{Learn, Create, Innovate};

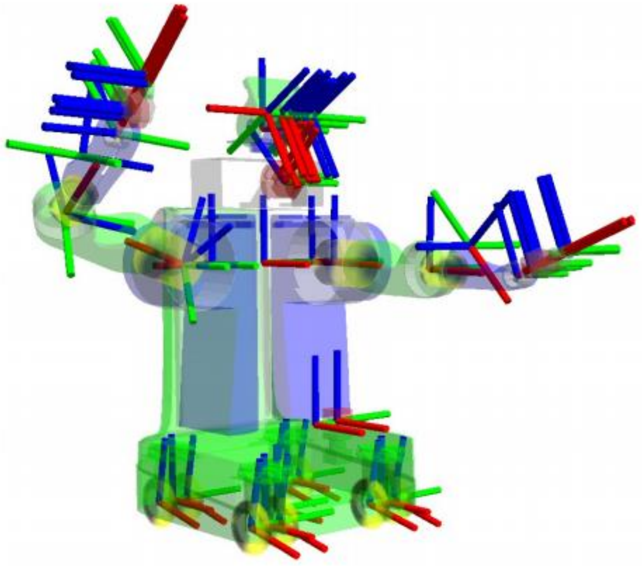
ROS

Transforms

Manchester Robotics

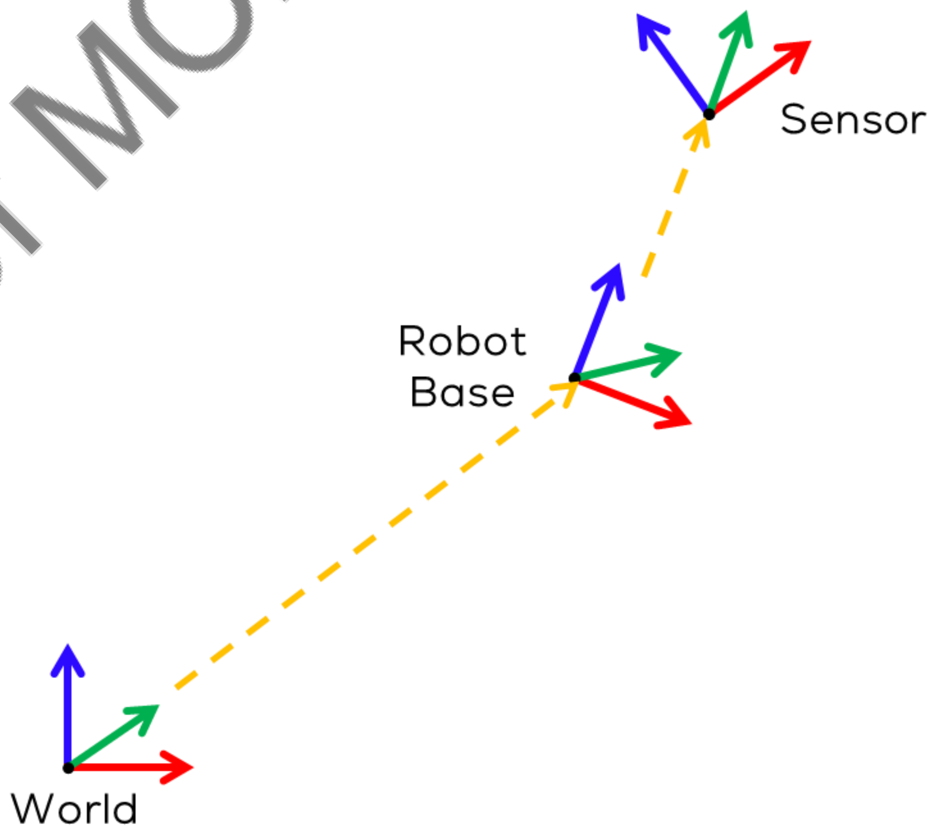
Coordinate transformations in ROS

- Coordinate transformations refer to the process of converting coordinates from one coordinate system to another.
- Coordinate transformation, maps points or vectors from one reference frame to another, typically using mathematical equations or transformations.
- The purpose of coordinate transformations is to describe the same object or phenomenon in different coordinate systems or to simplify calculations in a specific frame of reference.
- These transformations can include rotations, translations, scaling, or combinations thereof, depending on the nature of the coordinate systems involved.



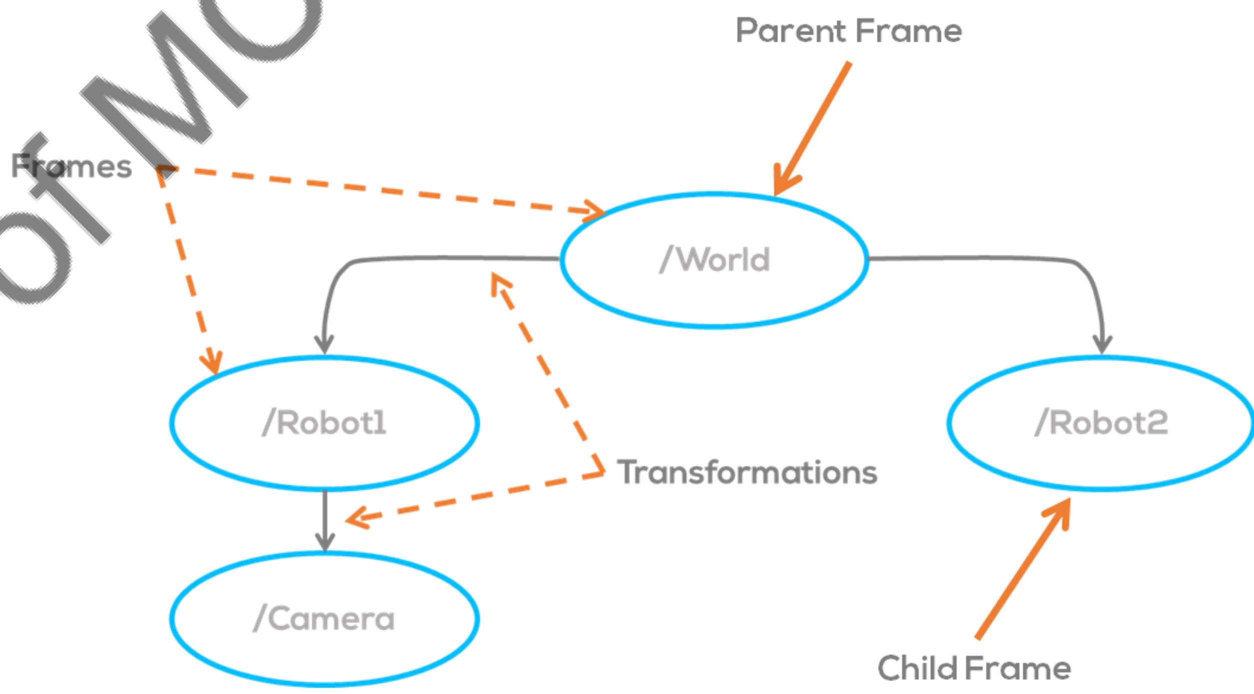
Coordinate transformations in ROS

- The tf library was created to establish a consistent method of monitoring coordinate frames and transforming data across the entire system.
- This ensures that users of individual components can trust that the data is in the desired coordinate frame without knowing all the coordinate frames used throughout the system.



Coordinate transformations in ROS

- The tf library is based on a tree structure, where each node represents a coordinate frame.
- The tree is rooted in a fixed frame, usually called the "world“ or “map” frame, which is typically a global reference frame.
- Each node in the tree represents a specific coordinate frame attached to a specific robot component through a transformation, such as a sensor or an actuator.

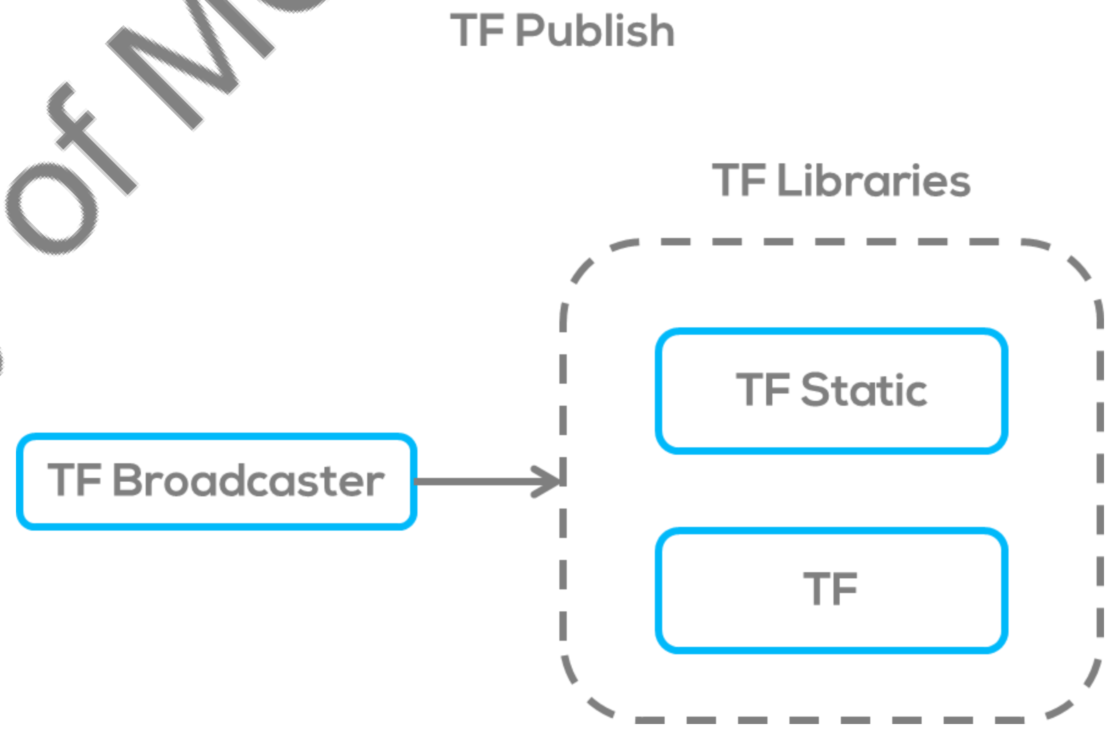


Coordinate transformations in ROS

The tf library provides two main functionalities:

1. **Broadcasting transformations:** Each component of the robot that has a coordinate frame associated with it can publish its transformation with respect to another frame.
    - For example, a sensor mounted on a robot arm may publish its transformation with respect to the robot's base frame.

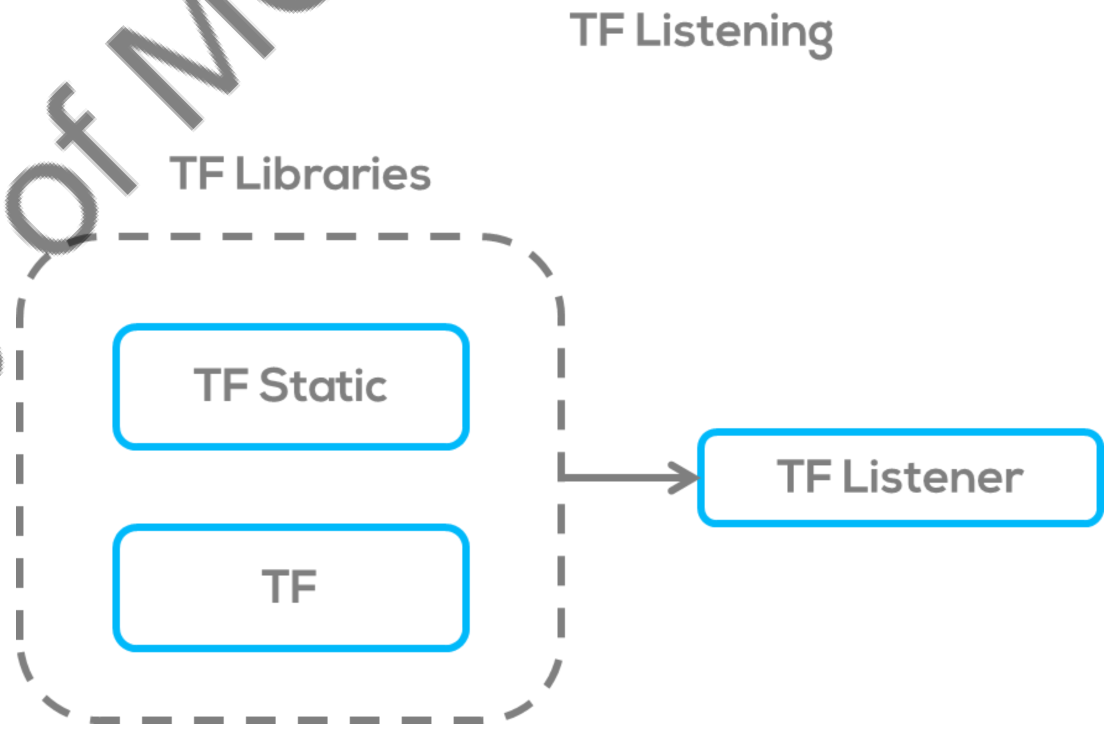
- These transformations are broadcasted over the ROS network, allowing other components to subscribe and receive updates.



Coordinate transformations in ROS

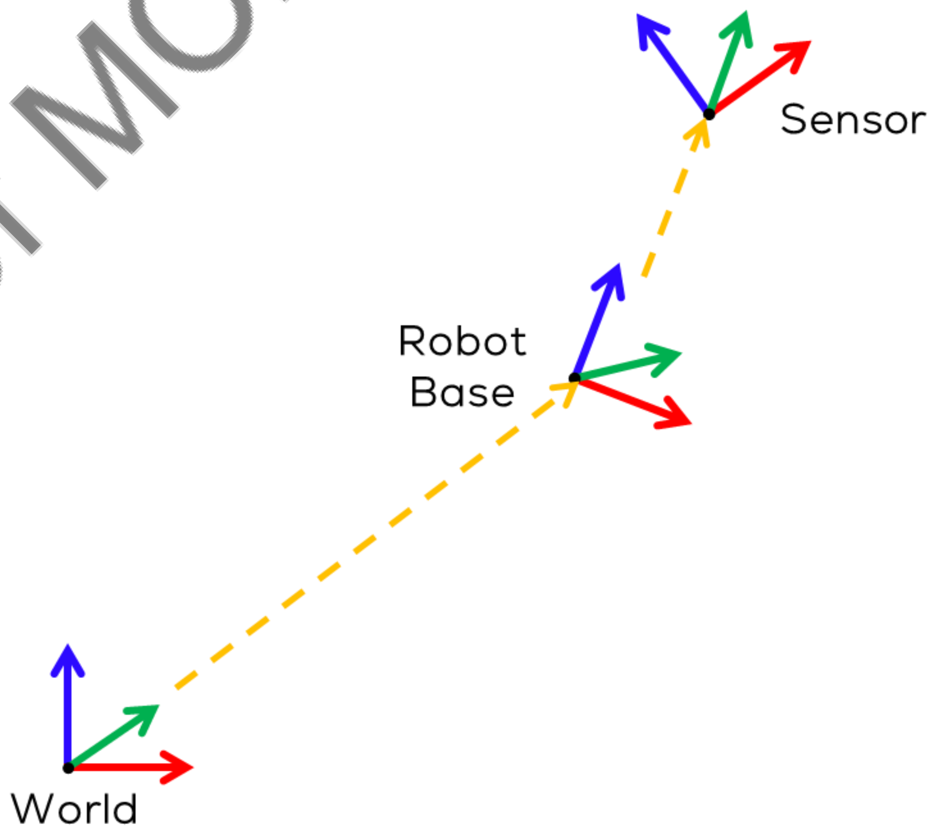
2. **Listening to transformations:** Components that need to know the transformation between two frames can subscribe to these transformations using tf listeners.
    - The listener keeps track of the transformations being broadcasted and allows components to query the transformation between any two frames at any given time as long as they are connected in the tree.

- This is particularly useful for performing coordinate frame transformations on points or vectors from different parts of the system.



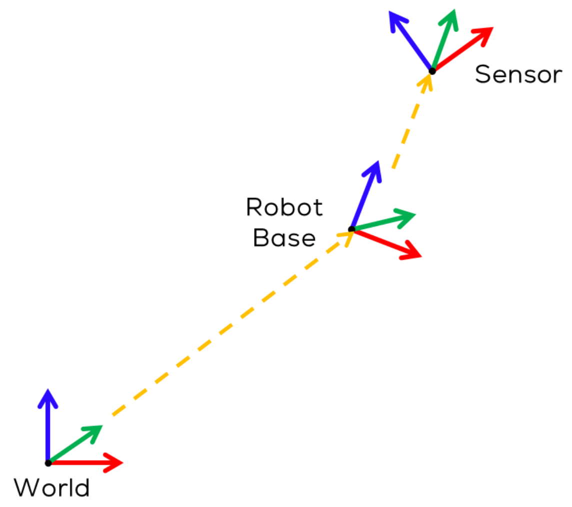
How it works?

- **Frame definitions:** Each component that needs to be tracked defines its own coordinate frame(s) and their relationship to other frames in the system.
    - For example, a robot arm may have a base frame and an end effector frame.
- **Broadcasting transformations**: Components with a defined frame can publish their transformations using a tf broadcaster. They periodically update the transformations based on their current state or sensor readings and broadcast them over the ROS network.



How it works?

- **Listening to transformations:** Components that need to use the transformations can create tf listeners. The listeners subscribe to the transformations being broadcasted and maintain an up-to-date view of the coordinate frame tree.
- **Querying transformations:** Components can query the tf listener for the transformation between two frames using the appropriate tf function.
  - For example, to transform a point from the sensor frame to the robot's base frame, a component would use the tf listener to get the transformation between the frames and apply it to the point.
- **Managing coordinate frame updates:** The library manages coordinate frame updates.
  - It handles situations where transformations arrive out of order, compensates for time delays, and interpolates between transformations to provide accurate and smooth frame transformations.



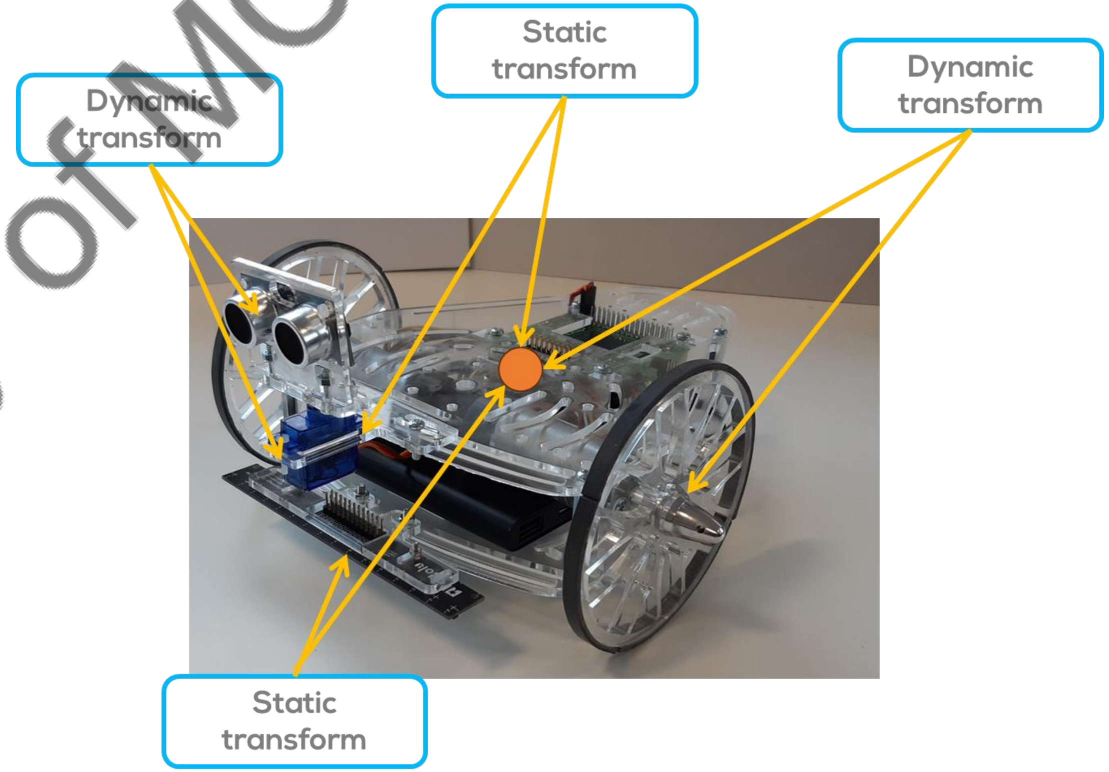
How it works?

Transform types:

- **Static:** Transforms that do not change over time. Sensor positions, actuator positions, etc.
- **Dynamic:** Can change over time. Sensor information frames of reference, other robots, etc.

The reason for having different transforms is that transformations that vary over time require to know if their information is out of date, to report an error if the broadcaster hasn't updated the transform over a period of time.

Static transforms, however, can be broadcast once and are assumed to be correct until a new one is broadcast.



Declaring a transform

- Usually, transforms (static and dynamic) are declared inside the script where the information is published.
- **Static transforms,** however (Only static transforms) can be also declared inside launch files, without needing to be compiled.
- This is because static transforms are expected to not change over time.
- The static transform requires the following information

```
static_transform_publisher x y z yaw pitch roll frame_id child_frame_id
```

- In command line this will look as follows

```
$ ros2 run tf2_ros static_transform_publisher x y z yaw pitch roll parent_frame child_frame
```

Activity 1

Static Transforms from command line

`{Learn, Create, Innovate};`

Manchester Robotics

Static Transform Example

Declaring a Static Transform from command line

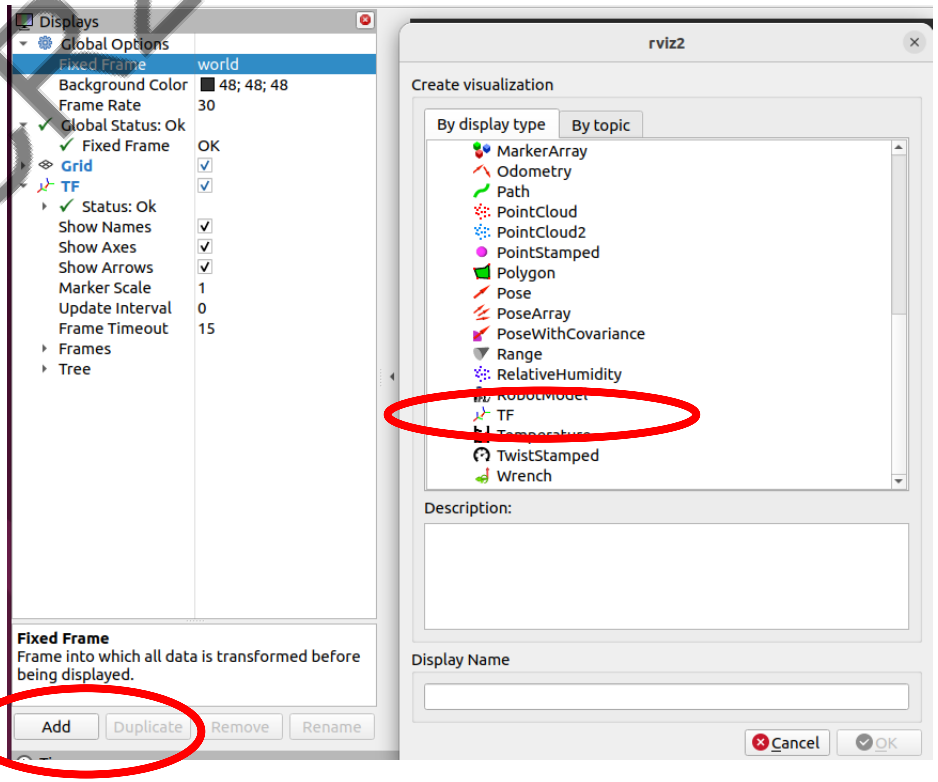
- Open RVIZ from a terminal as follows

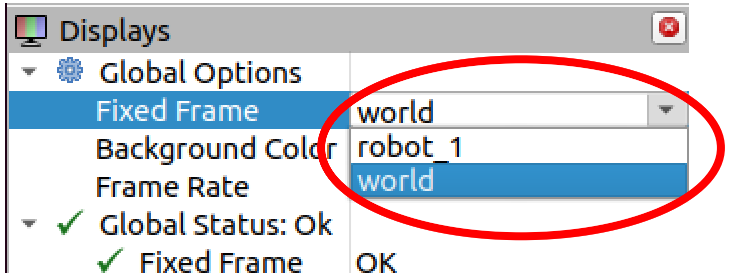
```
$ ros2 run rviz2 rviz2
```

- Open another terminal and type the following

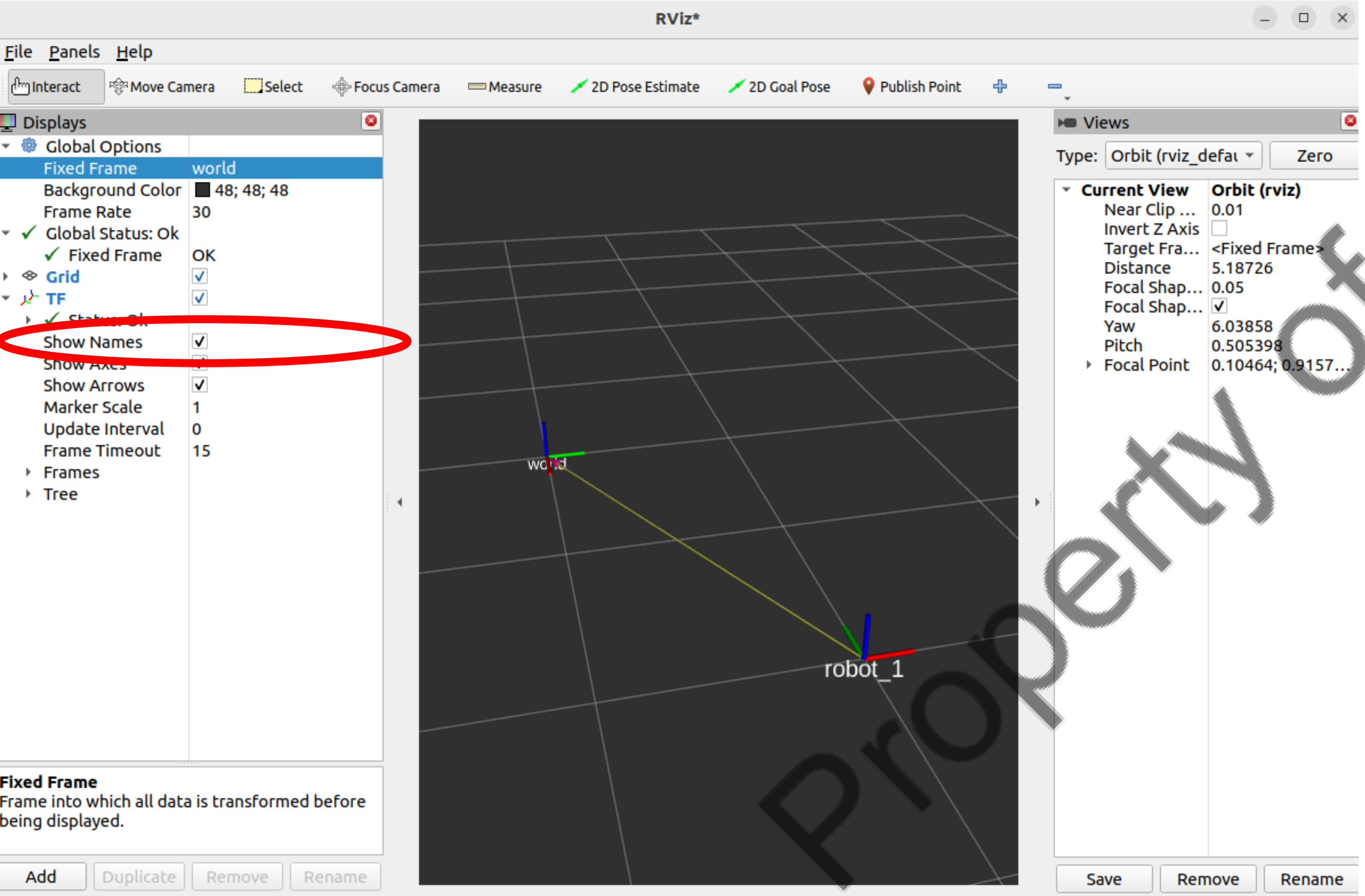
```
$ ros2 run tf2_ros static_transform_publisher 2 1 0 1.57 0 0 world robot_1
```

- In RVIZ press the "Add" button on the bottom left corner.
- On the Pop-Up windows select "TF" and press "OK"
- On the top right corner of RVIZ, select the "Fixed Frame" to be "world"
    - This steps tells RViz which frame to use as a reference (it defaults to map, which doesn't exist right now).



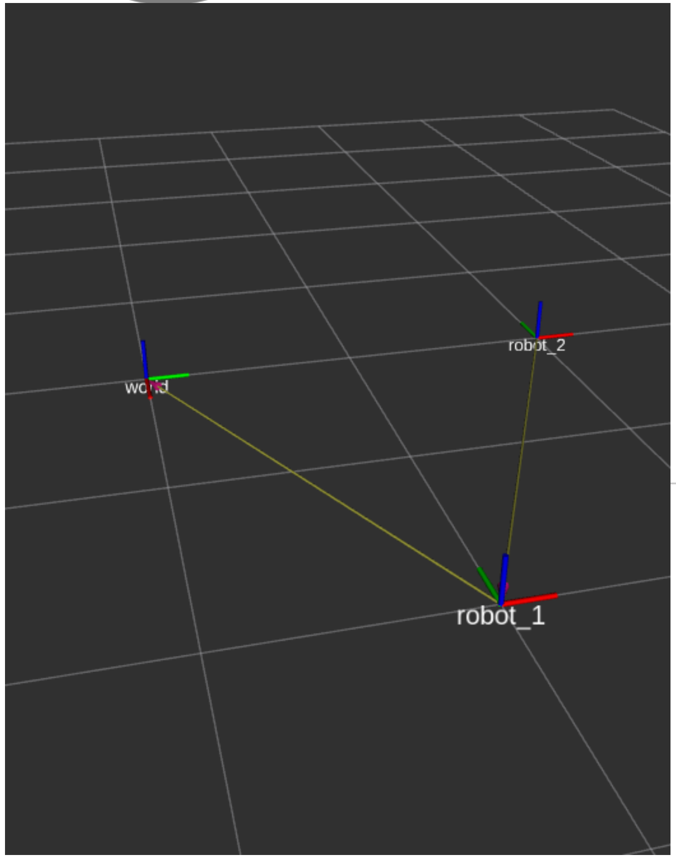


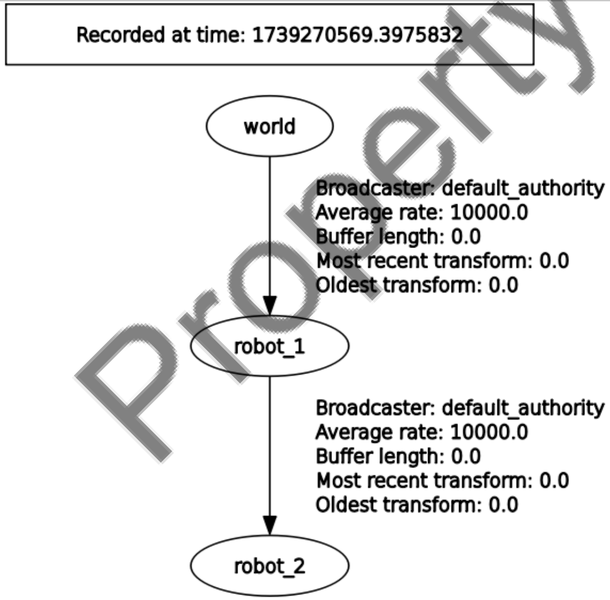
Static Transform Example



- On the Left pane, select the “TF” dropdown menu.
- Check the “Show Names” box.
  - Note that RViz represents the transform as an arrow from child to parent.

Static Transform Example





- Open another terminal and type the following

```
$ ros2 run tf2_ros static_transform_publisher 1 2 0 0 0 0
robot_1 robot_2
```

- A new coordinate system will appear on RVIZ
  - Be aware that the new coordinate system is with respect to “robot_1”
- To view the TF tree, type the following

```
$ ros2 run rqt_tf_tree rqt_tf_tree
```

- If not installed, you can install it as follows

```
# Replace <distro> with ROS2 distro, e.g., “humble”

$ sudo apt install ros-<distro>-rqt-tf-tree
```

Static Transform Example

```
mcr2@mcr2-virtual-machine:~$ ros2 run tf2_ros tf2_echo robot_2 world
[INFO] [1739271109.096574499] [tf2_echo]: Waiting for transform robot_2 ->
world: Invalid frame ID "robot_2" passed to canTransform argument target_fra
me - frame does not exist
[INFO] [1739271111.084715220] [tf2_echo]: Waiting for transform robot_2 ->
world: Invalid frame ID "robot_2" passed to canTransform argument target_fra
me - frame does not exist
At time 0.0
- Translation: [-2.002, -0.001, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.707, 0.707]
- Rotation: in RPY (radian) [0.000, 0.000, -1.570]
- Rotation: in RPY (degree) [0.000, 0.000, -89.954]
- Matrix:
  0.001  1.000  0.000 -2.002
 -1.000  0.001 -0.000 -0.001
 -0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
```

Listening to a Transform from command line

- The command "echo" as with the topics, ca be used to listen to a transform.

```
ros2 run tf2_ros tf2_echo [source_frame] [target_frame]
```

- On a new Terminal type

```
$ ros2 run tf2_ros tf2_echo robot_2 world
```

  - The "echo" command can take some time to "listen" to the TF.
  - This will provide the user with the transformation coordinates and TF Matrices, w.r.t. "robot_2"
  - For purely static transforms the time is always 0.0

Activity 1.2

Static Transforms from Launch Files

`{Learn, Create, Innovate};`

Manchester Robotics

Static Transform Example

Declaring a Static Transform from Launch File

- Make a package called "tf_examples" and a node called "static_tf.py" with the following dependencies:
  - geometry_msgs, python3-numpy, rclpy, tf2_ros_py, ros2launch, std_msgs

```
$ ros2 pkg create --build-type ament_python tf_examples --license Apache-2.0 --node-name static_tf --dependencies geometry_msgs python3-numpy rclpy tf2_ros_py ros2launch std_msgs --description TF2_Examples --maintainer-name "Mario Martinez" --maintainer-email mario.mtz@manchester-robotics.com
```

```
tf_examples/
├── launch
│   └── static_tf_launch.py
├── LICENSE
├── package.xml
├── resource
│   └── tf_examples
├── setup.cfg
├── setup.py
├── test
│   ├── test_copyright.py
│   ├── test_flake8.py
│   └── test_pep257.py
└── tf_examples
    ├── __init__.py
    └── static_tf.py
```

```
<depend>geometry_msgs</depend>
<depend>python3-numpy</depend>
<depend>rclpy</depend>
<depend>tf2_ros_py</depend>
<depend>ros2launch</depend>
<depend>std_msgs</depend>
```

- Do not forget to give executable permissions to the newly created files

```
$ chmod +x tf_examples/tf_examples/*
```

Static Transform Example

- Make a launch folder and a launch file called "static_tf_launch.py"

```
#replace "YOUR_WS" with the name of tour workspace

$ cd ~/<YOUR_WS>/src/tf_examples
$ mkdir launch
$ touch static_tf_launch.py
$ chmod +x tf_examples/launch/*
```

- Change the "setup.py" to find the launch files

```python
from setuptools import find_packages, setup
import os
from glob import glob

package_name = 'tf_examples'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*'))),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='Mario Martinez',
    maintainer_email='mario.mtz@manchester-robotics.com',
    description='TF2_Examples',
    license='Apache-2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'static_tf = tf_examples.static_tf:main'
        ],
    },
)
```

```python
from setuptools import find_packages, setup
import os
from glob import glob
...

 data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.join('launch', '*launch.[pxy][yma]*'))),
    ],
```

Static Transform Example

Declaring a Static Transform from Launch File

- A static transformation can be defined in a Launch file with the following architecture.

```
static_transform_node = Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        arguments = ['--x', '2', '--y', '1', '--z', '0.0',
           '--yaw', '1.57', '--pitch', '0', '--roll', '0.0',
           '--frame-id', 'world', '--child-frame-id', 'robot_1']
           )
```

- Open the Launch File
- Modify the Launch File as shown

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():

    static_transform_node = Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            arguments = ['--x', '2', '--y', '1', '--z', '0.0',
                '--yaw', '1.57', '--pitch', '0', '--roll', '0.0',
                '--frame-id', 'world', '--child-frame-id', 'robot_1']
            )

    static_transform_node_2 = Node(
             package='tf2_ros',
             executable='static_transform_publisher',
             arguments = ['--x', '1', '--y', '2', '--z', '0.0',
               '--yaw', '0.0', '--pitch', '0', '--roll', '0.0',
               '--frame-id', 'robot_1', '--child-frame-id', 'robot_2']
             )
```

Static Transform Example

```
    rviz_node = Node(name='rviz',
                    package='rviz2',
                    executable='rviz2'
                    )

    rqt_tf_tree_node = Node(name='rqt_tf_tree',
                    package='rqt_tf_tree',
                    executable='rqt_tf_tree'
                    )

    l_d = LaunchDescription([static_transform_node,
static_transform_node_2 , rqt_tf_tree_node, rviz_node])

    return l_d
```

Declaring a Static Transform from Launch File

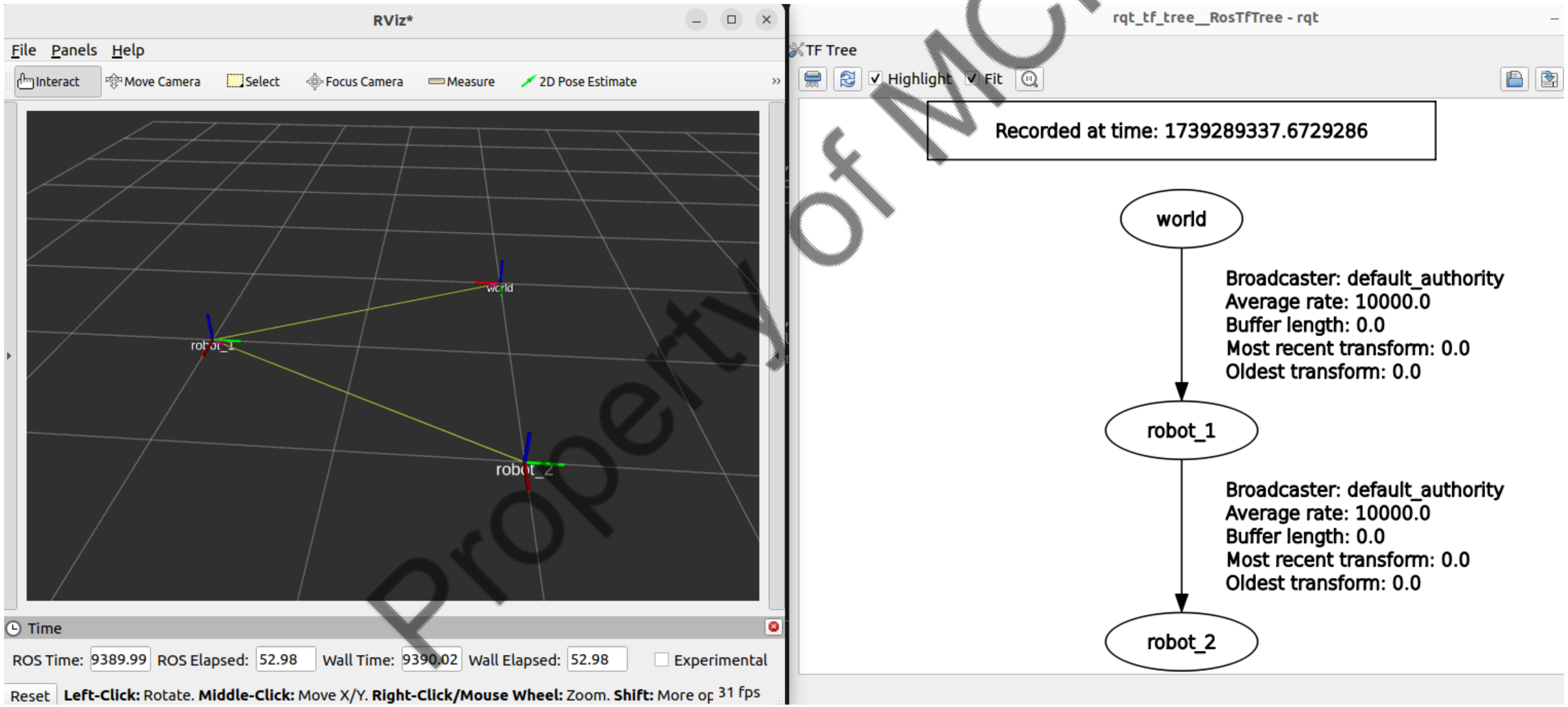
- Save the file
- Build the package

```
$ cd ~/<YOUR WORKSPACE>/
$ colcon build
$ source install/setup.bash
```

- Launch the file

```
$ ros2 launch tf_examples static_tf_launch.py
```

Static Transform Example



{Learn, Create, Innovate};

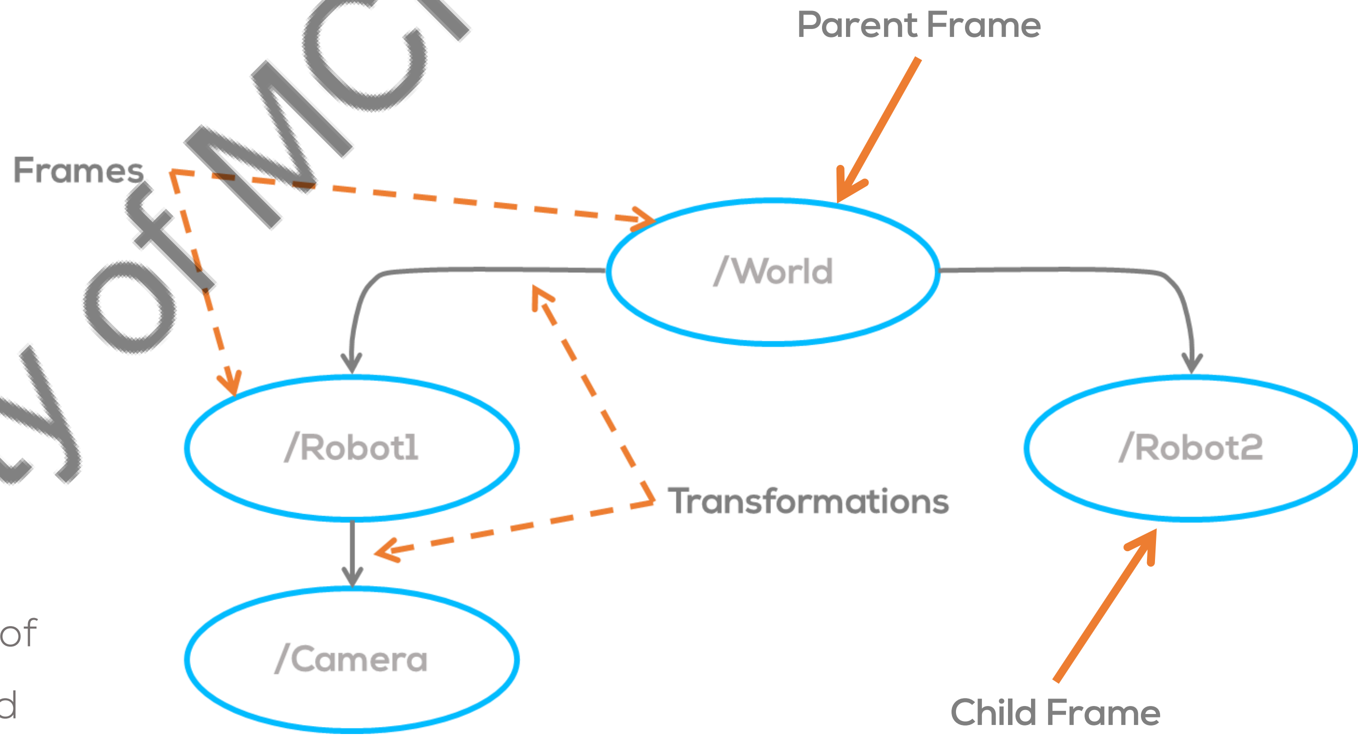
ROS

Declaring a Transform

Manchester Robotics

Declaring a transform

- Inside a script, transforms are declared using a *Transform Stamped Message*.
- The message is composed of a Header and the pose information of the frame, with respect to the parent frame and the name of the child frame.
- The Header contains the information about the message's time stamp and the parent frame.
- The rest of the message contains the information of the child frame ID, where the data is published and the pose transformation between the two frames.



Declaring a transform

Transform Stamped Message

- The Transform Stamped Message, is under ROS geometry_msgs.
- The pose is divided into translation and rotation.
- The translation is in meters for each coordinate.
- The rotation is a quaternion.
- More information here.
- To transform from Euler angles to quaternions and viceversa in ROS the python library “transforms3d” can be used.

Transform Stamped Message

Parent frame

```
ex_tf = TransformStamped()
ex_tf.header.frame_id = "inertial_frame"
ex_tf.header.stamp = get_clock().now().to_msg()
```

Header

```
ex_tf.child_frame_id = "ex_child_frame"
```

Child frame

```
ex_tf.transform.translation.x = 1
ex_tf.transform.translation.y = 1
ex_tf.transform.translation.z = 1.0
```

Translation

```
ex_tf.transform.rotation.x = 0
ex_tf.transform.rotation.y = 0
ex_tf.transform.rotation.z = 0
ex_tf.transform.rotation.w = 1
```

Rotation
(quaternion)

Attaching markers to frames

Headers

- Standard metadata for higher level stamped data types.
- Used to communicate timestamped data in a particular coordinate frame.
- In other words, the header, contains information about the data that is being published (metadata).

```
uint32 seq
time stamp
string frame_id
```

- The frame (frame_id) states the coordinate frame the data is associated with, for the transformation message is the "parent_frame".
- The sequence ID (seq) is a consecutively increasing ID, set automatically by ROS.
- The timestamp (stamp) is a ROS Time Message to keep time every time a message is published.
- ROS then uses the transform library (tf2) and the header file information of the marker message, to transform the coordinates of the marker into the one defined by the "frame_id".
- This transformation is done automatically once the marker message is published and if the coordinate frame is available.

Broadcasting / Listening a transform

- The "broadcaster" is closely related to the ROS "publisher".
- Allows to "broadcast" or publish a ROS transform.
- There are two types of broadcaster, depending on the type of transformation.
  - Static Broadcaster
  - Dynamic Broadcaster

```
staticbc_ex = StaticTransformBroadcaster()
dynamicbc_ex = TransformBroadcaster()
```

- The idea of the "listener" is closely related to the ROS "subscriber". More information here.
- Allows to "listen" or publish a ROS transform from one frame to another frame.
- The listener can be declared and used in any node, even if is unrelated to a frame, so long as there is a transformation relationship between the requested frames.
- To create a listener, a buffer is required to listen to the transformations and "buffer" them or get the latest transform.

```
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)
trans = tfBuffer.lookup_transform(frame1, frame2,
rclpy.time.Time())
```

Activity 2

Declaring Transforms

{Learn, Create, Innovate};

MCR2

Manchester Robotics

Declaring Transforms

- In this activity two Static transforms will be generated in a script.
- Install the python "transforms3d" package to transform rotations from Euler to quaternion.

```
$ pip install transforms3d
```

- In the package *"tf_examples"* open the node called *"static_tf.py"*
- Create a *"FramePublisher"* class to be executed.

```python
import rclpy
from rclpy.node import Node
from tf2_ros import StaticTransformBroadcaster
from geometry_msgs.msg import TransformStamped
import transforms3d

class FramePublisher(Node):

... # TO BE DONE IN NEXT STEP


def main(args=None):
    rclpy.init(args=args)

    node = FramePublisher()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        node.destroy_node()


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

Declaring Transforms

- Inside the *"FramePublihser"* class declare two static broadcasters and two transform stamped messages to be sent.

```
class FramePublisher(Node):
    def __init__(self):
        super().__init__('frame_publisher')
        self.static_br1 = StaticTransformBroadcaster(self)
        self.static_br2 = StaticTransformBroadcaster(self)

        t = TransformStamped()
        t.header.stamp = self.get_clock().now().to_msg()
        t.header.frame_id = 'world'
        t.child_frame_id = 'robot_3'
        t.transform.translation.x = -2.0
        t.transform.translation.y = -1.0
        t.transform.translation.z = 0.0
        q = transforms3d.euler.euler2quat(0, 0, 0)
        t.transform.rotation.x = q[1]
        t.transform.rotation.y = q[2]
        t.transform.rotation.z = q[3]
        t.transform.rotation.w = q[0]
```

- The function "euler2quat" is being used to transform from Euler Angles to Quaternions.
  - The input are the euler angles (roll, pitch, yaw)
  - The output of this function provides the quaternions as vectors q=[w,x,y,z]!
  - The function provides an extrinsic rotation based on the header frame.

```
        t2 = TransformStamped()
        t2.header.stamp = self.get_clock().now().to_msg()
        t2.header.frame_id = 'robot_2'
        t2.child_frame_id = 'robot_4'
        t2.transform.translation.x = 1.0
        t2.transform.translation.y = 1.0
        t2.transform.translation.z = 1.0
        q2 = transforms3d.euler.euler2quat(1.57, 1.57, 0) #output q=[w,
x, y, z]
        t2.transform.rotation.x = q2[1]
        t2.transform.rotation.y = q2[2]
        t2.transform.rotation.z = q2[3]
        t2.transform.rotation.w = q2[0]

        self.static_br1.sendTransform(t)
        self.static_br2.sendTransform(t2)
```

Declaring Transforms

- Open the file *package.xml*
- Add the dependency "python3-transforms3d"

```
<depend>geometry_msgs</depend>
<depend>python3-numpy</depend>
<depend>rclpy</depend>
<depend>tf2_ros_py</depend>
<depend>ros2launch</depend>
<depend>std_msgs</depend>
<depend>python3-transforms3d</depend>
```

- Add your newly created node to your launch file

```
    static_transform_node_3 = Node(
                                name="static_tf",
                                package='tf_examples',
                                executable='static_tf'
                                )


    # Add it to your Launch Description

l_d = LaunchDescription([static_transform_node,
static_transform_node_2, rqt_tf_tree_node, rviz_node,
static_transform_node_3])
```

- Compile the program

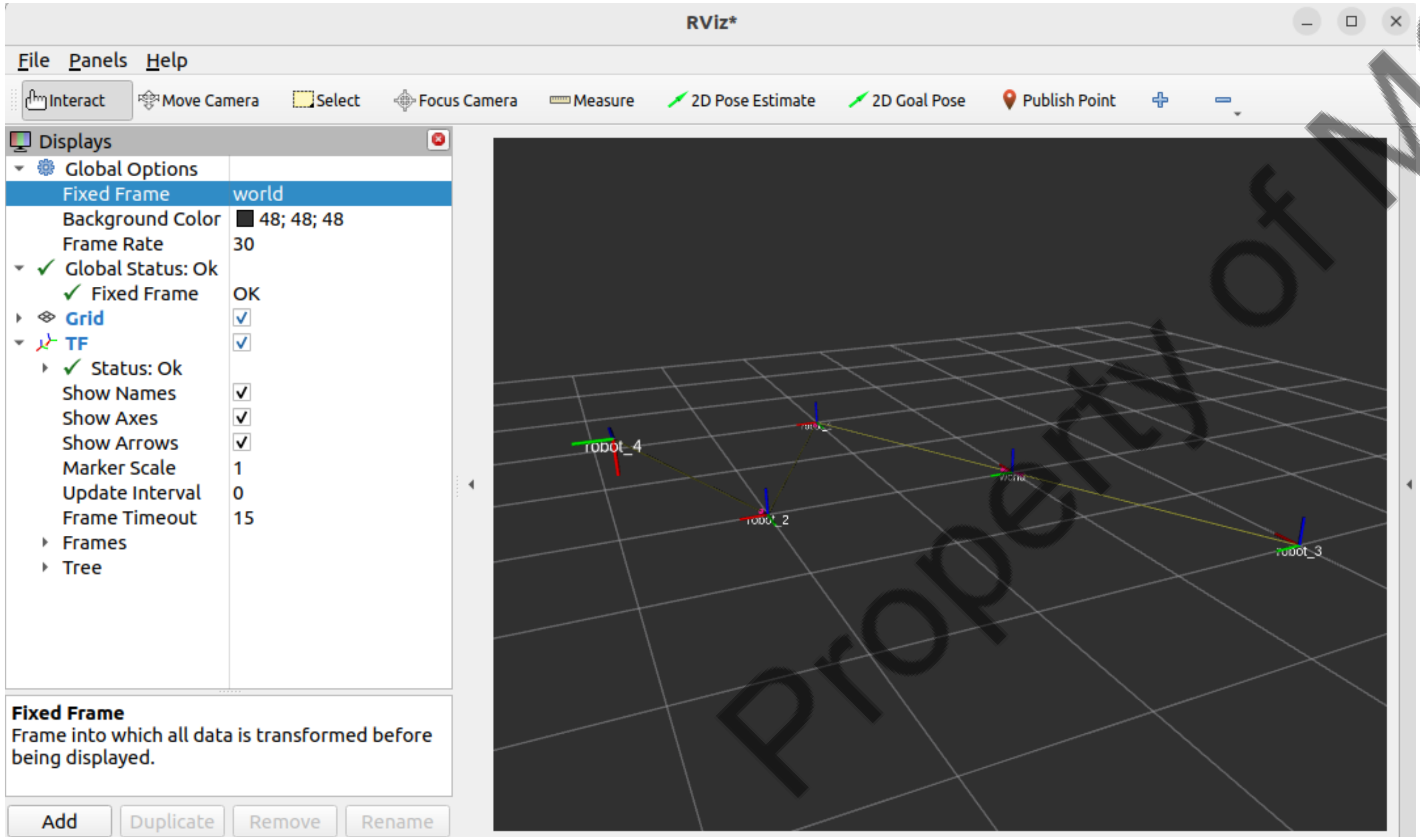
```
$ cd ~/<YOUR_WS>
$ colcon build
$ source install/setup.bash
```

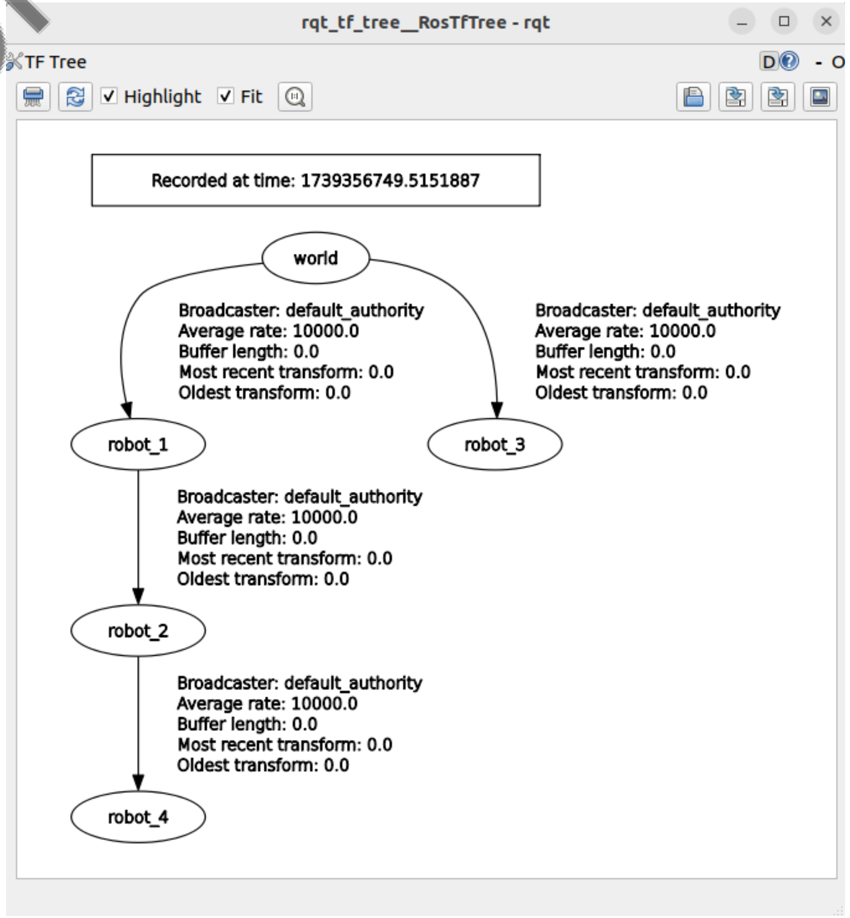
- Launch the program

```
$ ros2 launch tf_examples static_tf_launch.py
```

- Add the "TF" in RVIZ and set "world" as the fixed frame.
- Show the names of the "TF"

Results


RViz*
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Goal Pose
Publish Point
Displays
Global Options
Fixed Frame world
Background Color 48; 48; 48
Frame Rate 30
Global Status: Ok
Fixed Frame OK
Grid
TF
Status: Ok
Show Names
Show Axes
Show Arrows
Marker Scale 1
Update Interval 0
Frame Timeout 15
Frames
Tree
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename
robot_4
robot_2
robot_3


rqt_tf_tree__RosTfTree - rqt
TF Tree
Highlight
Fit
Recorded at time: 1739356749.5151887
world
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_1
robot_3
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_2
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_4

Activity 2.1

Declaring Dynamic Transforms

{Learn, Create, Innovate};

Manchester Robotics

Dynamical Transforms

- In the package *“tf_examples”* create a new node called *“dyn_tf.py”*

```
cd ~/<YOUR_WS>/src/tf_examples/tf_examples/
```

```
touch dyn_tf.py
```

- Give executable permission to the file

```
sudo chmod +x dyn_tf.py
```

- Modify the “setup.py” file entry points to include the newly created node

```
entry_points={
        'console_scripts': [
            'static_tf = tf_examples.static_tf:main',
            'dyn_tf = tf_examples.dyn_tf:main'
        ],
    },
```

- Follow the previous two activities to add a planet marker to the “sun” frame, make a moon rotate around the planet and an arrow pointing to the moon using transforms.

Declaring Transforms

- In this activity, two Dynamic transforms will be generated in a script.
- Make sure the python "transforms3d" package is installed as per the previous instructions.
- In the package *"tf_examples"* open the node called *"dyn_tf.py"*
- Create a *"FramePublisher"* class to be executed.

```python
import rclpy
from rclpy.node import Node
from tf2_ros import StaticTransformBroadcaster
from geometry_msgs.msg import TransformStamped
import transforms3d
import numpy as np

class FramePublisher(Node):

... # TO BE DONE IN NEXT STEP


def main(args=None):
    rclpy.init(args=args)

    node = FramePublisher()

    try:
        rclpy.spin(node)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():  # Ensure shutdown is only called once
            rclpy.shutdown()
        node.destroy_node()


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

Declaring Transforms

- Inside the *"FramePublihser"* class declare two broadcasters and two transform stamped messages to be sent.
- Add a timer to update the transforms
- Inside the timer update the transforms using a time changing variable "elapsed time"

```
class FramePublisher(Node):
    def __init__(self):
        super().__init__('frame_publisher')
        #Create Trasnform Messages
        self.t = TransformStamped()
        self.t2 = TransformStamped()
        #Create Transform Boradcasters
        self.tf_br1 = TransformBroadcaster(self)
        self.tf_br2 = TransformBroadcaster(self)
        #Create a Timer
        timer_period = 0.1 #seconds
        self.timer = self.create_timer(timer_period, self.timer_cb)
        #Variables to be used
        self.start_time = self.get_clock().now()
        self.omega = 0.1
```

```
#Timer Callback
def timer_cb(self):
    elapsed_time = (self.get_clock().now() - self.start_time).nanoseconds/1e9

    self.t.header.stamp = self.get_clock().now().to_msg()
    self.t.header.frame_id = 'world'
    self.t.child_frame_id = 'moving_robot_3'
    self.t.transform.translation.x = 0.5*np.sin(self.omega*elapsed_time)
    self.t.transform.translation.y = 0.5*np.cos(self.omega*elapsed_time)
    self.t.transform.translation.z = 0.0
    q = transforms3d.euler.euler2quat(0, 0, -self.omega*elapsed_time)
    self.t.transform.rotation.x = q[1]
    self.t.transform.rotation.y = q[2]
    self.t.transform.rotation.z = q[3]
    self.t.transform.rotation.w = q[0]

    self.t2.header.stamp = self.get_clock().now().to_msg()
    self.t2.header.frame_id = 'world'
    self.t2.child_frame_id = 'moving_robot_4'
    self.t2.transform.translation.x = 1.0
    self.t2.transform.translation.y = 1.0
    self.t2.transform.translation.z = 1.0
    q2 = transforms3d.euler.euler2quat(elapsed_time, elapsed_time, 0)
    self.t2.transform.rotation.x = q2[1]
    self.t2.transform.rotation.y = q2[2]
    self.t2.transform.rotation.z = q2[3]
    self.t2.transform.rotation.w = q2[0]

    self.tf_br1.sendTransform(self.t)
    self.tf_br2.sendTransform(self.t2)
```

Declaring Transforms

- Open the file *package.xml*
- Add the dependency “python3-transforms3d”, if not added

```
<depend>geometry_msgs</depend>
<depend>python3-numpy</depend>
<depend>rclpy</depend>
<depend>tf2_ros_py</depend>
<depend>ros2launch</depend>
<depend>std_msgs</depend>
<depend>python3-transforms3d</depend>
```

- Add your newly created node to your launch file

```
dynamic_transform_node = Node(
                              name="dyn_tf",
                              package='tf_examples',
                              executable='dyn_tf'
                              )

    # Add it to your Launch Description

l_d = LaunchDescription([static_transform_node,
static_transform_node_2, rqt_tf_tree_node, rviz_node,
static_transform_node_3, dynamic_transform_node])
```

- Compile the program

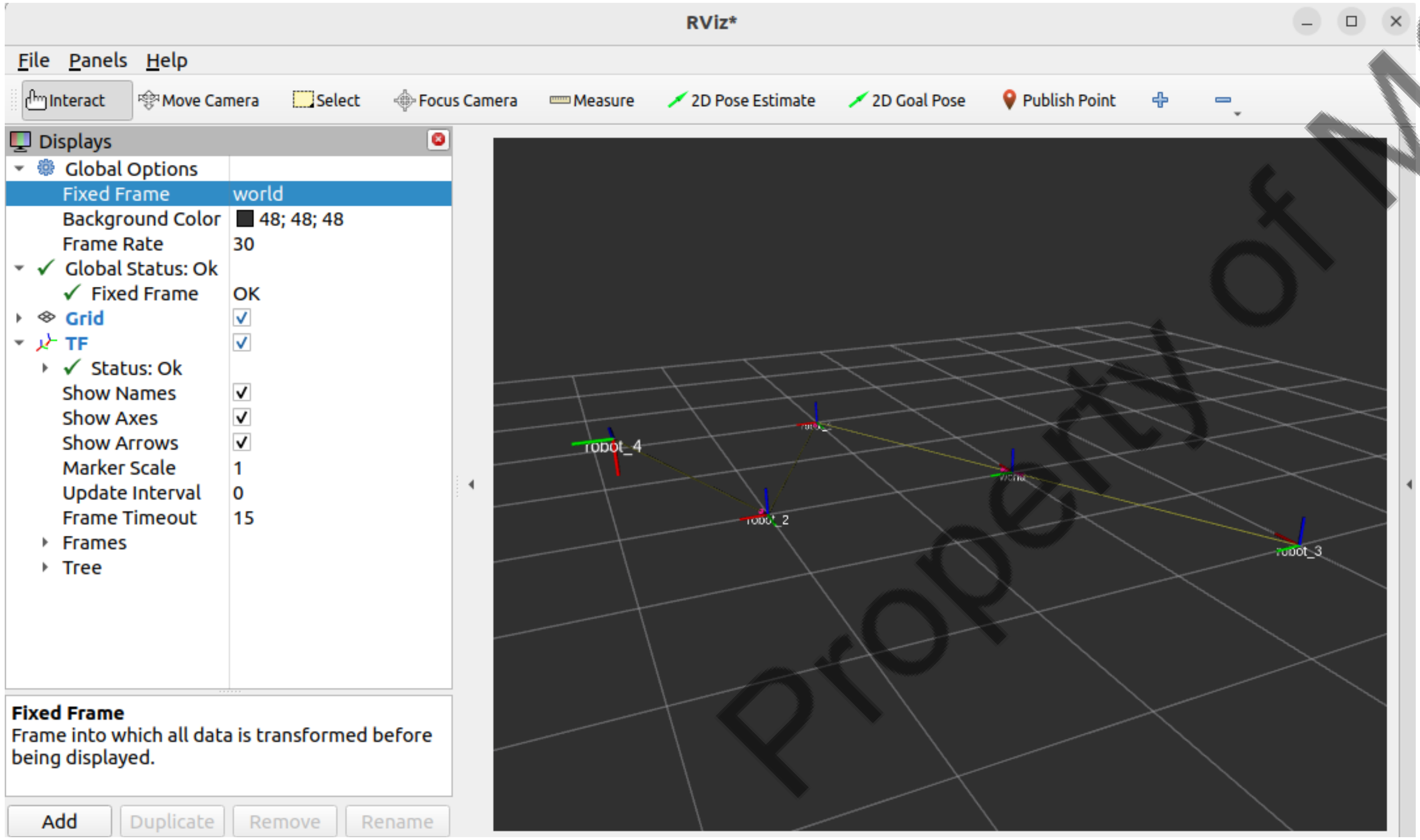
```
$ cd ~/<YOUR_WS>
$ colcon build
$ source install/setup.bash
```

- Launch the program

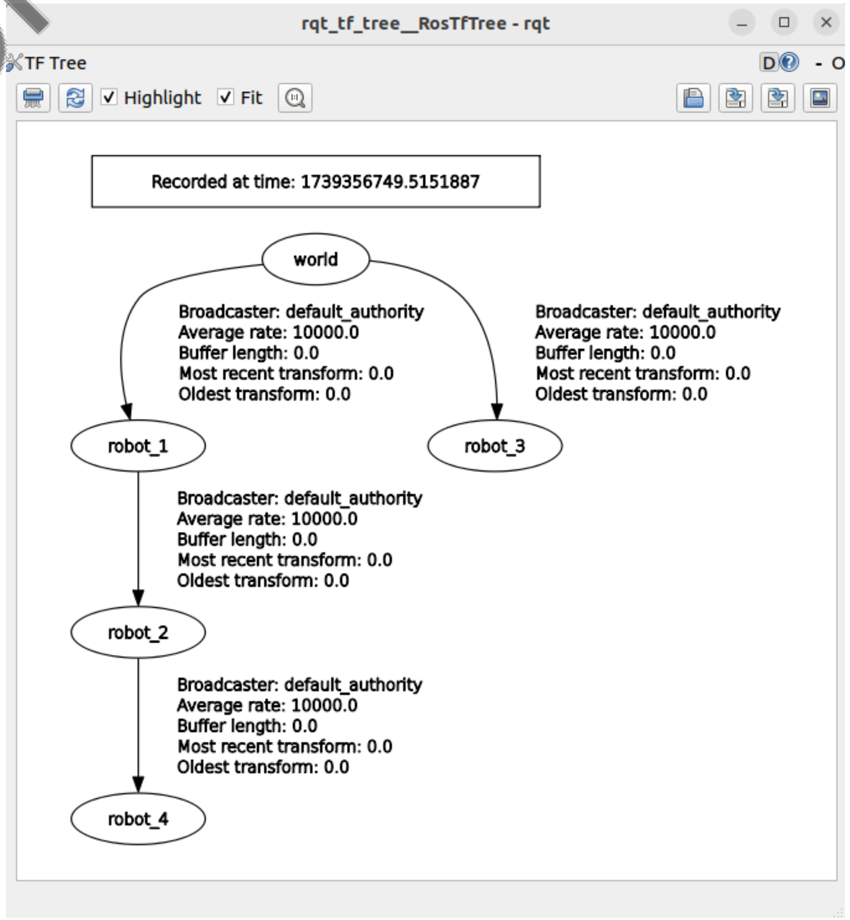
```
$ ros2 launch tf_examples static_tf_launch.py
```

- Add the “TF” in RVIZ and set “world” as the fixed frame.
- Show the names of the “TF”

Results

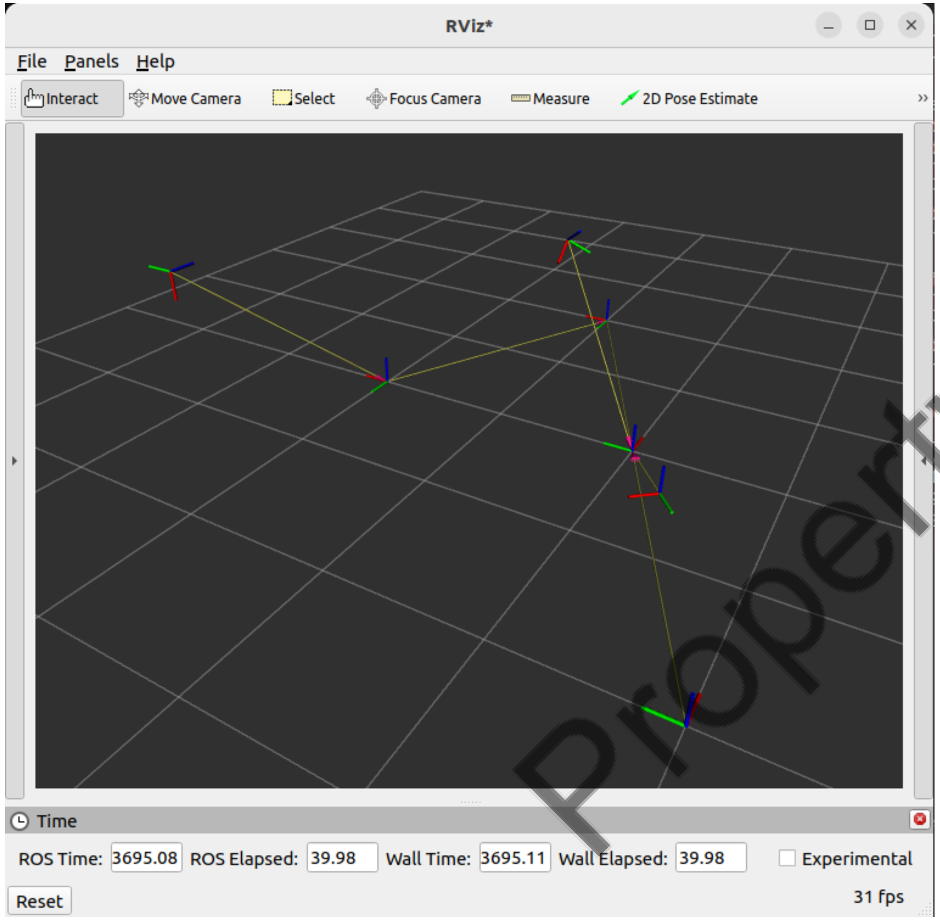


RViz*
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Goal Pose
Publish Point
Displays
Global Options
Fixed Frame
world
Background Color
48; 48; 48
Frame Rate
30
Global Status: Ok
Fixed Frame
OK
Grid
TF
Status: Ok
Show Names
Show Axes
Show Arrows
Marker Scale
1
Update Interval
0
Frame Timeout
15
Frames
Tree
Fixed Frame
Frame into which all data is transformed before being displayed.
Add
Duplicate
Remove
Rename
robot_4
robot_2
robot_3

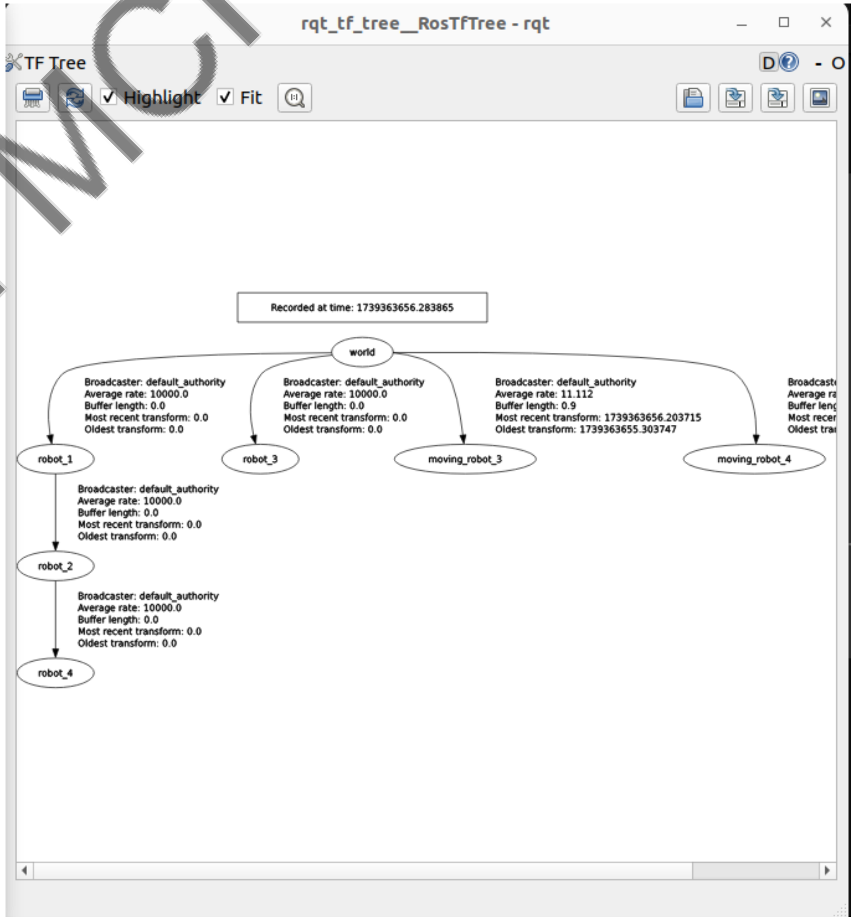


rqt_tf_tree__RosTfTree - rqt
TF Tree
Highlight
Fit
Recorded at time: 1739356749.5151887
world
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_1
robot_3
robot_2
robot_4

Result



RViz*
File
Panels
Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
Time
ROS Time: 3695.08
ROS Elapsed: 39.98
Wall Time: 3695.11
Wall Elapsed: 39.98
Experimental
Reset
31 fps

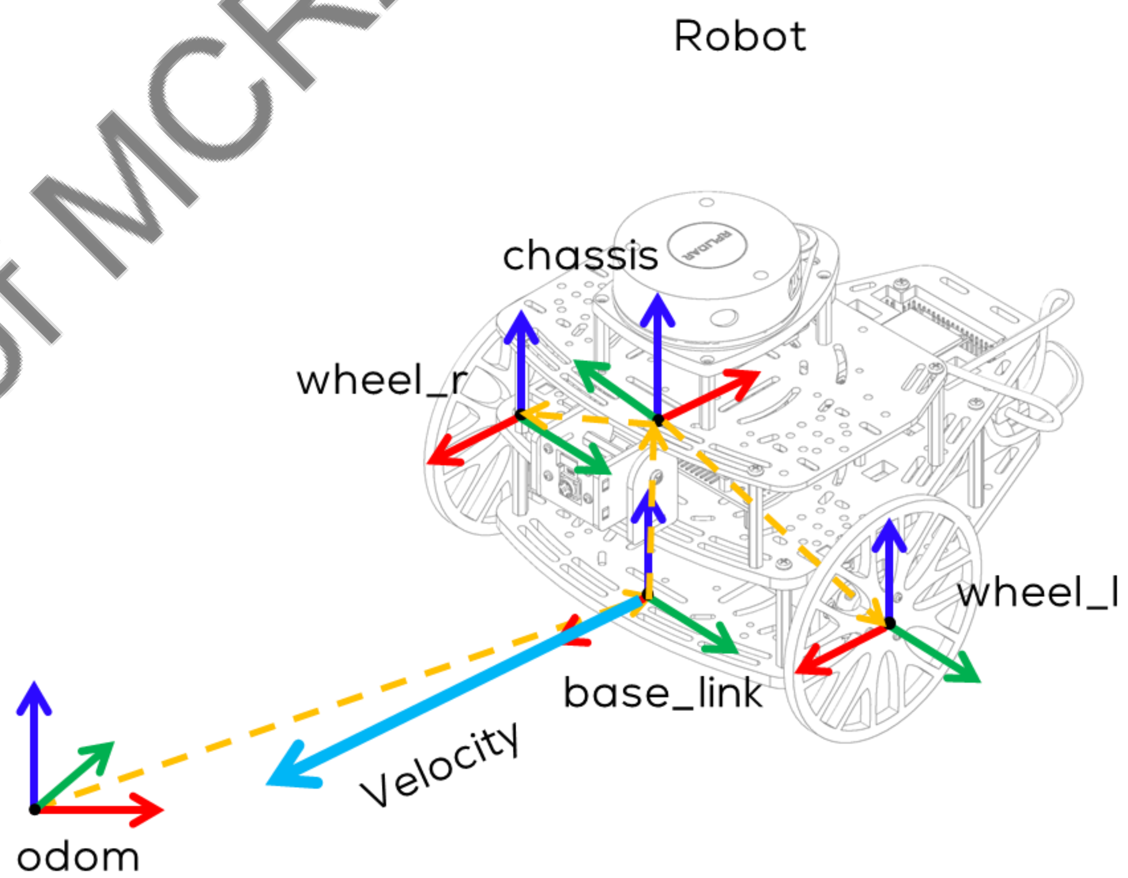


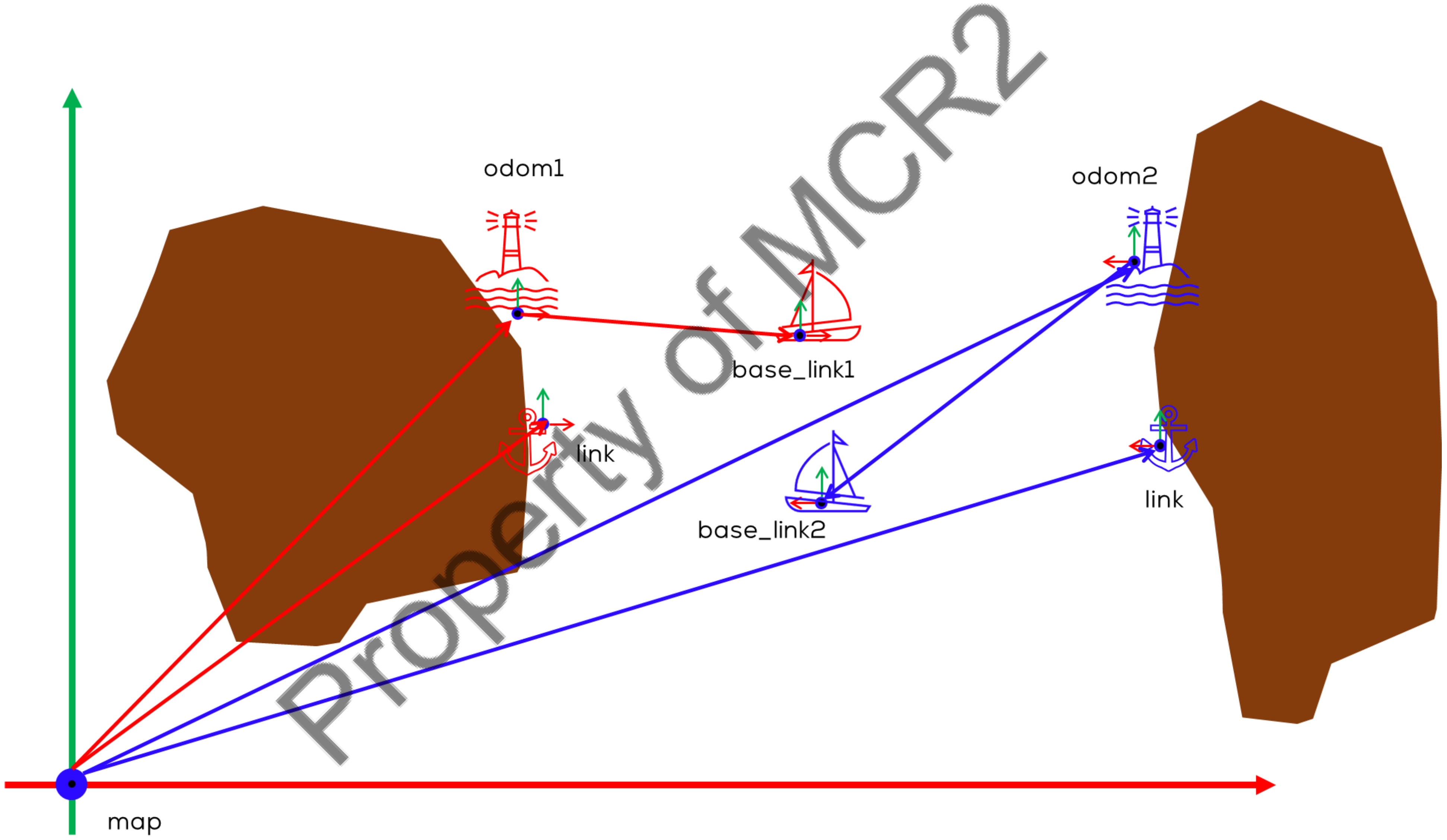
rqt_tf_tree__RosTfTree - rqt
TF Tree
Highlight
Fit
Recorded at time: 1739363656.283865
world
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
Broadcaster: default_authority
Average rate: 11.112
Buffer length: 0.9
Most recent transform: 1739363656.203715
Oldest transform: 1739363655.303747
robot_1
robot_3
moving_robot_3
moving_robot_4
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_2
Broadcaster: default_authority
Average rate: 10000.0
Buffer length: 0.0
Most recent transform: 0.0
Oldest transform: 0.0
robot_4

Coordinate frames mobile platforms

Motivation

- ROS2 follows a standard convention for defining coordinate frames, ensuring compatibility across sensors, localization, mapping, and control.
- Driver, model, and library developers need this convention for coordinate frames to improve integration and reuse of software components.
- Shared conventions for coordinate frames provide a specification for developers creating drivers and models for mobile bases.
- Similarly, developers creating libraries and applications can more easily use their software with various mobile bases compatible with this specification.





odom1
odom2
base_link1
link
base_link2
link
map

Coordinate frames mobile platforms

Coordinate Frames

In ROS2, mobile platforms follow a right-handed coordinate system that aligns with the REP-103 (Standard Units of Measure and Coordinate Conventions) and REP-105 (Coordinate Frames for Mobile Platforms) standards.

World/Earth

- Frame designed to allow the interaction of multiple robots in different map frames.
- If the application only needs one map the earth coordinate frame is not expected to be present.

Map

- World Fixed Coordinate frame.
- Z-axis pointing upwards.
- The pose of a mobile platform, relative to the map frame, should not significantly drift over time.
- Not continuous frame (pose of a mobile platform can change in discrete jumps)
  - In a typical setup, a localization component constantly re-computes the robot pose in the map frame based on sensor observations, therefore eliminating drift, but causing discrete jumps when new sensor information arrives.

Coordinate frames mobile platforms

odom

- The coordinate frame called odom is a world-fixed frame.
- The pose of a mobile platform in the odom frame can drift over time, without any bounds.
- Odom frame useless as a long-term global reference.
- However, the pose of a robot in the odom frame is guaranteed to be continuous, without discrete jumps.
- In a typical setup the odom frame is computed based on an odometry source, such as wheel odometry, visual odometry, etc.

base_link

- The coordinate frame called base_link is rigidly attached to the mobile robot base.
- For every hardware platform there will be a different place on the base that provides an obvious point of reference

Transformation Hierarchy

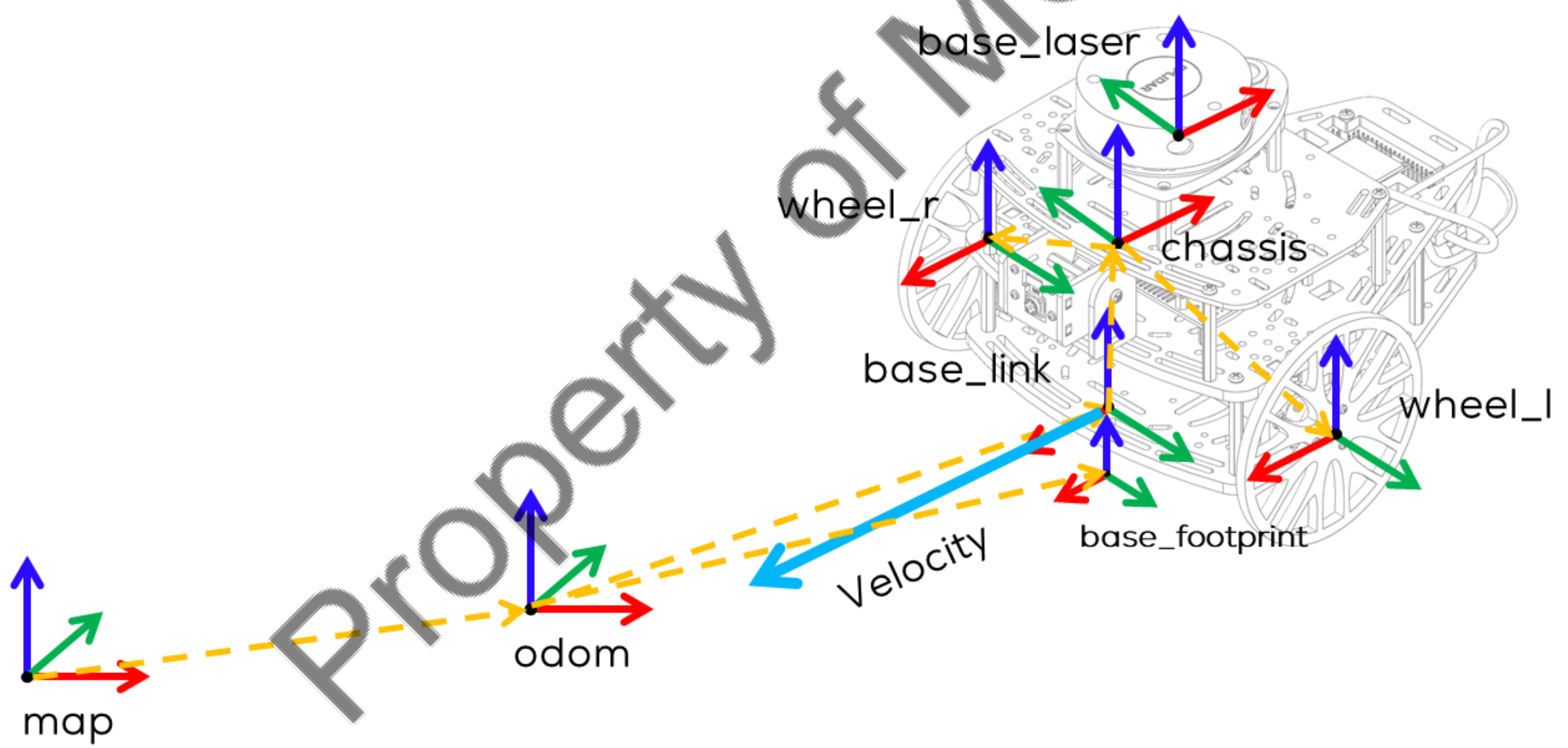
- world
  - map
    - odom
      - base_link
        - other…

Extra frames

- The basic topology should stay the same, however it is fine to insert additional links in the graph which may provide additional functionality.
- Examples:

  base_footprint: Same as base_link but aligned with the ground.
- base_laser: LiDAR sensor frame (mounted on the robot).
- camera_link: Camera frame (if available).

Coordinate frames mobile platforms



ROS Transformations

TF Listeners

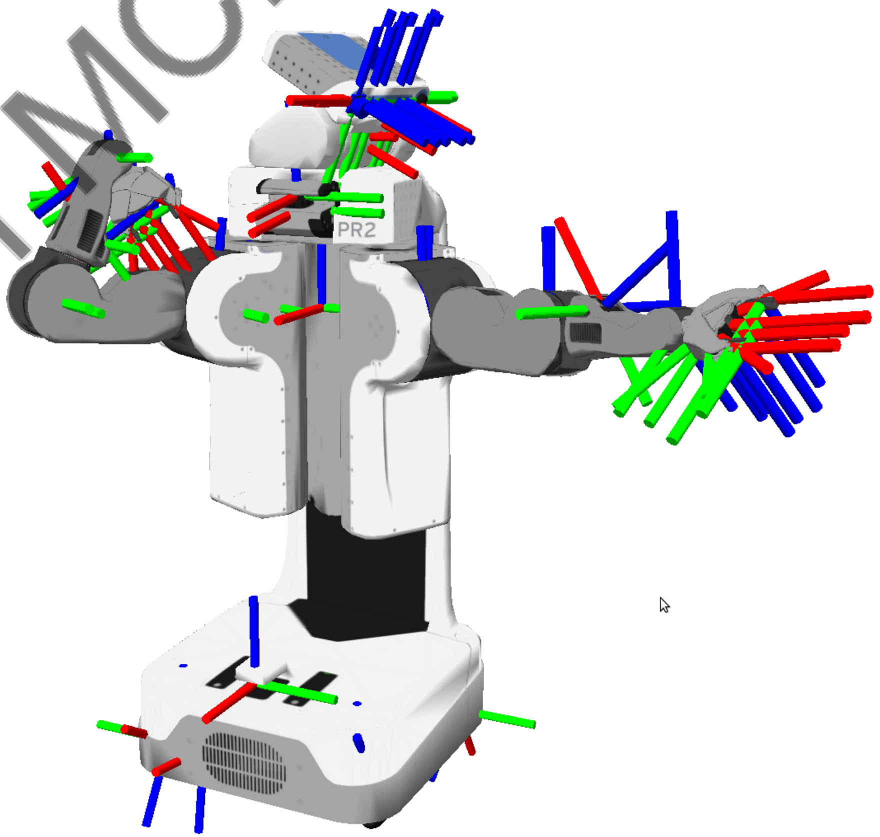
{Learn, Create, Innovate};

MCR²

Manchester **Robotics**

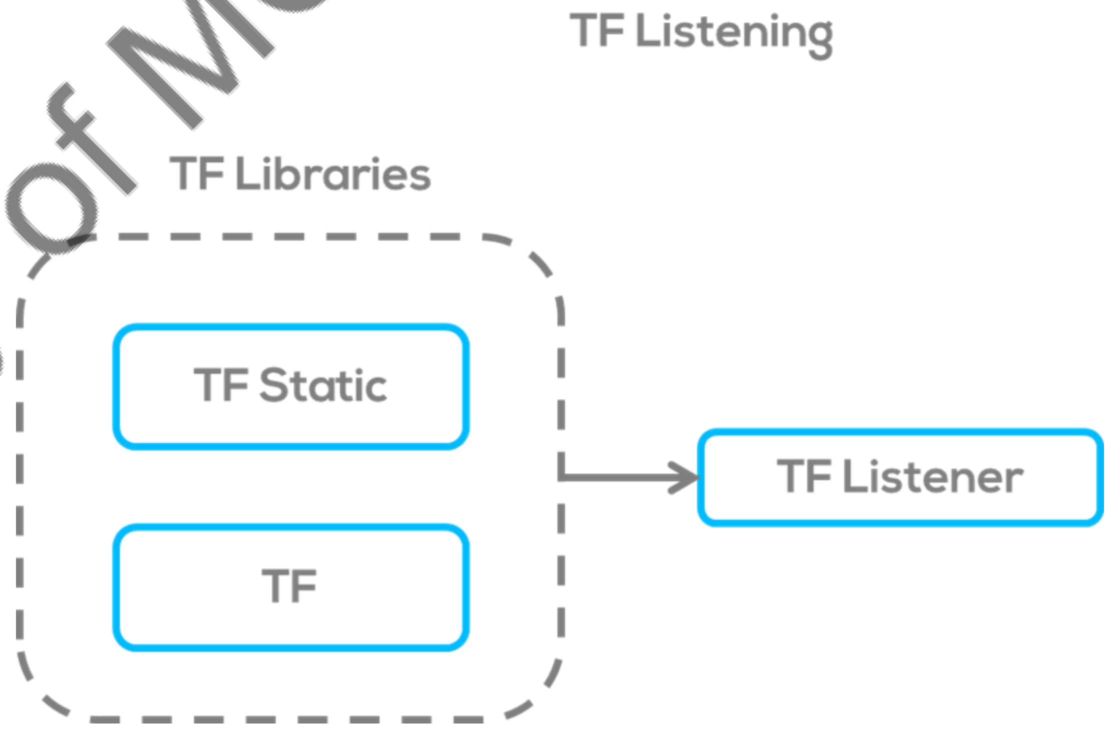
Transformation Listener in ROS

- A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc.
- When doing robotics, the user might have the following questions:
  - Where was the head frame relative to the world frame, 5 seconds ago?
  - What is the pose of the object in my gripper relative to my base?
  - What is the current pose of the base frame in the map frame?
- "Listening" to a transformations will solve such questions…



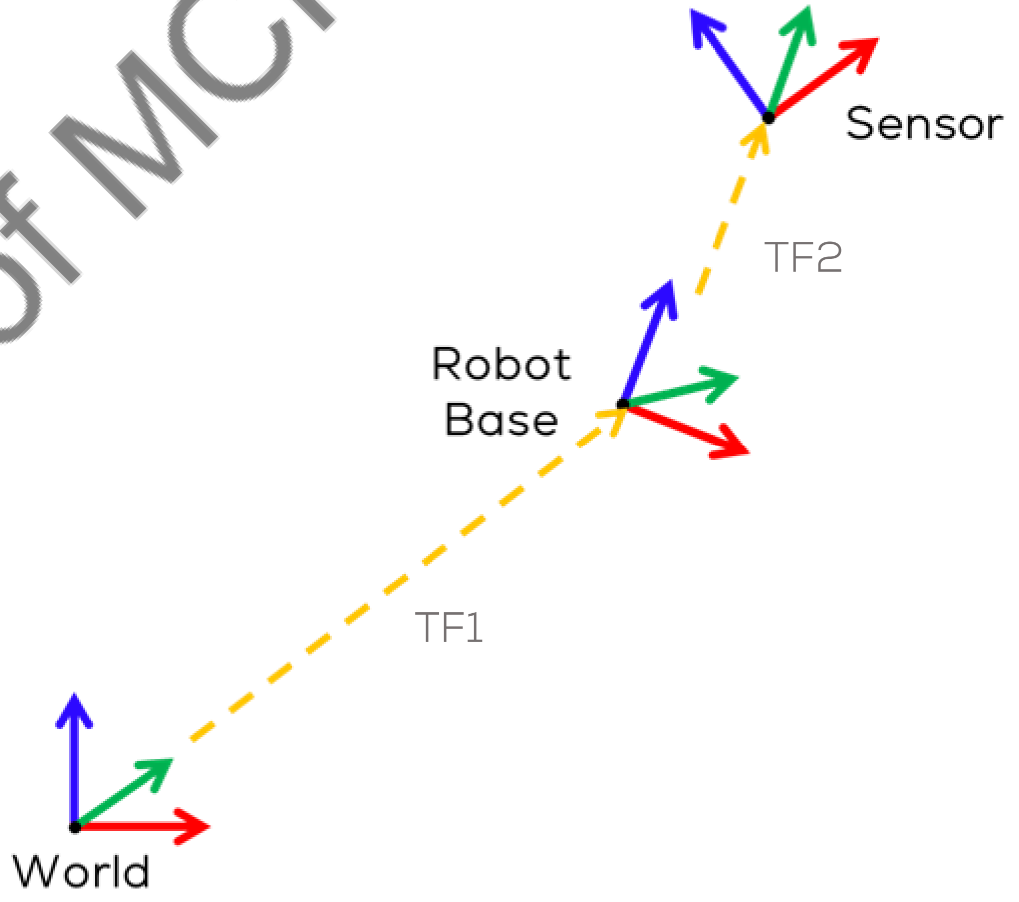
Transformation Listener in ROS

- As stated, transformation can be “listened” to; in other words, we can retrieve and manage transformation information between different coordinate frames in a robotic system.
- The Transformation Listener provides a way to keep track of the relationships between different coordinate frames as they change over time.



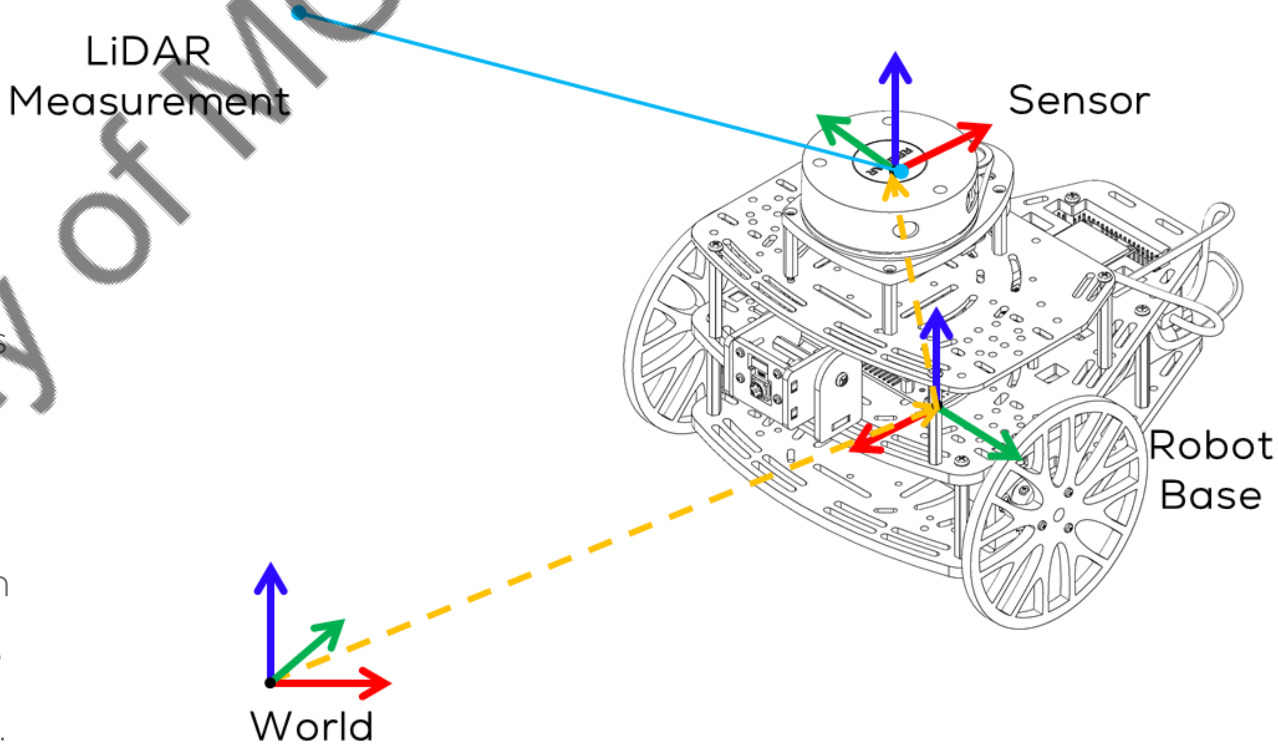
Transformation Listener in ROS

- Coordinate frames can represent the base of a robot, sensors like cameras and LIDAR, or objects in the robot's environment.
- The Transformation Listener allows you to query and receive the transformation information between these frames, essential for tasks like sensor fusion, motion planning, and robot control.



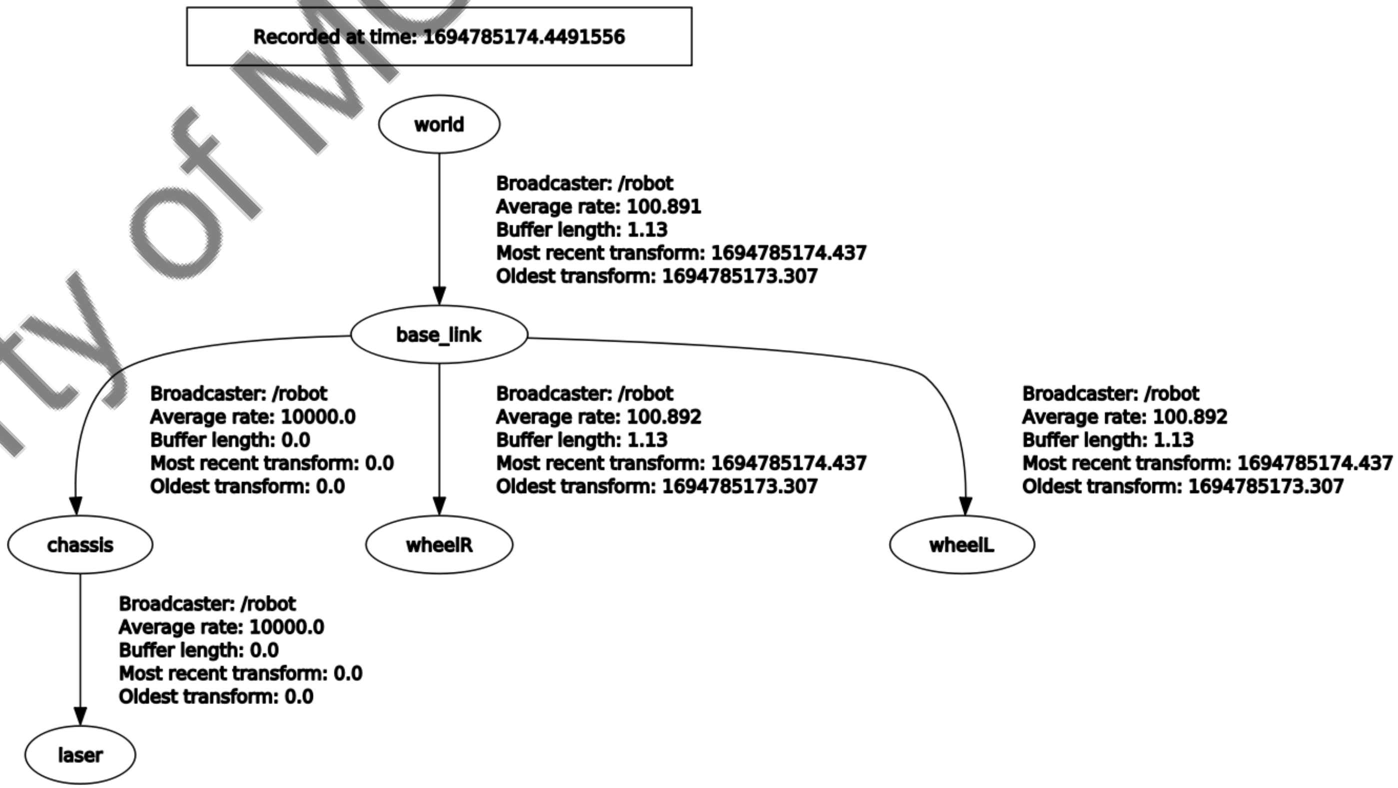
Transformation Listener ROS: Usage

- **Query Transformations:** You can use a Transformation Listener to query the transformation (translation and rotation) between two coordinate frames at a specific point in time. For example, you might want to know the position of a Lidar measurement, with respect to the robot's base frame.
- **Dynamic Updates:** The Transformation Listener is designed to handle dynamic transformations, which means it can provide you with the most up-to-date transformation information as it changes over time. This is crucial for real-time robotics applications.



Transformation Listener ROS: Usage

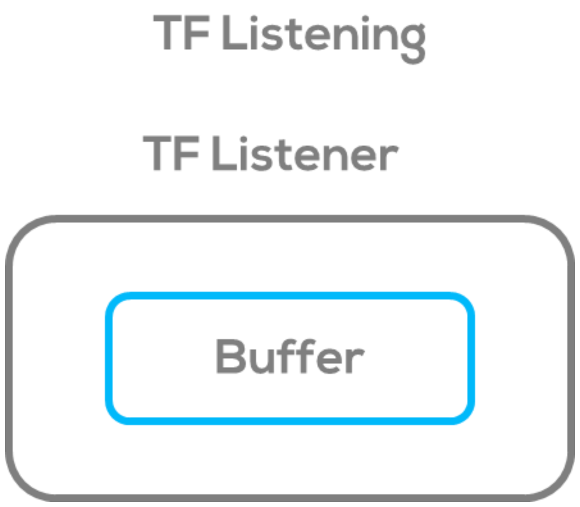
- **Buffering and Interpolation:** The TF package buffers and interpolates transformation data, ensuring smooth transitions between frames, even if the data arrives at irregular intervals.
- **Tree Structure:** TF organizes the coordinate frames into a tree structure, with each frame being a node in the tree. This tree structure represents the relationships between frames in the robot's ecosystem.



Coordinate transformations in ROS

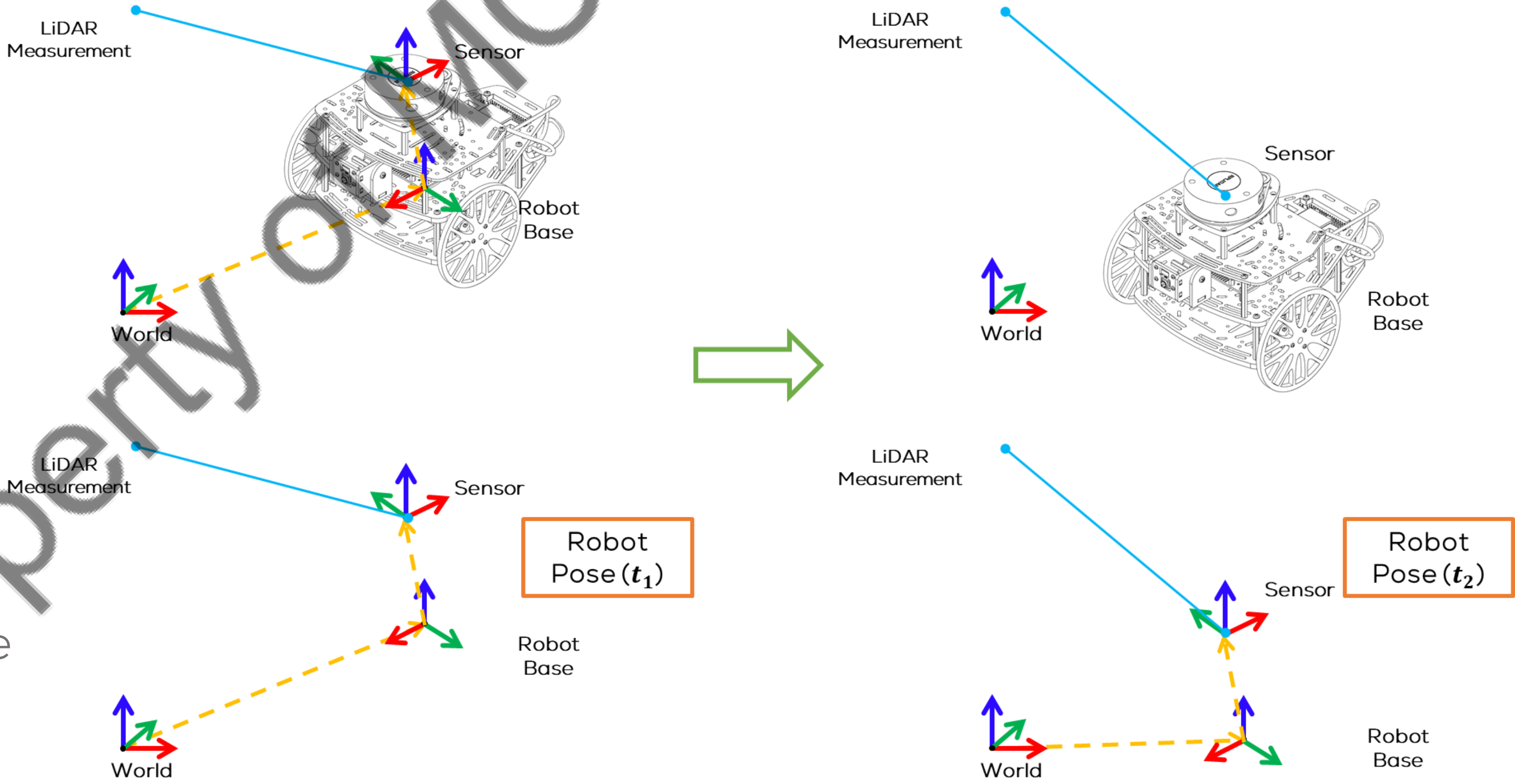
- The user can "listen" to a transformation if there is a link between them ,and there are no timing errors.
- In ROS, the capability of "listening" to a transform is divided into the Buffer and the Listener. These objects are essential for receiving and managing transformation data.
- Where the buffer is primarily used for storing and managing the history of transformation data. It acts as a buffer to keep track of transformation information over time.

- The listener is a higher-level interface built on top of the tf2::Buffer. It simplifies the process of querying transformations between frames for common use cases.



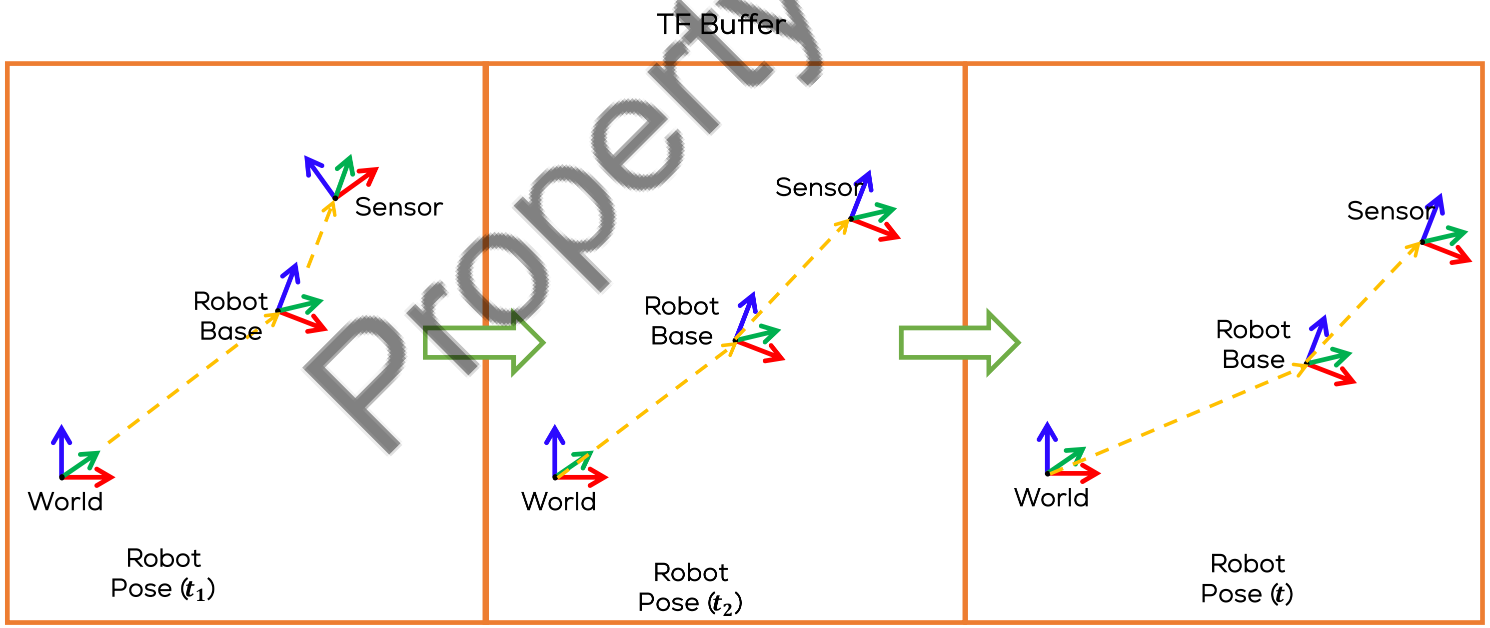
ROS TF2 Library

- As stated, ROS provides a simple way to listen to transformation using its [*tf2* library](https://).
- *tf2* maintains the relationship between coordinate frames in a tree structure buffered in time.
- The library lets the user transform points, vectors, etc., between any two coordinate frames at any desired point in time.



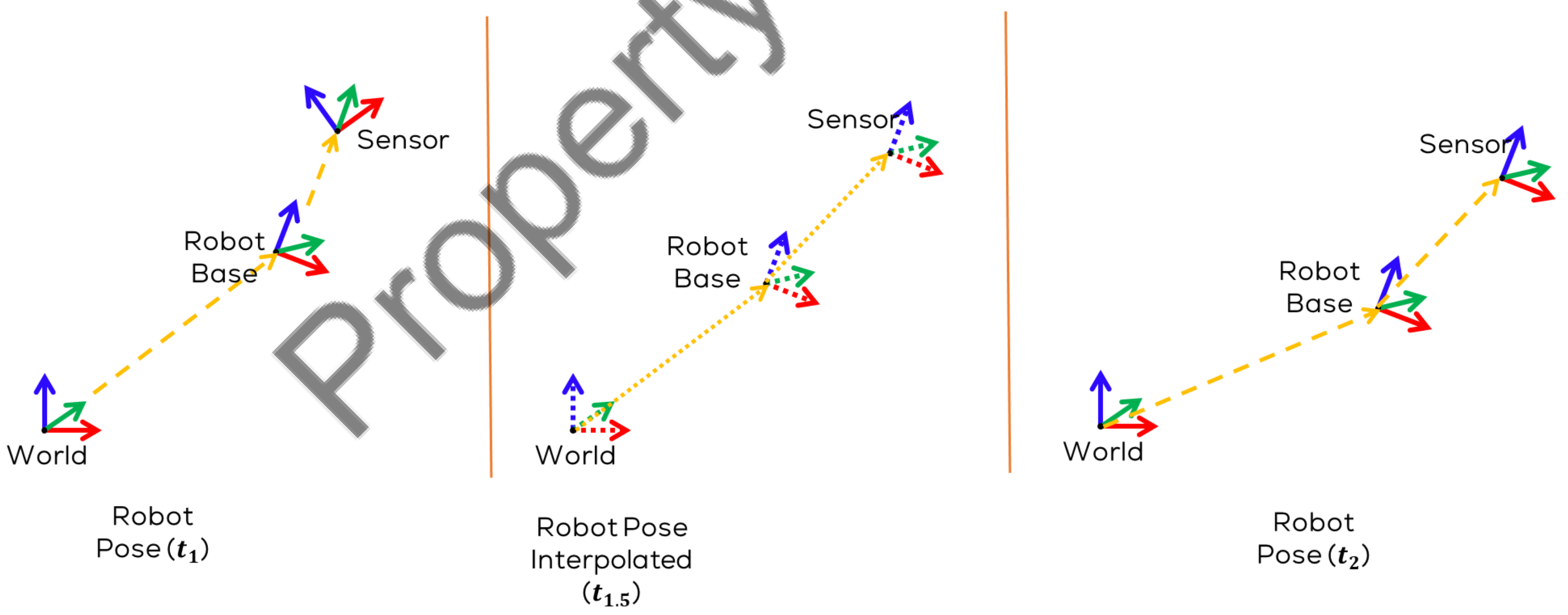
Coordinate transformations in ROS

- One of the most powerful options that the Listener of the TF2 library provides, is the ability to "Time Travel" in other words, the buffer maintains a history of transformation data, enabling you to query past transformations.
- The buffer can use this "History" to extrapolate or interpolate transformations and data.
- This capability can be used for tasks such as sensor fusion and control algorithms



Coordinate transformations in ROS

- **Extrapolation**: The buffer can extrapolate transformations when necessary, such as when you need to estimate a future transformation based on the history of data. For example, you might predict the position of an object a few milliseconds into the future.
- **Interpolation**: When you request a transformation at a specific time that falls between two available transformations, the buffer can interpolate the transformation data to provide a smooth and accurate result.



Activity 3

TF Listener

{Learn, Create, Innovate};

MCR²
Manchester Robotics

Activity 3

1. Crete a new node called "*tf_listener.py*" inside the previously defined package *tf_examples* *(full code on GitHub)*

```
cd ~/<YOUR_WS>/tf_examples/tf_examples
touch tf_listener.py
```

2. Make the file executable

```
sudo chmod +x tf_listener.py
```

3. Modify the "setup.py" entry points to include the newly created node to the

```
    entry_points={
        'console_scripts': [
            'static_tf = tf_examples.static_tf:main',
            'dyn_tf = tf_examples.dyn_tf:main',
            'tf_listener = tf_examples.tf_listener:main'
        ],
    },
```

4. Declare the transform listener and buffer in on the same section as a subscriber, using the following code (full code on Git Hub).

```
self.tf_buffer = Buffer()
self.tf_listener = TransformListener(self.tf_buffer, self)
```

5. Get the transform in the variable *transformation* as follows (main loop)

```
try:
  self.transformation = self.tf_buffer.lookup_transform(
            self.to_frame,
            self.from_frame,
            rclpy.time.Time())
except TransformException as ex:
  self.get_logger().info(
            f'Could not transform {self.to_frame} to
            {self.from_frame}: {ex}')
  return
```

Activity 3

6. Modify the static_tf_launch.py file to include the "tf_listener" node

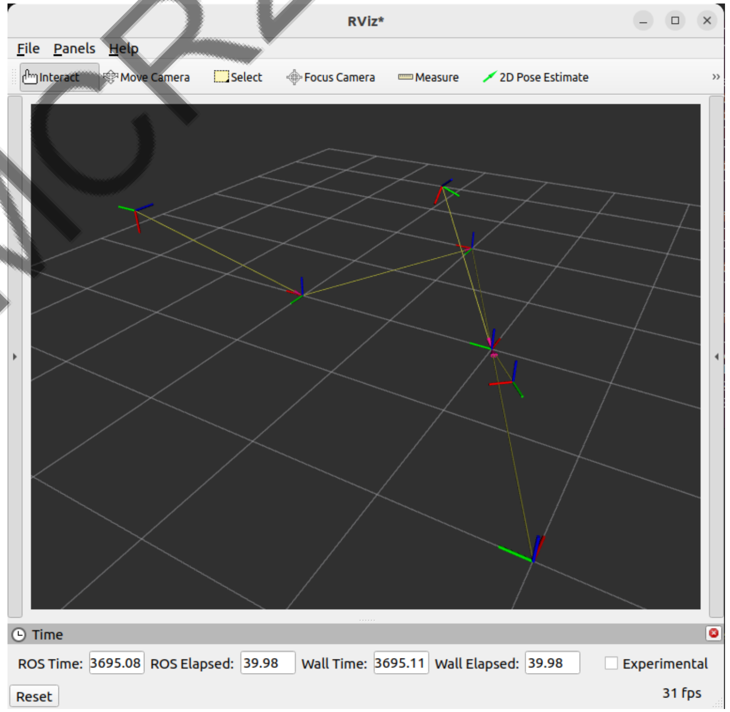
```
transform_listener_node = Node(
                            name="tf_listener",
                            package='tf_examples',
                            executable='tf_listener',
                            output = 'screen'
                            )

l_d = LaunchDescription([static_transform_node,
static_transform_node_2 , rqt_tf_tree_node, rviz_node,
static_transform_node_3, dynamic_transform_node,
transform_listener_node])
```

7. Build and launch the file

```
$ ros2 launch tf_examples static_tf_launch.py
```

8. Run rviz, add TF and set the fixed frame as "world"




```
[tf_listener-7] geometry_msgs.msg.TransformStamped(header=std_msgs.msg.Header(stamp=builtin_interfaces.msg.Time(sec=1739381027, nanosec=233808573), frame_id='world'), child_frame_id='moving_robot_3', transform=geometry_msgs.msg.Transform(translation=geometry_msgs.msg.Vector3(x=0.41568244420217637, y=-0.27786346572031473, z=0.0), rotation=geometry_msgs.msg.Quaternion(x=0.0, y=0.0, z=0.8819656828473059, w=-0.47131362624019824)))
[tf_listener-7] Transform ation Matrix:
[tf_listener-7] [[-0.55572693  0.83136489  0.          0.41568244]
[tf_listener-7]  [-0.83136489 -0.55572693  1.         -0.27786347]
[tf_listener-7]  [ 0.          0.          1.          0.        ]
[tf_listener-7]  [ 0.          0.          0.          1.        ]]
```

Thank you

{Learn, Create, Innovate};

MCR2

Manchester Robotics

T&C

Terms and conditions

`{Learn, Create, Innovate};`

Manchester Robotics

Terms and conditions