UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Tools & Simulators

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Tools

Transform and RViz

The Tools

- We will look at rviz (2) and Gazebo/CoppeliaSim

*"rviz shows you what the robot **thinks** it's happening, while Gazebo (CopelliaSim) shows you what is **really** happening."*
Morgan Quigley

- Gazebo and CopelliaSim are **physics simulators**
- Rviz is a visualization tool which enables the user to see the **robot's state** and **perception**

tf2 and rviz

- First install the needed packages

  sudo apt-get install ros-humble-turtle-tf2-py ros-humble-tf2-tools ros-humble-tf-transformations
- Then
  - Launch this particular turtle simulation

    ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
  - In another terminal launch the controller

    ros2 run turtlesim turtle_teleop_key

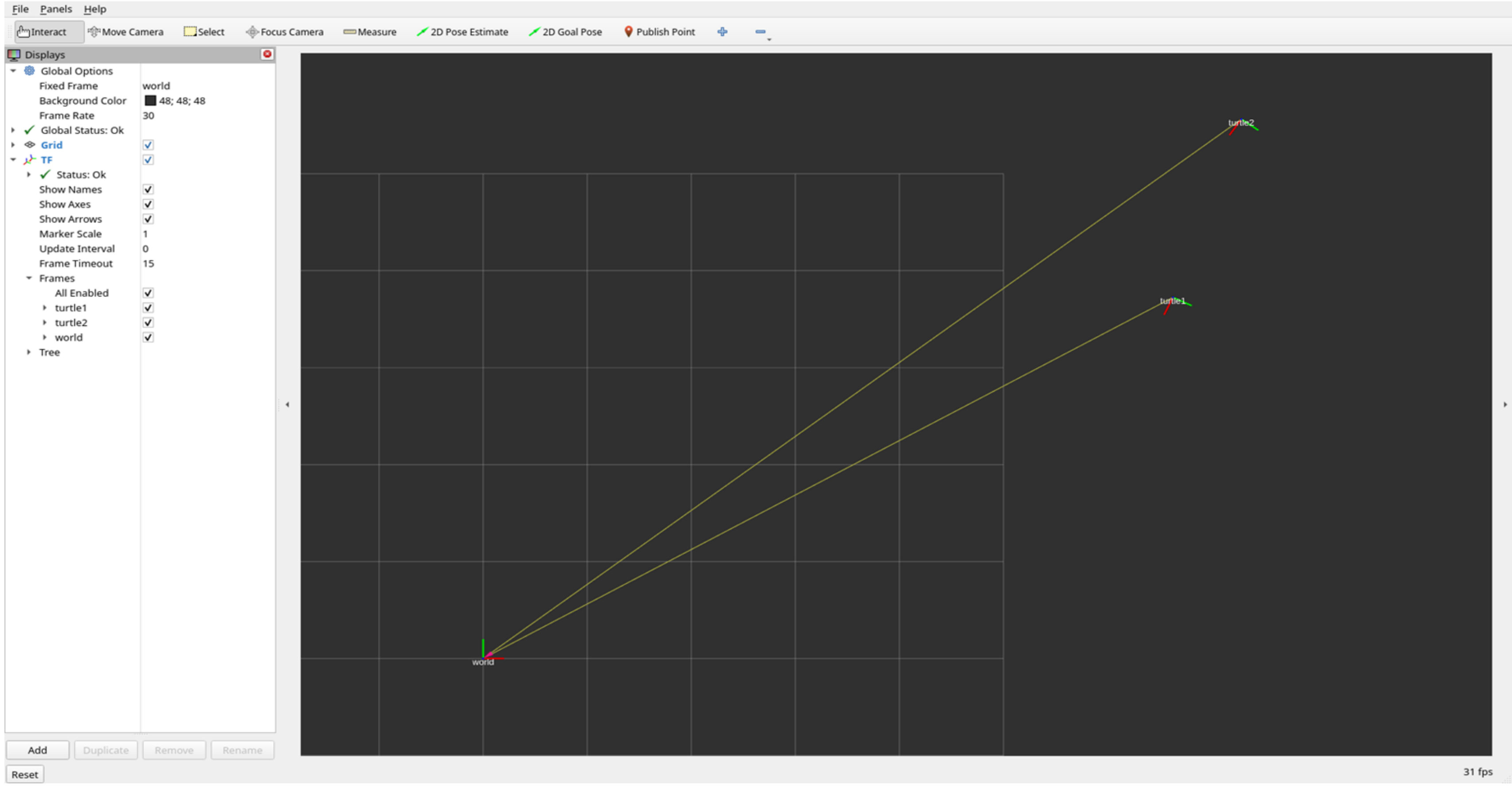
This demo is using the tf2 library to create three coordinate frames: a world frame, a turtle1 frame, and a turtle2 frame. This tutorial uses a tf2 broadcaster to publish the turtle coordinate frames and a tf2 listener to compute the difference in the turtle frames and move one turtle to follow the other.

- Open rviz, a visualization tool that is useful for examining tf2 frames:

  ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz

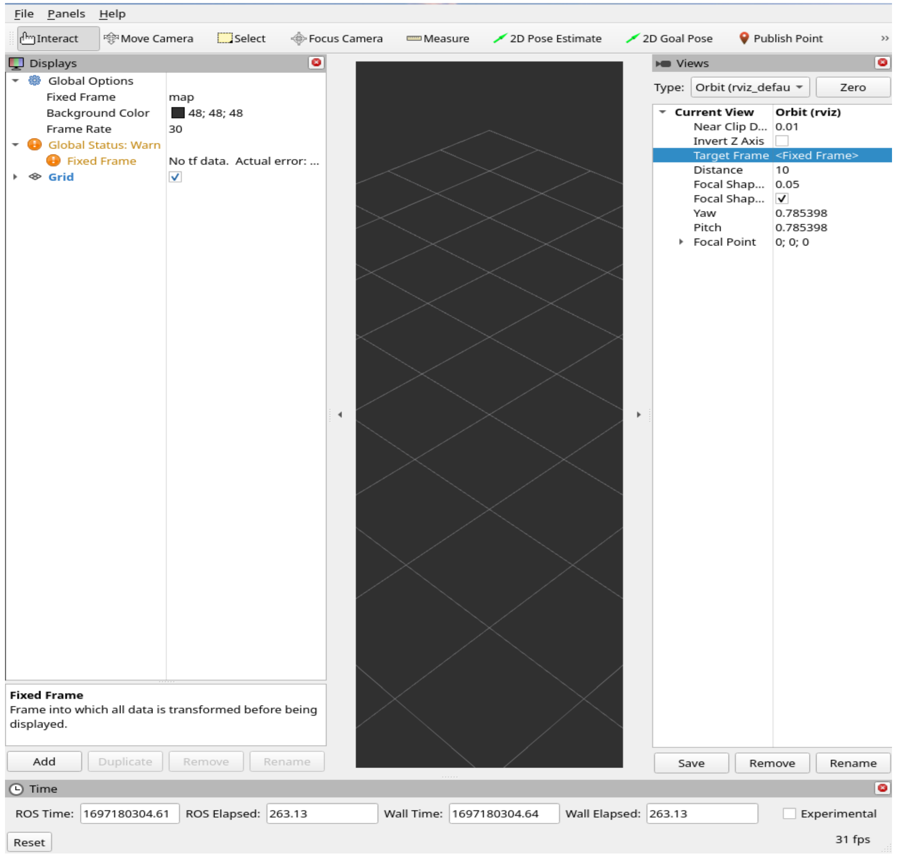
tf2 and rviz

In the side bar you will see the **coordinate frames** broadcasted by tf2.
As you drive the turtle around you will see the frames move in rviz.



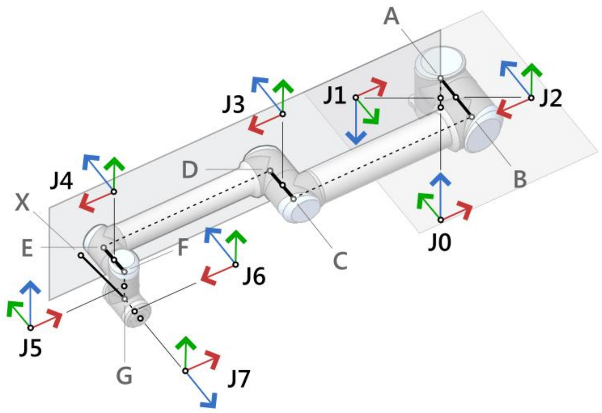
rviz

- It has two frames controlling how data is displayed:
  - **Fixed frame**: is the frame all incoming data is *transformed into* before being displayed, hence it should be set to either a *root element* (like map) or a *fixed frame*
  - **Target frame**: reference frame for the camera view
- “Pose estimate” used to initialize the position of your robot → sends the position on /initialpose



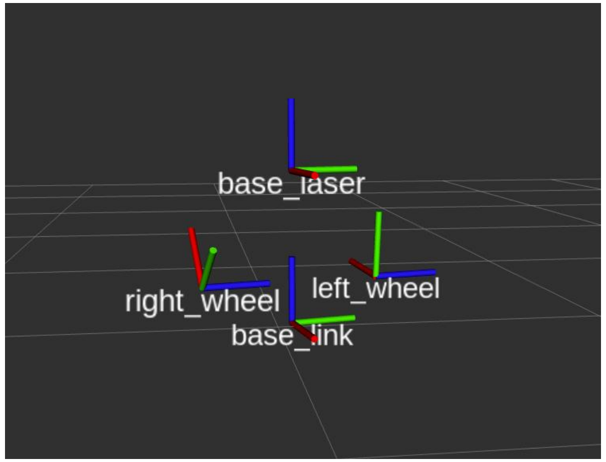
tf2

- We are looking at **coordinate frames**
- tf2 is a special library of ROS2 which publishes the **transformation matrices** between coordinates frames in the topic /tf2
- Necessary to easily understand the position of one coordinate frame w.r.t. another



UR5 coordinate frames

Mobile robot's coordinate frames



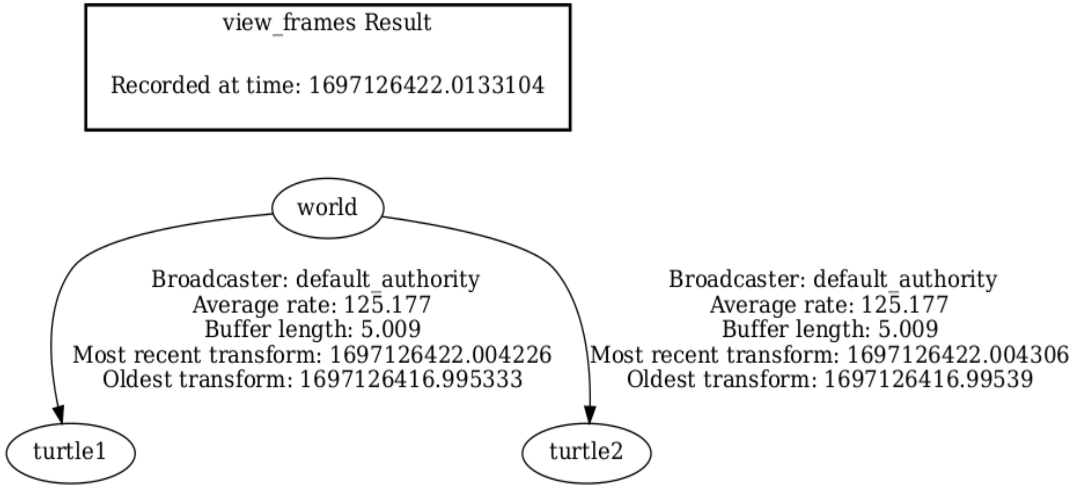
tf2

- If using wsl, install wslview
  sudo apt install wslu
- We can see the relations between frames using

  ros2 run tf2_tools view_frames

  wslview <name_of_pdf>
- We can look at the transform between two frames by running

  ros2 run tf2_ros tf2_echo [source_frame] [target_frame]



tf2

With tf2 you can:

- Define static broadcaster (define the relationship between a robot base and fixed sensors or non-moving parts);
- Define a broadcaster (define the relationship between a robot base and moving parts → timestamped transformations);
- Define a listener (to use the published transformation in a code)
- And more (s.ee https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html)

ROS 2 Tools

URDF

URDF

- rviz2 uses **Unified Robot Description Format (URDF)** for robot models → XML format
- We have to specify:
  - *Robot*: information on the robot
  - *Links*: the components of the robots
  - *Joints*: the interactions between links
  - Many more, see https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/URDF-Main.html#

```
<?xml version="1.0"?>
<robot name="myfirst">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

URDF - Links

A **link** describes a rigid body and may have the following properties:

- **Visual**: how the body **should appear** in the simulation. There may be *more than one* and together they represent the body. Within visual you specify:
  - *Origin*: the reference frame of the visual w.r.t. the reference frame of the link → expressed as **offset**
  - *Geometry*: the shape, which may be: box, sphere, cylinder, mesh
  - *Material*: the material of the visual element
- **Collision**: similar to visual, but used to check for collision during simulation
  - May not have the same shape → easier check on collisions and safer zones
- **Inertial**: define some physical properties of the robot for the simulation
  - *Inertia*: [rotational inertia matrix](#) → mandatory
  - *Mass*: in kilograms

URDF - Joints

- A **joint** describes how two links interact:
- When defining, we must specify the **type**:
  - **fixed**: the joint cannot move
  - **revolute**: it rotates *along* the axis and we *must limit* the range with upper and lower limits.
  - **continuous**: a continuous hinge joint that rotates around the axis and *has no* upper and lower limits.
  - **prismatic**: a sliding joint that slides along the axis, and *has a limited* range specified by the upper and lower limits.
  - **floating**: this joint allows motion for all 6 degrees of freedom.
  - **planar**: this joint allows motion in a plane perpendicular to the axis.
- The elements inside the joint may be:
  - **Origin**: represent a transform from the parent link to the child link
  - **Parent**: the parent link
  - **Child**: the child link
  - **Limit**: the limits to be respected when using type revolute or prismatic
  - See reference for more

Visualizing an URDF

Install the dependency:

```
sudo apt install ros-humble-urdf-tutorial -y
```

URDF models are usually placed in the *urdf* folder. We will visualize now with:

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/<robot.urdf>
```

This launch does three things:

- Loads the specified model and saves it as a parameter for the *robot_state_publisher* node;
- Runs nodes to publish sensor_msgs/msg/JointState and transforms;
- Starts Rviz with a configuration file.

Visualizing an URDF

Example URDFs are located in urdf_tutorial:

- Inspect the URDF file:

code /opt/ros/humble/share/urdf_tutorial/urdf/<robot.urdf>

- Visualize the robot in the URDF file in rviz:

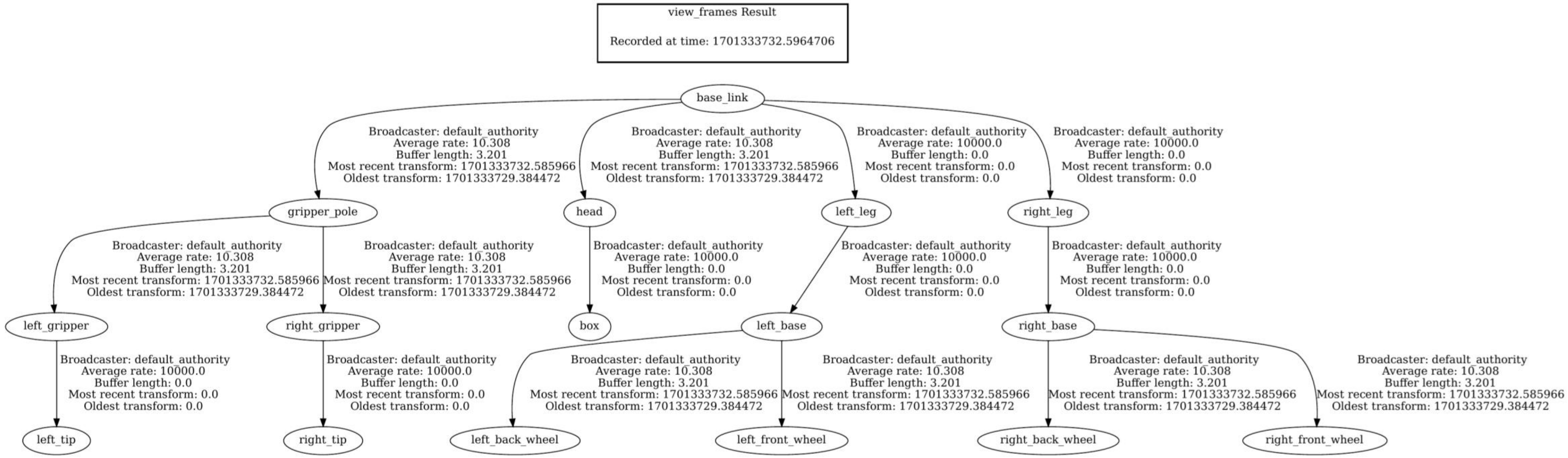
ros2 launch urdf_tutorial display.launch.py model:=urdf/<robot.urdf>

Examples:

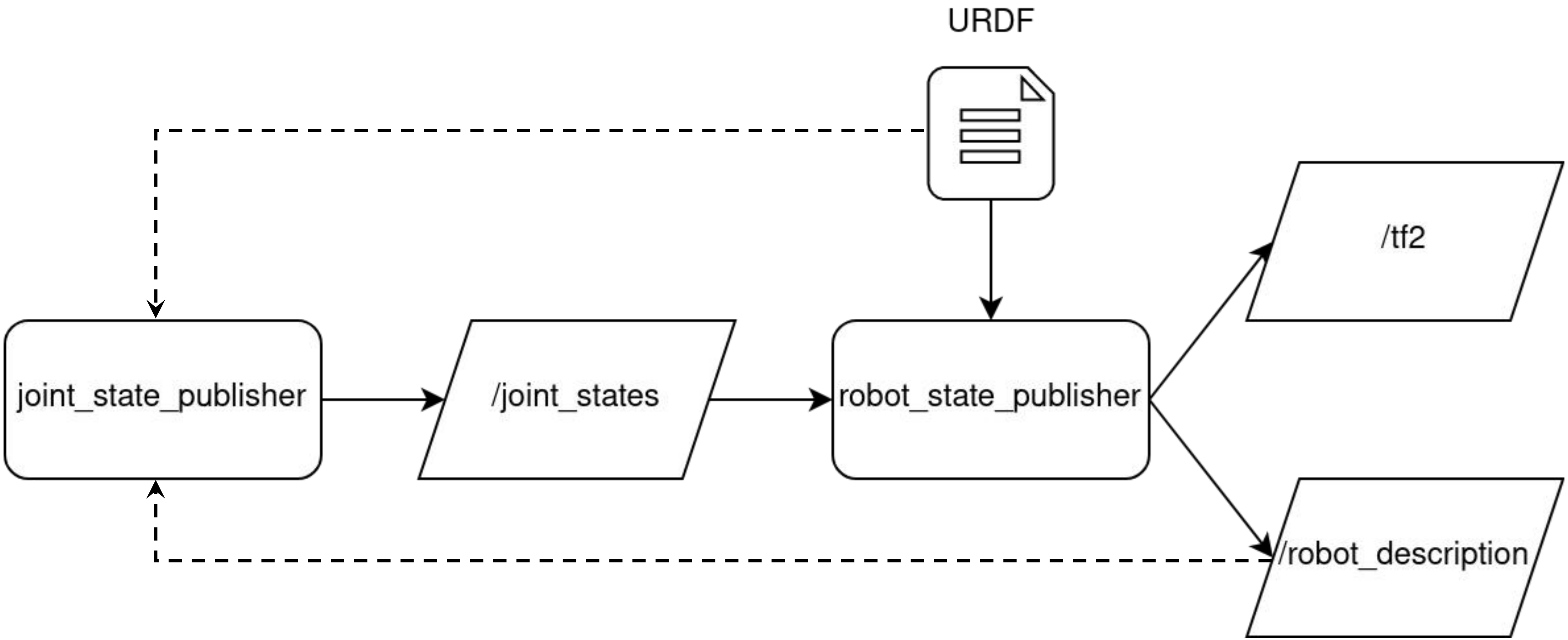
- Single link: 01-myfirst.urdf
- Two links with a fixed joint: 02-multipleshapes.urdf; 03-origins.urdf (specifies also where the second shape is originated)
- Three links of specific material/color with two fixed joints: 04-materials.urdf
- Multiple links and joints: 05-visual.urdf; 06-flexible.urdf (joints are flexible, with position, velocity or effort limits)
- Links with collision and inertial properties: 07-physics.urdf

Check the Tree Frames

ros2 run tf2_tools view_frames



Joint State and Robot State



Xacro and URDF

- ROS2 allows for using *xacro*, a macro language for XML (.xacro)
- This files use **macros** to ease some aspects of URDF files:
  - **constants** so to not have to change the same numeric value in many places;
  - **mathematical operations** to compute values;
  - **macros** to define whole pieces of code, also *parametrized*.

Xacro and URDF

It is possible to use xacro in 2 ways:

- To compile from .xacro to .urdf use

  ```
  xacro file.xacro -o file.urdf
  ```

- Automatically generate the urdf in a launch file. This is convenient because it stays **up to date** and doesn't use up hard drive space. However, it does take time to generate, so be aware that your launch file might take longer to start up.

  ```
  ros2 launch urdf_tutorial display.launch.py model:=urdf/08-macroed.urdf.xacro
  ```

Visualizing an URDF

urdf_launch, uses

- joint_state_publisher, which, will continuously publish all joint states to /joint_states. It reads the description of the robot by:
  - listening to /robot_description;
  - an input URDF.
- robot_state_publisher, which publishes the state of a robot to tf2 by reading the URDF and by subscribing to joint_state_publisher to get individual joint states

Visualizing an URDF

We can use the package *urdf_launch* to:

- **Load Robot description** (description.launch.py):
  - Loads the URDF/xacro robot model as a parameter, based on launch arguments;
  - Launches a single node, *robot_state_publisher*, with the robot model parameter;
  - *Robot description* becomes available as a topic.

```
def generate_launch_description():
ld = LaunchDescription()
ld.add_action(IncludeLaunchDescription(PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'description.launch.py']),
   launch_arguments={
    'urdf_package’: 'urdf_tutorial',
    'urdf_package_path': PathJoinSubstitution(‘urdf', '06-flexible.urdf'])}.items() ))
return ld
```

Visualizing an URDF

We can use the package *urdf_launch* to:

- **Display Robot Model (**display.launch.py**):**
  - Display just the robot model in Rviz with a preconfigured setup;
  - Launches a joint state publisher (with optional GUI), which publishes in the topic /joint_states msgs of the type sensor_msgs::msg::JointState.

```
def generate_launch_description():
ld = LaunchDescription()
ld.add_action(IncludeLaunchDescription(PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'display.launch.py']),
   launch_arguments={
     'urdf_package': 'urdf_tutorial',
     'urdf_package_path': PathJoinSubstitution(‘urdf', '06-flexible.urdf'])}.items() ))
return ld
```

Execise

- Clone the [UR Description repo]() and compile the workspace:
  git clone -b humble https://github.com/UniversalRobots/Universal_Robots_ROS2_Description.git ur_description
- Inspect *ur.urdf.xacro* and *ur.macro.xacro* the urdf folder. All the UR model URDFs can be generated using the same xacro files and the data in the config folder;
- Generate the URDF of the UR5e in a launch file and visualize it.

- Display the TF and their names;
- Modify the joint configuration; what happens with elbow_joint:=2.992? Why is happening?

ROS 2 Tools

Simulators

Physics Simulators

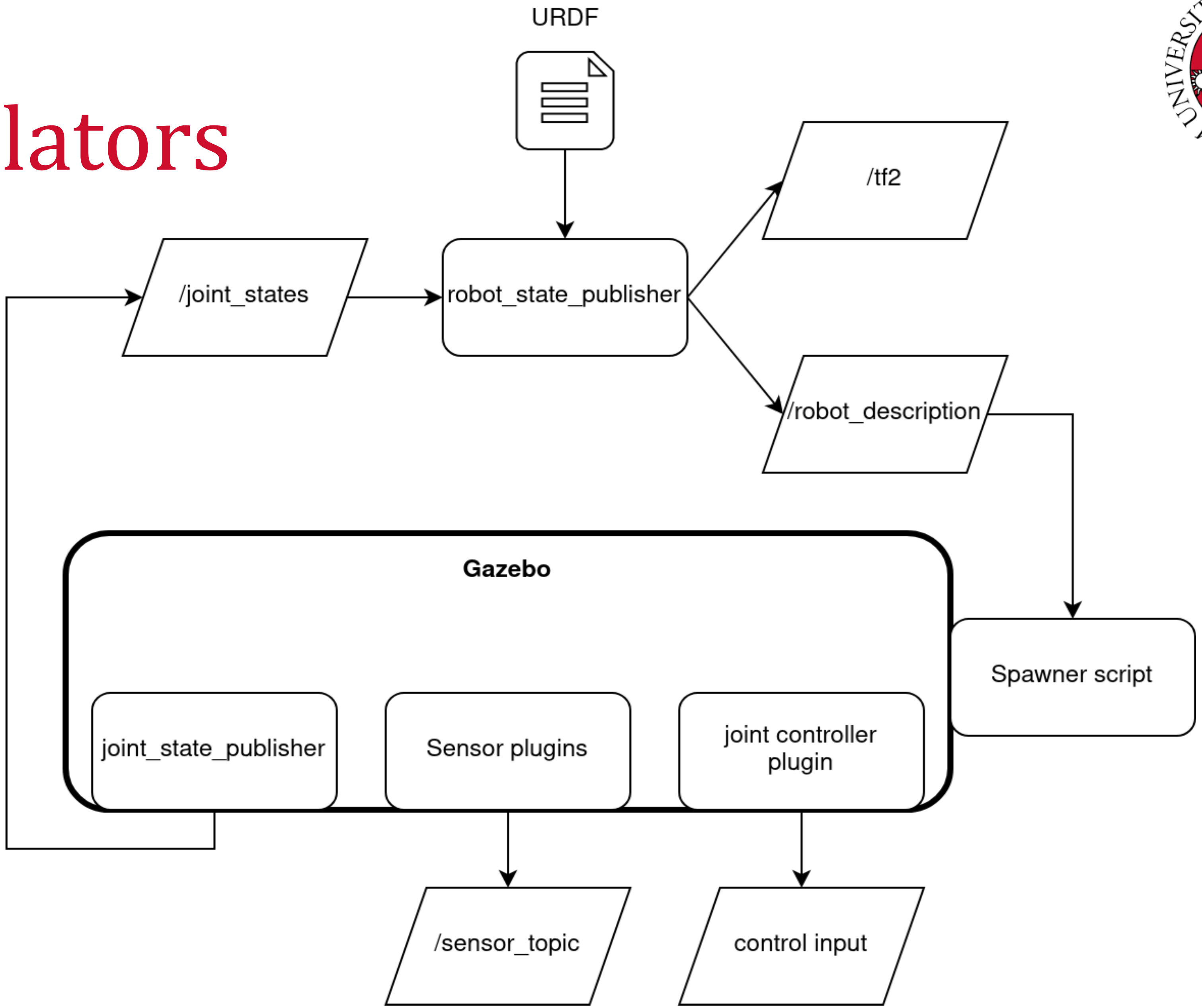
- Why is happening? Because **in rviz physics is not simulated** (it is a **visualization tool**), so one body can pass through another.
- We could manually publish the **joint states**, but this is not the task of the robot programmer → On a real robot this is **read from the sensors on the joint of the robot**.
- Before testing on a real robot, one can use a **physics engine** to get the joint states (here is a benchmark):
  - Open Dynamics Engine (ODE);
  - Dynamic Animation and Robotics Toolkit (DART);
  - Bullet;
  - Multi-Joint dynamics with Contact (MuJoCo);
  - And many more…

Physics Simulators

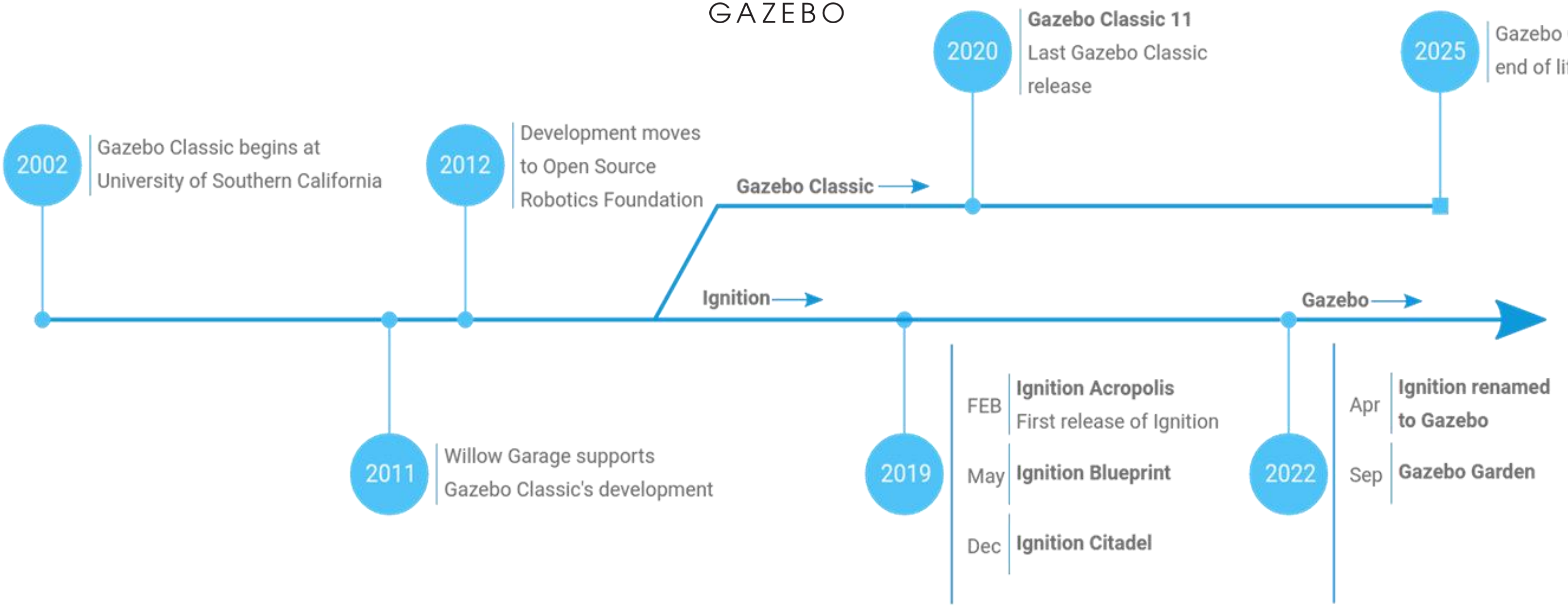
Usually, these general physics simulators are used in **robotic physics simulators** which allows simple integration with ROS (another benchmark):

- **Gazebo**: open-source from the OSRF with support for ROS2
- **CoppeliaSim**: free-educational with support for ROS2 and Python/Lua support for scripts in CoppeliaSim, lighweight
- **Unity**: free-educational with support for ROS2
- **Webots**: open-source with support for ROS2
- **NVIDIA Omniverse**/**Isaac Sim**: free for individuals, with ROS2 bridge, very demanding system requirements

Physics Simulators



Gazebo



2002
Gazebo Classic begins at University of Southern California
2011
Willow Garage supports Gazebo Classic's development
2012
Development moves to Open Source Robotics Foundation
Gazebo Classic
Ignition
2019
FEB
Ignition Acropolis
First release of Ignition
May
Ignition Blueprint
Dec
Ignition Citadel
2020
Gazebo Classic 11
Last Gazebo Classic release
2022
Apr
Ignition renamed to Gazebo
Sep
Gazebo Garden
Gazebo
2025
Gazebo Classic end of life

Gazebo

- The recommended version of Gazebo with ROS 2 Humble is GZ Fortress (list of features);
  sudo apt install ros-humble-ros-gz

- Launch a demo simulation:
  ign gazebo -v 4 -r visualize_lidar.sdf
  if it crashes, try with:
  ign gazebo -v 4 -r visualize_lidar.sdf --render-engine ogre

- Check gazebo topics: ign topic -l

- Check ros2 topics (gazebo and ros2 are separated)

Gazebo

- Install a bridge for the topics and the keyboard teleop package:

  sudo apt-get install ros-humble-ros-ign-bridge ros-humble-teleop-twist-keyboard

- We can map the ROS topic in a Gazebo topic:

  ros2 run ros_gz_bridge parameter_bridge
  /model/vehicle_blue/cmd_vel@geometry_msgs/msg/Twist]ignition.msgs.Twist

- And finally, teleoperate the blue robot:

  ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r
  /cmd_vel:=/model/vehicle_blue/cmd_vel

Gazebo

- Gazebo uses **.sdf** (Simulation Description Format) as model files, an XML format similar to URDF, but:
    - Describe also the **world** and not only the models;
    - Use **plugins** to describe how to interact with other programs, such as ROS.
- Gazebo concepts can be found [here];
- Robot models described by URDF can also be spawned in the simulation environment:

```
ign gazebo empty.sdf
ign service -s /world/empty/create --reqtype ignition.msgs.EntityFactory --reptype
ignition.msgs.Boolean --timeout 1000 --req 'sdf_filename:
"/opt/ros/humble/share/urdf_tutorial/urdf/07-physics.urdf", name: "urdf_model"'
```

Gazebo and Control

The robot controller can be directly written as a Gazebo plugin (shared library using Gazebo API), independent from ROS 2, but this has many limits:

- You **lose the standard interfaces** (joint state broadcaster, controller_manager, ros2 control CLI).
- Many ROS 2 packages (MoveIt, Nav2, tools expecting hardware_interface) **assume ros2_control**; without it, integration gets harder.
- You'll need to maintain your **own message schema**, plugins, and lifecycle/timeout/latency handling.

```
<gazebo>
  <plugin name="gazebo_ros_diff_drive"
      filename="libgazebo_ros_diff_drive.so">
   <left_joint>chassis_to_left_wheel_joint</left_joint>
   <right_joint>chassis_to_right_wheel_joint</right_joint>
   <wheel_separation>${body_width+0.04}</wheel_separation>
   <wheel_diameter>${wheel_radius*2.0}</wheel_diameter>

   <max_wheel_torque>200</max_wheel_torque>
   <max_wheel_accelaration>10</max_wheel_accelaration>

   <odometry_frame>odom</odometry_frame>
   <robot_base_frame>base_link</robot_base_frame>
   <publish_odom>true</publish_odom>
   <publish_wheel_tf>true</publish_wheel_tf>
   <publish_odom_tf>true</publish_odom_tf>
  </plugin>
</gazebo>
```

In this course we will use **ros2_control**, which provides scalable and reusable implementation of standard controllers that can work on **real hardware**!

CoppeliaSim

- Different OS supported;
- Low CPU usage;
- Different physics simulator integrated:
  - Bullet;
  - ODE;
  - Vortex;
  - Newton
- Comparison studies available:
  - https://www.sciencedirect.com/science/article/pii/S1569190X22001046
  - https://arxiv.org/pdf/2204.06433

CoppeliaSim

- Download from https://www.coppeliarobotics.com/downloads:

```
wget https://downloads.coppeliarobotics.com/V4_8_0_rev0/CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04.tar.xz
```

- Extract it:

```
tar -xf CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04.tar.xz4
```

- Install dependencies:

```
sudo apt update; sudo apt install xsltproc; python3 -m pip install pyzmq cbor2 xmlschema
```

- To run the application:

```
cd CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04
./coppeliaSim.sh
```

In the case of a newer release, please update the path/name to the file accordingly.

External controller with CoppeliaSim

CoppeliaSim provides several ways to specify a robot controller:

- With a **simulation script** (in Python or Lua), or writing a **plugin** (in a language able to generate a shared library and able to call exported C-functions), or calling the **client libraries** (C++/Python);
- Using the **remote API**, such as the ZeroMQ remote API. In this way, you can run the control code from an external application, from a robot or from another computer. This also allows you to control a simulation with the exact same code as the one that runs the real robot.
- Using a the **ROS2 Interface**, which is available in the simROS2 package. The interface will allow the creation of ROS2 Communication Interfaces (topic, services, etc.) that will be directly available in ROS.

CoppeliaSim – ROS2 Bridge

- CoppeliaSim is a general-purpose robotic simulator and does not require ROS to work;
- You can program directly the robot control/motion planner etc in CoppeliaSim using the Python/Lua scripts..
- But if we do this, we will lose the modularity of the ROS framework and the use of ROS tools!
- So, we will need a bridge between ROS2 and CoppeliaSim→ CoppeliaSim provides one in
  CoppeliaSim_{version}/programming/ros2_packages

CoppeliaSim – ROS2 Bridge

- In CoppeliaSim_{version}/programming/ros2_packages 2 ROS2 packages are available:
  - sim_ros2_interface;
  - ros2_bubble_rob.
- You can copy them into your workspace and then compile it:

```
$ export COPPELIASIM_ROOT_DIR=~/path/to/coppeliaSim/folder
$ ulimit -s unlimited #otherwise compilation might freeze/crash
$ colcon build --symlink-install --cmake-args -DCMAKE_BUILD_TYPE=Release
```

- Let’s follow the tutorial to run the *bubble_rob* example: https://www.coppeliarobotics.com/helpFiles/en/ros2Tutorial.htm