ROS Concepts 3

ECE 495/595 Lecture Slides

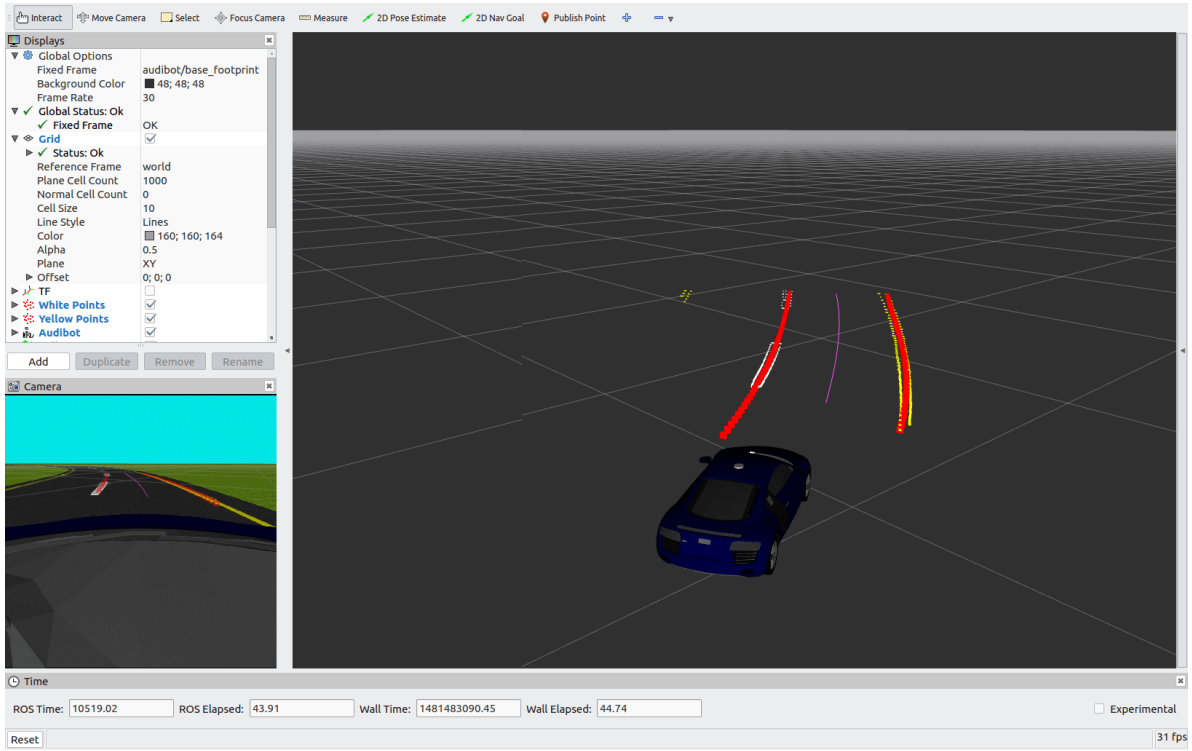
Winter 2017

Instructor: Micho Radovnikovich

Summary and Quick Links

These slides contain the following concepts:

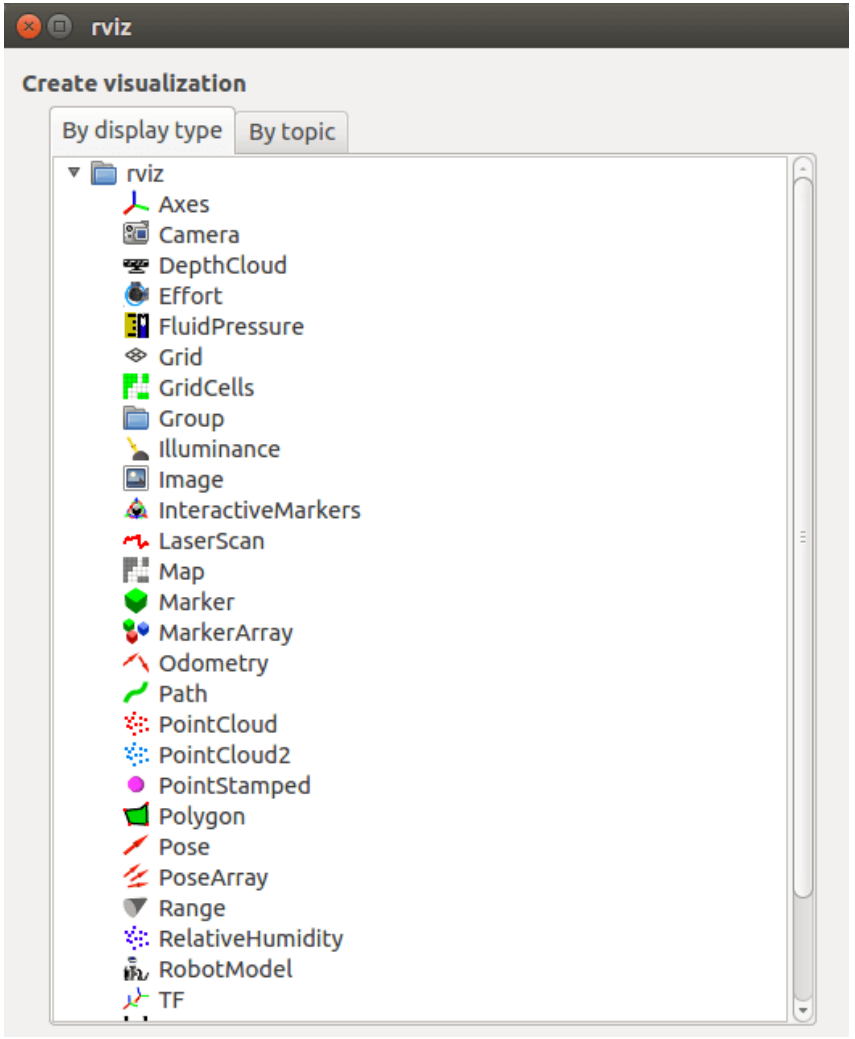
Rviz



- Rviz is a visualization tool that allows a user to easily display information in 3-D.
- Contains many built-in display types designed for a variety of data types.

Rviz

- Some common display types include:
  - *Marker* – A shape positioned at a given point and orientation. Markers can be combined in a *MarkerArray*.
  - *Grid* – Displays a grid plane.
  - *LaserScan* – Displays the raw laser points as sensed by a LIDAR.



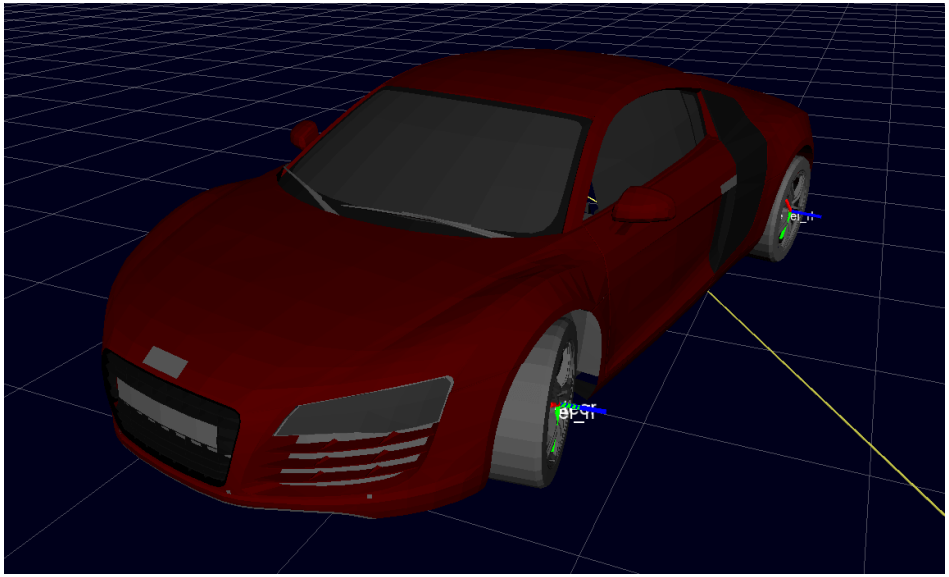
Rviz

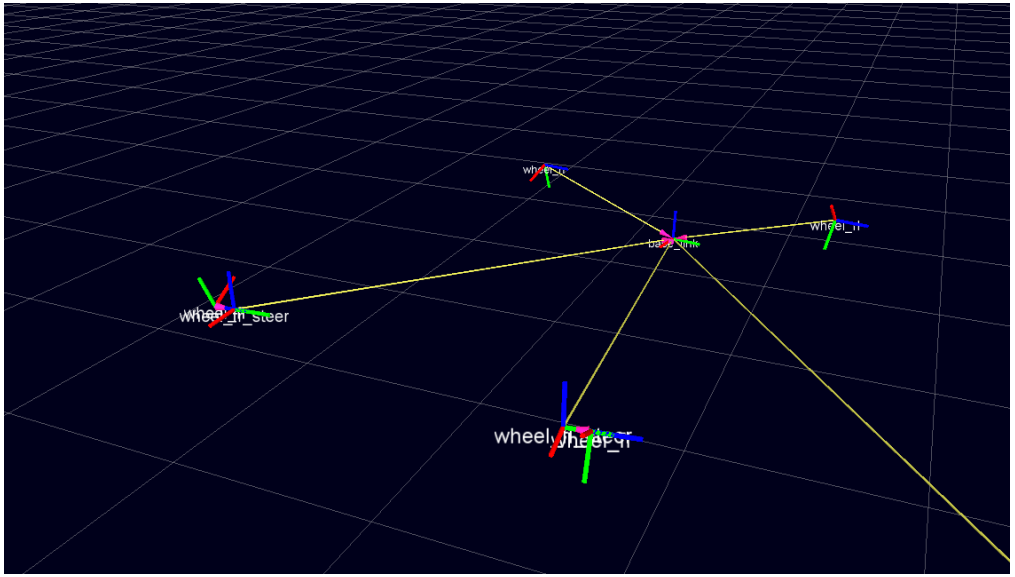
- More common display types:
  - *Path* – Illustrates a series of points as a thin line.
  - *PointCloud* – A collection of points in 3-D.
  - *Map* – Visualizes a robot's map of its environment.
  - *InteractiveMarkers* – Like regular markers, but can be manipulated by the user in the Rviz interface and provide feedback to a ROS node.
  - *TF* – Visualizes the various coordinate frames and their relationships to each other.

TF Frames

- ROS manages reference frames using a package called *tf*.
- Except for the root frame, a transformation is specified for each frame relative to its parent.
- *tf* maintains the tree structure and provides classes and functions to:
  - Update the transformation between a child and its parent.
  - Look up the transformation between any two frames in the tree.
  - Transform points from one frame to another in the same tree.

TF Frames





Static Transform Publishers

- For reference frames that don't move relative to their parent frames, a static transform publisher is used.

- A static transform publisher can be started from the command line in two ways:
  - Orientation given by a quaternion:

```
rosrun tf static_transform_publisher x y z qx qy qz qw
                  frame_id child_frame_id period_in_ms
```

  - Orientation given by roll-pitch-yaw

```
rosrun tf static_transform_publisher x y z yaw pitch roll
                  frame_id child_frame_id period_in_ms
```

Static Transform Publishers

- The number of arguments determines which method is used.

- Here are two equivalent examples of a fixed offset of 2 meters in $\underline{y}$ between *<u>/frame1</u>* and *<u>/frame2</u>*:
  - Using a quaternion ($w = 1,\ x = 0,\ y = 0,\ z = 0$):

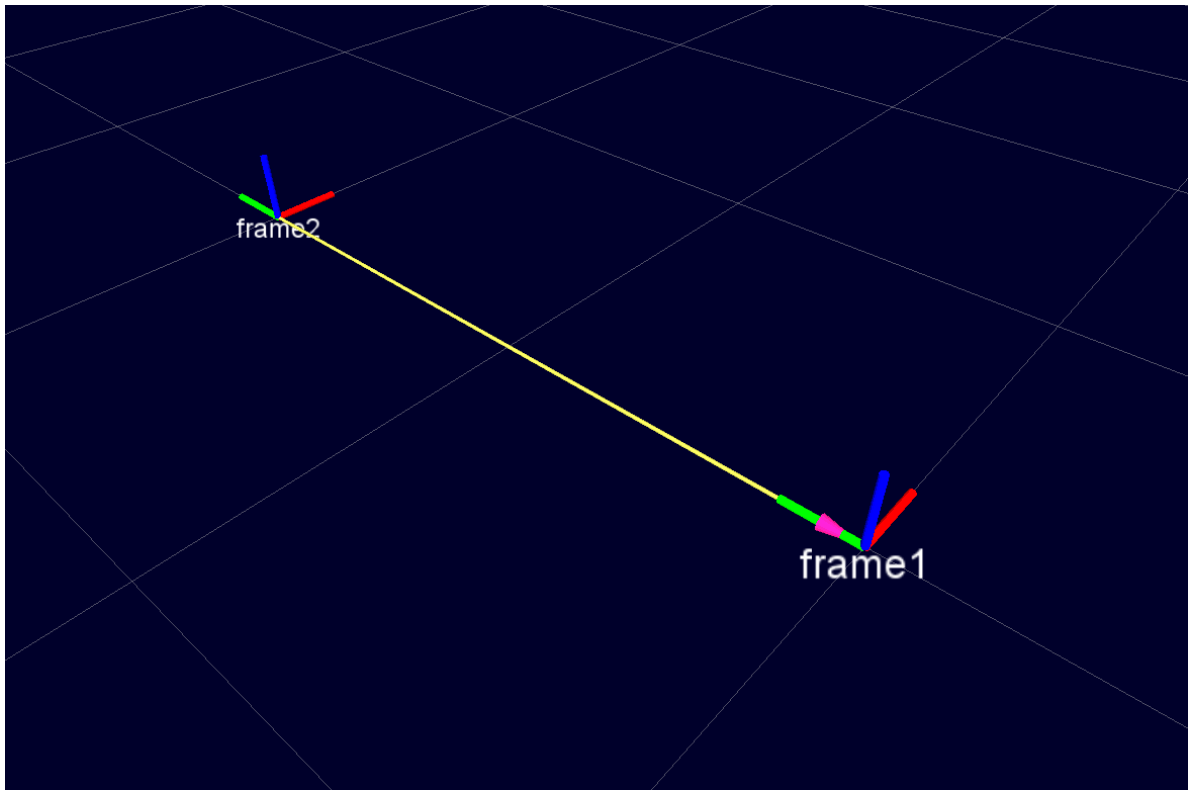
```
rosrun tf static_transform_publisher 0 2 0 0 0 0 1
                    /frame1 /frame2 30
```

  - Using roll-pitch-yaw:

```
rosrun tf static_transform_publisher 0 2 0 0 0 0
                    /frame1 /frame2 30
```

Static Transform Publishers

- In Rviz, the TF display shows the static relationship between *frame1* and *frame2*.



Static Transform Publishers

- Static transform publishers can be run from a launch file as well.

- The syntax is as follows:

```
<node pkg="tf" type="static_transform_publisher"
      name="frame1_to_frame2"
      args="0 2 0 0 0 0 /frame1 /frame2 30" />
```

- Keep in mind, all of this needs to be on the same line.

Marker Message

- See the ROS wiki for all the details about Rviz markers: (http://wiki.ros.org/rviz/DisplayTypes/Marker)

- A *visualization_msgs::Marker* message contains the following fields:
  - *header* – A standard ROS header with frame ID and time stamp.
  - *id* – A numerical ID that is used when the marker is in a *MarkerArray* message.
  - *type* – Type of marker (see slide 14)
  - *action* – Whether to add, modify or delete the marker.

Marker Message

- Marker message fields, continued:
  - *pose* – Position and orientation of the marker relative to the marker's TF frame.
  - *scale* – Used to control the size of the marker.
  - *color* – RGB values to set the color, and a transparency value.
  - *points* – Used for marker types that contain several points, such as a line strip.

Marker Types

- The marker type is selected using parameters specified in the message definition itself.
- Each different marker type uses some or all of the other fields in the marker message.
- Look up the precise behavior of each marker type on the wiki page: (http://wiki.ros.org/rviz/DisplayTypes/Marker)

```
micho@ubuntuvm: ~
micho@ubuntuvm:~$ rosmsg show Marker
[visualization_msgs/Marker]:
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
```

Marker Troubleshooting

- It is quite easy to make a marker completely invisible! The following are some common oversights that will cause your markers not to show up.

- Set the *scale* field. If they remain at the default of zero, the marker will not have any size.

```
geometry_msgs/Vector3 scale
  float64 x
  float64 y
  float64 z
```

Marker Troubleshooting

- Set the transparency. In the *color* field, be sure to change the transparency value *a* to something other than its default of zero. The marker will be completely transparent otherwise.

```
std_msgs/ColorRGBA color
  float32 r
  float32 g
  float32 b
  float32 a
```

Marker Troubleshooting

- Be sure the position of the marker is reasonable. You won't see the marker if it is literally miles away! Check the position values by displaying them in the terminal using *ROS_INFO*.

```
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Marker Troubleshooting

- Be sure to set the *frame_id* in the header of the marker message. If it is empty, Rviz will show that the marker has an error

- Also, if the frame does not exist in a *tf* tree, then Rviz won't know how to display it.

- Markers should always be time stamped. Be sure your node attaches a time stamp before you publish a marker message. Otherwise, Rviz may complain that the marker is too old.

- Make sure each marker in an array has a different *id* field. Rviz displays only the last marker with a particular ID.

Transform Broadcaster Class

- Transforms between reference frames can also be published from custom nodes.

- This is done using the *tf::TransformBroadcaster* class in two broad steps:

  > Populate a *tf::StampedTransform* object or *geometry_msgs::TransformStamped* structure with the desired translation and rotation, along with frame IDs and a time stamp.

  > Use a *tf::TransformBroadcaster* object to publish the transform.

Transform Broadcaster Class

- The *TransformBroadcaster* class cannot be declared as a global variable because it creates a *ros::NodeHandle* in its constructor, which isn't allowed to happen before *ros::init* is called.

- If the node is implemented as a class, it is fine to declare a *TransformBroadcaster* as a private property, since its constructor will run after the node's constructor, which in turn runs after *ros::init* is called.

Transform Broadcaster Class

- In a non-class node implementation, there are two ways to use a *TransformBroadcaster*:
  - Declare a global pointer to a *TransformBroadcaster*, and then instantiate the class in the main function using the *new* keyword.
  - Declare the *TransformBroadcaster* in the function where transforms will be published. Make sure it is declared as *static* so it doesn't get destroyed when it goes out of scope.

Transform Broadcaster Class

- The *StampedTransform* is declared like any other C++ variable:

```
tf::StampedTransform transform;
```

- The parent frame of the transform is specified in the *frame_id_* property of the *StampedTransform* object.
- The child frame is specified in the *child_frame_id_* property.
- The time stamp of the transform is specified as a ROS time in the *stamp_* property.
- The translation and rotation are specified using the *setOrigin* and *setRotation* methods, respectively.

Transform Broadcaster Class

- The *setOrigin* and *setRotation* methods work the same as in the *tf::Transform* case discussed previously.
- Once populated with translation and rotation information, the broadcaster object uses the stamped transform object to update the TF frame transformation.
- This is done with the *sendTransform* method:

```
broadcaster.sendTransform(transform);
```

Transform Broadcaster Class

- In a non-class node, the 2 meter y translation example from before can be done as follows:

```
void timerCallback(const ros::TimerEvent& event) {
  static tf::TransformBroadcaster broadcaster;
  tf::StampedTransform transform;
  transform.frame_id_ = "/frame1";
  transform.child_frame_id_ = "/frame2";
  transform.stamp_ = event.current_real;
  transform.setOrigin(tf::Vector3(0, 2, 0));
  transform.setRotation(tf::Quaternion(0, 0, 0, 1));
  broadcaster.sendTransform(transform);
}
```

- If the node is implemented as a class, the *TransformBroadcaster* would be a private property, and the static declaration in the timer callback method would not be necessary.