

Human-Robot Interaction and Autonomous Systems

AI 616

Master of Science in Artificial intelligence

Prepared by Dr. Khaled Eskaf

Lecture 4





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



Chapter

03

Week 3 Outlines



Course Outlines

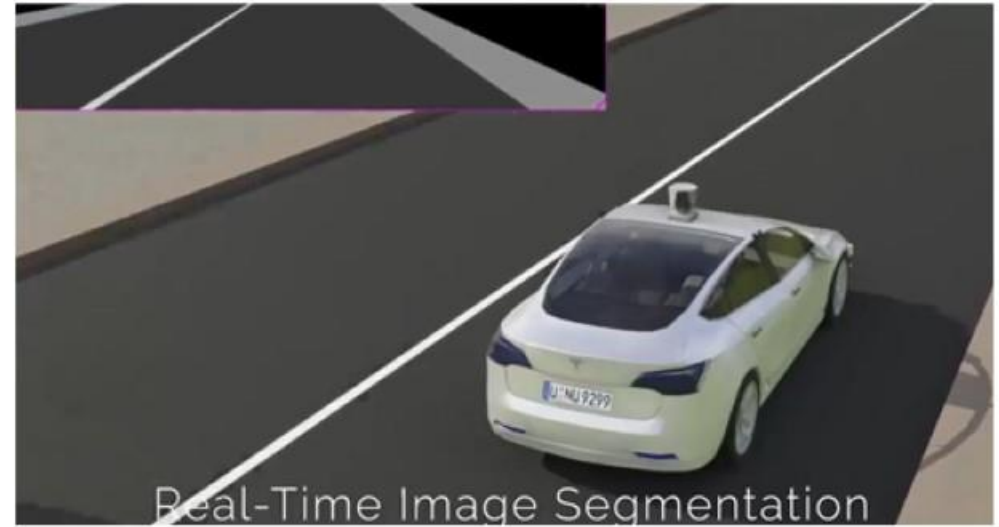
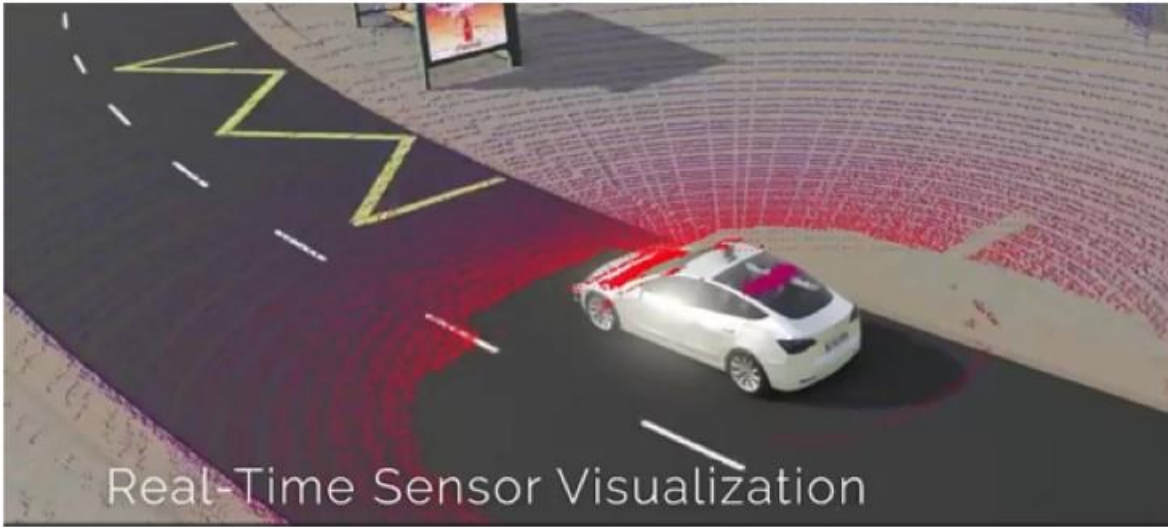
Module	Content	Simulation Lab
1. Introduction to Robotics	- Fundamentals of Robot Control - Classification of Robotics - Exploration of Robot Types	- Basic Robot Control - Modeling Robot Classes - Simulating Various Robot Types
2. Advanced HRI Studies	- Introductory to Advanced Concepts in HRI	- Basic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



Used in Thousands of Universities Worldwide



Rapidly Design your Robot Model



Webots is Amazing Because:

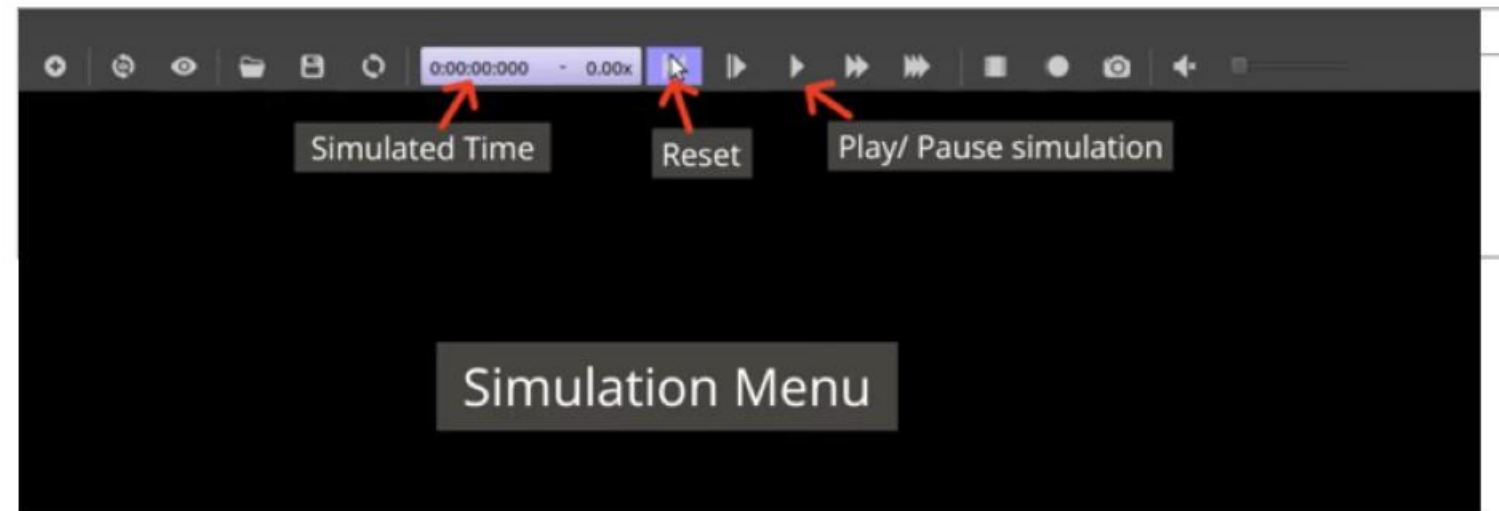
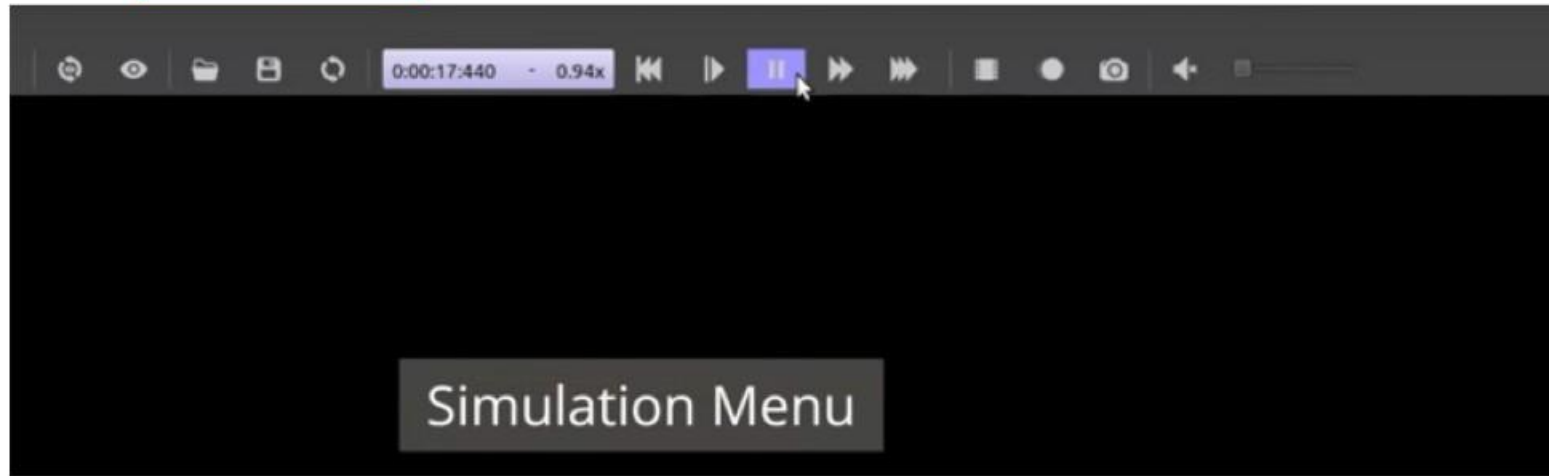
- **Real Robots on Your Screen:** Your virtual robots in Webots move just like real ones. It's like having a robot pal in your computer!
- **Create Your Own Robot Places:** Make all sorts of places for your robots to explore, like cities, parks, or space stations!
- **Loads of Ready-Made Robots:** Choose from many robots that are already built in Webots. Start playing and learning right away.
- **Works on Any Computer:** You can use Webots on Windows, Mac, or Linux. It's super flexible!
- **Easy Robot Coding with Python:** Use Python, a simple programming language, to tell your robots what to do. Perfect for beginners and experts!
- **Teach Robots to Think:** Make your robots smart with Python. They can learn to solve problems by themselves, like real AI!
- **Robots with Cool Parts:** Add neat parts to your robots, like cameras, hands, or sensors, so they can see, touch, and feel.
- **All-in-One Robot Kit:** Webots has everything you need to create and control amazing robots.
- **For School, Research, and Business:** You can play with Webots alone, use it in school, or even in big research projects and businesses.
- **Used by Top Universities and Companies:** Many of the best universities and big companies in America use Webots. Places like MIT, Stanford, and tech giants in Silicon Valley trust Webots for their robot projects.

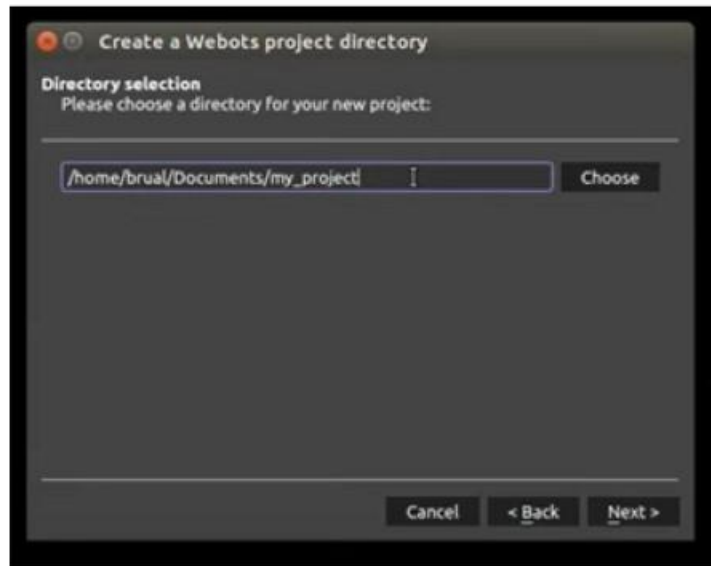
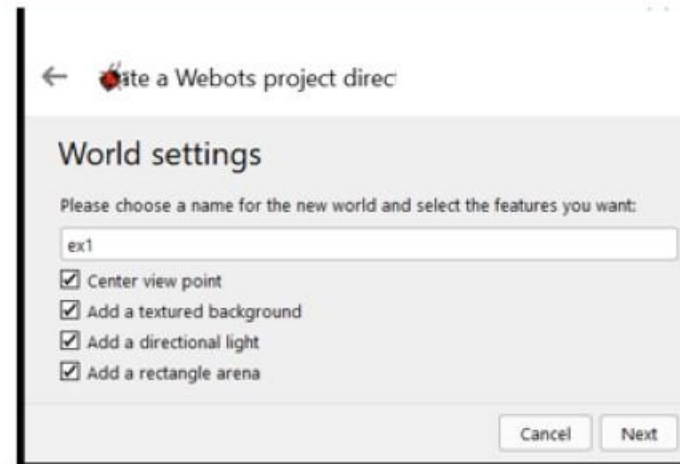
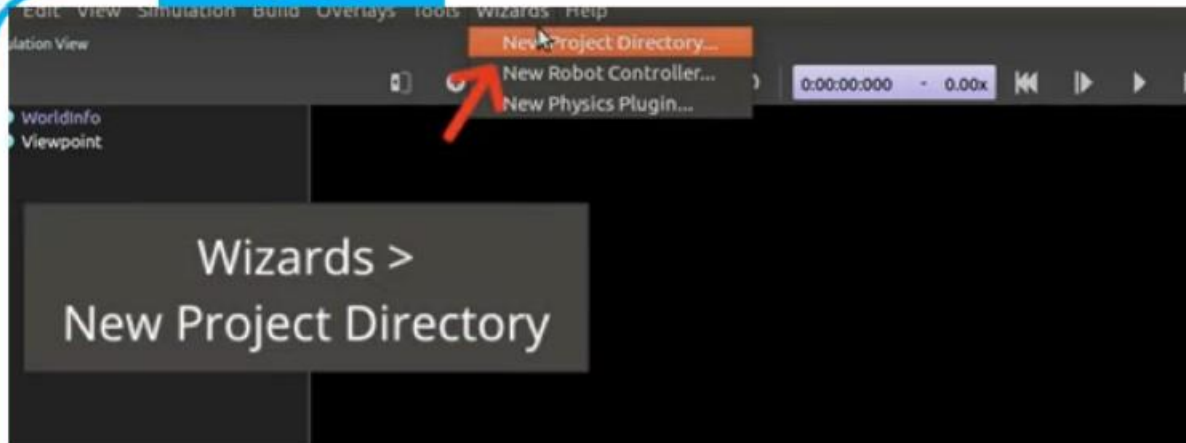
Download & Install
User interface (UI)
Create your first project
Make your own object
Understand fundamentals
Run your first simulation
Add a robot & run simulation
Where is the code

Objective

User Interface (UI)

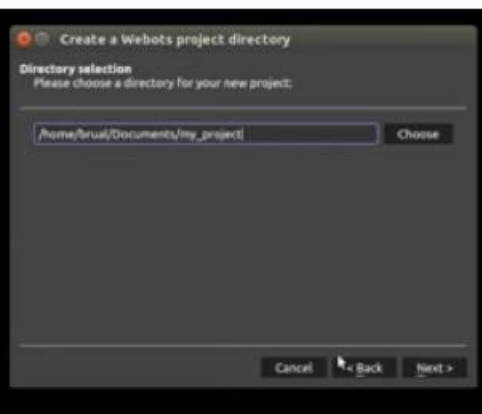
Create your first Project





(1) Use "Choose" to choose where to save your project.

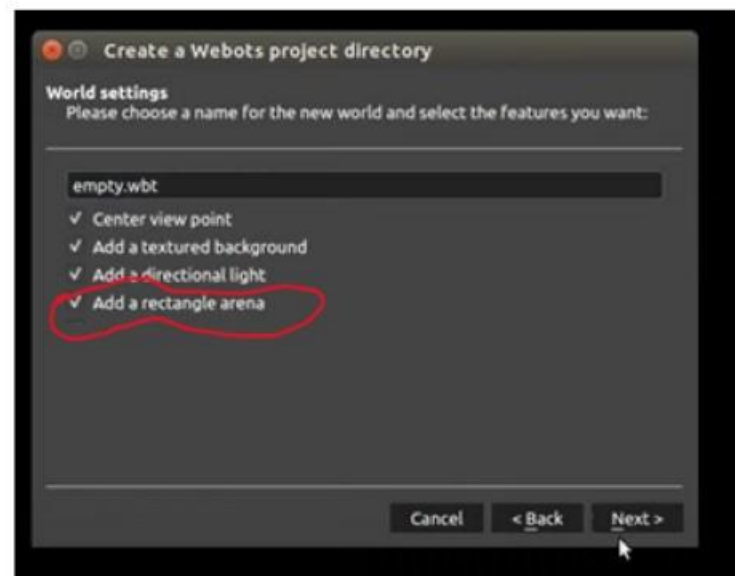
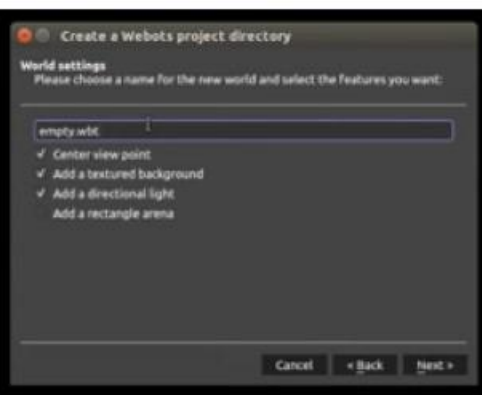
(2) Name your project.
Eg: my_project

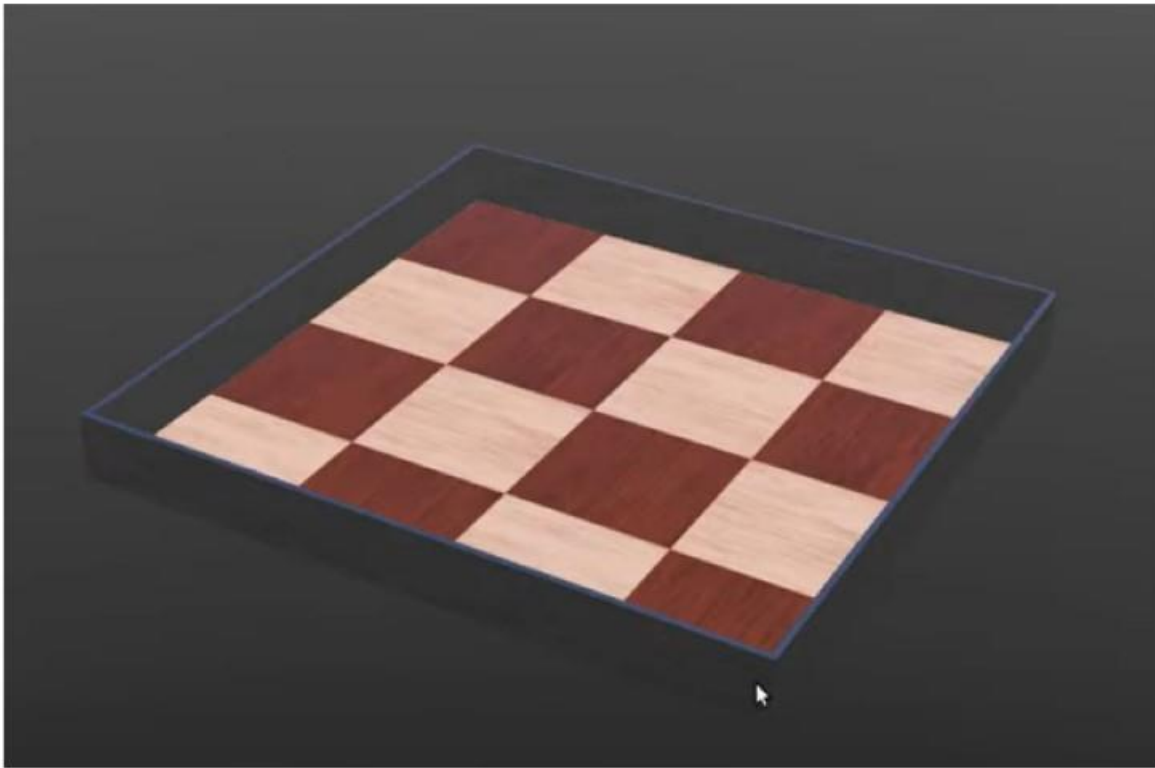


(3) Click Next

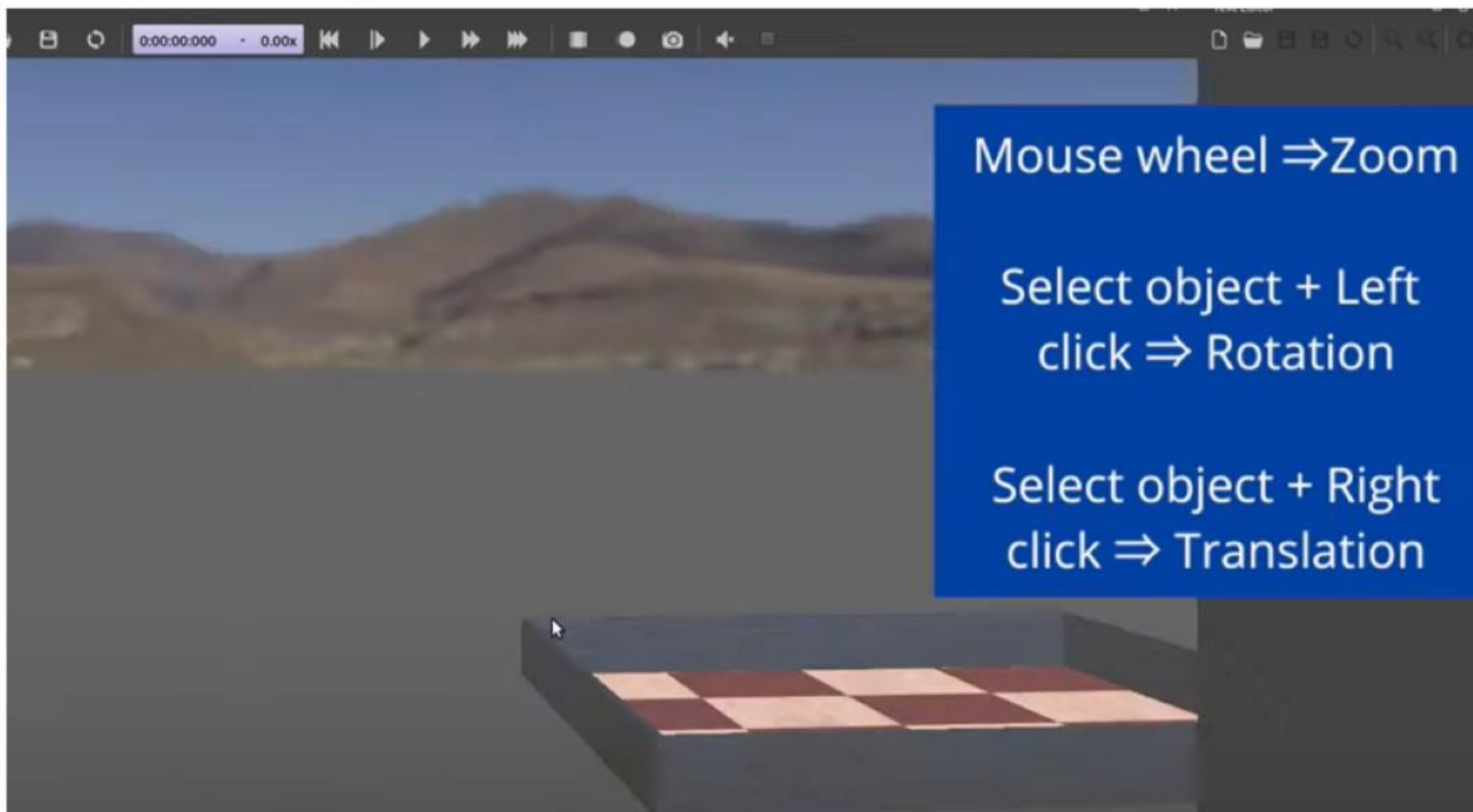
(4) Select "Add rectangle arena"

(5) Click Next & Finish





Loaded with a rectangular arena
essentially a floor with four walls.

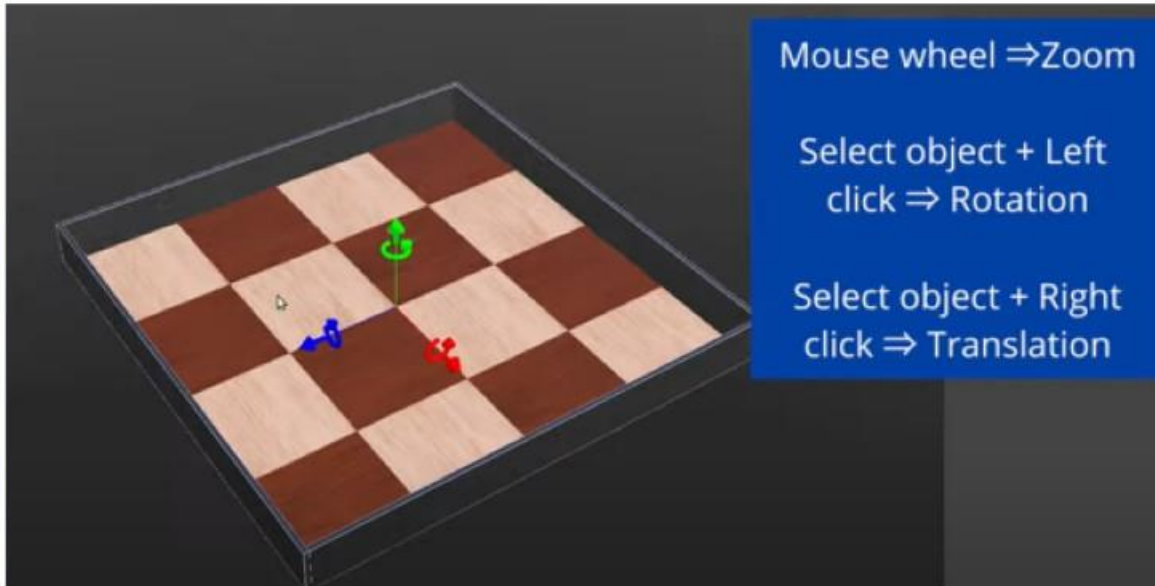


Mouse wheel $\Rightarrow$ Zoom

Select object + Left
click $\Rightarrow$ Rotation

Select object + Right
click $\Rightarrow$ Translation

learn how to zoom in
and zoom out this is
simply done by scrolling
in and out

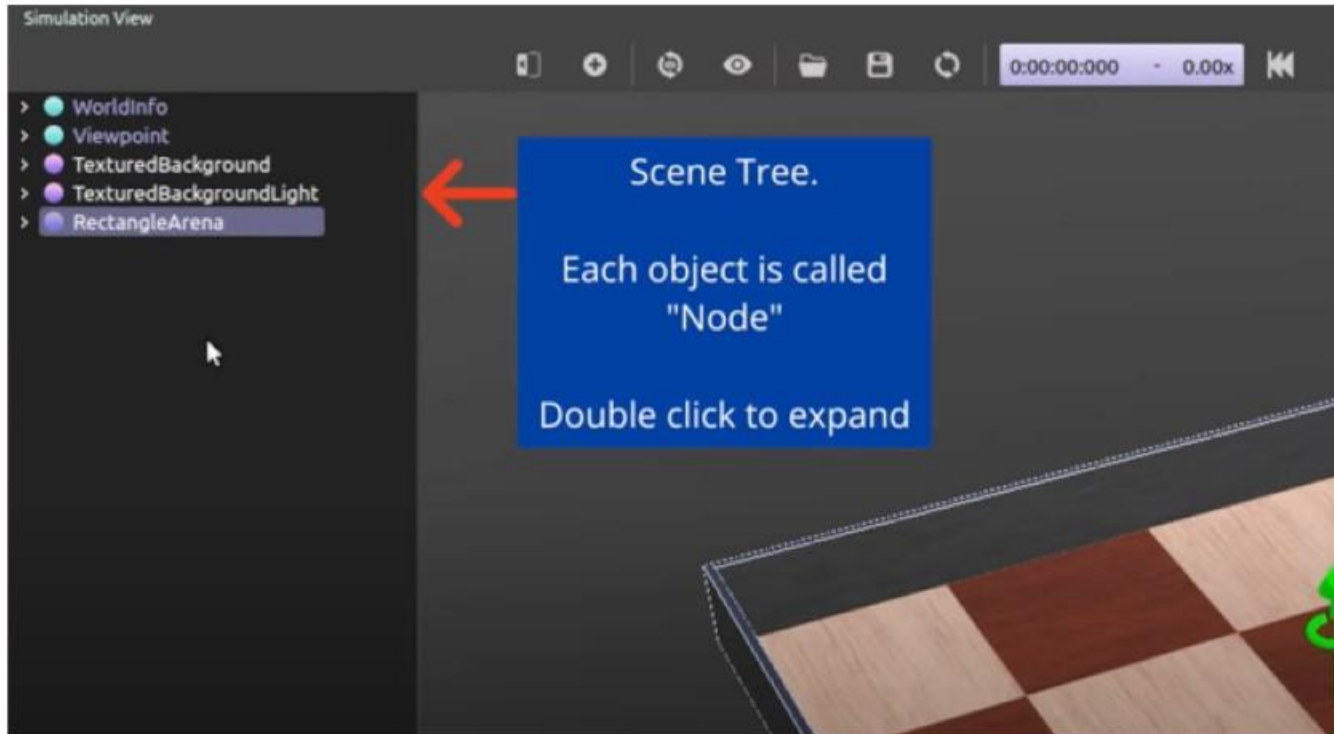


Mouse wheel $\Rightarrow$ Zoom

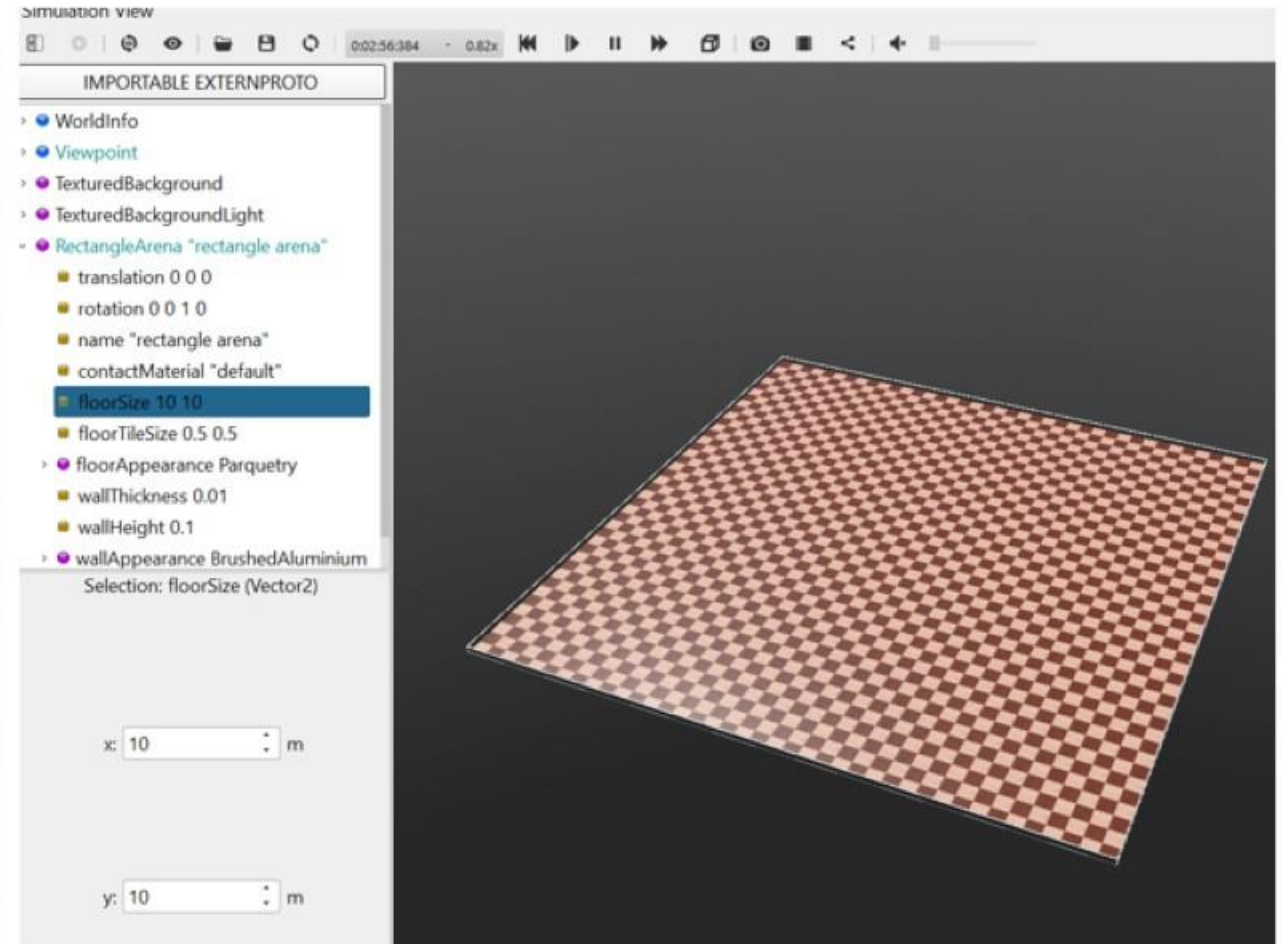
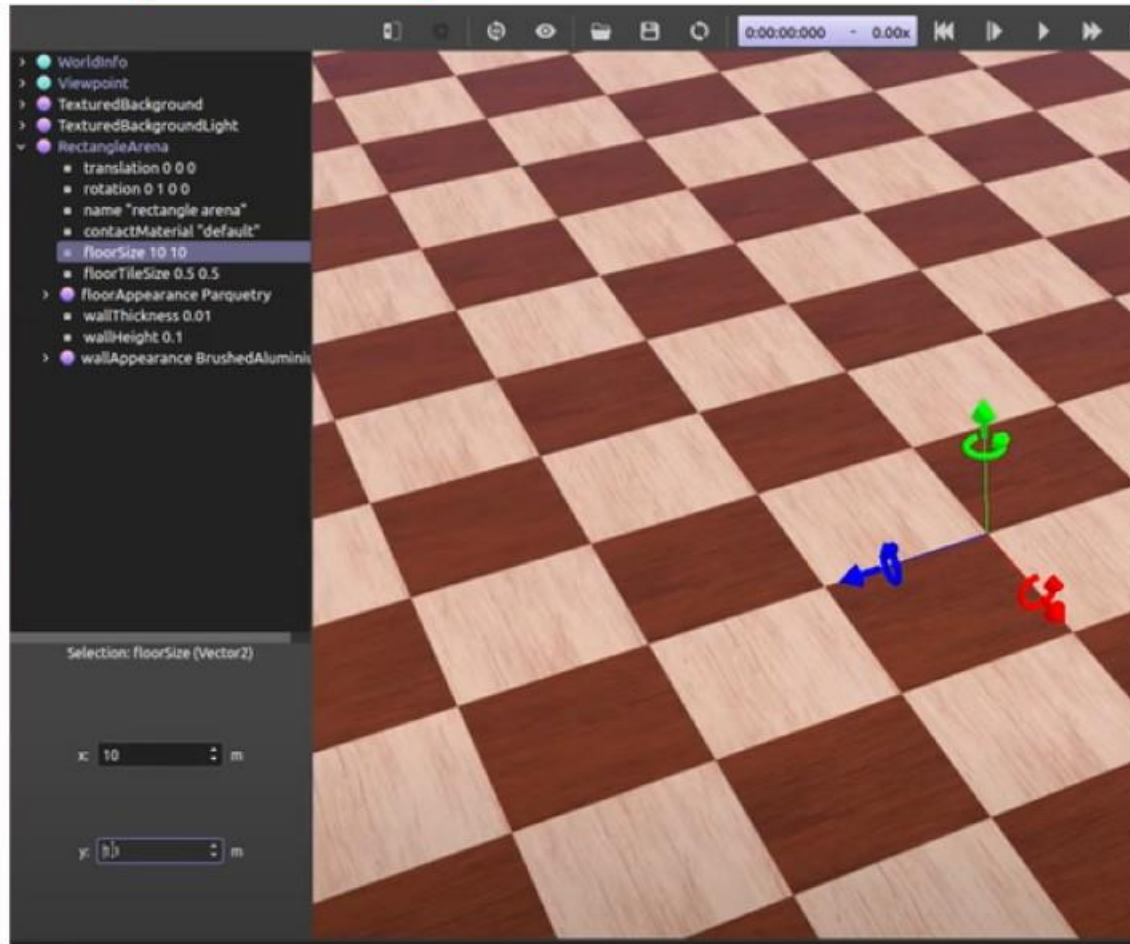
Select object + Left
click $\Rightarrow$ Rotation

Select object + Right
click $\Rightarrow$ Translation

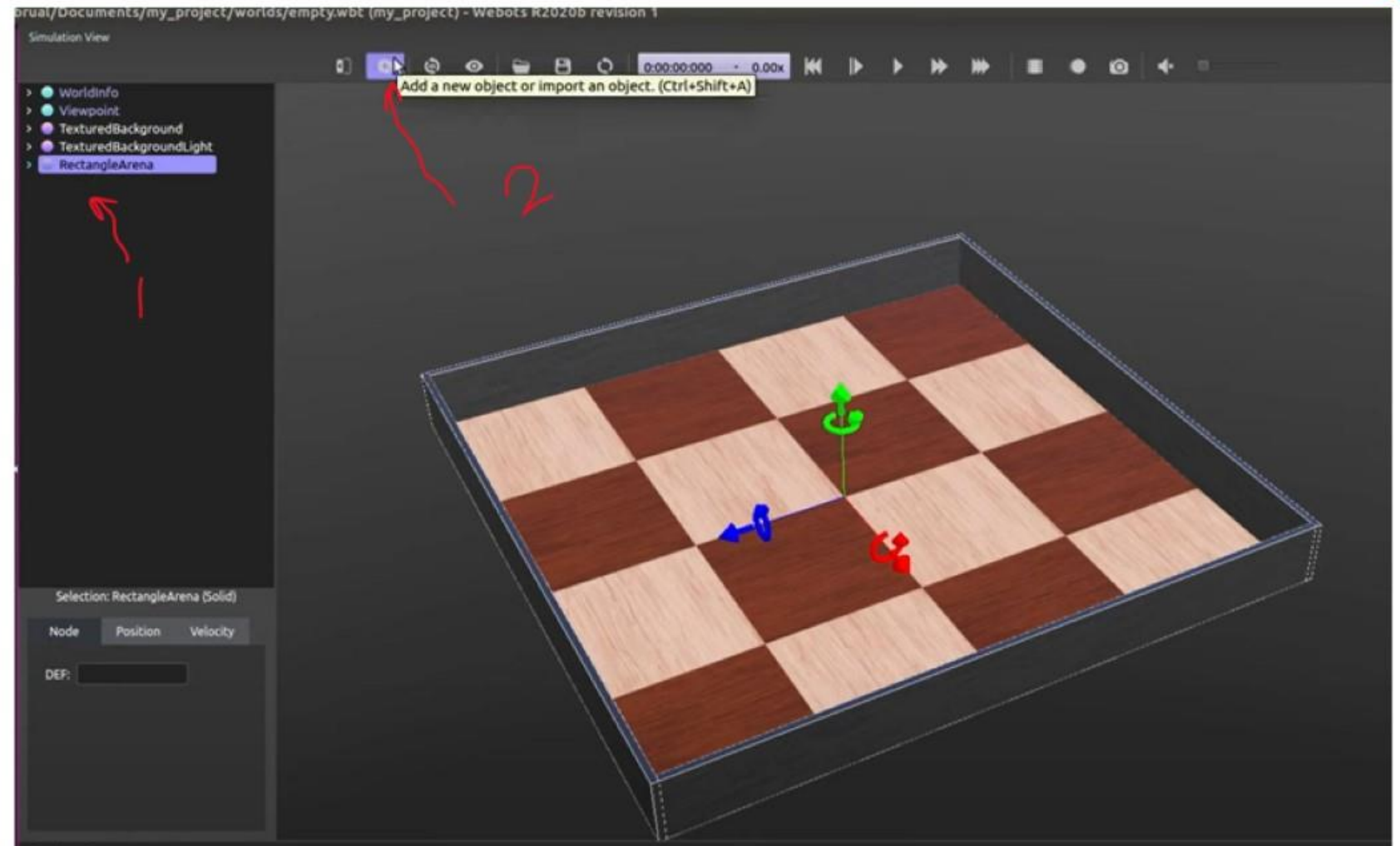
you can click on it and you can move it
around to look at different views.

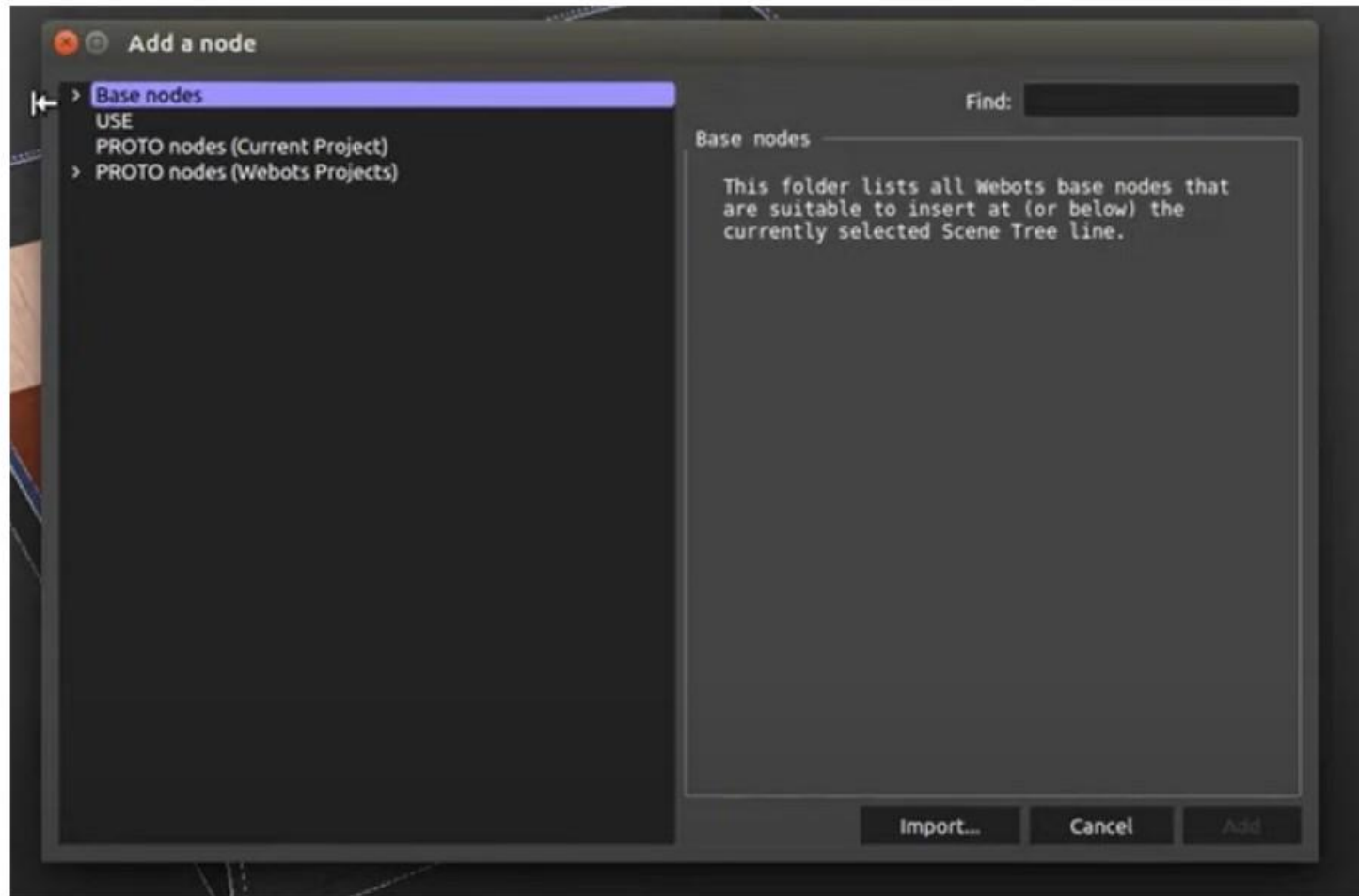


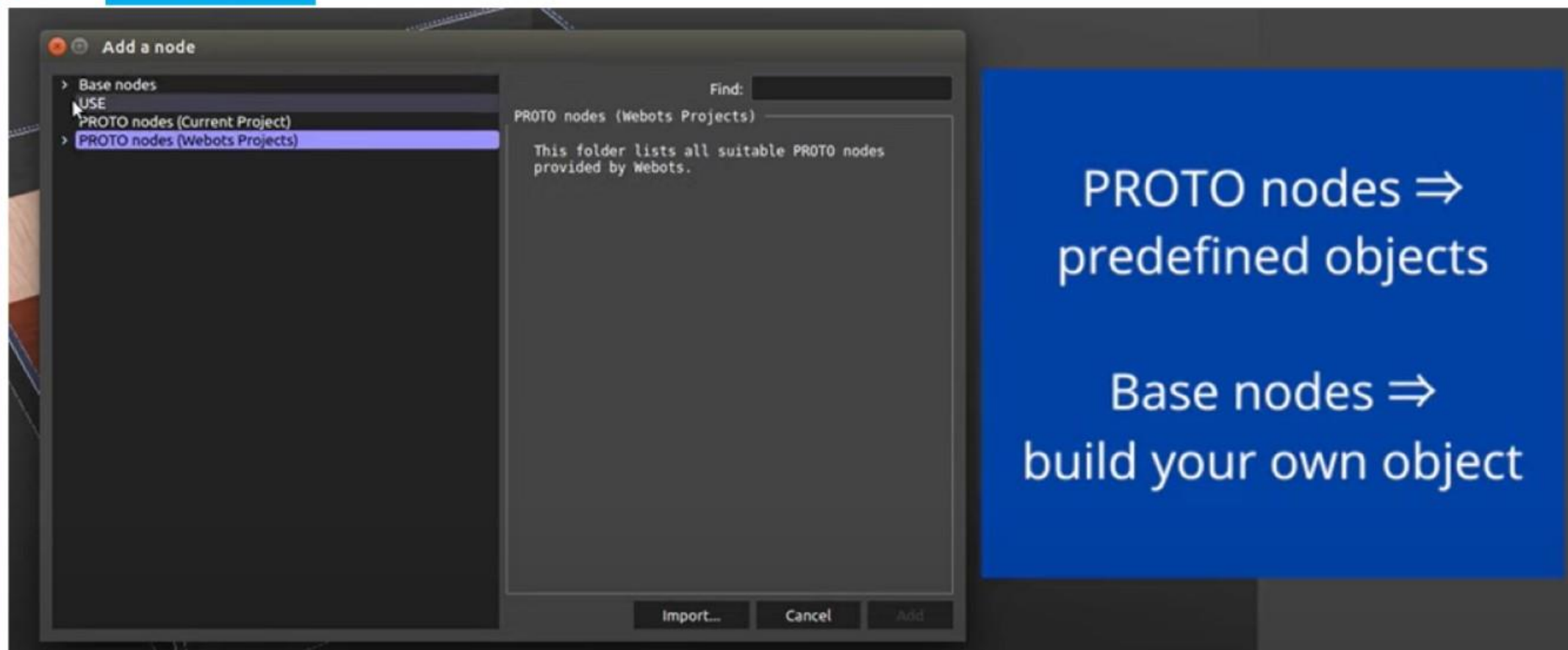
Change the Floor Size



create an object of our own we will first go over the steps to create an object

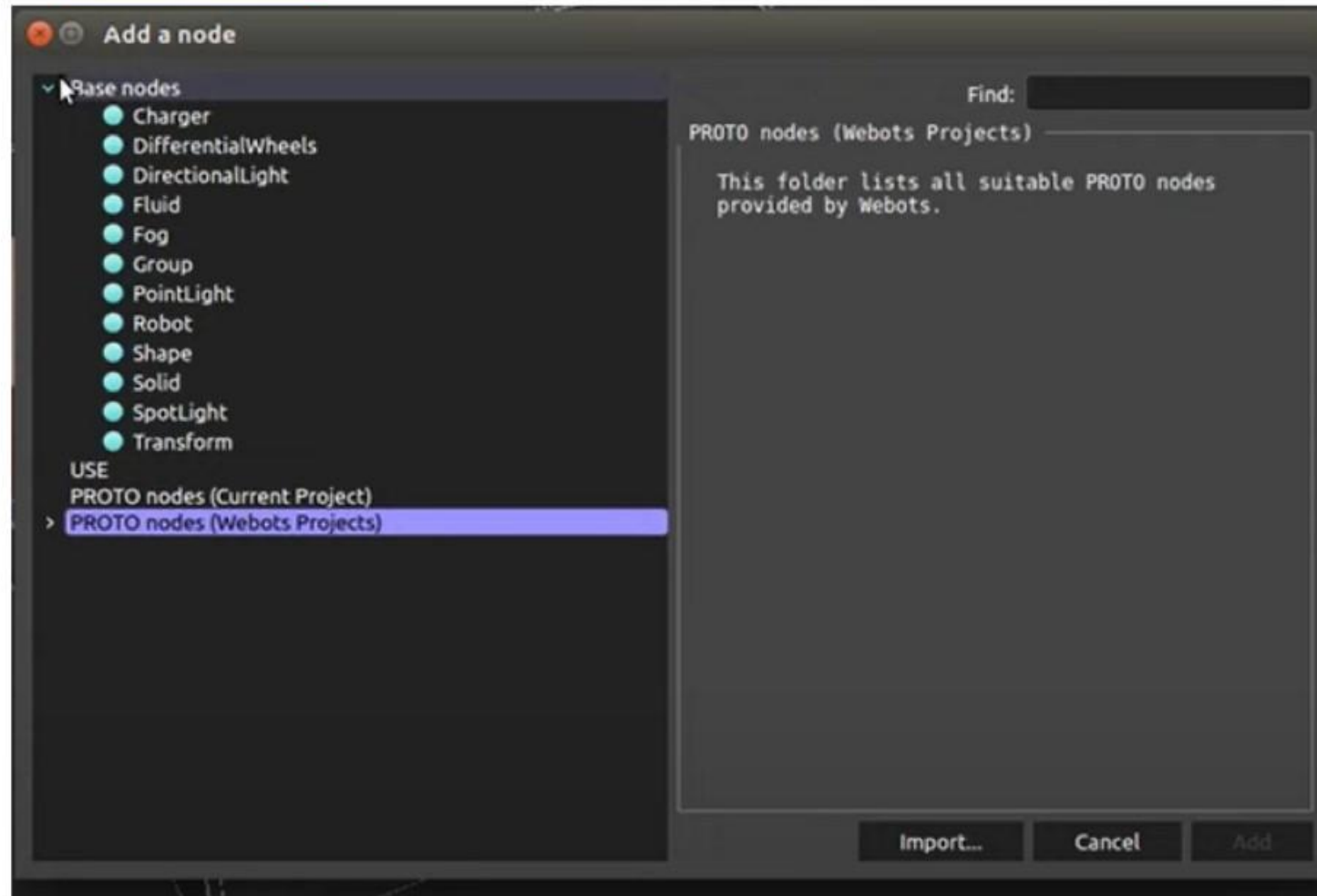




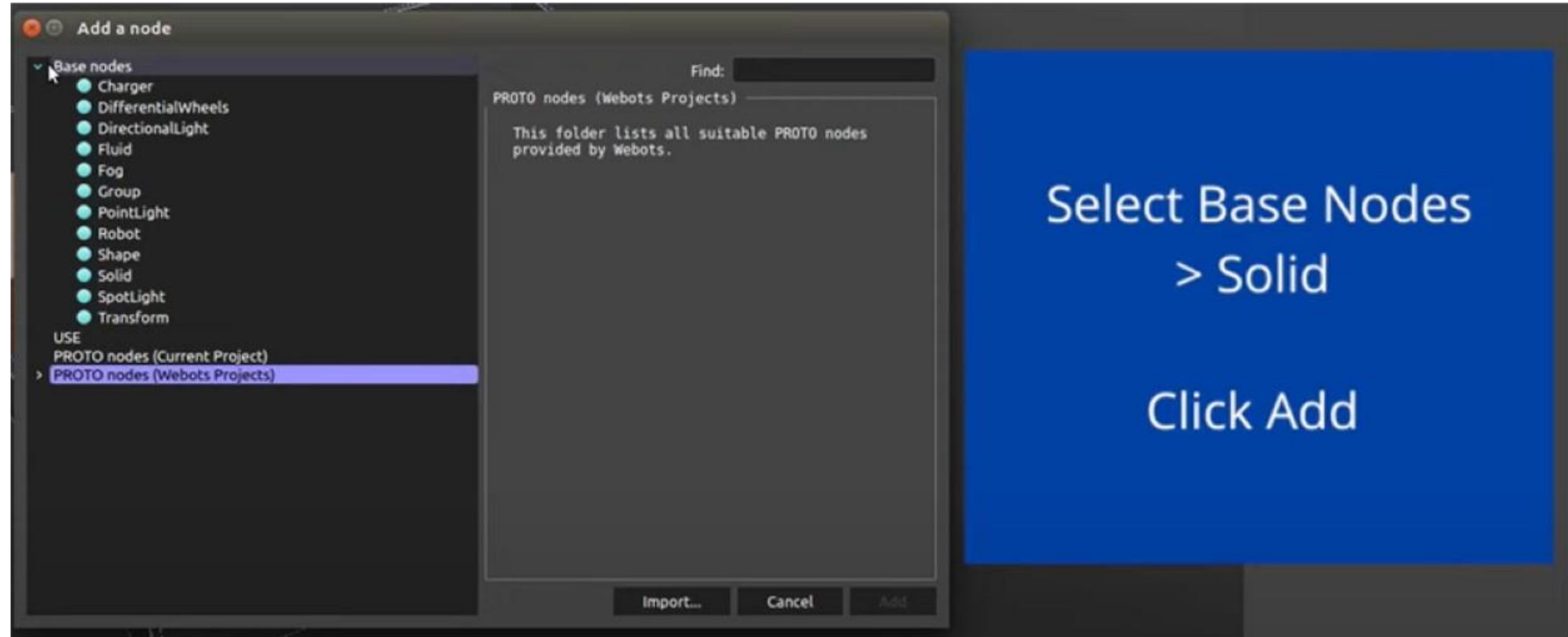


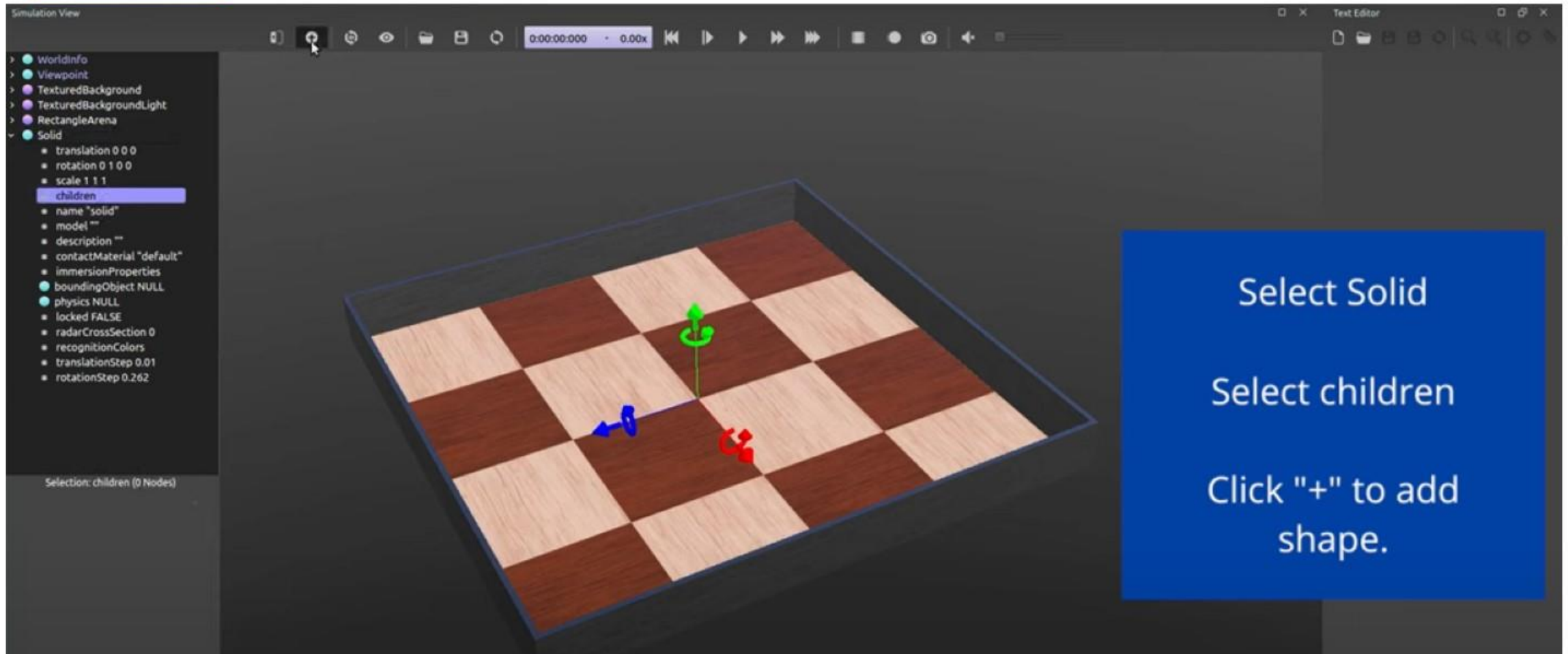
PROTO nodes $\Rightarrow$
predefined objects

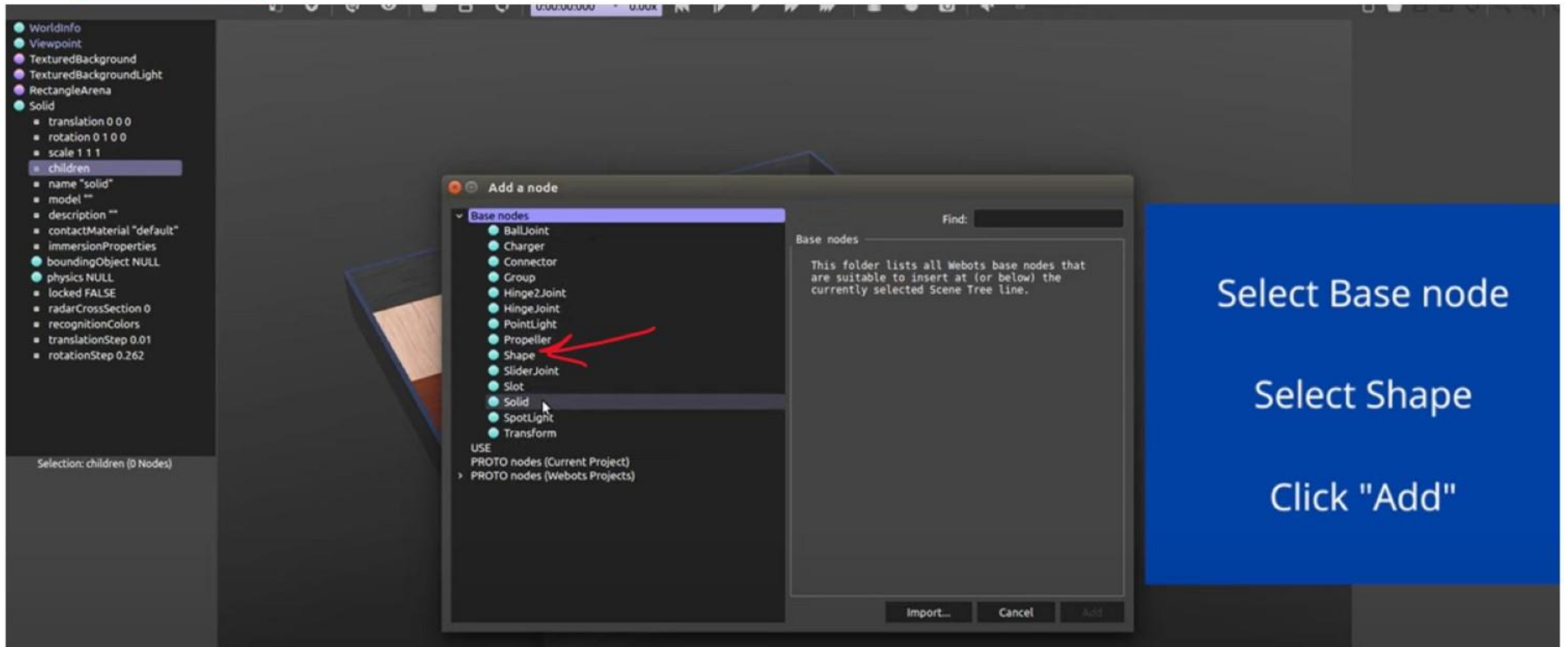
Base nodes $\Rightarrow$
build your own object



Create a NEW Object:



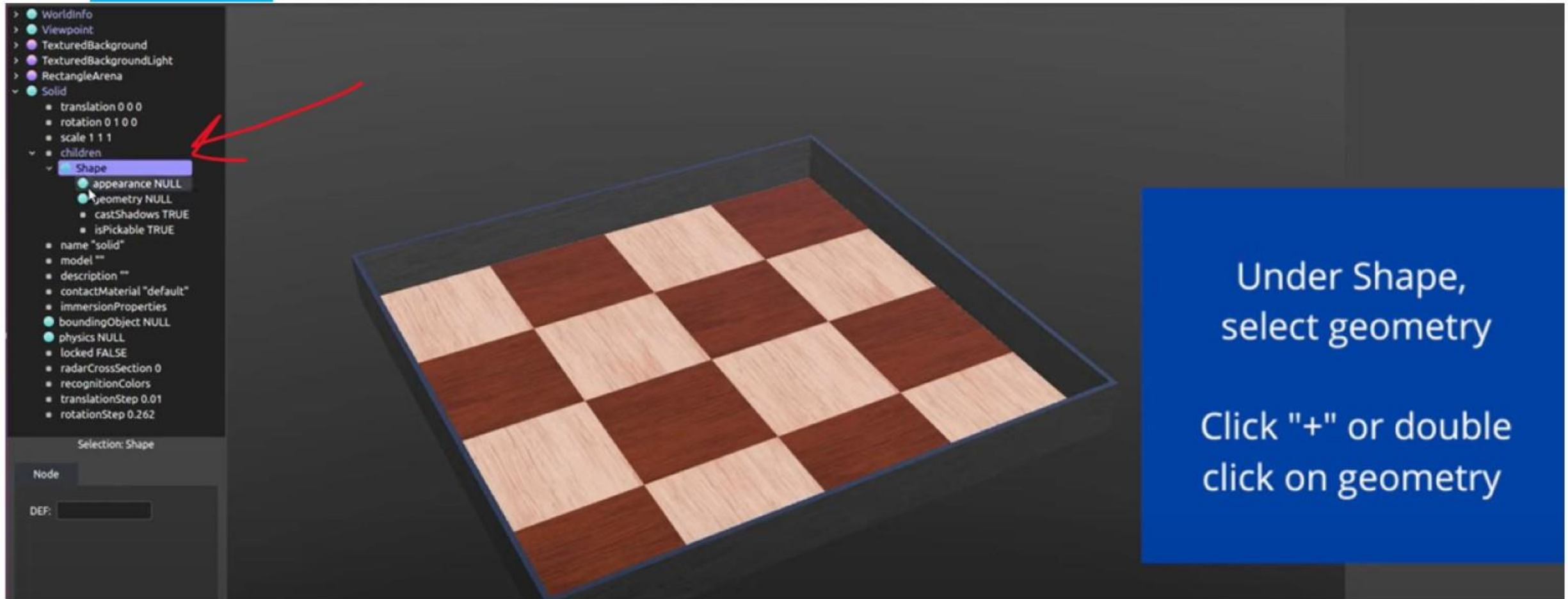




Select Base node

Select Shape

Click "Add"



The screenshot shows a 3D environment with a chessboard on a dark surface. On the left, a properties panel lists various objects and their attributes. A red arrow points to the 'Shape' property under the 'Solid' object.

Properties Panel:

- > WorldInfo
- > Viewpoint
- > TexturedBackground
- > TexturedBackgroundLight
- > RectangleArena
- > Solid
 - translation 0 0 0
 - rotation 0 1 0 0
 - scale 1 1 1
 - children
 - Shape (selected)
 - appearance NULL
 - geometry NULL
 - castShadows TRUE
 - isPickable TRUE
 - name "solid"
 - model ""
 - description ""
 - contactMaterial "default"
 - immersionProperties
 - boundingObject NULL
 - physics NULL
 - locked FALSE
 - radarCrossSection 0
 - recognitionColors
 - translationStep 0.01
 - rotationStep 0.262

Selection: Shape

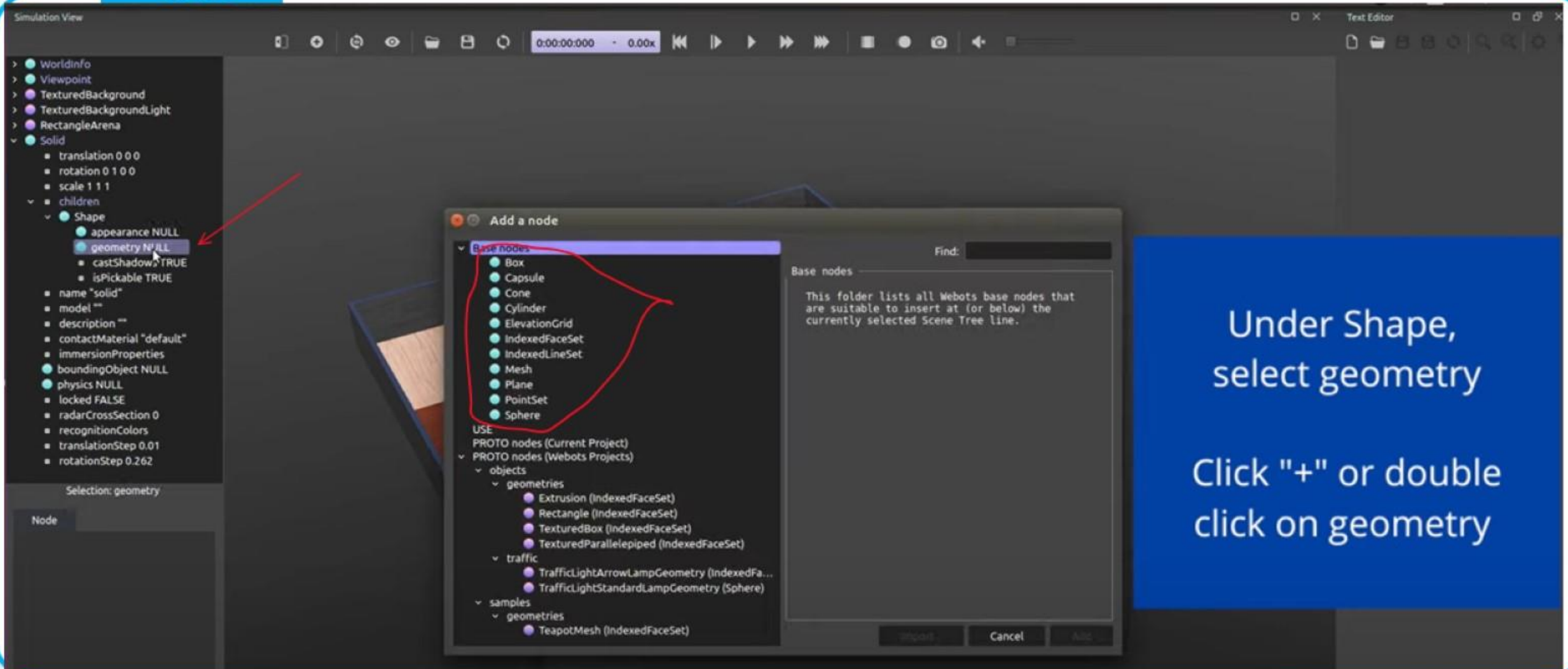
Node

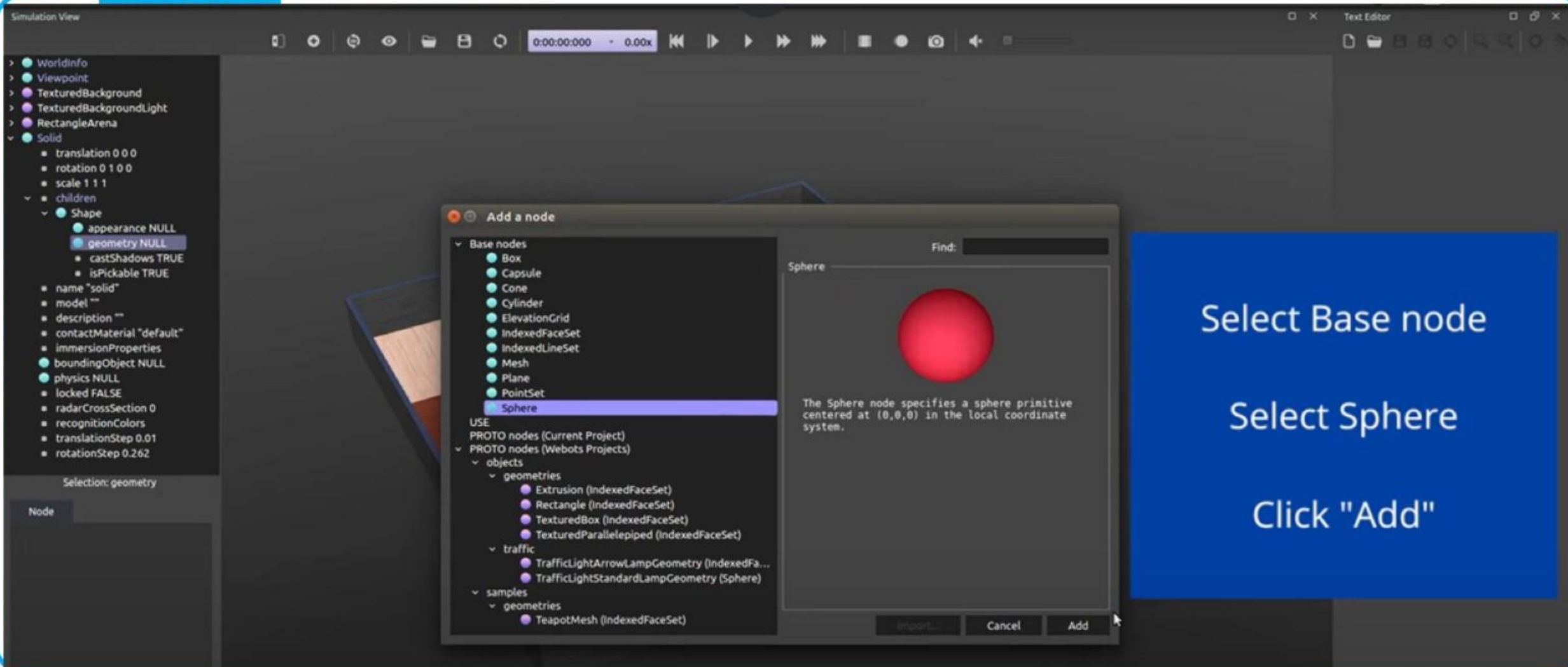
DEF: [input field]

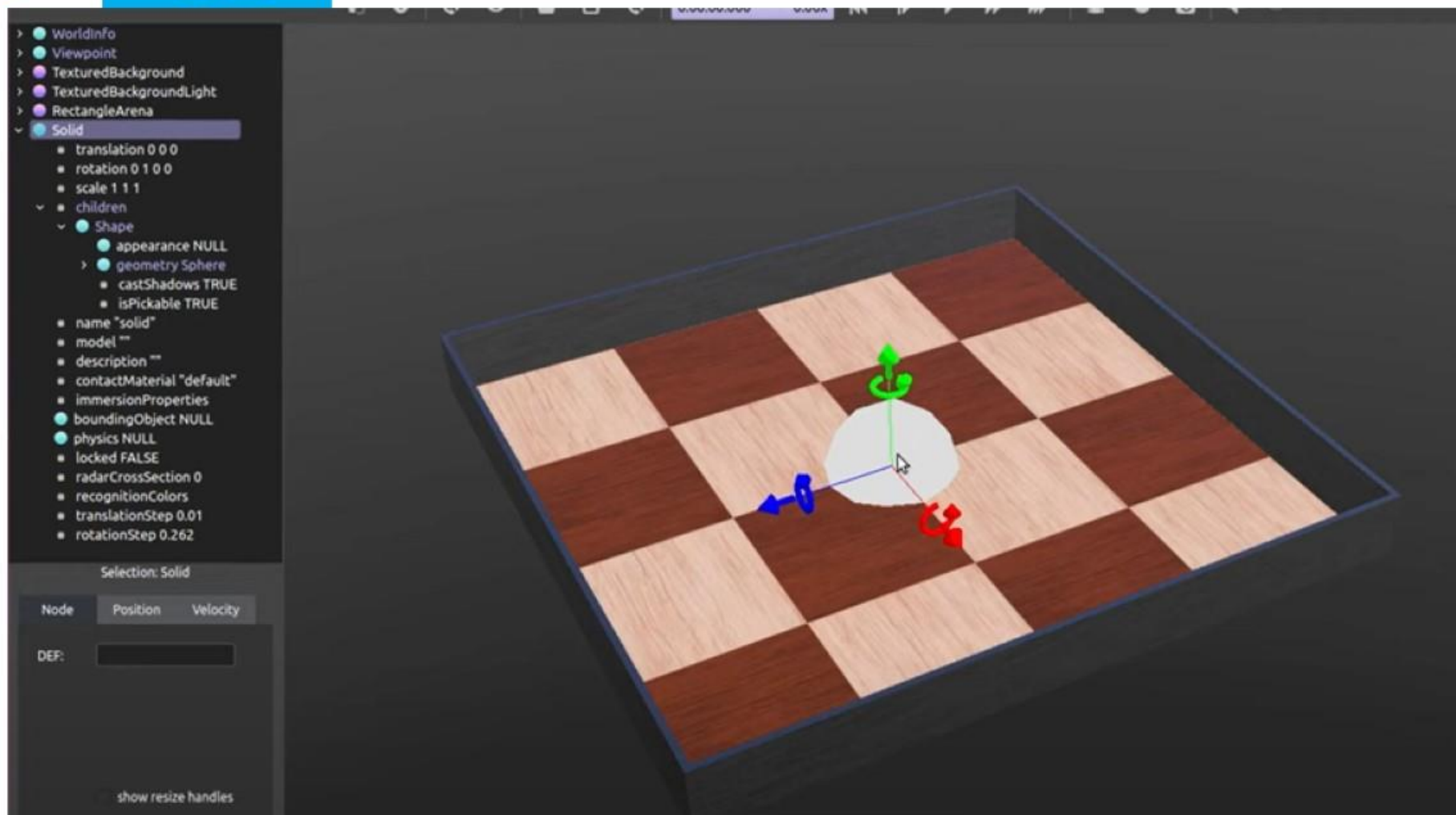
3D View: A chessboard with a dark brown border and a checkered pattern of light and dark brown squares.

Instructions:

- Under Shape, select geometry
- Click "+" or double click on geometry







All objects added at (0,0,0).

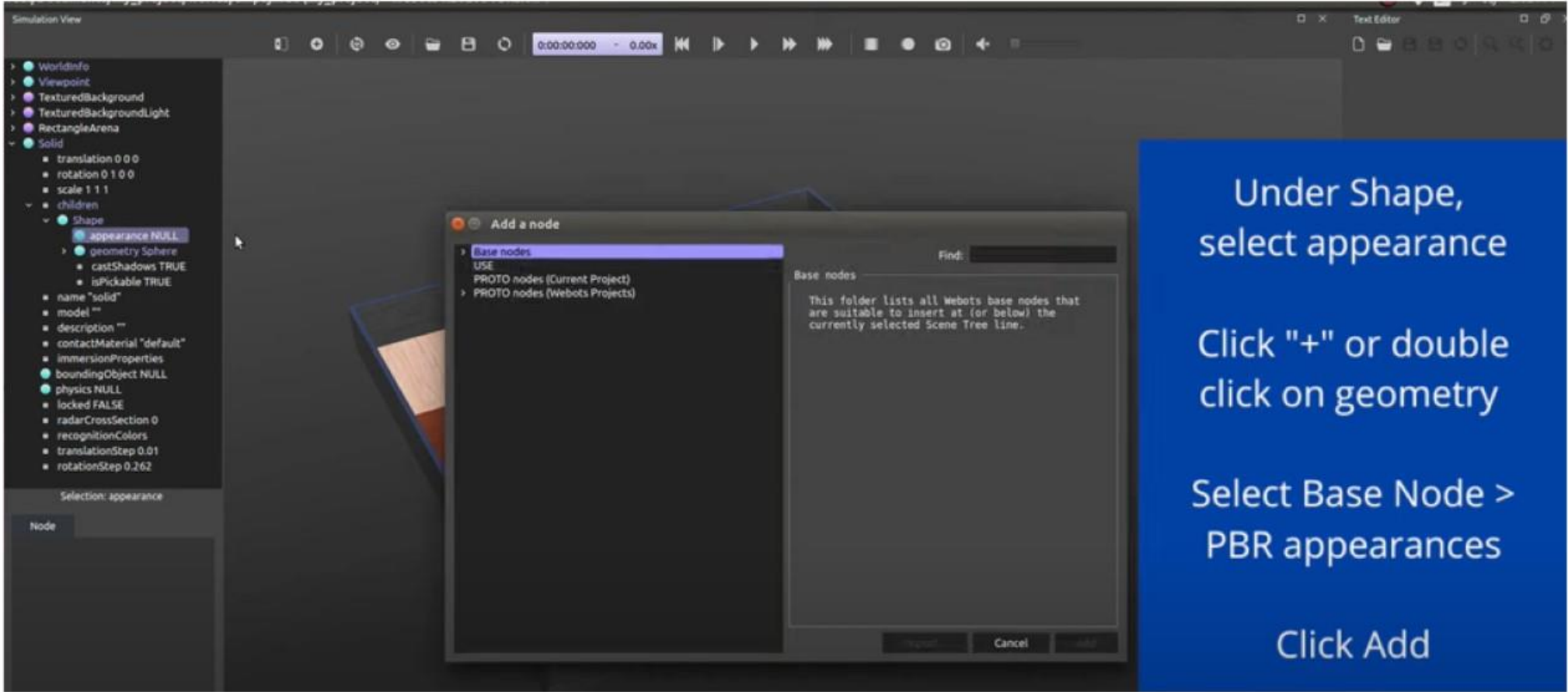
- > WorldInfo
- > Viewpoint
- > TexturedBackground
- > TexturedBackgroundLight
- > RectangleArena
- > Solid
 - translation 0 0 0
 - rotation 0 1 0 0
 - scale 1 1 1
 - > children
 - > Shape
 - appearance NULL
 - > geometry Sphere
 - radius 0.1
 - subdivision 1
 - ico TRUE
 - castShadows TRUE
 - isPickable TRUE
 - name "solid"
 - model ""
 - description ""
 - contactMaterial "default"
 - immersionProperties
 - boundingObject NULL
 - physics NULL
 - locked FALSE
 - radarCrossSection 0
 - recognitionColors

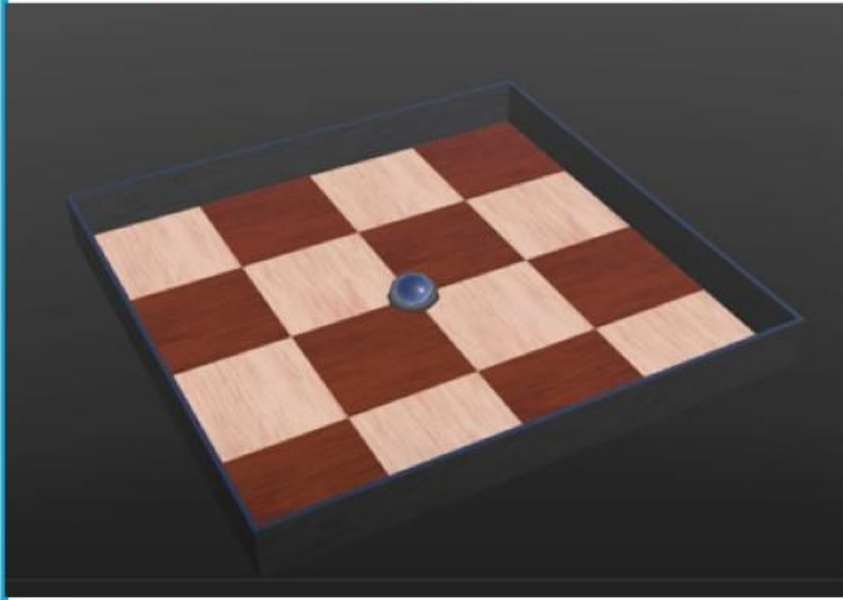
Selection: radius (Float)

Under geometry,
there are options to
change size.

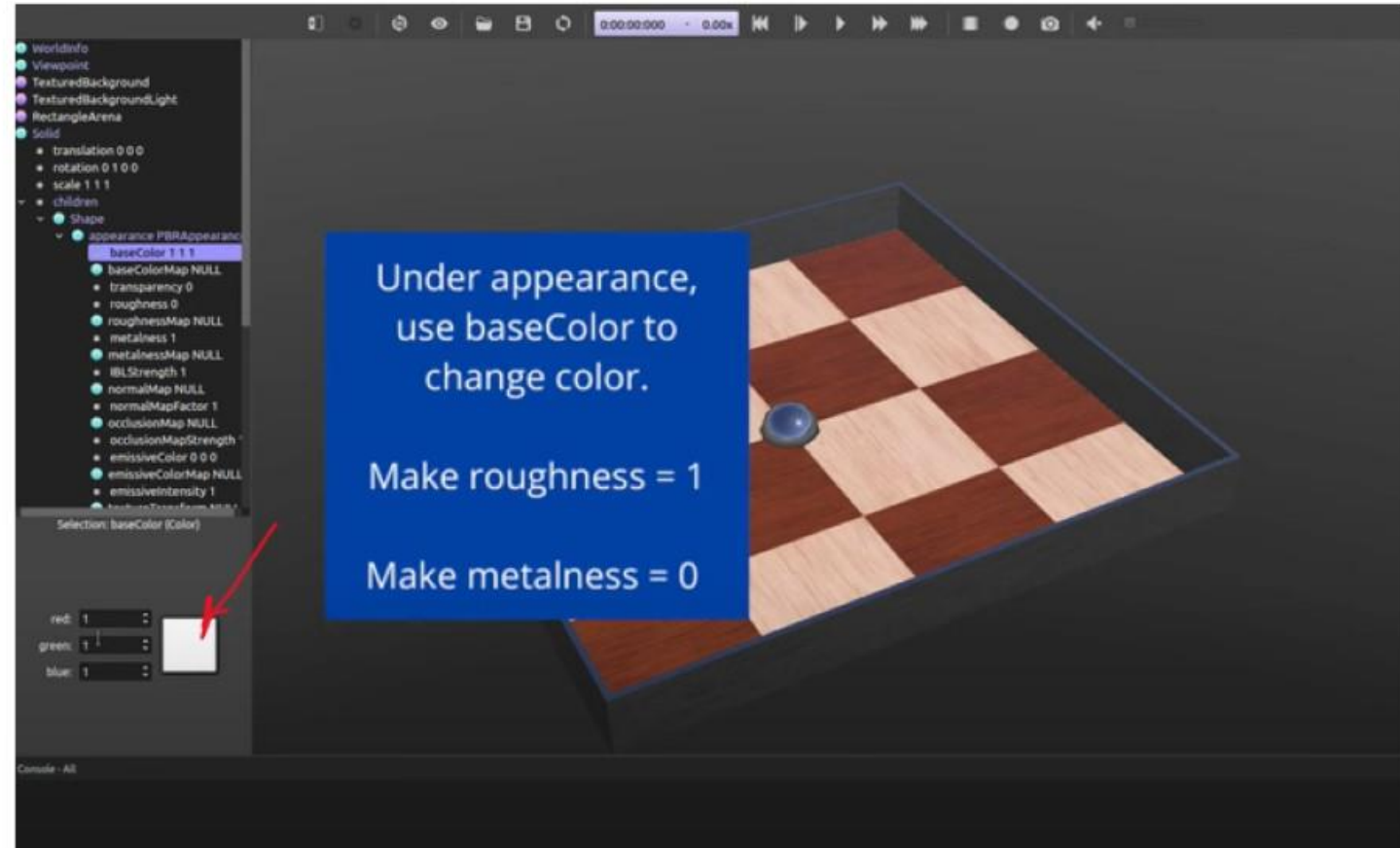
Select radius

the dimensions so we could go
from 0.1 to
0.05 and you can see the
sphere has reduced





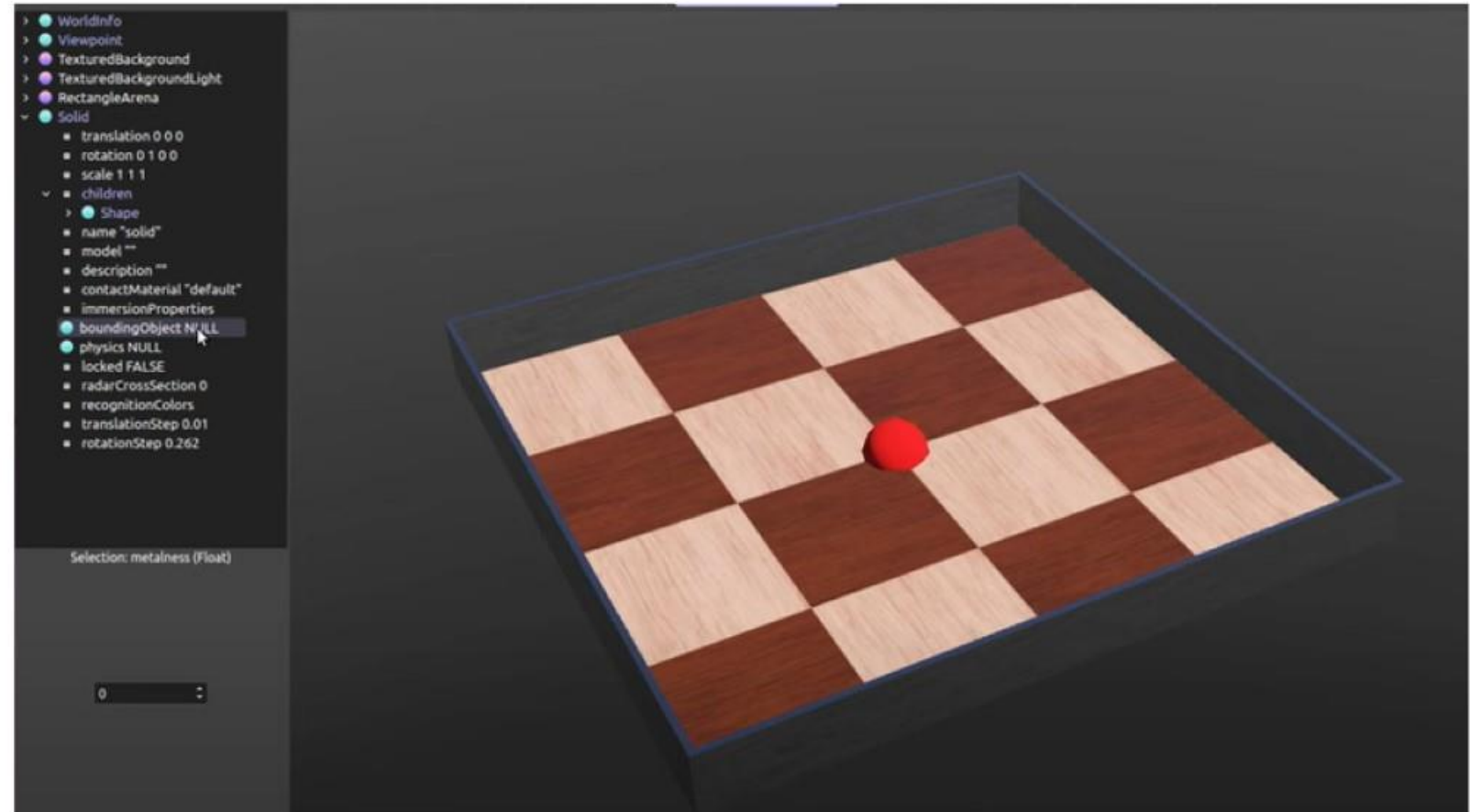
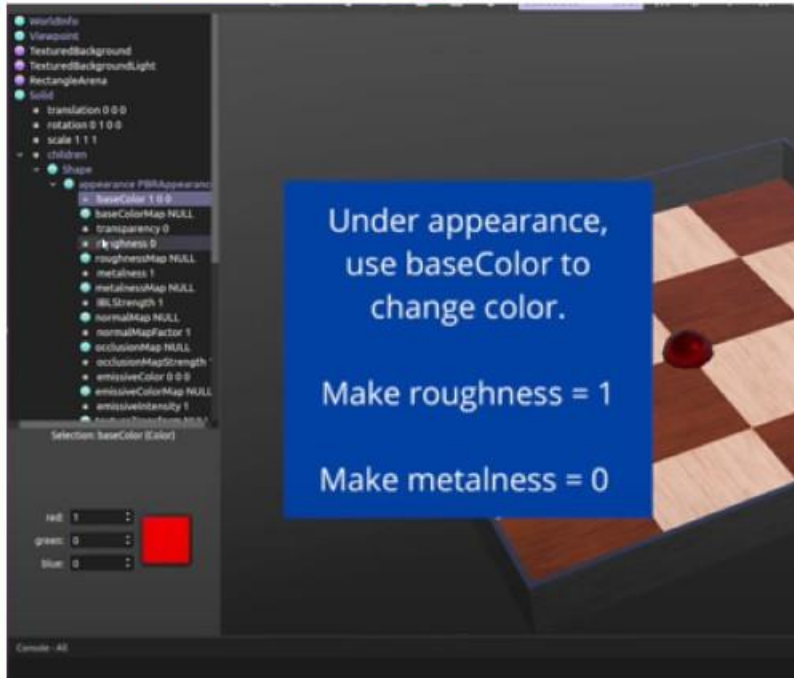
You have different options:



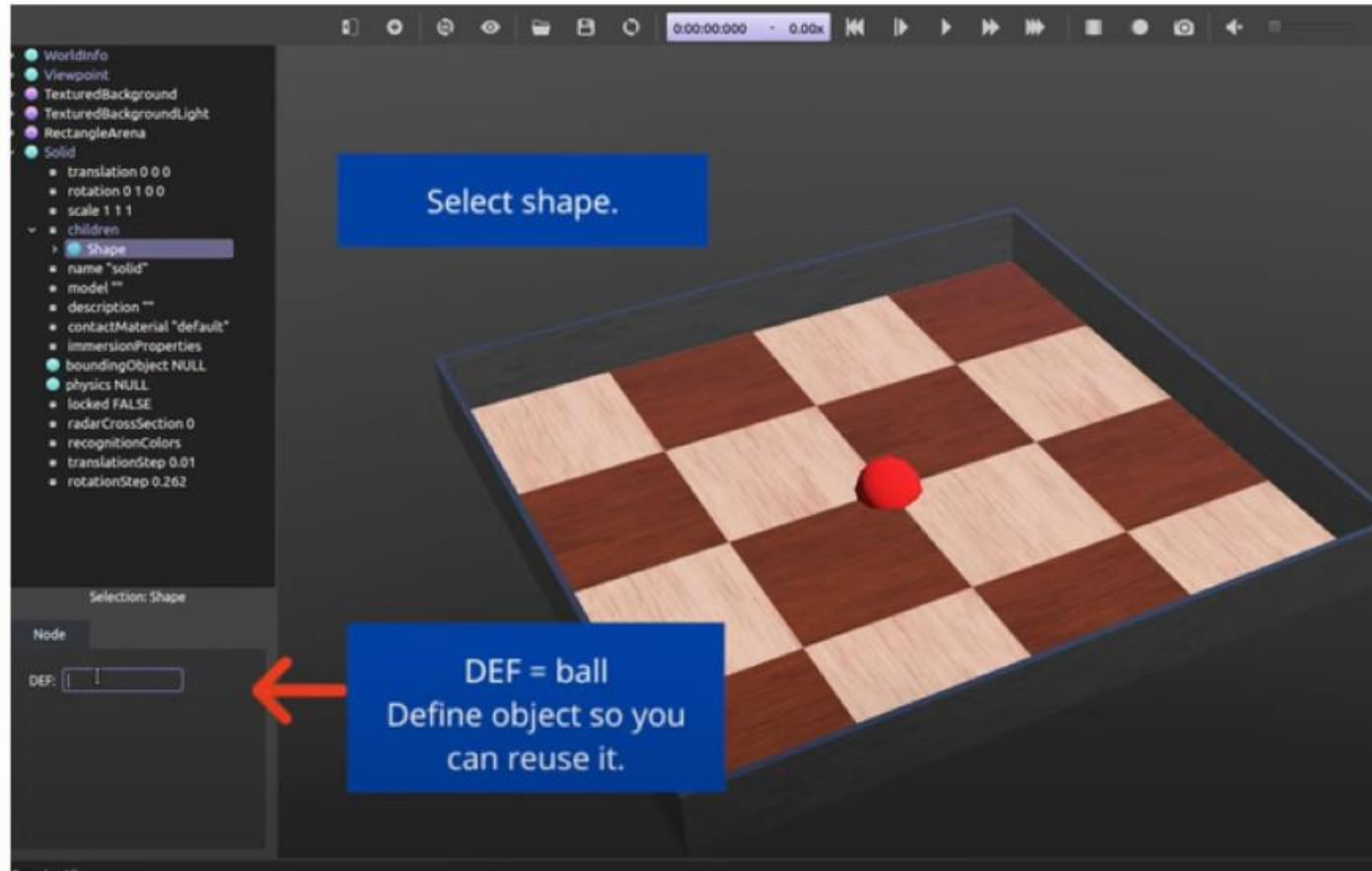
Under appearance,
use baseColor to
change color.

Make roughness = 1

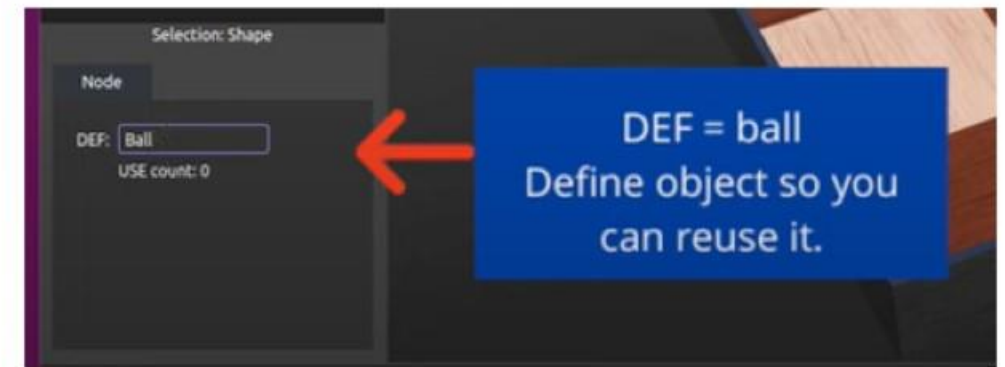
Make metalness = 0

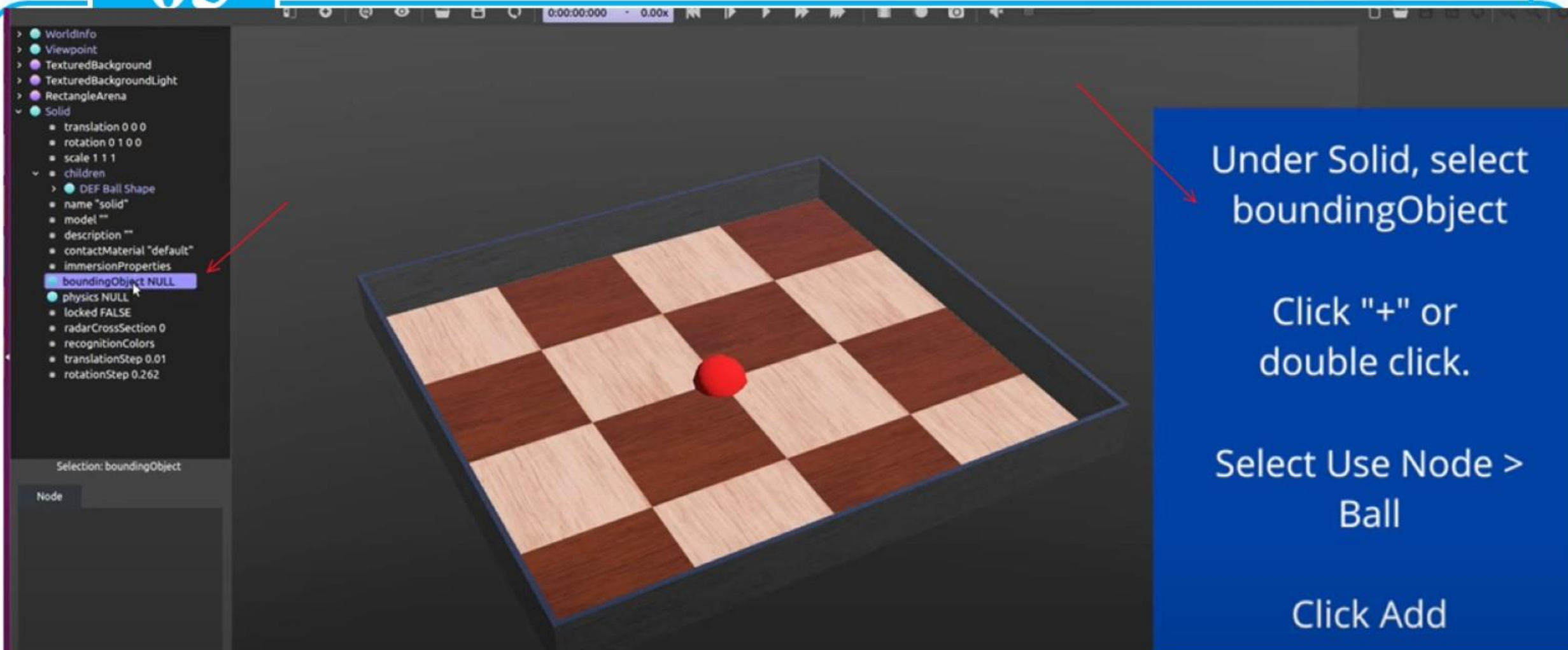


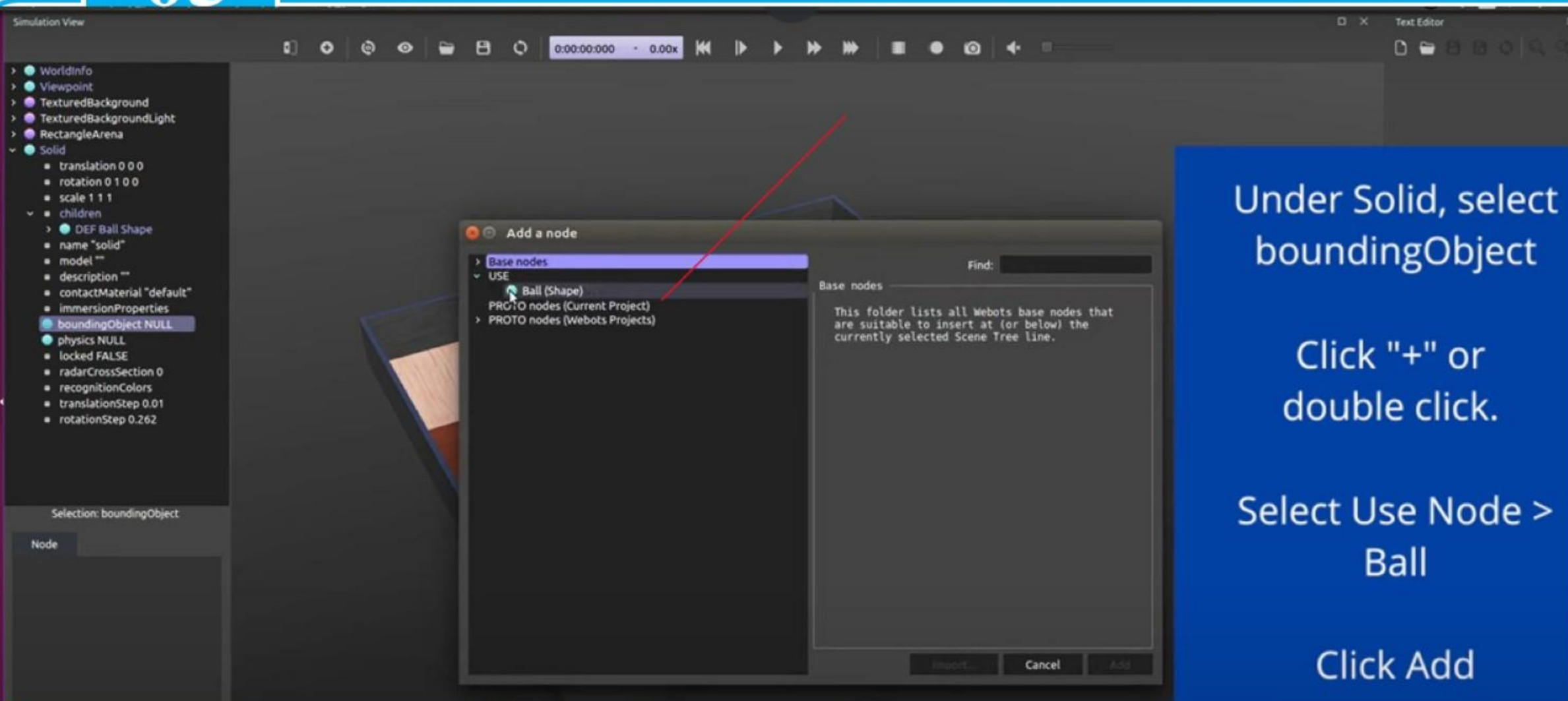
make it usable next we have to change the bounding object this is going to be the same thing as your shape so now instead of having to do it all over again you can reuse the same shape so click on shape and using def name it something that you can reuse so let's call it ball now on your bounding object when you double click you can click on use to use this shape that we just created which in this case is ball lastly let's add physics so in this case it's just a simple physics this helps with things like gravity and as you go into more advanced projects you can change this for different features



"DEF" refers to a "DEFinition" statement







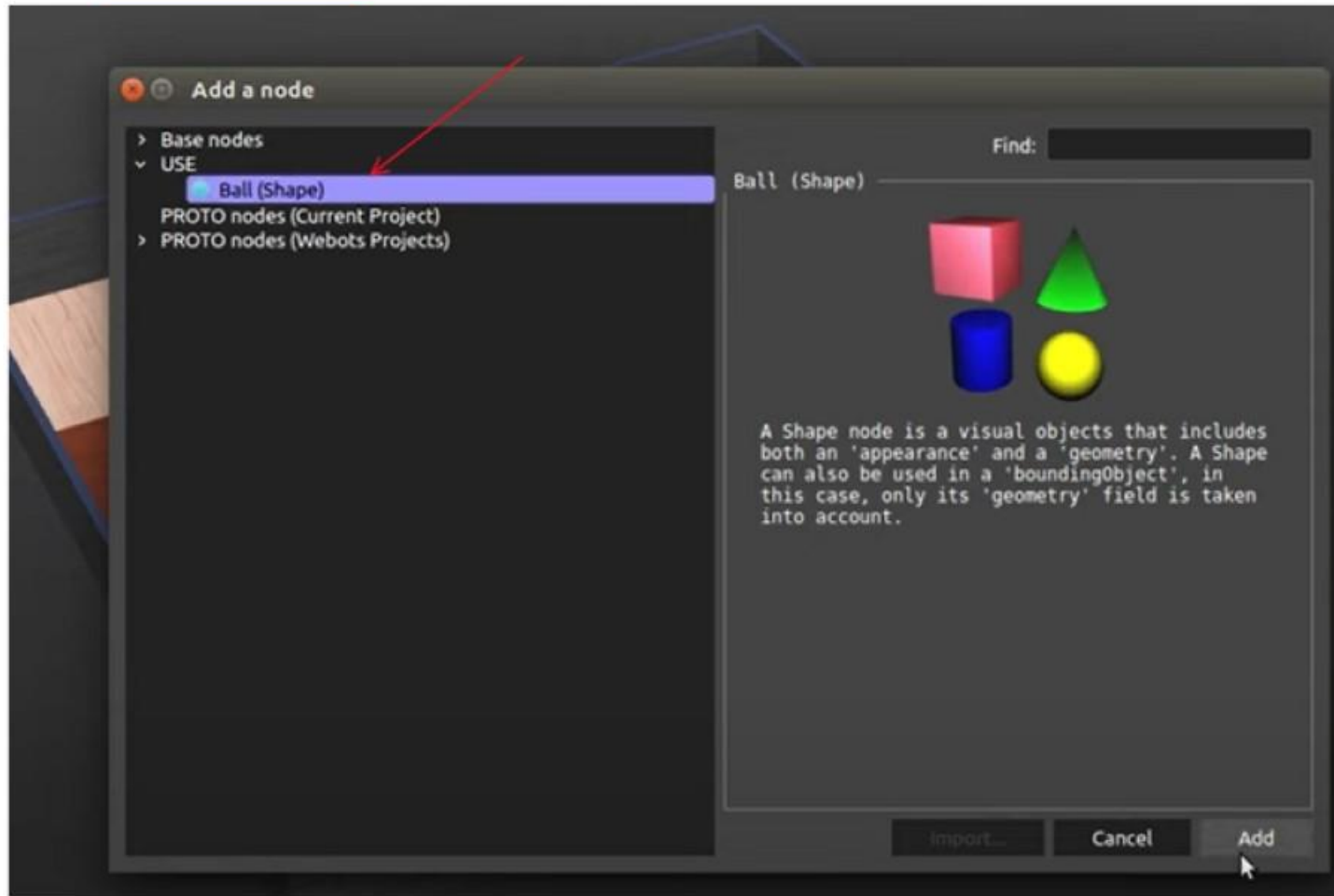
Under Solid, select
boundingObject

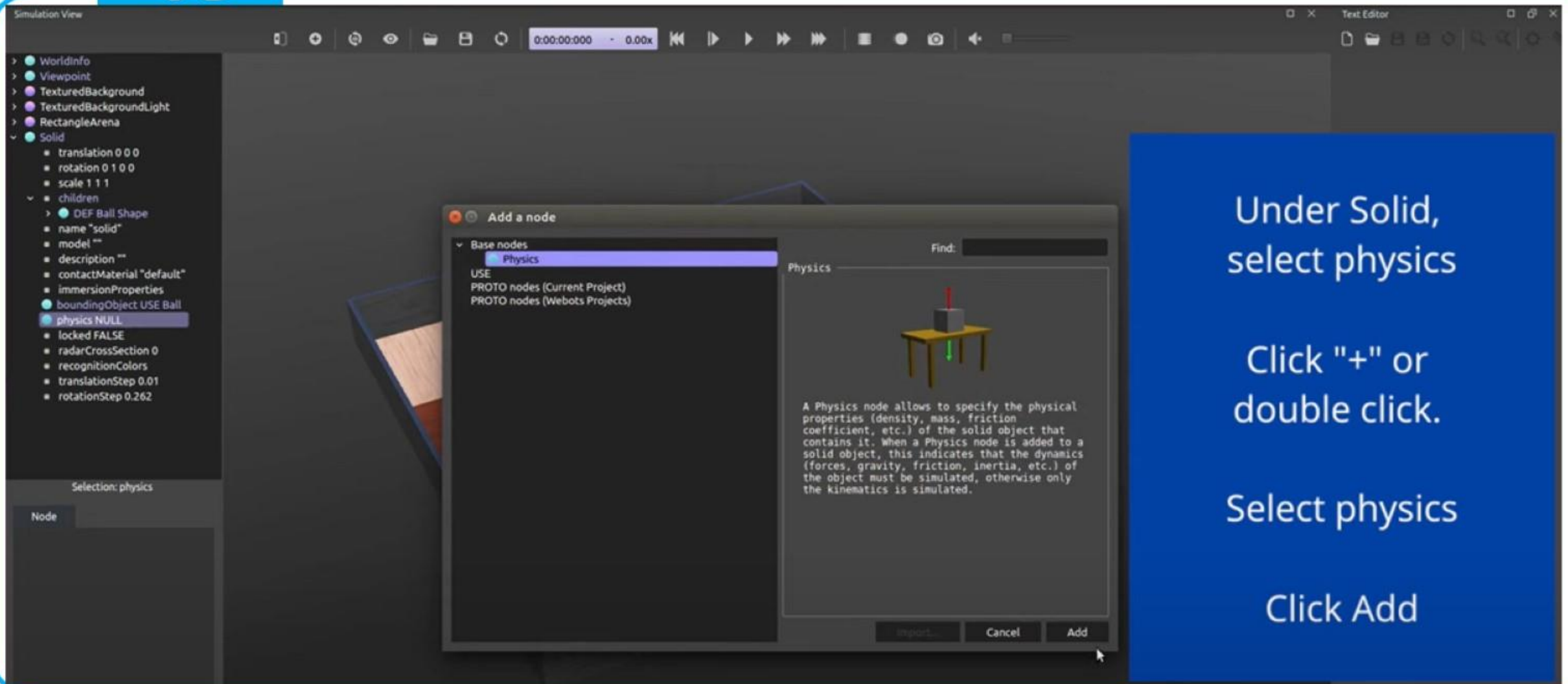
Click "+" or
double click.

Select Use Node >
Ball

Click Add

You can see the shape that you just create it





Under Solid,
select physics

Click "+" or
double click.

Select physics

Click Add

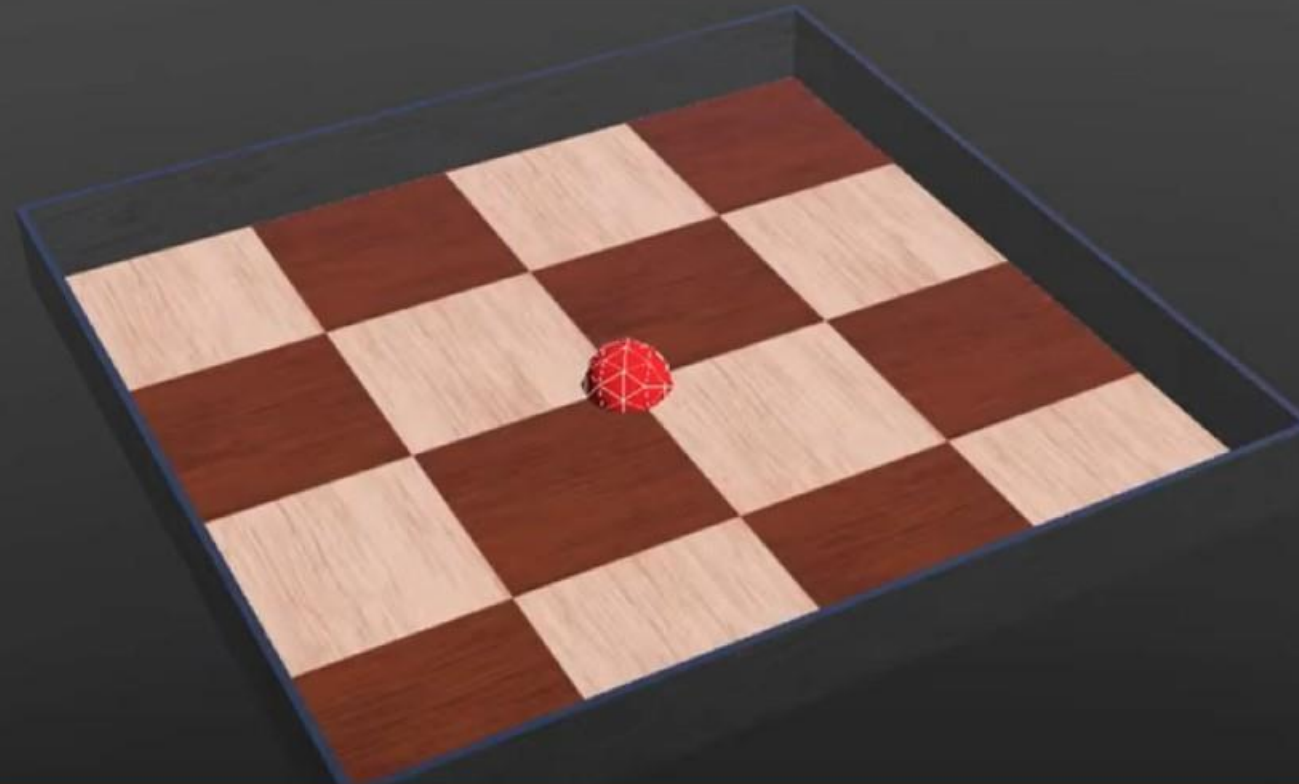
Now we will add a simple physics:

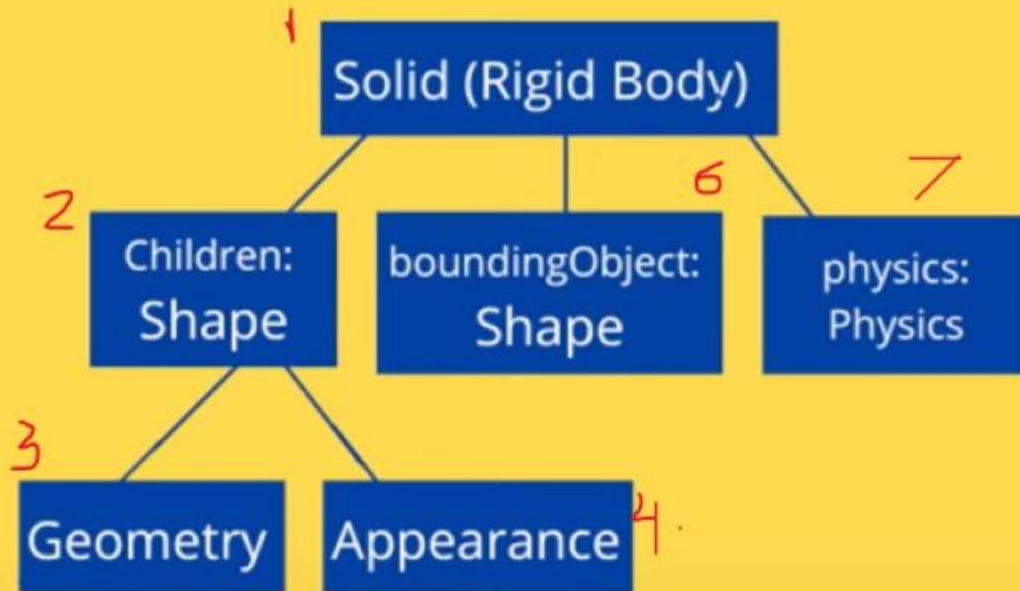
- > WorldInfo
- > Viewpoint
- > TexturedBackground
- > TexturedBackgroundLight
- > RectangleArena
- > Solid
 - translation 0 0 0
 - rotation 0 1 0 0
 - scale 1 1 1
 - children
 - > DEF Ball Shape
 - name "solid"
 - model ""
 - description ""
 - contactMaterial "default"
 - immersionProperties
 - boundingObject USE Ball
- > physics Physics
 - locked FALSE
 - radarCrossSection 0
 - recognitionColors
 - translationStep 0.01
 - rotationStep 0.262

Selection: physics

Node

DEF:





Creating an object

- (1) Start with a solid node.
- (2) Add a child node to give it shape.
- (3) Use geometry to choose the shape.
- (4) Add appearance and modify it.
- (5) Name shape using DEF for reuse.
- (6) Use DEF shape as bounding object
- (7) Add physics.

Creating an object in Webots involves a series of steps within its graphical interface, focusing on hierarchical node management and property adjustments. Here's a concise overview of the process:

1.Start with a Solid Node:

1. The initial step is to create a Solid node, which acts as a representation of a rigid body within the simulation environment. This is done by clicking the plus icon and navigating to "Base Nodes" > "Solid" to add a Solid node.

2.Add a Child Node (Shape):

1. Within the Solid node, you add a child node to define the visual and physical form of your object. This is achieved by adding a "Shape" node as a child of the Solid node.

3.Define the Geometry:

1. Under the Shape node, you specify the object's geometry. If you're creating a sphere, for instance, you select "Geometry" and then choose the "Sphere" option. This geometry can be customized further by modifying parameters such as the sphere's radius to alter its size.

4. Set the Appearance:

To define the object's visual characteristics, you click on "Appearance" and choose "PBRAppearance" as the base node. PBR (Physically Based Rendering) Appearance allows you to adjust visual properties like color, roughness, and metalness to achieve the desired look.

5. Define the Object for Reuse (DEF):

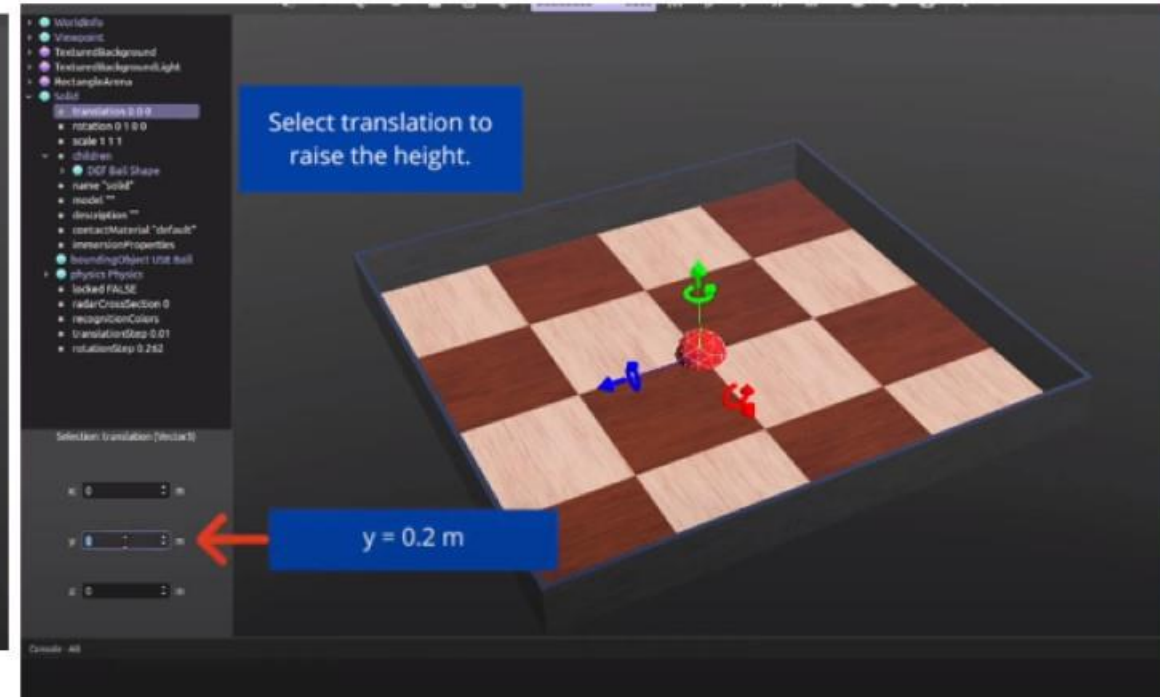
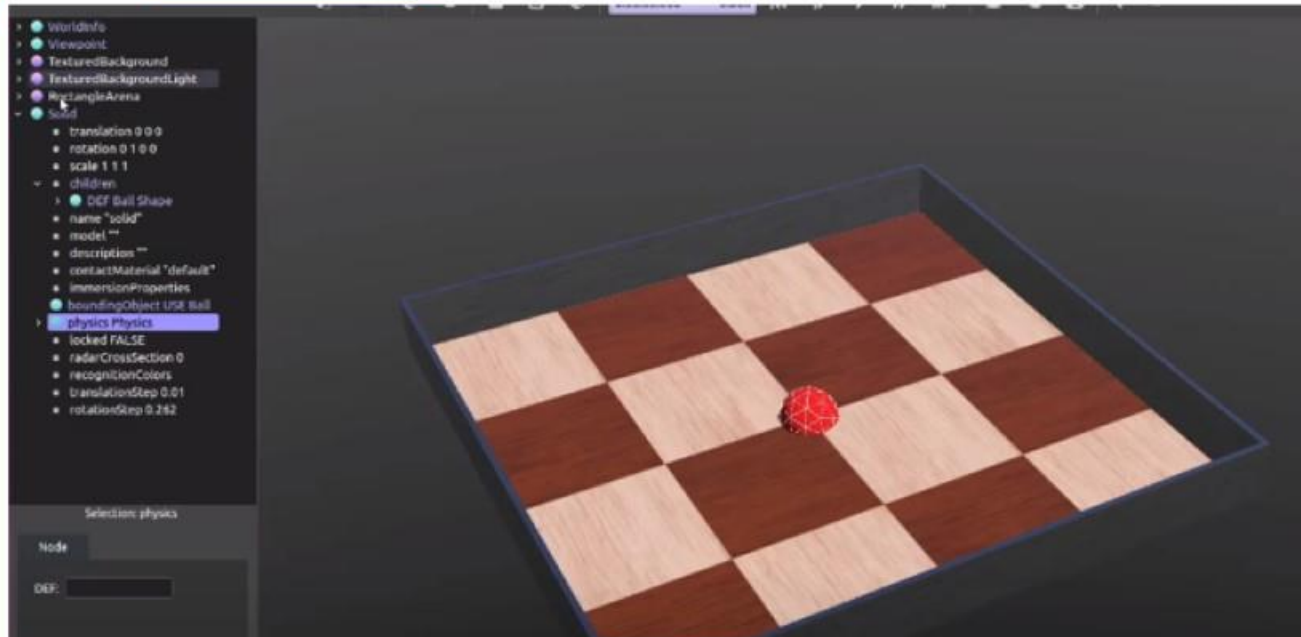
If you intend to reuse the shape within your simulation, you can define it as an object using the "DEF" keyword. This step involves giving a unique name to your shape, which allows you to reference and reuse it elsewhere in your simulation.

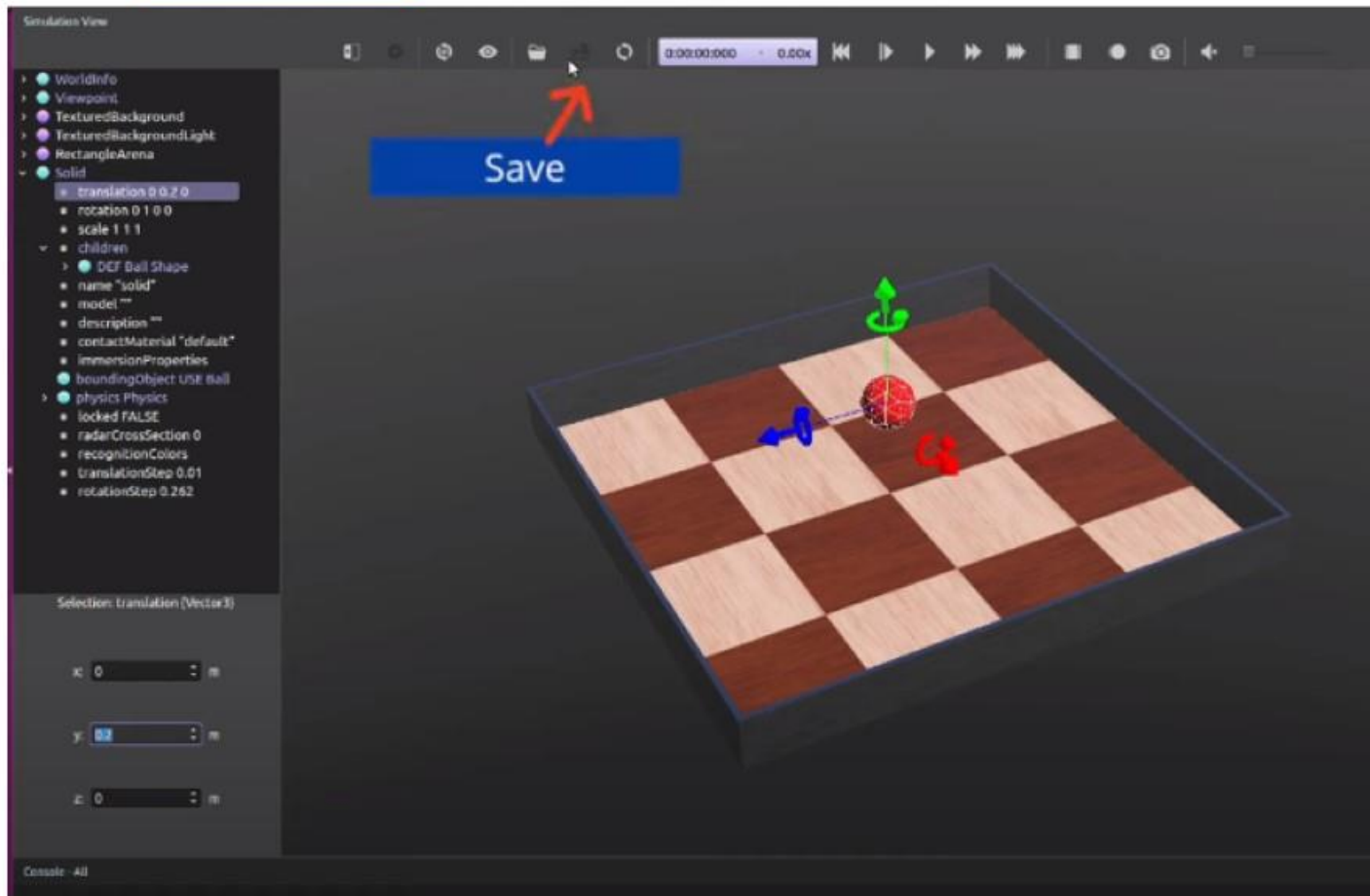
6. Specify the Bounding Object:

The bounding object is crucial for defining the collision boundaries of your object. You can set the Shape node you've already defined as the bounding object to ensure that collision detection works as expected within the simulation environment.

7. Add Physics:

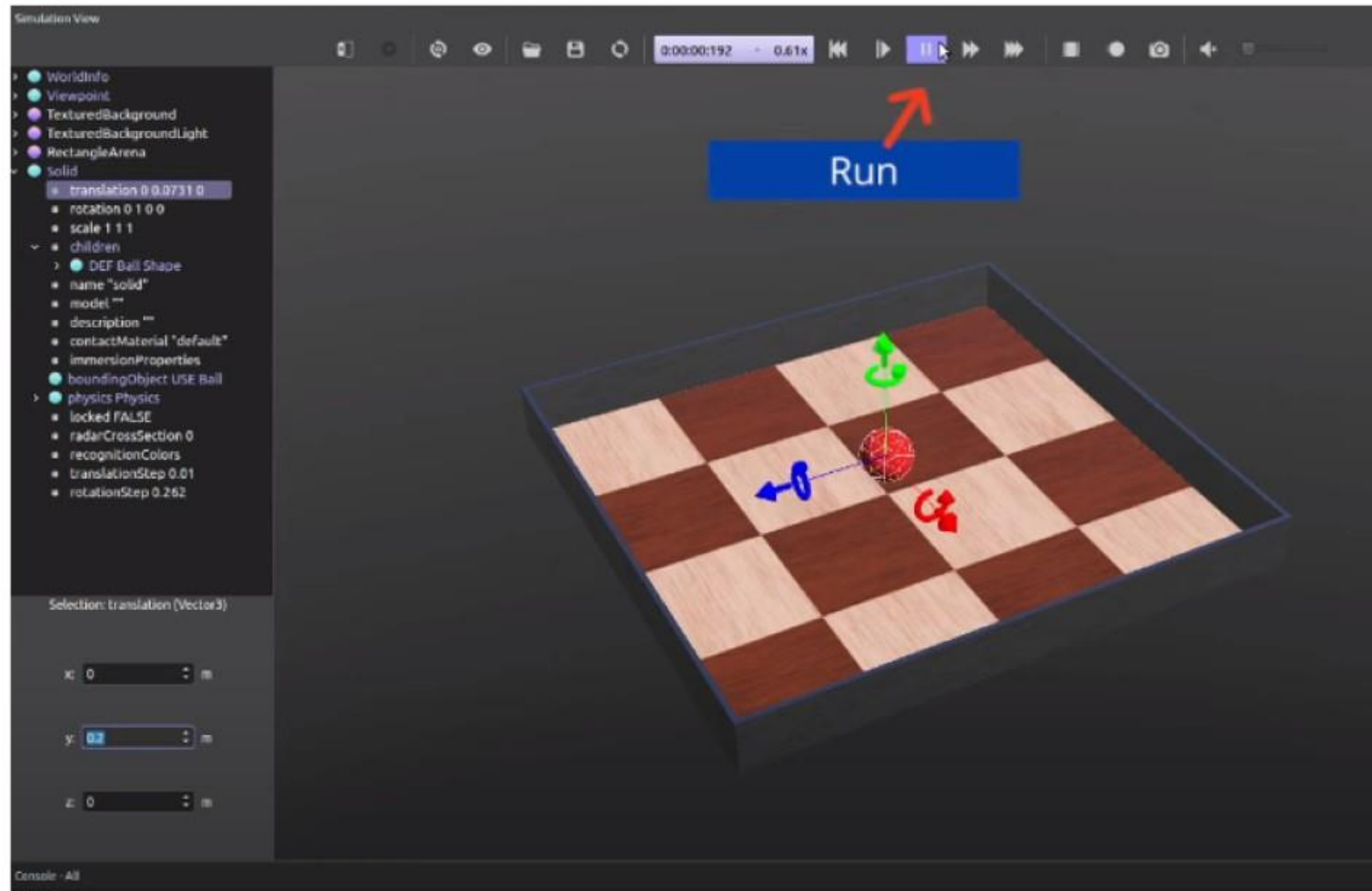
To make your object dynamic (capable of moving and interacting with other objects and the environment), you add physics properties to it. This distinguishes it from static elements in the simulation, like a rectangular arena, by enabling it to have physical behaviors such as bouncing, rolling, or colliding.





RUN the simulation:

The ball will drop on the floor, because the flood is static



What Are Social Robots?



Understanding Social Robots

Definition of social robots as machines designed to engage with humans in a social and interactive manner. Introduction to the concept of robots not just performing tasks but also relating to humans on a social level.



<https://www.aventine.org/robotics/history-of-robotics>



Overview of the sensors and AI technologies that enable social robots to perceive and understand human emotions, intentions, and social cues.

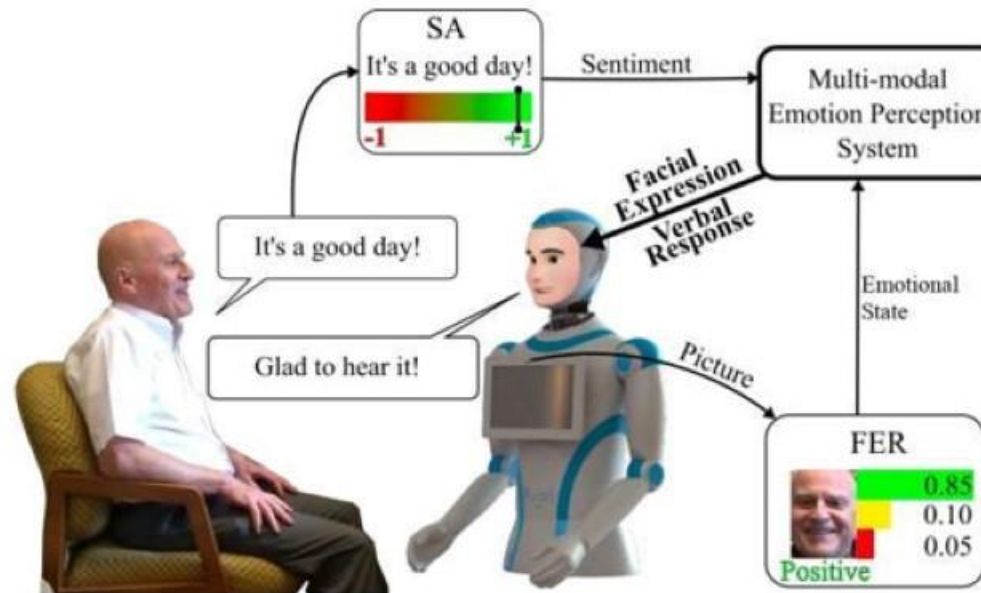


Fig. 1. Using a multimodal emotion perception system to interpret the input modalities and output appropriate responses in a multimodal emotion expression system (SA: Sentiment Analysis, FER: Facial Expression Recognition).

Exploring how social robots use AI to recognize and respond to human emotions, with examples like Jibo and Zenbo.



Zenbo.



Jibo - ROBOTS

How social robots assist in healthcare settings, such as providing companionship to the elderly or support to children with autism. Example: PARO therapeutic robot.



Paro is a robotic baby harp seal designed as a therapeutic tool for use in hospitals and nursing homes.



Overview of the use of social robots in educational settings, from language tutoring to STEM education. Example: The use of robots like Cozmo in teaching coding.

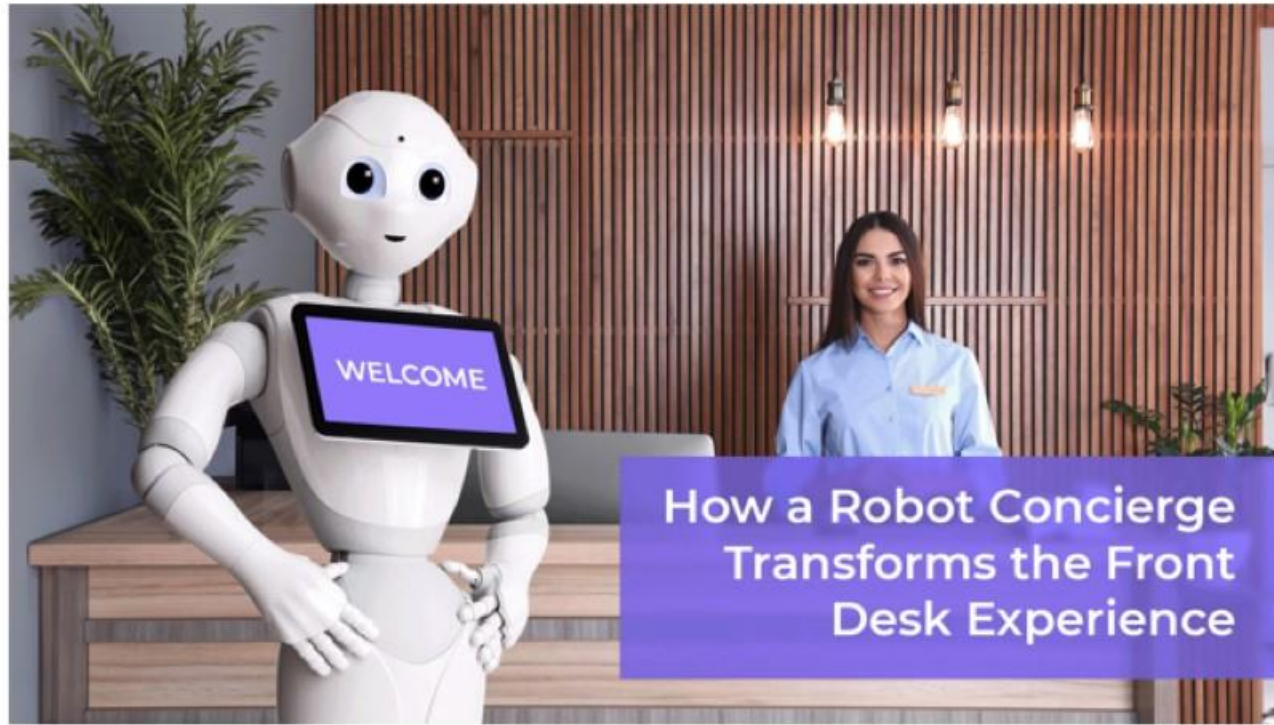
COZMO
+ Code Lab



anki



social robots in customer service and hospitality, such as greeting guests or providing information. Example: Robots in hotels and retail.



The technical, social, and ethical challenges in developing and integrating social robots into society.

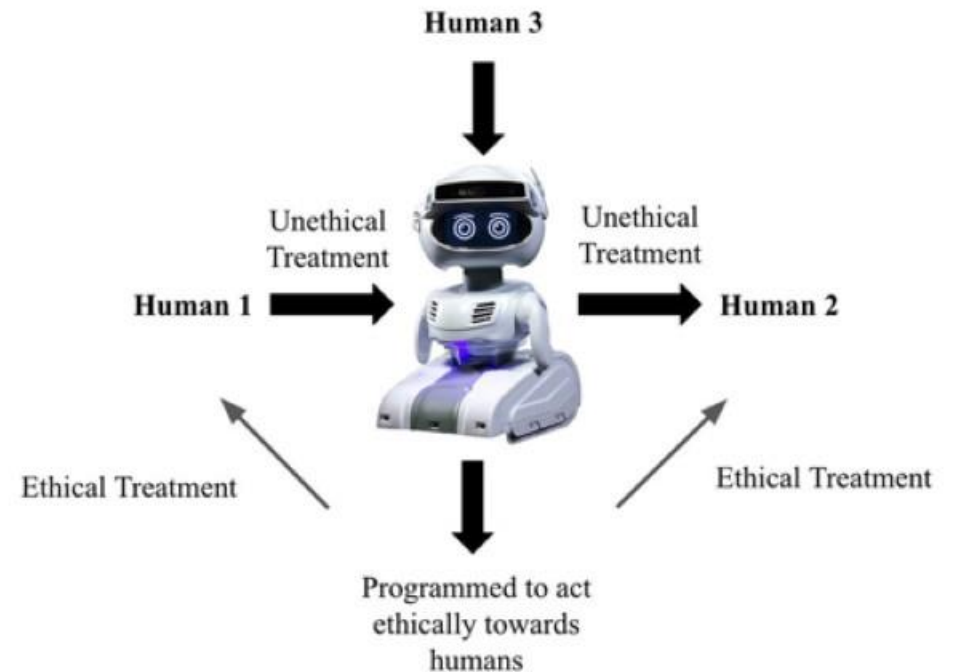


Examination of ethical issues, including privacy, autonomy, and the impact on employment. Questions of dependency and the role of robots in society.

Privacy Concerns:

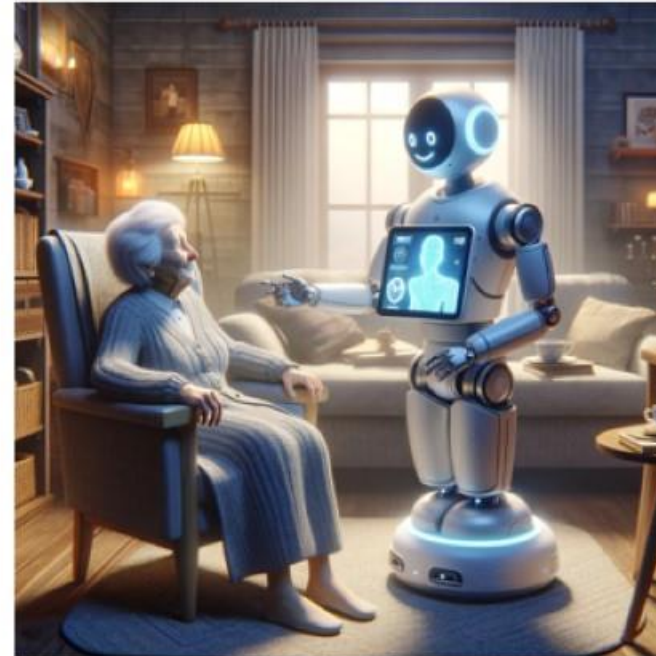
Autonomy and Human Agency:

Impact on Employment and Social Structures:



Privacy Concerns:

A home assistant robot that learns household routines and personal preferences could unintentionally expose sensitive information without robust data protection measures.



Autonomy and Human Agency:

Robots in elderly care settings making choices for residents about daily activities, thereby reducing their independence and decision-making capabilities.



Impact on Employment and Social Structures:

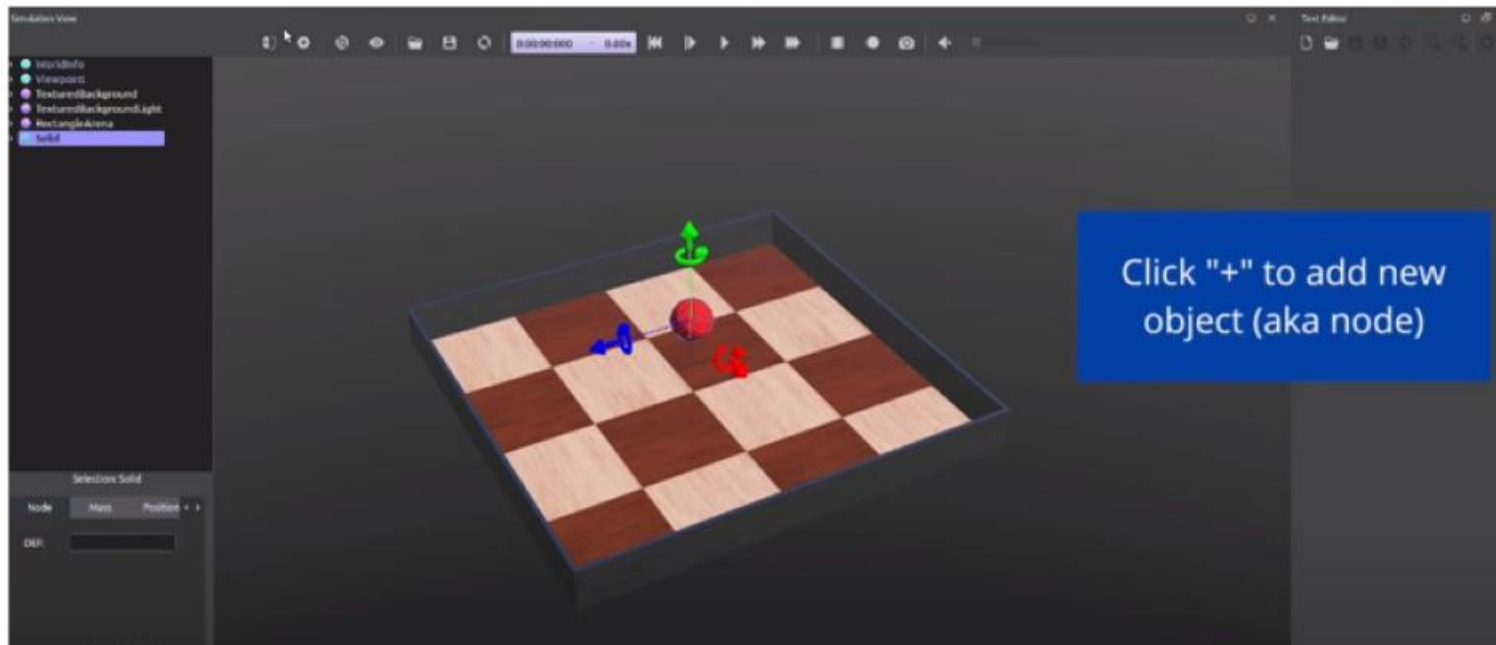
The use of robots for teaching or caregiving roles could alter employment opportunities in these sectors, necessitating a re-evaluation of human workers' roles and skills development for future job markets.



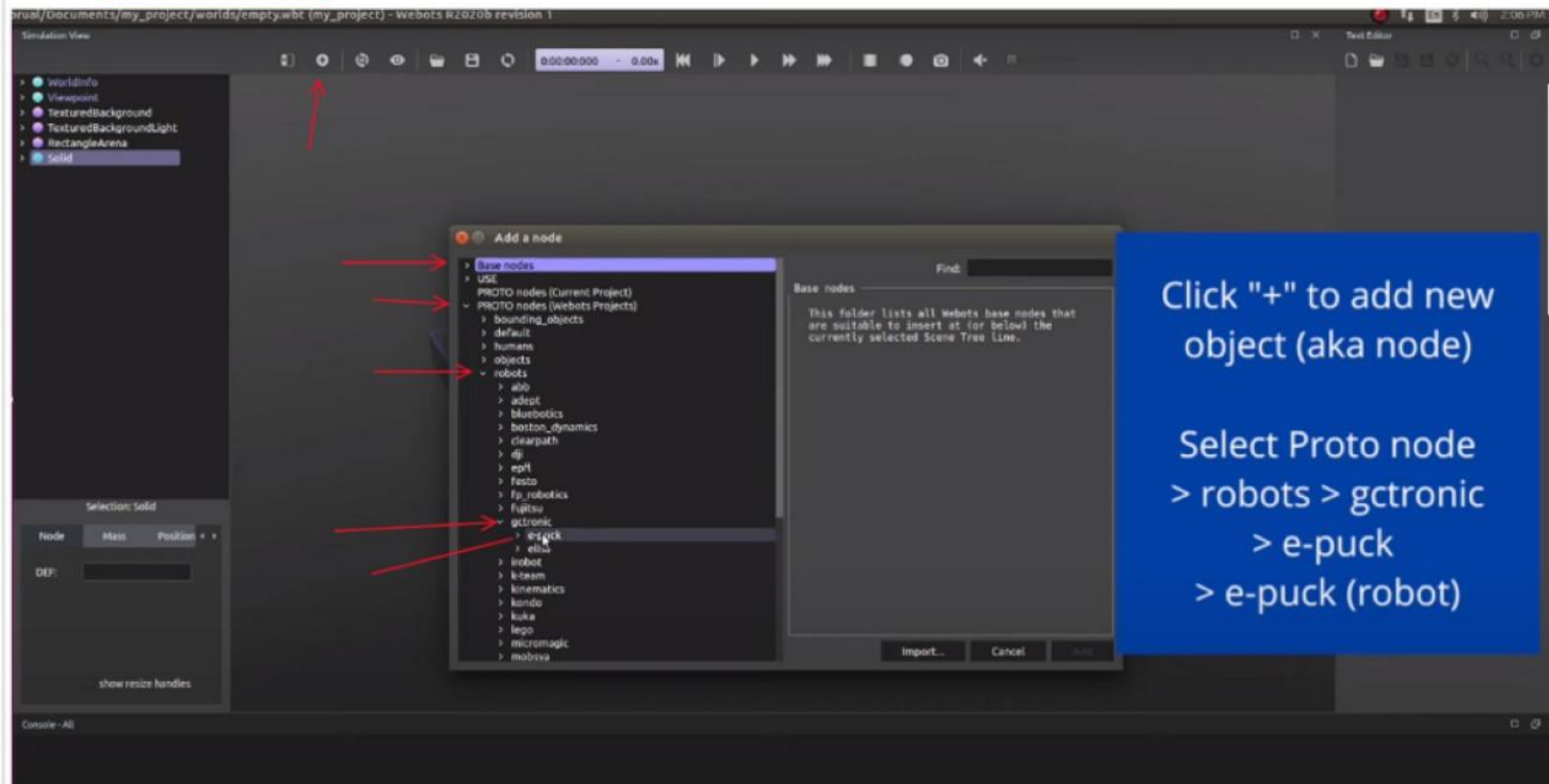
	History of HRI	
4. Strategic HRI System Design	- Core Principles and Communication Strategies in HRI	- Designing Interfaces - Communication Protocol Simulations

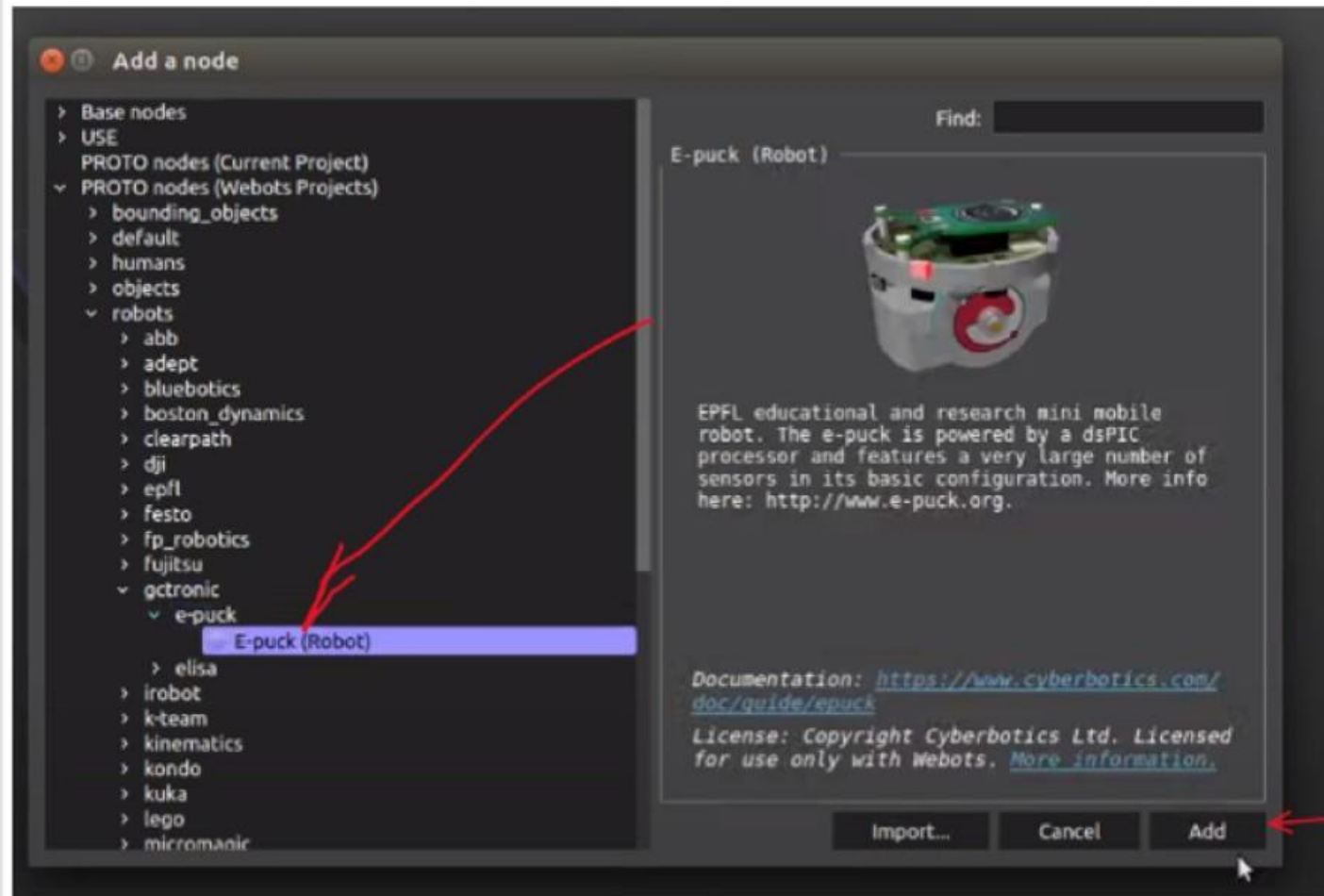
NOW using Existing Objects and ROBOTS

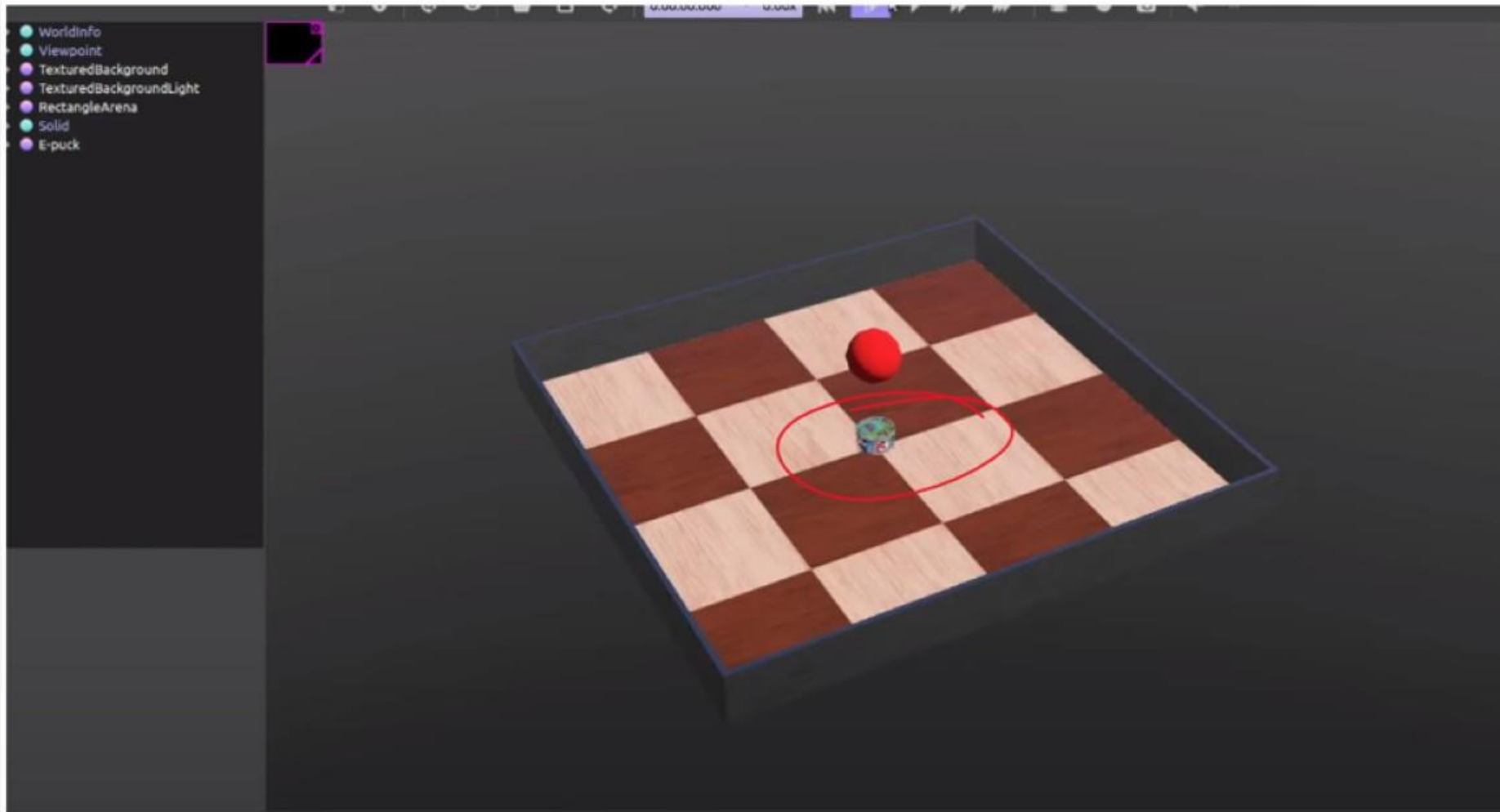




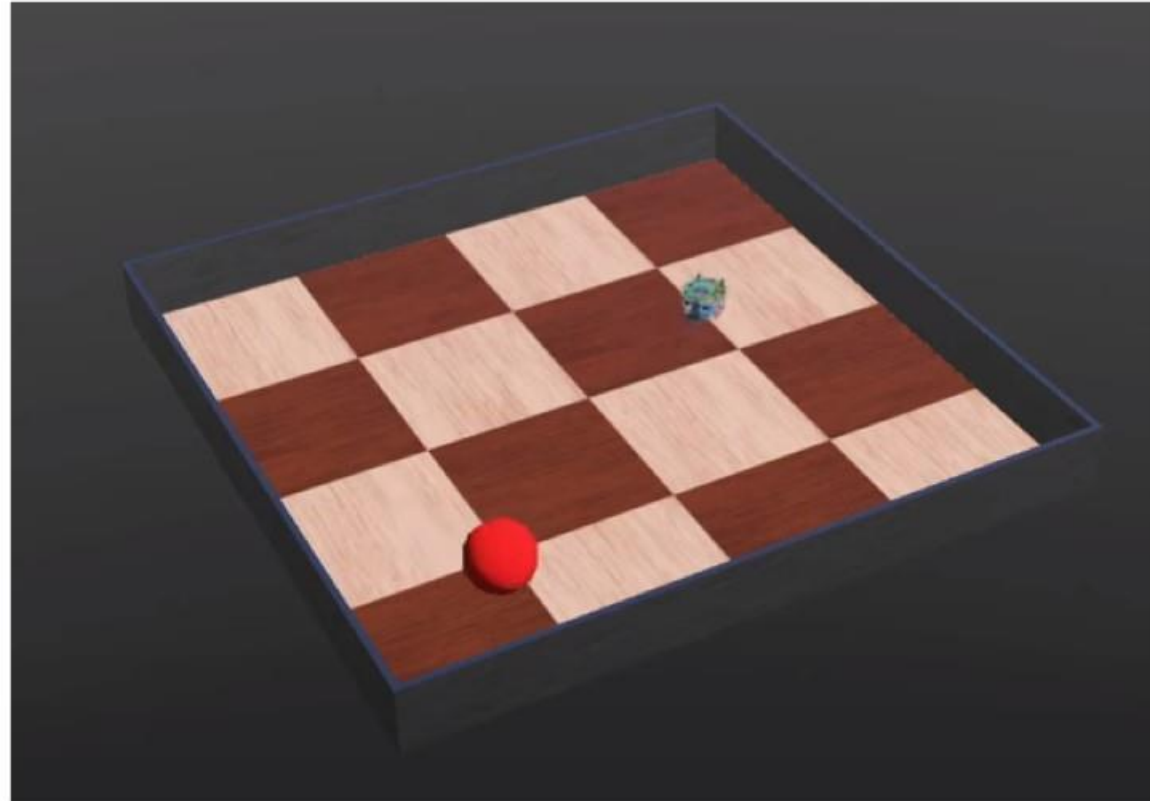
Follow the following selection sequences:

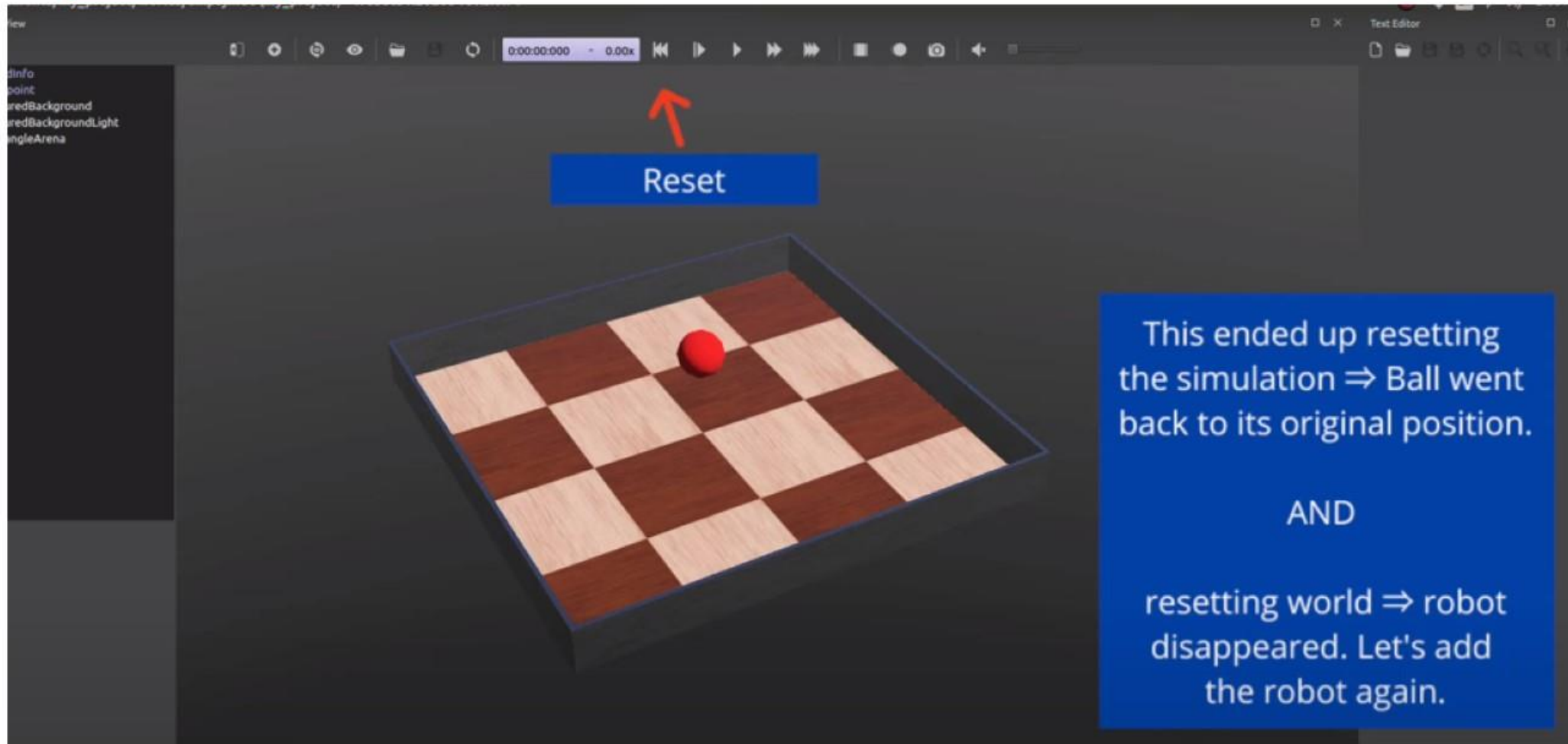






When start the simulation (RUN) the Robot continue to drive.

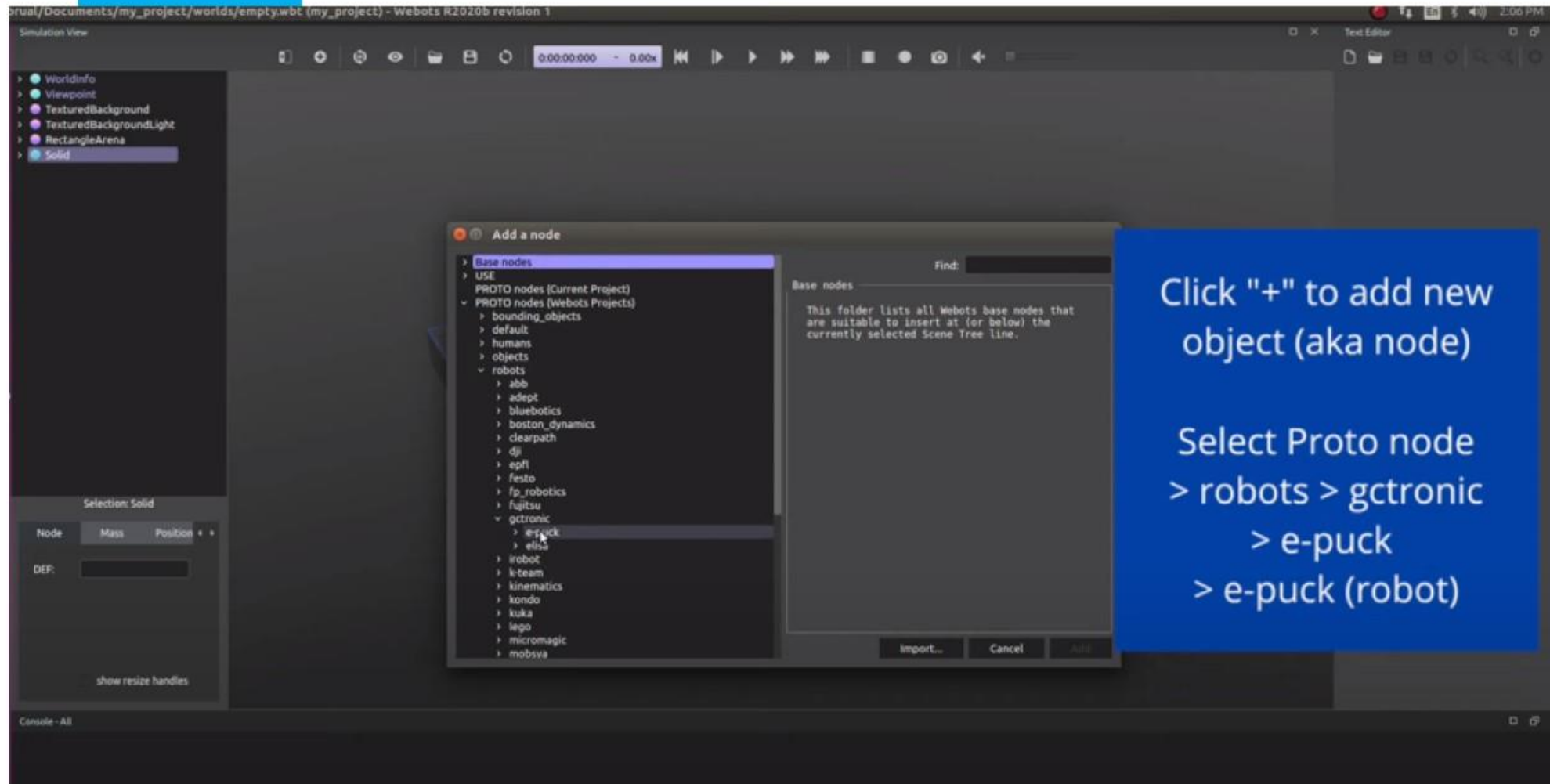


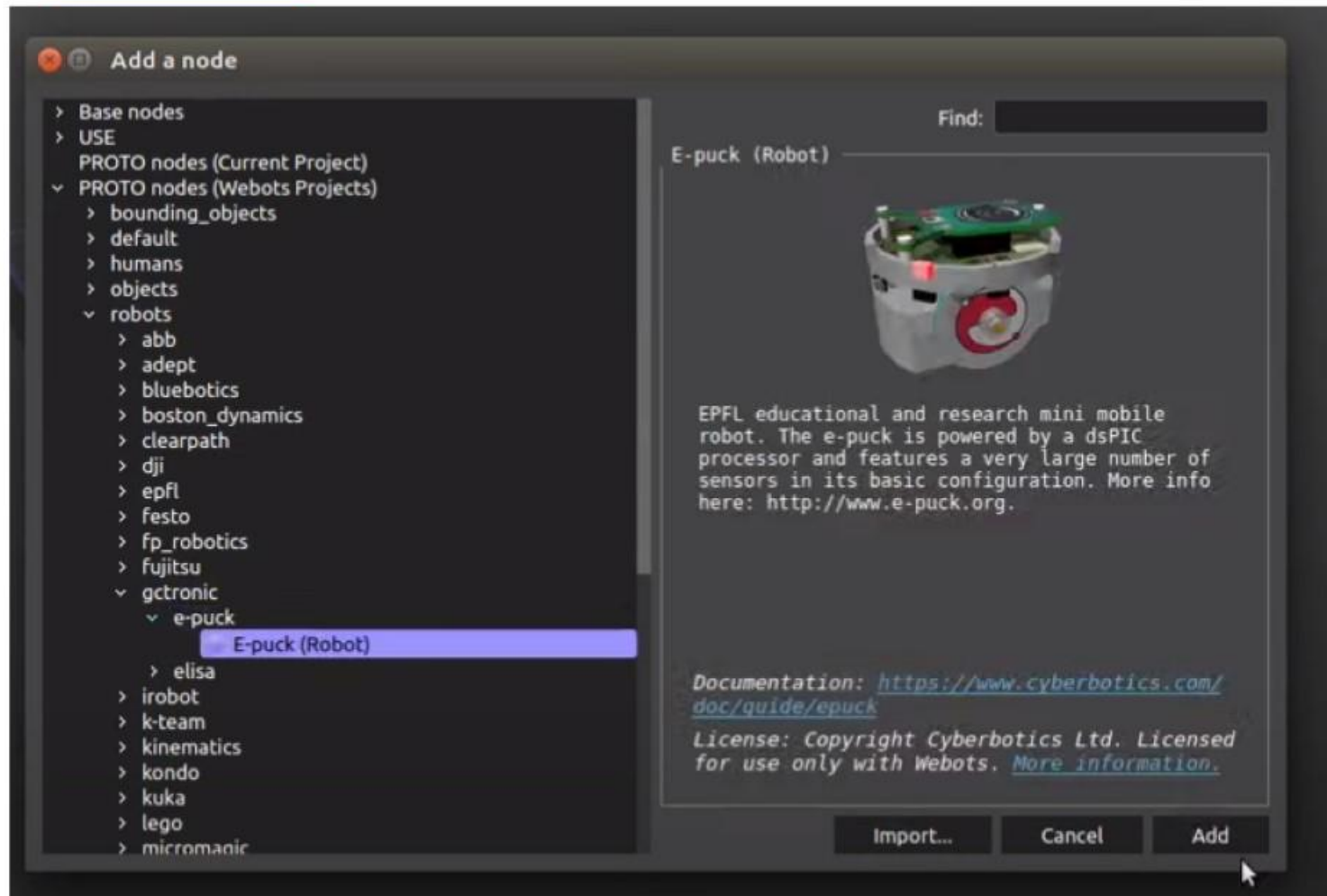


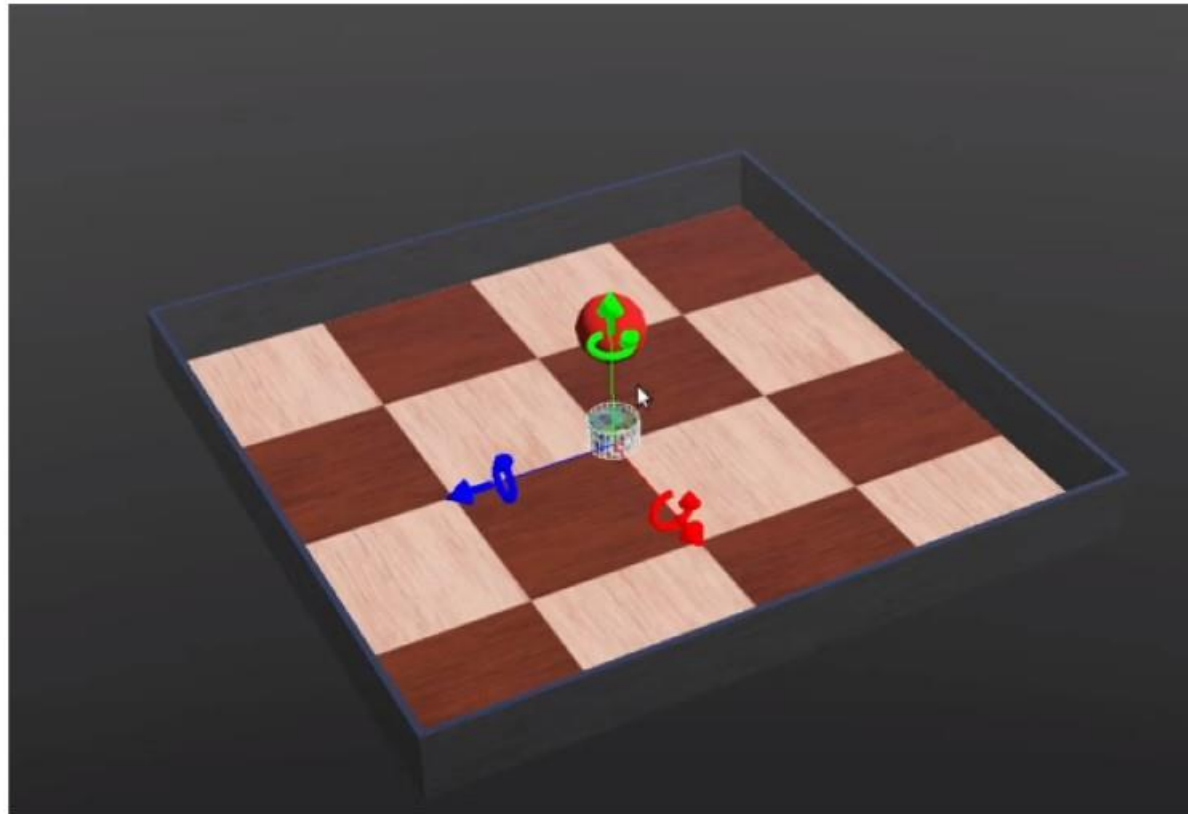
This ended up resetting the simulation $\Rightarrow$ Ball went back to its original position.

AND

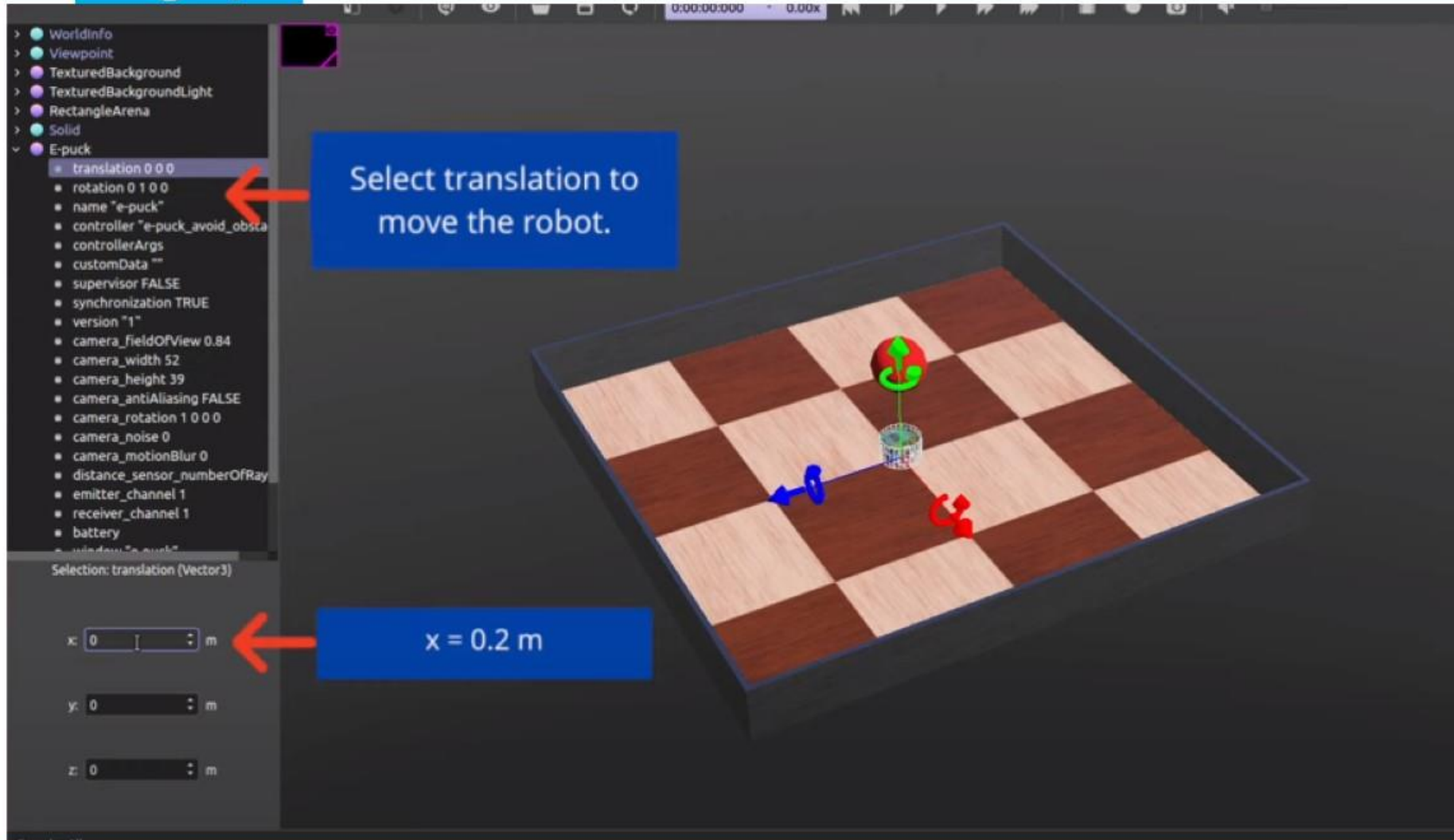
resetting world $\Rightarrow$ robot disappeared. Let's add the robot again.

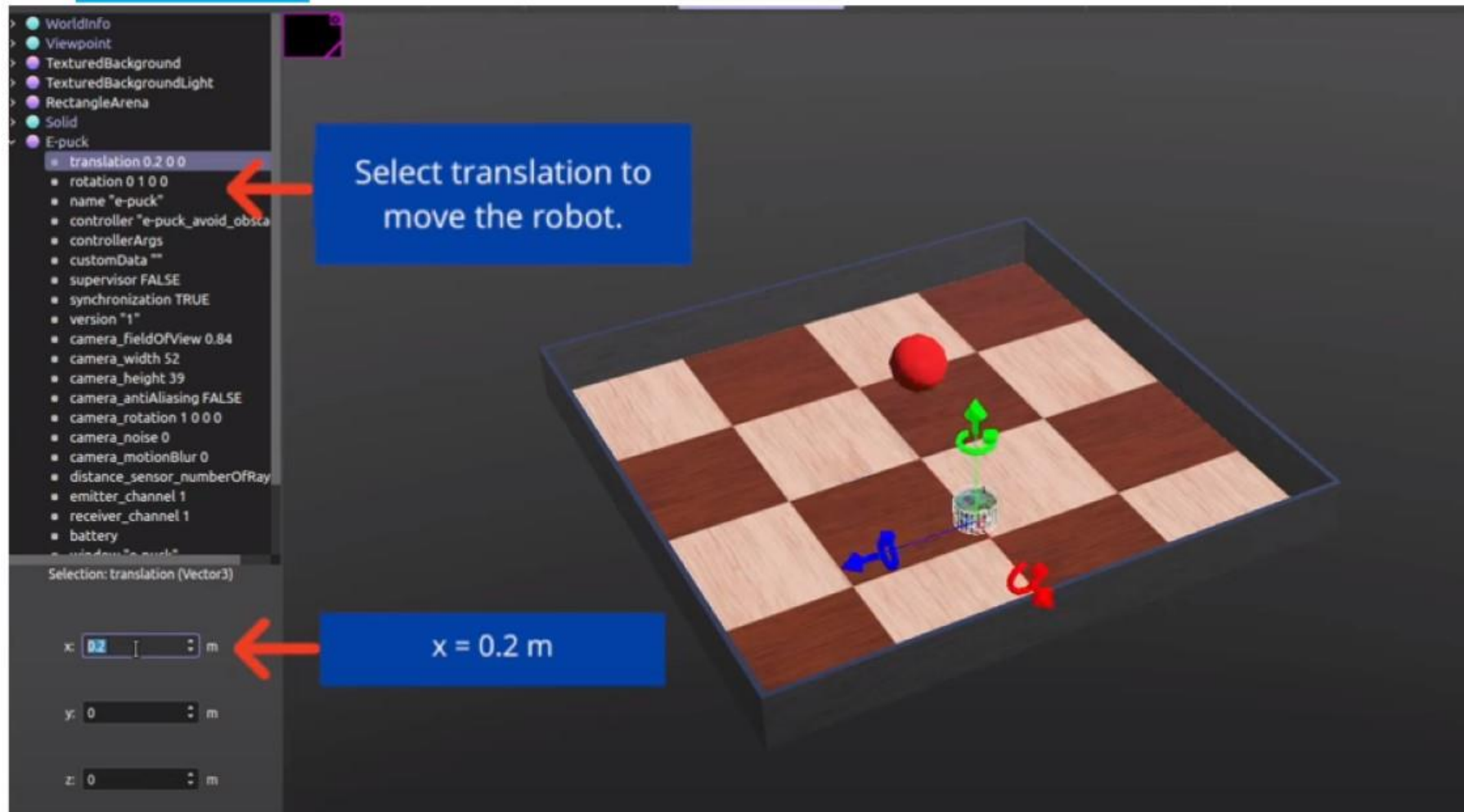


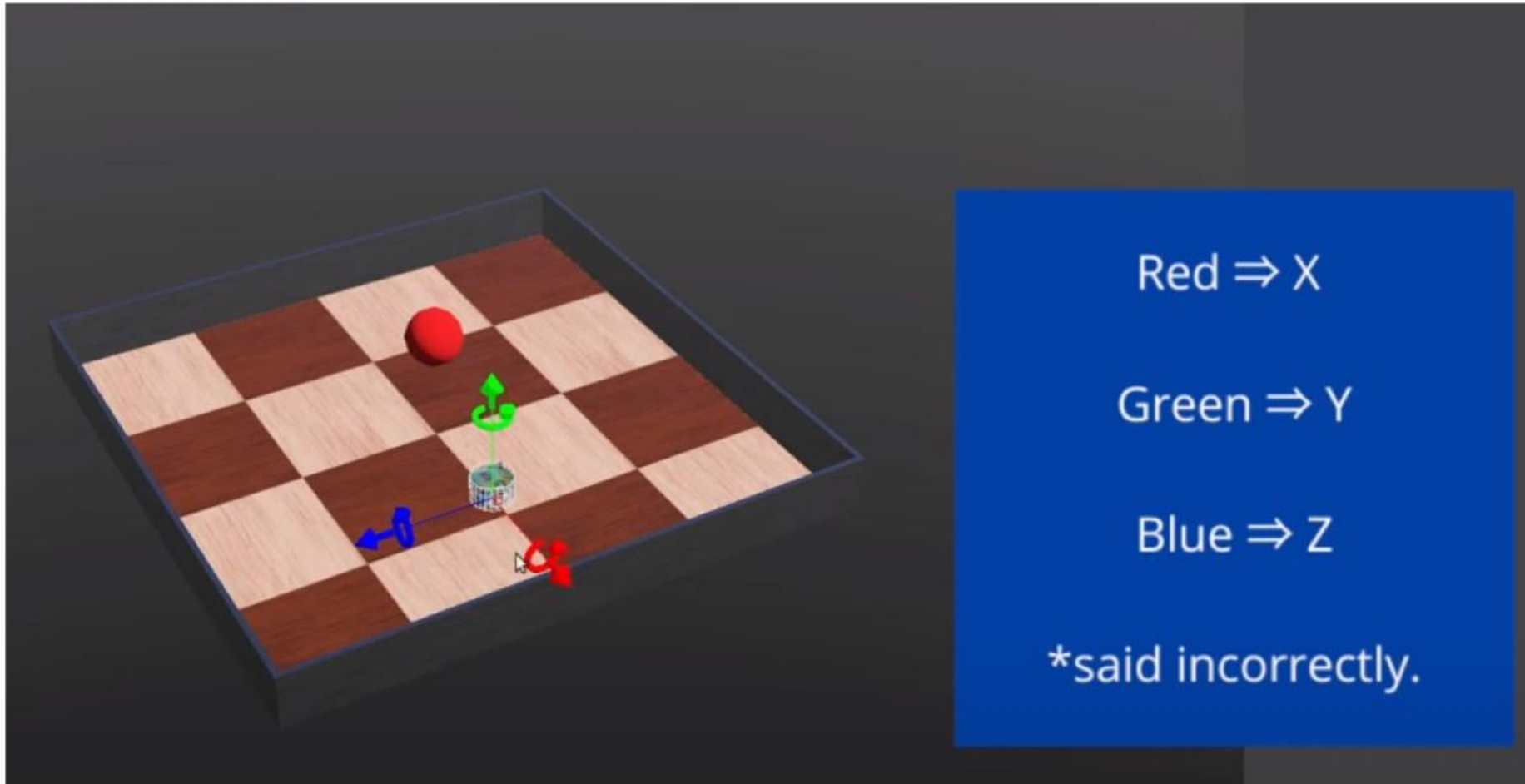


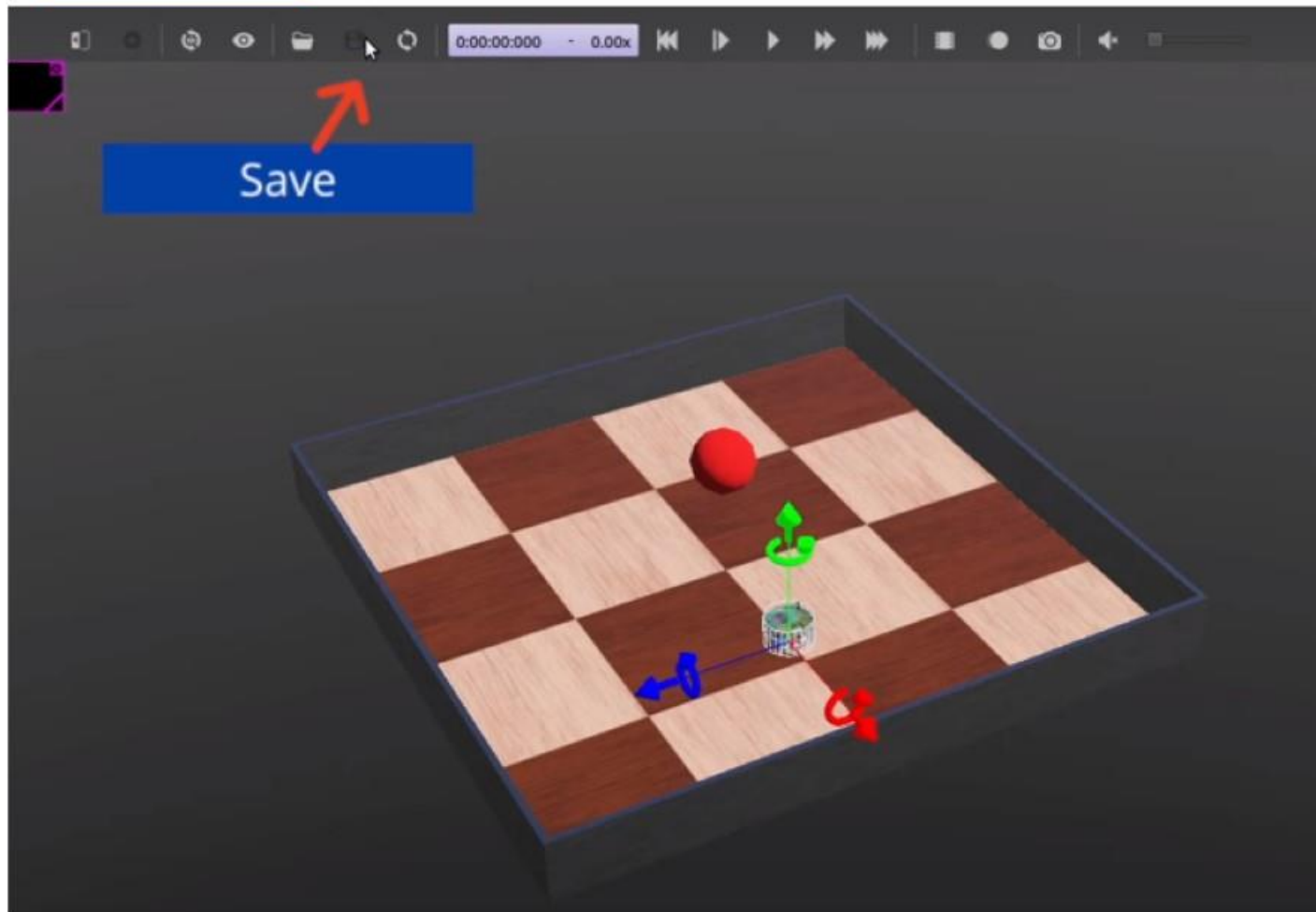


Lets shift robot away from the original point (0,0,0):





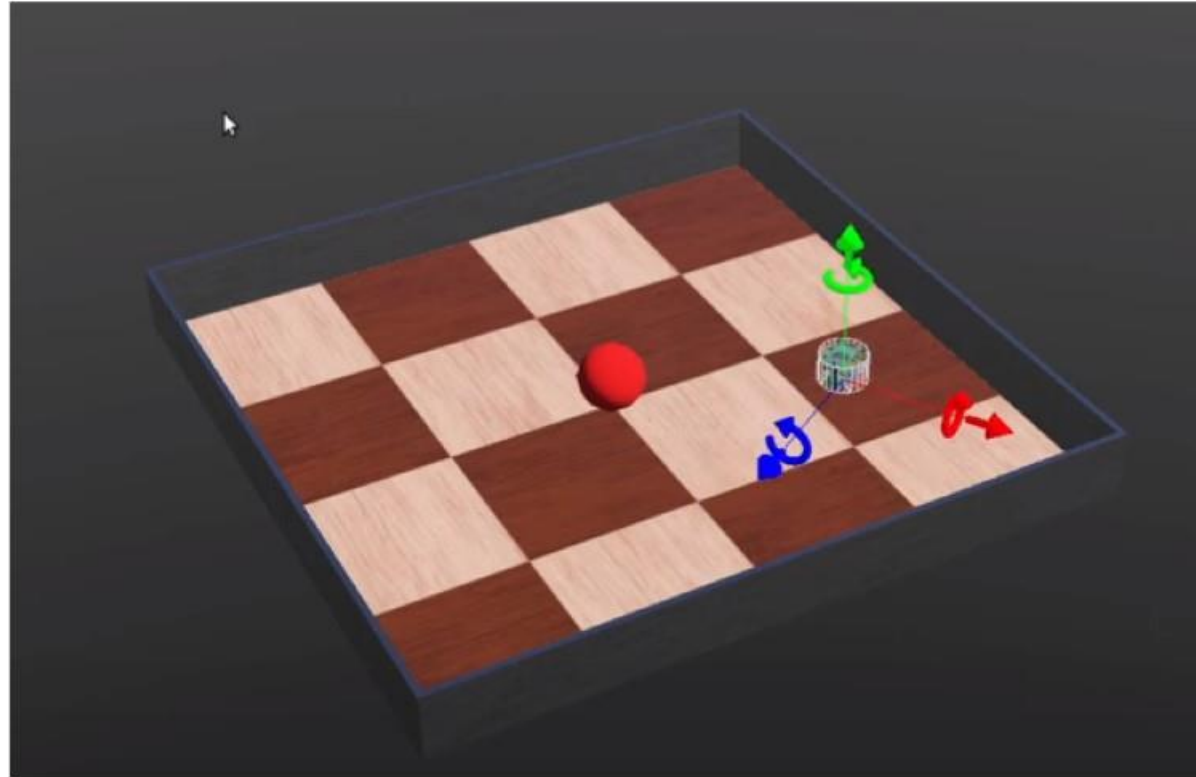




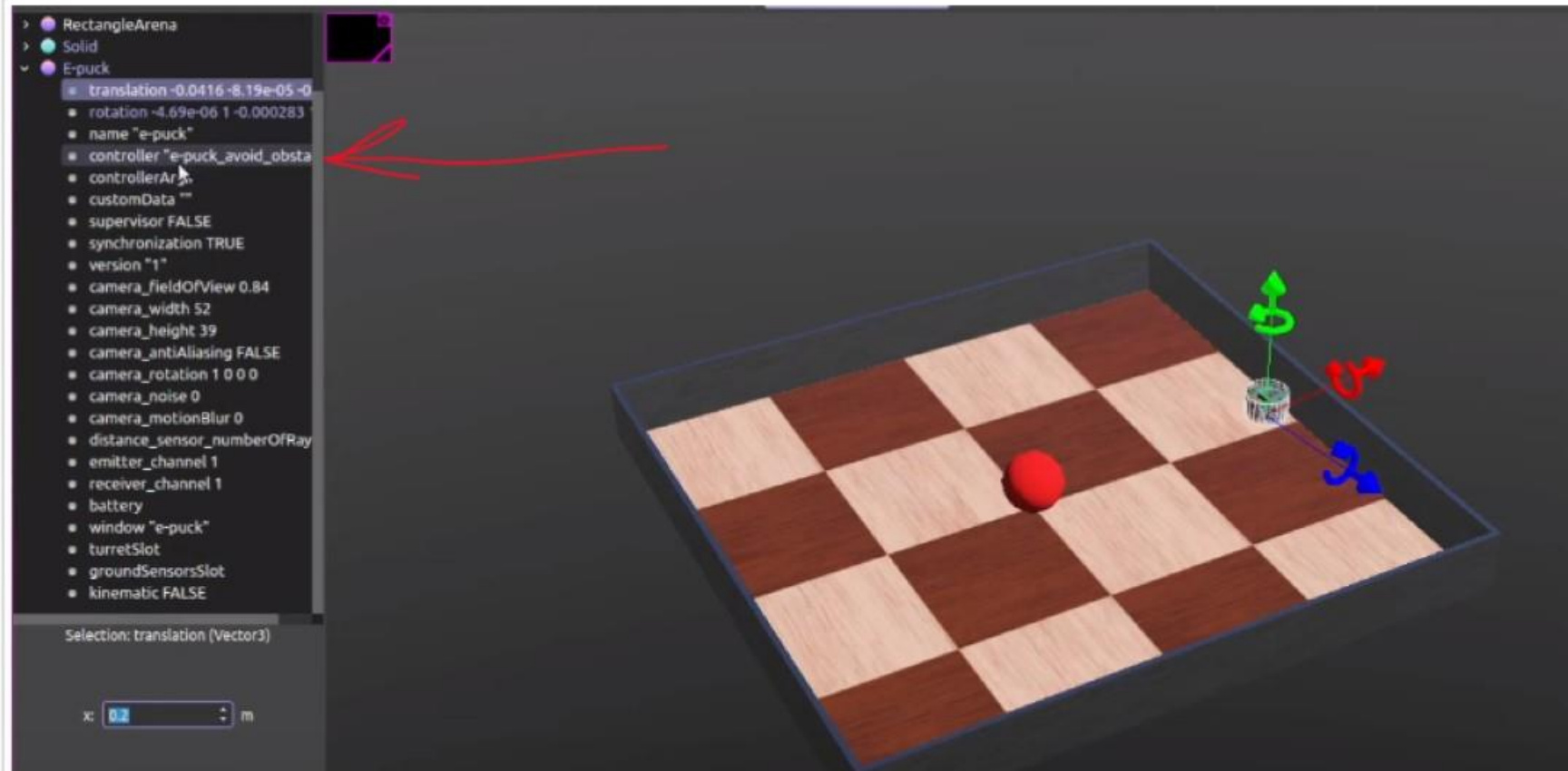
RUN the Simulation now: you will see the robot move ?

Why?

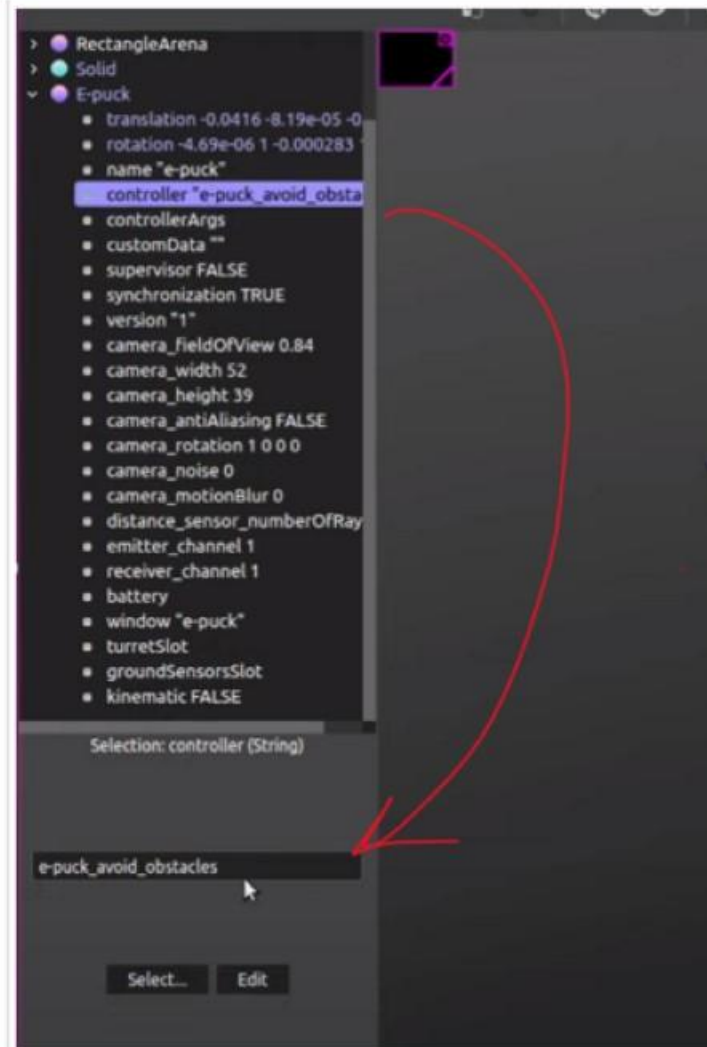
Because the robot comes with its controller.



Lets check the robot Controller:

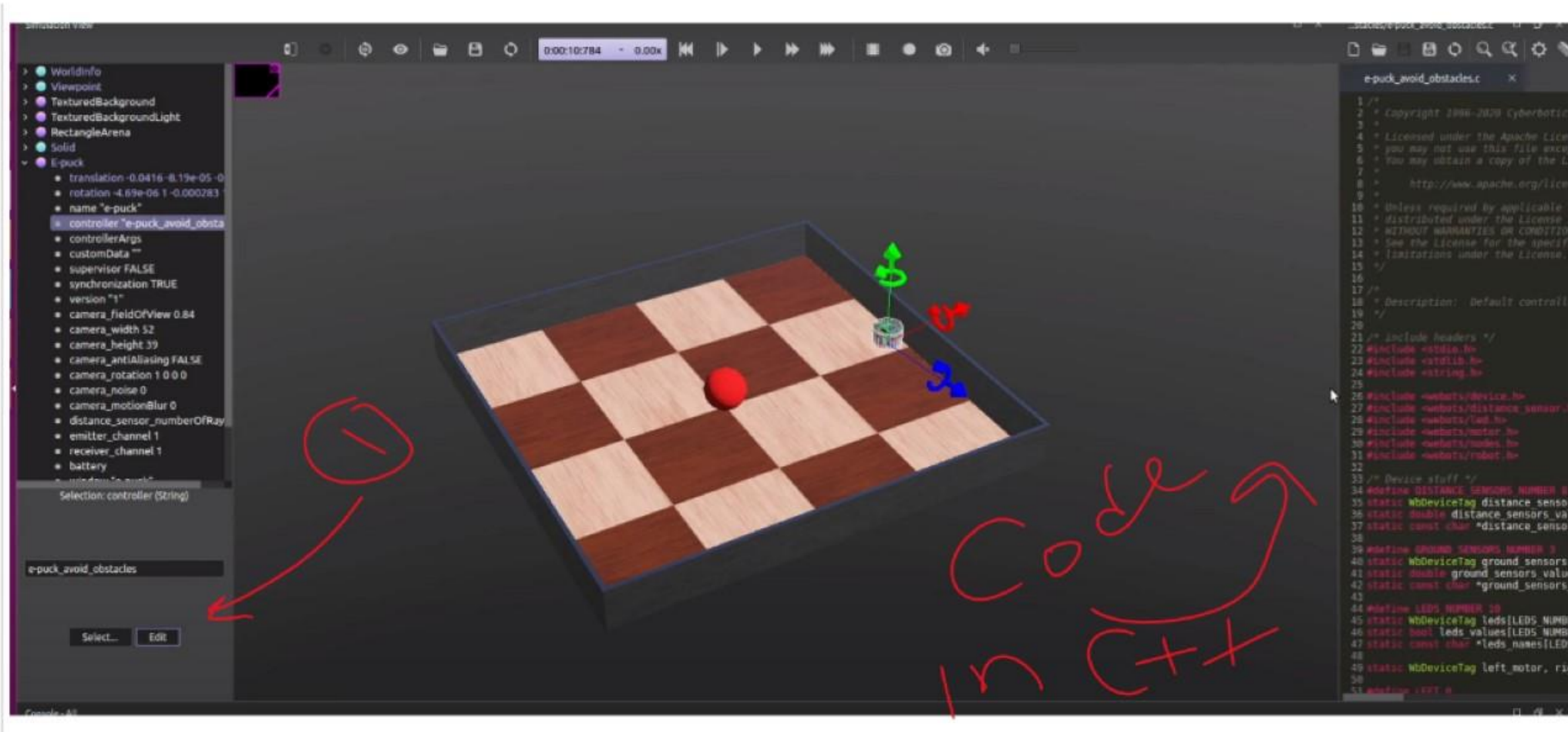


Double click on the Controller:



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.

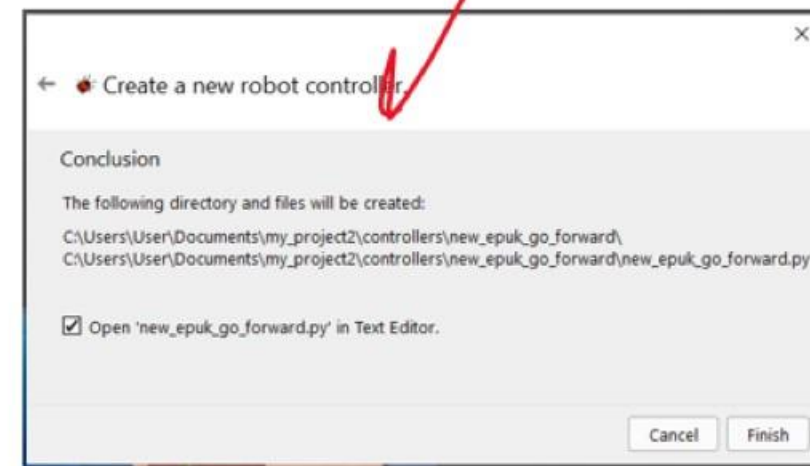
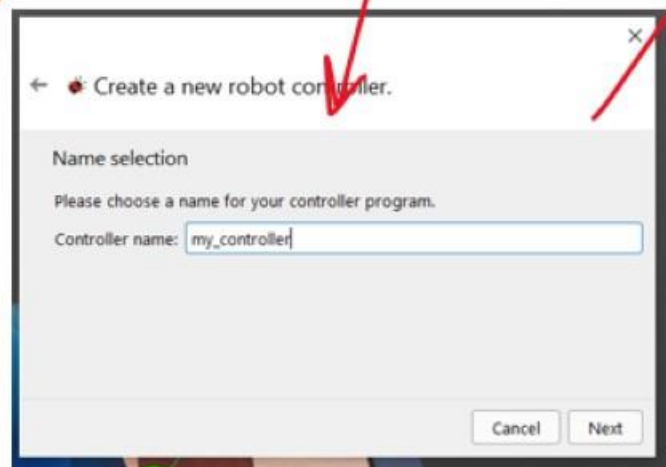
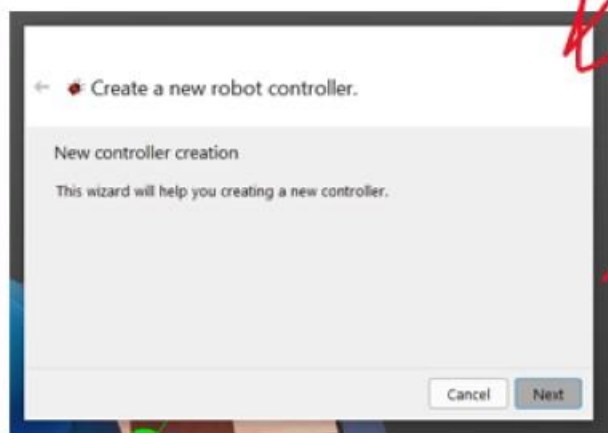
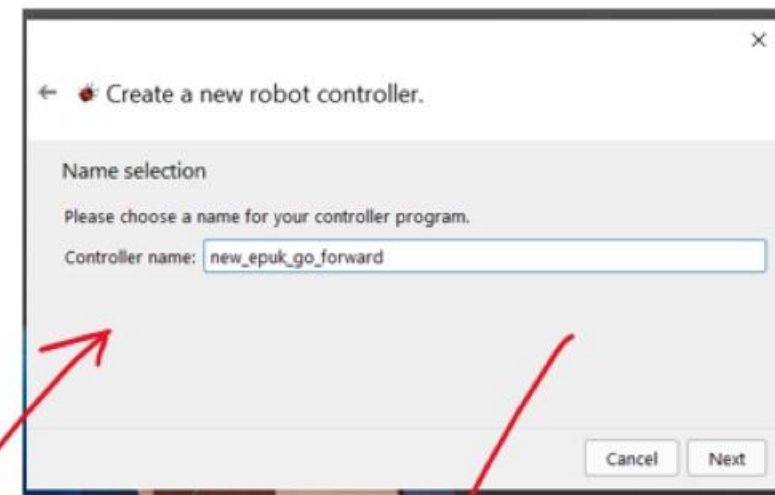
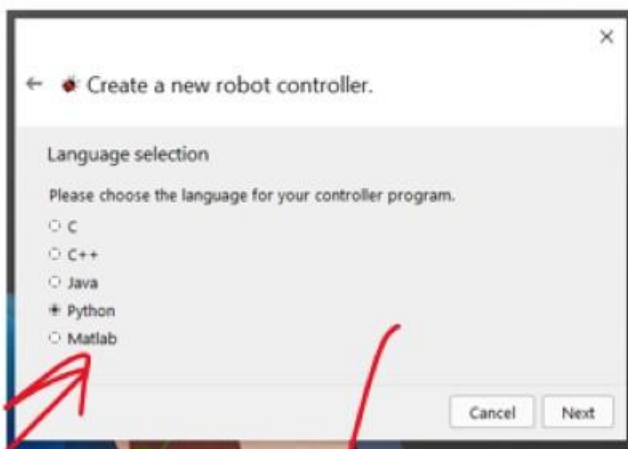
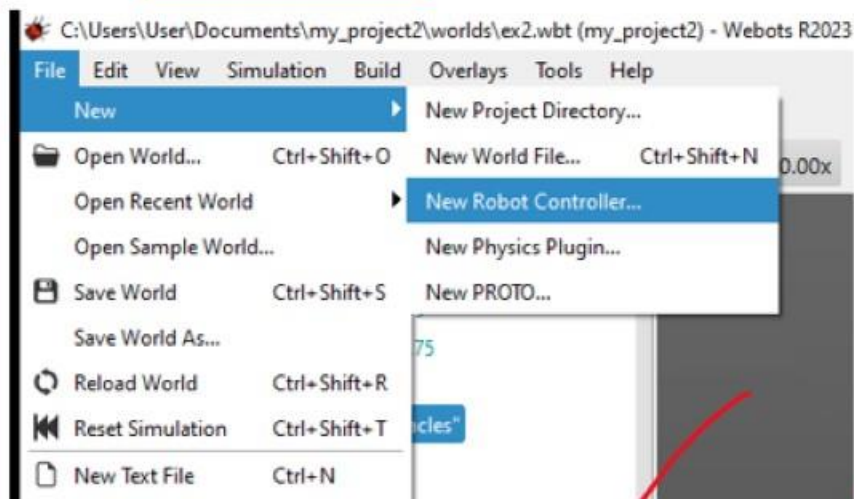
Webots controllers can be written in the following programming languages: C, C++, Java, **Python**, MATLAB, ROS, etc. C, C++ and Java controllers need to be compiled before they can be run as robot controllers.

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.

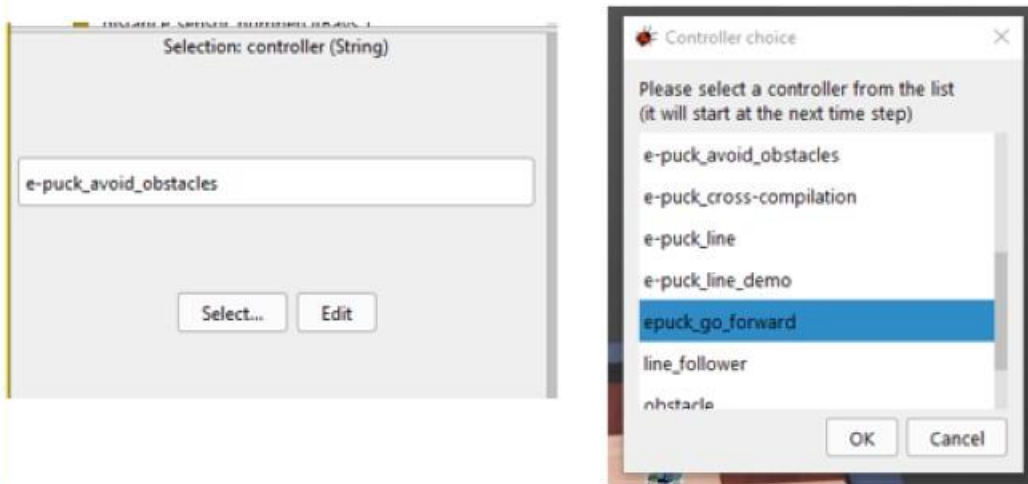
Each controller is executed in a separate child process usually spawned by Webots. Because they are independent processes, controllers don't share the same address space, and may run on different processor cores.

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)`):



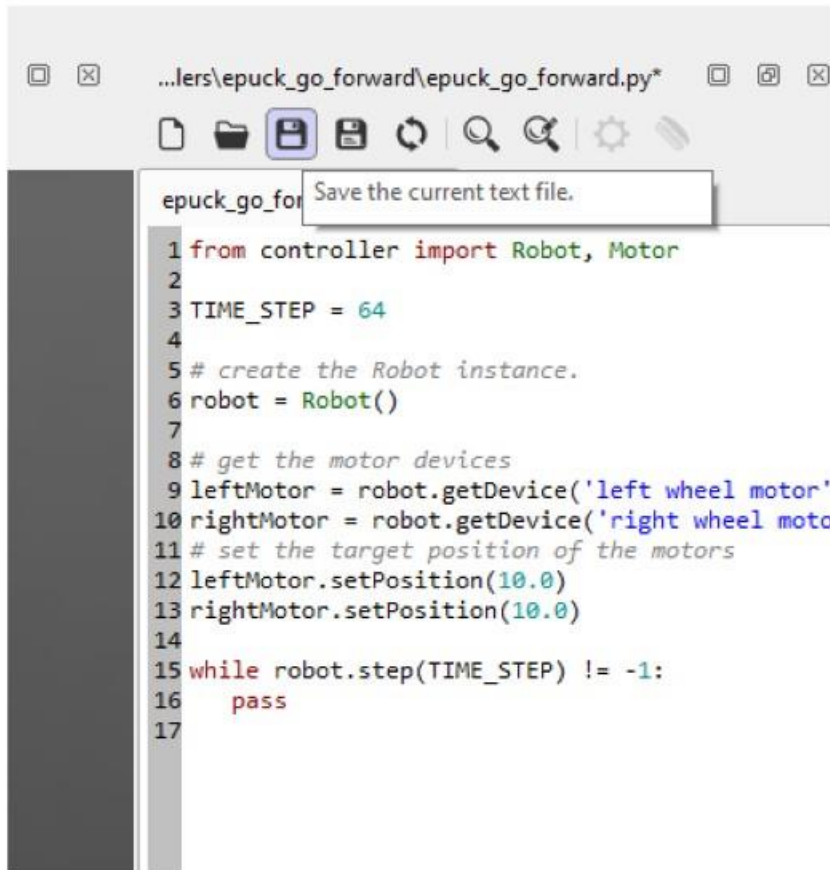
```
from controller import Robot, Motor

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 (new, open, save, close), editing (undo, redo, find, replace), and development (run, debug, settings). A "Save the current text file." dialog box is open over the 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'))
```

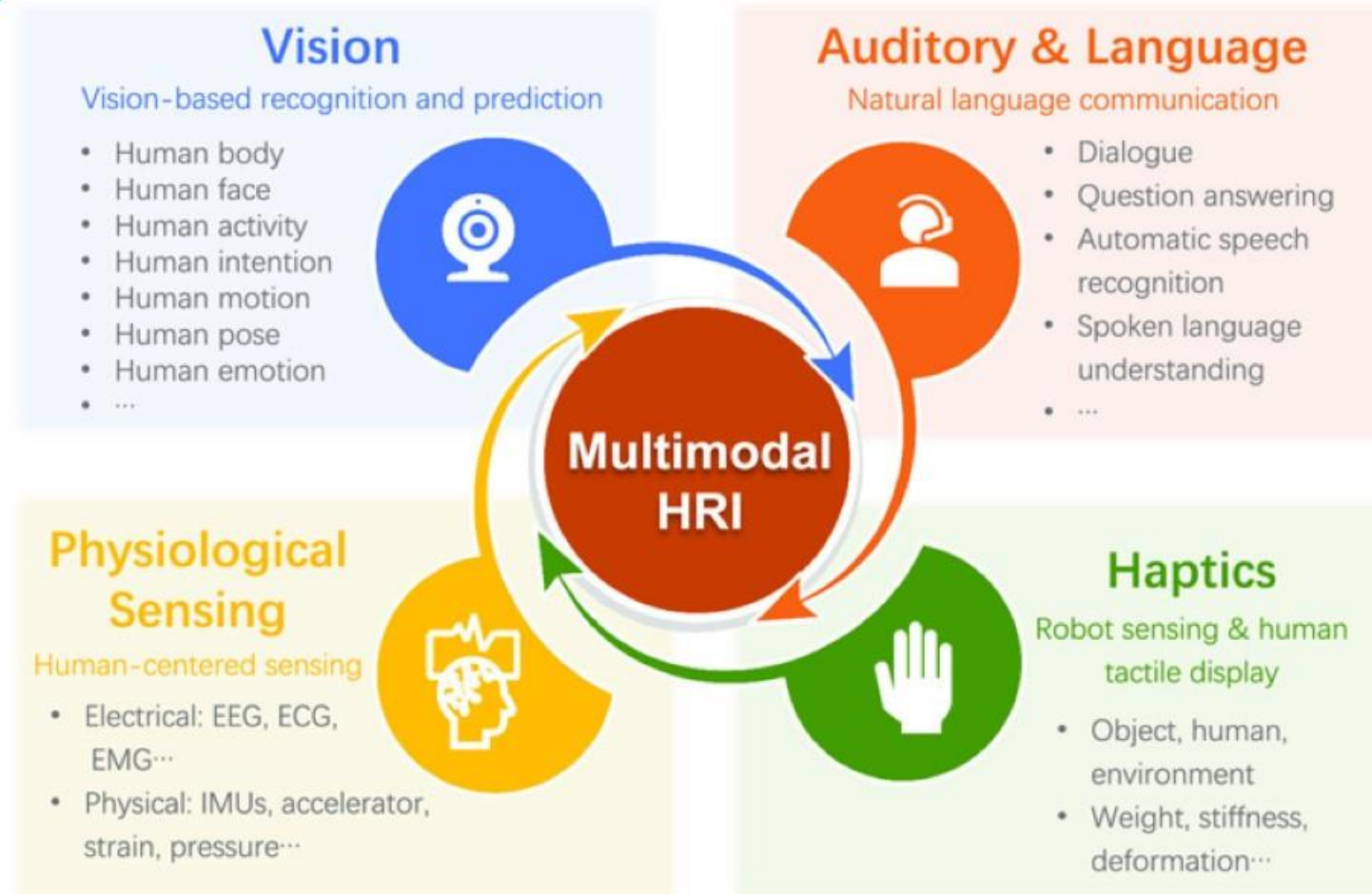
- Get Motor Handlers:** Retrieves references to the robot's left and right wheel motors using their names defined in the simulation world.
- Set Position to Infinity:** Configures the motors for speed control by setting their target positions to infinity (**float('inf')**).

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."