

Human-Robot Interaction and Autonomous Systems

AI 616

Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

Lecture 5





Course Outlines

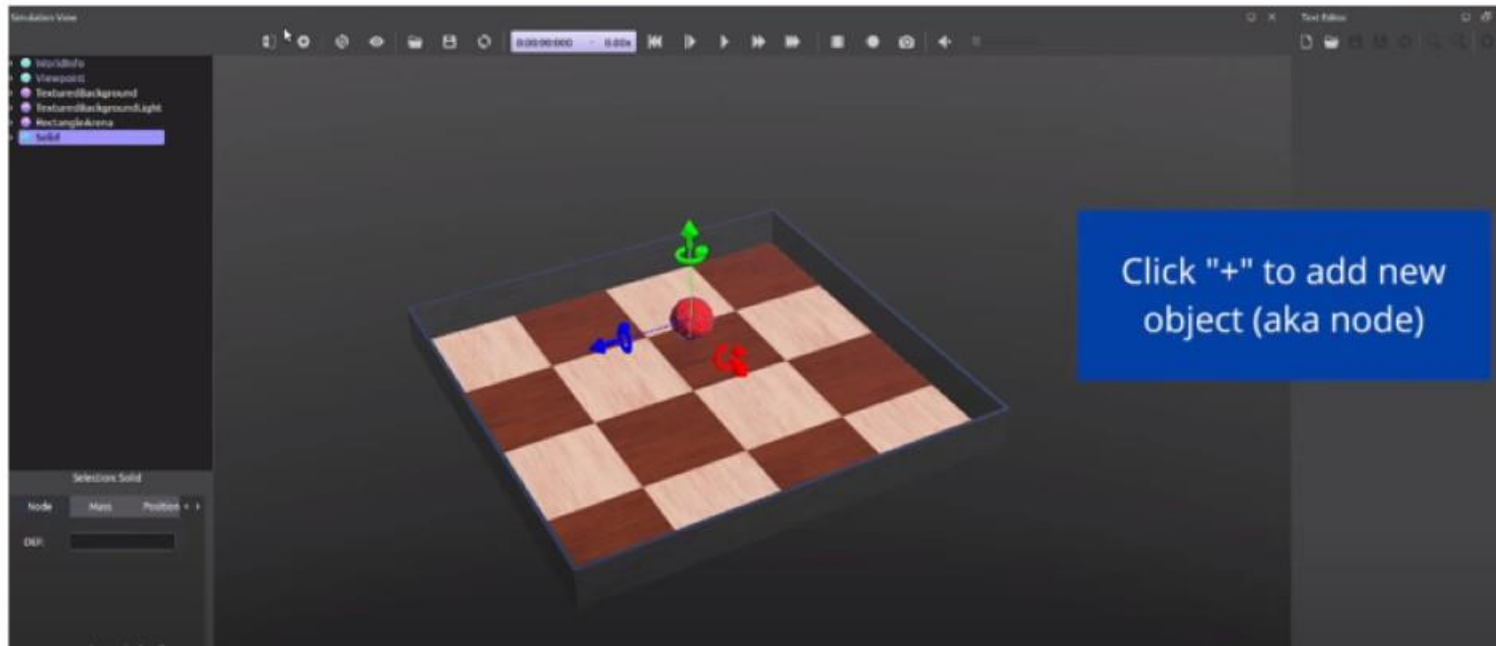
Module	Content	Simulation Lab
1. Introduction to Robotics	- Fundamentals of Robot Control - Classification of Robotics - Exploration of Robot Types	- Basic Robot Control - Modeling Robot Classes - Simulating Various Robot Types
2. Advanced HRI Studies	- Introductory to Advanced Concepts in HRI	- Basic 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



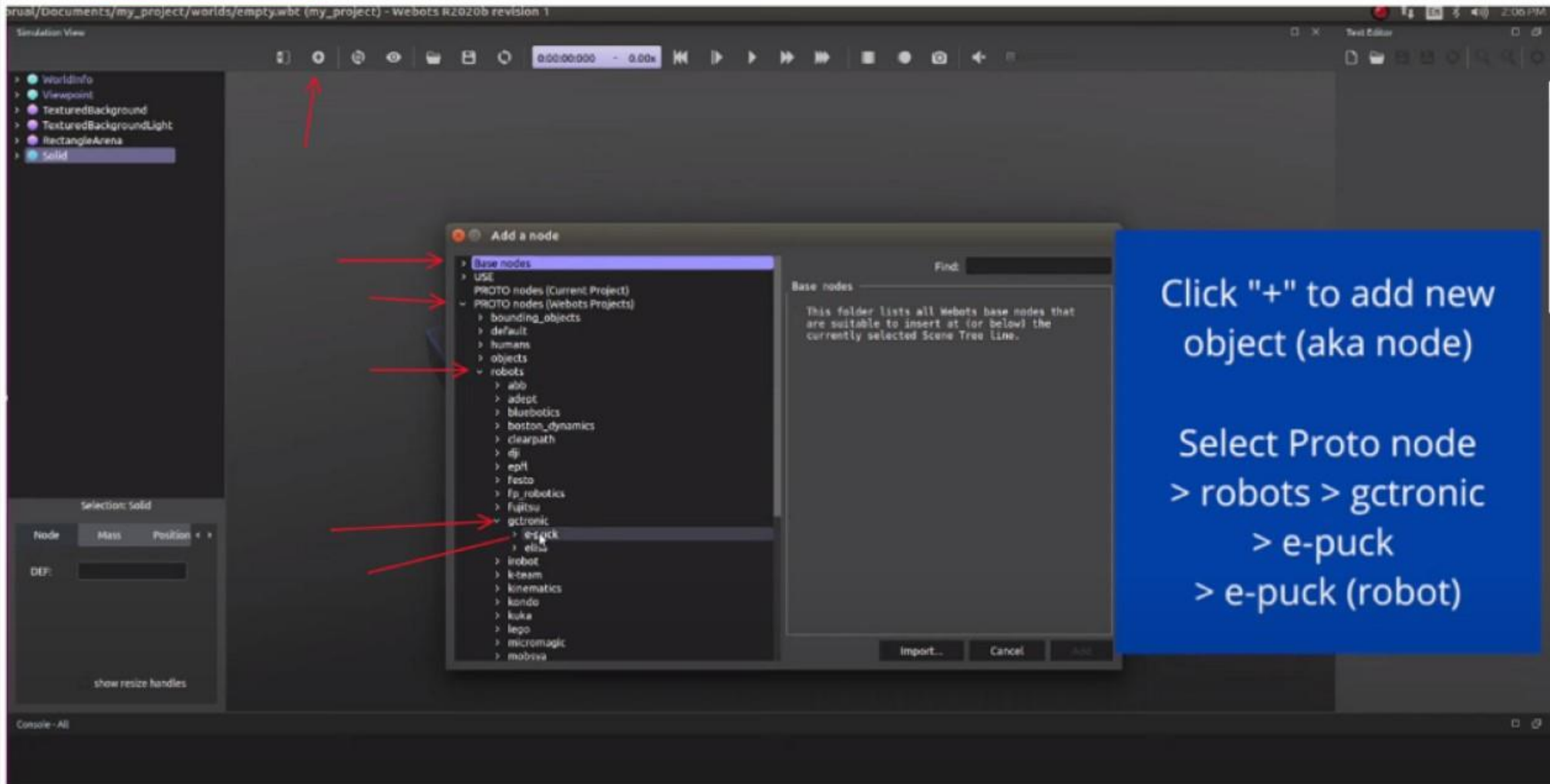
	History of HRI	
4. Strategic HRI System Design	- Core Principles and Communication Strategies in HRI	- Designing Interfaces - Communication Protocol Simulations

NOW using Existing Objects and ROBOTS



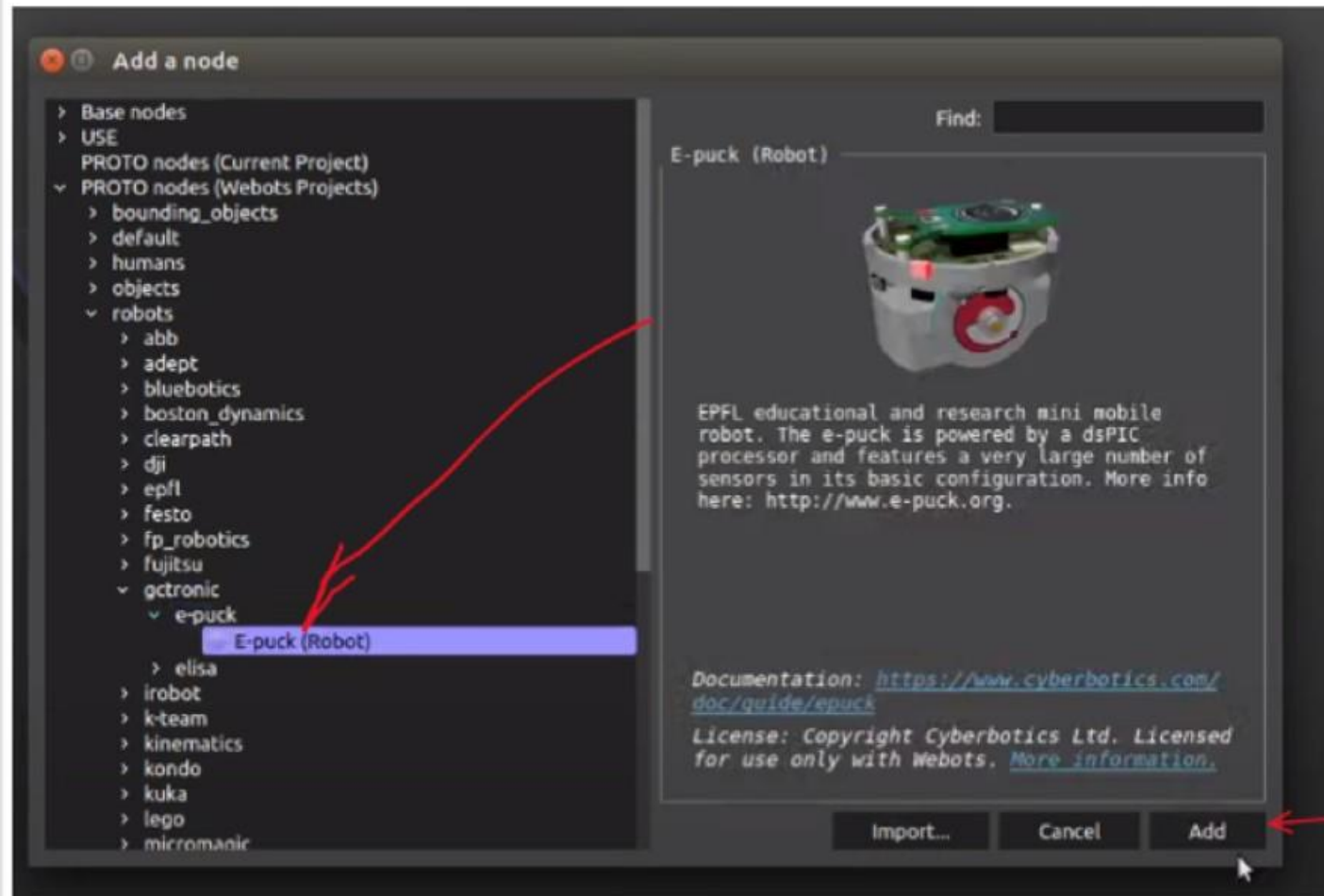


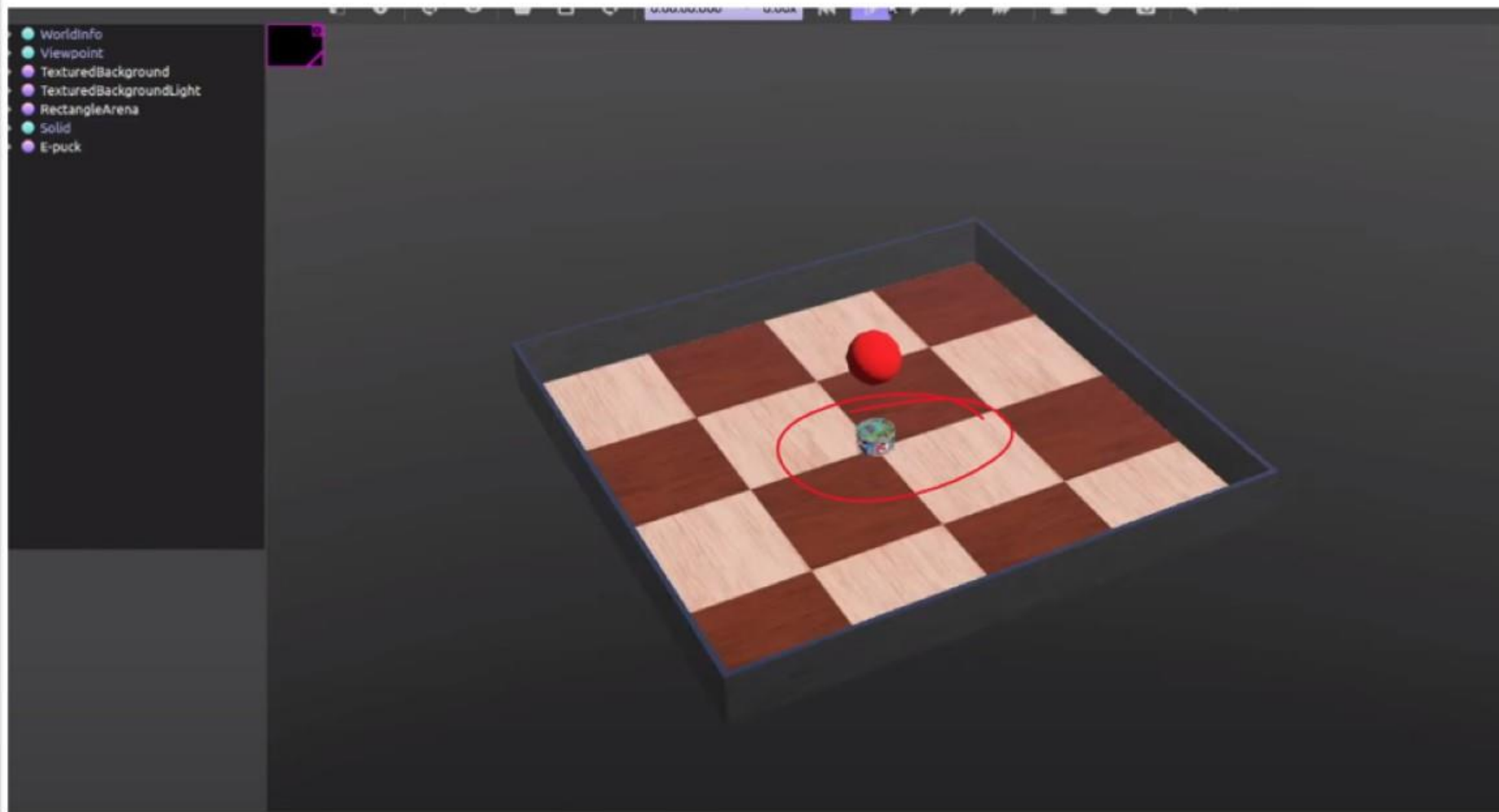
Follow the following selection sequences:



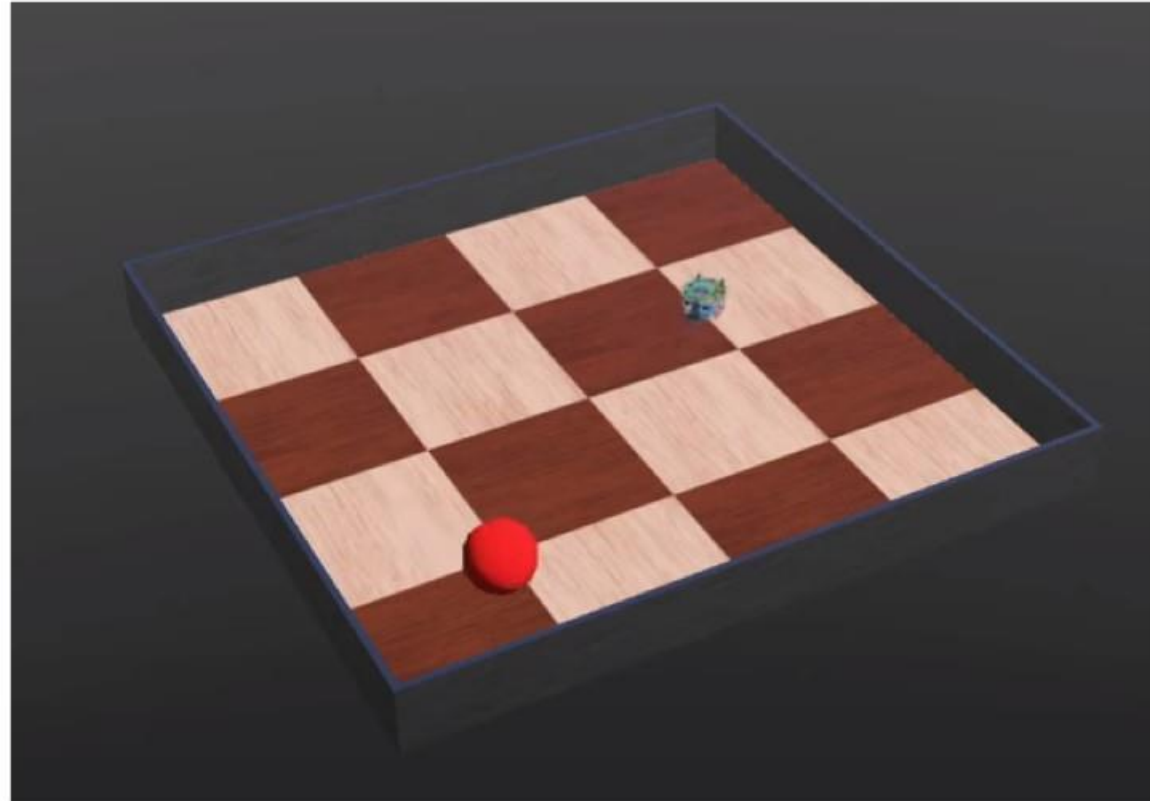
Click "+" to add new object (aka node)

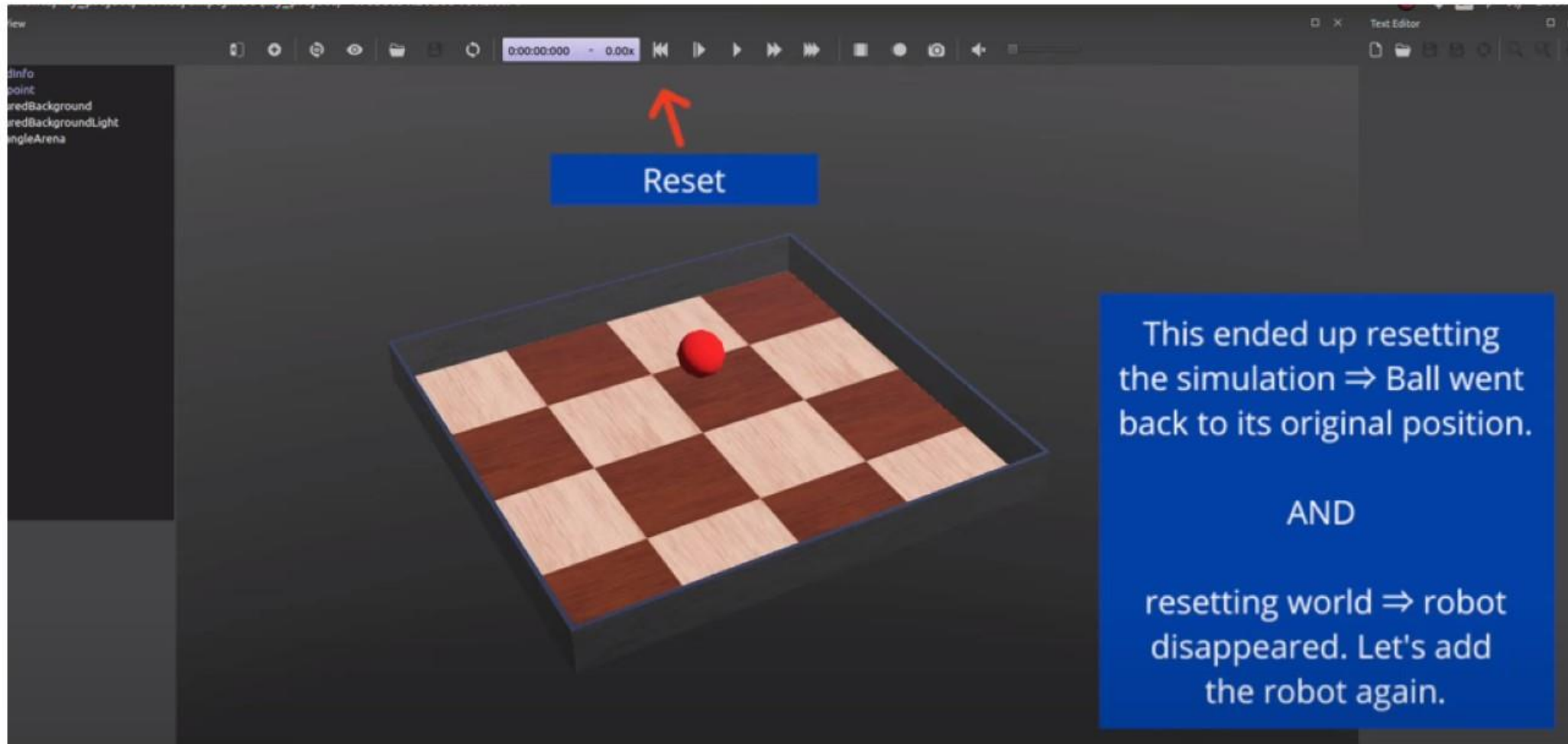
Select Proto node
> robots > gctronic
> e-puck
> e-puck (robot)





When start the simulation (RUN) the Robot continue to drive.

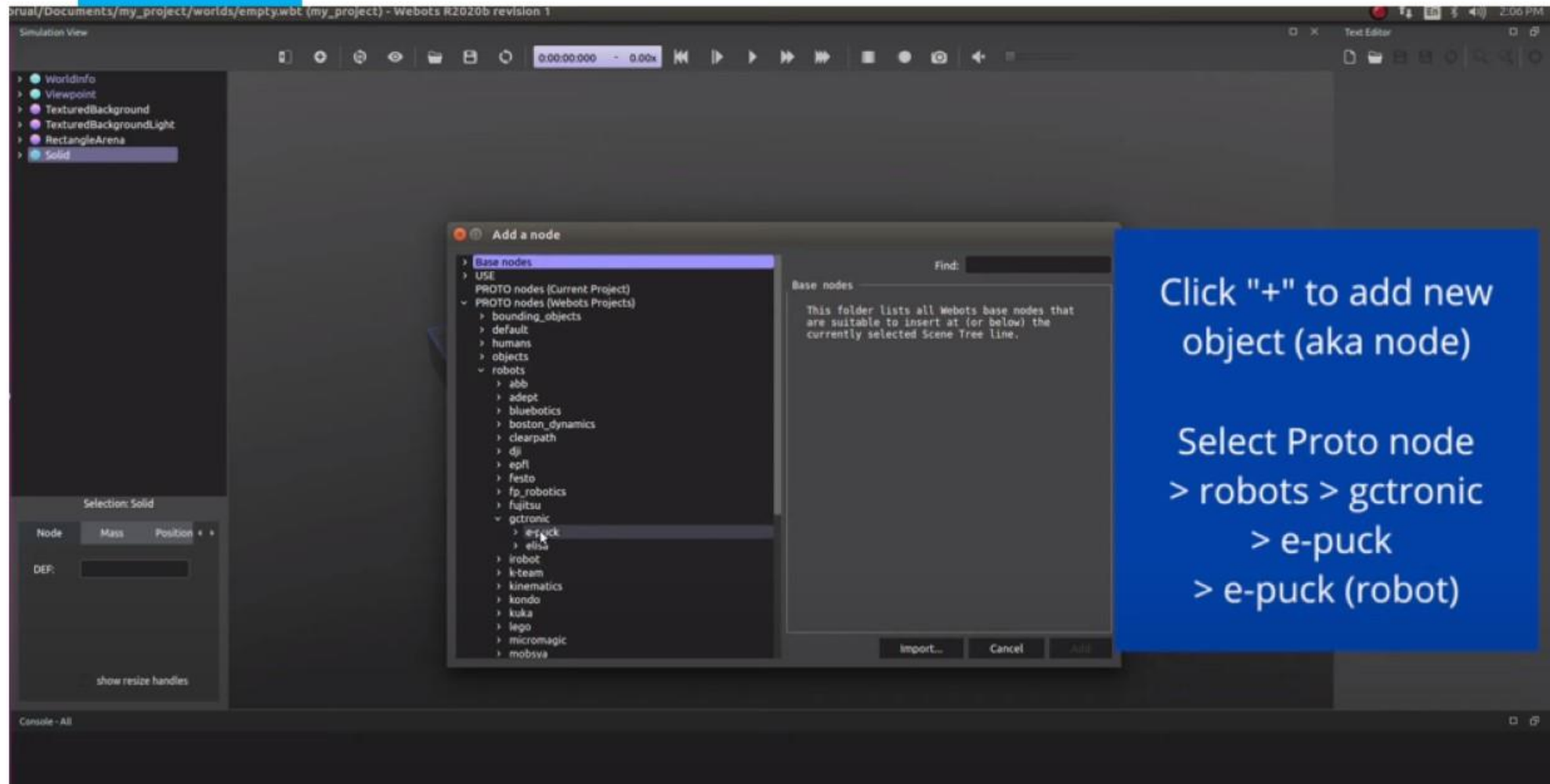


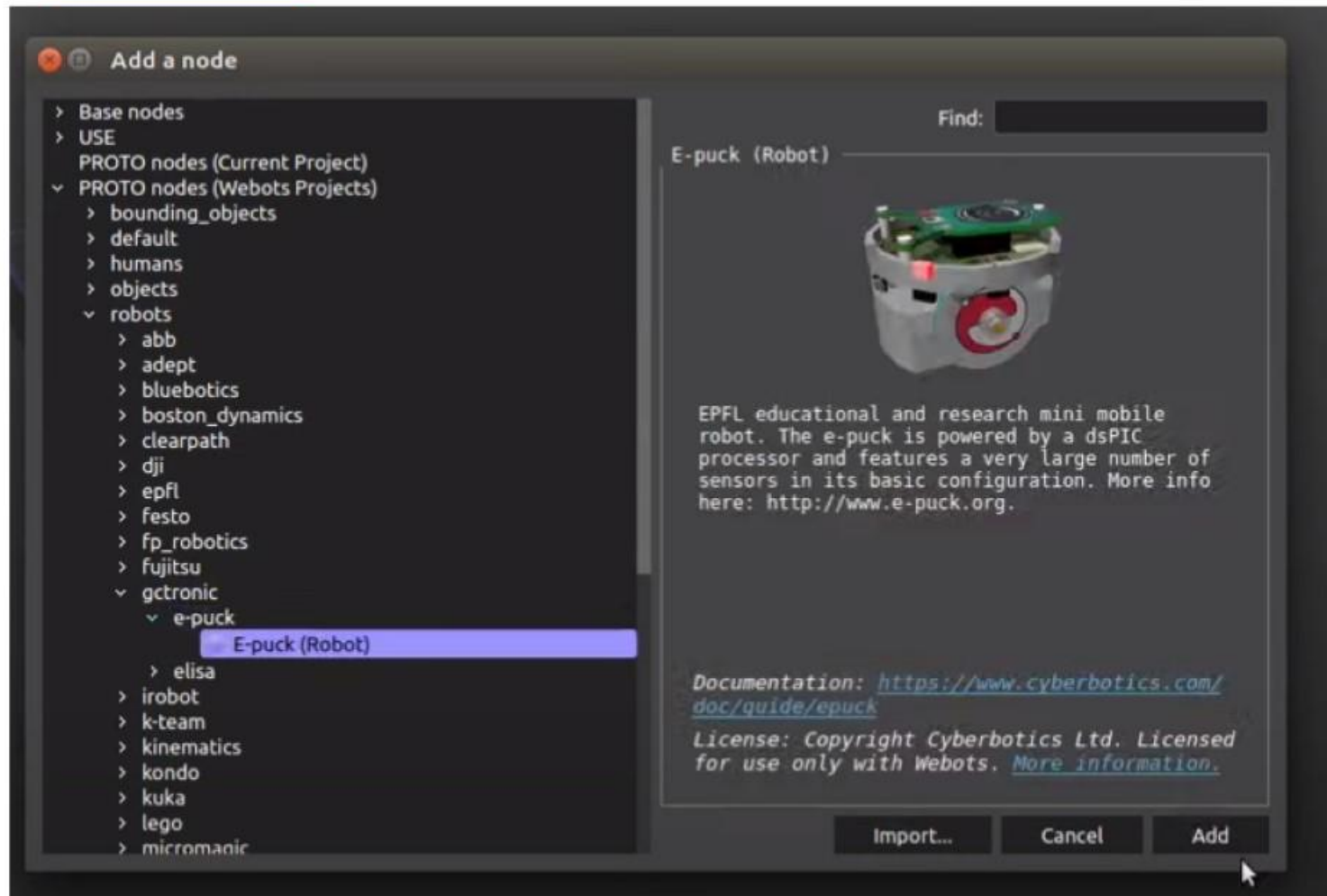


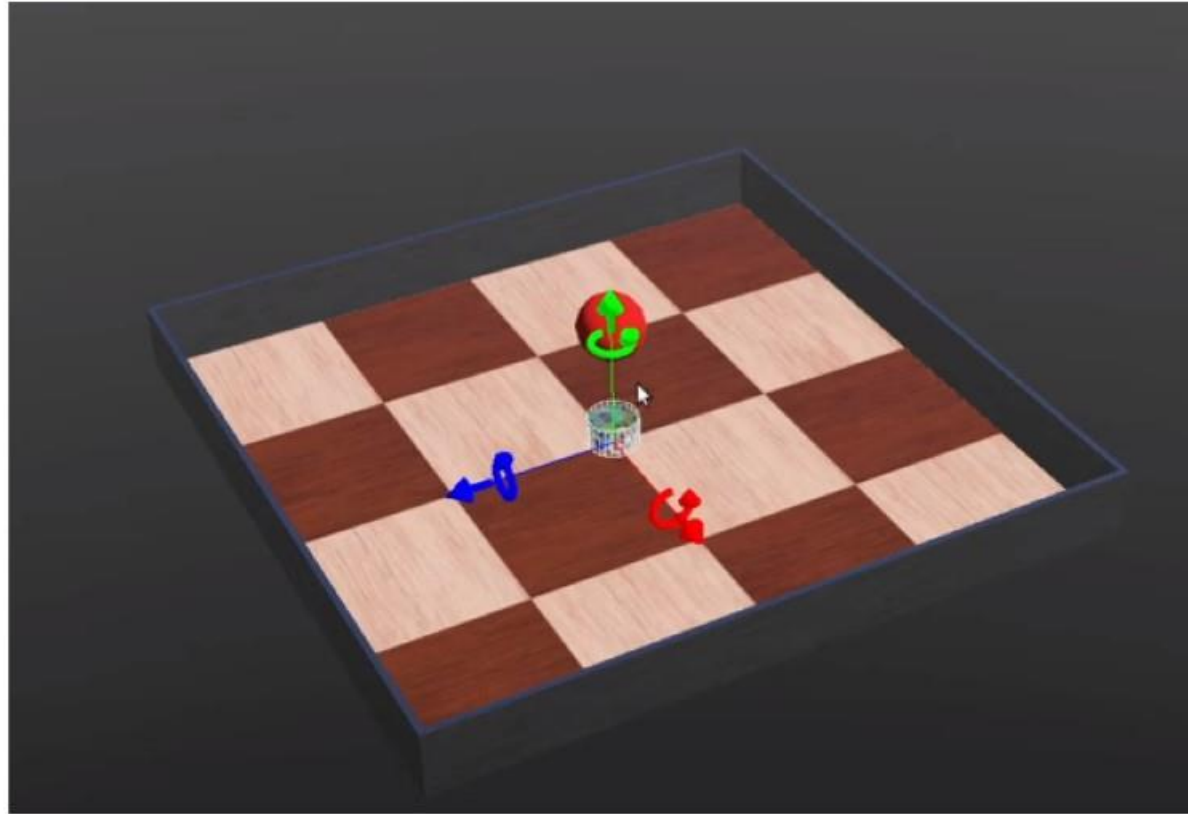
This ended up resetting the simulation $\Rightarrow$ Ball went back to its original position.

AND

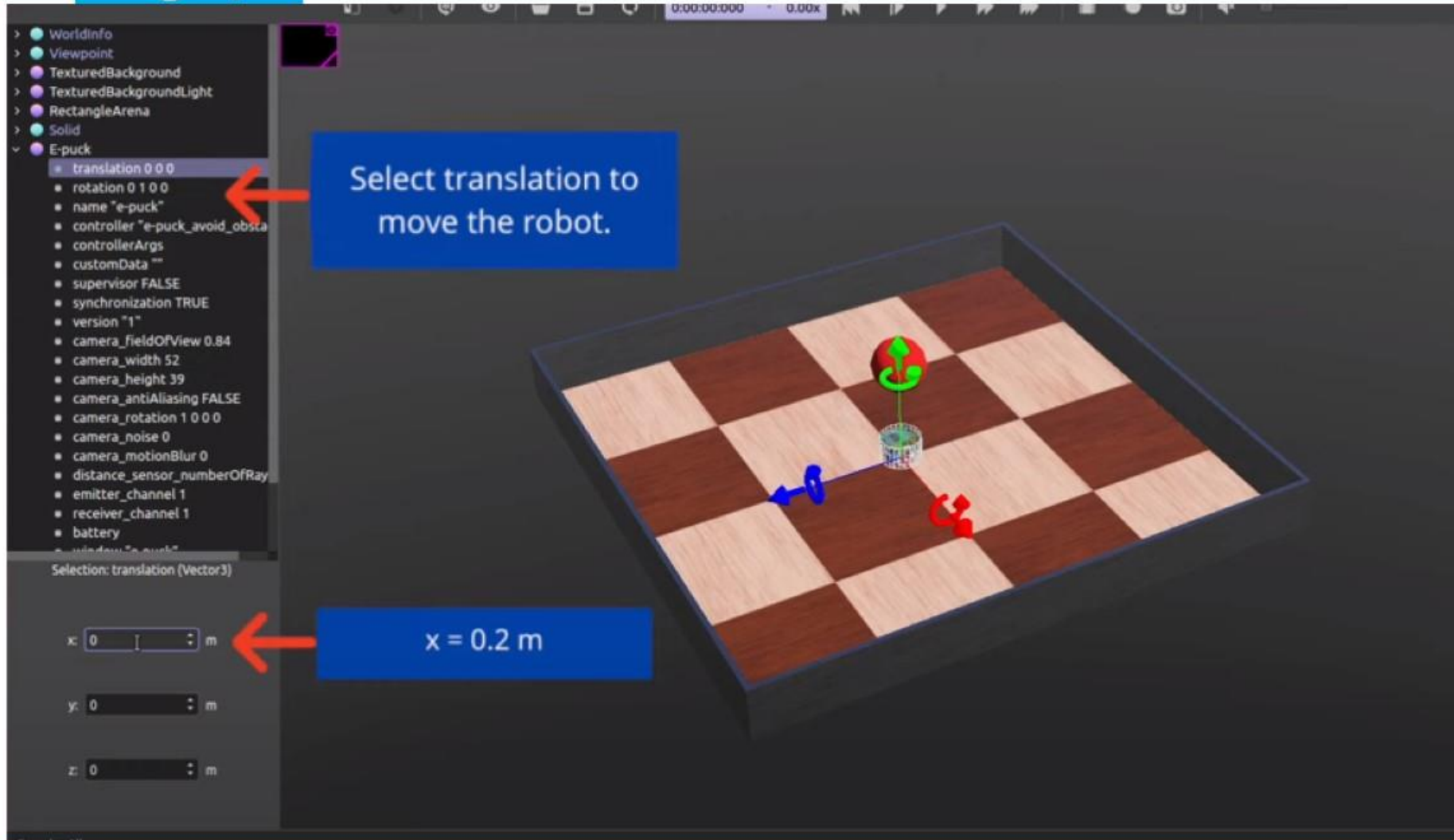
resetting world $\Rightarrow$ robot disappeared. Let's add the robot again.

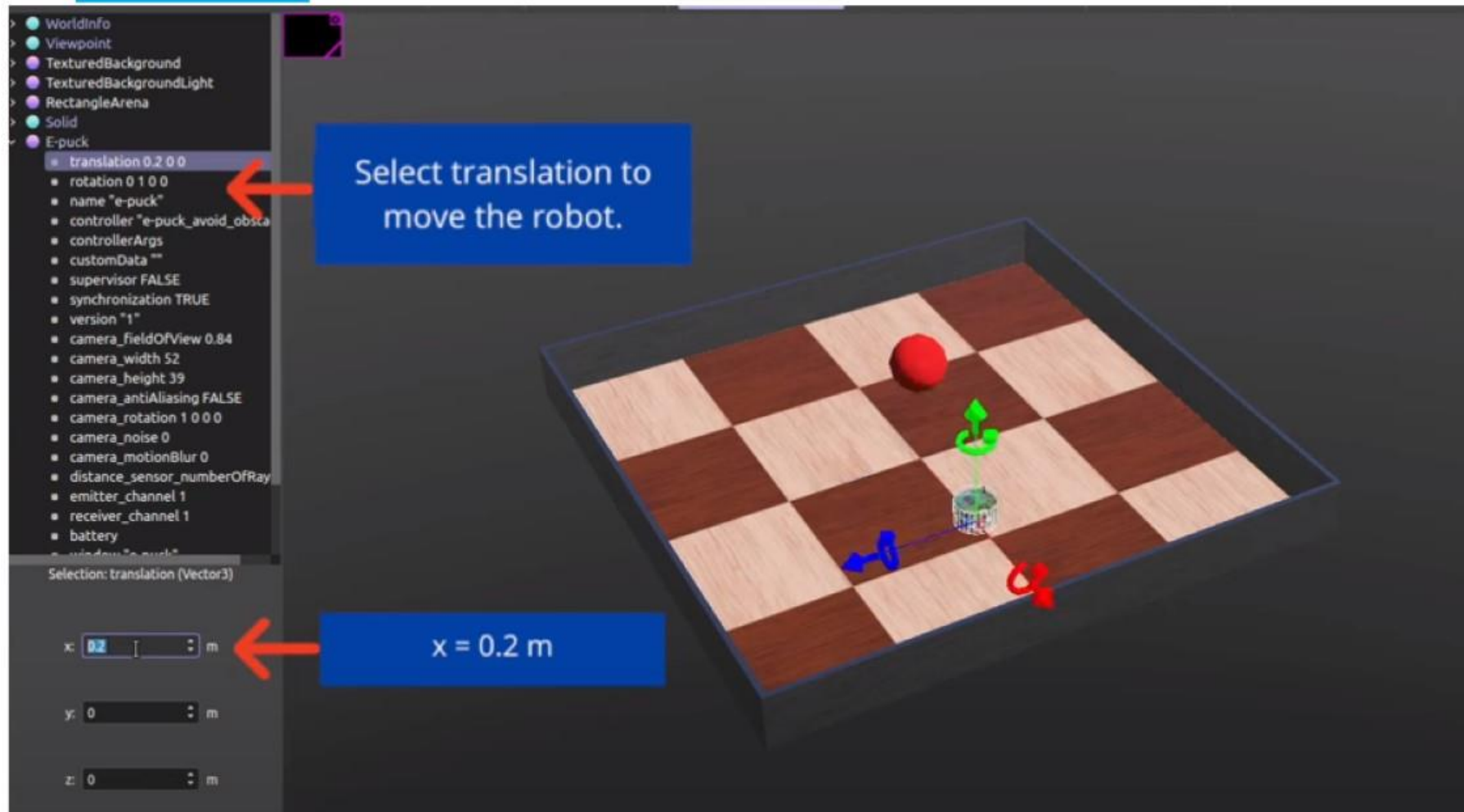


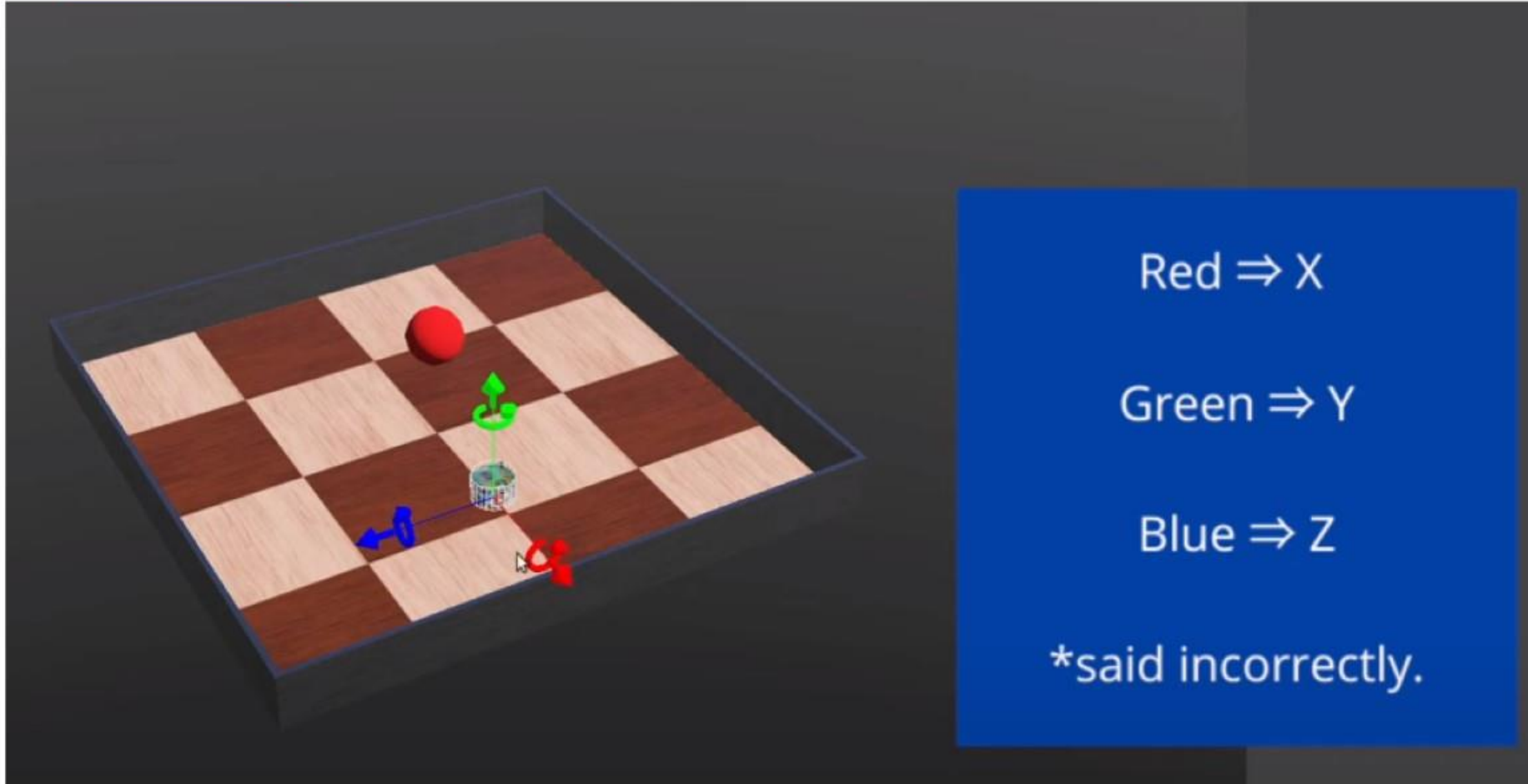


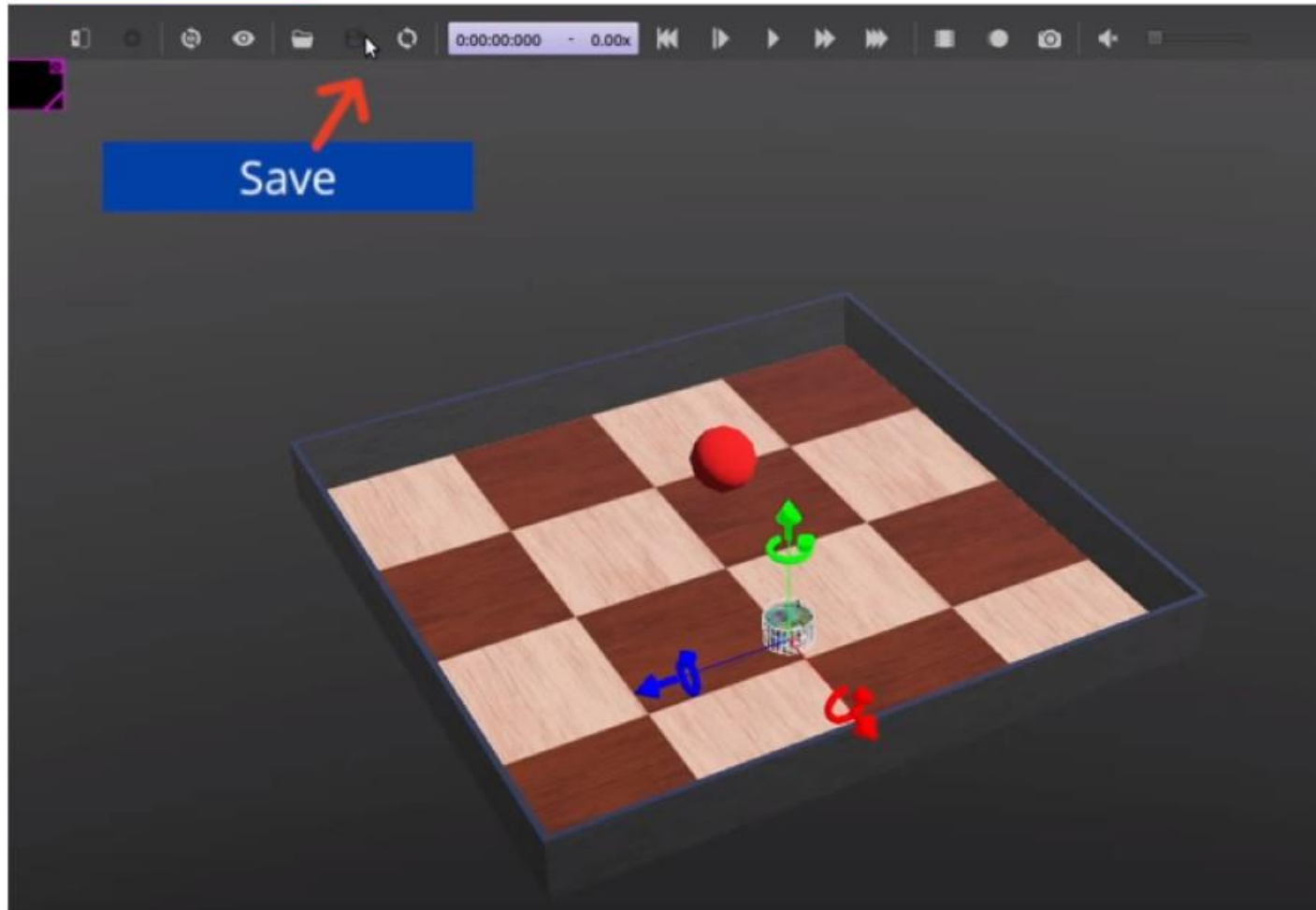


Lets shift robot away from the original point (0,0,0):





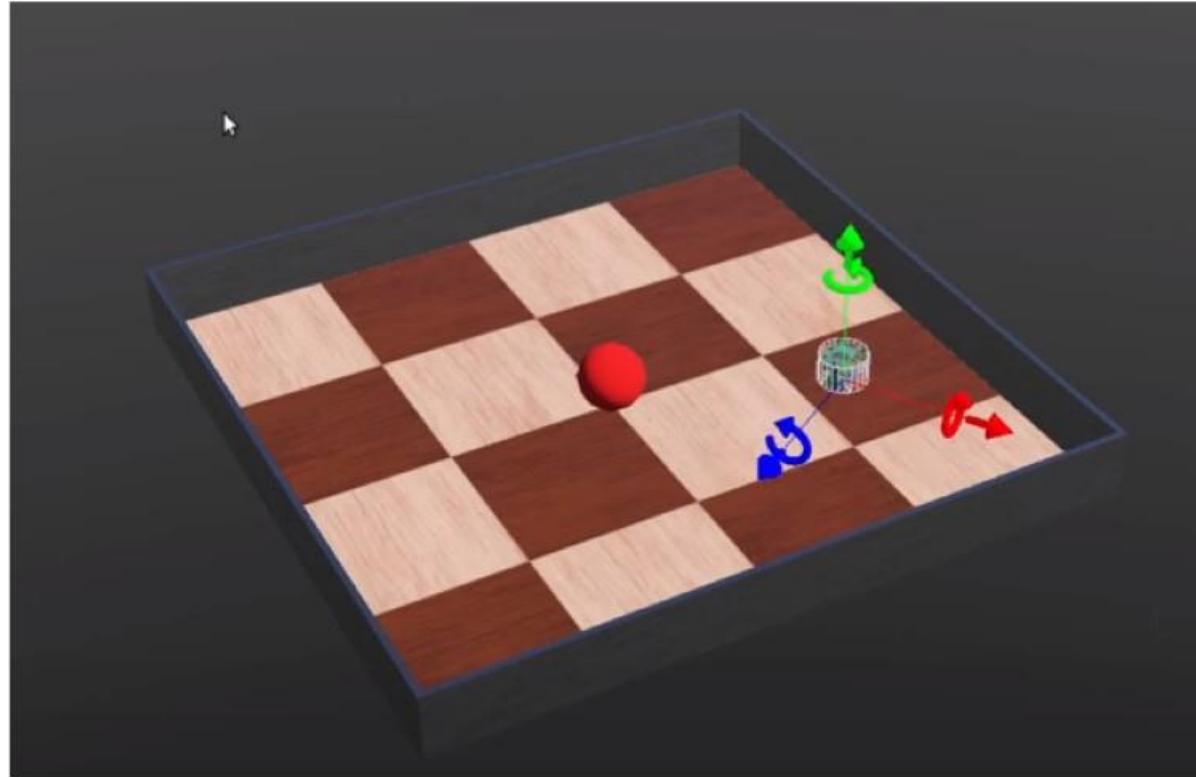


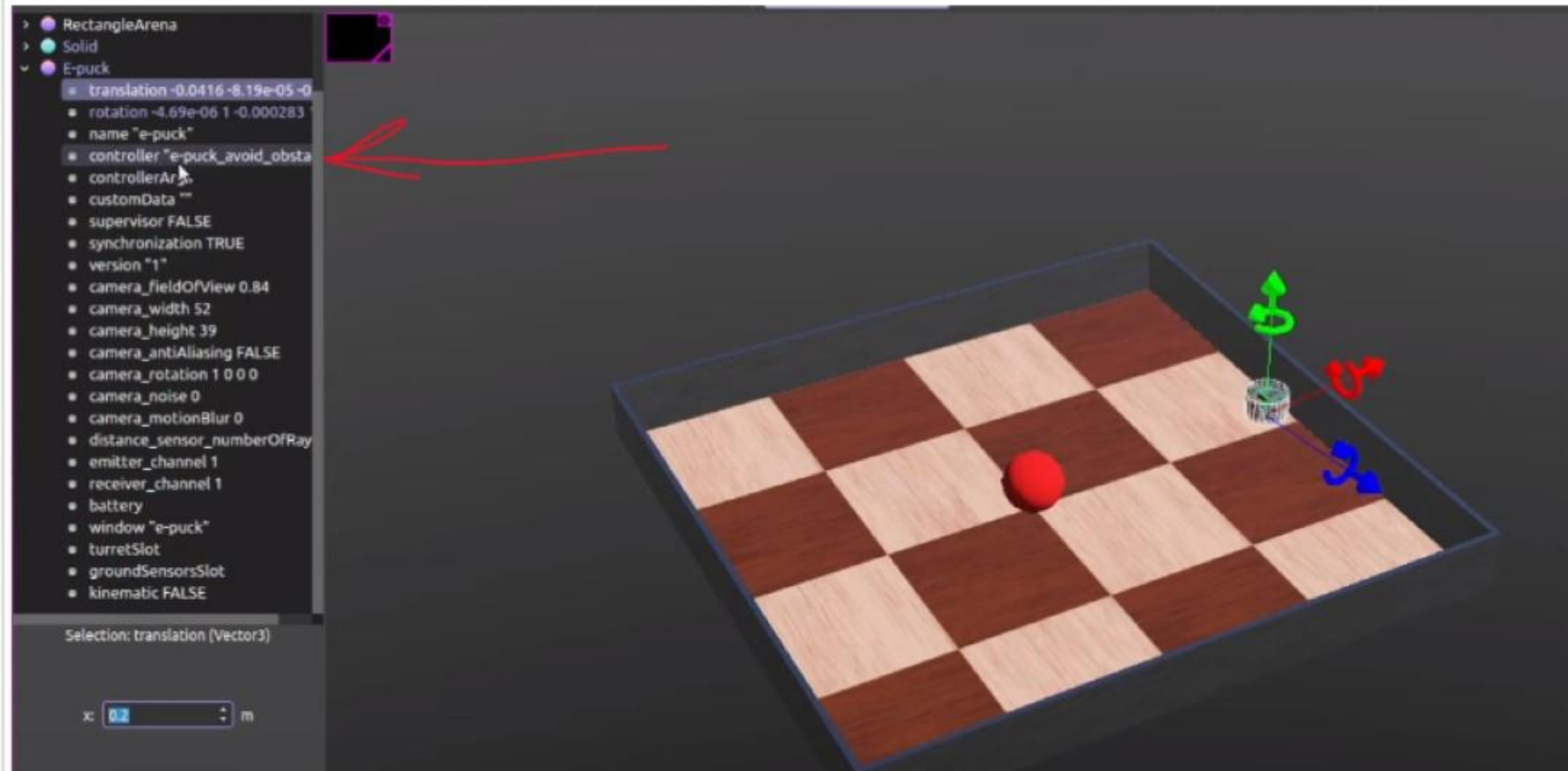


RUN the Simulation now: you will see the robot move ?

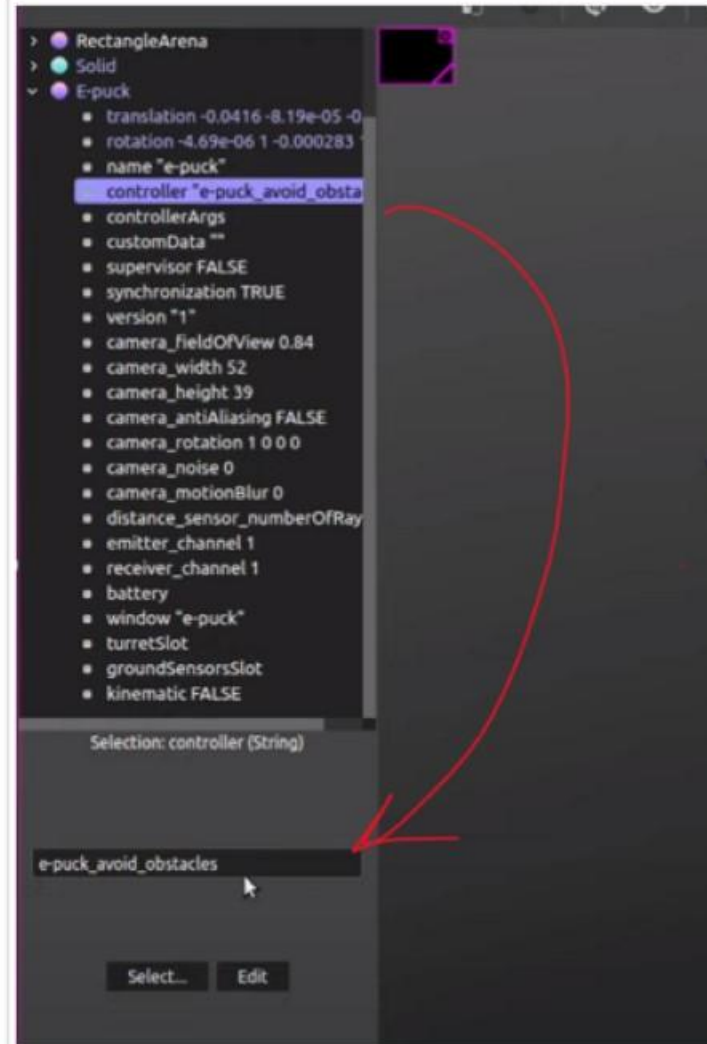
Why?

Because the robot comes with its controller.



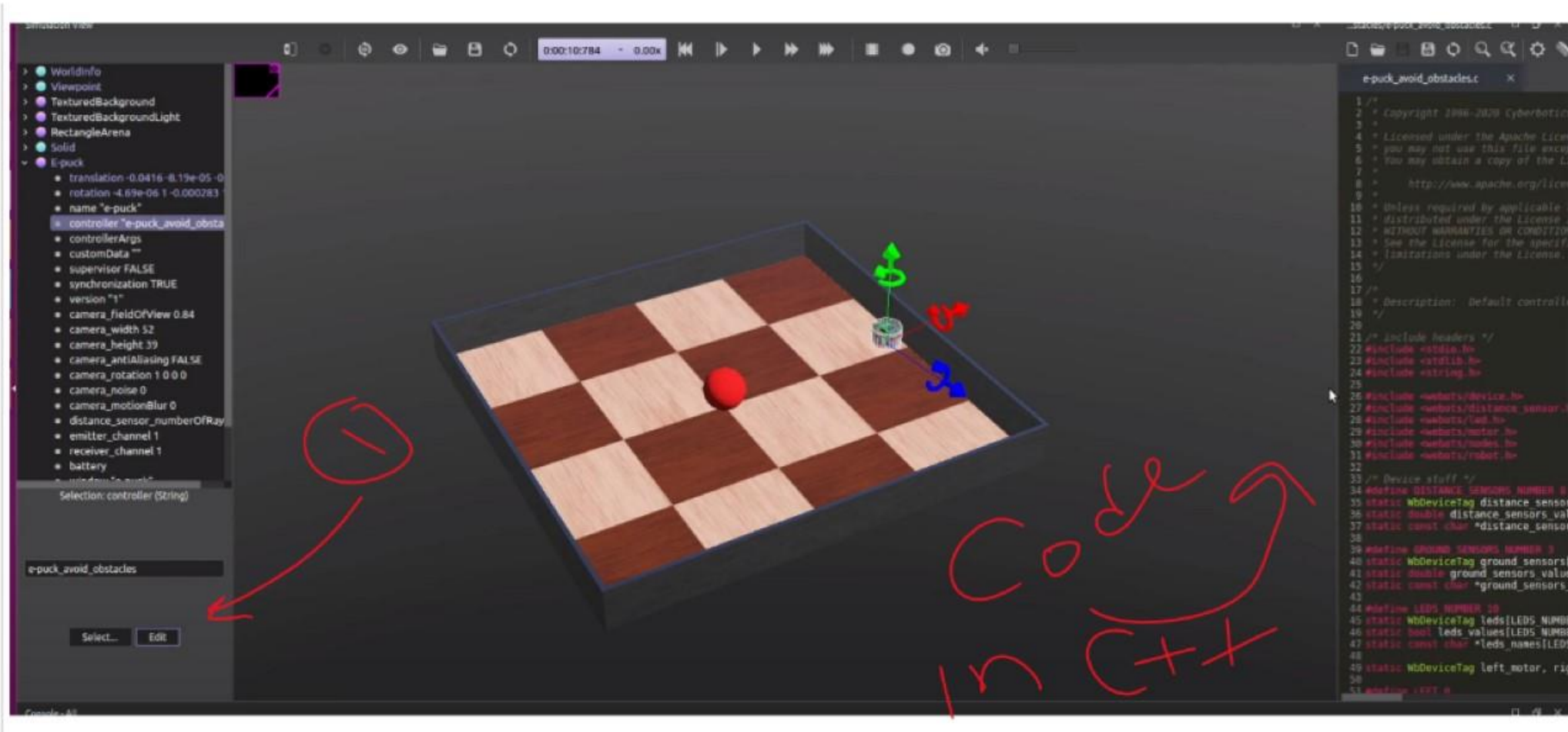
Lets check the robot Controller:

Double click on the Controller:



NOW you can add your own code , Click on the EDIT:

The code is written in C++ but you can write Python code.



We will now program a simple controller that will just make the robot **move forwards**.

A controller is a program that defines the behavior of a robot.

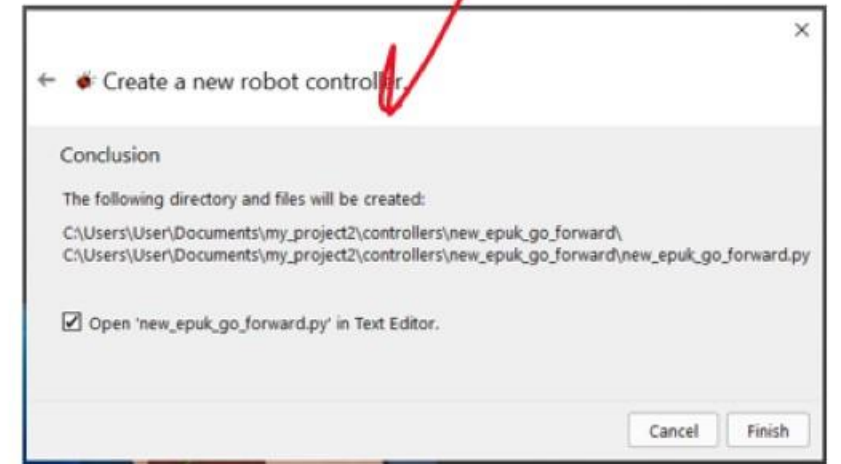
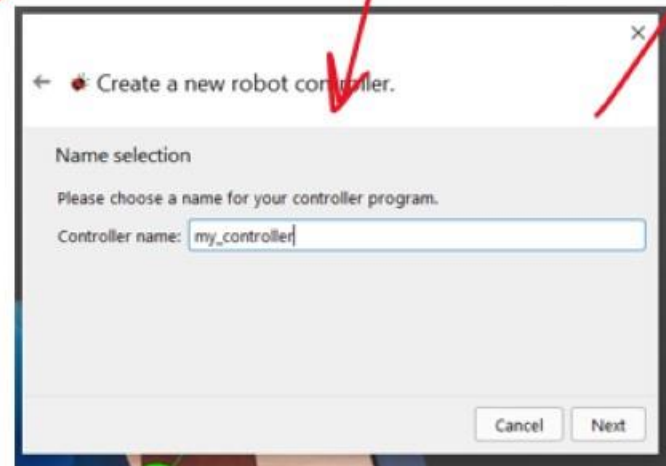
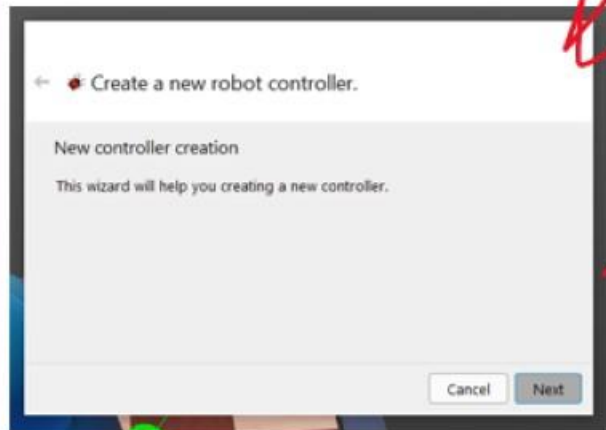
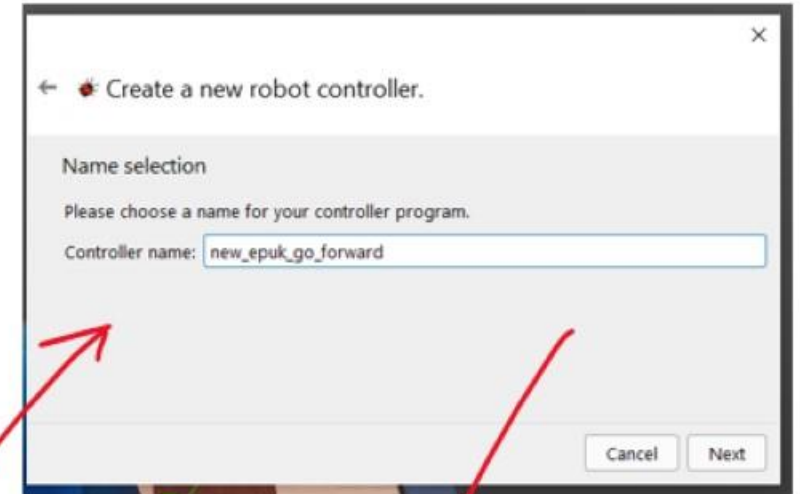
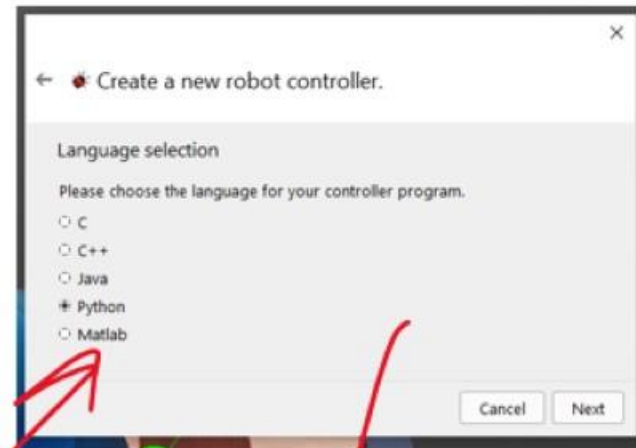
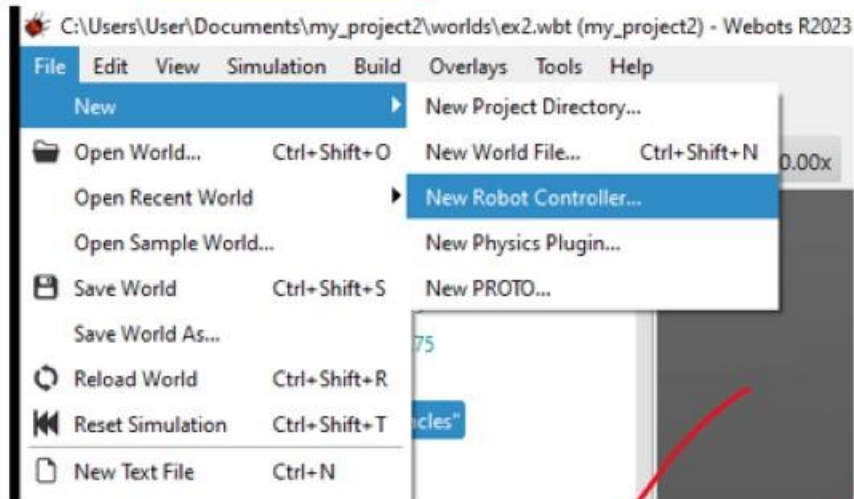
Webots controllers can be written in the following programming languages: C, C++, Java, **Python**, MATLAB, ROS, etc. C, C++ and Java controllers need to be compiled before they can be run as robot controllers.

Python and MATLAB controllers are interpreted languages so they will run without being compiled.

The controller field of a Robot node specifies which controller is currently associated to the robot. Note that the same controller can be used by several robots, but a robot can only use one controller at a time.

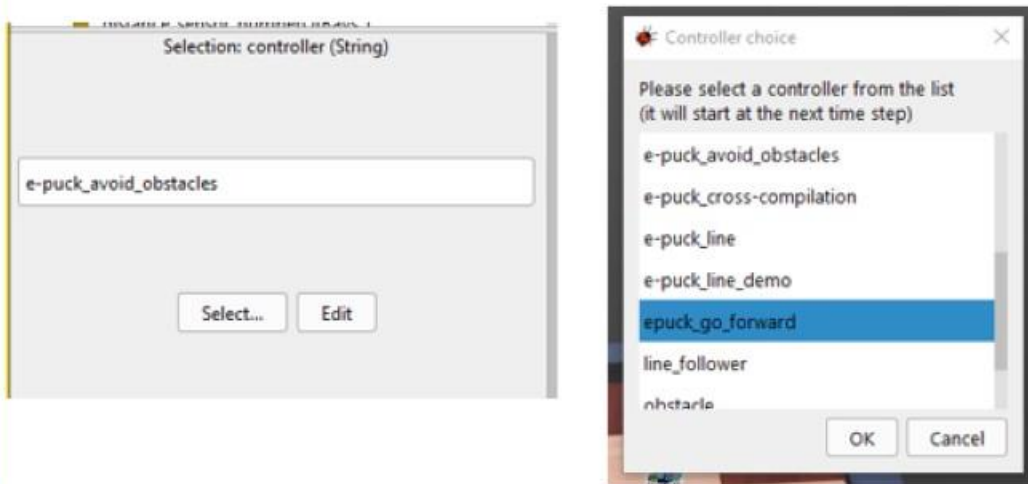
Each controller is executed in a separate child process usually spawned by Webots. Because they are independent processes, controllers don't share the same address space, and may run on different processor cores.

Hands-on #8: Create a new Python controller called `epuck_go_forward` using the **File / New / New Robot Controller...** menu item. This will create a new `epuck_go_forward` (or `EPuckGoForward`) directory in `my_first_simulation/controllers`. Select the option offering you to open the source file in the text editor.



The new source file is displayed in Webots text editor window. We will now associate new `epuck_go_forward` controller to the E-puck node.

In the scene tree view, select the controller field of the E-puck node, then use the field editor at the bottom of the Scene Tree view: press the Select... button and then select `epuck_go_forward` in the list. Once the controller is associated with the robot, save the world. Modify the program by getting the motor devices (`leftMotor = robot.getDevice('left wheel motor')`), and by applying a motor command (`leftMotor.setPosition(10.0)`):



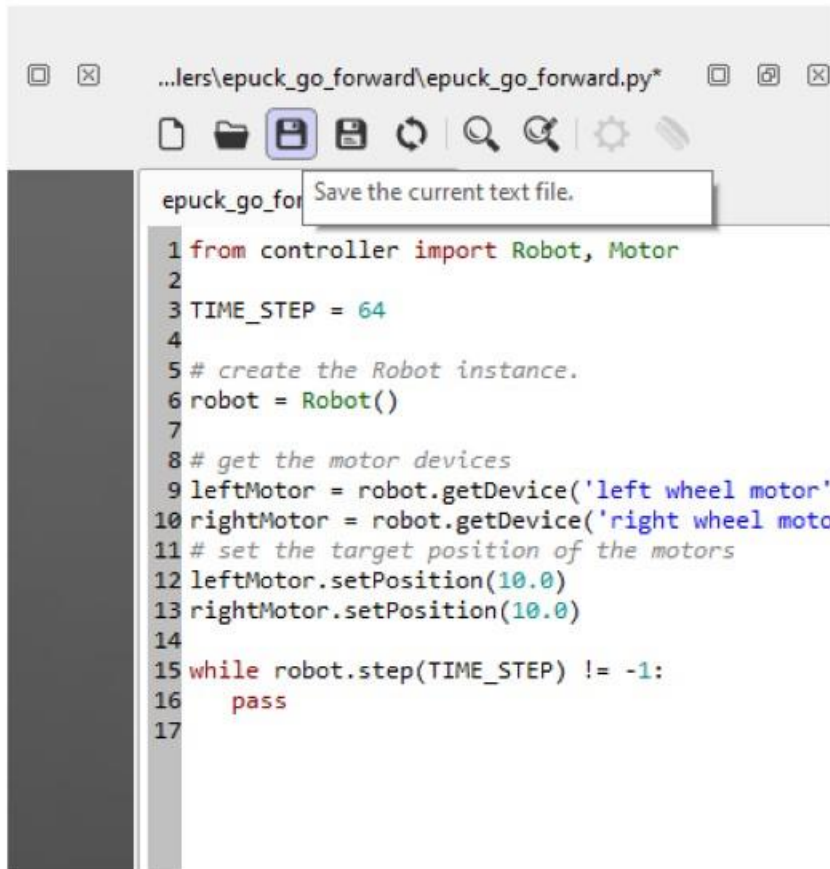
```
from controller import Robot, Motor

TIME_STEP = 64

# create the Robot instance.
robot = Robot()

# get the motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')

# set the target position of the motors
leftMotor.setPosition(10.0)
rightMotor.setPosition(10.0)
while robot.step(TIME_STEP) != -1: pass
```



The screenshot shows a Python IDE window titled "...lers\epuck_go_forward\epuck_go_forward.py*". The toolbar includes icons for file operations and editing. A "Save the current text file." dialog box is open over the code editor. The code in the editor is as follows:

```
1 from controller import Robot, Motor
2
3 TIME_STEP = 64
4
5 # create the Robot instance.
6 robot = Robot()
7
8 # get the motor devices
9 leftMotor = robot.getDevice('left wheel motor')
10 rightMotor = robot.getDevice('right wheel moto
11 # set the target position of the motors
12 leftMotor.setPosition(10.0)
13 rightMotor.setPosition(10.0)
14
15 while robot.step(TIME_STEP) != -1:
16     pass
17
```

robot should move forwards. The robot will move using its maximum speed for a while and then stop once the wheels have rotated of 10 radians.

Step 1: Import Required Classes

```
from controller import Robot, Motor
```

This line imports the **Robot** and **Motor** classes from the **controller** module.

In Webots, the **Robot** class allows you to create a robot object to control your robot in the simulation. The **Motor** class is used to control the motors of the robot, such as wheel motors for moving.

Step 2: Define the Time Step

```
TIME_STEP = 64
```

This sets the time step of the simulation loop in milliseconds. In Webots, simulations progress in discrete steps. Here, **64** milliseconds is chosen as the duration of each simulation step. This means that your simulation will update and execute your commands every 64 milliseconds.

Step 3: Create the Robot Instance

```
robot = Robot()
```

This line creates an instance of the **Robot** class. This instance, called **robot**, represents your robot in the Webots simulation. It is through this object that you can control the robot and access its sensors and actuators (like motors).

Step 4: Get the Motor Devices

```
leftMotor = robot.getDevice('left wheel motor')  
rightMotor = robot.getDevice('right wheel motor')
```

These lines retrieve the motor devices attached to the robot. The **getDevice()** method is used with the name of the device you want to control, which should match the names given in the Webots world file. Here, it's assumed your robot has two motors named 'left wheel motor' and 'right wheel motor'. These motors are then stored in the **leftMotor** and **rightMotor** variables, respectively.

Step 5: Set the Target Position of the Motors

```
leftMotor.setPosition(10.0)  
rightMotor.setPosition(10.0)
```

These lines set the target position for both motors in **radians**. The **setPosition()** method is used to specify where you want the motor to move. Here, **10.0** radians is an arbitrary value, and setting the same value for both motors implies you want the robot to move straight ahead.

Step 6: Main Simulation Loop

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

This is the main simulation loop where you would typically implement your robot's behavior. The **robot.step(TIME_STEP)** method is called to advance the simulation by one time step (64 milliseconds in this case). This method also returns **-1** if the simulation is stopped, which is used here to continue the loop until the simulation is manually stopped. The **pass** statement is a placeholder indicating that no action is taken in the loop.

Robot should move forwards. The robot will move using its maximum speed for a while and then stop once the wheels have rotated of 10 radians.

The robots wheels are often controlled using velocity, and not position like we did in the previous example. In order to control the motors of the wheels in speed you need to set the target position to the infinity and to set the desired speed:

Modify the controller program as shown below, recompile it and run it:

```
from controller import Robot, Motor
```

```
TIME_STEP = 64
```

```
MAX_SPEED = 6.28
```

```
# create the Robot instance.
```

```
robot = Robot()
```

```
# get a handler to the motors and set target position to infinity (speed control)
```

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

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

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

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

```
# set up the motor speeds at 10% of the MAX_SPEED.
```

```
leftMotor.setVelocity(0.1 * MAX_SPEED)
```

```
rightMotor.setVelocity(0.1 * MAX_SPEED)
```

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

```
    pass
```

Step 4: Initialize Motors for Speed Control

```
leftMotor = robot.getDevice('left wheel motor')  
rightMotor = robot.getDevice('right wheel motor')  
leftMotor.setPosition(float('inf'))  
rightMotor.setPosition(float('inf'))
```

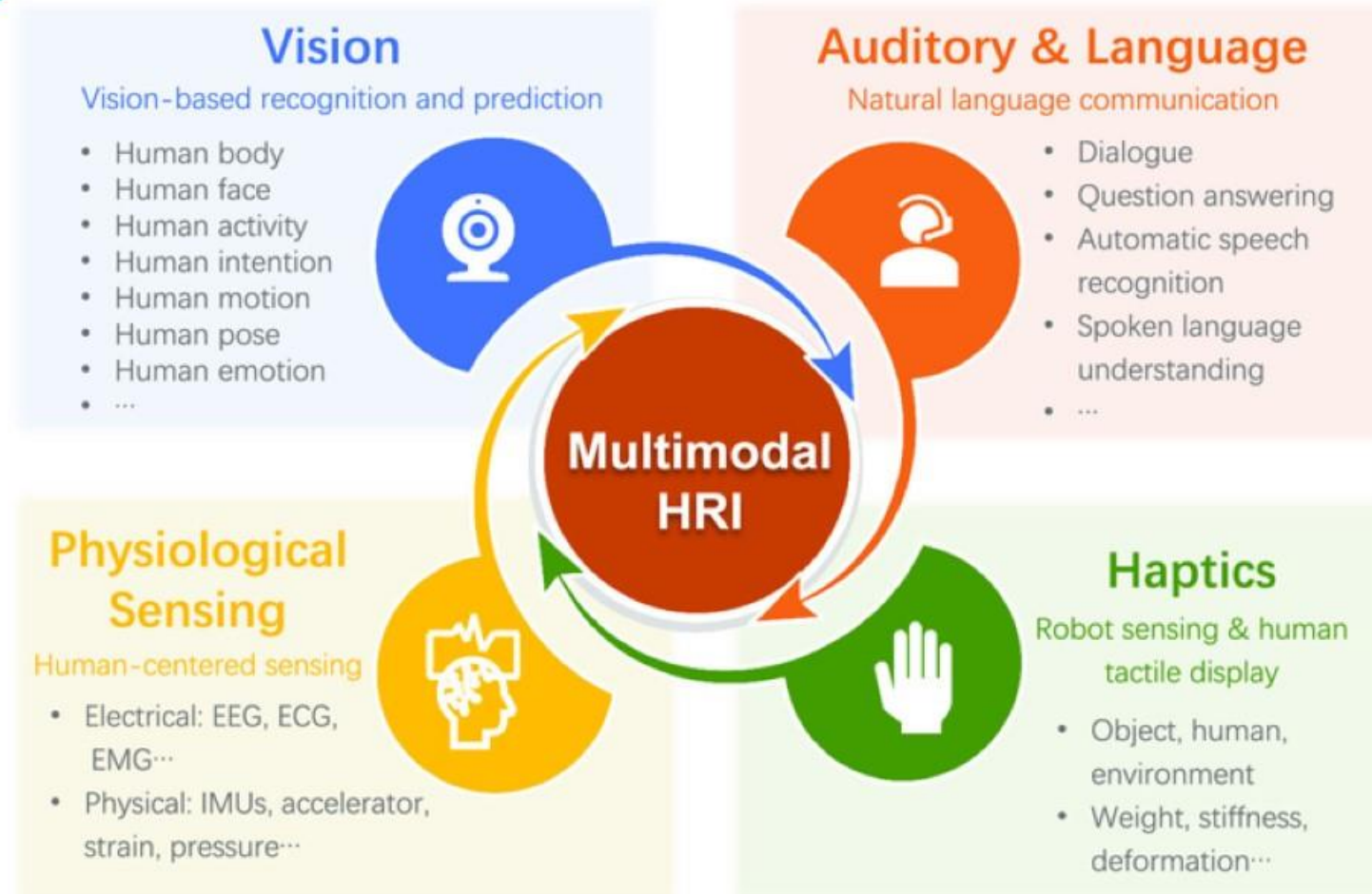
- Get Motor Handlers:** Retrieves references to the robot's left and right wheel motors using their names defined in the simulation world.
- Set Position to Infinity:** Configures the motors for speed control by setting their target positions to infinity (**float('inf')**).

Step 5: Set Motor Speeds

```
leftMotor.setVelocity(0.1 * MAX_SPEED)  
rightMotor.setVelocity(0.1 * MAX_SPEED)
```

Purpose: Determines the initial speed of the motors as a fraction of **MAX_SPEED**. Setting both motors to **0.1 * MAX_SPEED** means the robot will start moving forward at 10% of its maximum speed capability. This is a way to control the robot's speed directly, offering flexibility in how the robot moves.

Strategic Human-Robot Interaction (HRI) system design involves a comprehensive approach to creating systems where humans and robots interact in ways that are efficient, safe, and enhance the capabilities of both. This approach requires consideration of various factors, including user experience, robot autonomy levels, interaction modalities, and the context in which the interaction occurs



Strategic HRI System Design Explained: Begin with a simple definition. For example, "Strategic HRI System Design is the thoughtful planning of how robots and humans interact. It's like creating a friendship where both humans and robots understand and help each other."

Why It's Important:

- The goal of making interactions between humans and robots as smooth and beneficial as possible.
- "Think of it as organizing a team where each member, human or robot, knows their role perfectly and works together seamlessly."

Core Elements of Strategic HRI Design:

- **User Experience (UX):** Explain UX in simple terms, such as "how people feel when they interact with robots. Our goal is to make this experience enjoyable and easy, just like using your favorite app."
- **Autonomy:** Introduce the concept of robot autonomy with a basic explanation: "This is how much freedom a robot has to make its own decisions. Some robots need more guidance, while others can do tasks on their own, similar to how you learn to do new things with practice."
- **Interaction Modalities:** Describe interaction modalities in straightforward language: "This is how we communicate or interact with robots, whether it's talking, using gestures, or even touch screens, much like how you choose to talk, text, or video call your friends."
- **Contextual Adaptation:** Simplify contextual adaptation: "Robots, like us, need to understand different situations. For example, a robot needs to know if it's in a library, it should be quiet, just like we change our behavior based on where we are and who we're with."

			Simulations	
	5. HRI Efficiency and Effectiveness	- Metrics for Mission Effectiveness and Human Factors	- Implementing and Analyzing HRI Metrics	
	6. Innovative HRI	Exploring Innovative Designs	Testing Innovative Designs	



Webots

robot simulation

Tutorial 1: Moving the Robot Forward

Objective: Learn how to initialize your robot and make it move forward.

Step 1: Setting Up Your Webots Project

- Open Webots and create a new project.
- Add an e-puck robot to your scene from the Webots library.

Step 2: Creating Your Controller

- Navigate to **File > New > New Robot Controller...**
- Name your controller **move_forward**.
- Select Python as the programming language and confirm.

Step 3: Writing the Controller Code

Open the **move_forward.py** file generated by Webots and follow along:

```
# Import the Robot class to interact with the robot
from controller import Robot

# Create a robot instance to control it
robot = Robot()

# Define the simulation step time
TIME_STEP = 64

# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

This code initializes the robot but doesn't yet move it. Let's proceed to add movement.

```
# Import the Motor class to control the wheels
from controller import Robot, Motor

# Initialization steps as before

# Retrieve motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set the motors to velocity control mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
# Define a forward speed
FORWARD_SPEED = 6.28
# Apply the speed to both motors to move forward
leftMotor.setVelocity(FORWARD_SPEED)
rightMotor.setVelocity(FORWARD_SPEED)
```

```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Chapter

05

Example 1:

```
# Import the Robot class to interact with the robot
from controller import Robot
```

```
# Create a robot instance to control it
robot = Robot()
```

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

```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Example 2:

```
# Import the Motor class to control the wheels
from controller import Robot, Motor
```

```
# Initialization steps as before
```

```
# Retrieve motor devices
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
# Set the motors to velocity control mode
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
# Define a forward speed
FORWARD_SPEED = 6.28
# Apply the speed to both motors to move forward
leftMotor.setVelocity(FORWARD_SPEED)
rightMotor.setVelocity(FORWARD_SPEED)
```

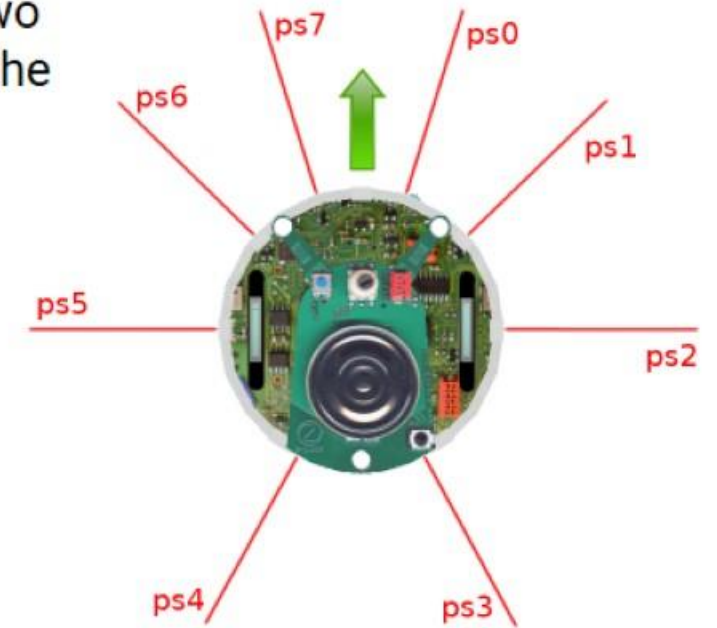
```
# Main control loop
while robot.step(TIME_STEP) != -1:
    # This loop runs until the simulation is stopped
    pass # We will add movement code in the next tutorial
```

Objective: Modify the robot's behavior to stop when an obstacle is detected.

Step 1: Enhancements to the Controller

Build upon the **move_forward** controller by adding sensor functionality to detect obstacles.

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**.



```
# Import necessary classes
```

```
from controller import Robot, Motor, DistanceSensor
```

```
# Initialization steps as before
```

```
# Initialize the front distance sensor
```

```
frontSensor = robot.getDevice('ps0')
```

```
frontSensor.enable(TIME_STEP)
```

```
# Control loop
```

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

```
    # Read the sensor value
```

```
    distance = frontSensor.getValue()
```

```
    # Check for obstacles
```

```
    if distance < 800: # Adjust based on your environment
```

```
        # An obstacle is detected, stop the robot
```

```
        leftMotor.setVelocity(0)
```

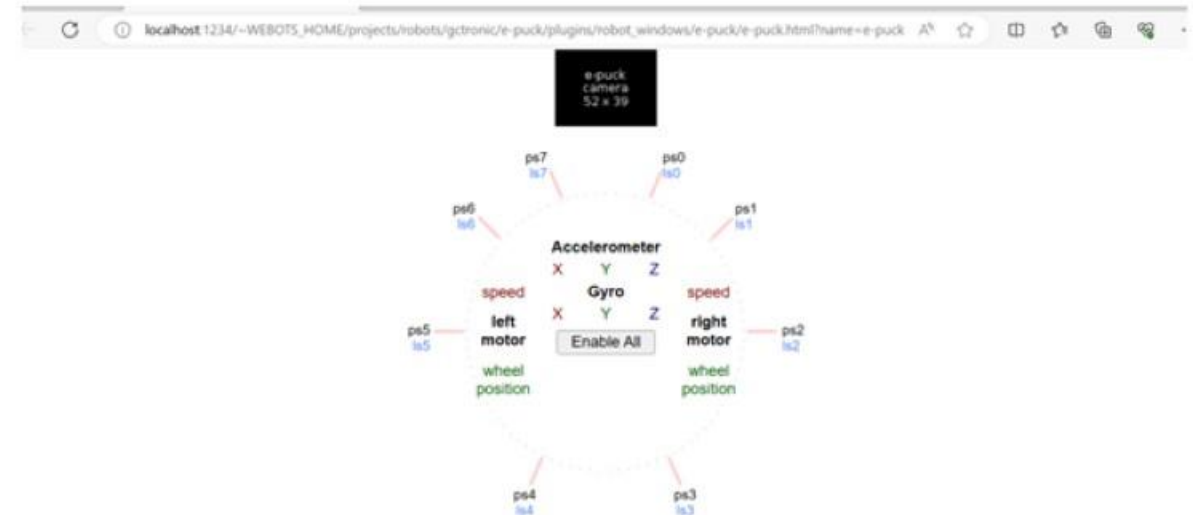
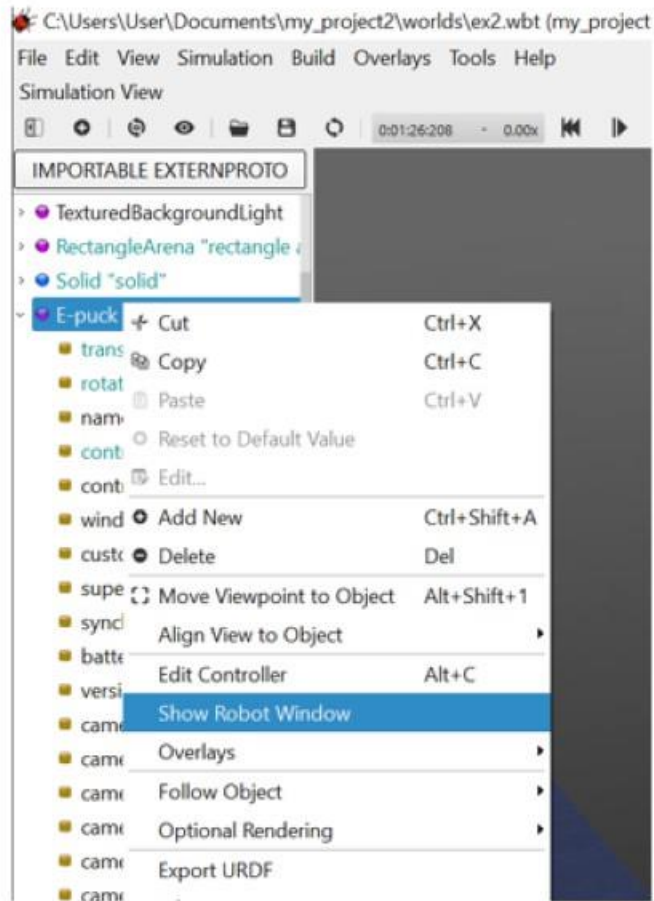
```
        rightMotor.setVelocity(0)
```

```
    else:
```

```
        # No obstacle, move forward
```

```
        leftMotor.setVelocity(FORWARD_SPEED)
```

```
        rightMotor.setVelocity(FORWARD_SPEED)
```



```
from controller import Robot, Motor, DistanceSensor

TIME_STEP = 64
MAX_SPEED = 6.28 # Adjusting MAX_SPEED for realism,
                  # assuming it's in radians/sec which is typical in
                  # simulations

# create the Robot instance.
robot = Robot()

# get a handler to the motors and set target position to
infinity (speed control)
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))

# Initialize and enable the distance sensor
distanceSensor = robot.getDevice('ps0') # Assuming
'sp0' is the correct sensor name
distanceSensor.enable(TIME_STEP)
```

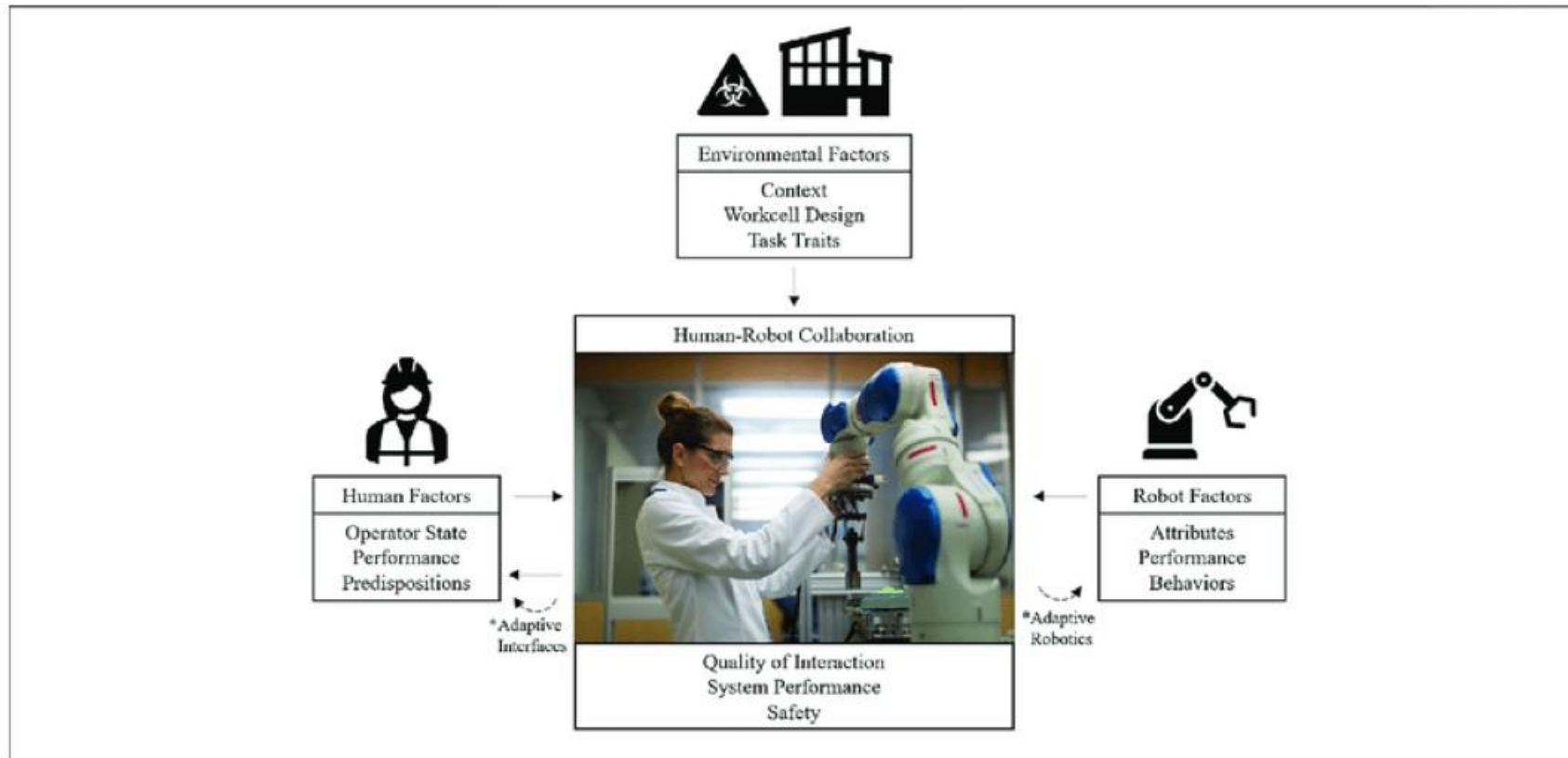
```
# set up the motor speeds at 10% of the MAX_SPEED.
leftMotor.setVelocity(0.1 * MAX_SPEED)
rightMotor.setVelocity(0.1 * MAX_SPEED)

while robot.step(TIME_STEP) != -1:
    # Read the sensor value
    distance = distanceSensor.getValue()
    print(distance)

    # Check for obstacles - assuming a threshold value for when to
    stop
    # You may need to adjust the threshold based on your
    simulation environment
    if distance < 62: # Example threshold value, adjust based on
        your sensor's range and sensitivity
        # An obstacle is detected, stop the robot
        leftMotor.setVelocity( MAX_SPEED)
        rightMotor.setVelocity(0)
    else:
        # No obstacle, continue moving forward
        leftMotor.setVelocity(0.1 * MAX_SPEED)
        rightMotor.setVelocity(0.1 * MAX_SPEED)
```



The concepts of HRI efficiency and effectiveness metrics for mission effectiveness and human factors.



How Fast Robots and Humans Complete Tasks Together:

Explanation: Checking if a robot and human can finish a job quickly, like cleaning up a room.

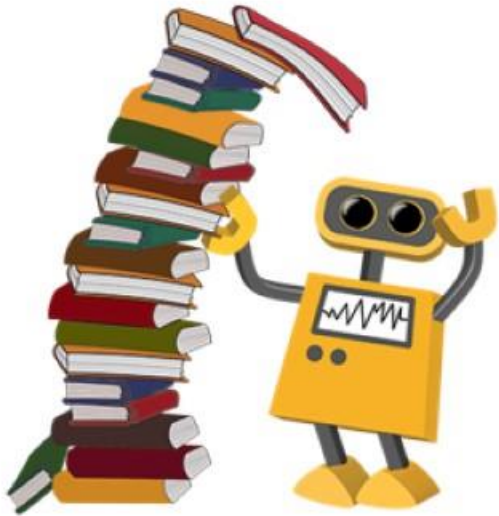
Example: If a robot helps a human clean a room and they finish in 30 minutes instead of an hour, that's good and means they work well together.



Doing the Job Right:

Explanation: Making sure the robot does its part of the job correctly, like a robot that sorts books by color without mistakes.

Example: A robot helps sort red and blue books into separate boxes. If it puts all the books in the right boxes without mixing them up, it's doing a great job.



Robots Working on Their Own:

Explanation: Seeing if the robot can do tasks by itself without needing a human to tell it what to do every time.

Example: A robot vacuum cleans the floors by itself every day without someone having to start it each time.



Changing Plans When Needed:

- Explanation:** The robot can change what it's doing based on new situations, like avoiding spilled juice on the floor it didn't know was there.
- Example:** If a robot is delivering mail indoors and finds a hallway blocked, it figures out a different way to go on its own.



Keeping the Robot Busy:

- **Explanation:** Making sure the robot has tasks to do so it's not just sitting around unused.
- **Example:** A robot in a store helps customers find items, clean up, and restock shelves all day instead of standing idle.



How Happy People Are with the Robot:

- Explanation:** After working with the robot, people feel good about it and think it's helpful.
- Example:** After using a robot assistant in class, students feel it's fun and helps them learn better.



Thinking Hard?:

- **Explanation:** Checking if using the robot makes people feel stressed or tired from thinking too much.
- **Example:** If a teacher uses a robot to grade tests and finds it easy to use without getting a headache, that's great.



Trusting the Robot:

- **Explanation:** People believe the robot will do its job well and not make mistakes.
- **Example:** A family trusts their robot pet sitter to feed and play with their cat correctly when they are not home.



Learning to Use the Robot:

- **Explanation:** Finding out if it's easy and quick for someone to learn how to work with the robot.
- **Example:** If someone can start using a robot to make coffee after a quick lesson, that means it's easy to learn.



Making Human Jobs Easier:

- **Explanation:** The robot takes over tasks that are hard or boring, so people can do other important things.
- **Example:** A robot takes over cleaning the floors at a restaurant, so the staff can spend more time making meals and helping customers.



