

# **RIME-222 Introduction to Mechatronics Design Fundamentals**

## **Assignment 01**

### **Introduction to Webots**



**Session: 2021**

**Section: C**

#### **SUBMITTED BY**

| <b>Name</b>                 | <b>CMS ID</b> |
|-----------------------------|---------------|
| <b>Muhammad Talha Irfan</b> | <b>368635</b> |

**School of Mechanical and Manufacturing Engineering**

**Deliverable 1:** Check list (with your name) of implemented / simulated steps, features, tutorial, e.gs., demos from internet (that you have done for understanding Webots).

Submission: <https://github.com/mtalirfan/RIME-222-Webots-Intro>

Main site: <https://cyberbotics.com/>

Github: <https://github.com/cyberbotics/webots>

User Guide: <https://cyberbotics.com/doc/guide/index>

Reference Manual: <https://cyberbotics.com/doc/reference/index>

Perquisites: mobile robotics, C, C++, Java, Python or MATLAB programming, VRML97 (Virtual Reality Modelling Language) <https://cyberbotics.com/doc/guide/language-setup>

Using Python: <https://cyberbotics.com/doc/guide/using-python>

Programming Fundamentals: <https://cyberbotics.com/doc/guide/programming-fundamentals>

Nodes and API Functions docs: <https://www.cyberbotics.com/doc/reference/nodes-and-api-functions>

Node Chart: <https://cyberbotics.com/doc/reference/node-chart>

Tutorials: <https://cyberbotics.com/doc/guide/tutorials>

Samples: <https://cyberbotics.com/doc/guide/sample-webots-applications>

Simulation Upload Service: <https://webots.cloud>

Cite Webots in scientific papers etc: <https://cyberbotics.com/doc/guide/citing-webots>



1. Download Webots.
2. Install it.
3. Start it.
4. Visit the Webots Guided Tour from the Help menu of Webots.
5. Follow the [Webots tutorials](#).
6. Explore [examples](#) and create your own simulation from them.

## Checklist:

Submitted by: Muhammad Talha Irfan 368635 ME-13-C

- ✓ Download Webots from: <https://cyberbotics.com/>
- ✓ Install and run Webots on computer.
- ✓ Visit the Webots guided tour from the Help menu of Webots.
- ✓ See Get Started steps on: <https://cyberbotics.com/doc/guide/getting-started-with-webots>
- ✓ Follow the Webots tutorials <https://cyberbotics.com/doc/guide/tutorials>
- ✓ Explore examples & create your own simulation from them.

## Documentation Notes:

### Getting Started:

**Keywords:** World (Scene/Robot Model), Robot Controller (C/C++/Python/Java/MATLAB), Physics Plugin (only C/C++), PROTO (add nodes to reuse complex objects), Simulation, 3D window, Nodes (objects), Scene tree, Text editor, Console, Speedometer (simulation speed relative to realtime), Virtual Time, WorldInfo (global parameters), Viewpoint (main camera parameters), TexturedBackground (scene background), TexturedBackgroundLight, RectangleArena (default object/node).

- Object bounding lines turn pink if the solid is colliding with another one and blue when the solid is idle (at rest, not interacting with other object).
- The console displays deterministic logs grouped by controllers at the end of simulation steps, ensuring output order by robot. At the end of simulation step, the output of robot A will be printed in the console before the output of robot B.
- The Startup mode allows to choose the state of the simulation when Webots is started: pause, realtime, run, fast.
- Cite Webots properly: <https://cyberbotics.com/doc/guide/citing-webots>
- Controllers are searched in the order shown, and the first one found is executed:
  1. "xyz\_controller[.exe]" (a binary executable compiled from C/C++)
  2. "xyz\_controller.class" (a Java bytecode class)
  3. "xyz\_controller.jar" (a Java .jar file)
  4. "xyz\_controller.bsg" (a Webots/BotStudio file)
  5. "xyz\_controller.py" (a Python script)
  6. "xyz\_controller.m" (a MATLAB function)
- The project root directory typically includes "worlds" (required), "controllers", "protos", and "plugins" directories for organizing world files (at least one required), controllers, PROTO files, and plugins respectively.
- Each world file in a project has a hidden corresponding project file and thumbnail file, denoted as ".wbproj" and ".jpg" respectively, with names based on the world file's name. These files store GUI information and provide a thumbnail for web viewing, with the thumbnail being 768px by 432px.

## Tutorial 1: Your First Simulation in Webots

- 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 defining the behaviour of robots.
- Controllers may be written in C or other languages (Python is used here).
- C, C++, and Java controllers must be explicitly compiled before they can be executed.
- Controllers are associated with robots via the controller fields of the Robot node.
- Pause, reset, modify, and save the simulation to avoid simulation default state errors.
- The same controller can be used by several robots, but a robot can only use one controller at a time.

## Tutorial 2: Modification of the Environment (Model)

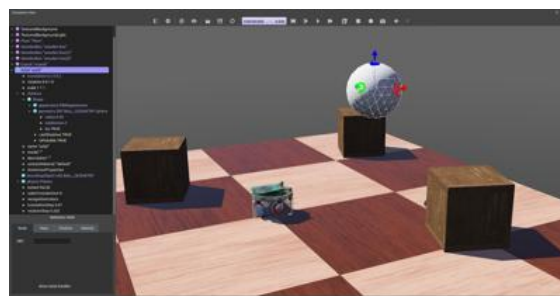
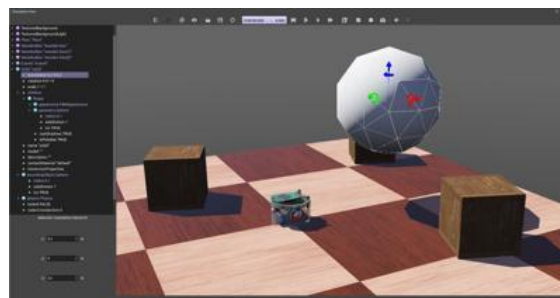
- Node Chart: <https://cyberbotics.com/doc/reference/node-chart>
- Solid Node (rigid body)

### Create a Ball

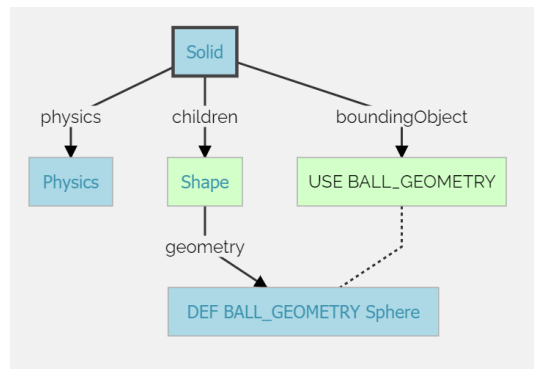
We will now add a ball to the simulation. That ball will be modeled as a rigid body as shown in this figure. A **Sphere** node will be used to define the geometry of our ball.

**Hands-on #4:** In the scene tree view, select the last node and press the **Add** button. In the dialog, open the **Bases nodes** section and select the **Solid** node. In the scene tree view, expand the **Solid** node and select its **children** field. Add a **Shape** node to it by using the **Add** button. Select the **appearance** field of the **Shape** node and use the **Add** button to add a **PBRAppearance** node.

1. Add a **Sphere** node as the **geometry** field of the newly created **Shape** node.
2. Expand the **PBRAppearance** node and change its **metalness** field to 0 and its **roughness** field to 1.
3. Add another **Sphere** node to the **boundingObject** field of the **Solid**.
4. Finally add a **Physics** node to the **physics** field of the **Solid**.
5. By modifying the **translation** field of the **Solid** node, place the ball in front of the robot (at  $\{0.2, 0, 0.2\}$  for example).
6. Save the simulation.
7. The result is depicted in this figure.



- DEF-USE mechanism allows defining a node once with a label (DEF) and reusing it elsewhere (USE), enabling avoidance of node duplication, simultaneous object modification, and field inheritance from the DEF node to the USE node.

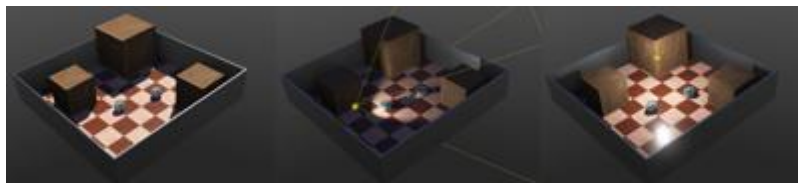


*DEF-USE mechanism on the Sphere node called BALL\_GEOMETRY.*

- Use as much as possible the DEF-USE mechanism at the Shape level rather than at the geometry level.

### Tutorial 3: Appearance (Model)

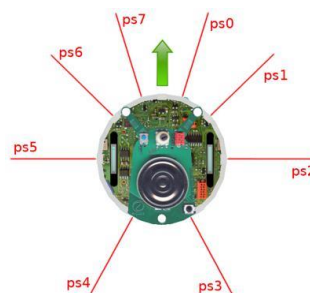
- DirectionalLight, PointLight, SpotLight.



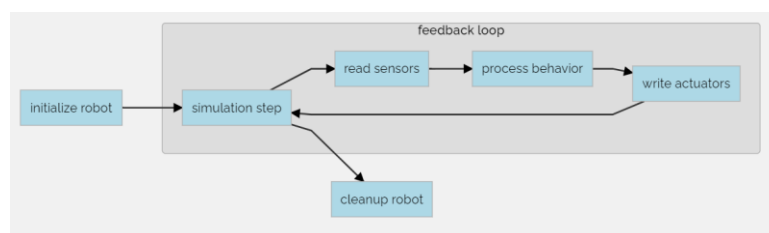
*Left DirectionalLight, middle SpotLight and right PointLight.*

- Rendering Options: Wireframe Rendering, Plain Rendering

### Tutorial 4: More about Controllers (Program)



*Top view of the e-puck model.*




*UML state machine of a simple feedback loop.*

- The values returned by the distance sensors are scaled between 0 and 4096 (piecewise linearly to the distance). While 4096 means that a big amount of light is measured (an obstacle is close) and 0 means that no light is measured (no obstacle).
- The controller entry point is the main function like any standard C program.
- No Webots API function should be called before the call of the `wb_robot_init` function.
- The last function to call before leaving the main function is the `wb_robot_cleanup` function.
- A device is referenced by the name field of its device node. The reference of the node can be retrieved thanks to the `wb_robot_get_device` function.
- Each controller program is executed as a child process of the Webots process. A controller process does not share any memory with Webots (except the cameras' images) and it can run on another CPU (or CPU core) than Webots.
- The controller code is linked with the "libController" dynamic library. This library handles the communication between your controller and Webots.

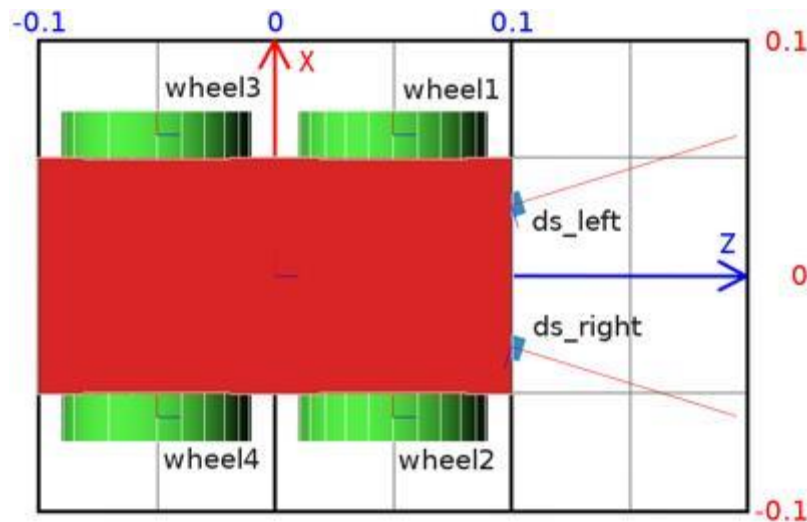
### **Tutorial 5: Compound Solid and Physics Attributes (Model) (not implemented)**

- Adding a Cylinder node crashed the Webots instance.
- Group nodes, sub-nodes, pose nodes.
- Only one of the density ( $\text{kg/m}^3$ ) and mass (kg) fields can be specified at a time (the other should be set to -1).
- The rotation field of the Pose node determines the rotation of this node (and of its children) using the Euler axis and angle representation. A Euler axis and angle rotation is defined by four components. The first three components are a unit vector that defines the rotation axis. The fourth component defines the rotation angle about the axis (in [rad]). e.g. a unit axis (0, 1, 0) and an angle of  $\pi/2$  ( $\sim 1.5708$ ).
- Grounds can be defined using the Plane or the ElevationGrid primitives. The Plane node is much more efficient than the ElevationGrid node, but it can only be used to model a flat terrain while the ElevationGrid can be used to model an uneven terrain.
- Contacts: ContactProperties node, The WorldInfo node has a contactProperties field.

 **Hands on #6:** Set the `contactMaterial` field of the dumbbell to "dumbbell". In the `WorldInfo` node, add a `ContactProperties` node between the `default` and `dumbbell` categories. Try to set the `coulombFriction` field to `0` and remark that the dumbbell slides (instead of rotating) on the floor because no more friction is applied.

- The most critical parameters for a physics simulation are stored in the `basicTimeStep`, `ERP` and `CFM` fields of the `WorldInfo` node.
- The `basicTimeStep` field determines the duration (in [ms]) of a physics step. The bigger this value is, the quicker the simulation is, the less precise the simulation is. Recommend values between 8 and 16 for regular use of Webots.

## Tutorial 6: 4-Wheeled Robot (main work)



- The robot has 4 DOF corresponding to the wheel motors. It can be divided in five solid nodes: the body and the four wheels.
- The initial position of the wheel is defined by the translation and the rotation fields of the Solid node.
- The rotation origin (anchor) and the rotation axis (axis) are defined by the optional HingeJointParameters child of the HingeJoint node.
- Sensor: 0 cm = 0 and 10 cm = 1000
- Most of it was modelling and making sure things were added and labelled correctly.
- During Modelling, the wheels were incorrectly named so the list was updated:  
wheelsNames = ['wheel1', 'wheel3', 'wheel2', 'wheel4']

## Tutorial 7: Your First PROTO (not implemented)

- Any PROTO file should at least respect the following structure:

```
#VRML_SIM R2023b utf8
PROTO protoName [
  protoFields
]
{
  protoBody
}
```

- The protoName should be the name of the PROTO file (i.e. FourWheelsRobot in this case), protoFields defines the modifiable fields of the PROTO node (this part should be empty for now) and the protoBody is the definition of the root node (i.e. the Robot node in this case).
- A sample PROTO:

```
#VRML_SIM R2023b utf8
PROTO FourWheelsRobot [
  field SFVec3f    translation  0 0 0
  field SFRotation rotation    0 0 1 0
  field SFFloat    bodyMass    1
]

```

```

{
  Robot {
    translation IS translation
    rotation IS rotation
    children [
      # list of children nodes
    ]
    boundingObject USE BODY
    physics Physics {
      density -1
      mass IS bodyMass
    }
    controller "four_wheels_collision_avoidance"
  }
}

```

## Tutorial 8: The Supervisor

- `IMPORTABLE EXTERNPROTO`
- "MFNode" stands for multi-field node whereas "sf\_node" stands for single-field node.
- Trick: If you are not familiar with VRML97, an easy trick to define these strings is to let Webots do it for you. You can manually create the object (using the Webots interface), and then save the world. When saving, Webots will translate what you built into a string of text. If you open the world file with a text editor, you can simply copy the description of the object.
- A Supervisor is nothing more than a Robot with extra powers, therefore anything that you can do with a Robot instance, you can do with a Supervisor instance.
- A Supervisor is not bound by physics, since its powers are unlimited, it can also "cheat".
- Whenever one wishes to alter the scene tree using a Supervisor, a reference needs to be obtained:
  - To insert a node, you need a reference of the field that will contain it.
  - To remove a node, you need a reference to the node (i.e., the object) itself.
  - To change the value of a parameter like (translation, color, size, etc.) you need a reference to said field.
- Spawning a robot or an object can be achieved by defining it as an `IMPORTABLE EXTERNPROTO` and using the `wb_supervisor_field_import_mf_node_from_string` API function.



- **Deliverable 2:** Problem formulation and description for Assignment 2. Assignment 2 guidelines and demo-date will be shared later. An existing work already reported in conference / journal / thesis literature can also be selected for Assignment 2.

Submission: <https://github.com/mtalirfan/RIME-222-Webots-Simulation>

Proposed Problem Statement:

### **Webots Driverless Simulation Competition**

The system design and code for a driverless vehicle needs to be validated before being tested on a driverless car in a physical environment. Unfortunately, before the code can be tested physically, we must undergo the expensive process of organising a suitable location/event, ensure the electrical and mechanical systems of the vehicle are operational, and accept the risk of errors causing damage to the car.

On the other hand, simulation software provides a virtual environment in which to experiment with various methods of autonomous navigation cheaply, easily, and safely.

A standard vehicle in the Webots robotics simulator will be given. This vehicle contains a basic kinematic tyre/axle model with a set of motors, as well as the following sensors:

- Distance Sensors
- Lidar
- Camera
- Gyroscope
- Accelerometer

Use a sample environment with a racetrack or build your own.

You are to program this autonomous vehicle to navigate for at least one full lap around a track. Use any of the supported languages to write the robot controller. Use any combination of sensors to accomplish this, as well as computer vision implementations.

Once functional, the goal is to optimise the navigation algorithm to achieve the fastest lap times possible, avoiding collisions, for multiple laps around each track.

Credits and Many Thanks to UNSW Redback Racing for the inspiration:

[https://github.com/UNSW-Redback-Racing/NMP\\_Webots/tree/main](https://github.com/UNSW-Redback-Racing/NMP_Webots/tree/main)