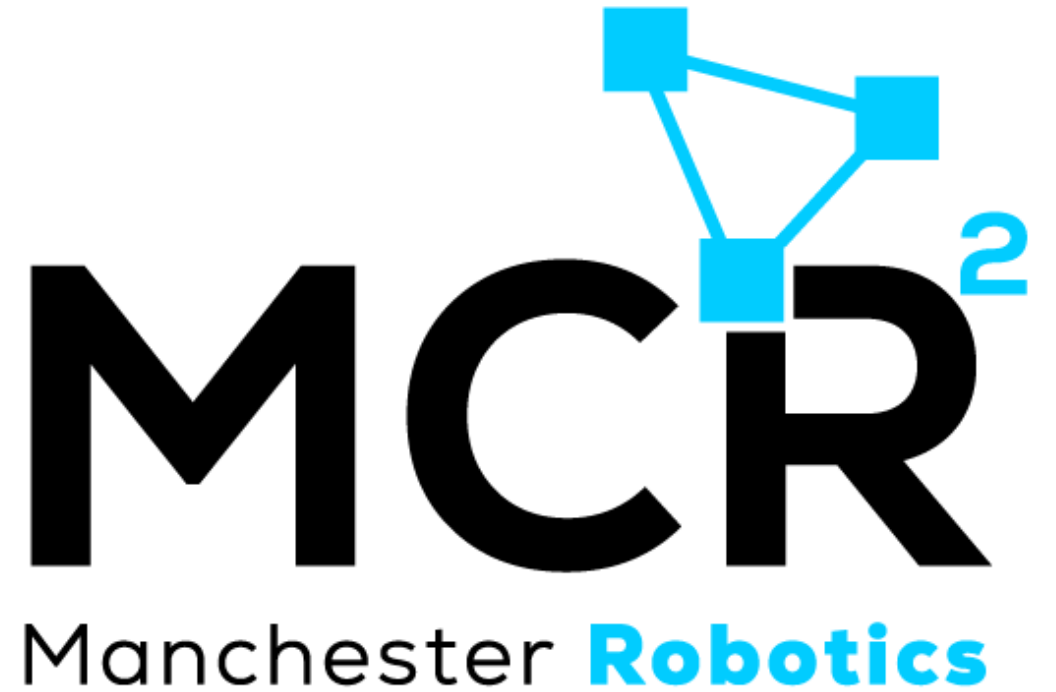


{Learn, Create, Innovate};

ROS

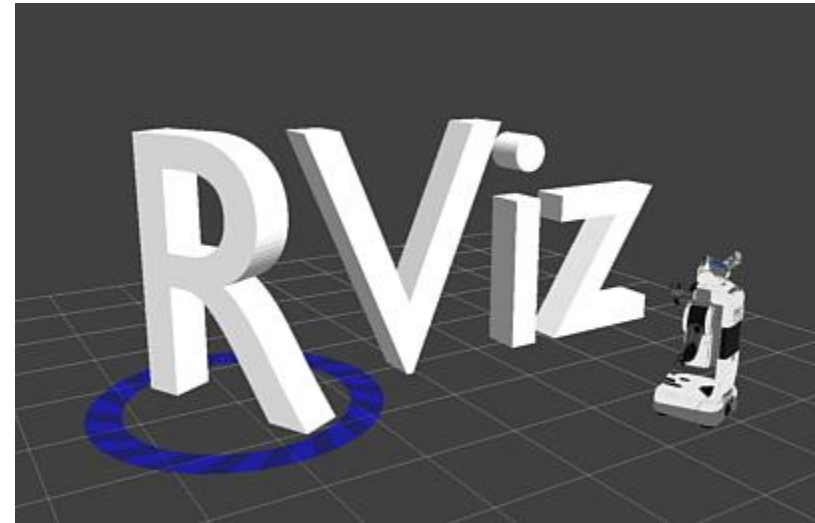
Visualisation Tools



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.”

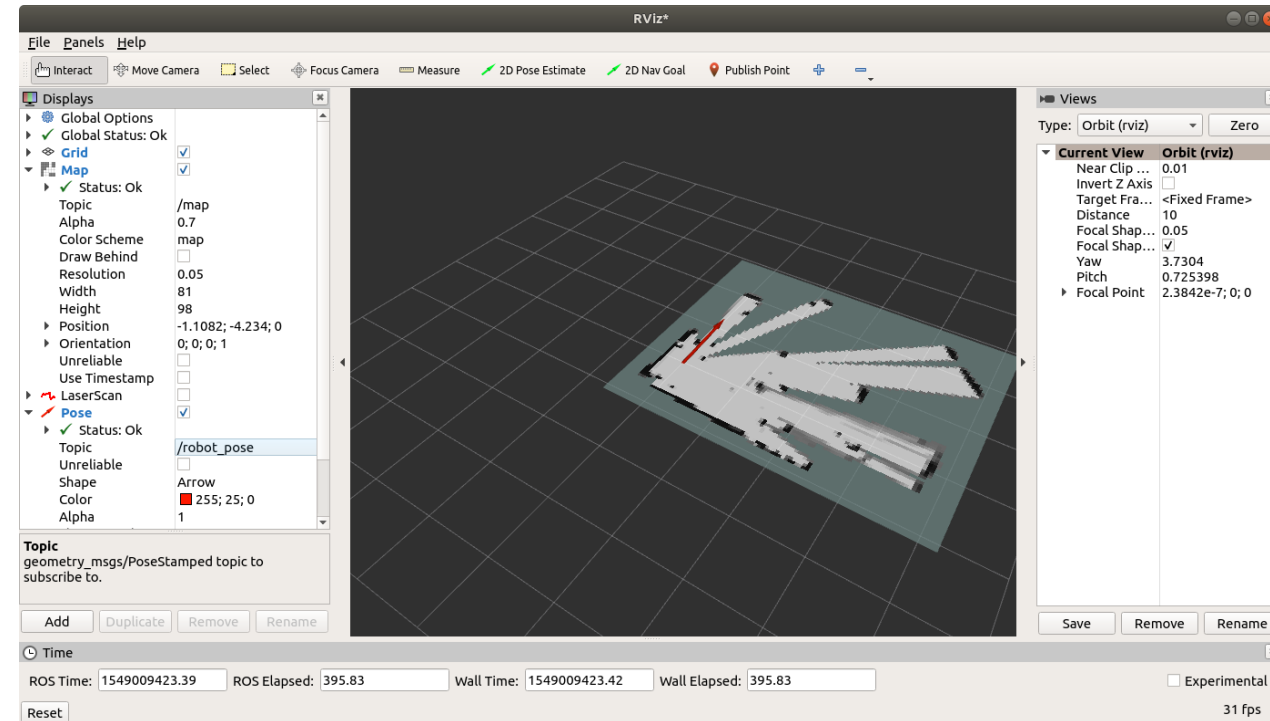




RVIZ



- 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-<$DISTR0>-usb_cam
```

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

```
roscore
```

- Run the camera driver

```
roslaunch 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

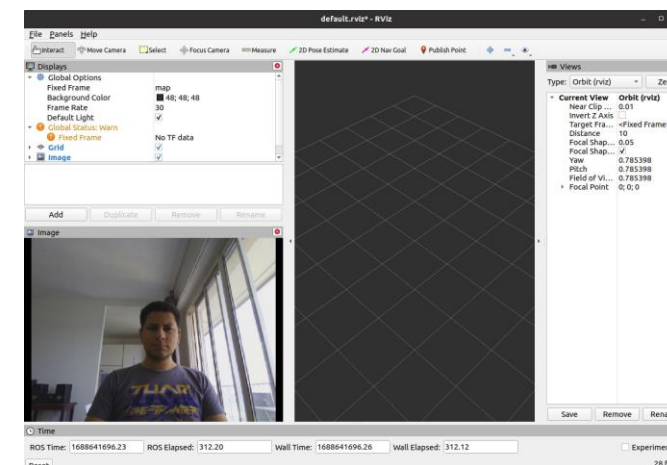
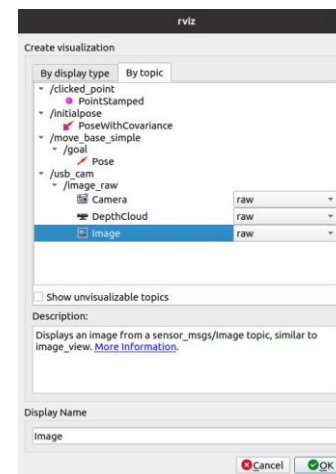
- Check that the topics are being published

```
rostopic list
```

- Start RVIZ

```
roslaunch rviz rviz
```

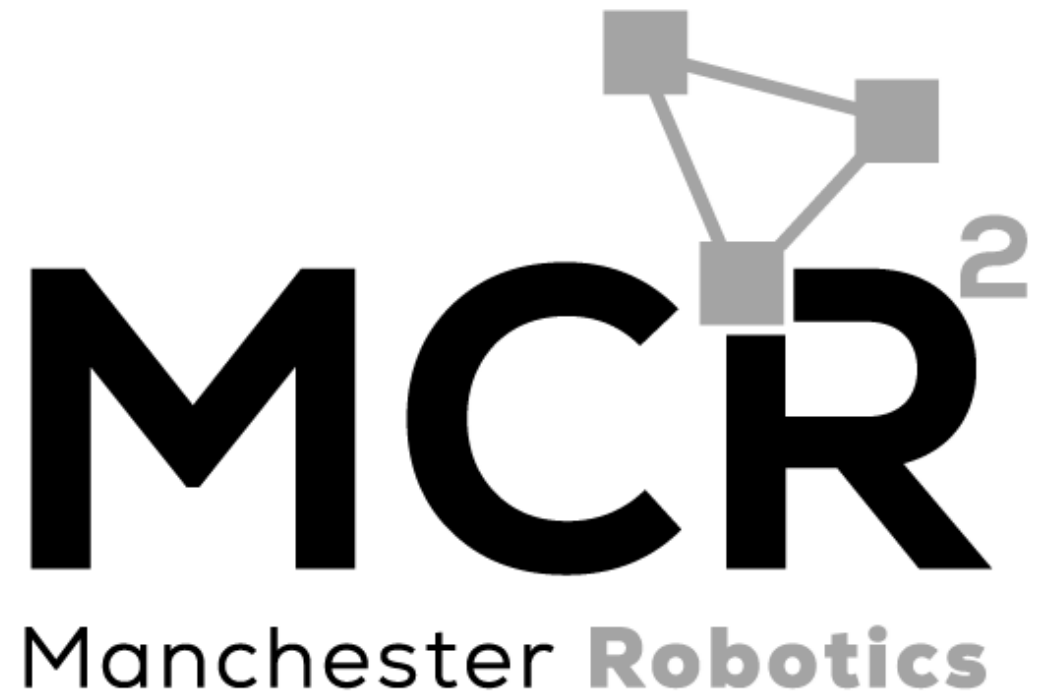
- Press the “add” button
- Go to the tab “By topic”
- Add the topic Camera, located under the topics `/usb_cam -> /image_raw`



RVIZ

Markers

{Learn, Create, Innovate};



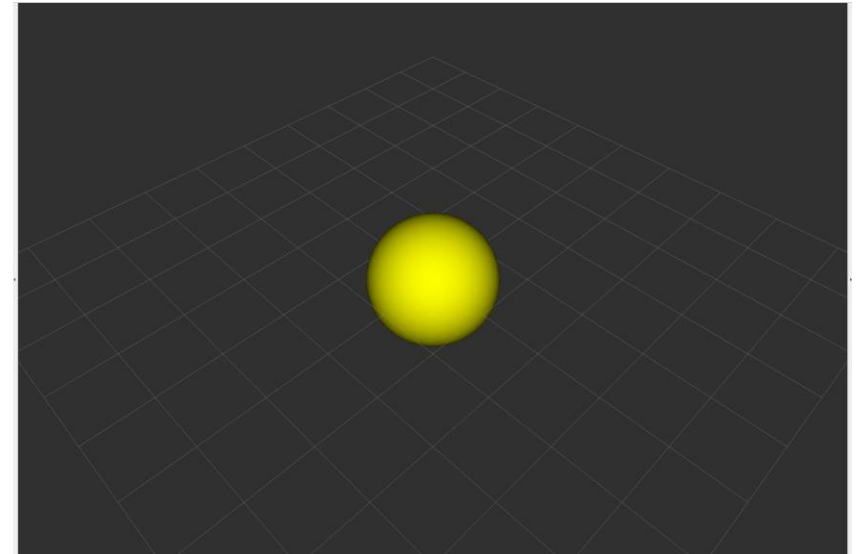


Markers



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.



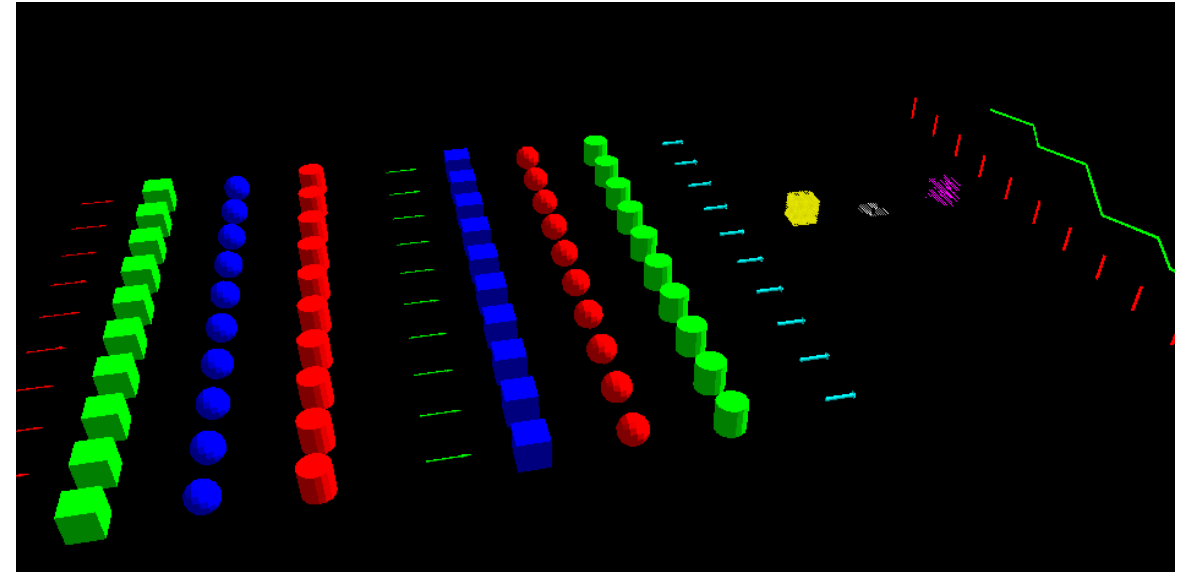


Markers



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
```

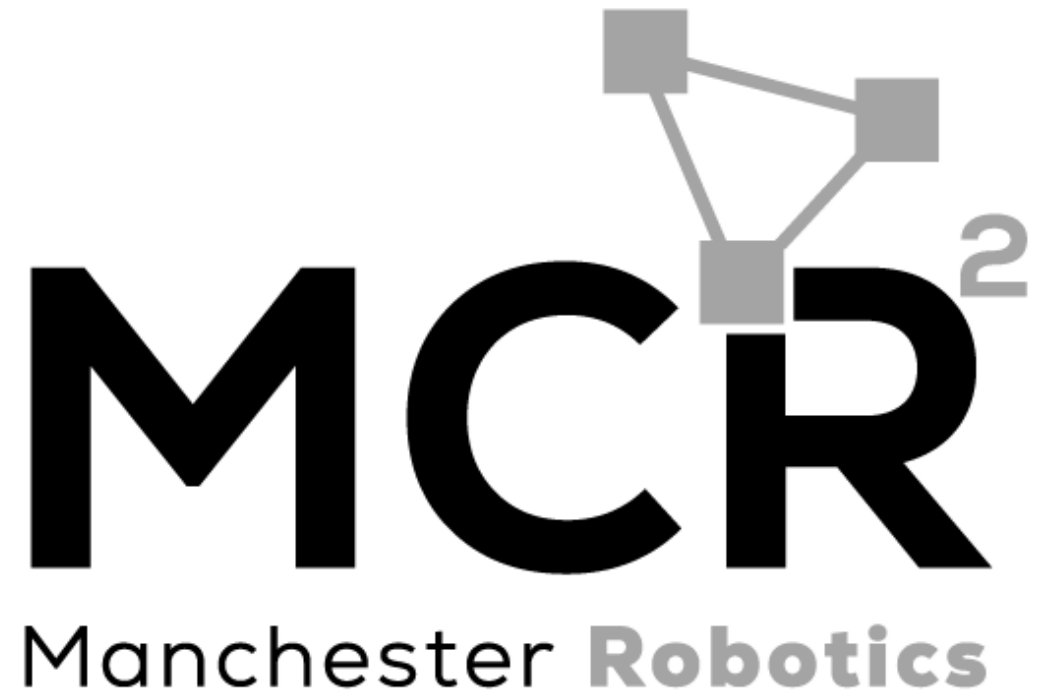
```
# NOTE: only used for text markers
string text
```

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


Activity 1

Markers

{Learn, Create, Innovate};





Marker Generator



- Make a new package, 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
```

- 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, its publisher and publish the marker.

```
cd ~/catkin_ws
```

```
catkin_make
```

- Start ROS

```
roscore
```

- Run the node

```
roslaunch markers marker.py
```

- Start RViz

```
roslaunch rviz rviz
```

- Add the marker
 - Press Add
 - >>By topic>>/sun>> marker

Marker Message Example

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

```
#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
```

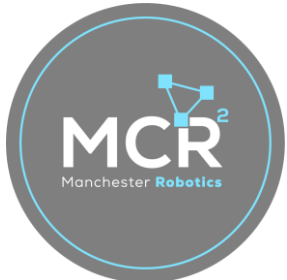
```
sun.color.g = 1.0
```

```
sun.color.b = 0.0
```

```
sun.color.a = 1.0
```

```
#Set Duration
```

```
sun.lifetime = rospy.Duration(0)
```



Marker Generator



```
#!/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)
```

```
#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
```

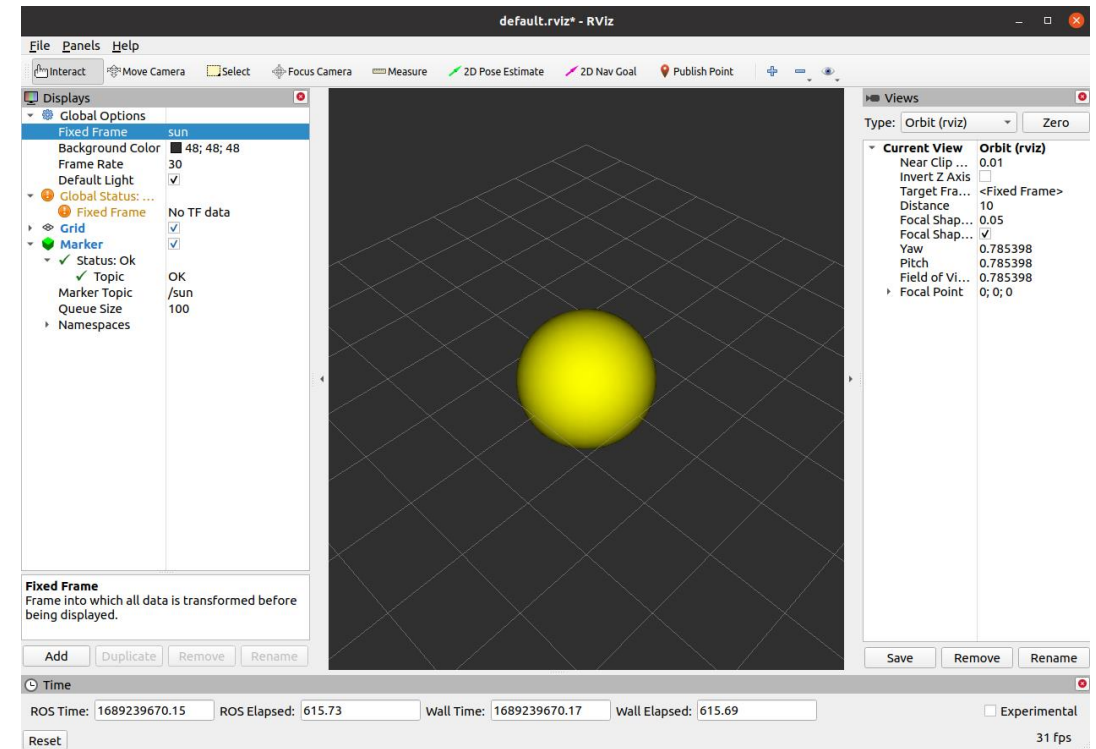
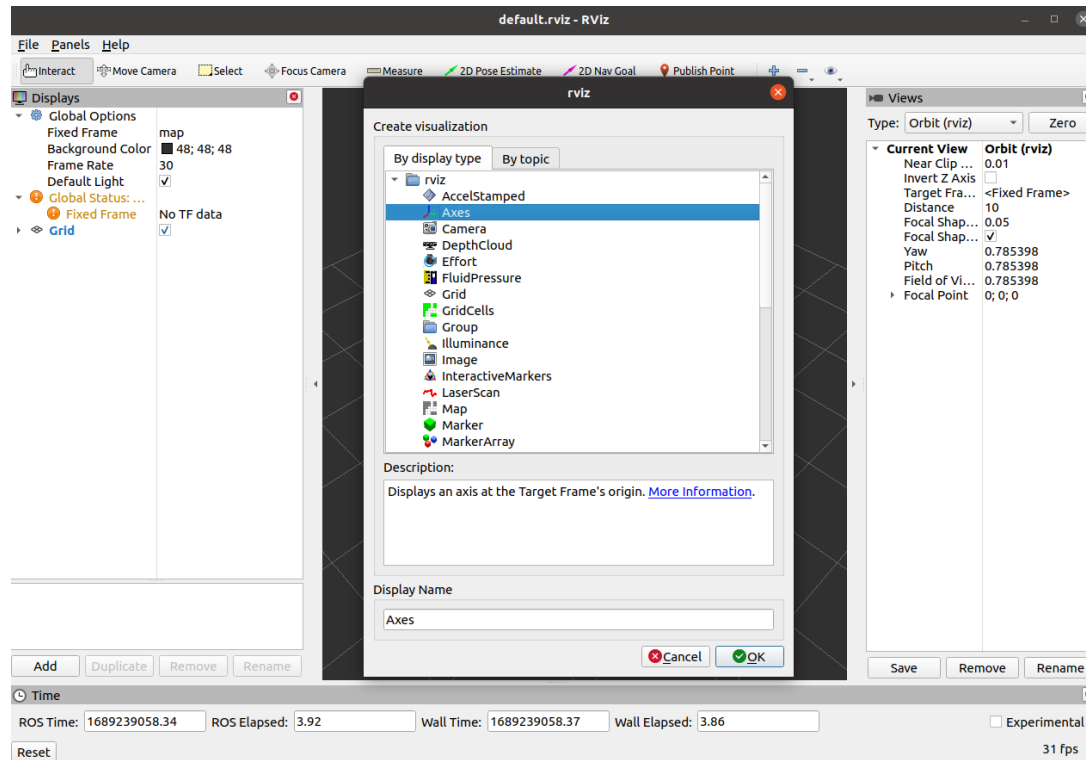
Manchester Robotics



Marker Generator



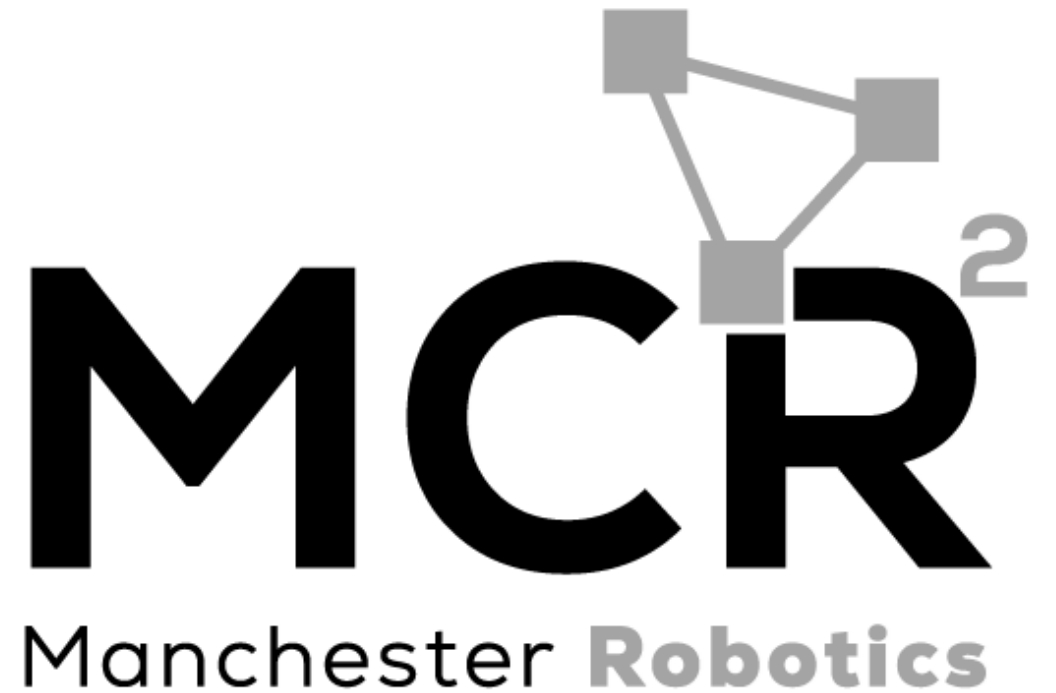
- Change the fixed frame on top of RViz to “sun”



ROS Transformations

TF

{Learn, Create, Innovate};

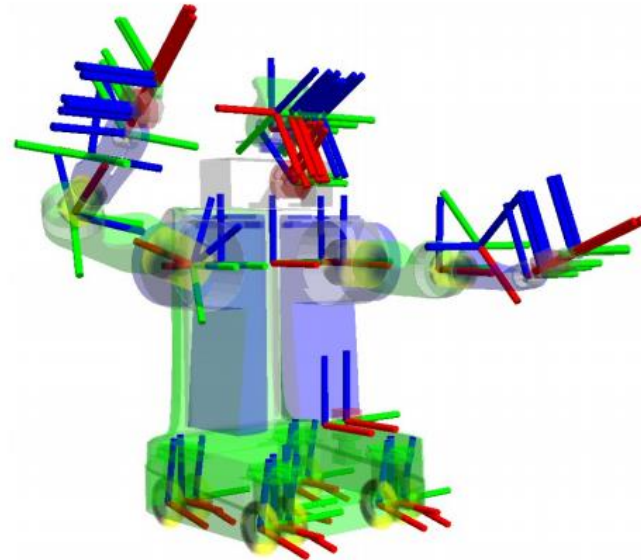




Coordinate transformations in ROS



- 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.

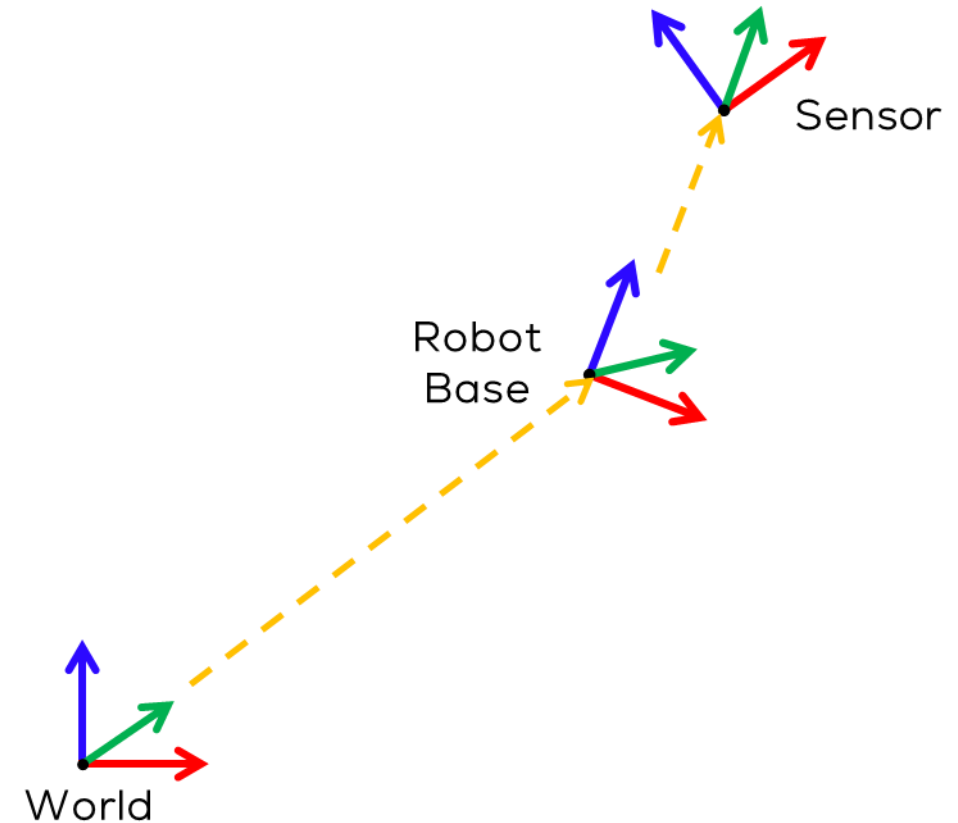




Coordinate transformations in ROS



- 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.

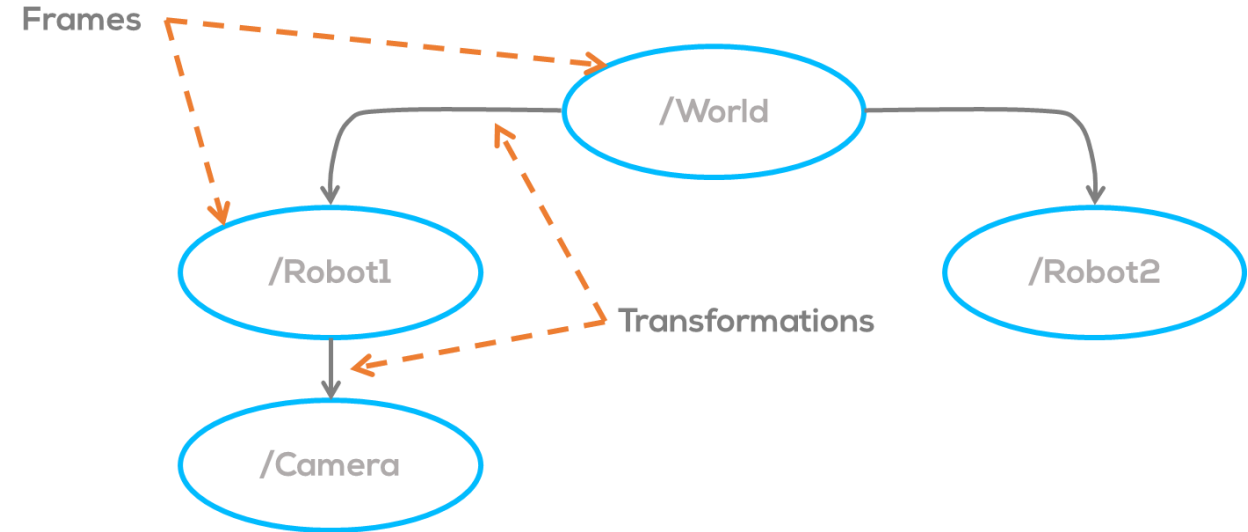




Coordinate transformations in ROS



- 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.



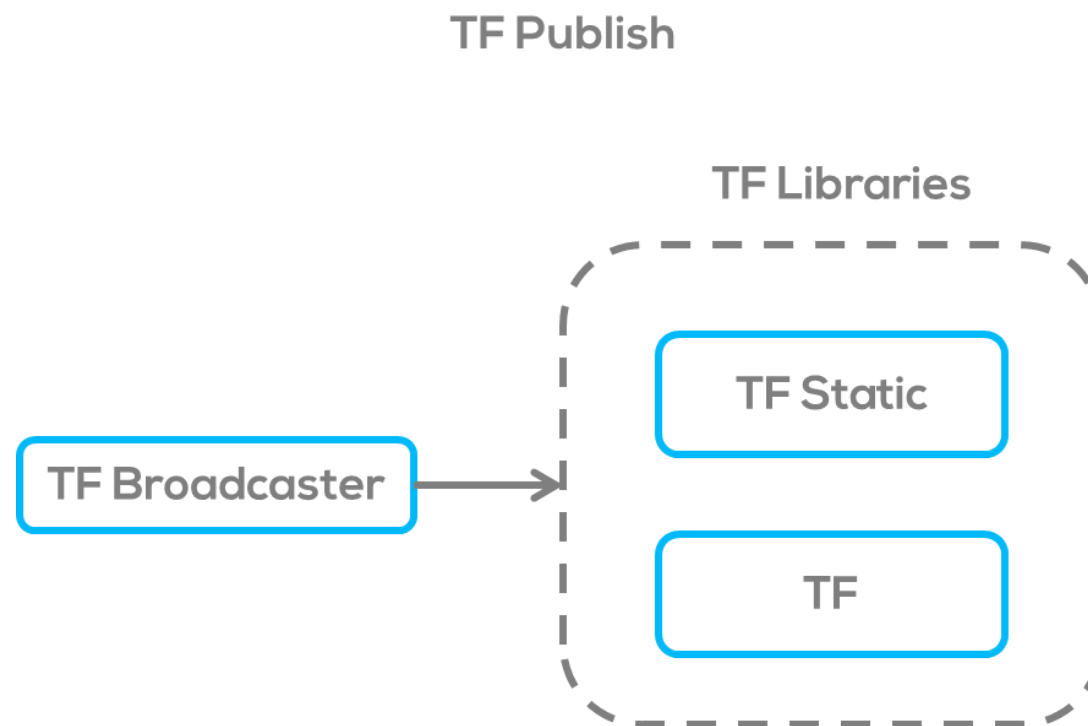


Coordinate transformations in ROS



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.





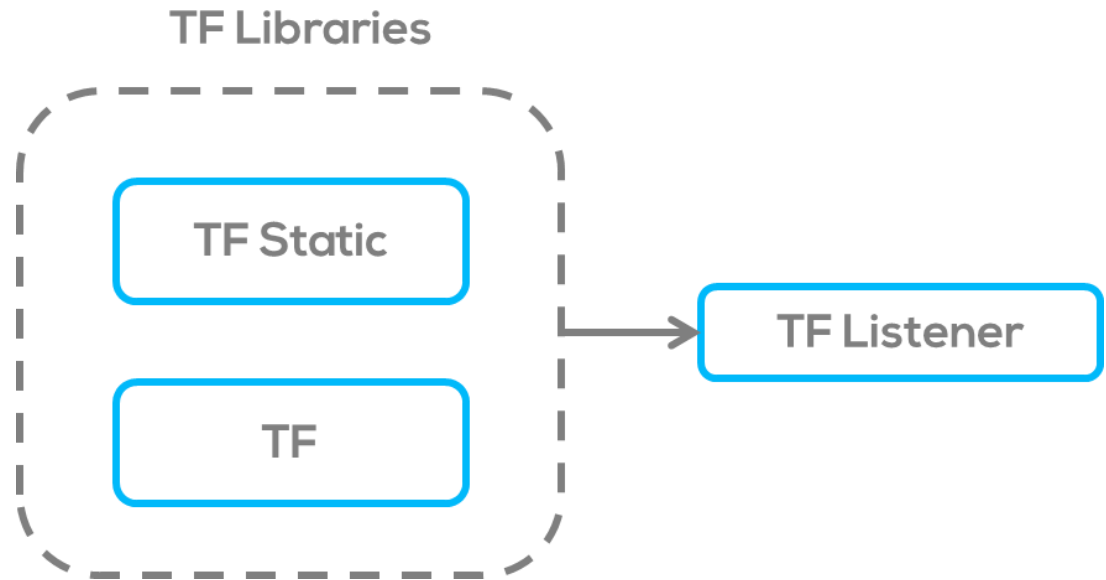
Coordinate transformations in ROS



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

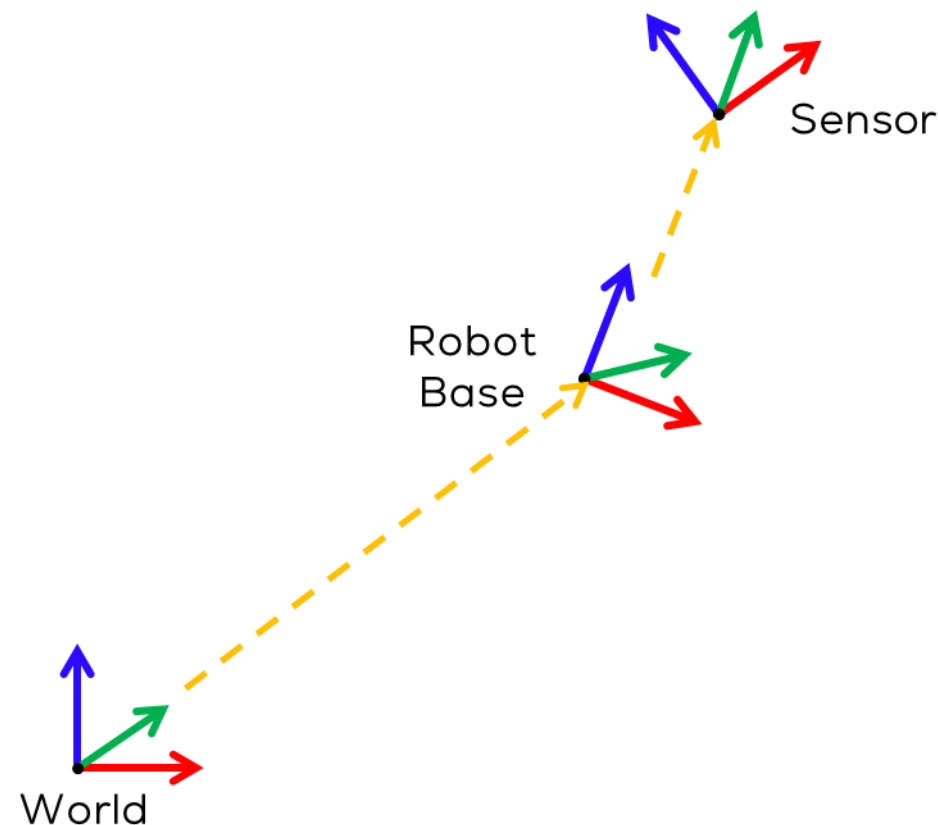




How it works?



- **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.

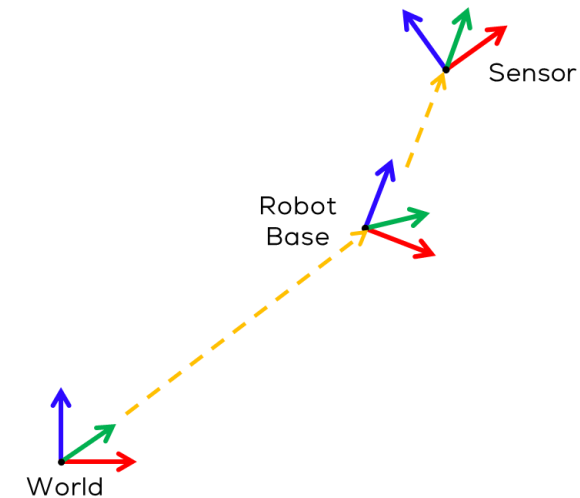




How it works?



- **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.



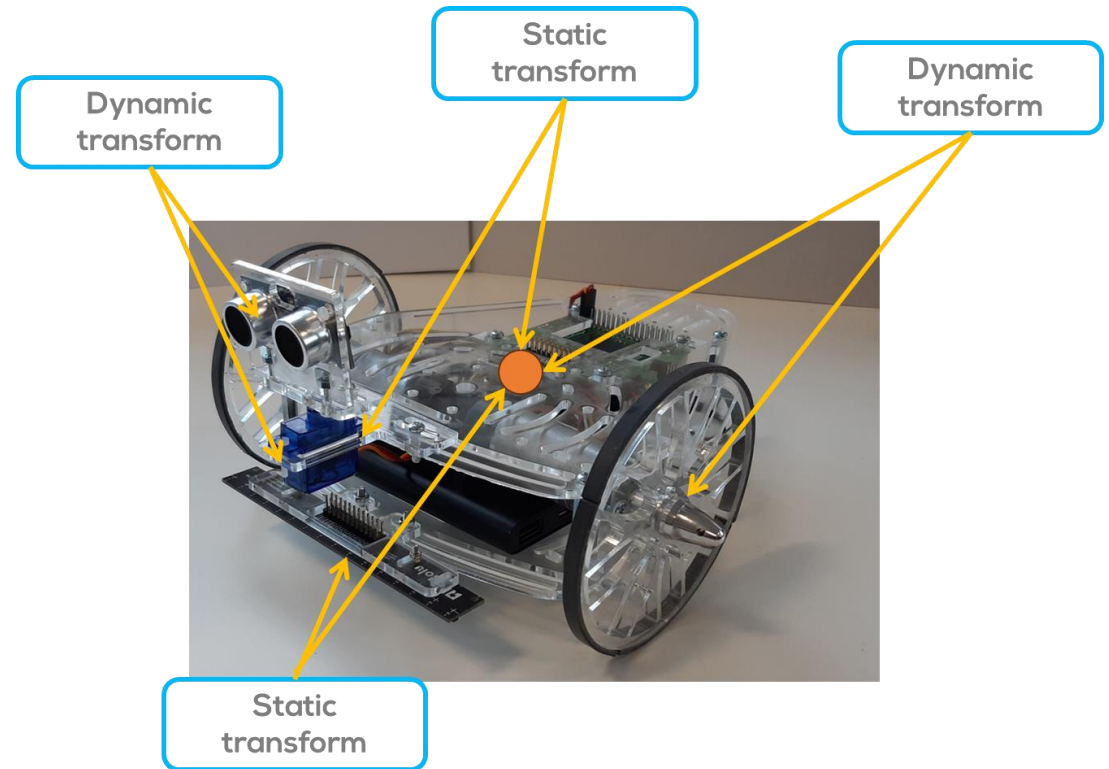
How it works?

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

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 world  
link1"/>  
</launch>
```

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

```
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

```
roslaunch 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

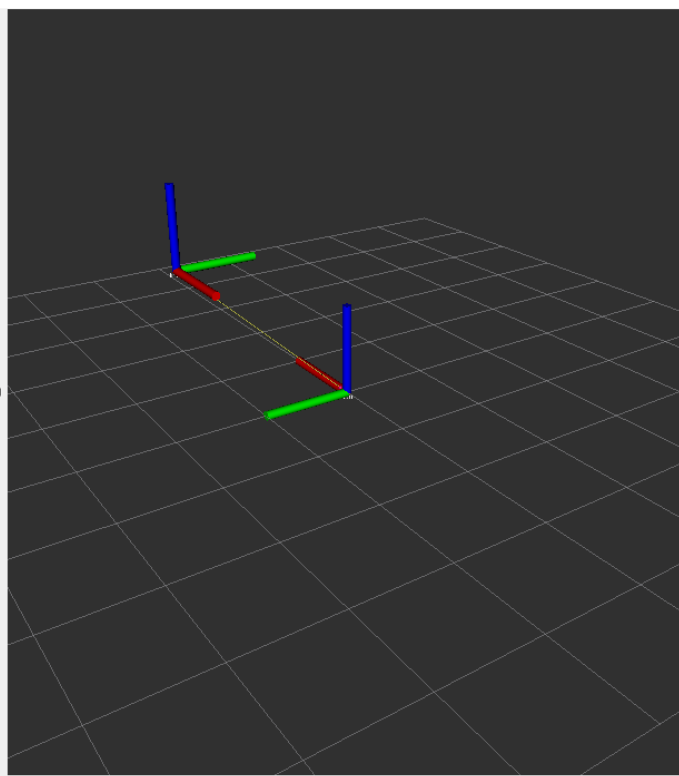


Displays

- Global Options
 - Fixed Frame: sun
 - Background Color: 48; 48; 48
 - Frame Rate: 30
 - Default Light: ☒
- Global Status: Ok
 - Fixed Frame: OK
- Grid
 - ☒
- Axes
 - ☒
- Status: Ok
 - Reference Frame: link1
 - Length: 1
 - Radius: 0.1
 - Show Trail: ☐
 - Alpha: 1
- Marker
 - ☐
- TF
 - ☒

Reference Frame
The TF frame these axes will use for their origin.

Add Duplicate Remove Rename

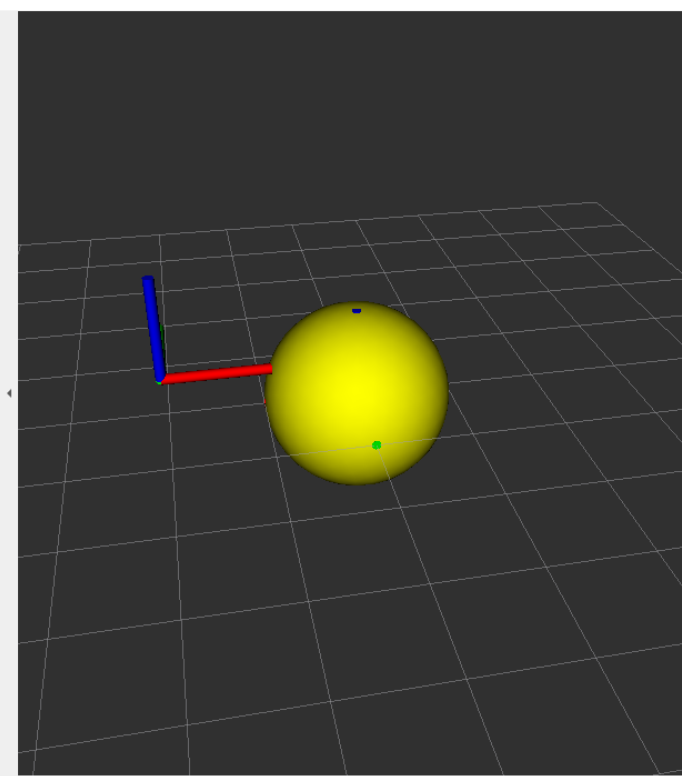


Displays

- Global Options
 - Fixed Frame: sun
 - Background Color: 48; 48; 48
 - Frame Rate: 30
 - Default Light: ☒
- Global Status: Ok
 - Fixed Frame: OK
- Grid
 - ☒
- Axes
 - ☒
- Status: Ok
 - Reference Frame: link1
 - Length: 1
 - Radius: 0.1
 - Show Trail: ☐
 - Alpha: 1
- Marker
 - ☒
- TF
 - ☒

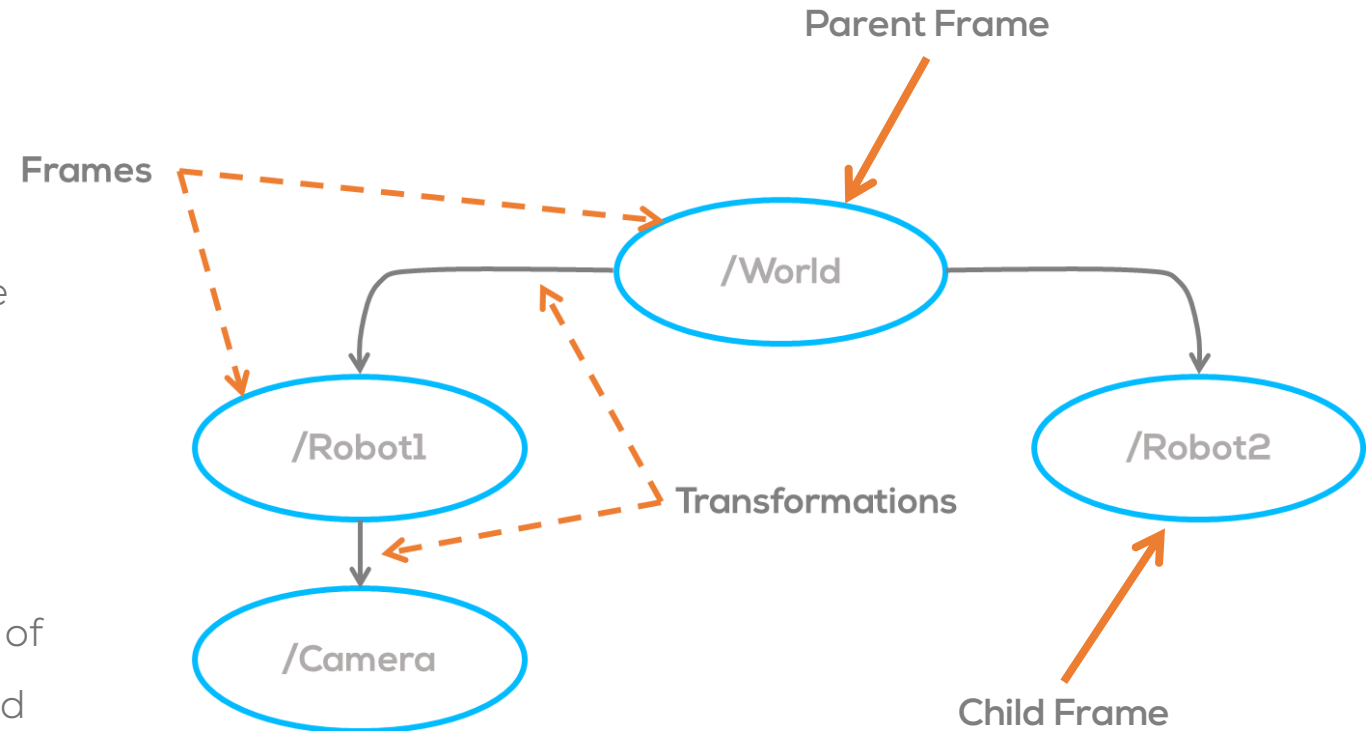
Reference Frame
The TF frame these axes will use for their origin.

Add Duplicate Remove Rename



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.

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
```

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



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
- 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.

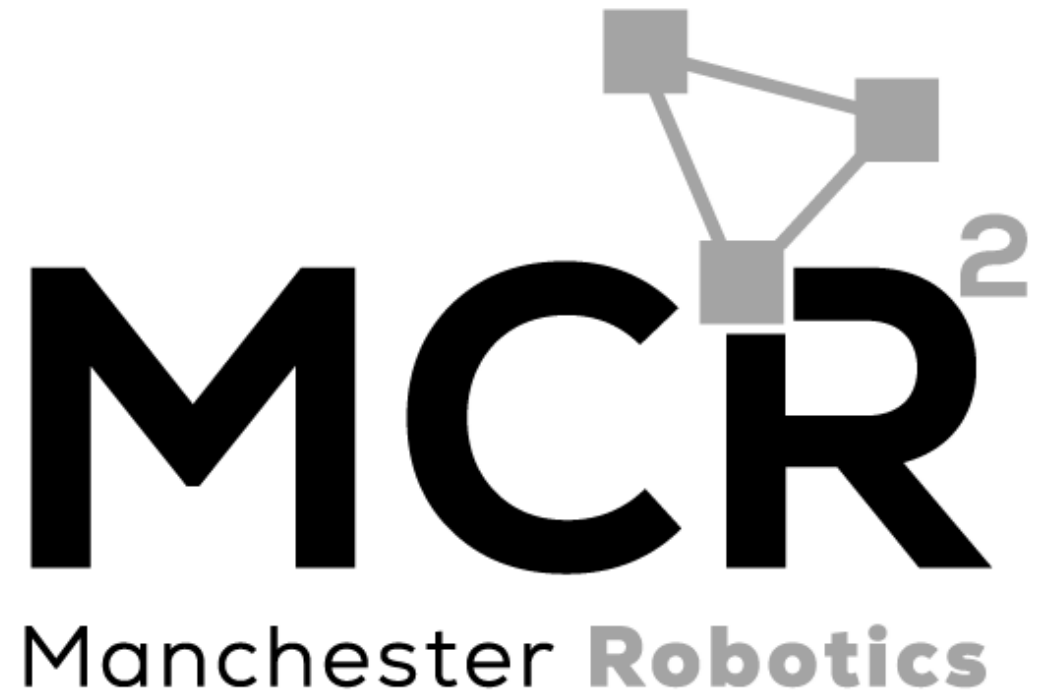
```
staticbc_ex = StaticTransformBroadcaster()  
dynamicbc_ex = TransformBroadcaster()
```

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

Activity 2

Transformations

{Learn, Create, Innovate};





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
- Run the node

```
roscore

roslaunch markers tf_act.py
```

- Start RViz

```
roslaunch rviz rviz
```

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

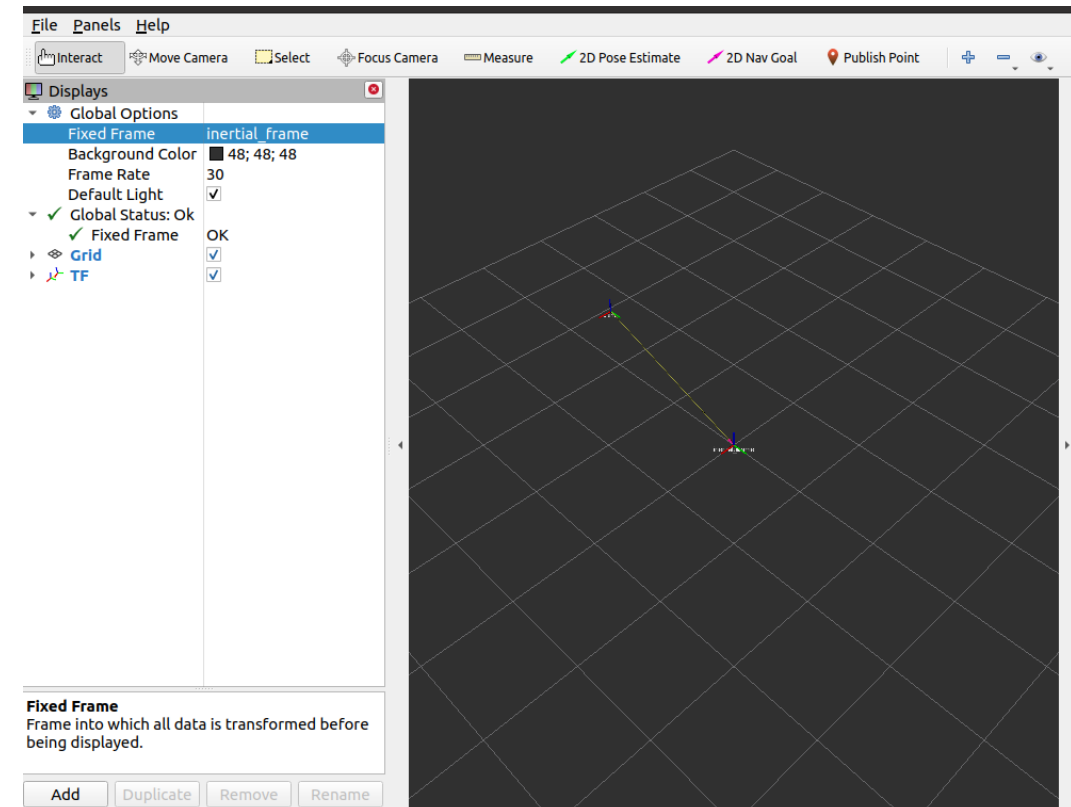
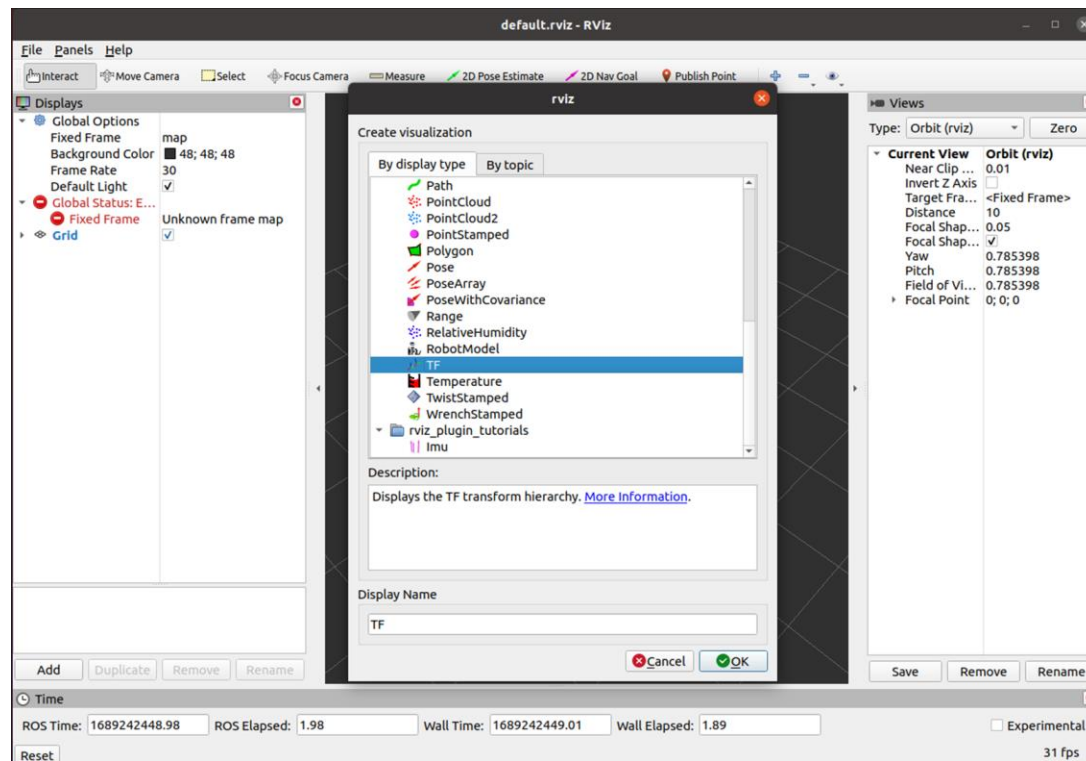
```
ex_tf = TransformStamped()
ex_tf.header.frame_id = "inertial_frame"
ex_tf.child_frame_id = "ex"
ex_tf.header.stamp = rospy.Time.now()
ex_tf.transform.translation.x = 1
ex_tf.transform.translation.y = 1
ex_tf.transform.translation.z = 1.0
ex_tf.transform.rotation.x = 0
ex_tf.transform.rotation.y =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



Activity 2



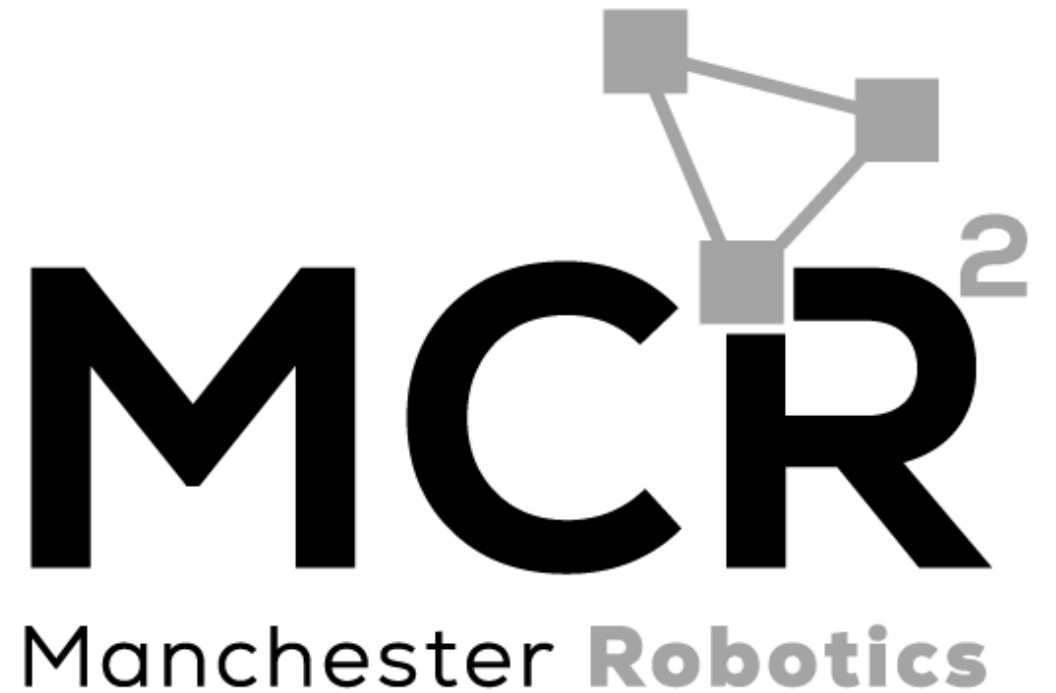
- Change the fixed frame on top of RViz to *"inertial_frame"*



Activity 3

Planets

{Learn, Create, Innovate};





Multiple Marker Generator



- In this activity the knowledge acquired in the previous two activities will be used to create a series of planets orbiting a sun.
- In the package *“markers”* create a new node called *“markers.py”*

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

- Give executable permission to the file

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

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

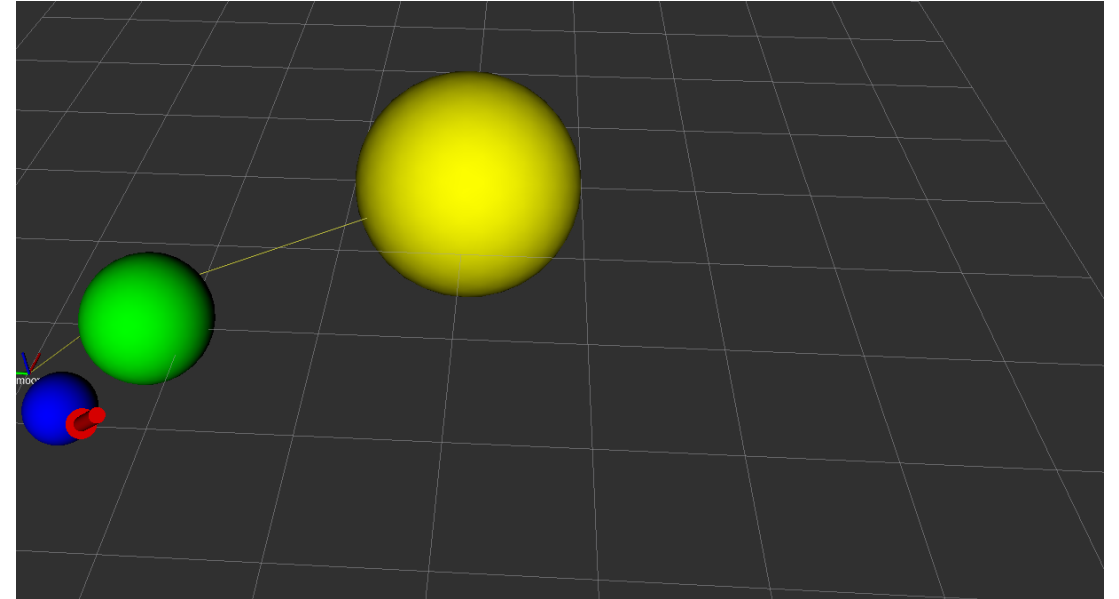
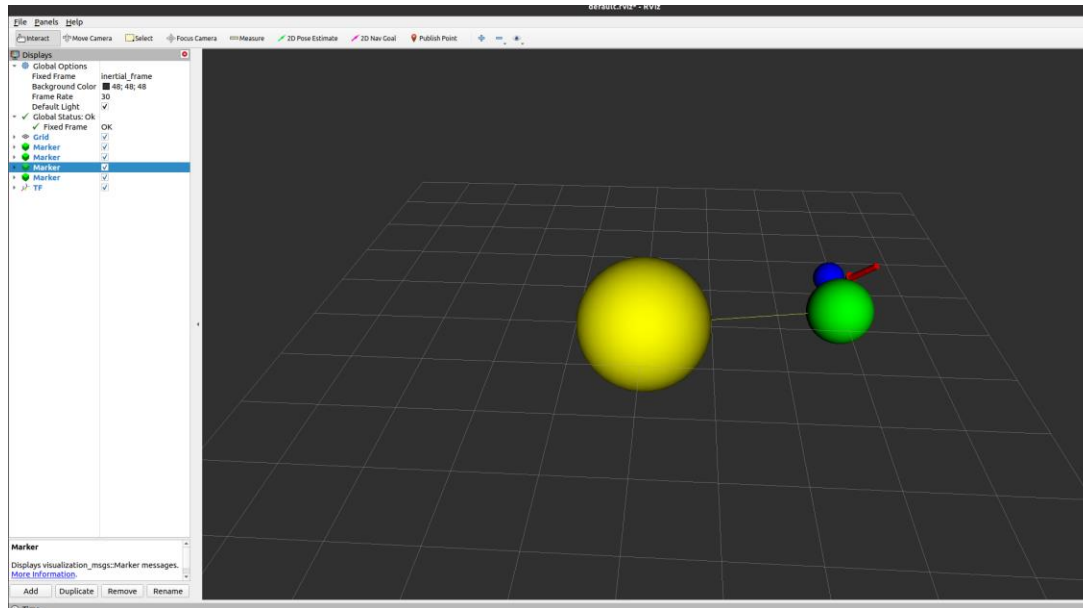
```
catkin_install_python(PROGRAMS scripts/marker.py scripts/markers.py  
DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

- 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.



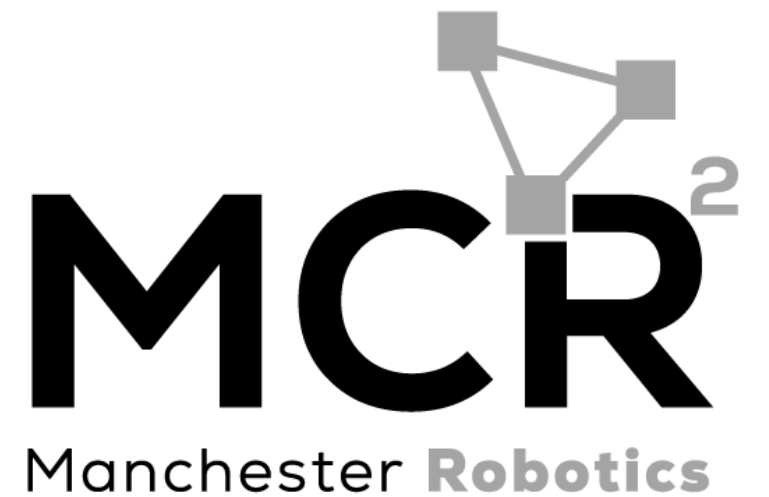


Result



Thank you

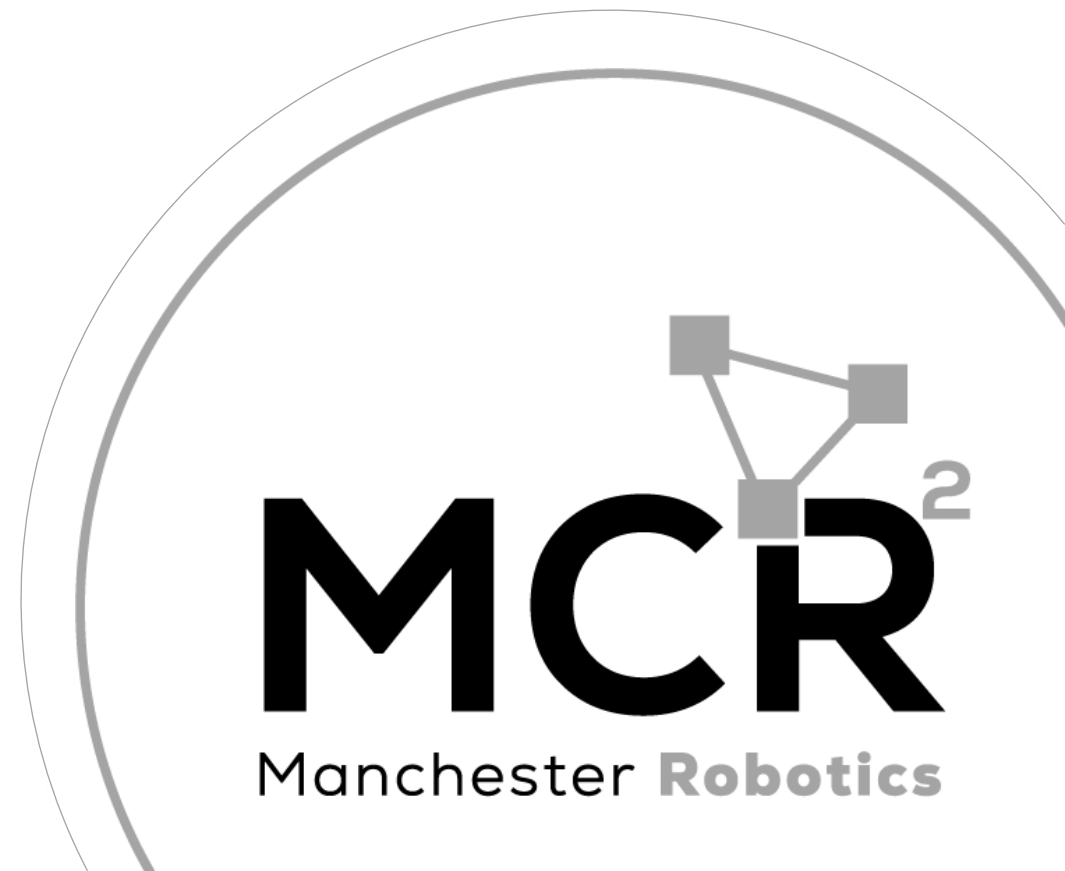
{Learn, Create, Innovate};



T&C

Terms and conditions

{Learn, Create, Innovate};





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.*