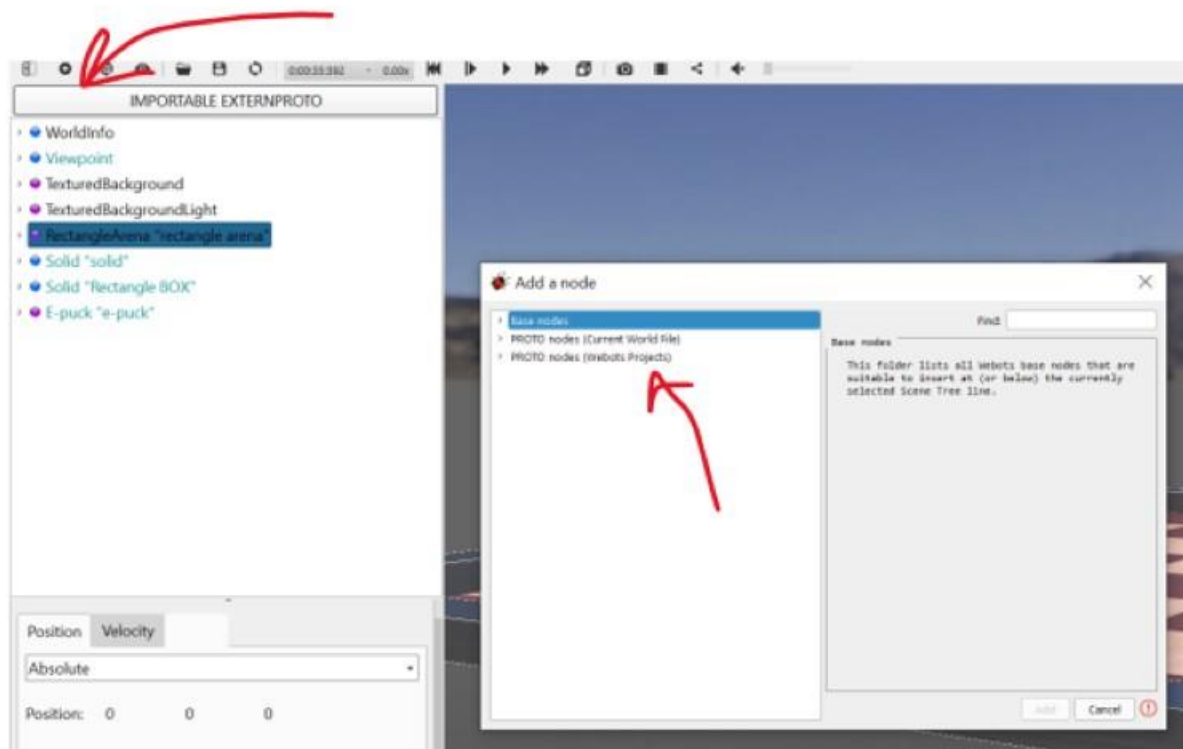
Lecture 8







ADD Obstacle (ROCK) in Webots



Add a node

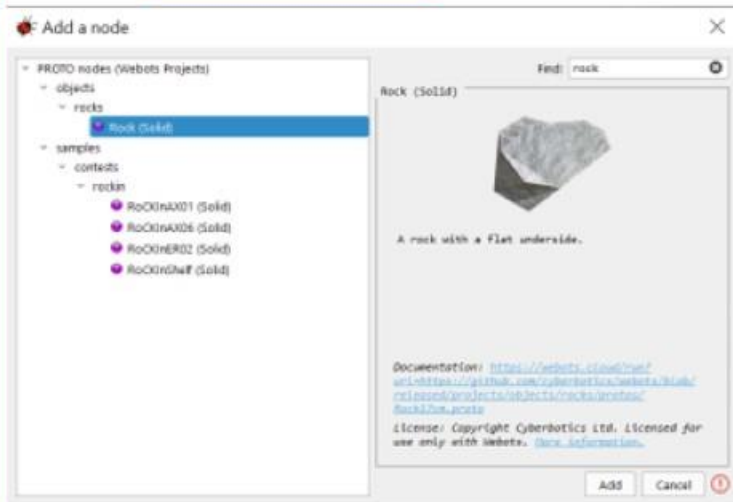
- Base nodes
- PROTO nodes (Current World File)
- PROTO nodes (Webots Projects)
 - bounding_objects
 - default
 - devices
 - humans
 - objects
 - advertising_board
 - animals
 - apartment_structure
 - backgrounds
 - balls
 - bathroom
 - bedroom
 - buildings
 - cabinet
 - chairs
 - computers
 - create_wall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



Add a node

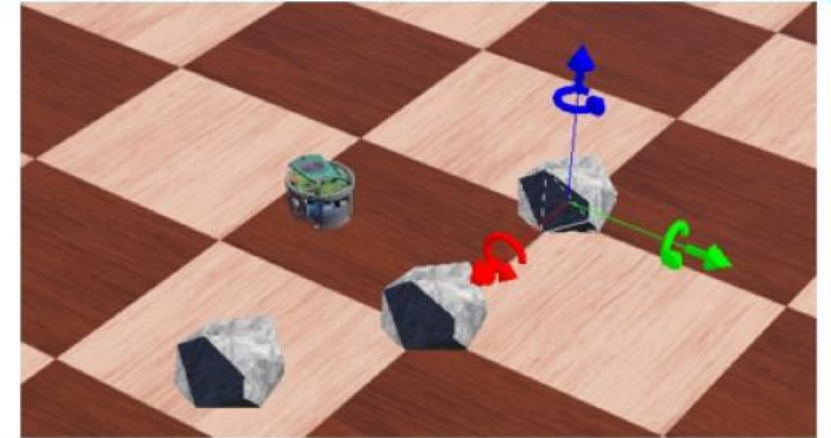
- garden
- gear
- hospital
- kitchen
- lego
- lights
- living_room_furniture
- mirror
- obstacles
 - OilBarrel (Solid)
 - Ramp30deg (Solid)
- paintings
- panels
- plants
- road
- robotstadium

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

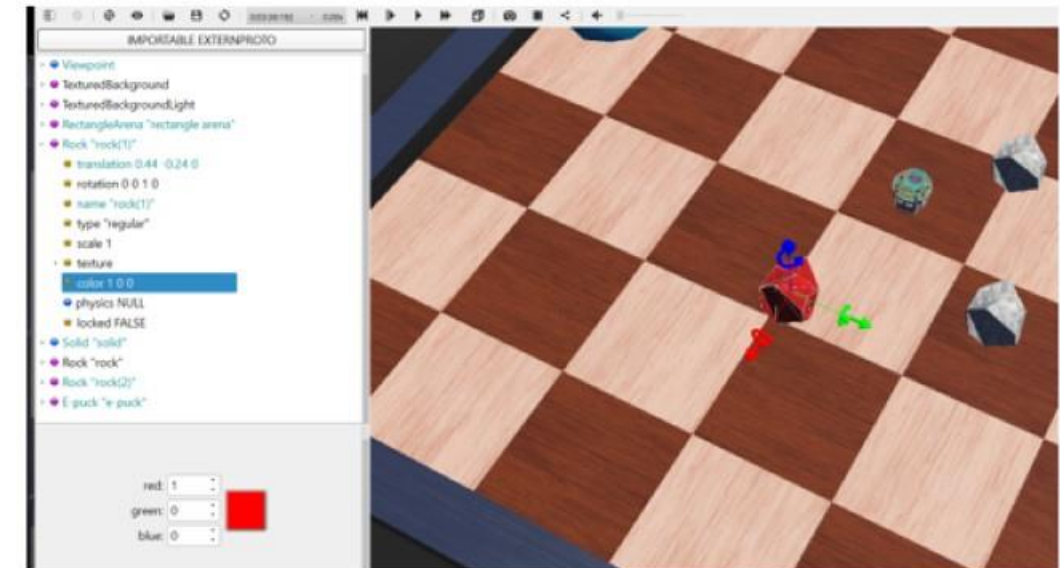


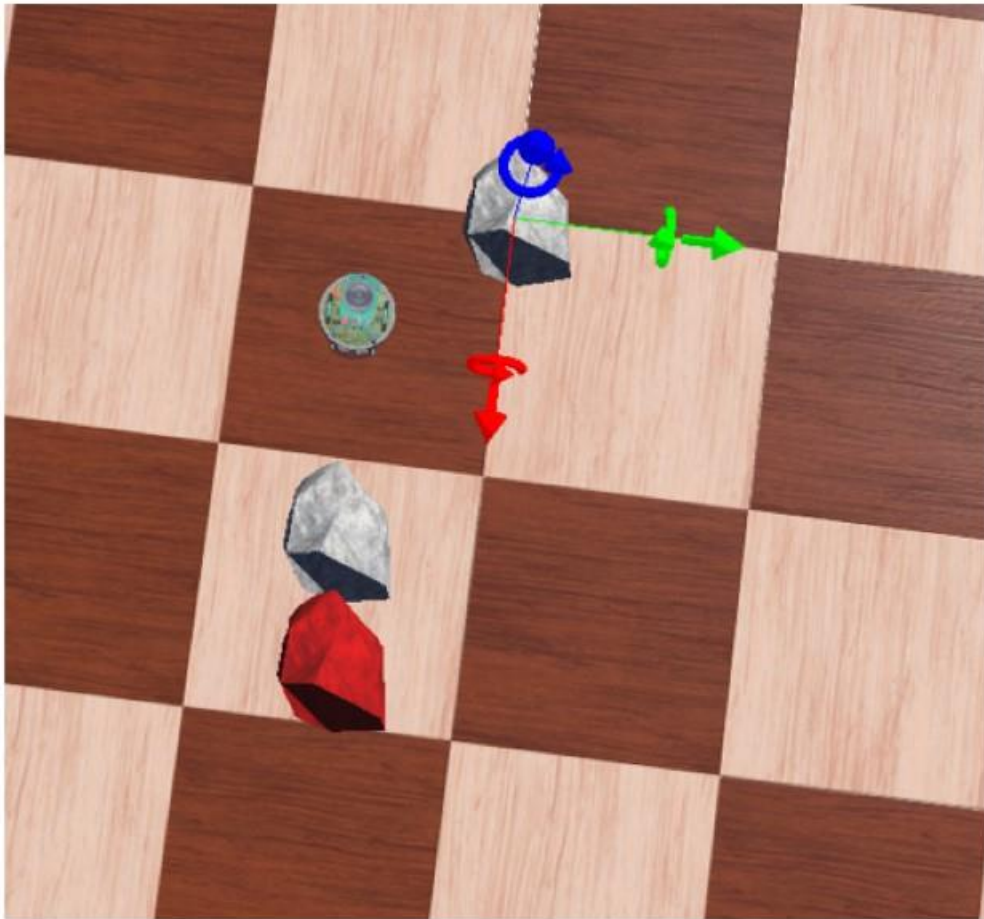
Click on the Arrows to move the object

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

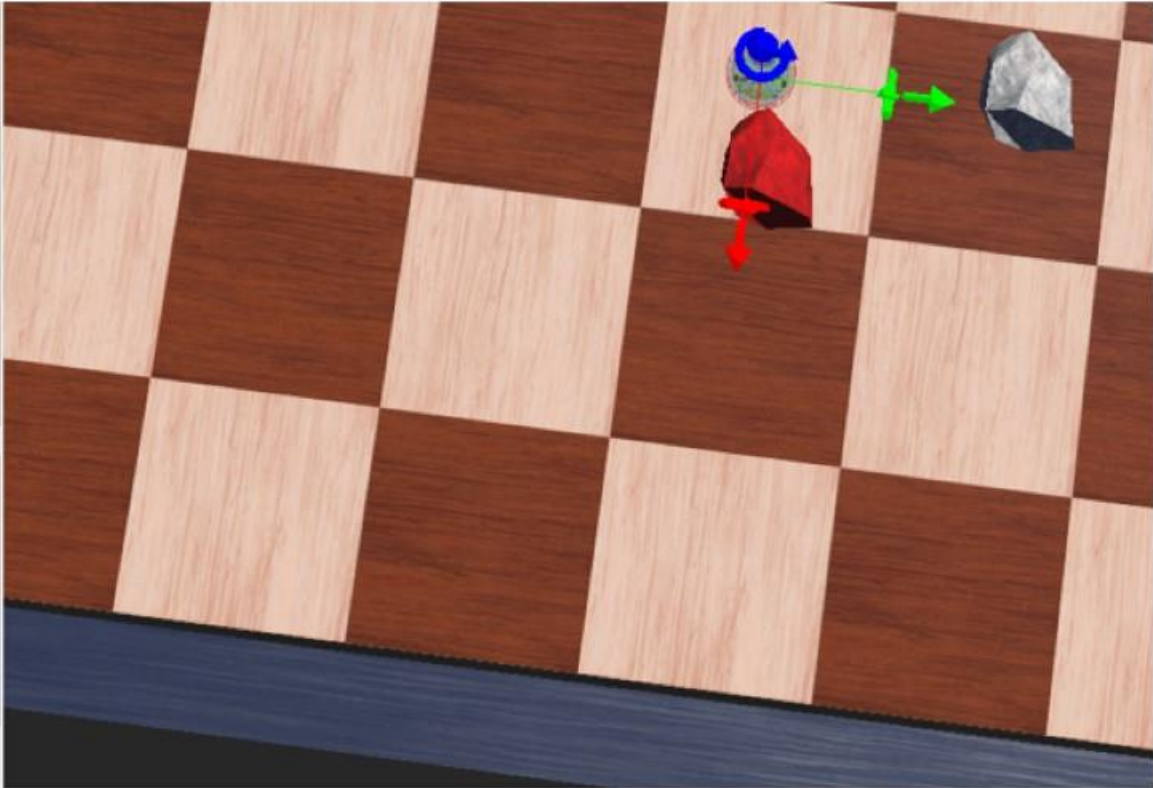
E-puck "e-puck"

- 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

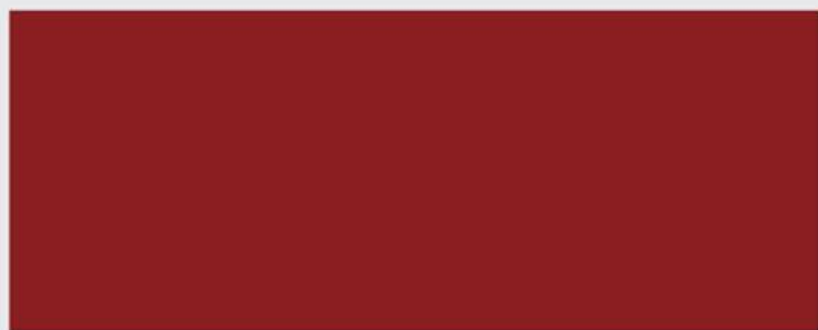
Node	Position	Mass	Velocity
Absolute			
Position: 0.341646 -0.138202 -4.46378e-05			
Rotation: 0.040719 -0.937809 -0.344755 0.00362851			

Console - All

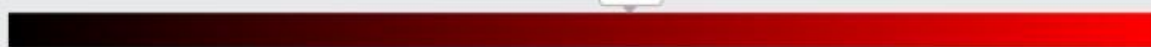
```
color of the central pixel: R=138, G=30, B=33  
captured an image of size: 52x39  
color of the central pixel: R=138, G=30, B=33
```



RGB Calculator

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

138

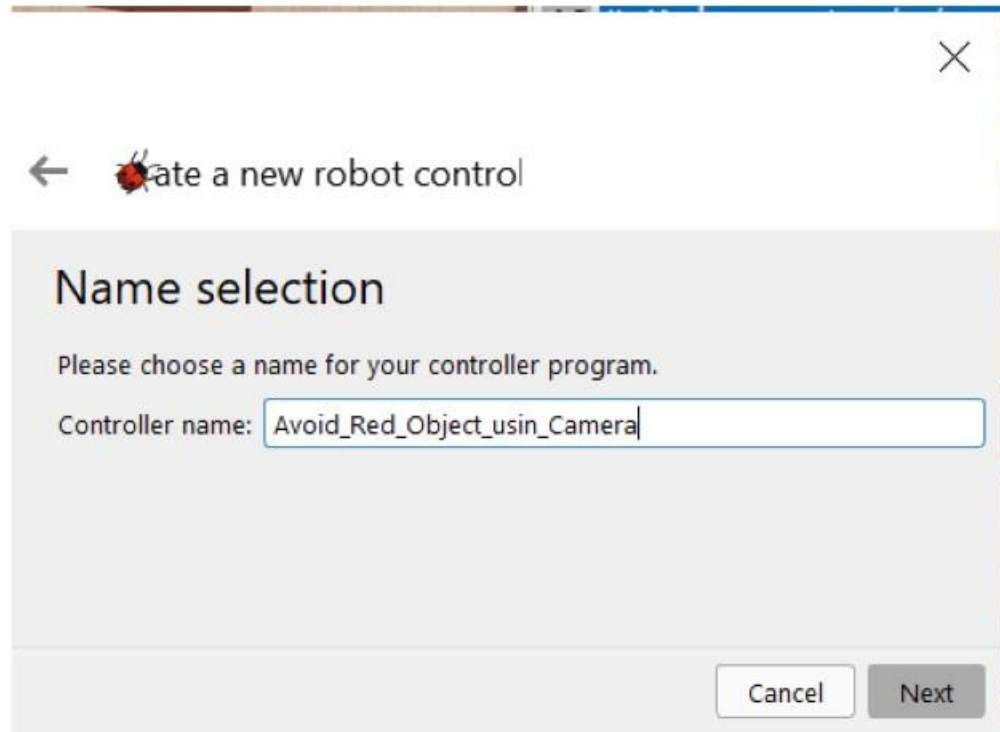
G:

30

B:

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

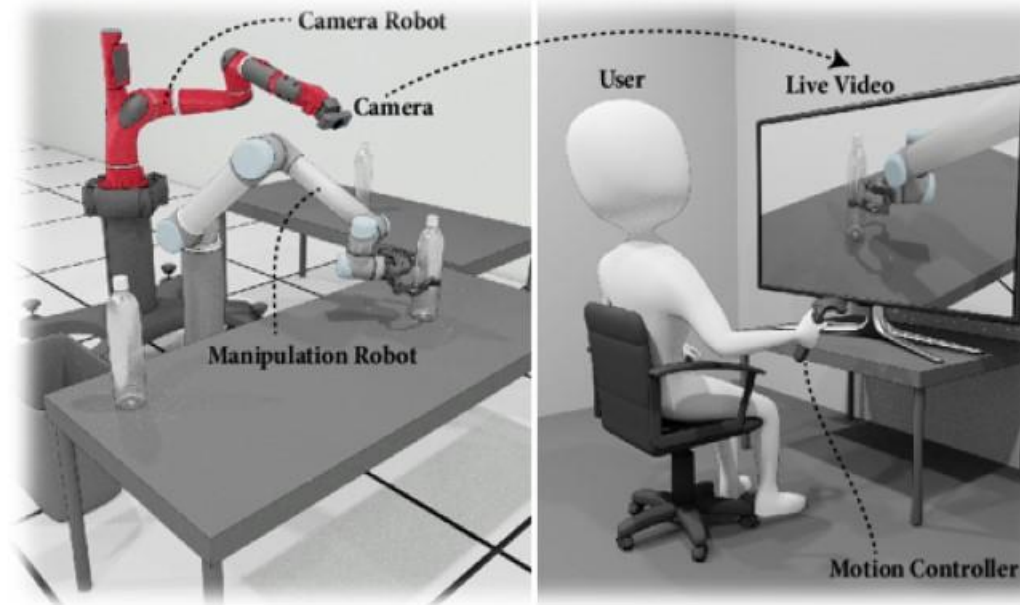
# Detect whether the central pixel is red enough
if red > RED_THRESHOLD and green < GREEN_THRESHOLD
and blue < BLUE_THRESHOLD:
    # If red enough, it means there's a red obstacle ahead, so
    turn right
    left_motor.setVelocity(2) # positive speed turns the wheel
    forward
    right_motor.setVelocity(-2) # 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.

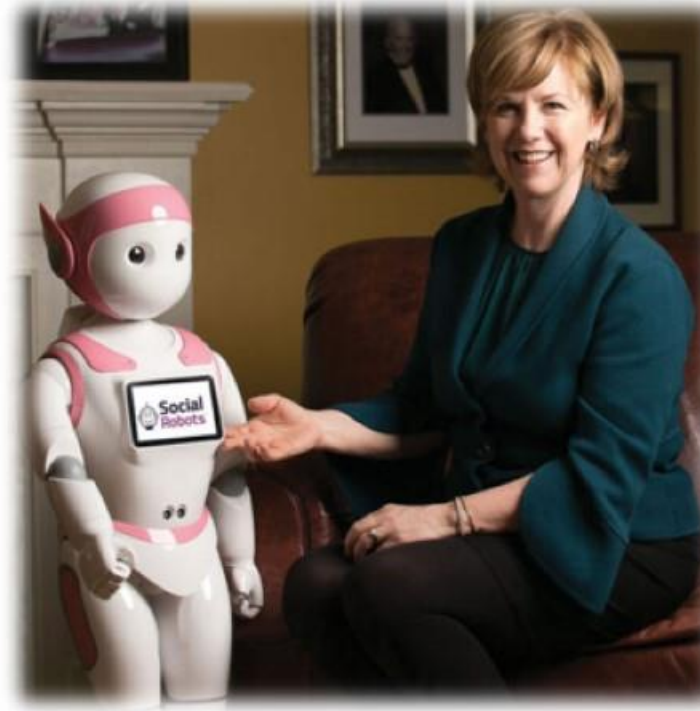


Collaborative Robots (Cobots): Robots working alongside humans in shared spaces, directly assisting human workers.

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.



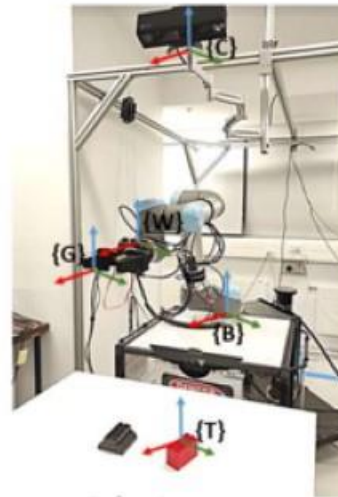
Utility Robots: Focus on functional tasks without the need for social or emotional interaction, like warehouse robots or agricultural drones.



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



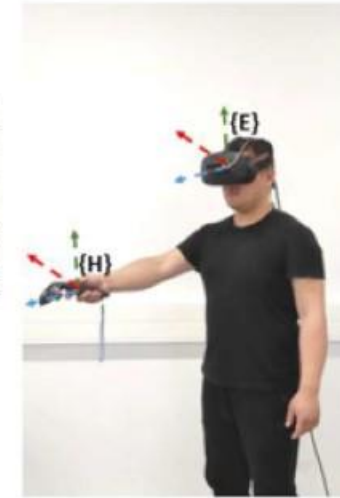
- **Semi-Autonomous:** Robots that require occasional human guidance or decision-making to perform tasks.
- **Manual/Teleoperated:** Robots fully controlled by humans, without autonomous capabilities.



Robot Space
(a)



MR Subspace
(b)



User Space
(c)

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.



5. Purpose or Application Area

- **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.

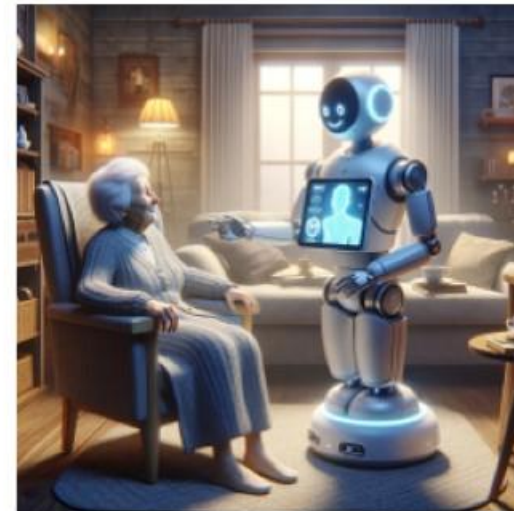


- **Service HRI:** Encompasses service and assistance robots in retail, hospitality, and personal care. Usability and user experience are focal points.
- **Educational HRI:** Robots used as teaching aids or learning platforms. Engagement and adaptability to different learning styles are important.



6. Duration and Nature of Interaction

- **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.



EXAMPLES



Webots

robot simulation

Analyze a 3x3 matrix around the central pixel and use the average color values for decision-making.

P1	P2	P3
P4	P5	P6
P7	P8	P9

-1	0	+1
-1	0	+1
-1	0	+1

We'll calculate the average red, green, and blue values of the nine pixels at the center of the image and use these averages to determine the robot's movement.

```
while robot.step(TIME_STEP) != -1:
    image = camera.getImage()
    width = camera.getWidth()
    height = camera.getHeight()

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

    # Initialize sums for the color components
    sum_red = 0
    sum_green = 0
    sum_blue = 0
```

```
# Loop over a 3x3 matrix around the central pixel
for dx in range(-1, 2): # -1, 0, 1
    for dy in range(-1, 2): # -1, 0, 1
        # Calculate the pixel's coordinates
        x = centerX + dx
        y = centerY + dy

        # Sum the color components of the pixel
        sum_red += camera.imageGetRed(image, width, x, y)
        sum_green += camera.imageGetGreen(image, width, x, y)
        sum_blue += camera.imageGetBlue(image, width, x, y)
```

-1	0	+1
-1	0	+1
-1	0	+1

```
# Calculate the average color components
avg_red = sum_red // 9
avg_green = sum_green // 9
avg_blue = sum_blue // 9
```

```
# Adjust thresholds and decisions based on the average color
```

```
RED_THRESHOLD = 100
```

```
GREEN_THRESHOLD = 45
```

```
BLUE_THRESHOLD = 45
```

```
RED_THRESHOLD2 = 45
```

```
GREEN_THRESHOLD2 = 45
```

```
BLUE_THRESHOLD2 = 100
```

```
if avg_red > RED_THRESHOLD and avg_green < GREEN_THRESHOLD and avg_blue < BLUE_THRESHOLD:
```

```
    left_motor.setVelocity(5) # Turn right
```

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

```
elif avg_blue > BLUE_THRESHOLD2 and avg_green < GREEN_THRESHOLD2 and avg_red < RED_THRESHOLD2:
```

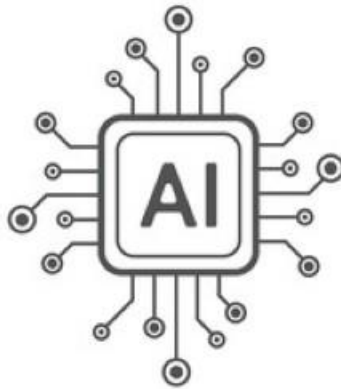
```
    left_motor.setVelocity(-5) # Move backward
```

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

```
else:
```

```
    left_motor.setVelocity(5) # Move forward
```

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



Webots

robot simulation

Control the robot's speed based on color detection—moving faster if the color is green and very slowly if the color is red.

We are going to use Machine Learning Algorithm.
We are going to use Linear regression.

Let's assume:

- A high "color score" represents green, encouraging faster movement.
- A low "color score" represents red, dictating slower movement.

For simplicity, let's say our model has determined the relationship between color score and speed as $y=0.2x-1$,
where y is the robot's speed,
and x is the color score (ranging from 0 for red to 5 for green).

- Color Score: This is a conceptual representation of how "green" or "red" the camera's view is. (by averaging the RGB values).
- Linear Regression Model: The model $y = 0.2x - 1$ is a simple formula we're using to decide the robot's speed based on the detected color. The specific coefficients (0.2 and -1) are placeholders for this example and would be determined through actual model training in a real-world application.
- Speed Adjustment: The speed is adjusted based on the color score, with a higher score (more green) resulting in faster speed and a lower score (more red) resulting in slower speed or stopping.

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

```
# Initialization
```

```
robot = Robot()
```

```
timestep = int(robot.getBasicTimeStep())
```

```
# Assuming motor and camera initialization here
```

```
camera = robot.getDevice('camera')
```

```
camera.enable(timestep)
```

```
left_motor = robot.getDevice('left wheel motor')
```

```
right_motor = robot.getDevice('right wheel motor')
```

```
left_motor.setPosition(float('inf')) # Enable speed control
```

```
right_motor.setPosition(float('inf'))
```

```
left_motor.setVelocity(0) # Initially stop
```

```
right_motor.setVelocity(0)
```

```
while robot.step(timestep) != -1:
    # Simulate getting a color score from the camera
    # Let's say color score calculation logic has been applied here
    # For simplification, assume we get this score directly from the camera
    # A real implementation would involve analyzing the camera image
    color_score = 3 # Placeholder for the actual color score calculation (0 = red, 5 = green)

    # Apply the linear regression model
    # For our example, the model equation is  $y = 0.2x - 1$ 
    predicted_speed = 0.2 * color_score - 1 # Simplified model to control speed based on color
score

# Ensure the speed is within the robot's operational range
predicted_speed = max(-10, min(10, predicted_speed))

# Set the robot's motors to the predicted speed
left_motor.setVelocity(predicted_speed)
right_motor.setVelocity(predicted_speed)
```

A simple machine learning (ML) algorithm into the previous Webots robot navigation example introduces beginners to the intersection of robotics and ML. A straightforward approach is to use a decision tree classifier to decide the robot's movement based on the average color detected by the camera in a 3x3 pixel area. This example assumes you have basic data to train the classifier, representing average color values for different actions (move forward, turn right, move backward).

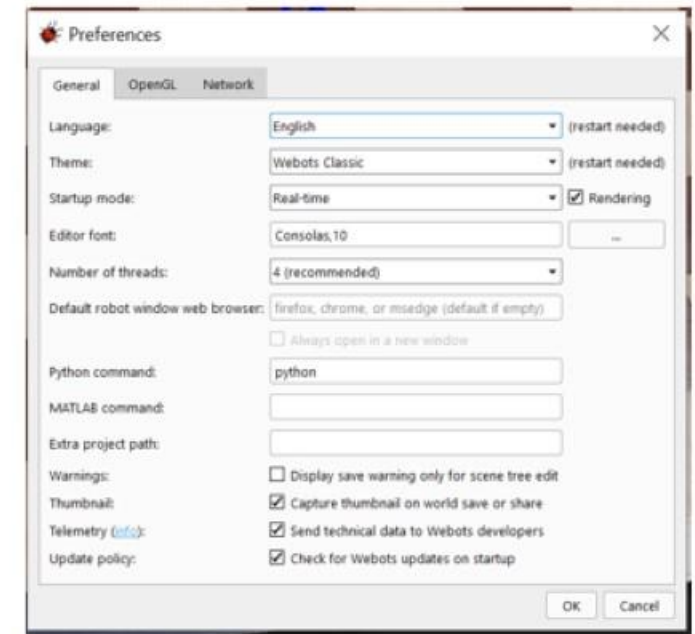
This example will be conceptual, as implementing a full ML pipeline involves steps (like data collection and model training) that go beyond the scope of running code directly in the Webots environment.

We are going to use a simple **decision tree classifier using scikit-learn**, a popular ML library for Python.

To install scikit-learn in Webots, or any Python environment within a simulation or application, you generally need to ensure that Python is correctly set up and that you have the necessary permissions to install additional libraries. Here's a step-by-step guide to installing scikit-learn for use with Webots:

Step 1: Identify Your Python Environment

- First, determine which Python environment Webots is using. Webots can use a global Python installation or a virtual environment. You can specify or check this in the Webots preferences:
- Open Webots.
- Go to Tools > Preferences (on macOS, Webots > Preferences).
- In the General tab, check the Python command setting to see which Python is being used.



Step 2: Install scikit-learn

Using a Global Python Installation

If Webots is configured to use your global Python installation, you can install scikit-learn using pip in your command line or terminal. Make sure you have pip installed and then run:

pip install scikit-learn

Using a Virtual Environment

If Webots is using a Python virtual environment:

Activate the Virtual Environment: First, you need to activate the virtual environment. Navigate to the virtual environment directory and activate it using the command line or terminal:

On Windows: `.\venv\Scripts\activate`

Error : `.\venv\Scripts\activate`

The system cannot find the path specified.

Solution:

1- Navigate to the Correct Directory

`cd path\to\my_project`

```
cd C:\Users\User\Documents\my_project2\controllers
```

2- Verify the Path to the Virtual Environment

`dir`

If you don't see the **venv** folder, or if the **Scripts** directory is missing inside **venv**, the virtual environment was not created correctly.

```
C:\Users\User\Documents\my_project2\controllers>dir
Volume in drive C has no label.
Volume Serial Number is CAE4-EDA3

Directory of C:\Users\User\Documents\my_project2\controllers

2024-03-19  01:47 AM    <DIR>          .
2024-03-19  01:47 AM    <DIR>          ..
2024-03-13  10:49 AM    <DIR>          Avid_Color_Obstacle_Using_Camera
2024-02-27  12:38 PM    <DIR>          Avoid_1
2024-03-12  12:17 PM    <DIR>          Avoid_Red_Object_usin_Camera
2024-03-19  01:47 AM    <DIR>          color_Based_Navigation_Using_Camera3X3
2024-02-20  11:24 AM    <DIR>          epuck_go_forward
2024-03-05  04:28 PM    <DIR>          multi_infrared_sensors_Avoid_Obstacle
2024-03-03  10:39 AM    <DIR>          my_camera
2024-03-05  12:32 PM    <DIR>          Obstacle_with_flash
               0 File(s)              0 bytes
              10 Dir(s)  39,366,397,952 bytes free
```

3- Recreate the Virtual Environment (If Necessary)

If the virtual environment doesn't exist or was not created properly, you'll need to recreate it. Make sure you have Python installed, and then run:

```
python -m venv venv
```

This command creates a new virtual environment named venv in the current directory. After creation, try activating it again:

```
.\venv\Scripts\activate
```

```
C:\Users\User\Documents\my_project2\controllers>.\venv\Scripts\activate
```

```
(venv) C:\Users\User\Documents\my_project2\controllers>cd path\to\my_project
The system cannot find the path specified.
```

```
C:\Users\User\Documents\my_project2\controllers>python -m venv venv

C:\Users\User\Documents\my_project2\controllers>
C:\Users\User\Documents\my_project2\controllers>dir
Volume in drive C has no label.
Volume Serial Number is CAE4-EDA3

Directory of C:\Users\User\Documents\my_project2\controllers

2024-03-19  01:20 PM    <DIR>          .
2024-03-19  01:20 PM    <DIR>          ..
2024-03-13  10:49 AM    <DIR>          Avid_Color_Obstacle_Using_Camera
2024-02-27  12:38 PM    <DIR>          Avoid_1
2024-03-12  12:17 PM    <DIR>          Avoid_Red_Object_usin_Camera
2024-03-19  01:47 AM    <DIR>          color_Based_Navigation_Using_Camera3X3
2024-02-20  11:24 AM    <DIR>          epuck_go_forward
2024-03-05  04:28 PM    <DIR>          multi_infrared_sensors_Avoid_Obstacle
2024-03-03  10:39 AM    <DIR>          my_camera
2024-03-05  12:32 PM    <DIR>          Obstacle_with_flash
2024-03-19  01:20 PM    <DIR>          venv
                                0 File(s)                0 bytes
```

4- Install scikit-learn

Once your virtual environment is activated, you can install scikit-learn as previously mentioned:

pip install scikit-learn

```
(venv) C:\Users\User\Documents\my_project2\controllers>pip install scikit-learn
Collecting scikit-learn
  Obtaining dependency information for scikit-learn from https://files.pythonhosted.org/packages/76/d8/02fa68
learn-1.4.1.post1-cp312-cp312-win_amd64.whl.metadata
  Downloading scikit_learn-1.4.1.post1-cp312-cp312-win_amd64.whl.metadata (11 kB)
Collecting numpy<2.0,>=1.19.5 (from scikit-learn)
  Obtaining dependency information for numpy<2.0,>=1.19.5 from https://files.pythonhosted.org/packages/16/2e
mpy-1.26.4-cp312-cp312-win_amd64.whl.metadata
  Downloading numpy-1.26.4-cp312-cp312-win_amd64.whl.metadata (61 kB)
----- 61.0/61.0 kB 540.4 kB/s eta 0:00:00
Collecting scipy>=1.6.0 (from scikit-learn)
  Obtaining dependency information for scipy>=1.6.0 from https://files.pythonhosted.org/packages/f3/31/91a2a
12.0-cp312-cp312-win_amd64.whl.metadata
  Downloading scipy-1.12.0-cp312-cp312-win_amd64.whl.metadata (60 kB)
----- 60.4/60.4 kB 1.6 MB/s eta 0:00:00
Collecting joblib>=1.2.0 (from scikit-learn)
  Obtaining dependency information for joblib>=1.2.0 from https://files.pythonhosted.org/packages/10/40/d551
1.3.2-py3-none-any.whl.metadata
  Downloading joblib-1.3.2-py3-none-any.whl.metadata (5.4 kB)
Collecting threadpoolctl>=2.0.0 (from scikit-learn)
  Obtaining dependency information for threadpoolctl>=2.0.0 from https://files.pythonhosted.org/packages/b1/
threadpoolctl-3.3.0-py3-none-any.whl.metadata
  Downloading threadpoolctl-3.3.0-py3-none-any.whl.metadata (13 kB)
  Downloading scikit_learn-1.4.1.post1-cp312-cp312-win_amd64.whl (10.6 MB)
----- 10.6/10.6 MB 7.4 MB/s eta 0:00:00
Using cached joblib-1.3.2-py3-none-any.whl (302 kB)
  Downloading numpy-1.26.4-cp312-cp312-win_amd64.whl (15.5 MB)
----- 15.5/15.5 MB 7.4 MB/s eta 0:00:00
  Downloading scipy-1.12.0-cp312-cp312-win_amd64.whl (45.8 MB)
----- 45.8/45.8 MB 6.7 MB/s eta 0:00:00
  Downloading threadpoolctl-3.3.0-py3-none-any.whl (17 kB)
Installing collected packages: threadpoolctl, numpy, joblib, scipy, scikit-learn
Successfully installed joblib-1.3.2 numpy-1.26.4 scikit-learn-1.4.1.post1 scipy-1.12.0 threadpoolctl-3.3.0

[notice] A new release of pip is available: 23.2.1 -> 24.0
[notice] To update, run: python.exe -m pip install --upgrade pip
(venv) C:\Users\User\Documents\my_project2\controllers>
```

5- Verify Installation

To verify that scikit-learn has been successfully installed, you can run a small Python script to import scikit-learn and print its version:

First: Type python

Then use the following command

```
import sklearn  
print(sklearn.__version__)
```

```
(venv) C:\Users\User\Documents\my_project2\controllers>python  
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import sklearn  
>>> print(sklearn.__version__)  
1.4.1.post1  
>>> _
```

Exit the Python Interactive Shell

6- Use scikit-learn in Your Webots Project

Able to use scikit-learn in your Webots projects. You can import scikit-learn in your Python controller scripts and use its functionalities as needed.

For simplicity, let's simulate training data. In a real scenario, you'd collect data based on the robot's observations and corresponding actions.

```
# Example training data: [avg_red, avg_green, avg_blue, action]
# Actions: 0 - Move Forward, 1 - Turn Right, 2 - Move Backward
training_data = [
    [150, 30, 30, 1], # Red-like colors prompt turning right.
    [30, 30, 150, 2], # Blue-like colors prompt moving backward.
    [80, 80, 80, 0], # Neutral colors prompt moving forward.
    # Add more samples as needed
]
```

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

# Prepare training data
X = np.array([x[:-1] for x in training_data]) # Color components
y = np.array([x[-1] for x in training_data]) # Actions

# Train the classifier
clf = DecisionTreeClassifier()
clf.fit(X, y)
```

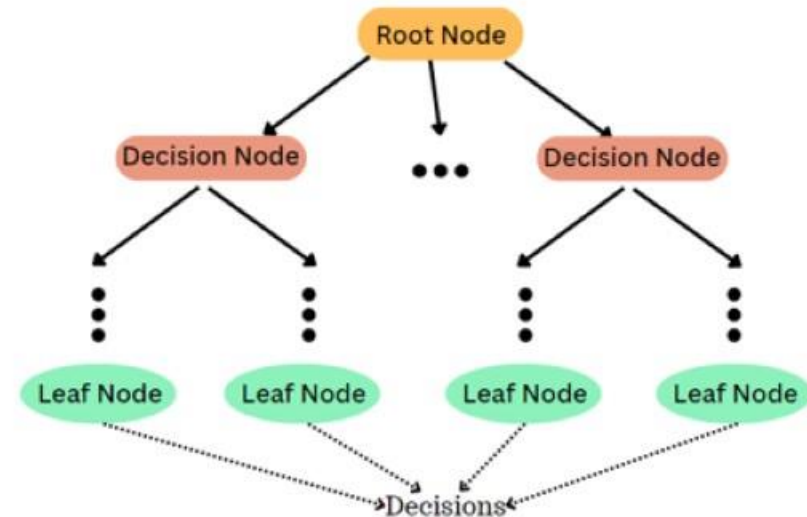
```
# Main simulation loop with ML decision-making
while robot.step(TIME_STEP) != -1:
    # Image processing to get the average color...
    avg_red, avg_green, avg_blue = get_average_color(camera)

    # Predict the action using the trained classifier
    action = clf.predict([[avg_red, avg_green, avg_blue]])[0]

    # Define robot movements based on the predicted action
    if action == 0: # Move Forward
        set_velocity(left_motor, right_motor, 5, 5)
    elif action == 1: # Turn Right
        set_velocity(left_motor, right_motor, 5, -5)
    elif action == 2: # Move Backward
        set_velocity(left_motor, right_motor, -5, -5)
```

Understanding Decision Trees

A **decision tree** is a flowchart-like tree structure where an **internal node represents a feature(or attribute)**, the **branch represents a decision rule**, and **each leaf node represents the outcome**. In the context of our robot, the features are the average red, green, and blue (RGB) color values of a 3x3 pixel area in the camera's field of view, and the outcome is the action the robot should take (move forward, turn right, move backward).



This example assumes that we have a pretrained decision tree model. The training of the model is conceptually described, but in practice, you would need to collect data, label it, and train the model using a tool like scikit-learn in a Python environment outside of Webots.

First, you would collect data and train your model. This step is conceptual and would typically happen outside of your Webots project using scikit-learn or a similar library:

```
# Simulate training data collection
# Features: Average R, G, B values; Label: Action (0=Forward, 1=Right, 2=Backward)
training_data = [
    [120, 40, 40, 1], # Red objects mean turn right
    [40, 40, 120, 2], # Blue objects mean move backward
    [80, 110, 90, 0], # Neutral colors mean move forward
]

# Normally, here you would split your data into features (X) and labels (y),
# train a decision tree model using scikit-learn, and then save the model.
# For this example, assume we have a trained model ready to use.
```

Now, integrating the model into your Webots controller script. This part simulates loading a pretrained model and using it within the main robot control loop:

```
from controller import Robot, Camera, Motor
import numpy as np
```

```
# Define the robot's actions based on prediction
def robot_action(action, left_motor, right_motor):
    speed = 5
    if action == 0: # Move forward
        left_motor.setVelocity(speed)
        right_motor.setVelocity(speed)
    elif action == 1: # Turn right
        left_motor.setVelocity(speed)
        right_motor.setVelocity(-speed)
    elif action == 2: # Move backward
        left_motor.setVelocity(-speed)
        right_motor.setVelocity(-speed)
```

```
# Main robot control code
```

```
TIME_STEP = 64
```

```
robot = Robot()
```

```
# Initialize camera
```

```
camera = robot.getDevice('camera')
```

```
camera.enable(TIME_STEP)
```

```
# Initialize motors
```

```
left_motor = robot.getDevice('left wheel motor')
```

```
right_motor = robot.getDevice('right wheel motor')
```

```
left_motor.setPosition(float('inf'))
```

```
right_motor.setPosition(float('inf'))
```

```
left_motor.setVelocity(0)
```

```
right_motor.setVelocity(0)
```

```
# Assume 'model' is our pretrained decision tree classifier
```

```
# For educational purposes, we will simulate its predictions  
based on color thresholds
```

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

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

```
    width = camera.getWidth()
```

```
    height = camera.getHeight()
```

```
# Center coordinates of the image
```

```
centerX = width // 2
```

```
centerY = height // 2
```

```
# Calculate the average color of the central 3x3 pixels
```

```
avg_color = np.zeros(3)
```

```
for dx in range(-1, 2):
```

```
    for dy in range(-1, 2):
```

```
        # Correcting the method to get RGB components for each pixel
```

```
        red = camera.imageGetRed(image, width, centerX + dx, centerY + dy)
```

```
        green = camera.imageGetGreen(image, width, centerX + dx, centerY + dy)
```

```
        blue = camera.imageGetBlue(image, width, centerX + dx, centerY + dy)
```

```
        avg_color += np.array([red, green, blue])
```

```
avg_color /= 9 # Average RGB values
```

```
# Determine action based on average color
# Red detection
if avg_color[0] > 100 and avg_color[1] < 50 and avg_color[2] < 50:
    action = 1 # Turn right due to red
# Green detection
elif avg_color[1] > 100 and avg_color[0] < 50 and avg_color[2] < 50:
    action = 0 # Move forward due to green
else:
    action = 2 # Move backward for any other color as default

# Execute the action
robot_action(action, left_motor, right_motor)
```



Thank You