{Learn, Create, Innovate};

ROS

Visualisation Tools

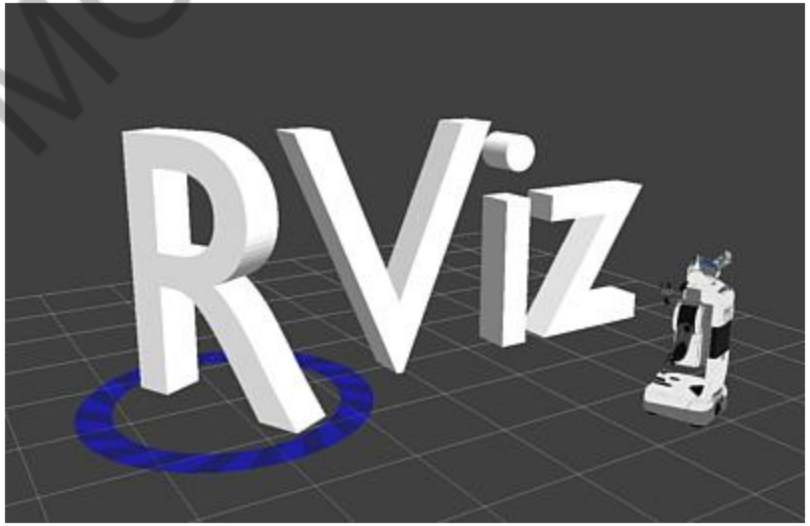
Manchester Robotics

RVIZ

What is RVIZ?

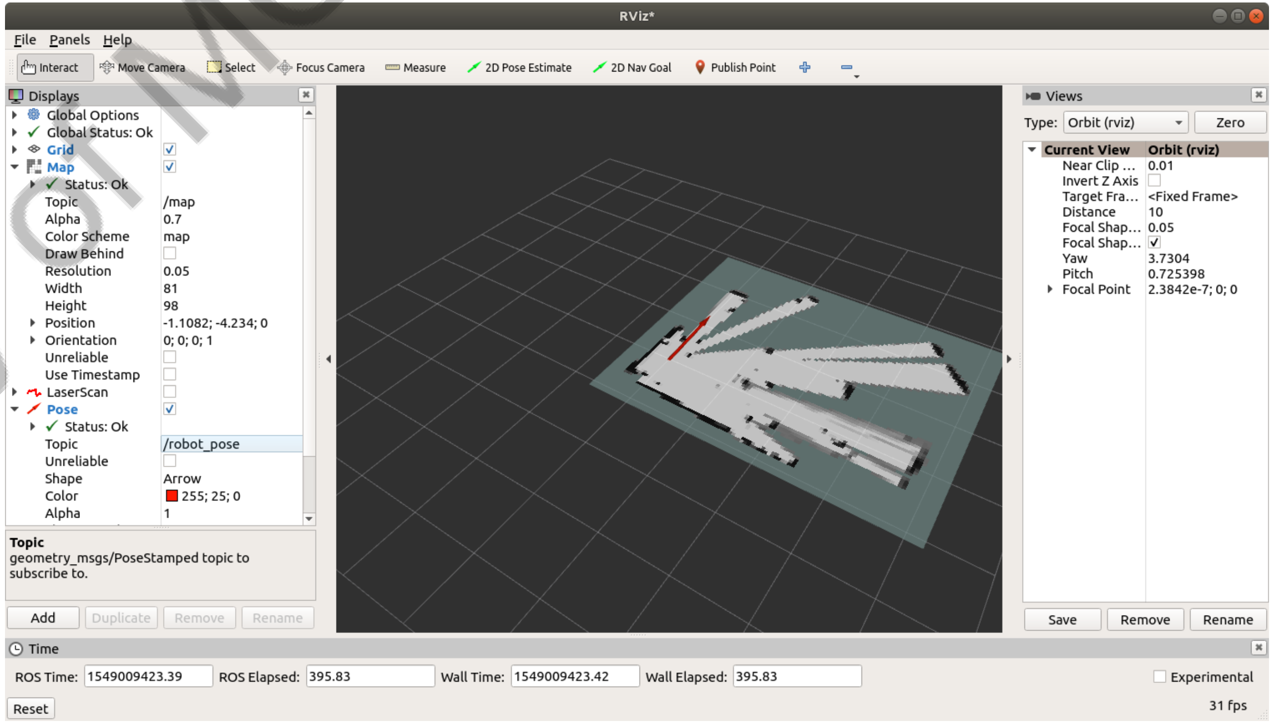
- RVIZ (ROS Visualization)
- Is a 3D visualisation environment
- Made to simplify debugging using visual tools.
- RVIZ allows the user to see what the robot is seeing, thinking and doing.

  “See the world through the robot’s eyes.”



RVIZ

- There are two main ways of putting data into RVIZ.
  - Via messages, where it understands sensors and state information, like laser scans, point clouds, cameras, and coordinate frames.
    - They have specialised displays to let the user configure how to view that information.
  - Information markers, letting the user send cubes, arrows and lines coloured however you want.
- The combination of sensor data and custom visualisation markers makes RVIZ a powerful tool for robotic development.



RVIZ

Quick Start (USB camera)

- Download the rospackage usb_cam

```
sudo apt install ros-<$DISTRO>-usb-cam
```

- Start ROS core (each step in a new terminal)

```
roscore
```

- Run the camera driver

```
rosrun usb_cam usb_cam_node _camera_name:='usb_cam' _camera_frame_id:='usb_cam’
```

*Add parameter *_pixel_format:=yuyv* if there is an error while decoding the frame

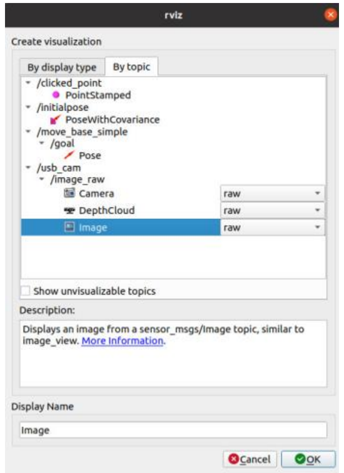
- Check that the topics are being published

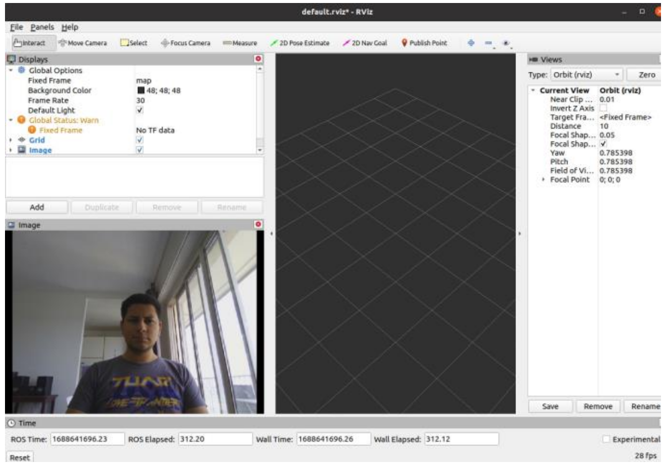
```
rostopic list
```

- Start RVIZ

```
rosrun rviz rviz
```

- Press the “add” button
- Go to the tab ”By topic”
- Add the topic Camera, located under the topics /usb_cam -> /image_raw





RVIZ

Markers

{Learn, Create, Innovate};

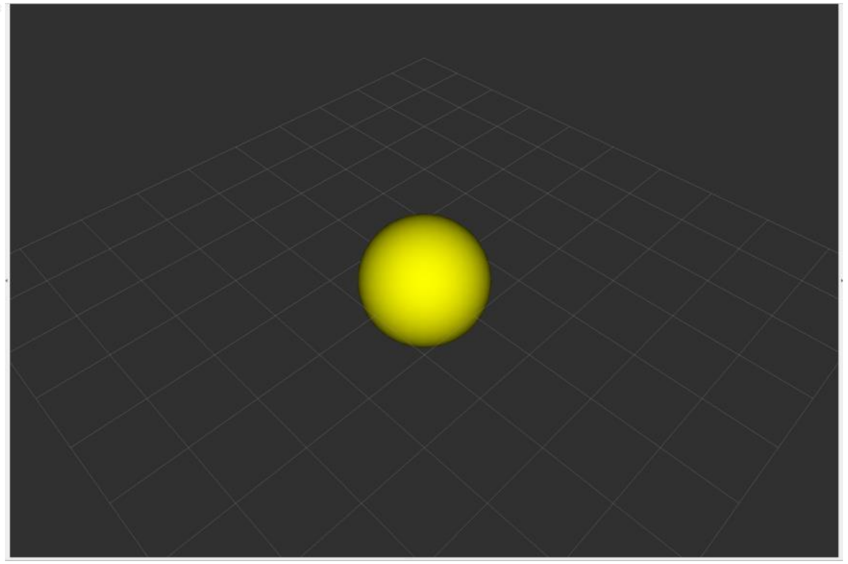
MCR2

Manchester **Robotics**

Markers

What are markers?

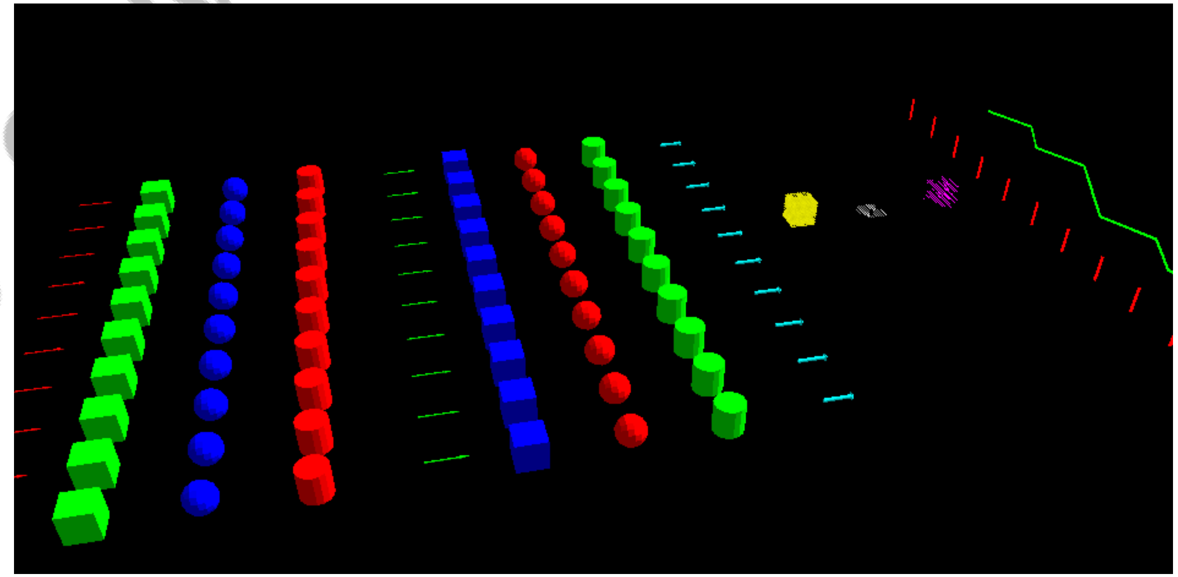
- One of the key features of RViz is the ability to visualize markers.
- Markers are graphical objects that represent different types of data in the 3D space.
- They can display points, lines, meshes, text, and more.
- Markers are typically published as ROS messages and can be subscribed to by RViz for visualisation.
- RViz provides a user-friendly interface for adding, configuring, and visualizing markers.



Markers

Type of markers?

- There are different types of markers that help us to visualise information in RViz.
- The basic markers are points, cubes, spheres, arrows, and lines.
- The user can set up and define its own markers and make them move.



Markers

- Markers are defined as ROS messages in which the user inputs the type of marker, configuration and characteristics. More information here and here.

```
uint8 ARROW=0, uint8 CUBE=1, uint8 SPHERE=2, uint8 CYLINDER=3, uint8 LINE_STRIP=4
uint8 LINE_LIST=5, uint8 CUBE_LIST=6, uint8 SPHERE_LIST=7, uint8 POINTS=8, uint8 TEXT_VIEW_FACING=9, uint8 MESH_RESOURCE=10,
uint8 TRIANGLE_LIST=11

uint8 ADD=0, uint8 MODIFY=0, uint8 DELETE=2,
uint8 DELETEALL=3

Header header                     # header for time/frame information
string ns                         # Namespace to place this object in... used in conjunction with id to create a unique name for the object
int32 id                         # object ID useful in conjunction with the namespace for manipulating and deleting the object later
int32 type                       # Type of object
int32 action                      # 0 add/modify an object, 1 (deprecated), 2 deletes an object, 3 deletes all objects
geometry_msgs/Pose pose              # Pose of the object
geometry_msgs/Vector3 scale          # Scale of the object 1,1,1 means default (usually 1 meter square)
std_msgs/ColorRGBA color          # Color [0.0-1.0]
duration lifetime                 # How long the object should last before being automatically deleted.  0 means forever
bool frame_locked                 # If this marker should be frame-locked, i.e. retransformed into its frame every timestep

#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
geometry_msgs/Point[] points
#Only used if the type specified has some use for them (eg. POINTS, LINE_STRIP, ...)
#number of colors must either be 0 or equal to the number of points
#NOTE: alpha is not yet used
std_msgs/ColorRGBA[] colors

# NOTE: only used for text markers
string text

# NOTE: only used for MESH_RESOURCE markers
string mesh_resource
bool mesh_use_embedded_materials
```

Activity 1

Markers

{Learn, Create, Innovate};

Manchester Robotics

Marker Generator

- Make a new package called "markers", with the following packages
  - rospy, std_msgs, tf2_ros, visualization_msgs tf_conversions, geometry_msgs

```
catkin_create_pkg markers rospy std_msgs tf2_ros visualization_msgs tf_conversions geometry_msgs tf2_geometry_msgs
```

- Create a node called *marker.py* inside the scripts folder

```
mkdir scripts && touch scripts/marker.py
```

- Give executable permission to the file

```
cd ~/catkin_ws/src/markers/scripts/
sudo chmod +x marker.py
```

- Modify the CMake file to include the newly created node to the

```
catkin_install_python(PROGRAMS scripts/marker.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Marker Generator

- Open the file *marker.py*
- Define a new marker called sun in the message declaration.

```
# Message Declaration
sun = Marker()
```

- Define the marker in “def init_sun” function, as shown on the right.
- Define its publisher

```
#Setup Publishers and subscribers here
pub_sun = rospy.Publisher('/sun', Marker, queue_size=1)
```

- Publish the marker (updating the time stamp).

```
while not rospy.is_shutdown():
        #Update time stamp
        sun.header.stamp = rospy.Time.now()
        #Publish the marker
        pub_sun.publish(sun)
        loop_rate.sleep()
```

```
#Header
    sun.header.frame_id = "sun"
    sun.header.stamp = rospy.Time.now()
    #Set Shape, Arrow: 0; Cube: 1 ; Sphere: 2 ; Cylinder: 3
    sun.id = 0
    sun.type = 2
    #Add Marker
    sun.action = 0     #Action 0 add/modify, 2 delete object, 3 deletes all objects
    # Set the pose of the marker
    sun.pose.position.x = 0.0
    sun.pose.position.y = 0.0
    sun.pose.position.z = 0.0
    sun.pose.orientation.x = 0.0
    sun.pose.orientation.y = 0.0
    sun.pose.orientation.z = 0.0
    sun.pose.orientation.w = 1.0
    # Set the scale of the marker
    sun.scale.x = 2.0
    sun.scale.y = 2.0
    sun.scale.z = 2.0
    # Set the colour
    sun.color.r = 1.0
    sun.color.g = 1.0
    sun.color.b = 0.0
    sun.color.a = 1.0
    #Set Duration
    sun.lifetime = rospy.Duration(0)
```

Marker Generator

```python
#!/usr/bin/env python
import rospy
from visualization_msgs.msg import Marker
from std_msgs.msg import Time

#Setup parameters, variables and callback functions here (if required)
# Declare message
sun = Marker()
def init_sun():
    #Header
    sun.header.frame_id = "sun"
    sun.header.stamp = rospy.Time.now()
    #Set Shape, Arrow: 0; Cube: 1 ; Sphere: 2 ; Cylinder: 3
    sun.id = 0
    sun.type = 2
    #Add Marker
    sun.action = 0
    # Set the pose of the marker
    sun.pose.position.x = 0.0
    sun.pose.position.y = 0.0
    sun.pose.position.z = 0.0
    sun.pose.orientation.x = 0.0
    sun.pose.orientation.y = 0.0
    sun.pose.orientation.z = 0.0
    sun.pose.orientation.w = 1.0
    # Set the scale of the marker
    sun.scale.x = 2.0
    sun.scale.y = 2.0
    sun.scale.z = 2.0
    # Set the color
    sun.color.r = 1.0
    sun.color.g = 1.0
    sun.color.b = 0.0
    sun.color.a = 1.0
    #Set Duration
    sun.lifetime = rospy.Duration(0)
```

```python
#Stop Condition
def stop():
    print("Stopping")


if __name__=='__main__':

    #Initialise and Setup node
    rospy.init_node("RVIZ_marker")

    # Configure the Node
    loop_rate = rospy.Rate(10)
    rospy.on_shutdown(stop)
    print("The Sun is ready")

    #Setup the messages
    init_sun()

    #Setup Publishers and subscribers here
    pub_sun = rospy.Publisher('/sun', Marker, queue_size=1)

    try:
    #Run the node
        while not rospy.is_shutdown():

            sun.header.stamp = rospy.Time.now()

            pub_sun.publish(sun)
            loop_rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

Marker Generator

- Compile the program

```
cd ~/catkin_ws
catkin_make
```

- Start ROS

```
roscore
```

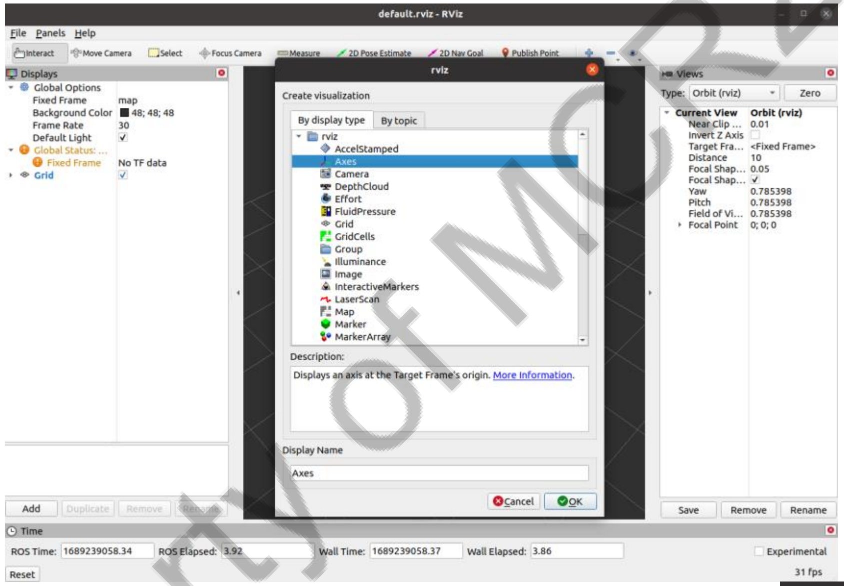
- Run the node

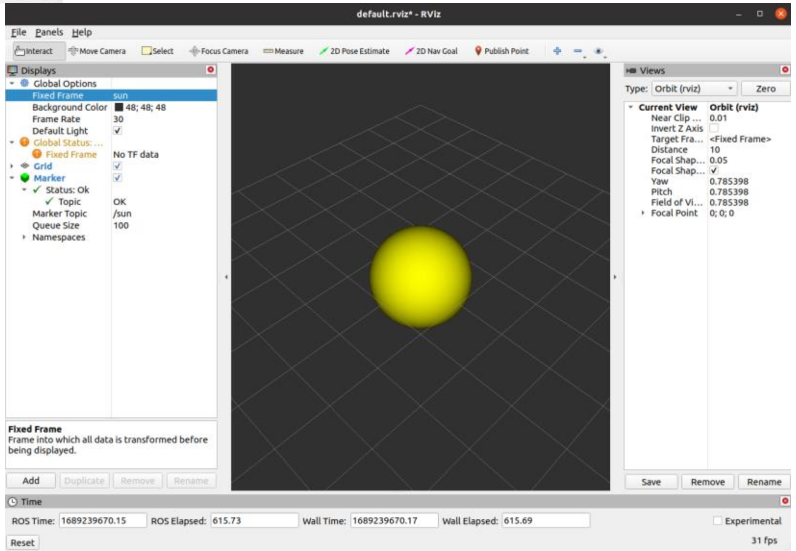
```
rosrun markers marker.py
```

- Start RViz

```
rosrun rviz rviz
```

- Add the marker
    - Press Add
    - ››By topic››/sun›› marker
- Change the fixed frame on top of RViz to “sun”





ROS Transformations

TF

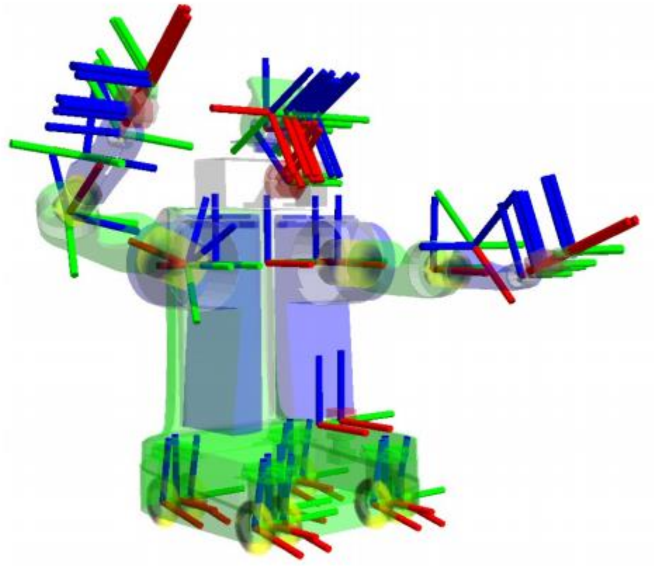
{Learn, Create, Innovate};

MCR²

Manchester **Robotics**

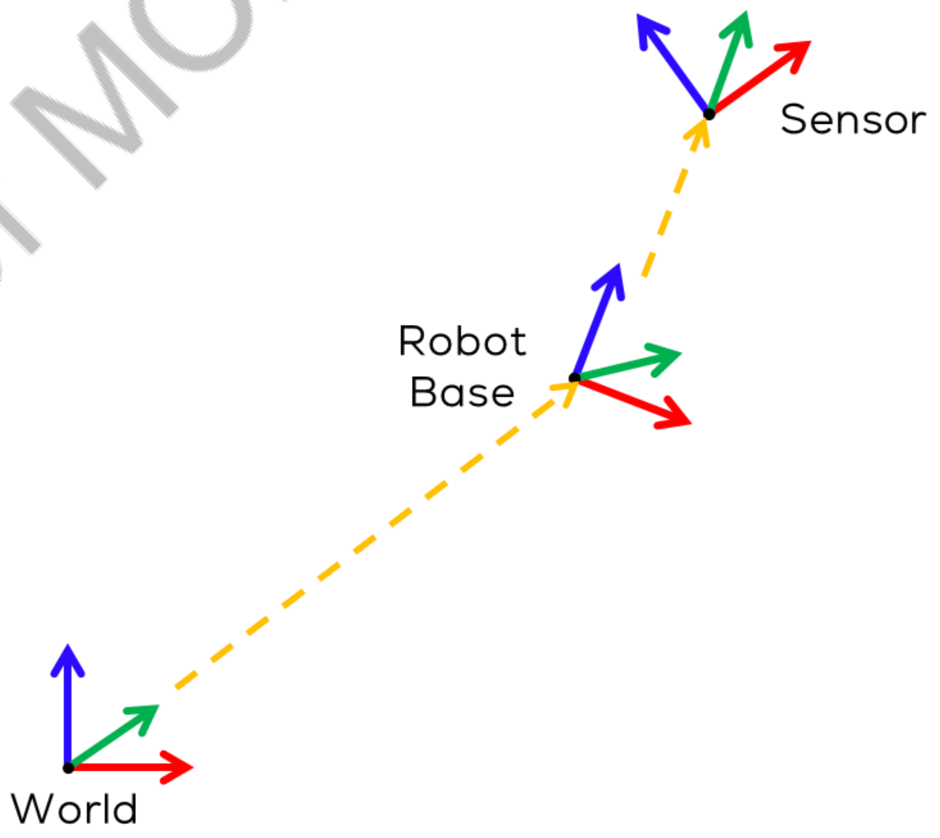
Coordinate transformations in ROS

- 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.



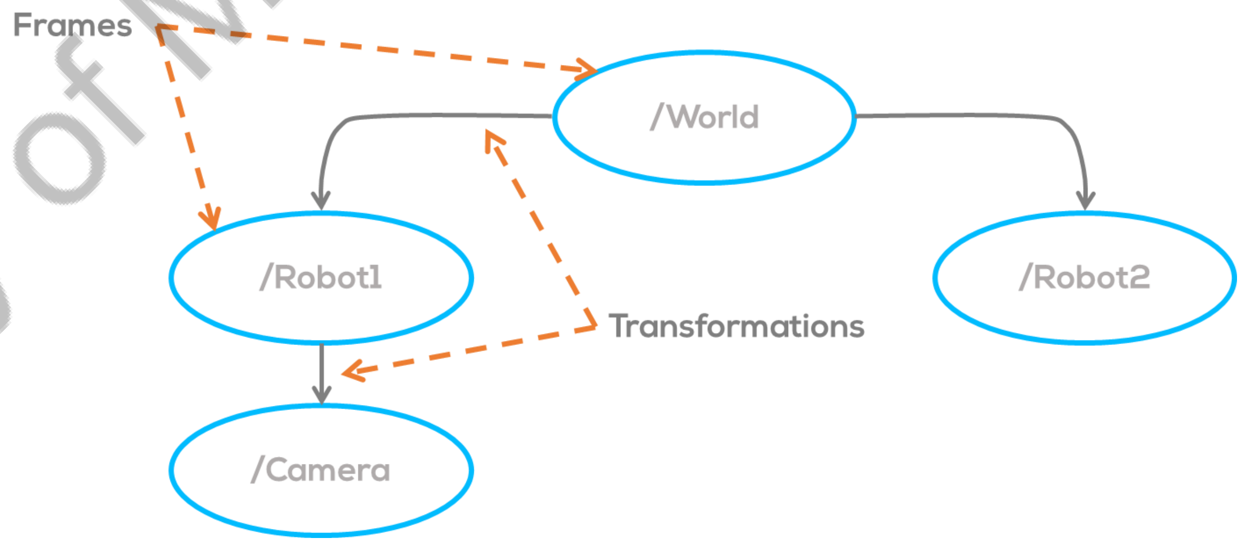
Coordinate transformations in ROS

- 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" 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.

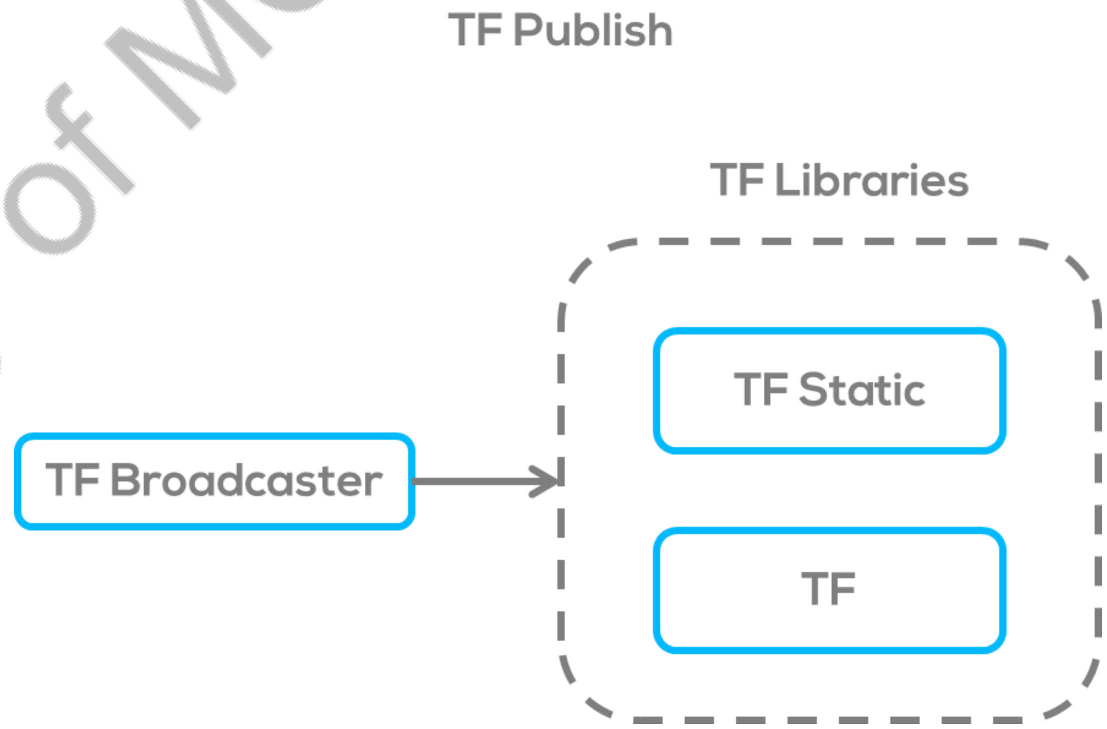


Coordinate transformations in ROS

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.

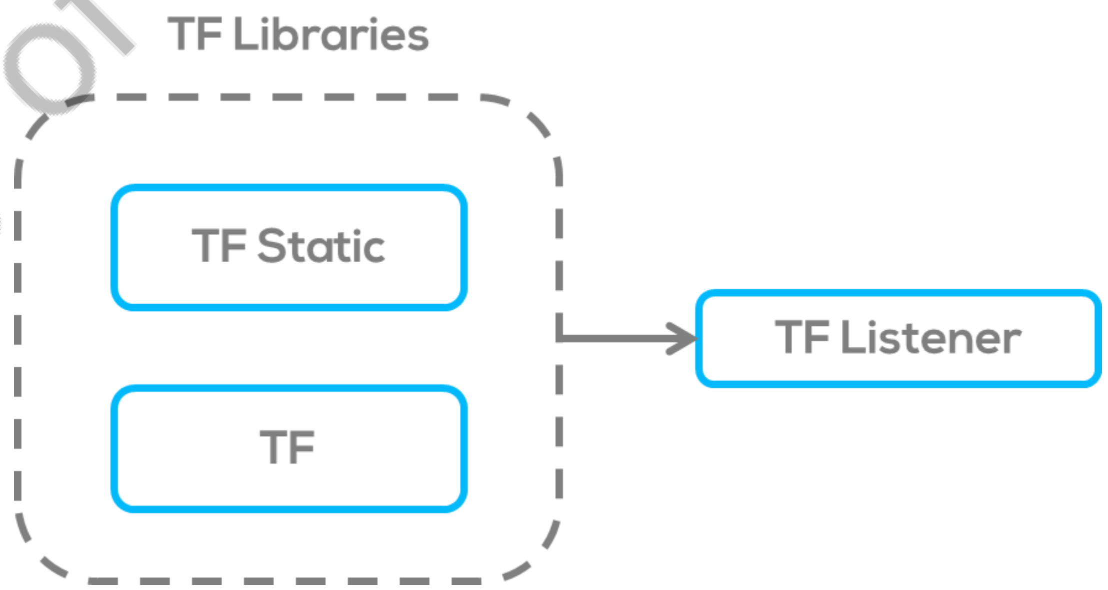


Coordinate transformations in ROS

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.

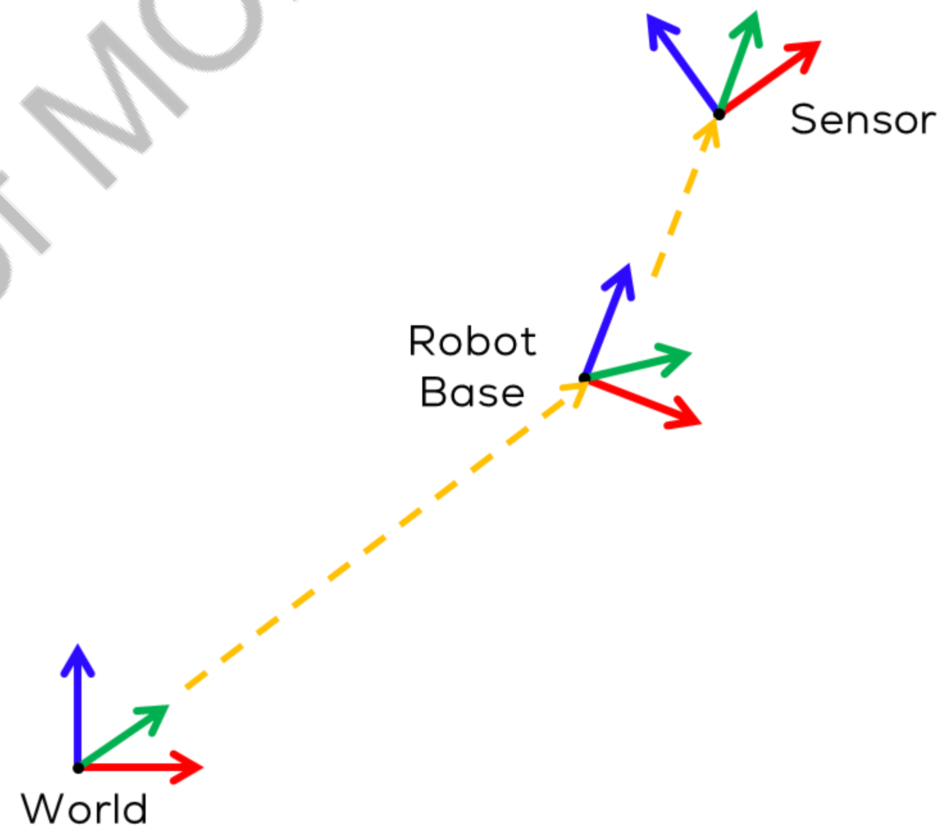
- This is particularly useful for performing coordinate frame transformations on points or vectors from different parts of the system.

TF Listening



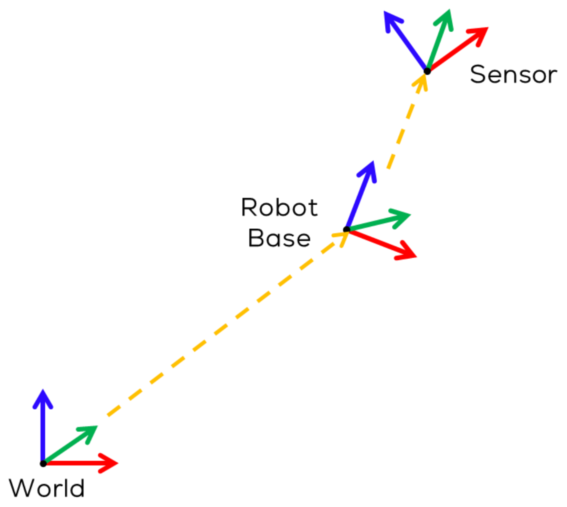
How it works?

- **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.



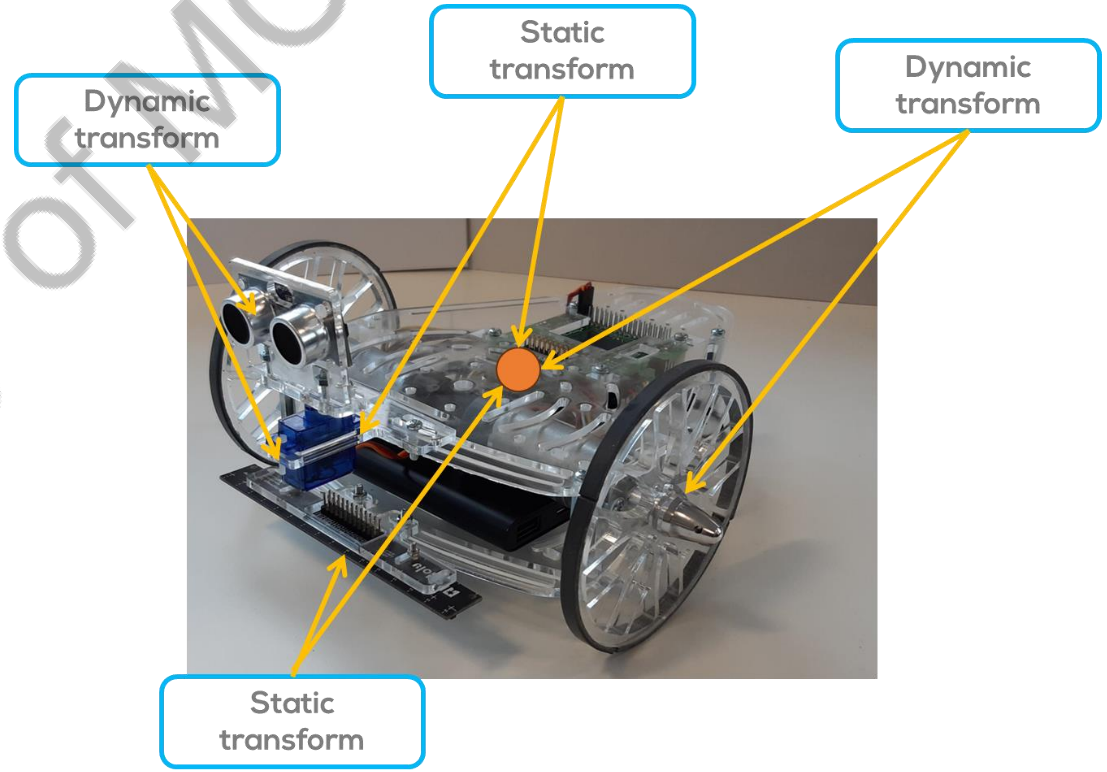
How it works?

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
```

Static Transform Example

Declaring a Static Transform Example

In the package *"markers"* create a launch file called *marker.launch"*

```
cd ~/catkin_ws/src/markers/ && mkdir launch
cd launch && touch marker.launch
```

- Write and save the following inside the launch file

```
<?xml version="1.0" ?>
<launch>
    <node name="marker" pkg="markers" type="marker.py" output="screen"/>

<node pkg="tf2_ros" type="static_transform_publisher" name="link1" args="2  1   1   3.1416  0   0   sun link1"/>
</launch>
```

- The Launch file Launches the previously created marker and creates a static transform.

- Launch the file

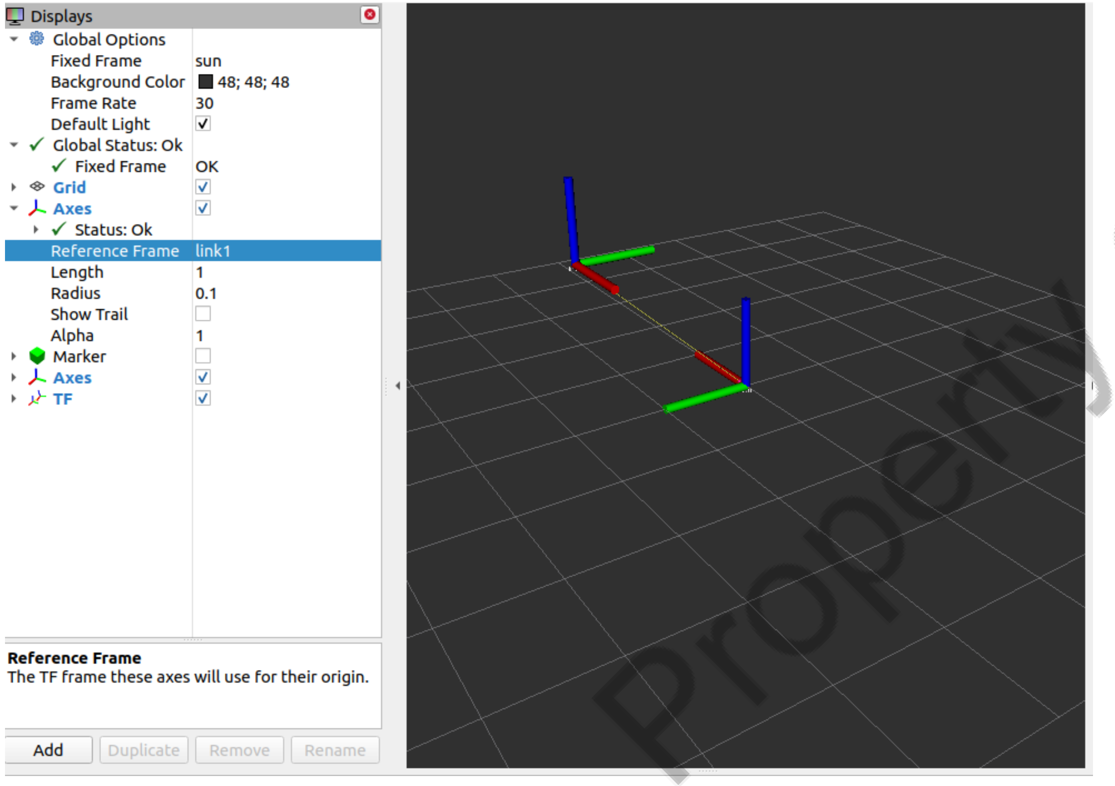
```
roslaunch marker marker.launch
```

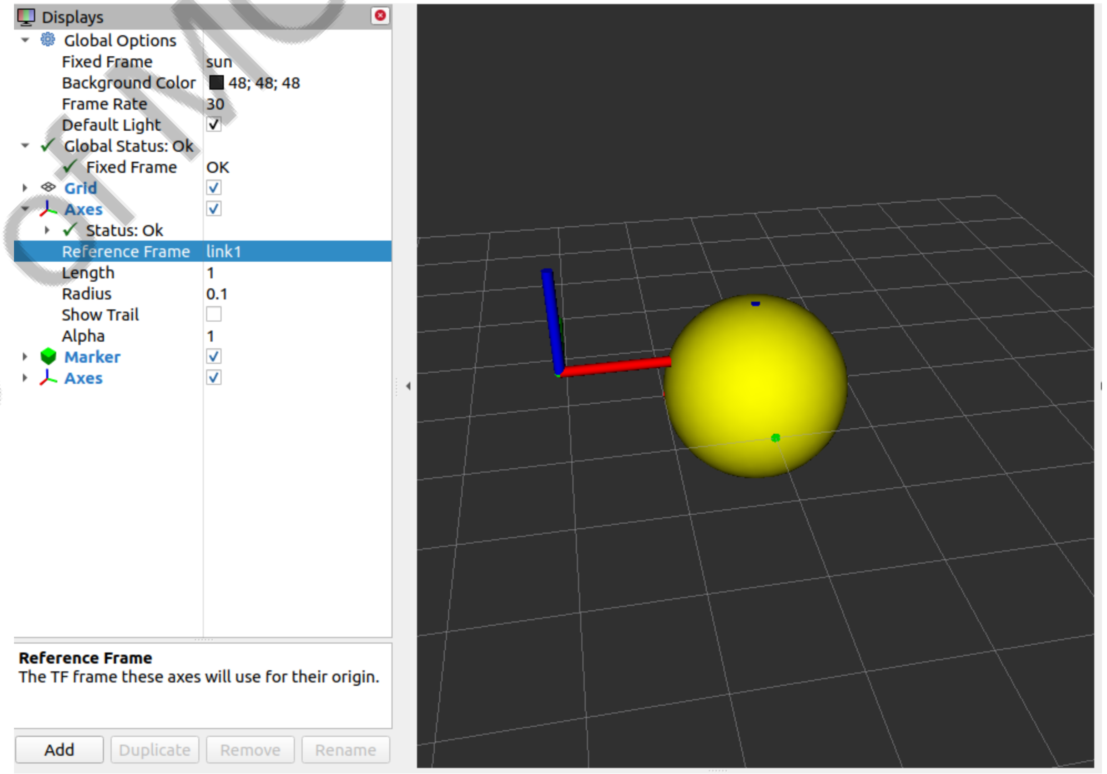
- Open RVIZ in another terminal

```
rosrun rviz rviz
```

- Change the Fixed Frame to "world"
- Click the button "Add" and on the "By display type" tab, select "Axes".
    - Repeat to Add two Axes
- Select one of the axes and change its "Reference Frame" to "link1"
- Click the button "Add" and on the "By Topic" tab, select Marker
- Click the button "Add" and on the "By display type" tab, select "TF".

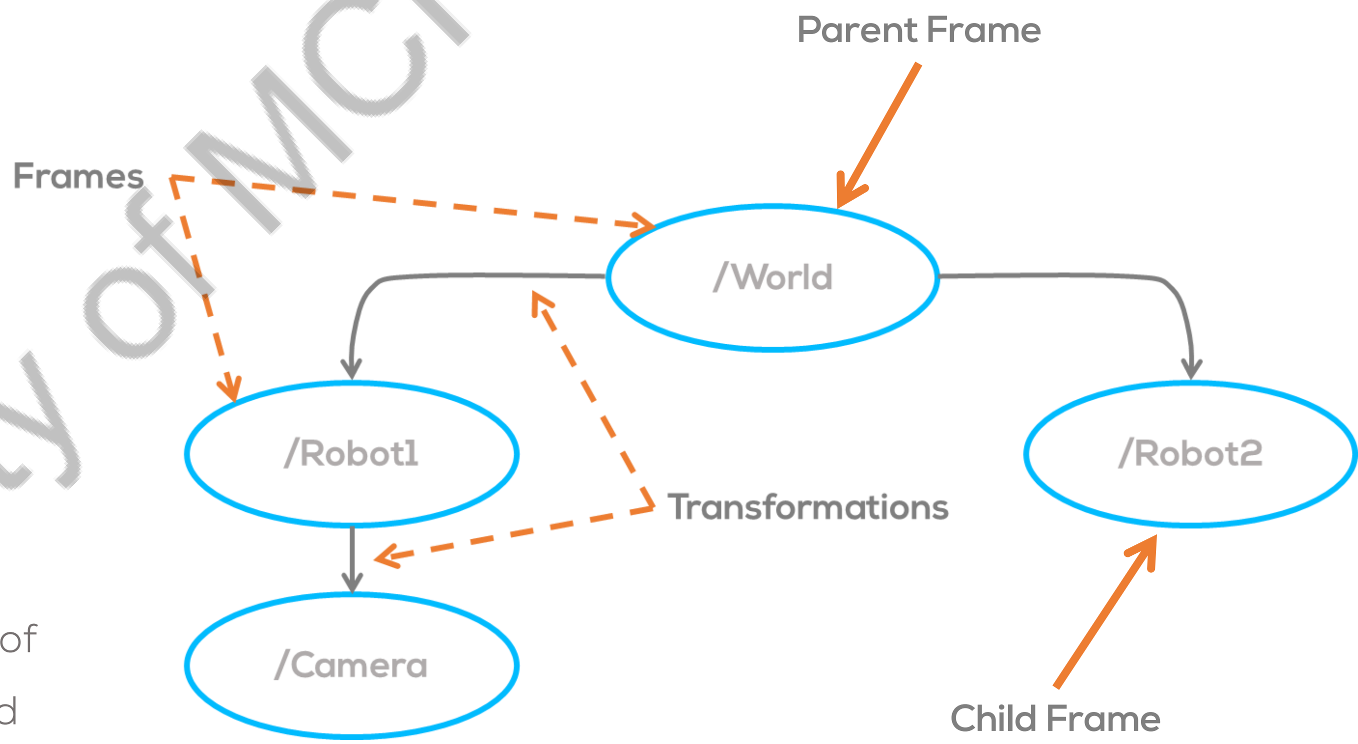
Static Transform Example





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 in ROS the tf_conversions package can be used.

```
q = tf_conversions.transformations.quaternion_from_euler(roll,pitch,yaw)
```

Transform Stamped Message

```
ex_tf = TransformStamped()
ex_tf.header.frame_id = "inertial_frame"
ex_tf.child_frame_id = "ex"
ex_tf.header.stamp = rospy.Time.now()
ex_tf.transform.translation.x = 1
ex_tf.transform.translation.y = 1
ex_tf.transform.translation.z = 1.0
ex_tf.transform.rotation.x = 0
ex_tf.transform.rotation.y = 0
ex_tf.transform.rotation.z = 0
ex_tf.transform.rotation.w = 1
```

Broadcasting / Listening a transform

- The idea of 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 for 10 s.

```
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)
trans = tfBuffer.lookup_transform(frame1, frame2,
rospy.Time(0))
```

Activity 2

Transformations

{Learn, Create, Innovate};

Manchester Robotics

Transforms

- In this activity, Static and Dynamic transforms will be generated in a script.
- In the package *"markers"* create a new node called *"tf_act.py"*

```
cd ~/catkin_ws/src/markers/scripts/
touch scripts/tf_act.py
```

- Give executable permission to the file

```
cd ~/catkin_ws/src/markers/scripts/
sudo chmod +x tf_act.py
```

- Modify the CMake file to include the newly created node to the

```
catkin_install_python(PROGRAMS scripts/marker.py scripts/tf_act.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

Activity 2

- Open the file *tf_act.py*
- Define three new frames called, inertial, sun and planet, their publishers and publish the transforms.
- Compile the program

```
cd ~/catkin_ws
catkin_make
```

- Start ROS

```
roscore
```

- Run the node

```
rosrun markers tf_act.py
```

- Start RViz

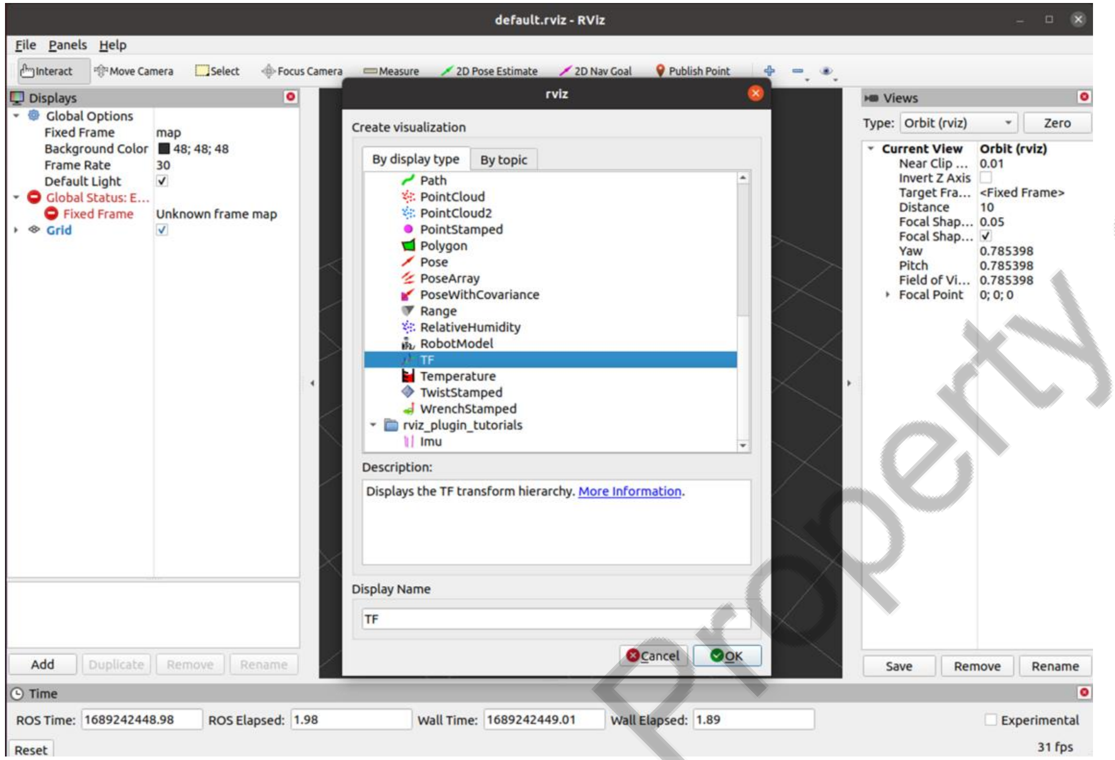
```
rosrun rviz rviz
```

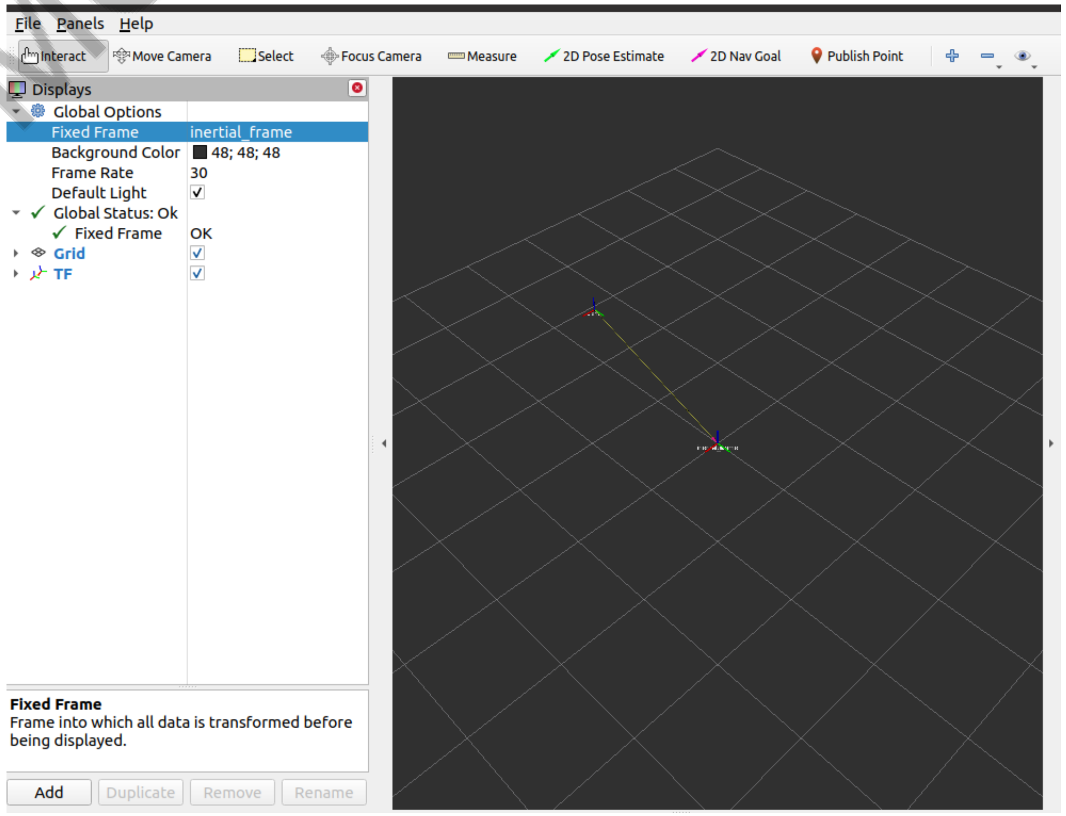
- Add the marker
  - Press Add
  - >>By display type>>TF

```
ex_tf = TransformStamped()
ex_tf.header.frame_id = "inertial_frame"
ex_tf.child_frame_id = "ex"
ex_tf.header.stamp = rospy.Time.now()
ex_tf.transform.translation.x = 1
ex_tf.transform.translation.y = 1
ex_tf.transform.translation.z = 1.0
ex_tf.transform.rotation.x = 0
ex_tf.transform.rotation.y = 0
ex_tf.transform.rotation.z = 0
ex_tf.transform.rotation.w = 1
```

Activity 2

- Change the fixed frame on top of RViz to *"inertial_frame"*





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 part of the marker message, 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.
- 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.

Activity 3

Planet

{Learn, Create, Innovate};

Manchester Robotics

Marker Generator

- In this activity the knowledge acquired in the previous two activities will be used to create a movable marker.
- Open the previously created package *"marker"*
- Change the "frame_id" of the sun to "planet" and modify the pose

```
sun.header.frame_id = "planet"

sun.pose.position.x = 0.5
sun.pose.position.y = 0.5
sun.pose.position.z = 0.5
```

- Make a roslaunch file to launch both the "maker" node and the "tf_ act" node

```
<?xml version="1.0" ?>
<launch>
    <node name="transforms" pkg="markers" type="tf_act.py" output="screen"/>
    <node name="sun" pkg="markers" type="marker.py" output="screen"/>
</launch>
```

- Launch the file and RVIZ. Make "inertial_frame" to be the fixed frame and add the marker.

ROS Transformations

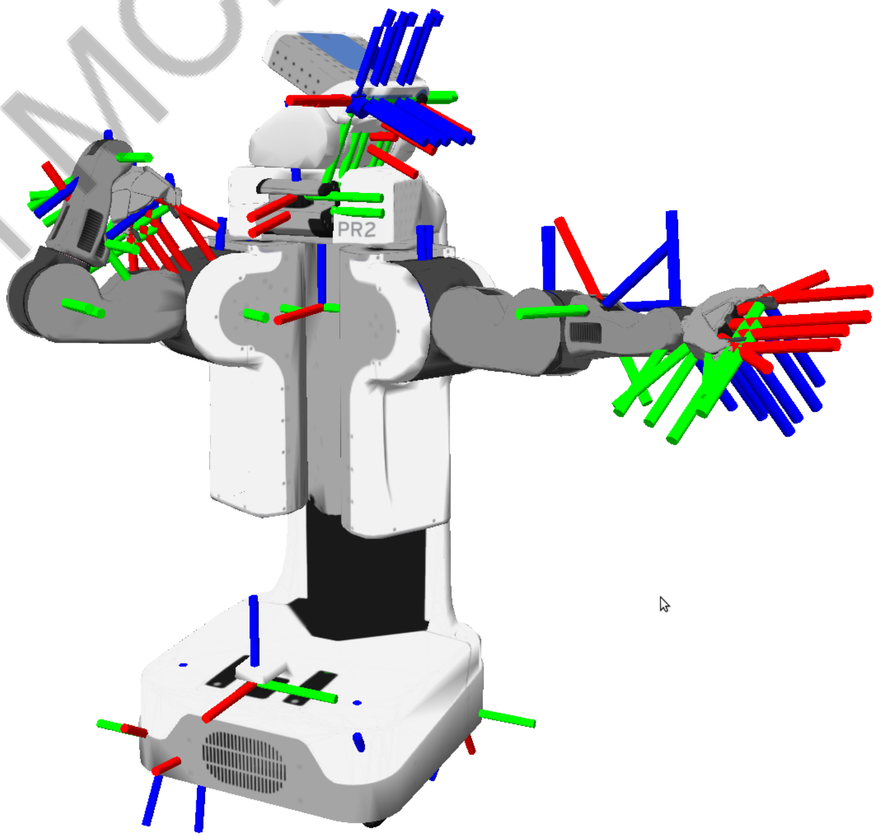
TF Listeners

{Learn, Create, Innovate};

Manchester Robotics

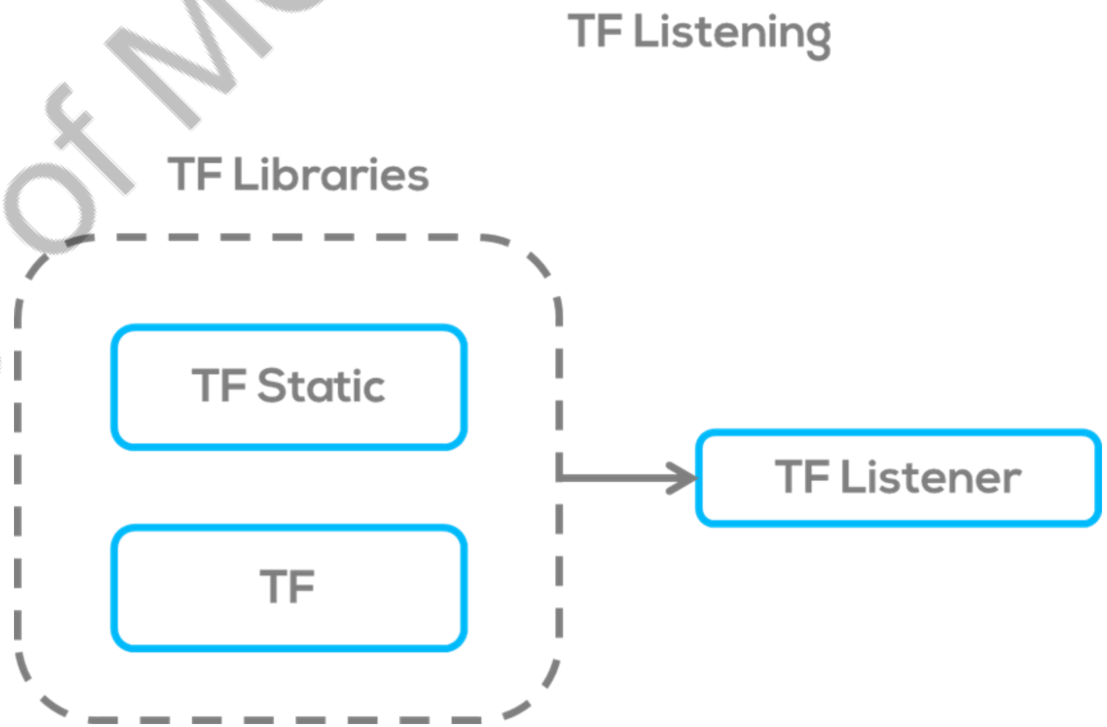
Transformation Listener in ROS

- 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...



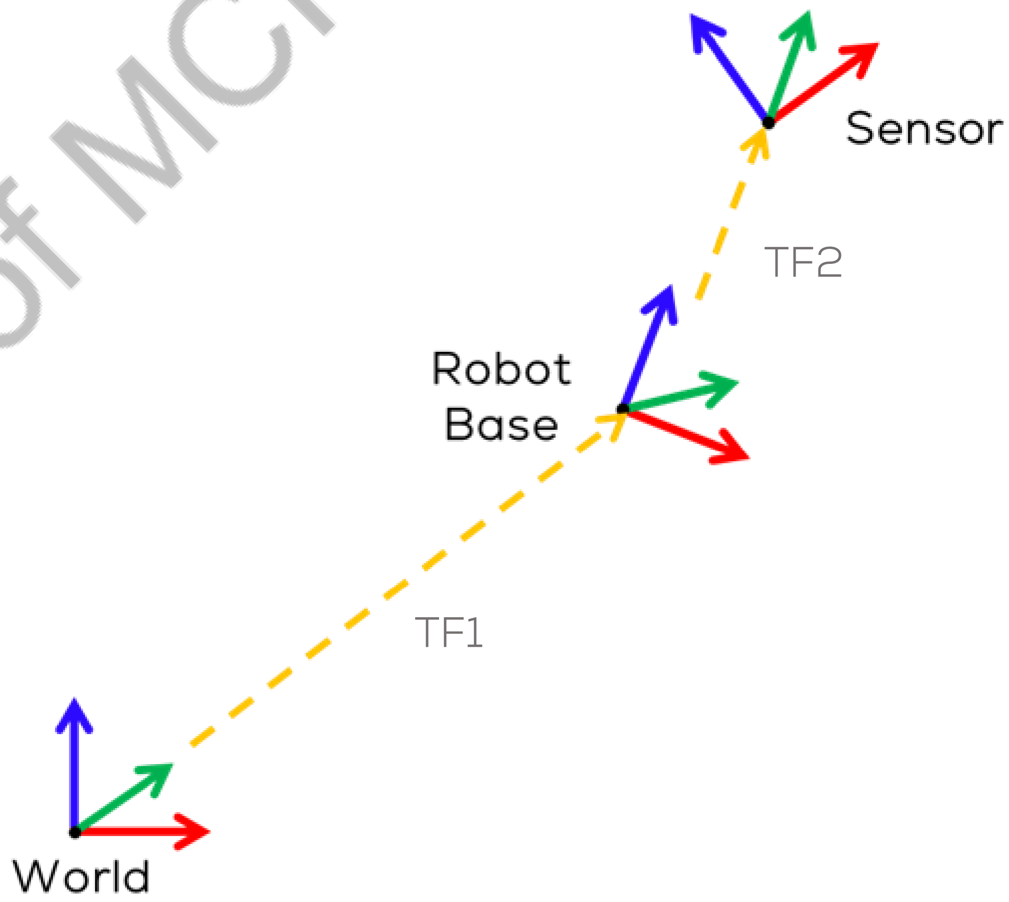
Transformation Listener in ROS

- 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.



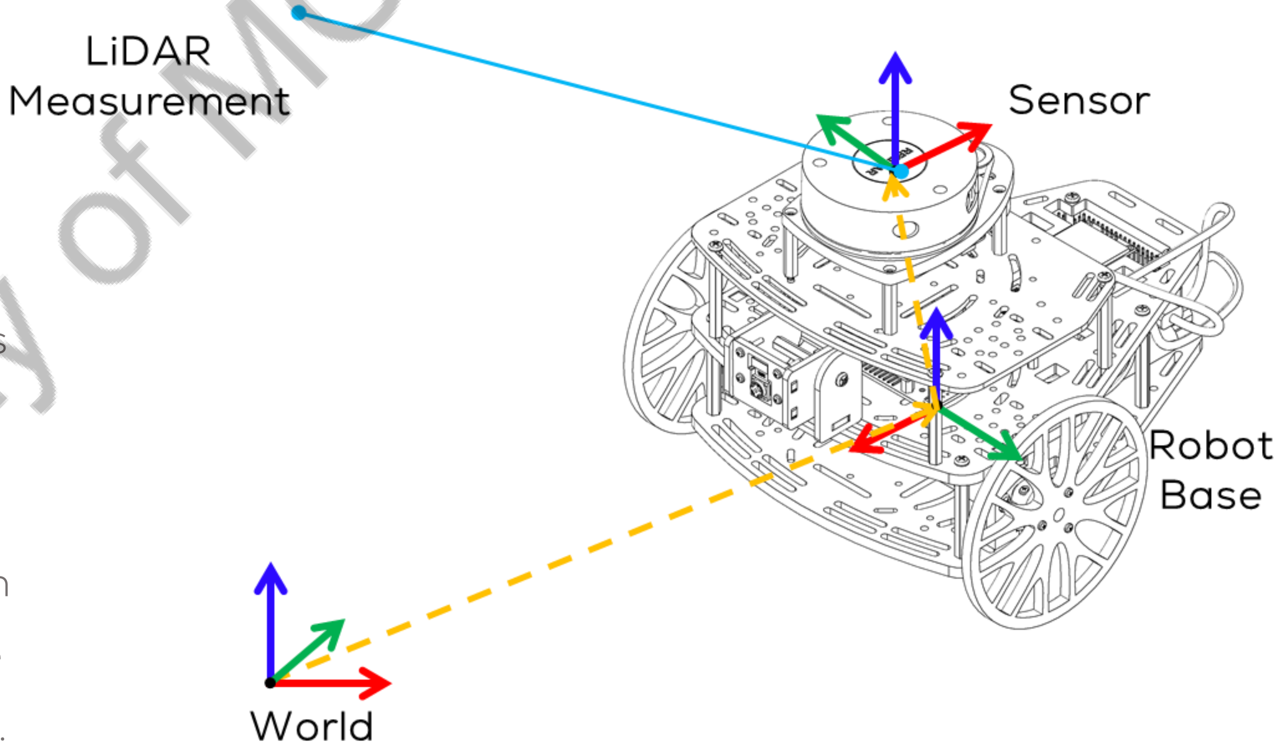
Transformation Listener in ROS

- 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.



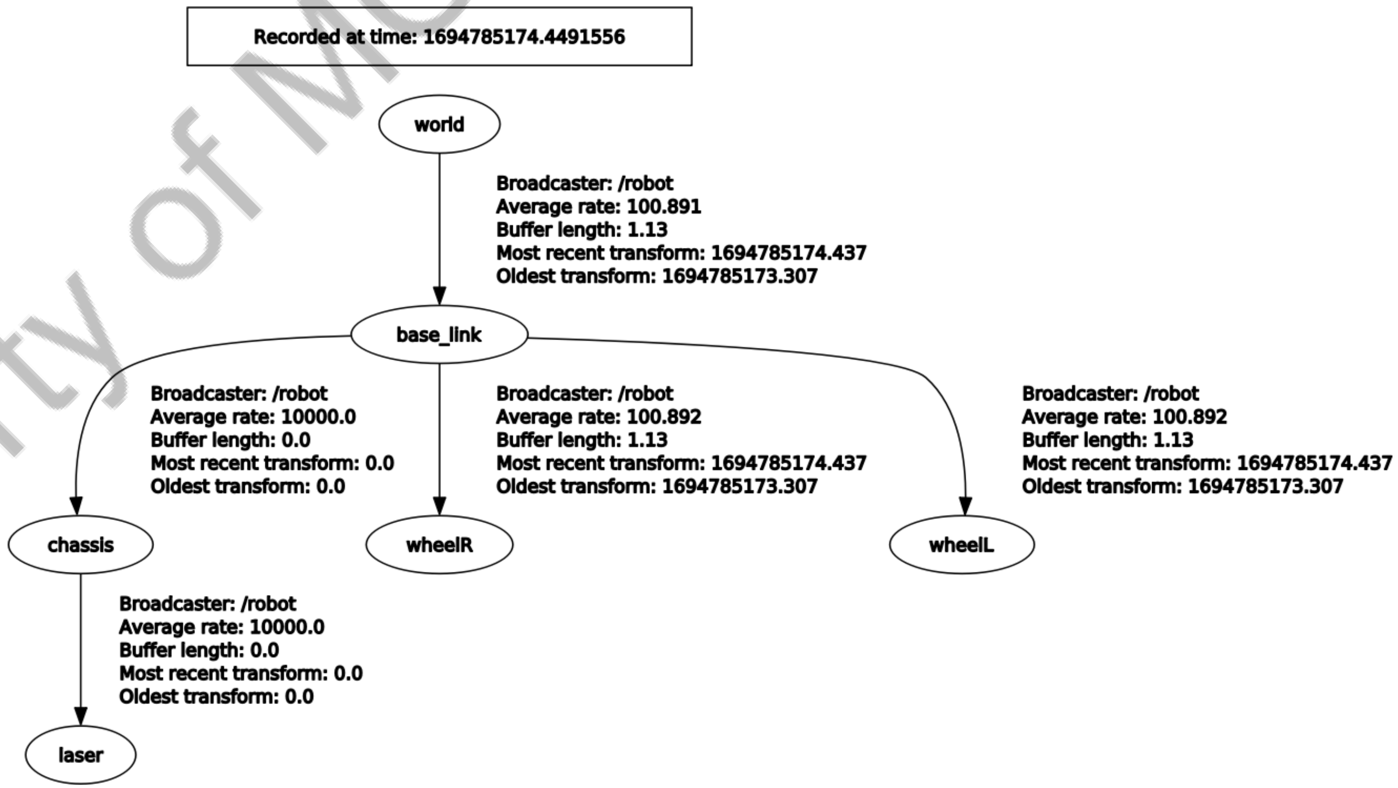
Transformation Listener ROS: Usage

- **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.



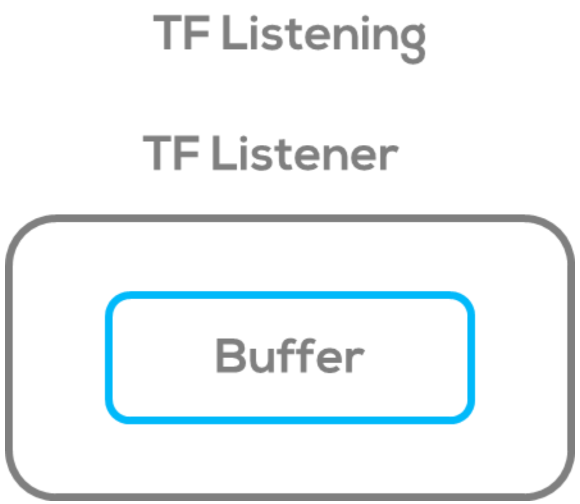
Transformation Listener ROS: Usage

- **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.
- **Listener API:** ROS provides a listener API that allows you to subscribe to transformation updates. You can use this API to receive notifications when transformations change.



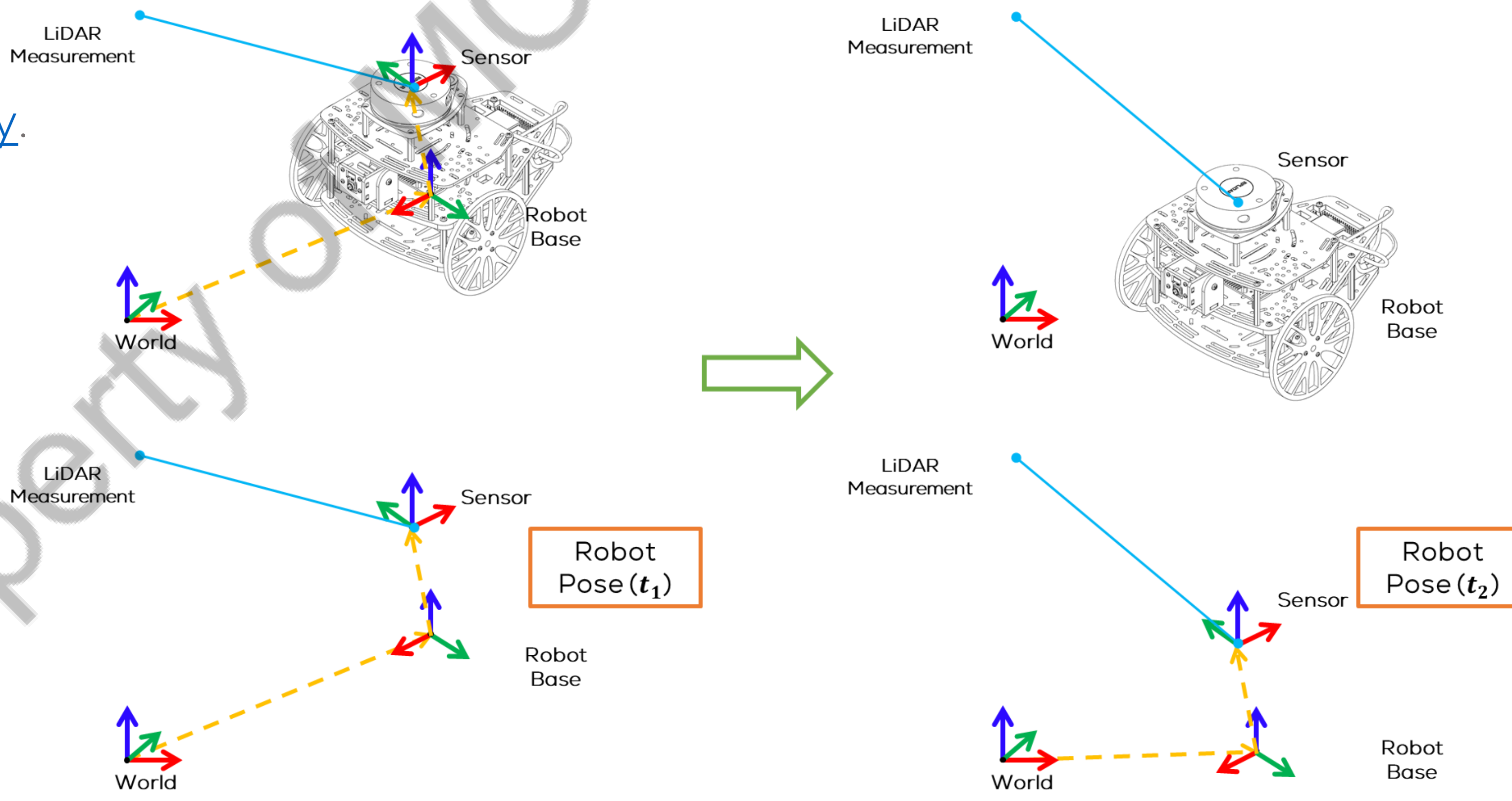
Coordinate transformations in ROS

- 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.



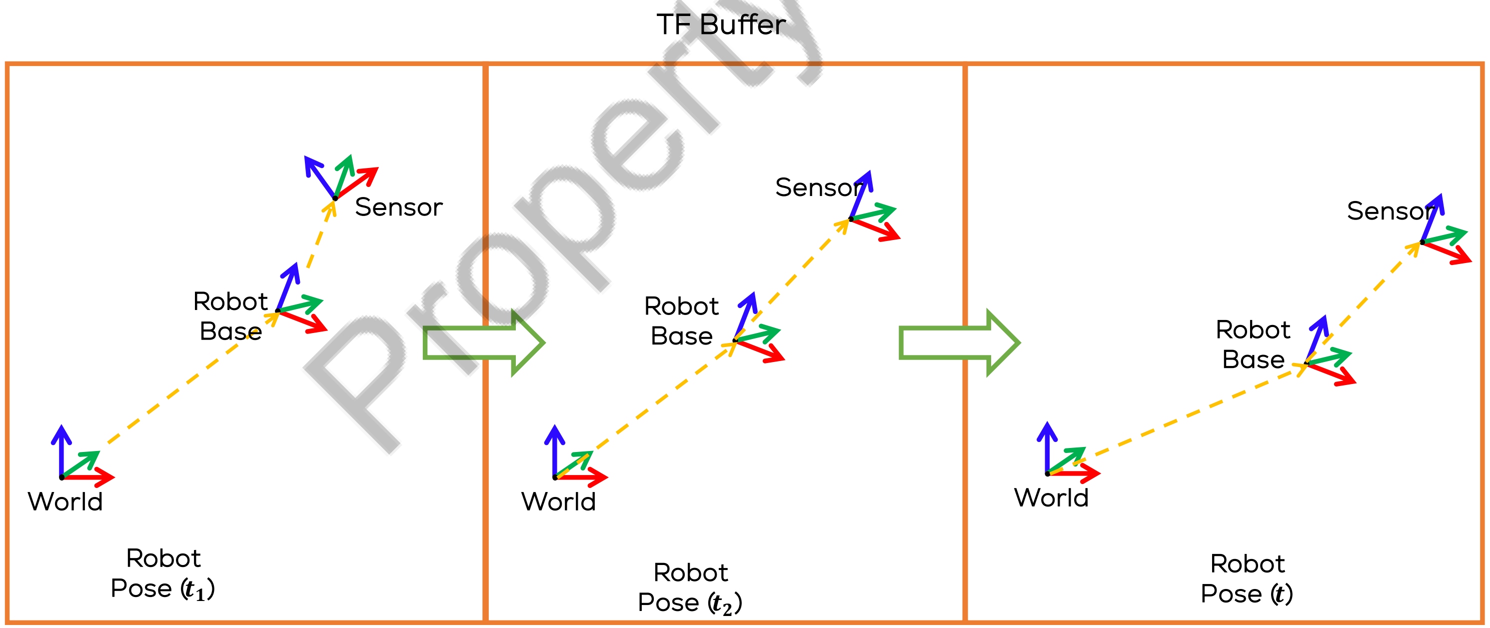
ROS TF2 Library

- As stated, ROS provides a simple way to listen to transformation using its *tf2* library.
- *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.



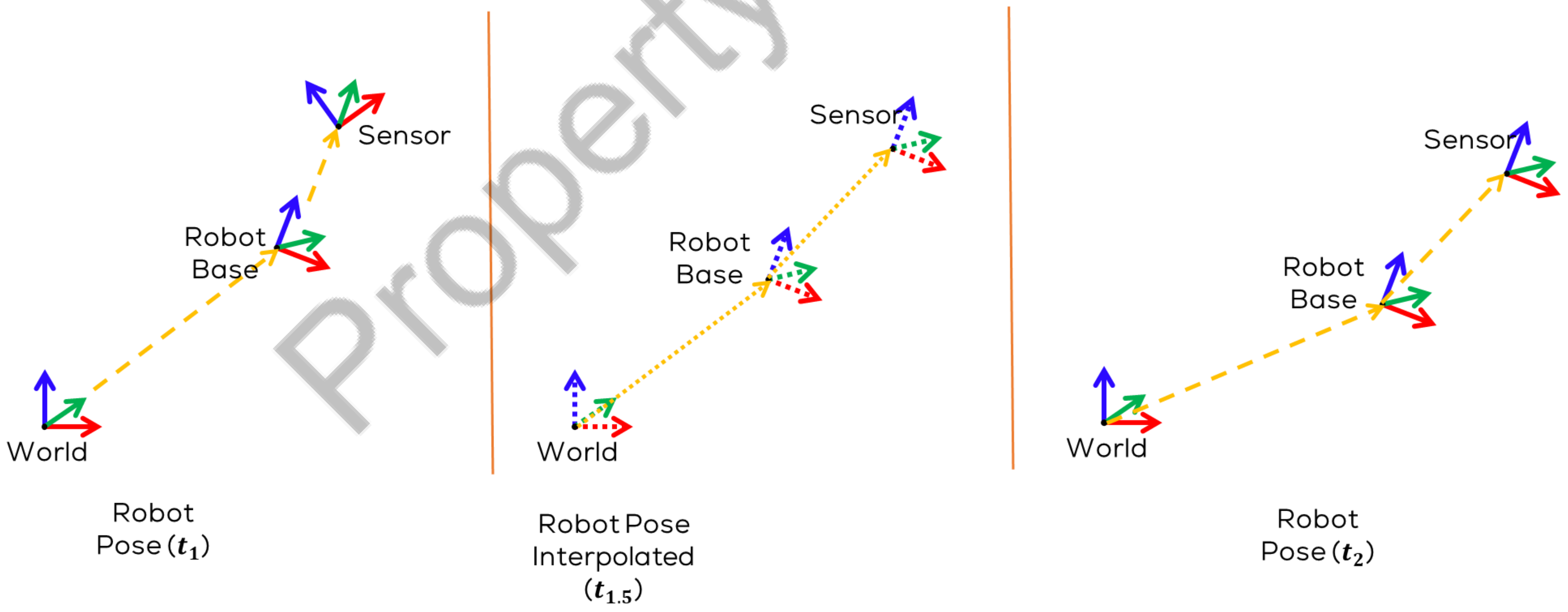
Coordinate transformations in ROS

- 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 4

TF Listener

{Learn, Create, Innovate};

Manchester Robotics

Activity 4

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

```
cd ~/catkin_ws/src/markers/scripts/
touch markers_listener.py
```

2. Make the file executable

```
sudo chmod +x markers_listener.py
```

3. Modify the CMake file to include the newly created node to the

```
catkin_install_python(PROGRAMS scripts/markers_listener.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

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

```
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)
```

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

```
try:
    trans = tfBuffer.lookup_transform('inertial_frame',
'planet', rospy.Time())
except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
            tf2_ros.ExtrapolationException):
    loop_rate.sleep()
    continue

print(trans)
```

Activity 4

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

```
cd ~/catkin_ws/src/markers/scripts/
touch markers_listener.py
```

2. Make the file executable

```
sudo chmod +x markers_listener.py
```

3. Modify the CMake file to include the newly created node to the

```
catkin_install_python(PROGRAMS scripts/markers_listener.py
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

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

```
tfBuffer = tf2_ros.Buffer()
listener = tf2_ros.TransformListener(tfBuffer)
```

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

```
try:
     trans = tfBuffer.lookup_transform('inertial_frame',
'planet', rospy.Time())
except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
            tf2_ros.ExtrapolationException):
     loop_rate.sleep()
     continue
print(trans)
```

Activity 4

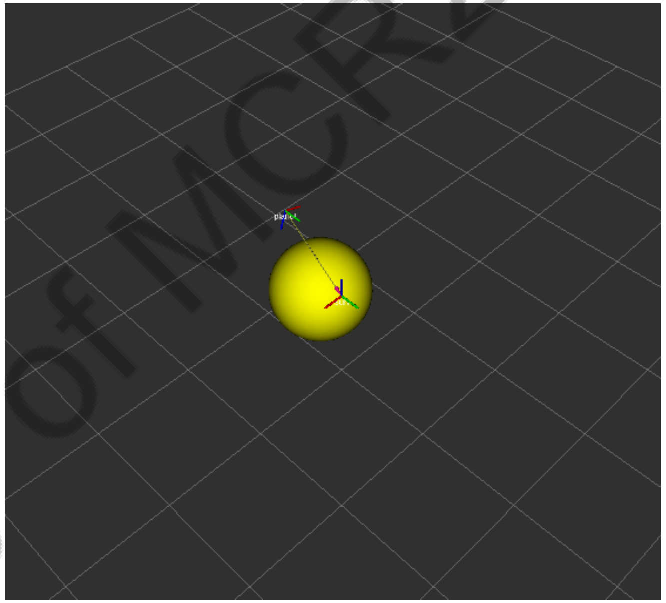
6. Modify the tf.launch file to include the "markers_listener"node

```
<?xml version="1.0" ?>
<launch>
    <node name="transforms" pkg="markers" type="tf_act.py" output="screen"/>
    <node name="sun" pkg="markers" type="marker.py" output="screen"/>
    <node name="tf_listener" pkg="markers" type="markers_listener.py" output="screen"/>
</launch>
```

7. Launch the file

```
roslaunch markers tf.launch
```

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



```
header:
  seq: 0
  stamp:
    secs: 1701432083
    nsecs: 995396614
  frame_id: "inertial_frame"
child_frame_id: "planet"
transform:
  translation:
    x: 0.9773711785823902
    y: -0.19978662121280966
    z: 1.0
  rotation:
    x: 0.0
    y: -0.0073377125649905965
    z: 0.0
    w: 0.9999730786247766
```

Activity 4: Getting the transformation matrix

1. Modify the markers_listener.py file, adding the following libraries

```
import tf_conversions
```

2. Add the code on the right to the "main loop"

3. Save and launch the node

```
[[ 0.84310642  0.          -0.53774675  0.84367718]
 [ 0.53774675  1.           0.84310642  2.18977442]
 [ 0.53774675  0.           0.84310642  1.        ]
 [ 0.          0.           0.          1.        ]]
header:
  seq: 0
  stamp:
    secs: 1701432149
    nsecs: 819315434
  frame_id: "inertial_frame"
child_frame_id: "planet"
transform:
  translation:
    x: 0.8436771800242078
    y: 2.1897744222981164
    z: 1.0
  rotation:
    x: 0.0
    y: 0.280083542081063
    z: 0.0
    w: -0.9599756296153176
```

```
translation = trans.transform.translation
            rotation = trans.transform.rotation

            translation_matrix = np.array([
                [1, 0, 0, translation.x],
                [0, 1, 1, translation.y],
                [0, 0, 1, translation.z],
                [0, 0, 0, 1]
            ])

            rotation_matrix =
tf_conversions.transformations.quaternion_matrix([
                rotation.x,
                rotation.y,
                rotation.z,
                rotation.w
            ])

            trans_matrix = np.dot(translation_matrix,
rotation_matrix)
```

Activity 4: Getting the pose of the marker

1. Modify the markers_listener.py file, adding the following libraries

```
from visualization_msgs.msg import Marker
from geometry_msgs.msg import PoseStamped
import tf2_geometry_msgs
```

2. Subscribe to the marker message

```
rospy.Subscriber("/sun", Marker, marker_callback)
```

3. Use the information of the marker message to a Pose Stamped message
   - Tf Library only transforms certain types of messages automatically, marker message is not included.

```
marker_msg = PoseStamped()
def marker_callback(msg):
    global marker_msg
    marker_msg.header = msg.header
    marker_msg.pose = msg.pose
```

4. Transform the Pose Stamped message by adding this to the main loop.

```
marker_transformed =
tf2_geometry_msgs.do_transform_pose(marker_msg,trans)

print(marker_transformed)
```

5. Save and launch the node.

```
header:
  seq: 0
  stamp:
    secs: 1701432149
    nsecs: 719290972
  frame_id: "inertial_frame"
pose:
  position:
    x: 0.6573808214947504
    y: 2.6682119938920685
    z: 1.703805731308754
  orientation:
    x: 0.0
    y: -0.4268724009016546
    z: 0.0
    w: 0.90431186730489
```

Thank you

{Learn, Create, Innovate};

MCR2 Manchester Robotics

T&C

Terms and conditions

{Learn, Create, Innovate};

Manchester Robotics

Terms and conditions

- *THE PIECES, IMAGES, VIDEOS, DOCUMENTATION, ETC. SHOWN HERE ARE FOR INFORMATIVE PURPOSES ONLY. THE DESIGN IS PROPRIETARY AND CONFIDENTIAL TO MANCHESTER ROBOTICS LTD. (MCR2). THE INFORMATION, CODE, SIMULATORS, DRAWINGS, VIDEOS PRESENTATIONS ETC. CONTAINED IN THIS PRESENTATION IS THE SOLE PROPERTY OF MANCHESTER ROBOTICS LTD. ANY REPRODUCTION, RESELL, REDISTRIBUTION OR USAGE IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF MANCHESTER ROBOTICS LTD. IS STRICTLY PROHIBITED.*
- *THIS PRESENTATION MAY CONTAIN LINKS TO OTHER WEBSITES OR CONTENT BELONGING TO OR ORIGINATING FROM THIRD PARTIES OR LINKS TO WEBSITES AND FEATURES IN BANNERS OR OTHER ADVERTISING. SUCH EXTERNAL LINKS ARE NOT INVESTIGATED, MONITORED, OR CHECKED FOR ACCURACY, ADEQUACY, VALIDITY, RELIABILITY, AVAILABILITY OR COMPLETENESS BY US.*
- *WE DO NOT WARRANT, ENDORSE, GUARANTEE, OR ASSUME RESPONSIBILITY FOR THE ACCURACY OR RELIABILITY OF ANY INFORMATION OFFERED BY THIRD-PARTY WEBSITES LINKED THROUGH THE SITE OR ANY WEBSITE OR FEATURE LINKED IN ANY BANNER OR OTHER ADVERTISING.*