



Virtual Short Course In Future Technology “ROBOTICS SIMULATION USING WEBOTS”



**School of Applied Science, Telkom University
BANDUNG, 10 December 2022**

SECTION 1 : BASIC THEORY

Introduction to Webots

Webots is a professional mobile robot simulation software package. It offers a rapid prototyping environment, that allows the user to create 3D virtual worlds with physics properties such as mass, joints, friction coefficients, etc. The user can add simple passive objects or active objects called mobile robots. These robots can have different locomotion schemes (wheeled robots, legged robots, or flying robots). Moreover, they may be equipped with a number of sensor and actuator devices, such as distance sensors, drive wheels, cameras, motors, touch sensors, emitters, receivers, etc. Finally, the user can program each robot individually to exhibit the desired behavior. Webots contains a large number of robot models and controller program examples to help users get started.

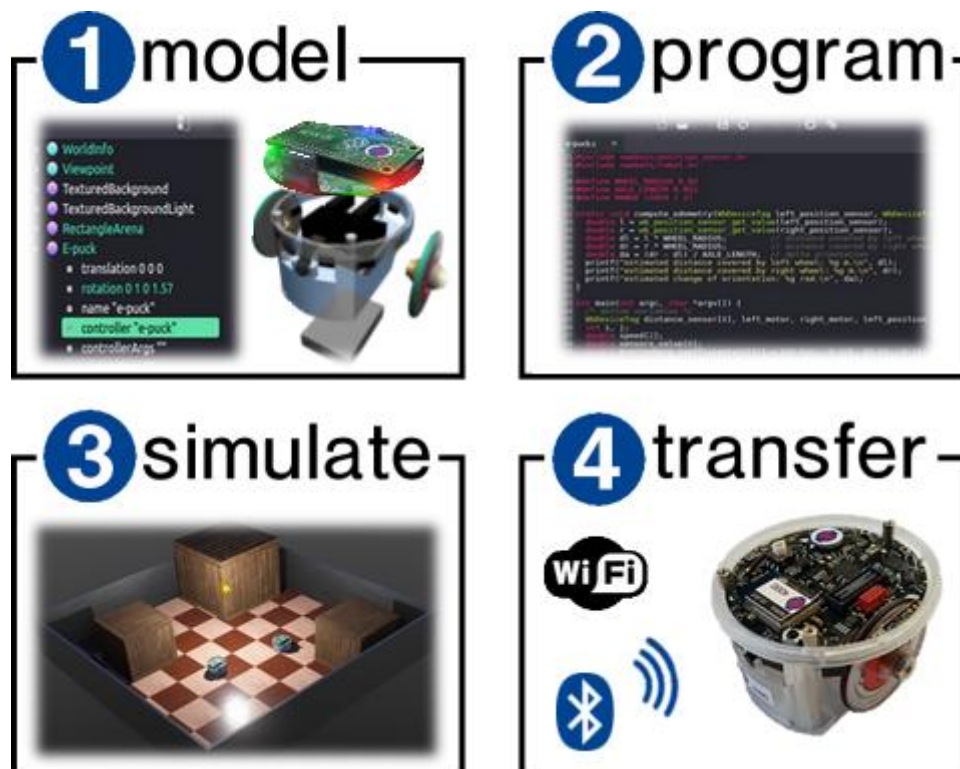


Figure 1 Webots Purposes.

Webots also contains a number of interfaces to real mobile robots, so that once your simulated robot behaves as expected, you can transfer its control program to a real robot like e-puck, DARwIn-OP, Nao, etc. Adding new interfaces is possible through the related system.

What Can I Do with Webots?

Webots is well suited for research and educational projects related to mobile robotics. Many mobile robotics projects have relied on Webots for years in the following areas:

- Mobile robot prototyping (academic research, the automotive industry, aeronautics, the vacuum cleaner industry, the toy industry, hobbyists, etc.)
- Robot locomotion research (legged, humanoids, quadrupeds robots, etc.)
- Multi-agent research (swarm intelligence, collaborative mobile robots groups, etc.)
- Adaptive behavior research (genetic algorithm, neural networks, AI, etc.).
- Teaching robotics (robotics lectures, C/C++/Java/Python programming lectures, etc.)
- Robot contests (e.g. Robotstadium or Rat's Life)

What Do I Need to Know to Use Webots?

You will need a minimal amount of technical knowledge to develop your own simulations:

- A basic knowledge of the **C, C++, Java, Python** or **MATLAB** programming language is necessary to program your own robot controllers. However, even if you don't know these languages, you can still program the e-puck and Hemisson robots using a simple graphical programming language called BotStudio.
- If you don't want to use existing robot models provided within Webots and would like to create your own robot models, or add special objects in the simulated environments, you will need a basic knowledge of 3D computer graphics and VRML97 description language. That will allow you to create 3D models in Webots or import them from 3D modeling software.

Webots Simulation

A Webots simulation is composed of following items:

1. A Webots *world* file (.wbt) that defines one or several robots and their environment. The .wbt file does sometimes depend on external PROTO files (.proto) and textures.
2. One or several controller programs for the above robots (in C/C++/Java/Python/MATLAB).
3. An optional physics plugin that can be used to modify Webots regular physics behavior (in C/C++).

What Is a World?

A world, in Webots, is a 3D description of the properties of robots and of their environment. It contains a description of every object: position, orientation, geometry, appearance (like color or brightness), physical properties, type of object, etc. Worlds are organized as hierarchical structures where objects can contain other objects (like in VRML97). For example, a robot can contain two wheels, a distance sensor and a joint which itself contains a camera, etc. A world file doesn't contain

the controller code of the robots; it only specifies the name of the controller that is required for each robot. Worlds are saved in ".wbt" files. The ".wbt" files are stored in the "worlds" subdirectory of each Webots project.

What Is a Controller?

A controller is a computer program that controls a robot specified in a world file. Controllers can be written in any of the programming languages supported by Webots: C, C++, Java, Python or MATLAB. When a simulation starts, Webots launches the specified controllers, each as a separate process, and it associates the controller processes with the simulated robots. Note that several robots can use the same controller code, however a distinct process will be launched for each robot.

Some programming languages need to be compiled (C and C++) other languages need to be interpreted (Python and MATLAB) and some need to be both compiled and interpreted (Java). For example, C and C++ controllers are compiled to platform-dependent binary executables (for example ".exe" under Windows). Python and MATLAB controllers are interpreted by the corresponding runtime systems (which must be installed). Java controller need to be compiled to byte code (".class" files or ".jar") and then interpreted by a Java Virtual Machine.

The source files and binary files of each controller are stored together in a controller directory. A controller directory is placed in the "controllers" subdirectory of each Webots project.

What Is a Supervisor Controller?

The Supervisor controller is the controller of a Robot whose `supervisor` field is set to `TRUE`, it can execute operations that can normally only be carried out by a human operator and not by a real robot. The Supervisor controller can be written in any of the above mentioned programming languages. However, in contrast with a regular Robot controller, the Supervisor controller will have access to privileged operations. The privileged operations include simulation control, for example, moving the robots to a random position, making a video capture of the simulation, etc.

How to Install Webots on Our PC

- Download installation source at : <https://cyberbotics.com/#download>
- Choose according to your PC's OS
- Install the source and follow the instruction



Figure 2. Webots Loading Screen

HARDWARE REQUIREMENT:

- A fairly recent **PC or Mac** computer with **at least a 2 GHz dual core CPU clock speed** and **2 GB of RAM** is a **minimum requirement**. A **quad-core CPU** is however **recommended**.
- An **NVIDIA or AMD OpenGL (minimum version 3.3)** capable graphics adapter with at least **512 MB of RAM** is required. We do not recommend any other graphics adapters, including Intel graphics adapters, as they often lack a good OpenGL support which may cause 3D rendering problems and application crashes. Nevertheless, in some cases, the installation of the latest Intel graphics driver can fix such problems and let you use Webots. However, we don't provide any guarantee on this. For Linux systems, we recommend only NVIDIA graphics cards. Webots works well on all the graphics cards included in fairly recent Apple computers.

SOFTWARE REQUIREMENT:

- Windows: Webots runs on Windows 10 and Windows 8.1 (64-bit versions only).
- Mac: macOS 10.15 "Catalina" and 10.14 "Mojave".
- Linux: Latest Ubuntu Long Term Support (LTS) release, currently version 20.04, RedHat, Mandrake, Debian, Gentoo, Arch, SuSE, and Slackware. We recommend using a recent version of Linux. Webots is provided for Linux 64 (x86-64) systems. Webots doesn't run on Ubuntu versions earlier than 16.04, but only packages for Ubuntu 18.04 and 20.04 are provided.






OS	File(s)
 Windows 10 64-bit	webots-R2021b_setup.exe
 Linux 64-bit	webots_2021b_amd64.deb (Ubuntu 18.04 & 20.04) webots-R2021b-x86-64.tar.bz2 (Ubuntu 20.04) webots-R2021b-x86-64_ubuntu-18.04.tar.bz2 (Ubuntu 18.04)  latest/stable R2021b (Linux snap package)  dockerhub (Docker Image)
 macOS	webots-R2021b.dmg

Figure 3. Webots OS Requirement.

GCTronic' e-puck

E-puck is a miniature mobile robot originally developed at EPFL for teaching purposes by the designers of the successful Khepera robot. The hardware and software of e-puck is fully open source, providing low level access to every electronic device and offering unlimited extension possibilities.

The model includes support for the differential wheel motors (encoders are also simulated, as position sensors), the infra-red sensors for proximity and light measurements, the Accelerometer, the Gyro, the Camera, the 8 surrounding LEDs, the body and front LEDs, bluetooth communication (modeled using Emitter / Receiver devices) and ground sensors extension. The other e-puck devices are not yet simulated in the current model.



Figure 4. GCTronic e-puck.

E-puck is equipped with a large number of devices, as summarized in this table.

Table 1 E-puck features

Feature	Description
Size	7.4 cm in diameter, 4.5 cm high
Weight	150 g
Battery	about 3 hours with the provided 5Wh LiION rechargeable battery
Processor	Microchip dsPIC 30F6014A @ 60MHz (about 15 MIPS)
Motors	2 stepper motors with 20 steps per revolution and a 50:1 reduction gear
IR sensors	8 infra-red sensors measuring ambient light and proximity of obstacles in a 4 cm range
Camera	color camera with a maximum resolution of 640x480 (typical use: 52x39 or 640x1)
Microphones	3 omni-directional microphones for sound localization
Accelerometer	3D accelerometer along the X, Y and Z axes
Gyroscope	3D gyroscope along the X, Y and Z axes
LEDs	8 red LEDs on the ring and one green LED on the body
Speaker	on-board speaker capable of playing WAV or tone sounds
Switch	16 position rotating switch
Bluetooth	Bluetooth for robot-computer and robot-robot wireless communication
Remote Control	infra-red LED for receiving standard remote control commands
Expansion bus	expansion bus to add new possibilities to your robot
Programming	C programming with the GNU GCC compiler system
Simulation	Webots facilitates the programming of e-puck with a powerful simulation, remote control and cross-compilation system

E-puck Model

Table 2 E-puck characteristics

Characteristics	Values
Diameter	71 mm
Height	50 mm
Wheel radius	20.5 mm
Axle Length	52 mm
Weight	0.16 kg
Max. forward/backward speed	0.25 m/s
Max. rotation speed	6.28 rad/s

The standard model of the e-puck is provided in the "E-puck.proto" PROTO file which is located in the "WEBOTS_HOME/projects/robots/gctronic/e-puck/protos" directory of the Webots distribution (see also "E-puckDistanceSensor.proto" PROTO file and "E-puckGroundSensors.proto" PROTO file); you will find complete specifications in it. The two PROTO fields `groundSensorSlot` and `turretSlot` have been included in the simulation model in order to support extension modules. In particular, the ground sensors module extension of the real e-puck robot is modelled using the "E-puckGroundSensors.proto" PROTO in Webots to provide 3 optional infra-red sensors pointing to the ground in front of the robot. The names of the simulated devices which are to be used as an argument of the `wb_robot_get_device` function are presented in the table below.

Table 3 Devices names

Device	Name
Motors	'left wheel motor' and 'right wheel motor'
Position sensors	'left wheel sensor' and 'right wheel sensor'
Proximity sensors	'ps0' to 'ps7'

Device	Name
Light sensors	'ls0' to 'ls7'
LEDs	'led0' to 'led7' (e-puck ring), 'led8' (body) and 'led9' (front)
Camera	'camera'
Accelerometer	'accelerometer'
Gyro	'gyro'
Ground sensors (extension)	'gs0', 'gs1' and 'gs2'
Speaker	'speaker'

The `wb_motor_set_velocity` and `wb_position_sensor_get_value` functions allow you to set the speed of the robot and to use its encoders.

Table 4 Devices orientations

Device	x (m)	y (m)	z (m)	Orientation (rad)
ps0	0.010	0.033	-0.030	1.27
ps1	0.025	0.033	-0.022	0.77
ps2	0.031	0.033	0.00	0.00
ps3	0.015	0.033	0.030	5.21
ps4	-0.015	0.033	0.030	4.21
ps5	-0.031	0.033	0.00	3.14159
ps6	-0.025	0.033	-0.022	2.37
ps7	-0.010	0.033	-0.030	1.87
camera	0.000	0.028	-0.030	4.71239

The forward direction of the e-puck is given by the negative z-axis of the world coordinates. This is also the direction in which the camera eye is looking. The direction vector of the camera is pointing in the opposite direction, namely the direction of the positive z-axis. The axle's direction is given by the positive x-axis. Proximity sensors, light sensors and LEDs are numbered clockwise. Their location and orientation are shown in this figure. The last column of the latter lists the angles between the negative x-axis and the direction

of the devices, the plane zOx being oriented counter-clockwise. Note that the proximity sensors and the light sensors are actually the same devices of the real robot used in a different mode, so their direction coincides. Proximity sensor responses are simulated in accordance to the lookup table in this figure; this table is the outcome of calibrations performed on the real robot.

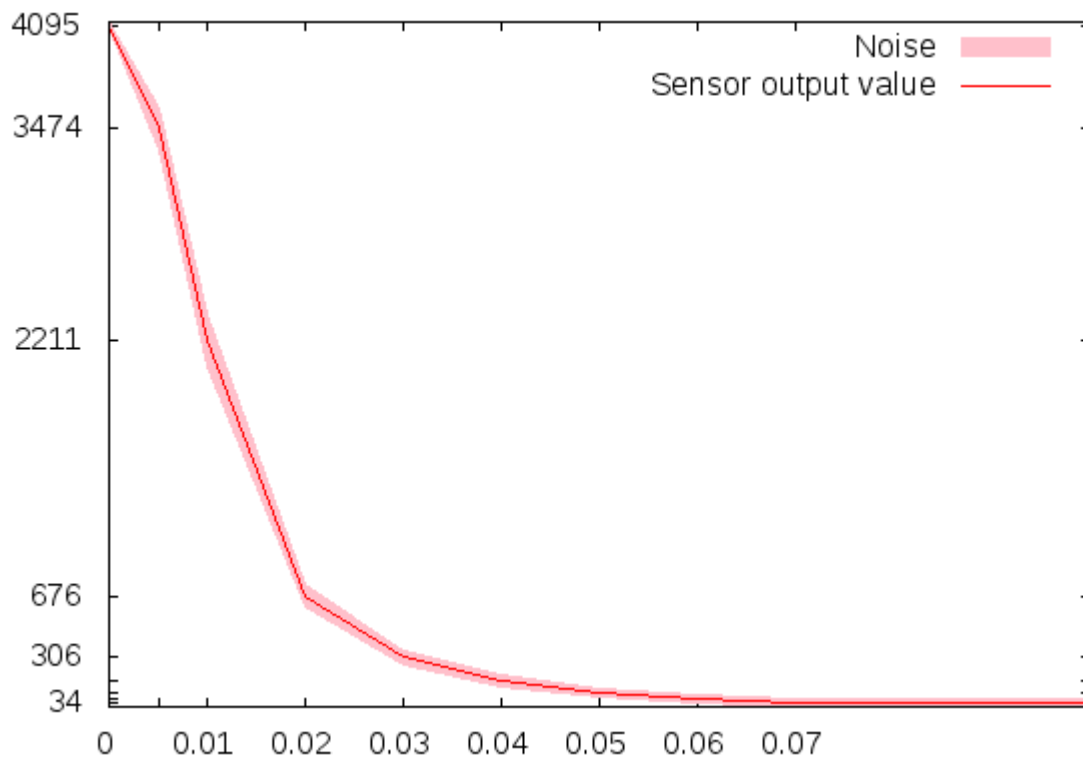


Figure 5 Proximity sensor response against distance

The resolution of the camera was limited to 52x39 pixels, as this is the maximum rectangular image with a 4:3 ratio which can be obtained from the remote control interface with the real robot.

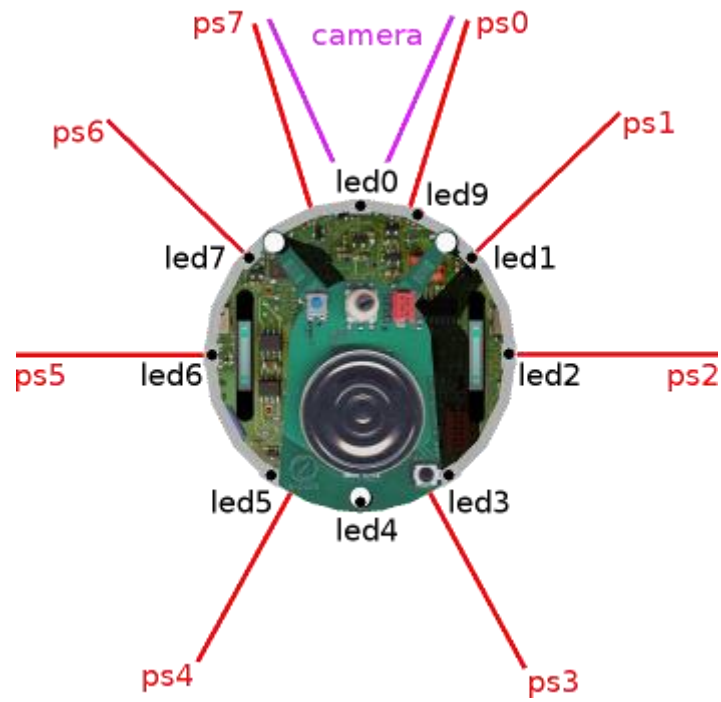


Figure 6 Sensors, LEDs and camera

SECTION 2 : LINE FOLLOWER ROBOT IN WEBOTS

Purposes:

After following this workshop (exercise), you are expected to be able:

- 2.1. Understand the working of the line follower
- 2.2. To show the work of line follower on the arena that has been made

Line Follower

Line follower Robot is a machine which follows a line, either a black line or white line. Basically there are two types of line follower robots: one is black line follower which follows black line and second is white line follower which follows white line. Line follower actually senses the line and run over it.

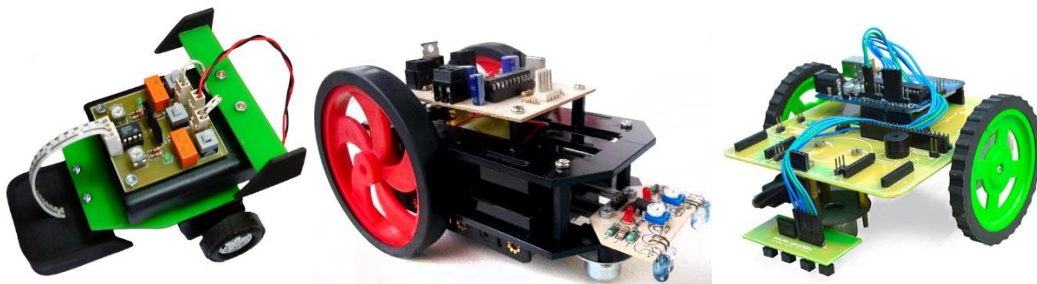


Figure 7. Line follower Robots

How Line Follower Robot Works

Concept of working of line follower is related to light. E-puck robot used IR Transmitters and IR receivers as a sensors. They are used for sending and receiving light. IR transmits infrared lights. When infrared rays falls on white surface, it's reflected back and caught by photodiodes which generates some voltage changes. When IR light falls on a black surface, light is absorb by the black surface and no rays are reflected back, thus photo diode does not receive any light or rays.

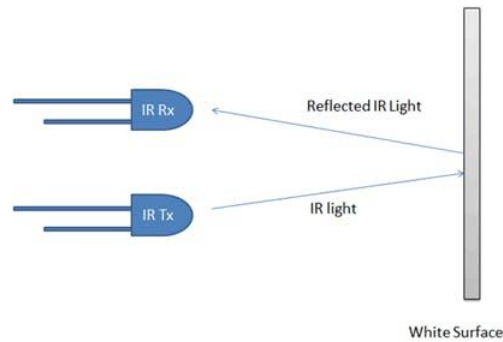


Figure 8. IR Sensor Work Process.

In this project we are using two IR sensor modules namely **IR0** for left sensor and **IR1** for right sensor. When both left and right sensor senses white then robot move forward. If left sensor comes on black line then robot turn left side. If right sensor sense black line, then robot turn right side until both sensor comes at white surface. When white surface comes robot starts moving on forward again.

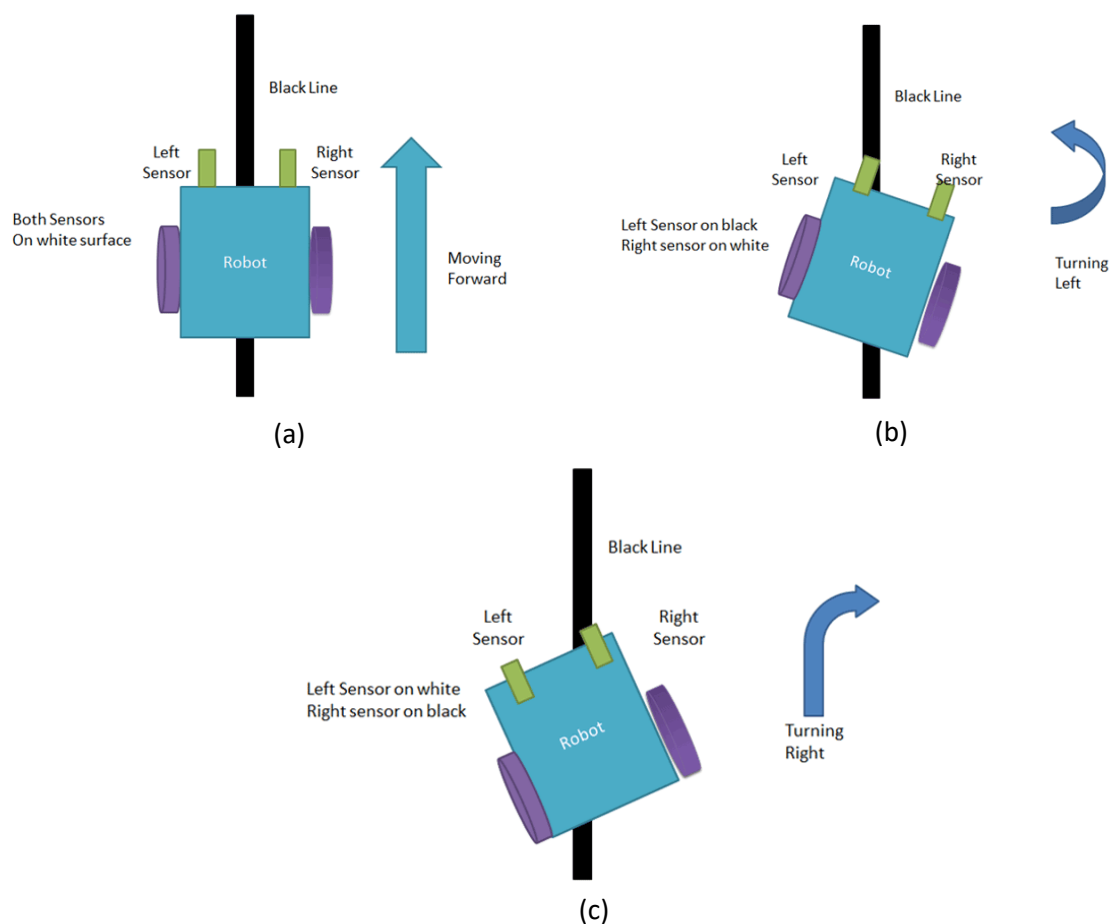


Figure 9. Line follower robot move (a) forward (b) turning left (c) turning right.

Create Webots New Project Directory

Now we will start the line follower robot simulation project. First of all, create a new project from the **Wizards** menu by selecting the **New Project Directory...** menu item and follow the instructions:

1. Name the project directory **LineFollowerRobot** instead of the proposed **my_project**.
2. Name the world file **NewArena.wbt** instead of the proposed **empty.wbt**.
3. Click all the tick boxes, including the "Add a rectangle arena" which is not ticked by default.

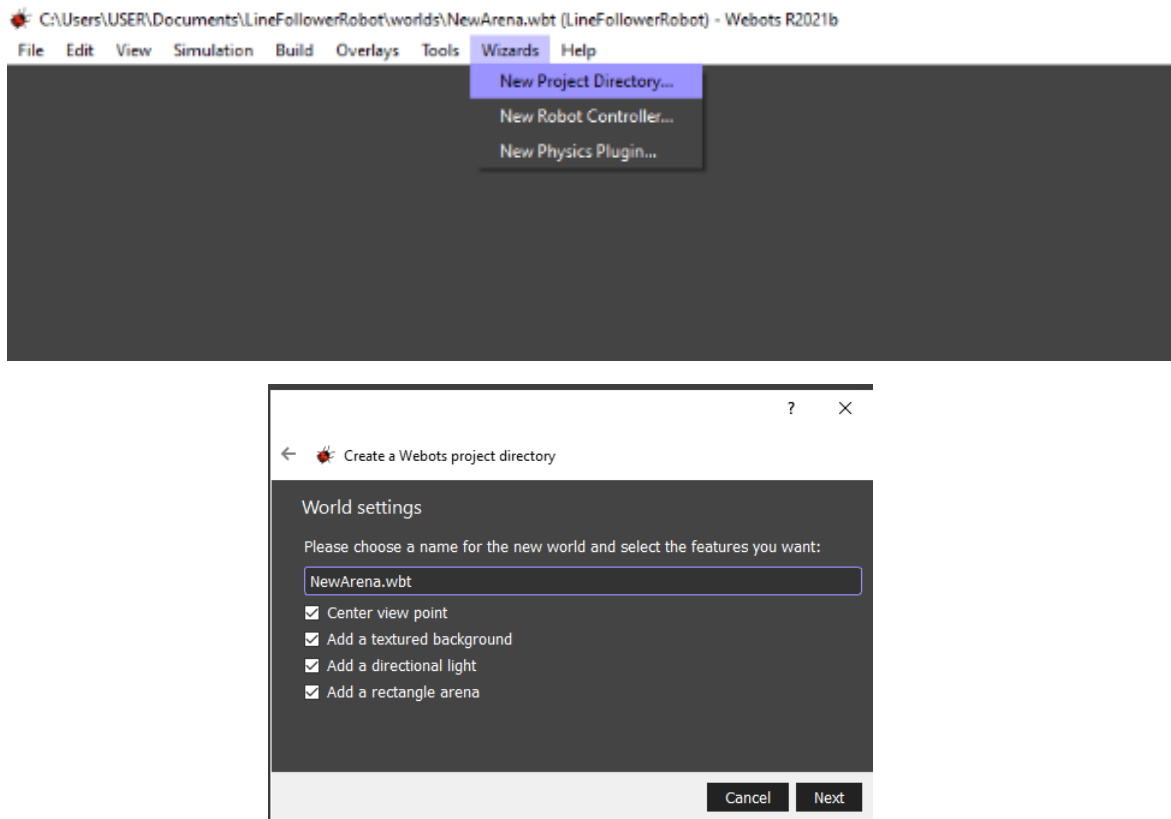


Figure 10. Create Webots New Project Directory.

Webots displays a list of directories and files it just created. This corresponds to the standard file hierarchy of a Webots project. Click on the **Finish** (Windows, Linux) or **Done** (macOS) button to close this window.

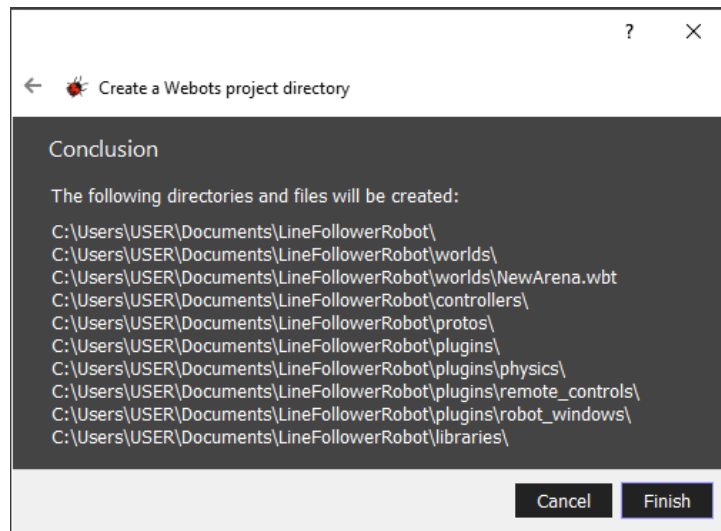


Figure 11. Webots displays a list of directories and files it just created.

The 3D view should display a square arena with a checkered floor. You can move the viewpoint in the 3D view using the mouse: left button, right button and the wheel.

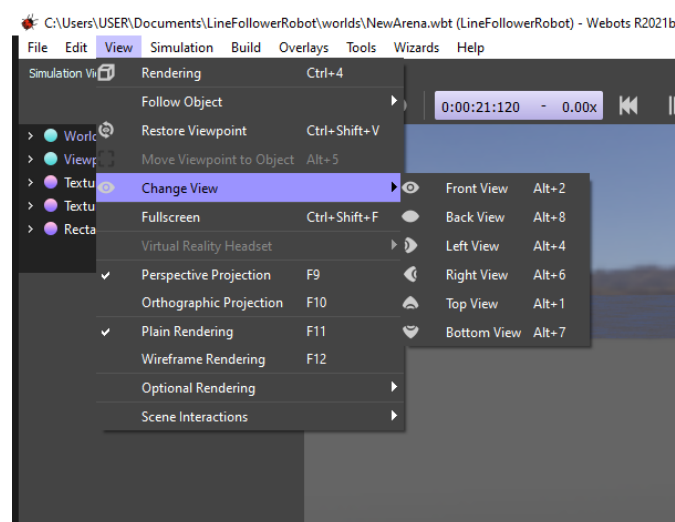


Figure 12. Change view menu.

Webots nodes stored in world files are organized in a tree structure called the **scene tree**. The scene tree can be viewed in two sub-windows of the main window: the **3D view** (at the center of the main window) is the 3D representation of the scene tree and the **scene tree view** (on the left) is the hierarchical representation of the scene tree. The scene tree view is where the nodes and fields can be modified. It should currently list the following nodes:

- **WorldInfo:** contains global parameters of the simulation.
- **Viewpoint:** defines the main viewpoint camera parameters.

- **TexturedBackground**: defines the background of the scene (you should see mountains far away if you rotate a little bit the viewpoint)
- **TexturedBackgroundLight**: defines the light associated with the above background.
- **RectangleArena**: define the only object you see so far in this scene.

Each node has some customizable properties called **Fields**. Let's modify these fields to change the rectangle arena:

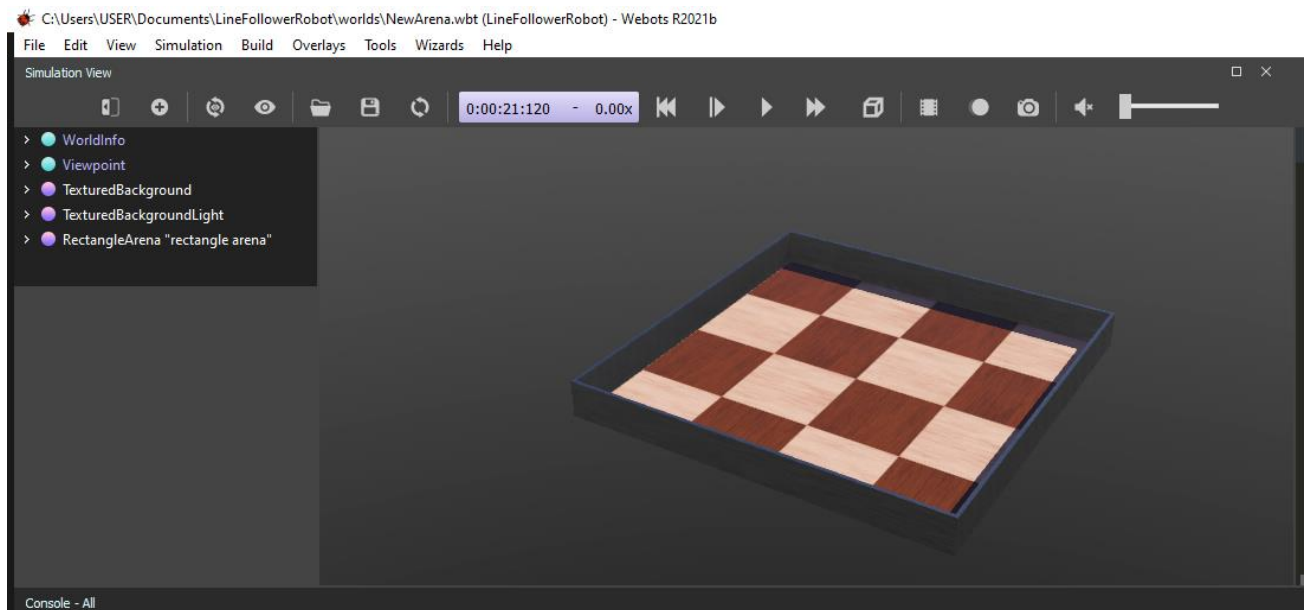


Figure 13. Rectangle arena with checkered tiles floor.

The default size of arena is 1 x 1, let's modify these size to change the rectangle arena:

- Double-click on the `RectangleArena` node in the scene tree. This should open the node and display its fields.

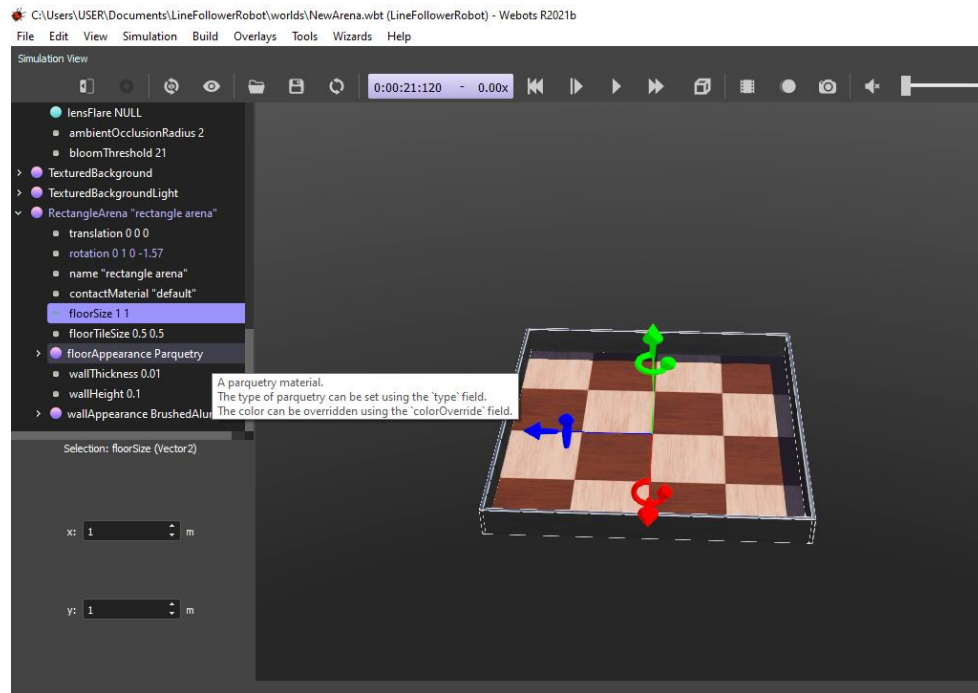


Figure 14. Arena size.

- Select the `floorSize` field and set its value to `2 2` instead of `1 1`. You should see the effect immediately in the 3D view.

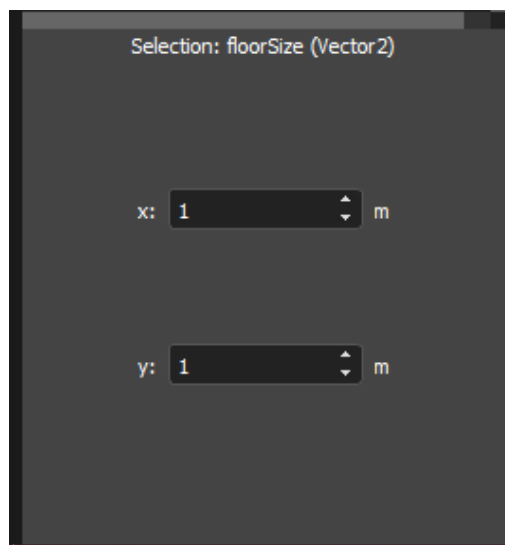



Figure 15. Floor size.

Add an e-puck Robot

Now, we are going to add an e-puck model to the world. Make sure that the simulation is paused and that the virtual time elapsed is 0. If this is not the case, reset the simulation with the **Reset** button .

When a Webots world is modified with the intention of being saved, it is fundamental that the simulation is first paused and reloaded to its initial state, i.e. the virtual time counter

on the main toolbar should show **0:00:00:000**. Otherwise at each save, the position of each 3D object can accumulate errors. Therefore, any modification of the world should be performed in that order: **pause, reset, modify and save the simulation**.

We don't need to create the e-puck robot from scratch, we will just have to import a E-puck node. This node is actually a PROTO node, like the RectangleArena or the WoodenBox we introduced before. Prototyping allows you to create custom objects and to reuse them.

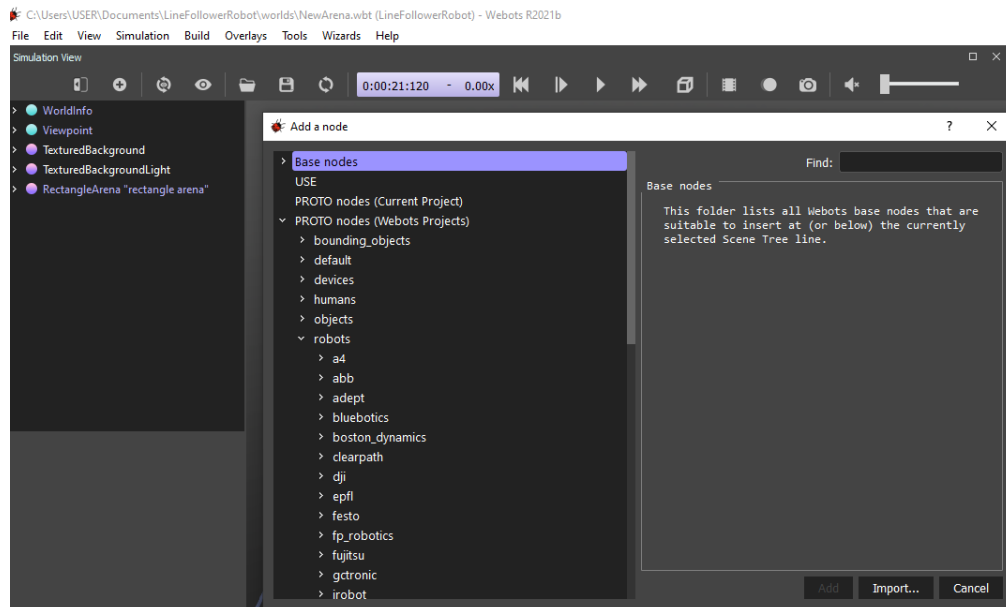


Figure 16. Robot's list.

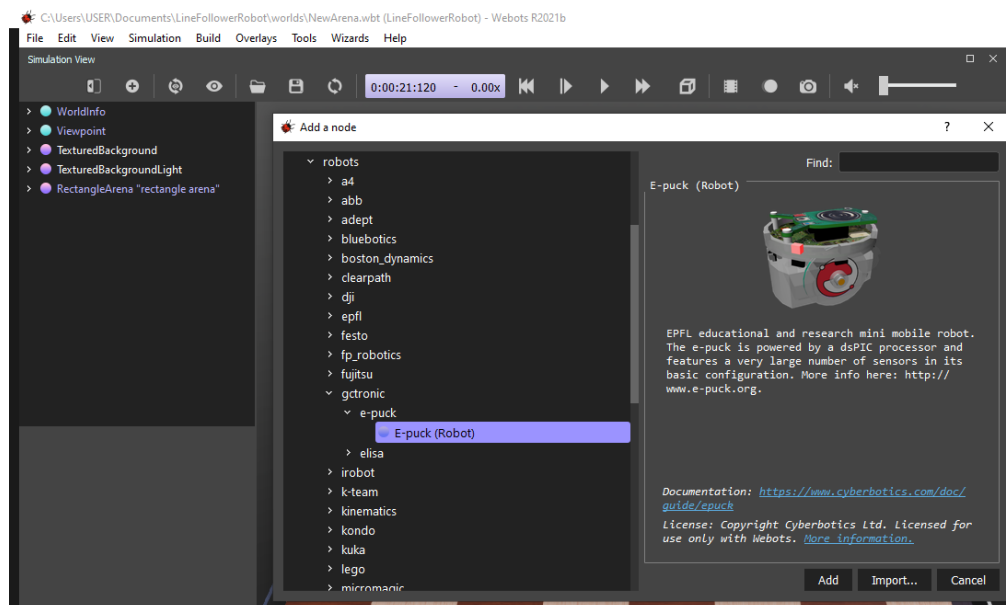
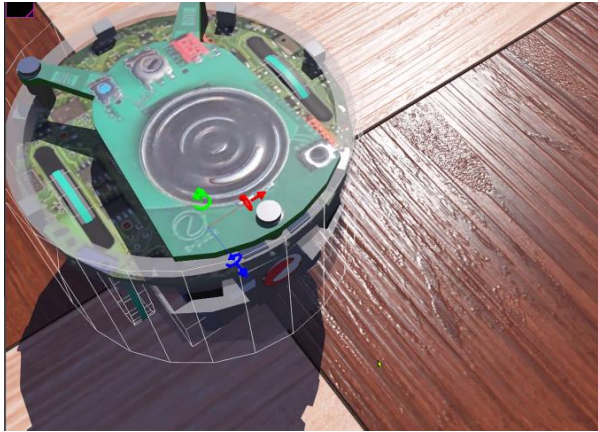
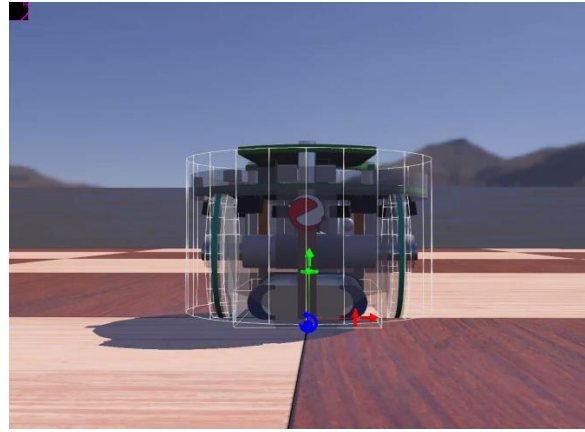


Figure 17. GCTronic E-puck description.

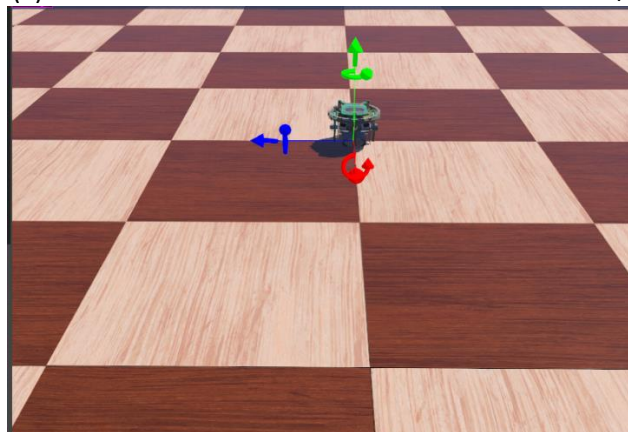
By default, the robot should move, blink LEDs and avoid obstacles. That's because it has a default controller with that behavior.



(a)



(b)



(c)

Figure 18. Robot is placed on arena.

You may have noticed a small black window appearing in the upper-left corner of the 3D view. It shows the image taken by the Camera of the e-puck robot. This image will remain black until the camera is explicitly enabled by the controller of the robot. This small image window can be moved around by dragging it.

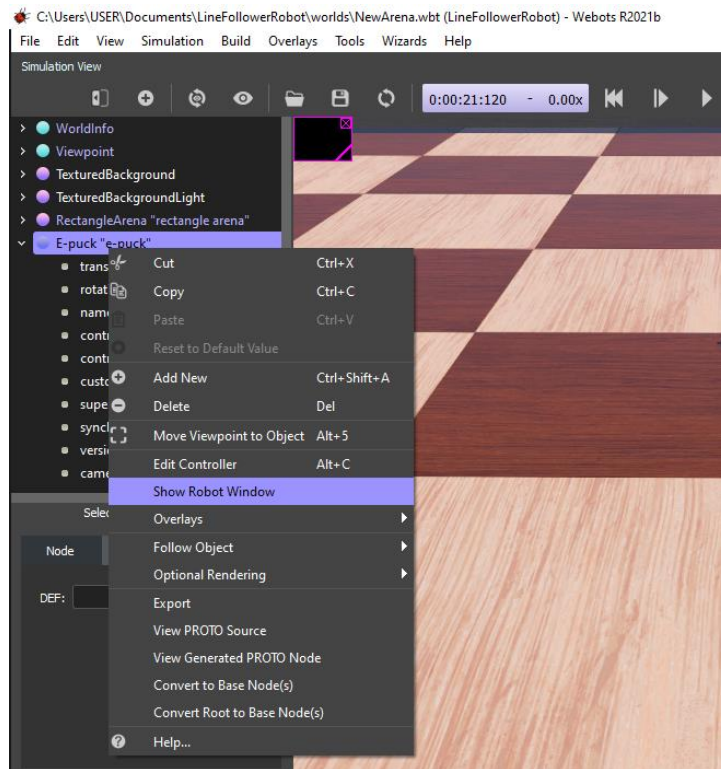


Figure 19. Show robot window's menu.

It can also be resized by dragging the bottom-right corner. Finally, it can be closed by clicking the "x" in the top-right corner. You can make it visible again from the **Overlays** menu, by selecting it in the **Camera Devices** submenu. Because we won't need it, you can actually close it.

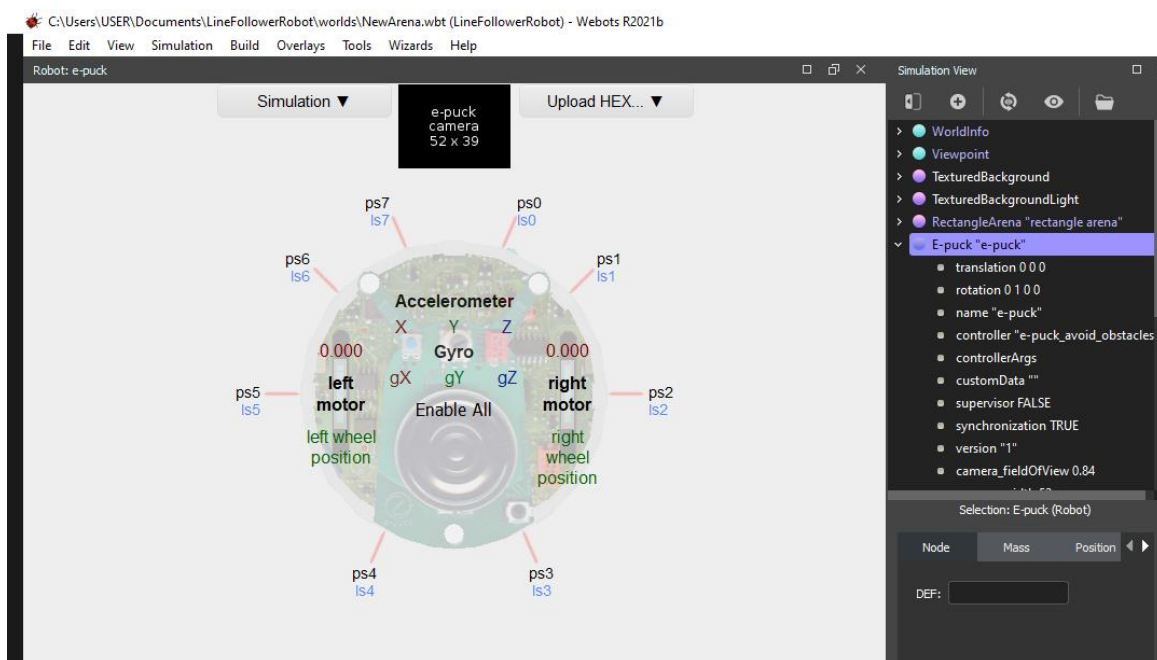


Figure 20. E-puck robot window detail.

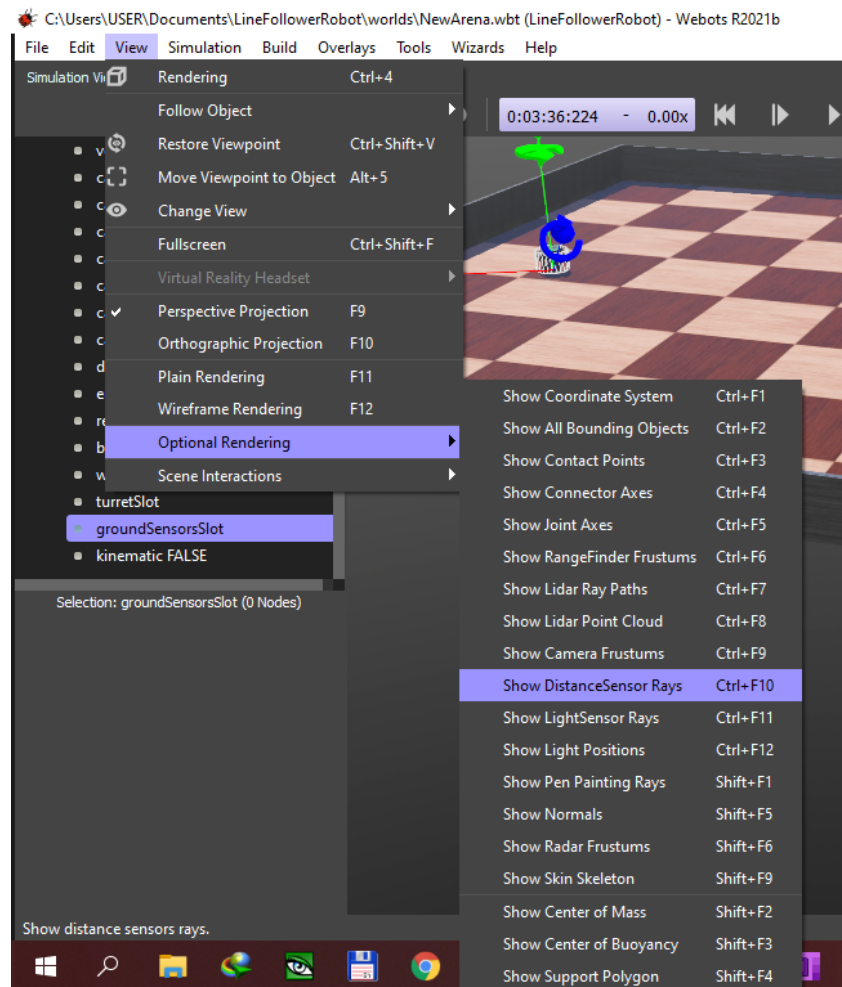


Figure 21. Show DistanceSensor Rays.

The **DistanceSensor** node can be used to model a generic sensor, an infra-red sensor, a sonar sensor, or a laser range-finder. This device simulation is performed by detecting the collisions between one or several sensor rays and objects in the environment. In case of generic, sonar and laser type the collision occurs with the bounding objects of Solid nodes, whereas infra-red rays collision detection uses the Solid nodes themselves.

The rays of the **DistanceSensor** nodes can be displayed by checking the menu **View / Optional Rendering / Show Distance Sensor Rays**. The red/green transition on the rays indicates the points of intersection with the bounding objects.

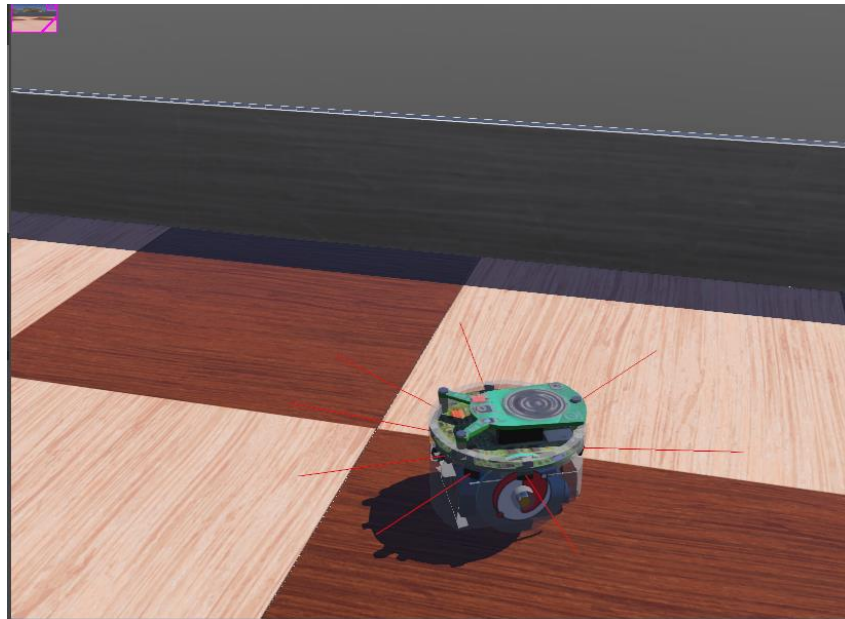
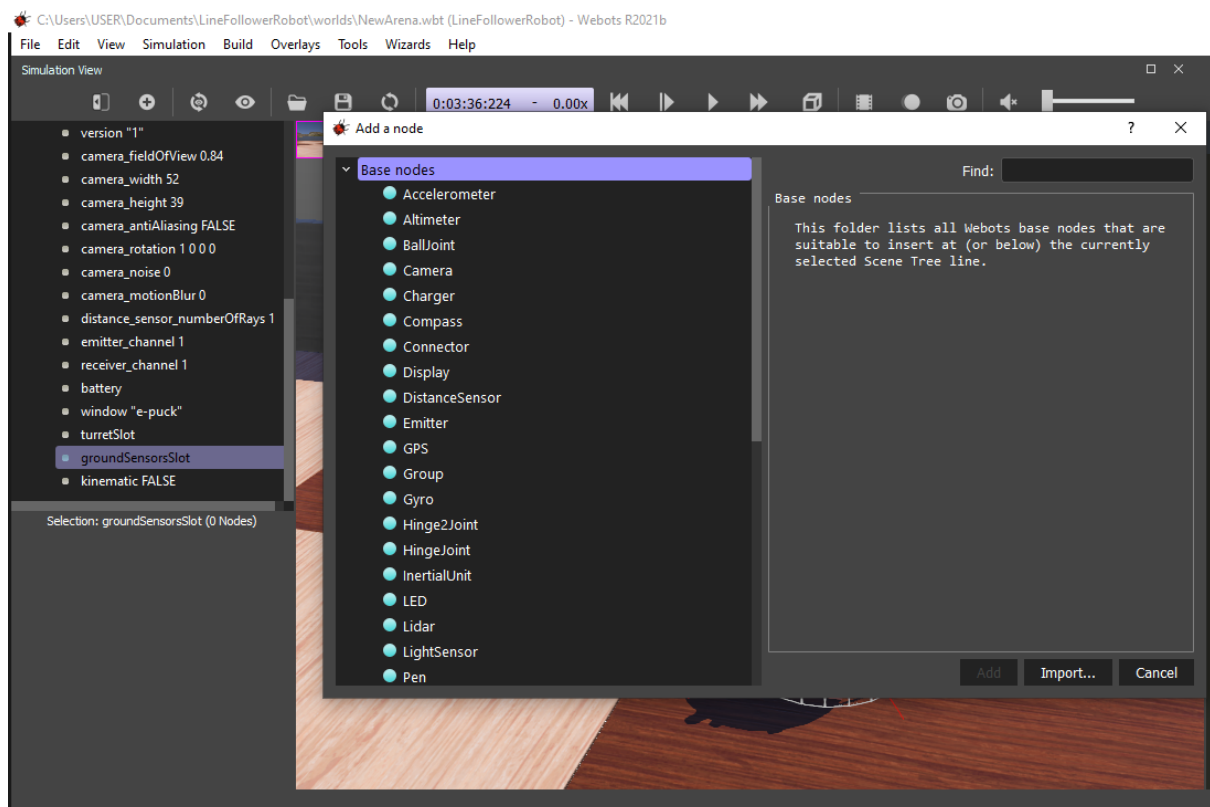
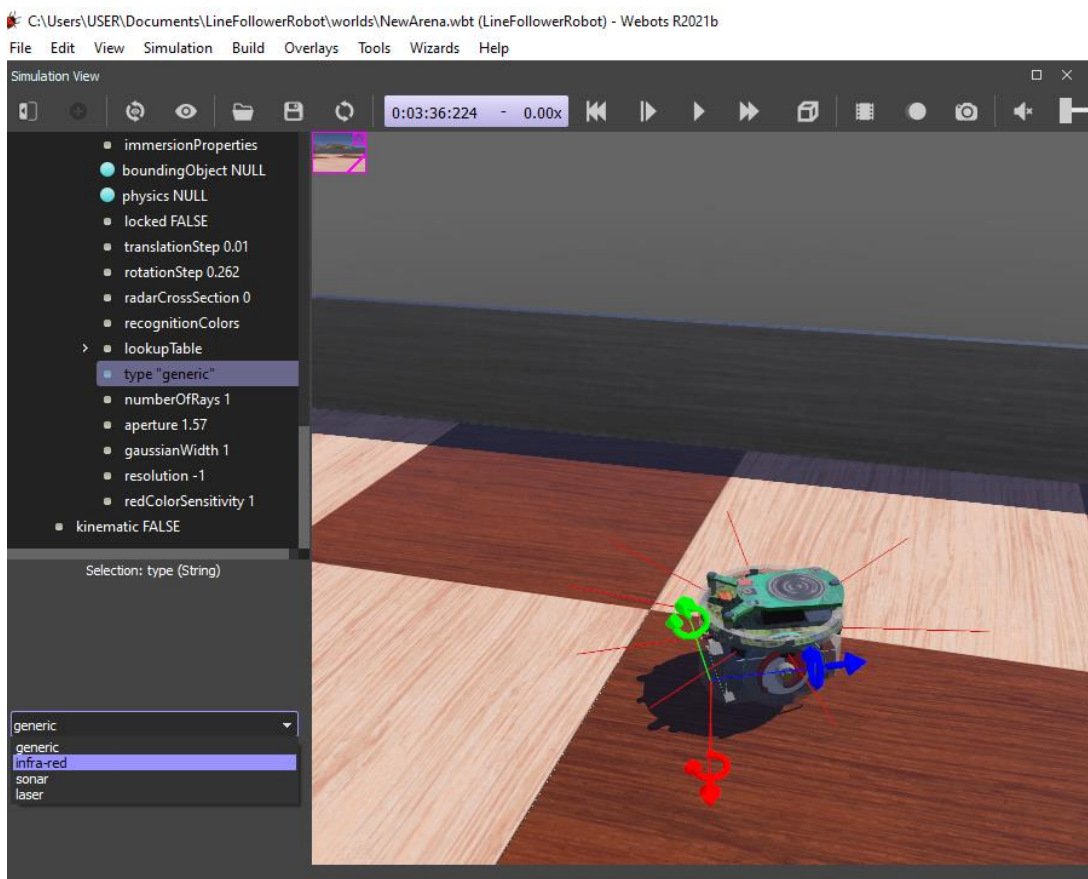
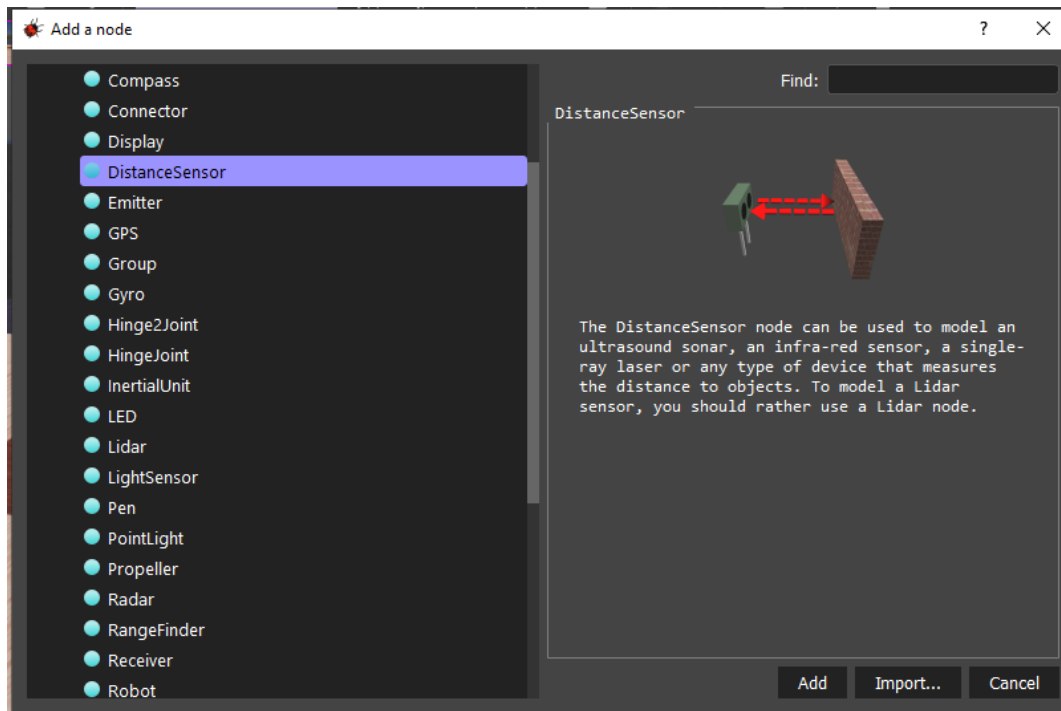
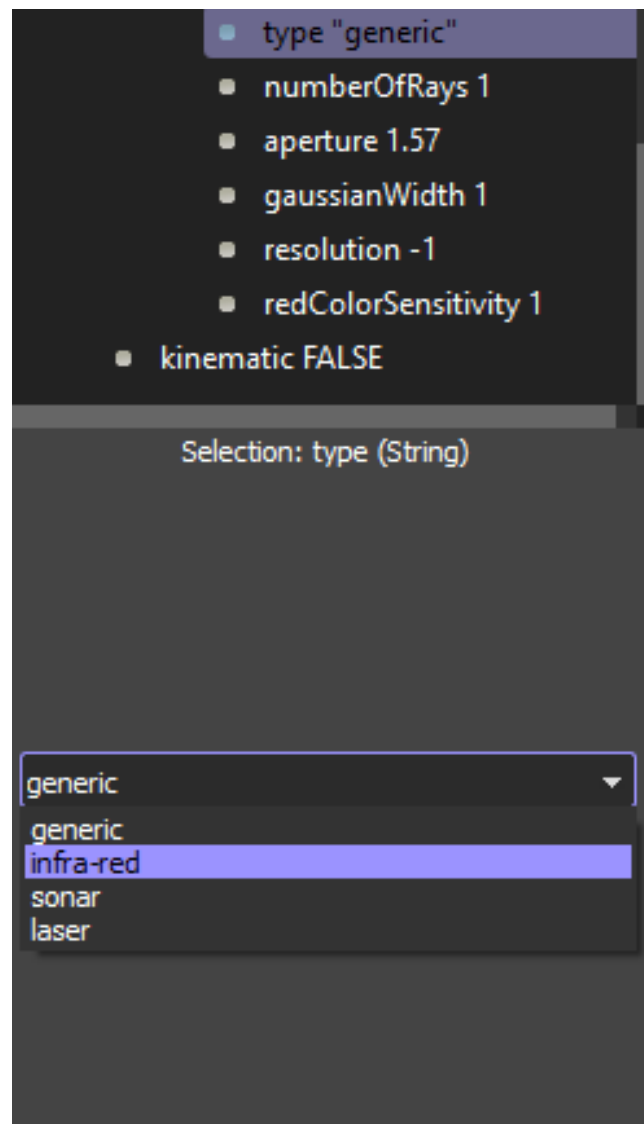


Figure 22. DistanceRays has been activated.

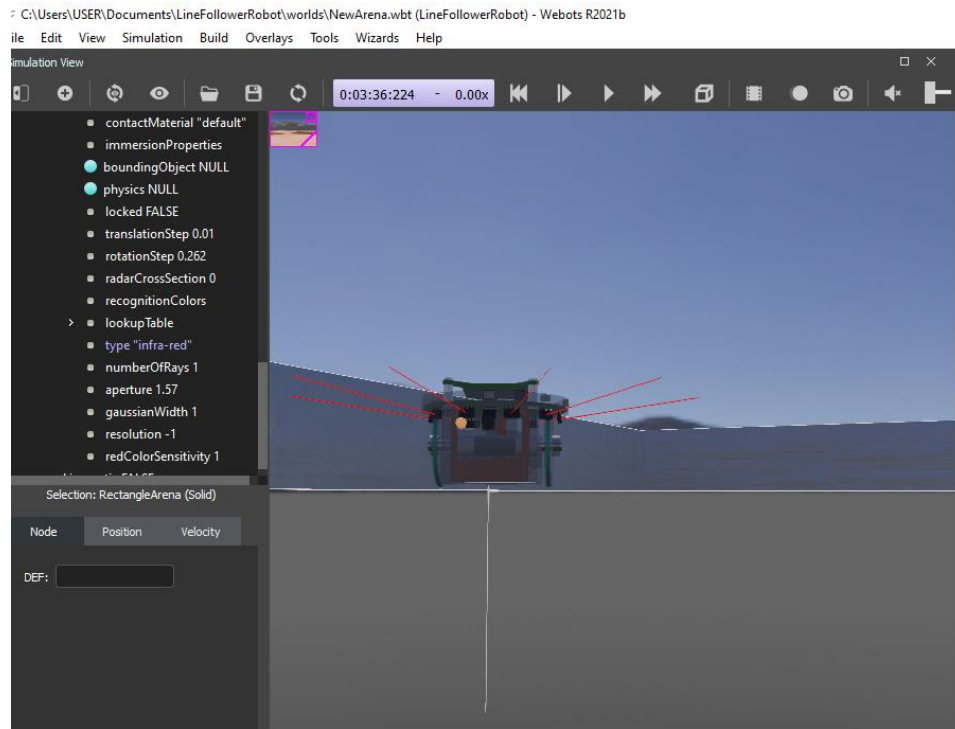
To activate the line following capability of the robot, the infra red sensor is required.



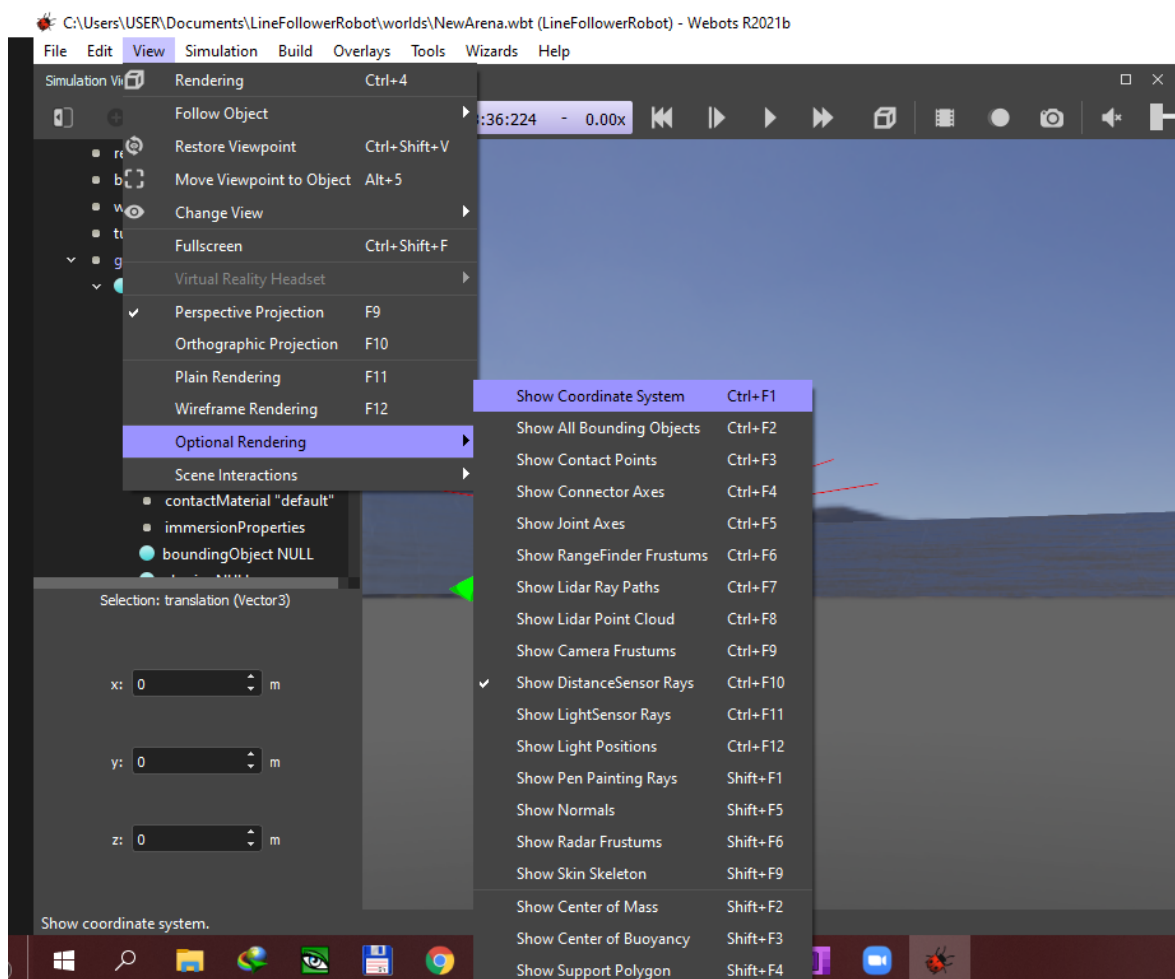




If the procedure is complete, the infra red sensor's ray will be appeared.

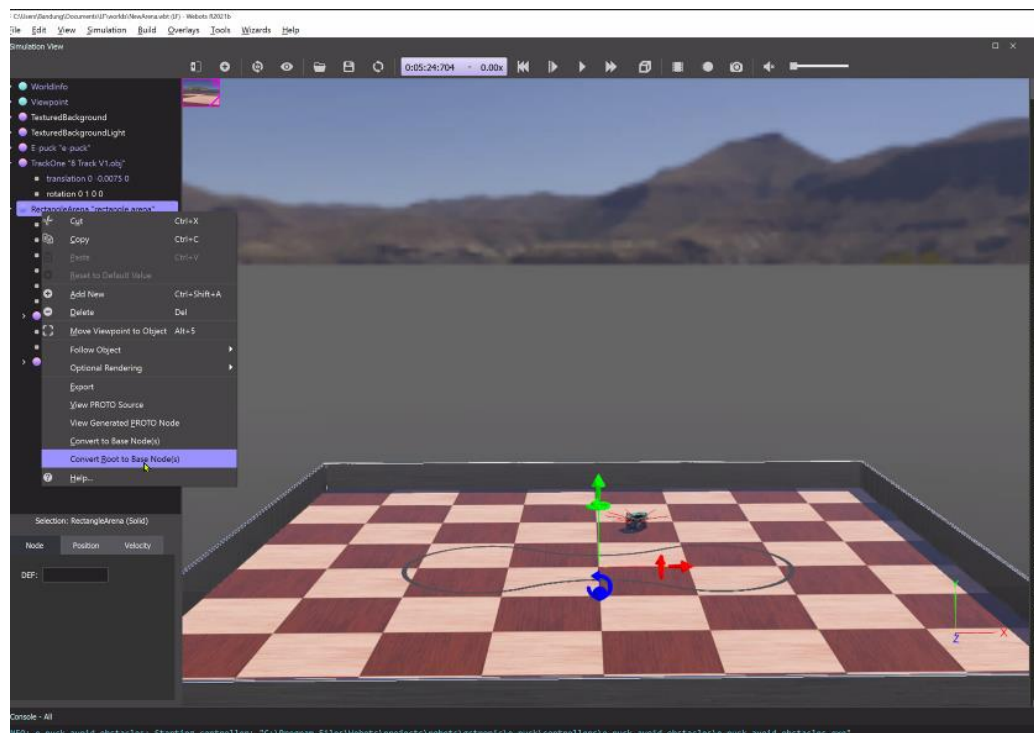
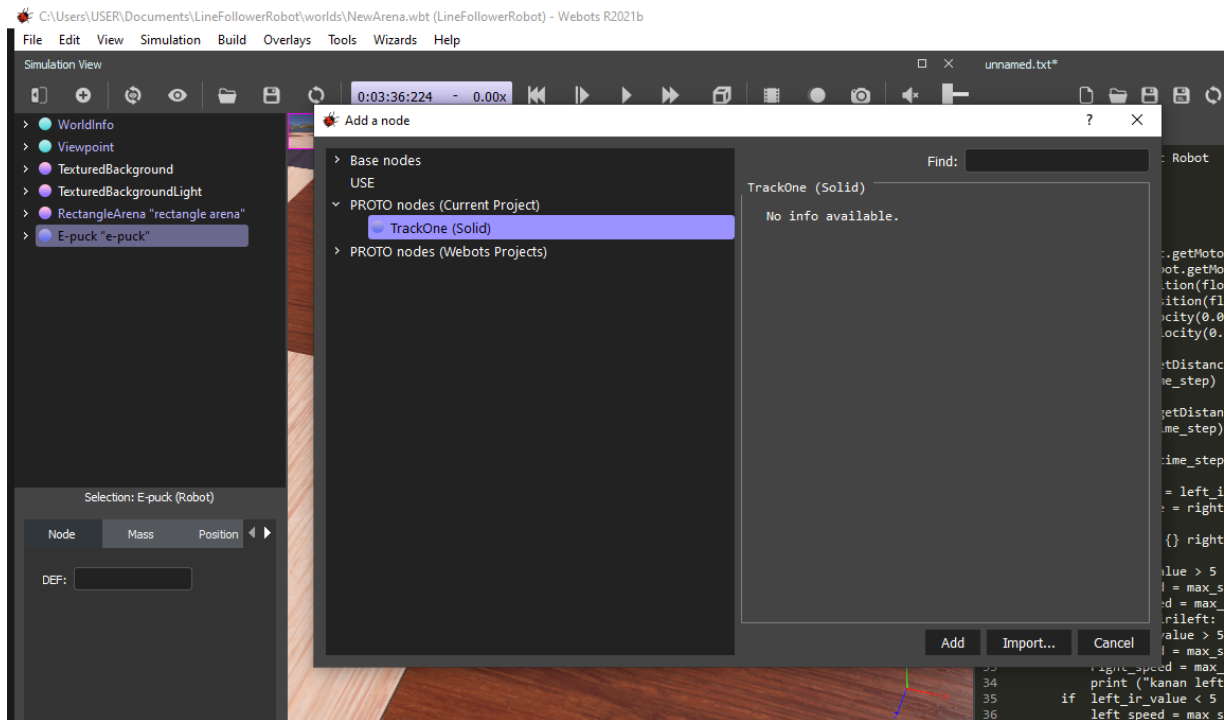


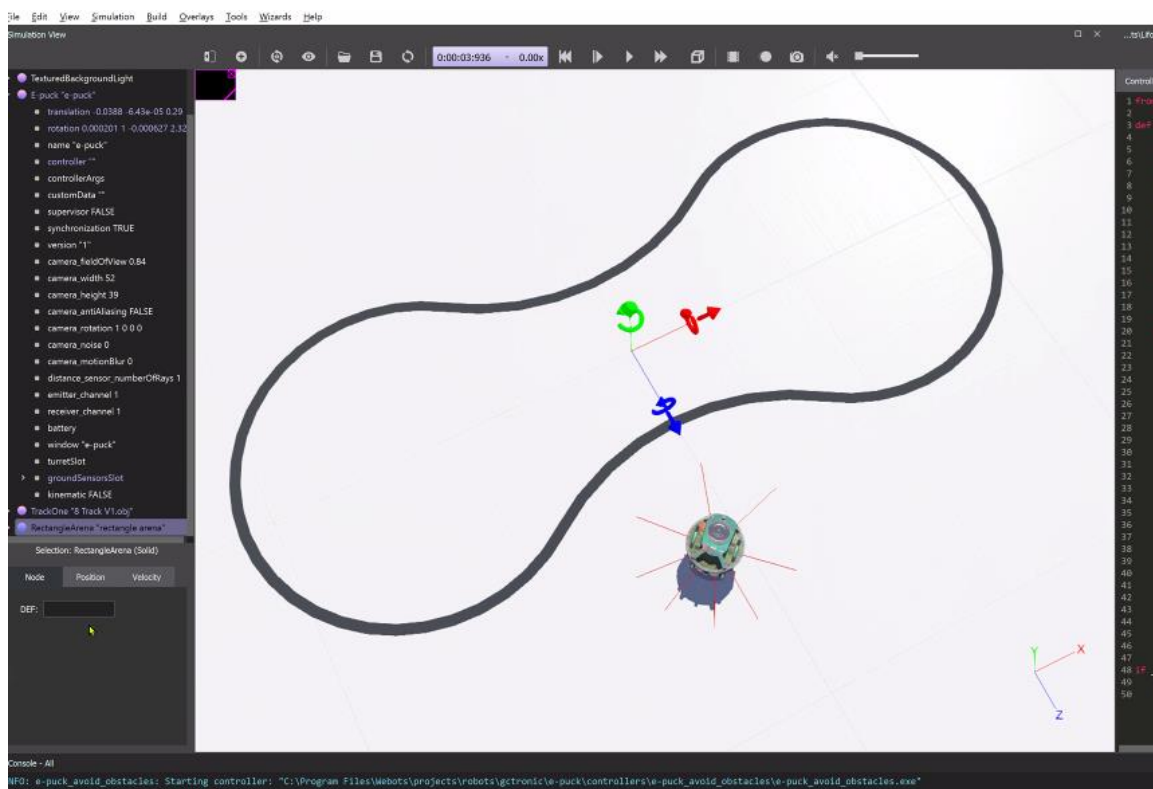
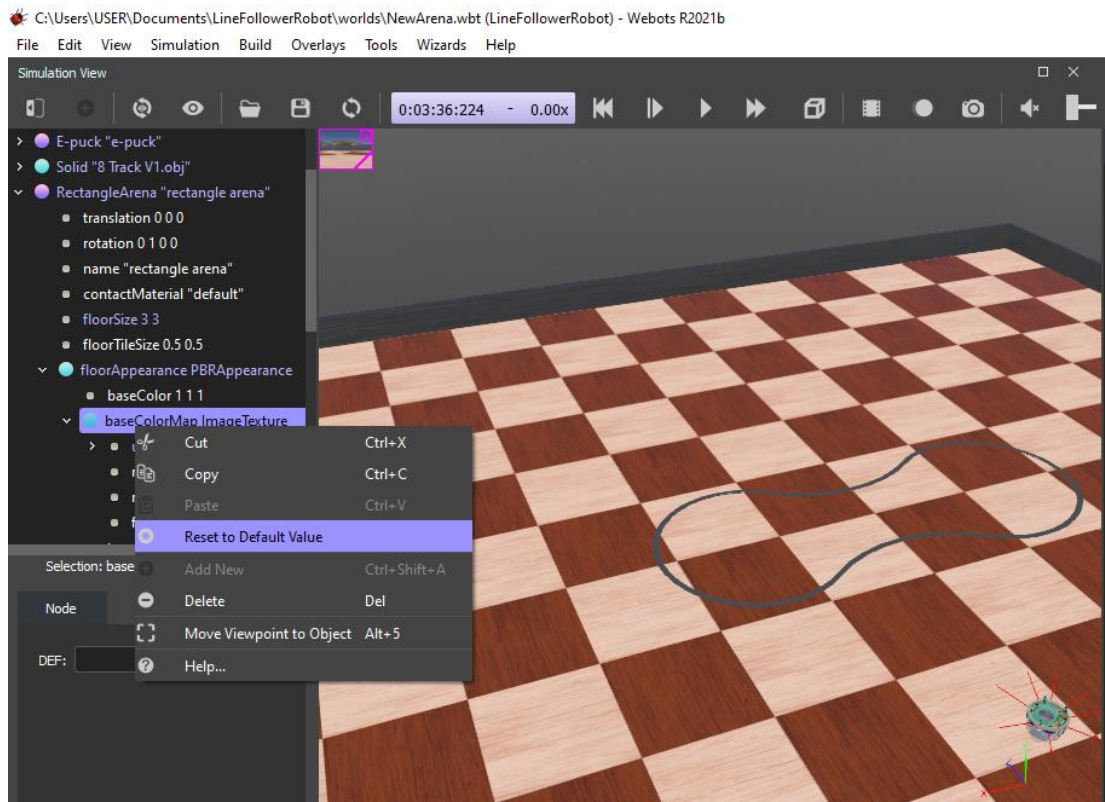
To Adjust Position.



Creating Track For Line Follower Robot

By using trackone proto file, it can be used as an arena for testing the line follower robot.

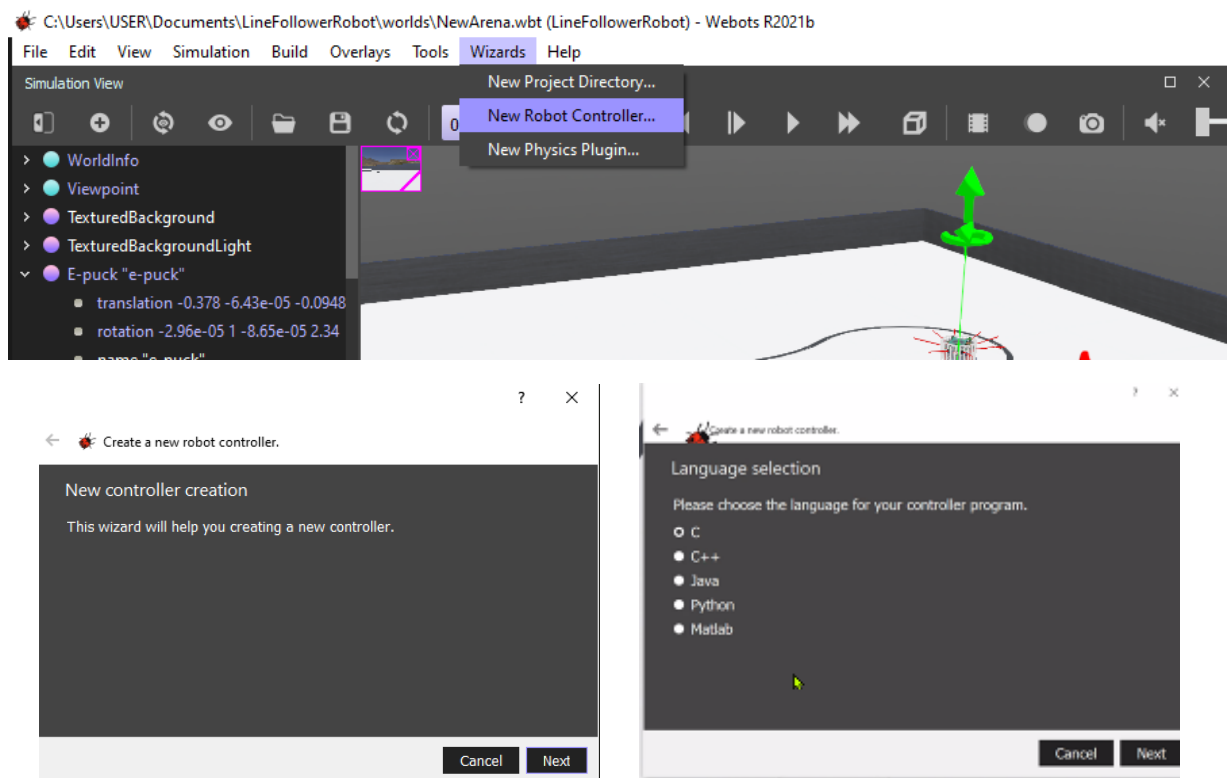




Creating Controller for Robot

Now we start to tackle the topics related to programming robot controllers. We will design a simple controller that avoids the obstacles created in the previous tutorials. This tutorial will introduce you

to the basics of robot programming in Webots. At the end of this chapter, you should understand what is the link between the scene tree nodes and the controller API, how the robot controller has to be initialized and cleaned up, how to initialize the robot devices, how to get the sensor values, how to command the actuators, and how to program a simple feedback loop. This tutorial only addresses the correct usage of Webots functions. The study of robotics algorithms is beyond the goals of this tutorial and so it won't be addressed here.



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. In this tutorial, we are going to use Python as a reference language.

We will now program a simple controller that will just make the robot move with a simple line detection behavior. You will program the robot to go forwards until a line is detected by the infra-red sensors, and then follow that line. The complete code of this controller is given below. Don't forget to save the file first before running the simulation.

```
# You may need to import some classes of the controller module. Ex:
# from controller import Robot, Motor, DistanceSensor

from controller import Robot

def run_robot(robot):
    # get the time step of the current world.
    time_step = 1
    max_speed = 3.28
    # You should insert a getDevice-like function in order to get the
    # instance of a device of the robot. Something like:
    # motor = robot.getDevice('motorname')
    # ds = robot.getDevice('dsname')
    # ds.enable(timestep)

    left_motor = robot.getMotor('left wheel motor')
    right_motor = robot.getMotor('right wheel motor')
    left_motor.setPosition(float('inf'))
    right_motor.setPosition(float('inf'))
    left_motor.setVelocity(0.0)
    right_motor.setVelocity(0.0)

    left_ir = robot.getDistanceSensor('ir1')
    left_ir.enable(time_step)

    right_ir = robot.getDistanceSensor('ir0')
    right_ir.enable(time_step)

    # Main loop:
    # - perform simulation steps until Webots is stopping the controller
    while robot.step(time_step) != -1:
        # Read the sensors:
        # Enter here functions to read sensor data, like:
        # val = ds.getValue()
        # Process sensor data here.
        # Enter here functions to send actuator commands, like:
        # motor.setPosition(10.0)
        left_ir_value = left_ir.getValue()
        right_ir_value = right_ir.getValue()

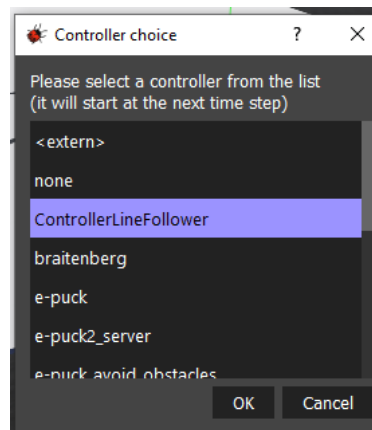
        print ("left: {} right: {}".format(left_ir_value, right_ir_value))






        if left_ir_value > 5 and right_ir_value < 5:
            left_speed = max_speed * -0.1
            right_speed = max_speed
            print ("kirileft: {} right: {}".format(left_speed, right_speed))
        if right_ir_value > 5 and left_ir_value < 5:
            left_speed = max_speed
            right_speed = max_speed * -0.3
            print ("kanan left: {} right: {}".format(left_speed, right_speed))
        if left_ir_value < 5 and right_ir_value < 5:
            left_speed = max_speed * 1
            right_speed = max_speed * 1
            print ("lurus")
            print ("lurus left: {} right: {}".format(left_speed, right_speed))

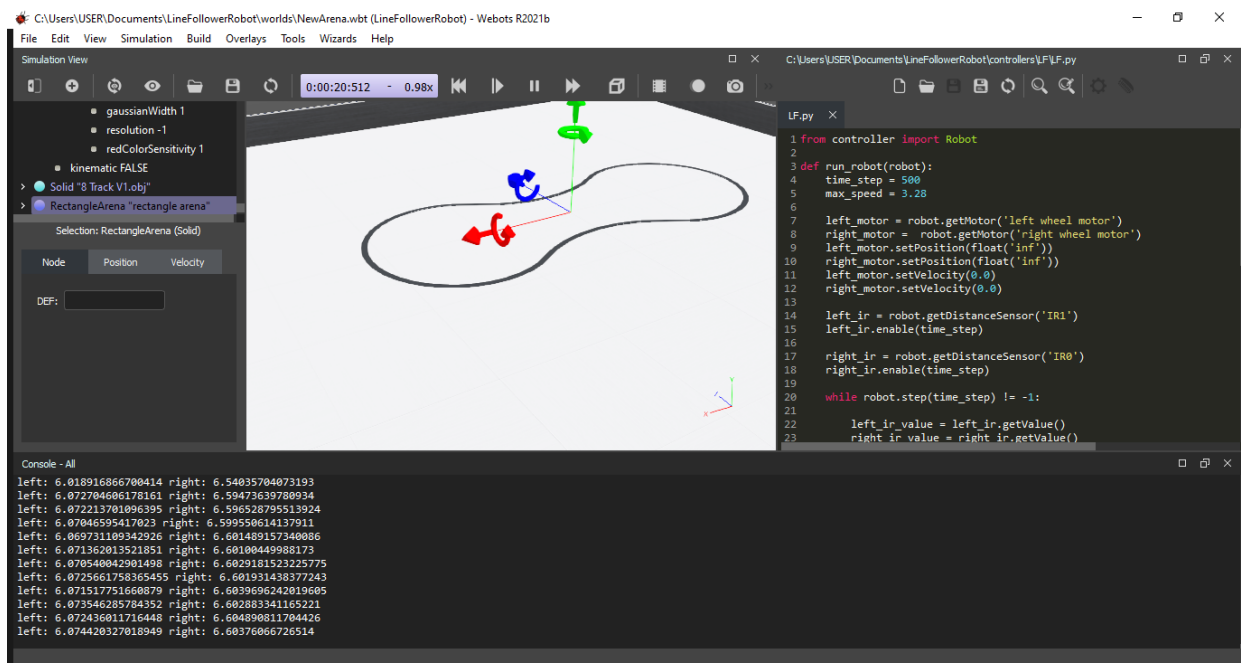
    right_speed))
    left_motor.setVelocity(left_speed)
    right_motor.setVelocity(right_speed)

if __name__ == "__main__":
    my_robot = Robot()
    run_robot(my_robot)
```

After creating the controller file, choose the controller before running the simulation.



Save the simulation and press the Run real-time button . The line follower robot will follow the track accordingly. The simulation may be paused , run step-by-step , in real time  or in fast  modes. Now we are going to modify the world and decrease the step of the physics simulation: this will increase the accuracy and stability of the simulation (but reduce the maximum simulation speed).



Conclusion

In this tutorial, you have been introduced to set up a world, add a robot and program it. The important thing is that you learned the fundamental concepts summarized below:

- A world is made up of nodes organized in a tree structure.
- A world is saved in a .wbt file stored in a Webots project.
- The project also contains the robot controller programs which define the behavior of the robots.

- Controllers may be written in C or other languages.
- C, C++ and Java controllers have to be explicitly compiled before they can be executed.
- Controllers are associated with robots via the `controller` fields of the `Robot` node.

SESSION 2 : Case Study

Case Study : Adaptive Speed Control

Track Design:

TASK: Try to develop a program that you create with arduino IDE to follow the customized line track, create controller for the line follower robot and run the robot in the track. We'll compete each robot at last day to see the performance of each robot!

There will be a certificate for the best presenter, so enjoy and be creative! :)

REFERENCE

- [1] M. Spong, Hutchinson & Vidyasagar, *Robot Modelling and Control*, Wiley, 2001.
- [2] Budiharto W., Santoso A., Purwanto D., Jazidie A., *A Navigation System for Service robot using Stereo Vision*, International conference on Control, Automation and Systems, Korea, pp 101-107, 2011.
- [3] Hutchinson S., Hager G., Corke P., *A tutorial on visual servo control*, IEEE Trans. On Robotics and Automation, vol. 12(5), pp. 651-670, 1996.
- [4] Budiharto W., Purwanto D., Jazidie A., *A Robust Obstacle Avoidance for Service Robot using Bayesian Approach*, International Journal of Advanced Robotic Systems, Intech publisher, vol 8(1), 2011.