PSD simulator

Edoardo Caciorgna

IN SUPREMÆ DIGNITATIS

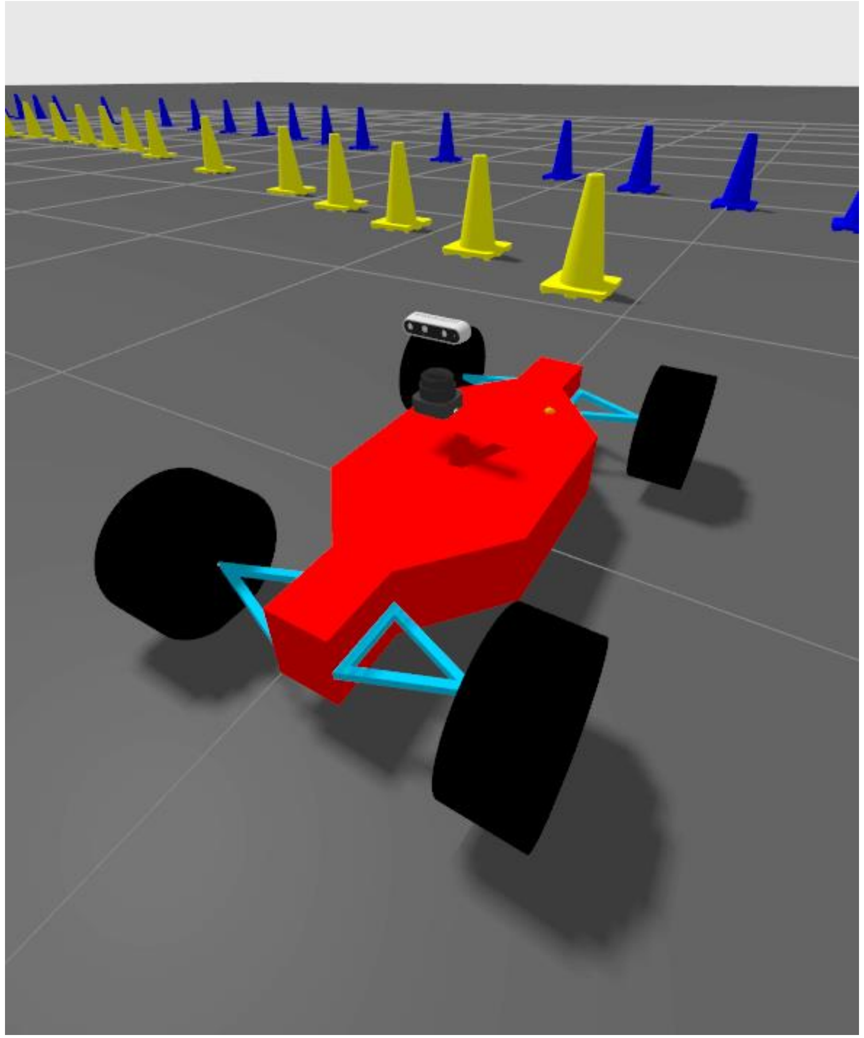
1343

UNIVERSITÀ DI PISA

Introduction

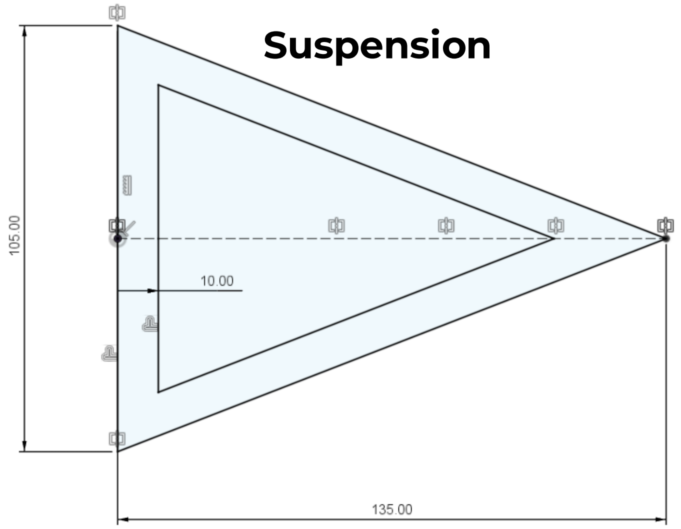
Objectives:

- **Simulate** vehicle control **dynamics**
- **Simulate sensors** for **perception** (camera, depth camera and LiDAR)
- **Test code and algorithm** without the need of the real model
- **Try various scenarios** and track delimiter easily (cones, aruko, lines, ...)

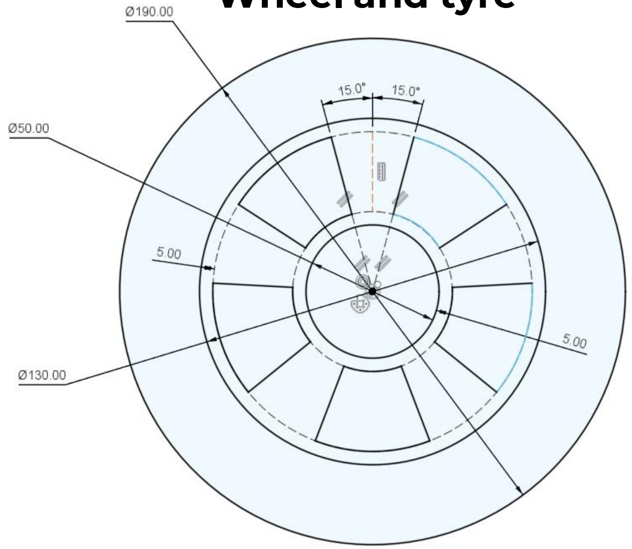


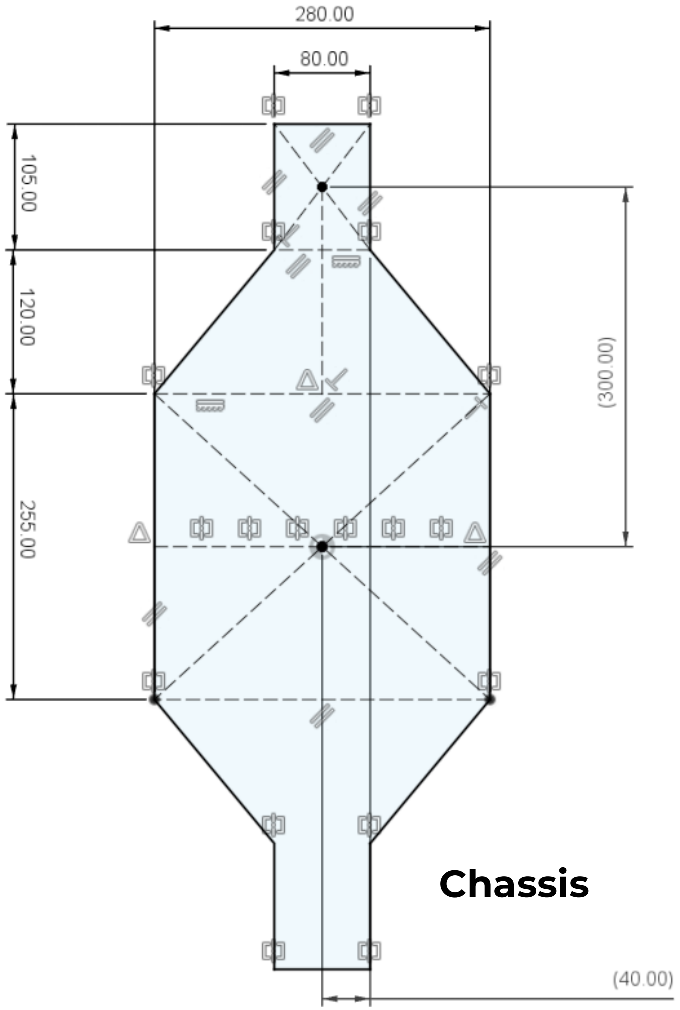
Hardware – 4WD vehicle

Suspension

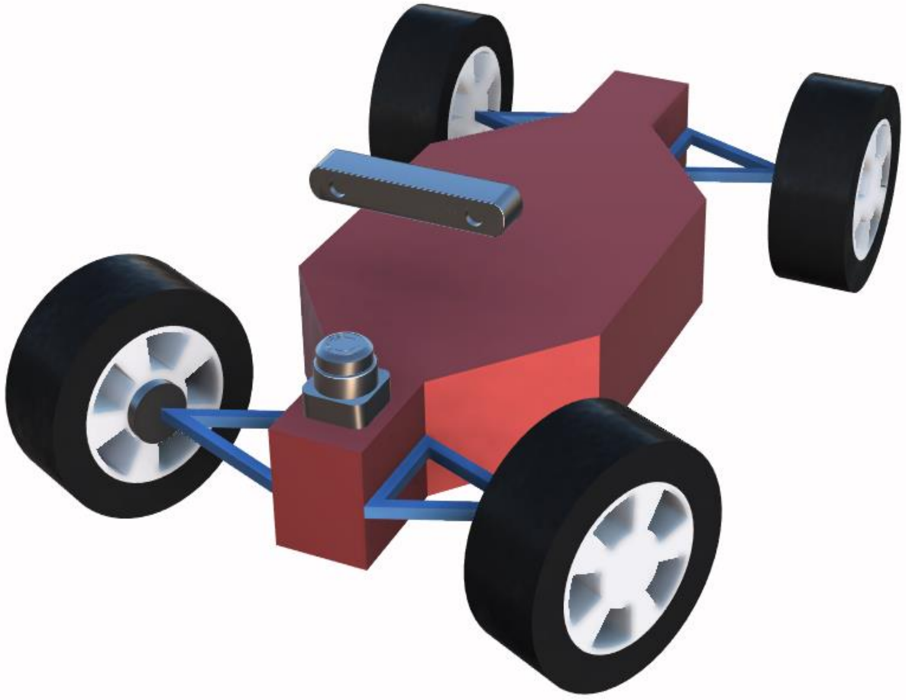


Wheel and tyre





Chassis

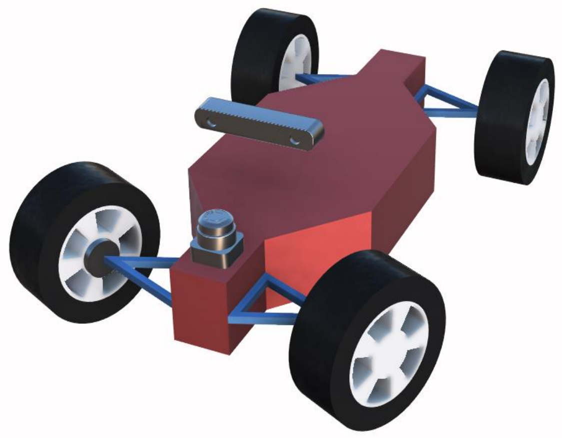


Hardware - sensors

LiDAR
SLAMTEC RPLiDAR S2

- Low-cost
- Medium range
- Easy-to-use
- Good resolution

Angular Range 360 °	Measuring Range 30M	Measuring Accuracy ±30mm	Sampling Frequency 32K
Rotational Speed 10Hz	Angular Resolution ≤0.12°	RPlidar Dimensions 77 x 77 x 38.85mm	Applications Indoor & Outdoor



Stereocamera
Stereolabs ZED2

- High resolution Depth sensing
- Wide Field of View (FOV)
- Advanced AI and Computer Vision Algorithms
- Seamless integration

Wide-angle 3D AI camera

Combine long-range depth perception with AI to perceive your environment in 3D with up to a 120° (Diagonal) wide-angle field of view.

Secure USB Type-C connection

Use a highly reliable USB 3.0 type-C cable with thumbscrew locking connectors and ensure a secure interconnection for your systems.

IP66-rated enclosure

Resistant to dust, water and humidity, the ZED 2i is designed for outdoor applications and challenging medical, industrial, agricultural environments and more.

Built-in IMU, barometer & magnetometer

Featuring 9-DoF sensors for spatial and positional awareness. Factory calibrated on 6-axis with robotic arms.

Multiple lens selection with polarizer

Select a 2.1 mm or a 4 mm lens depending on your application. Add a built-in CPL polarizing filter when working outdoors.

Multiple mounting options

With flexible mounting options and a flat bottom, the ZED 2i can be easily integrated in any system and environment.

Software

Development platform containing the operating system and all necessary dependencies

Framework that manages every part of the robot and allows **interaction** between the various algorithms implemented

Base system:
Ubuntu 24.04 Noble
+
ROS2 Jazzy Jalisco

Gazebo Harmonic: simulates the environment and physical interactions within it (*.sdf file* for the description of the contents of the virtual environment);

Rviz: shows what the robot sees (display of sensor data and contents of some topics)

Platforms for cooperative development and code management

Workspace infrastructure

- **psd_ws/**
  - build/
  - install/
  - log/

    *Automatically generated by **colcon build***
  - **src/**
    - psd_vehicle_**description**/ — *Contains a **description** of the **vehicle** in **.urdf format** (link, joint, frame)*
    - ros_components_description/ — *Contains **description** files of the **sensors** (LiDAR and stereocamera) for **vehicle perception** (**.urdf** and **.dae**)*
    - psd_**gazebo_worlds**/ — *Contains a **description** of the **environment** for **Gazebo in .sdf format** (cones, obstacles, etc.)*
    - psd_vehicle_**bringup**/ — *Contains **launch file** to spawn nodes and start services to **start the sim, spawn the vehicle and control it***
    - psd_vehicle_**stack**/
      - psd_vehicle_**slam**/
      - psd_vehicle_**control**/
      - psd_vehicle_**perception**/

psd_vehicle_description/

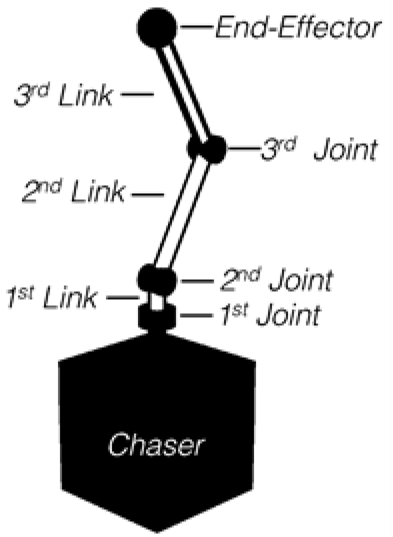
How to describe the vehicle?

Key objectives:

- Define vehicle **components physics**
- Define **joint type** and **actuation**

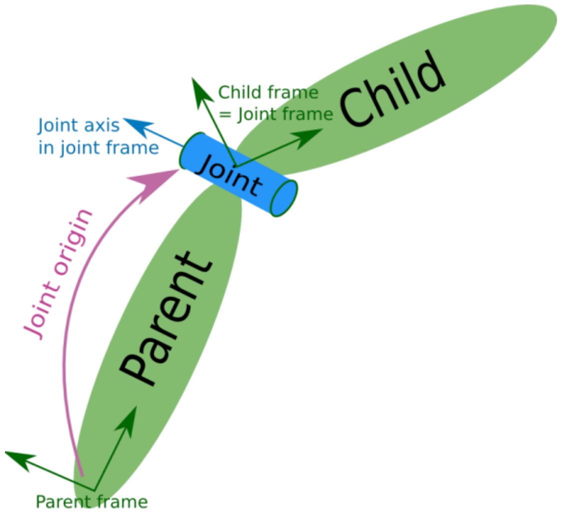
vehicle **dynamics and kinematics**

- Enable **realistic simulations** in Gazebo
- Facilitate **motion planning** and **control**
- Seamless **integration** with **ROS2 navigation** and **autonomy tools**



URDF

(**U**nified **R**obot **D**escription **F**ormat)



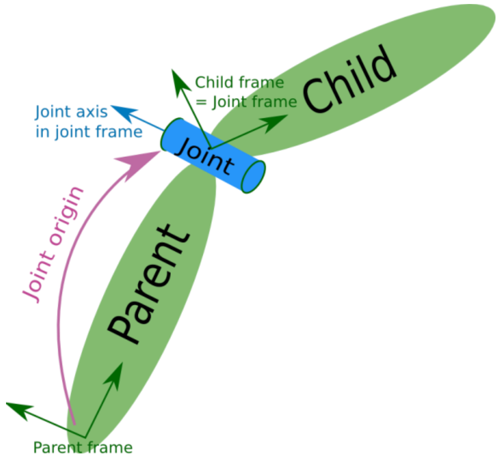
Why URDF?

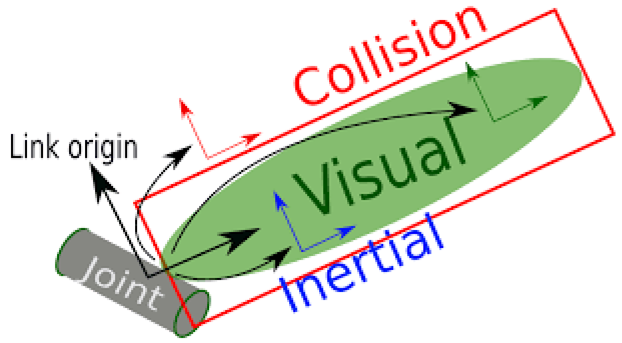
URDF is a **XML file** that provides a consistent and standardized way to define robot's structure using:

- **Links:** components of a robot (e.g. chassis, wheels, ...);
- **Joints:** connections between links, defining how they move relative to each other;

- **Sensors:** devices that perceive the environment (e.g. LiDAR, cameras, ...);

- **Visual Meshes:** 3D models used for visualization;
- **Collision Meshes:** semplified representations used for collision detections in simulations



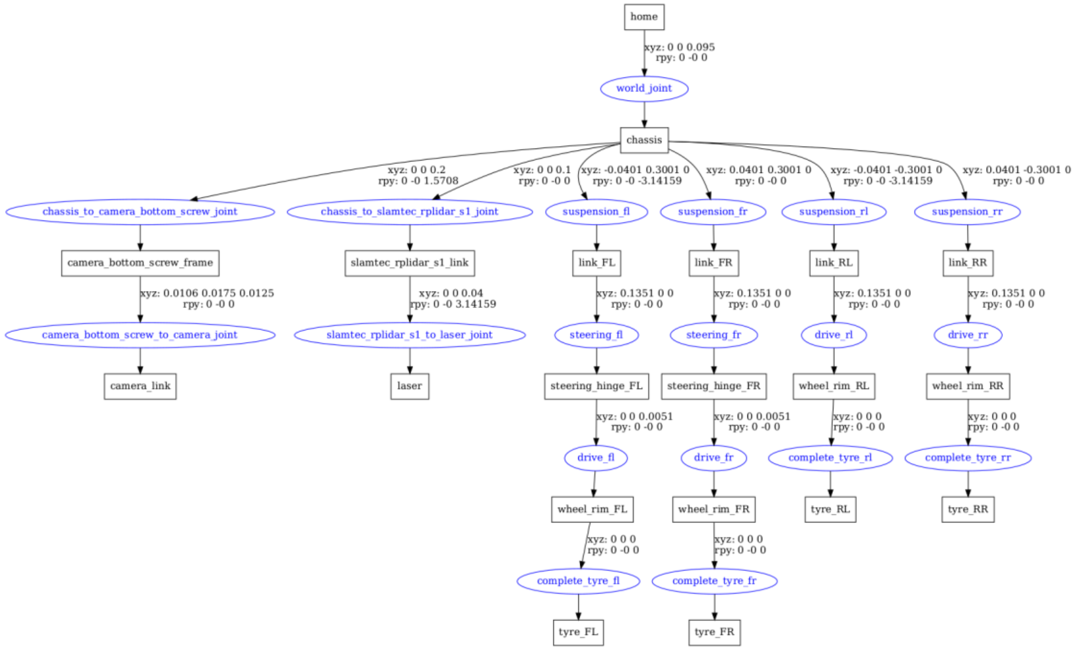


URDF structure

File structure

```xml
<?xml version="1.0" ?>
<robot name="psd_vehicle" xmlns:xacro="http://www.ros.org/wiki/xacro">
    <!-- LINKS -->

    <!-- Chassis START-->
    <link name="chassis">
      <inertial>
        <origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
        <mass value="10.0"/>
        <inertia ixx="0.354236" iyy="0.082009" izz="0.401568" ixy="0.0" iyz="0.0" ixz="0.0"/>
      </inertial>
      <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <mesh filename="$(find psd_vehicle_description)/meshes/stl/chassis.stl" scale="0.001 0.001 0.001"/>
        </geometry>
        <material name="anodized_red"/>
      </visual>
      <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
          <mesh filename="$(find psd_vehicle_description)/meshes/stl/chassis.stl" scale="0.001 0.001 0.001"/>
        </geometry>
      </collision>
    </link>

    <joint name="world_joint" type="fixed">
      <xacro:insert_block name="origin" />
      <parent link="${parent}" />
      <child link="chassis" />
    </joint>
</robot>
```

Graph structure



How to create it:

xacro your_robot.urdf.xacro | urdf_to_graphviz robot.pdf

Simplify URDF using XACRO

An usefull macro language developed by ROS dev is **XACRO**, that permits to:

- ***make it easier to maintain*** the robot description files;
- increase their ***readability***;
- ***avoid duplication*** in the robot description files;
- perform ***mathematical calculations within URDF***, reducing the need for external tools;
- ***easy to adjust robot parameters*** (dimensions, joint limits) without editing the entire URDF.

Without using XACRO

```xml
<link name="wheel_fl">
  <visual>
    <geometry>
      <cylinder length="0.5" radius="0.1"/>
    </geometry>
  </visual>
</link>

<link name="wheel_rl">
  <visual>
    <geometry>
      <cylinder length="0.3" radius="0.08"/>
    </geometry>
  </visual>
</link>
```

Using XACRO

```xml
<xacro:macro name="wheel" params="name length radius">
  <link name="${name}">
    <visual>
      <geometry>
        <cylinder length="${length}" radius="${radius}"/>
      </geometry>
    </visual>
  </link>
</xacro:macro>

<xacro:wheel name="wheel_fl" length="0.5" radius="0.1"/>
<xacro:wheel name="wheel_rl" length="0.3" radius="0.08"/>
```

General workflow

To start creating the robot representation we need to:

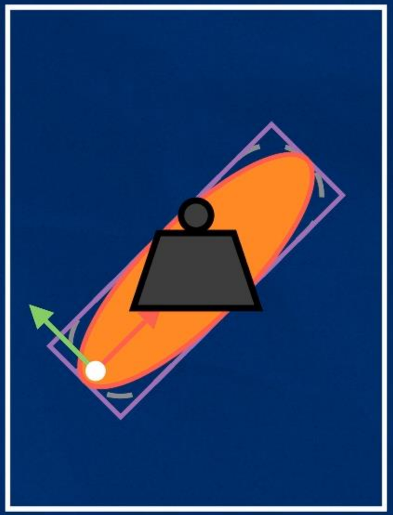
1. **Prepare 3D model** of the robot (e.g. using SolidWorks, Fusion360, FreeCAD, …)

2. **Export** each components in **STL/OBJ format** for mesh representation

3. **Convert to URDF**:
    - **Manually** (tedious, but good to start)
    - Using **automated tools** like:
        - i. **SW2URDF** (Plugin for SolidWorks)
        - ii. **Fusion2urdf** (Plugin for Fusion360)

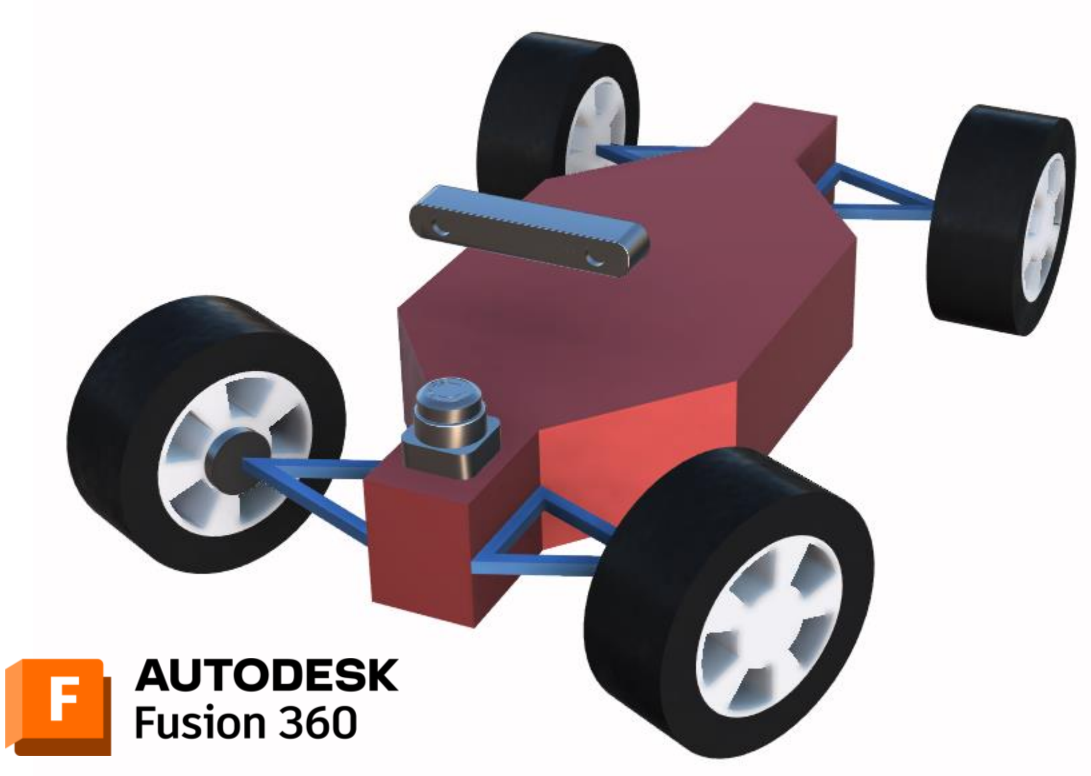
4. **Refine and test**

```
<link name="arm_link">
    <visual>
        <geometry>
        <origin>
        <material>
    </visual>

    <collision>
        <geometry>
        <origin>
    </collision>

    <inertial>
        <mass>
        <origin>
        <inertia>
    </inertial>
</link>
```



3D model

e.g. using **Fusion 360**



Parts	Materials
Chassis	Red anodized aluminum
Suspension (link)	Blue anodized aluminum
Wheel	White ABS plastics
Tyre	Vulcanized black rubber
Lidar	*(not designed)*
Camera	*(not designed)*

STL/OBJ export

PSD_VEHICLE_DESCRIPTION ...

```
> launch
v meshes
  v obj+mtl
      chassis.mtl
      chassis.obj
      steering_hinge.mtl
      steering_hinge.obj
      suspension.mtl
      suspension.obj
      tyre.mtl
      tyre.obj
      wheel.mtl
      wheel.obj
  v stl
      chassis.stl
      steering_hinge.stl
      suspension.stl
      tyre.stl
      wheel.stl
> rviz
v urdf
    materials.xacro
    psd_vehicle_macro.ros2_control.xacro
    psd_vehicle_macro.urdf.xacro
    psd_vehicle.urdf.xacro
  CMakeLists.txt
  package.xml
```

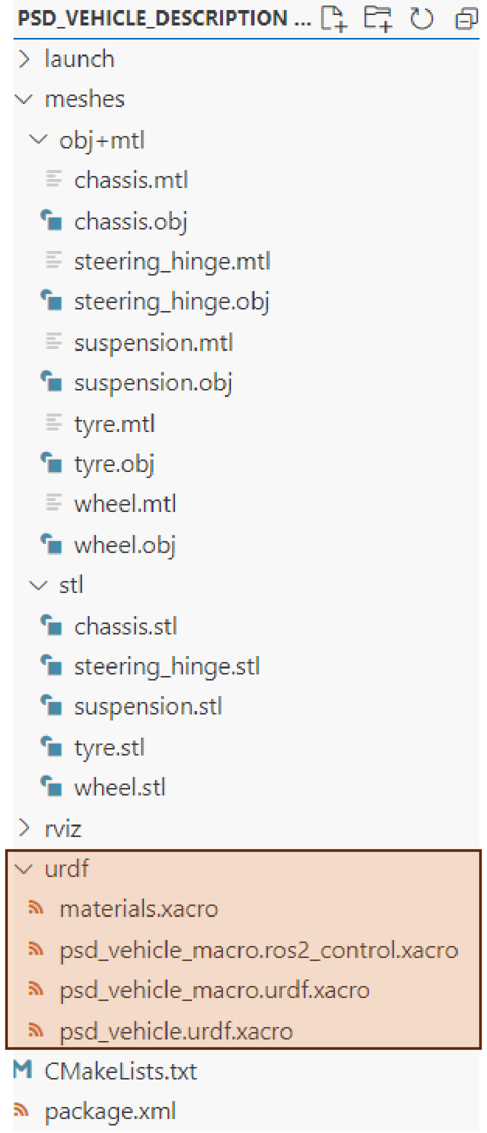
Inside the **PSD_VEHICLE_DESCRIPTION** package, we'll create a folder (in this case named *meshes*) that contains all the exported files

Each component will be included in the *link* definition to describe the **visual** and (eventually) also the **collison mesh**

Achtung!
All measures in URDF file are considered in **meters for lenghts** and **radians for angles**

```xml
<!-- Chassis START-->
<link name="chassis">
  <inertial>
    <origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
    <mass value="10.0"/>
    <inertia ixx="0.354236" iyy="0.082009" izz="0.401568" ixy="0.0" iyz="0.0" ixz="0.0"/>
  </inertial>
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="$(find psd_vehicle_description)/meshes/stl/chassis.stl" scale="0.001 0.001 0.001"/>
    </geometry>
    <material name="anodized_red"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="$(find psd_vehicle_description)/meshes/stl/chassis.stl" scale="0.001 0.001 0.001"/>
    </geometry>
  </collision>
</link>
```

URDF and XACRO description



The **urdf/** folder contains the complete description of the vehicle.
In particular:

- **materials.xacro** defines the materials used for each component of the car (e.g., metal for the chassis, rubber for the tires);
- **psd_vehicle.urdf.xacro** is the «main» file that combine the others in this folder to complete the description;
- **psd_vehicle_macro.urdf.xacro** contains the *classical* description of the car with all the links, joints (for the dynamics and kinematics) and sensors on board;
- **psd_vehicle_macro.ros2_control.xacro** is a description of the way each joint will be actuated and how we want to control it *(more information later).*

URDF and XACRO description

e.g. **materials.xacro :**

```xml
<material name="rubber_tyre">
  <color rgba="0.0 0.0 0.0 1.000"/>
  <lighting>1</lighting> <!-- Enable lighting effects -->
  <ambient>0.1 0.1 0.1 1.0</ambient> <!-- Ambient light color -->
  <diffuse>0.6 0.6 0.6 1.0</diffuse> <!-- Diffuse reflection color -->
  <specular>0.3 0.3 0.3 1.0</specular> <!-- Specular reflection color -->
  <emissive>0.0 0.0 0.0 1.0</emissive> <!-- Emissive color -->
  <transparency>0.0</transparency> <!-- Transparency (0.0 for opaque) -->
  <friction>
    <ode>
      <mu>0.8</mu> <!-- Friction coefficient -->
      <mu2>0.8</mu2> <!-- Friction coefficient -->
    </ode>
  </friction>
  <elasticity>
    <ode>
      <epsilon>0.8</epsilon> <!-- Coefficient of restitution
      <kp>1000.0</kp> <!-- Contact stiffness -->
      <kd>10000.0</kd> <!-- Contact damping -->
    </ode>
  </elasticity>
</material>
```

`<gazebo>` Elements For Links

List of elements that are individually parsed:

Source: https://classic.gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros

Name	Type	Description
material	value	Material of visual element
gravity	bool	Use gravity
dampingFactor	double	Exponential velocity decay of the link velocity - takes the value and multiplies the previous link velocity by (1-dampingFactor).
maxVel	double	maximum contact correction velocity truncation term.
minDepth	double	minimum allowable depth before contact correction impulse is applied
mu1 mu2	double	Friction coefficients μ for the principal contact directions along the contact surface as defined by the Open Dynamics Engine (ODE) (see parameter descriptions in ODE's user guide)
fdir1	string	3-tuple specifying direction of mu1 in the collision local reference frame.
kp kd	double	Contact stiffness k_p and damping k_d for rigid body contacts as defined by ODE (ODE uses erp and cfm but there is a mapping between erp/cfm and stiffness/damping)
selfCollide	bool	If true, the link can collide with other links in the model.
maxContacts	int	Maximum number of contacts allowed between two entities. This value overrides the max_contacts element defined in physics.
laserRetro	double	intensity value returned by laser sensor.

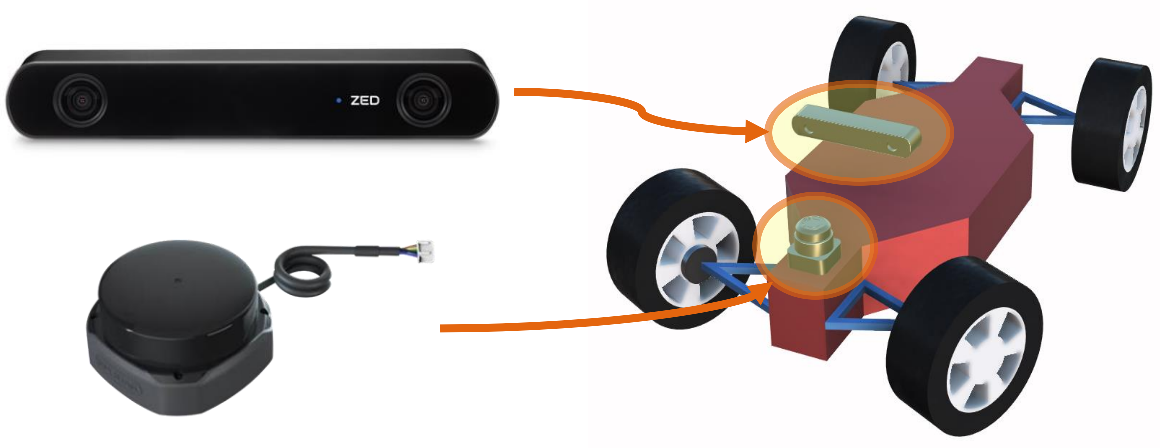
Similar to `<gazebo>` elements for `<robot>`, any arbitrary blobs that are not parsed according to the table above are inserted into the the corresponding `<link>` element in the SDF. This is particularly useful for plugins, as discussed in the ROS Motor and Sensor Plugins tutorial.

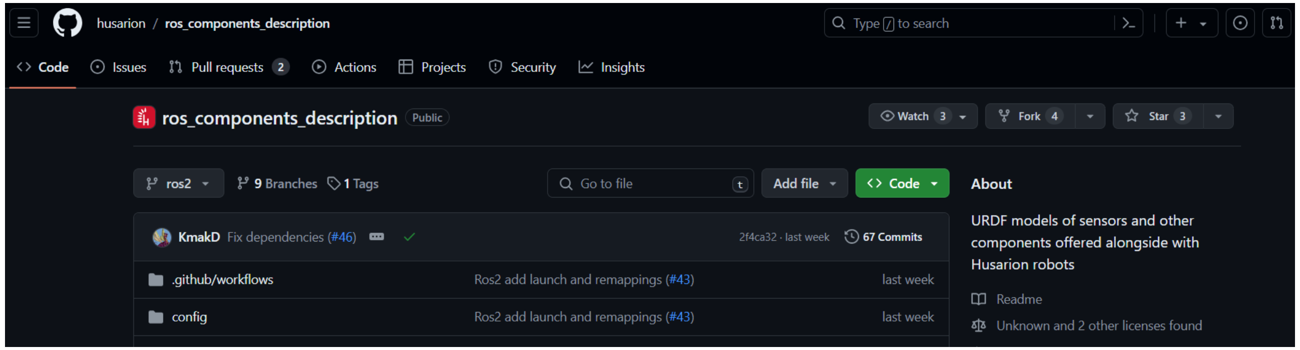
ros_components_description/

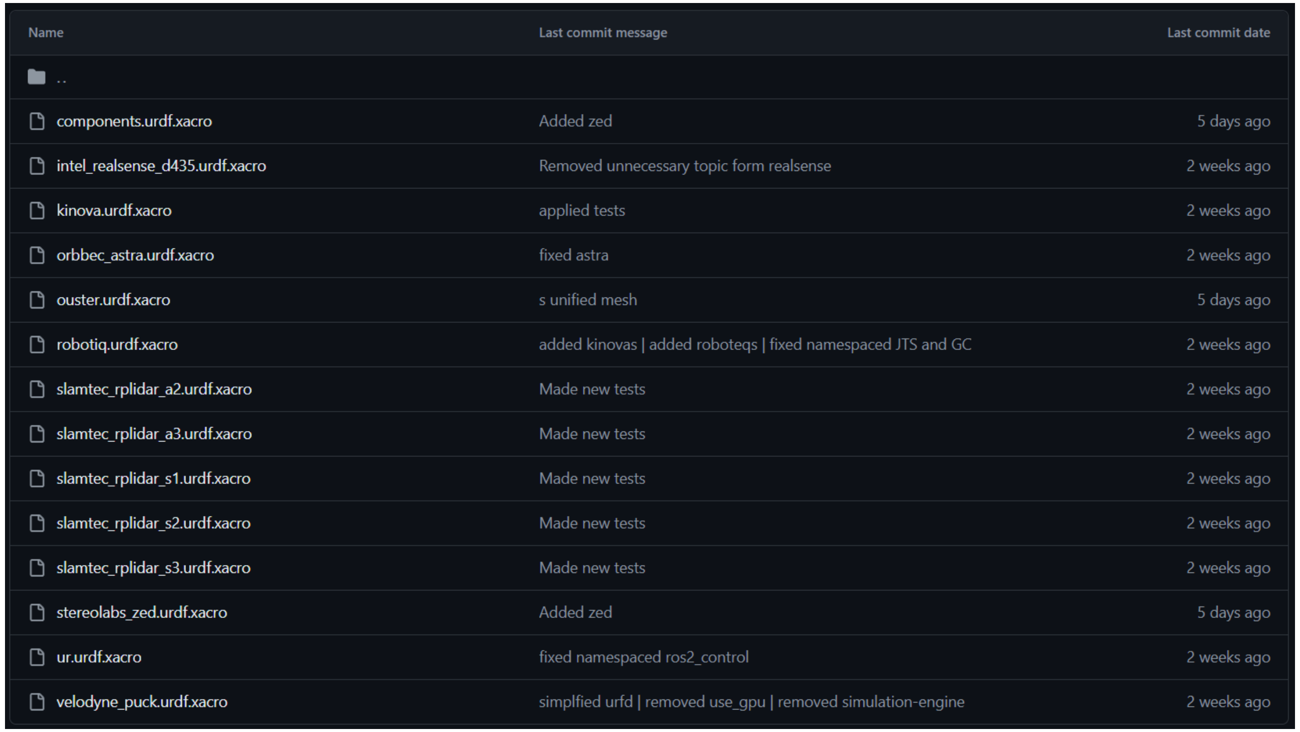
It's a repository that contains a really accurate description of sensors that exists in real world like:

- **Cameras:** Simulate various types (RGB, depth, stereo) with different resolutions, noise models, and distortion effects.

- **LiDAR:** Generate realistic point clouds with varying densities and ranges, simulating different LiDAR models.







Source:
https://github.com/husarion/ros_components_description/tree/ros2-zed

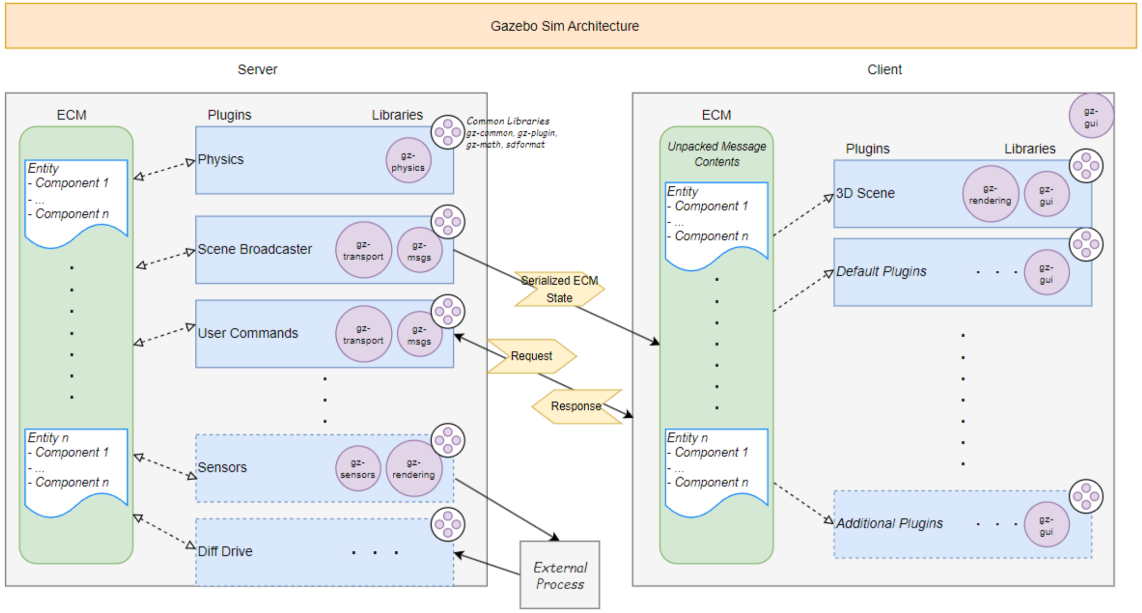
psd_gazebo_
worlds/

Gazebo Harmonic

Gazebo Harmonic is the **latest iteration** *(Supported Sep, 2023 to Sep, 2028)* of the open-source Gazebo robotics simulator **integrated with ROS2**. It supports **high-fidelity physics** simulation thanks to accurate sensor models, that makes it perfect to test code before real-world deployment.

Key Features:

- **Highly customizable:** thanks to its modular architecture based on plugin for specific needs;
- **Physics engines:** supports multiple engines like DART, Bullet, ODE or Simbody;
- **Sensor simulations:** offers various sensor models like cameras, LiDAR, IMUs, GPS and others;



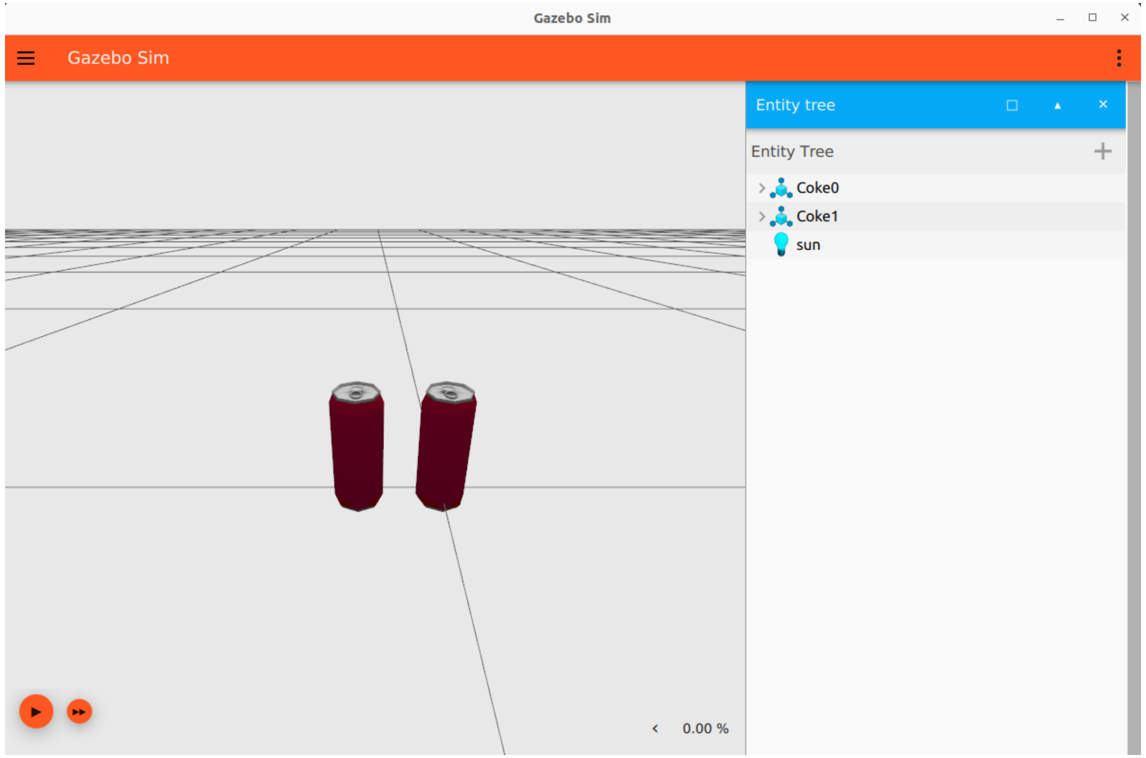
Describe world using SDF

SDFormat

Describe your world.

SDF (Simulation **D**escription **F**ormat**)** is an XML format that **describes objects** and **environments** for robot simulators, visualization and control.

```
<?xml version="1.0" ?>
<sdf version="1.8">
  <world name="world_demo">
    <physics name="1ms" type="ignored">
      <max_step_size>0.001</max_step_size>
      <real_time_factor>1.0</real_time_factor>
    </physics>
    <plugin
      filename="gz-sim-physics-system"
      name="gz::sim::systems::Physics">
    </plugin>
    <plugin
      filename="gz-sim-user-commands-system"
      name="gz::sim::systems::UserCommands">
    </plugin>
    <plugin
      filename="gz-sim-scene-broadcaster-system"
      name="gz::sim::systems::SceneBroadcaster">
    </plugin>

    <!--light-->
    <light type="directional" name="sun">
      <cast_shadows>true</cast_shadows>
      <pose>0 0 10 0 0 0</pose>
      <diffuse>0.8 0.8 0.8 1</diffuse>
      <specular>0.2 0.2 0.2 1</specular>
      <attenuation>
        <range>1000</range>
        <constant>0.9</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <direction>-0.5 0.1 -0.9</direction>
    </light>
    <include>
        <name>Coke0</name>
        <pose>0 0 0 0 0 0</pose>
        <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models/Coke</uri>
    </include>
    <include>
        <name>Coke1</name>
        <pose>0 0.1 0 0 0 0</pose>
        <uri>https://fuel.gazebosim.org/1.0/OpenRobotics/models/Coke</uri>
    </include>
  </world>
</sdf>
```



psd_gazebo_worlds/

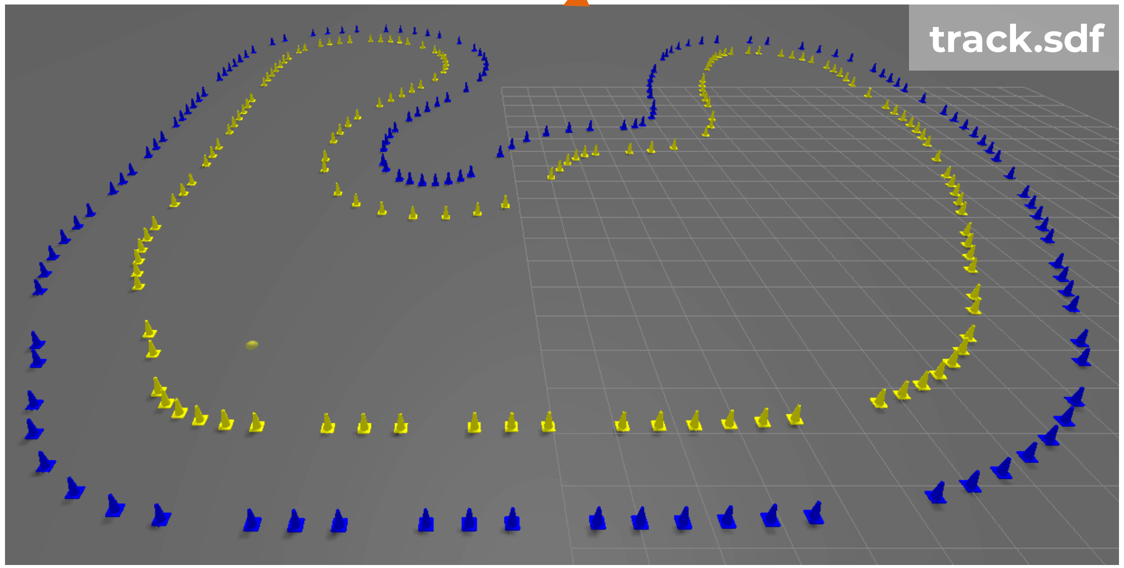
```
psd_gazebo_worlds/
├── CMakeLists.txt
├── package.xml
└── world/
    ├── models/
    │   └── cone/
    │       ├── meshes/
    │       │   ├── cone.dae
    │       │   └── cone.stl
    │       ├── model.config
    │       └── model.sdf
    ├── track_creator.py
    └── track.sdf
```

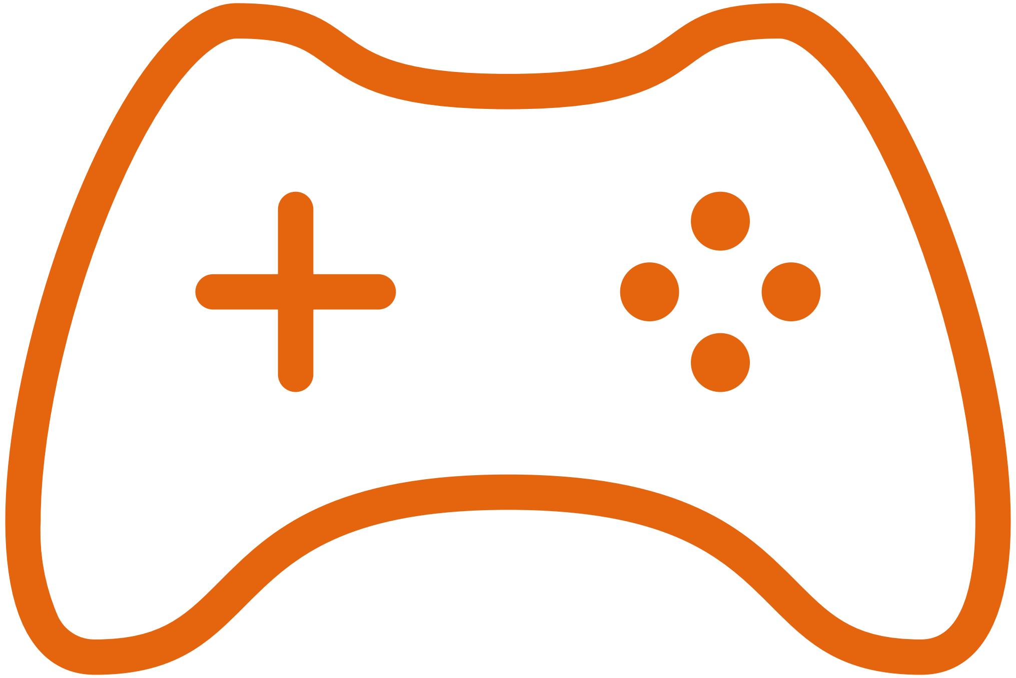
```python
def main():
    innerConePosition = np.array([[6.49447171658257,41.7389113024907, 0, 1],[
    outerConePosition = np.array([[8.29483356036796,47.8005083348189, 0, 2],[

    scale_factor = 0.3
    scaled_inner = scale_track(innerConePosition, scale_factor)
    scaled_outer = scale_track(outerConePosition, scale_factor)

    sdf_content = generate_sdf(scaled_inner, scaled_outer)

    with open("track.sdf", "w") as f:
        f.write(sdf_content)

if __name__ == "__main__":
    main()
```

track_creator.py

```xml
<?xml version="1.0" ?>
<sdf version="1.9">
  <model name="Cone">
    <link name="link">
      <collision name="collision">
        <geometry>
          <mesh>
            <scale>1 1 1</scale>
            <uri>model://cone/meshes/cone.stl</uri>
          </mesh>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <mesh>
            <scale>1 1 1</scale>
            <uri>model://cone/meshes/cone.stl</uri>
          </mesh>
        </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

model.sdf



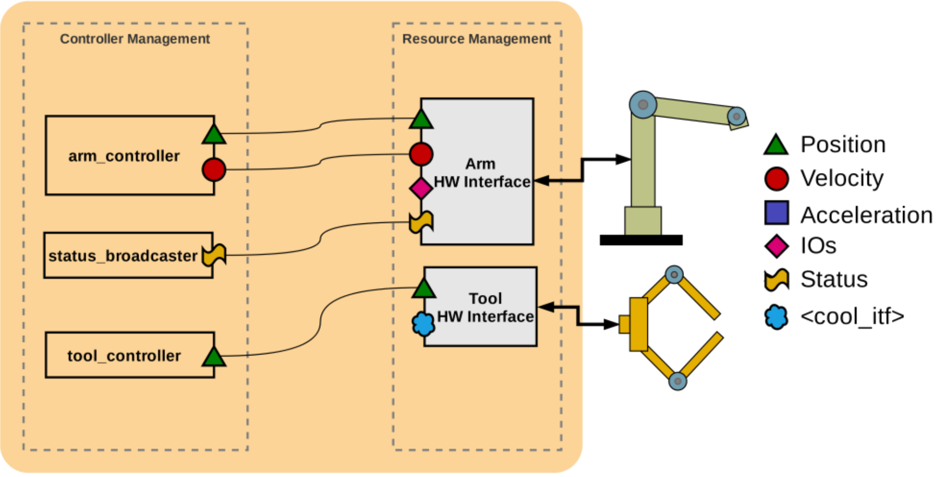


psd_gazebo_ bringup/

psd_vehicle_bringup/

```
├── CMakeLists.txt
├── LICENSE
├── config/
│   └── psd_vehicle_controllers.yaml
├── launch/
│   └── gz_sim.launch.py
└── package.xml
```

Contains the **definition of controllers** and how they interact with specific joints



This launch file **sets up the robot simulation environment** using *Gazebo Gz* and *ROS2.*

```python
IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                               'launch',
                               'gz_sim.launch.py'])]),
    launch_arguments=[('gz_args', [f' -r {world_file}'])]),
```

```python
gz_spawn_entity = Node(
    package="ros_gz_sim",
    executable="create",
    name="spawn_psd_vehicle",
    arguments=[
        "-name",
        "psd_vehicle",
        "-allow_renaming",
        "true",
        "-topic",
        "robot_description",
        "-x",
        "0.0",
        "-y",
        "0.0",
        "-z",
        "1.0",
    ],
    output="screen",
)
```

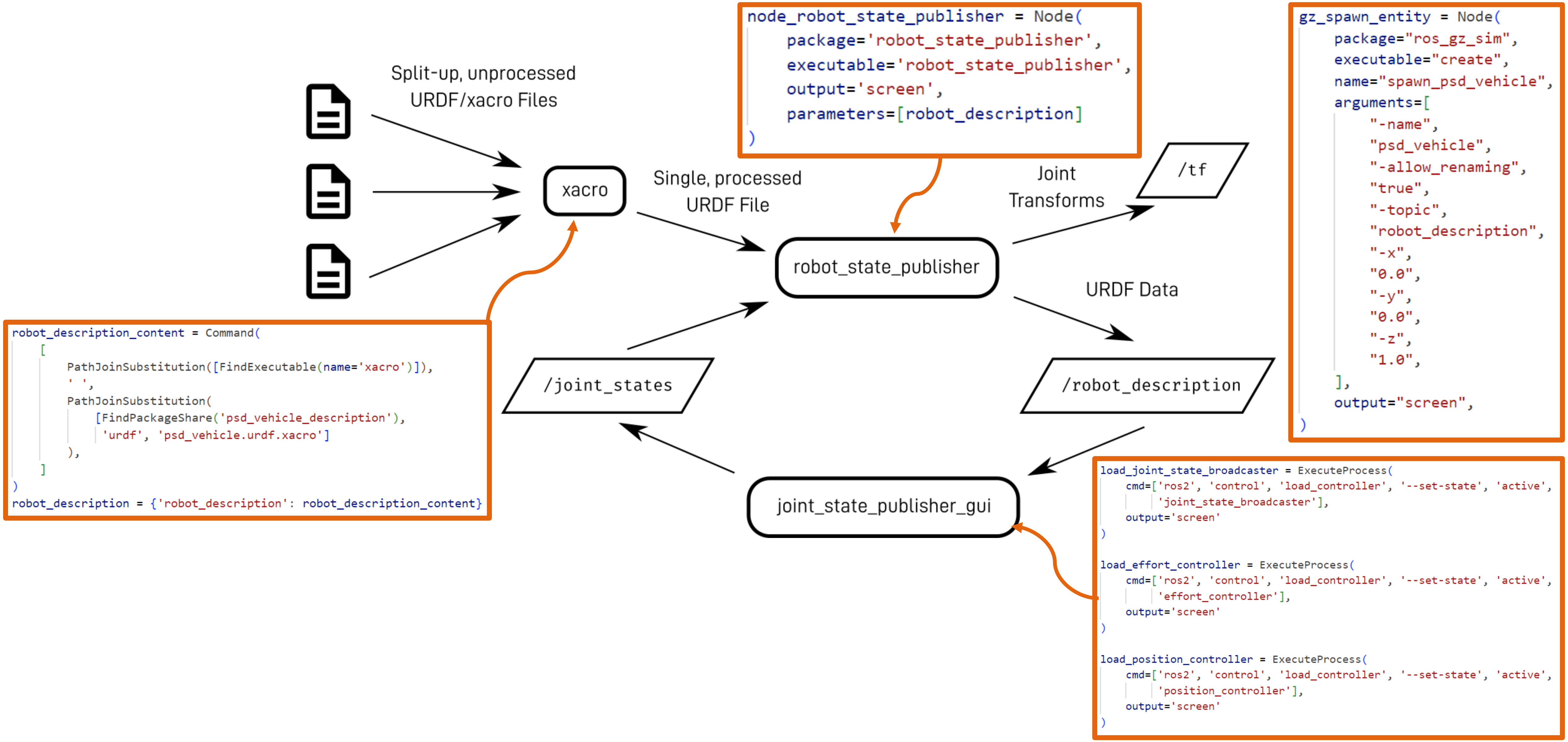
psd_vehicle_bringup/

```
├── CMakeLists.txt
├── LICENSE
├── config/
│     └── psd_vehicle_controllers.yaml
├── launch/
│     └── gz_sim.launch.py
└── package.xml
```

It’s needed to **map** the **topics** published on **Gazebo to ROS** (and **viceversa**)

```python
gz_bridge = Node(
    package='ros_gz_bridge',
    executable='parameter_bridge',
    parameters=[{"config_file": gz_bridge_config_path}],
    arguments=[
        "<ros_topic_1>" + "@" + "<ros_message_type_1>" + "[" + "<gazebo_topic_1>" + "]",
        "<ros_topic_2>" + "@" + "<ros_message_type_2>" + "[" + "<gazebo_topic_2>" + "]",
    ],
    remappings=[
        ("<original_ros_topic_1>", "<new_ros_topic_1>"),
        ("<original_ros_topic_2>", "<new_ros_topic_2>"),
    ],
    output='screen'  # Optional: Directs node's output to the terminal screen
)
```

gz_sim.launch.py




```python
robot_description_content = Command(
    [
        PathJoinSubstitution([FindExecutable(name='xacro')]),
        ' ',
        PathJoinSubstitution(
            [FindPackageShare('psd_vehicle_description'),
             'urdf', 'psd_vehicle.urdf.xacro']
        ),
    ]
)
robot_description = {'robot_description': robot_description_content}
```

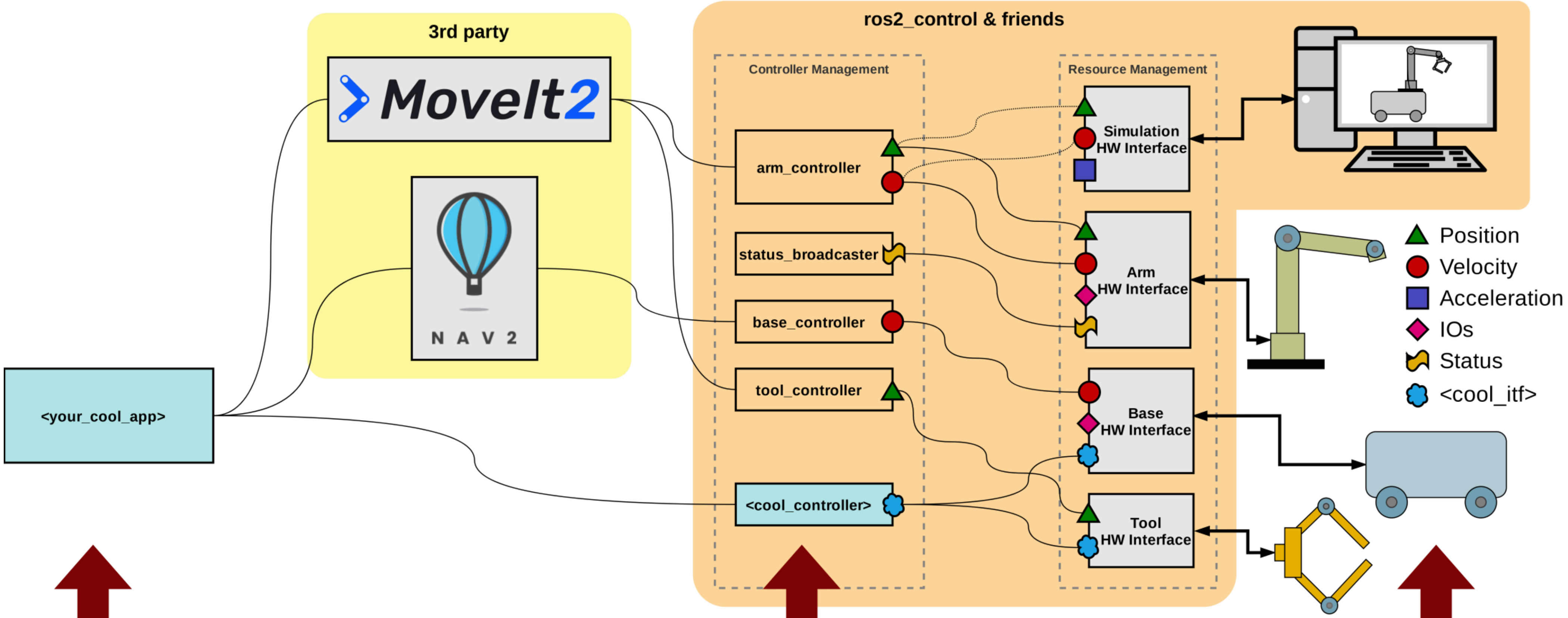
```python
node_robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    output='screen',
    parameters=[robot_description]
)
```

```python
gz_spawn_entity = Node(
    package="ros_gz_sim",
    executable="create",
    name="spawn_psd_vehicle",
    arguments=[
        "-name",
        "psd_vehicle",
        "-allow_renaming",
        "true",
        "-topic",
        "robot_description",
        "-x",
        "0.0",
        "-y",
        "0.0",
        "-z",
        "1.0",
    ],
    output="screen",
)
```

```python
load_joint_state_broadcaster = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'joint_state_broadcaster'],
    output='screen'
)

load_effort_controller = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'effort_controller'],
    output='screen'
)

load_position_controller = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'position_controller'],
    output='screen'
)
```

Ros2_control

Basic structure of the complete controller setup



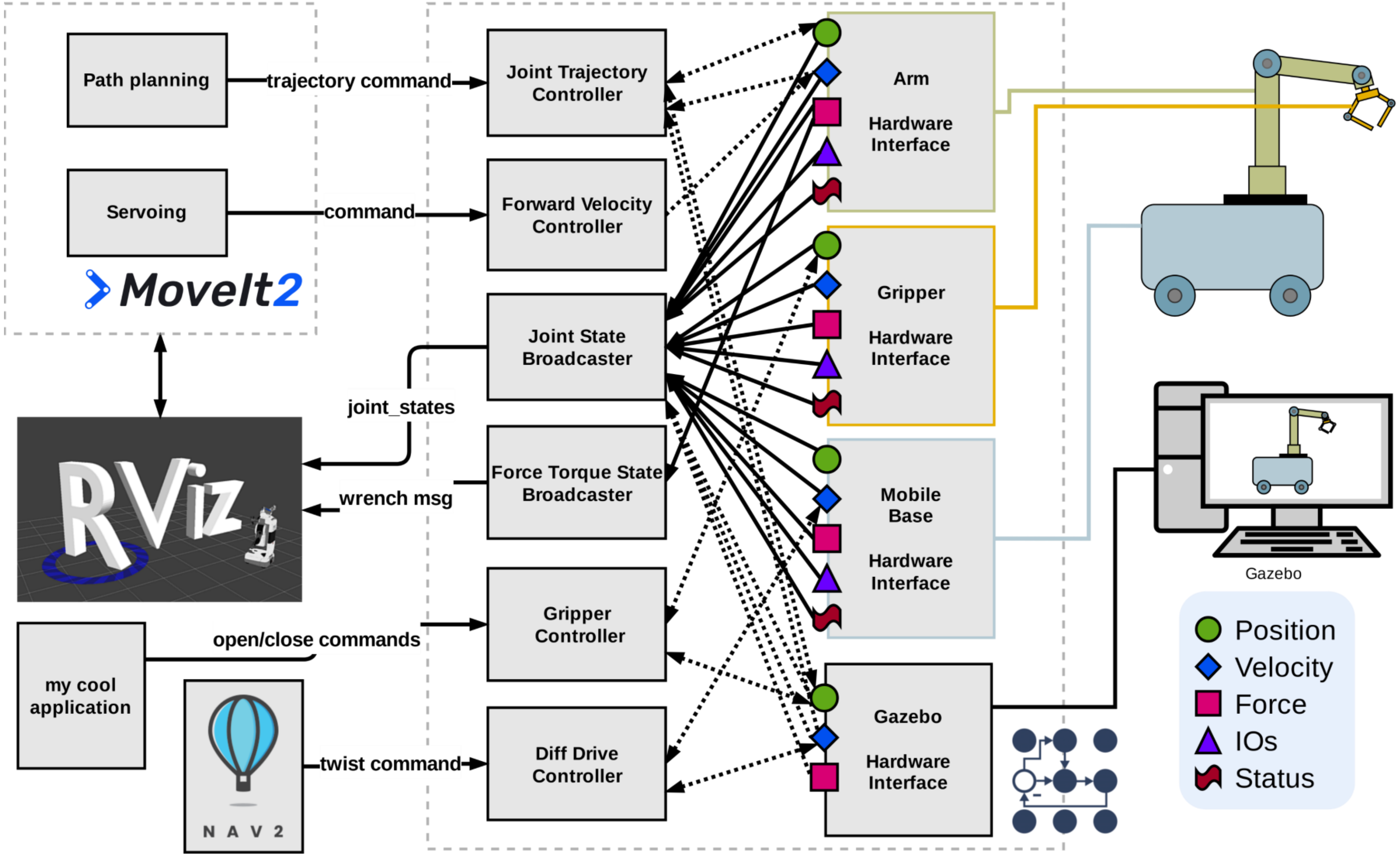
Maximize resources spent on actual **client application**

- **Leverage** existing controllers
- **Implement** custom ones, extend existing

- **Leverage** simulation backend
- **Real robot** backend - extend existing ones or create your own

Ros2_control

Complete structure of the complete controller setup



Ros2_control in URDF

The controls definition to actuate the joints are defined inside the
psd_vehicle_description/urdf/psd_vehicle_macro.ros2_control.xacro

```xml
<?xml version='1.0'?>
<robot xmlns:xacro="http://wiki.ros.org/xacro">
  <xacro:macro name="psd_vehicle_ros2_control" params="
               name"
               >

    <!-- control -->
    <xacro:property name="PI" value="3.14159265359" />

    <xacro:property name="max_torque" value="100" />
    <xacro:property name="max_steering_angle" value="${20 * PI / 180}" />

    <ros2_control name="GazeboSimSystem" type="system">
      <hardware>
        <plugin>gz_ros2_control/GazeboSimSystem</plugin>
      </hardware>
```

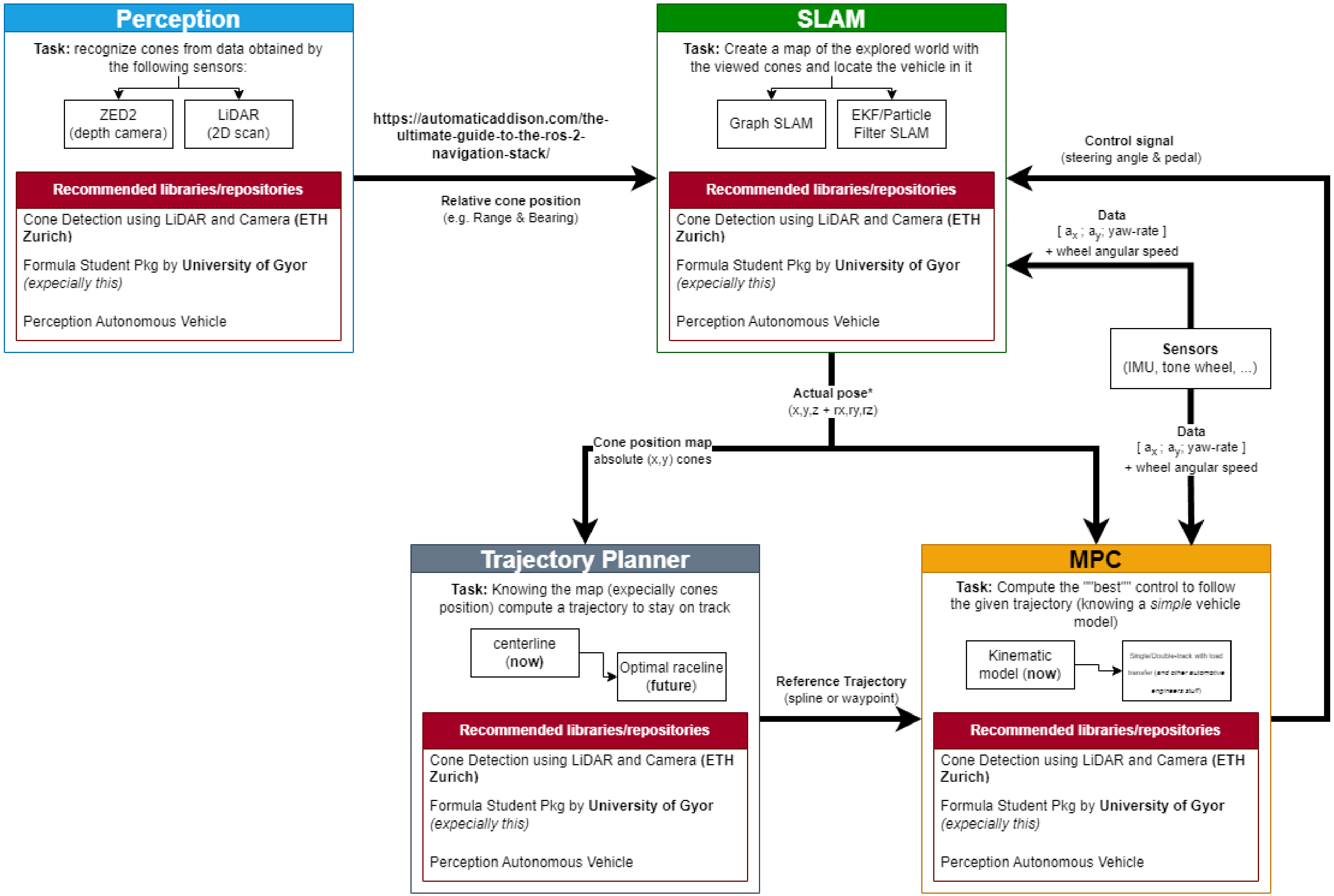
Joint actuation control

```xml
    </ros2_control>

    <gazebo>
      <!-- Joint state publisher -->
      <plugin filename="gz_ros2_control-system" name="gz_ros2_control::GazeboSimROS2ControlPlugin">
        <parameters>$(find psd_vehicle_bringup)/config/psd_vehicle_controllers.yaml</parameters>
      </plugin>
    </gazebo>

  </xacro:macro>
</robot>
```

```xml
<!-- steering control -->
<joint name="steering_fr">
  <command_interface name="position">
    <param name="min">"${-max_steering_angle}"</param>
    <param name="max">"${max_steering_angle}"</param>
  </command_interface>
  <state_interface name="position">
    <param name="initial_value">0.0</param>
  </state_interface>
</joint>

<!-- driving control -->
<joint name="drive_rl">
  <command_interface name="effort">
    <param name="min">"${-max_torque}"</param>
    <param name="max">"${max_torque}"</param>
  </command_interface>
  <state_interface name="position"/>
  <state_interface name="velocity"/>
  <state_interface name="effort"/>
</joint>
```

psd_stack/

Autonomous driving code stack



Bibliography and websites

ROS2 Jazzy Jalisco:

- https://docs.ros.org/en/jazzy/Tutorials.html

Docker container:

- https://www.docker.com/resources/what-container/

URDF:

- https://wiki.ros.org/urdf
- https://gazebosim.org/docs/harmonic/spawn_urdf
- https://articulatedrobotics.xyz/tutorials/mobile-robot/concept-design/concept-urdf/

SDF/Gazebo:

- https://gazebosim.org/docs
- https://gazebosim.org/docs/harmonic/sdf_worlds
- http://sdformat.org/spec

ROS2_Control:

- https://control.ros.org/jazzy/doc/resources/resources.html#diagrams