Introduction to Frames and Coordinate Transforms

Wyatt Newman
July, 2015

4x4, PoseStamped, Affine3d, transformStamped:
4x4: uses R matrix, to ref to child or parent frame;
Affine3d: similar to 4x4, but does not include 4th row; is an object w/ member fncs, such as inverse()
geometry_msgs/ transformStamped (header has frame ID, addl field for child_frame ID,
transform has Vector3 translation and Quaternion rotation)
Pose: has Point and Quaternion
PoseStamped has Header and Pose (and Header contains frame ID)

displaying kinematic tree
(defining coord frames in URDF)
what does static-transform publisher do? (e.g. for sensor xform)
what does robot_state_publisher do?
How does tfListener work?
Transforming sensor data w/ tfListener...

Introduction: Coordinate frames are ubiquitous in robotics. A robot's location in the world is defined by expressing a 6-DOF "pose" with respect to a reference frame in the world. Sequential (serial) links of a robot arm are described in terms of coordinate frames assigned uniquely to each link; as the arm moves, the relationships of these frames change with respect to their upstream (proximal, or "parent") frame. A sensor has a 6-DOF mounting pose, and the sensing element will express its data with respect to its own reference frame. An object of interest (e.g., an item to be grasped) can be described in terms of a frame defined with respect to the object plus the pose of the object's frame with respect to some frame of interest (e.g., the sensor frame, the base frame of the robot, a frame defined on the gripper, …).

In ROS, there are multiple ways to define spatial relationships between frames. These include a "Pose", a "Transform", a 4x4 matrix, and the "Affine3" transform object from the "Eigen" library. We begin with the conventional 4x4 or "homegenous transformation" representation.

Defining coordinate frames: A "frame" is defined by a point in 3-D space, **p**, that is the frame's origin, and three vectors: **n**, **t** and **b** (which define the local x, y and z axes, respectively). The axis vectors are normalized (have unit length), and they form a right-hand triad, such that **n** crossed into **t** equals **b:** $\boldsymbol{b}=\boldsymbol{n}\times\boldsymbol{t}$ These three directional axes can be stacked side-by-side as column vectors, comprising a 3x3 matrix, **R**.

$$\boldsymbol{R}=[\boldsymbol{n}\,\boldsymbol{t}\,\boldsymbol{b}]=\begin{bmatrix} n_x & t_x & b_x \\ n_y & t_y & b_y \\ n_z & t_z & b_z \end{bmatrix}$$

We can include the origin vector as well to define a 3x4 matrix as:

$$\boldsymbol{A}=[\boldsymbol{n}\,\boldsymbol{t}\,\boldsymbol{b}\,\boldsymbol{p}]=\begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \end{bmatrix}$$

A useful trick to simplify mathematical operations is to define an augmented matrix, converting the above 3x4 matrix into a square 4x4 matrix by adding a fourth row consisting of [0 0 0 1]. This augmented matrix is a 4x4, which we will refer to as a "**T**" matrix:

$$\boldsymbol{T}=\begin{bmatrix} n_x & t_x & b_x & p_x \\ n_y & t_y & b_y & p_y \\ n_z & t_z & b_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Conveniently, matrices constructed as above (consistent with valid frame specifications) are always invertible. Further, computation of the inverse of a **T** matrix is efficient.

Abstractly, one can refer to an origin and a set of orientation axes (vectors) without having to specify numerical values. However, to perform computations, numerical values are required—and this requires further definitions. Specifically, when numerical values are given, one must define the coordinate system in which the values are measured. For example, to specify the origin of frame "B" (i.e. point "**p**") with respect to frame "A", we can measure p_x along the x axis of frame "A", measure p_y along the y axis of frame "A", and p_z along the z axis of frame A. These can be referred to explicitly as $p_{x/A}$, $p_{y/A}$ and $p_{z/A}$, respectively. If we had provided coordinates for the origin of frame B from any other viewpoint, the numerical values of the components of **p** would be different.

Similarly, we can describe the components of frame-B's **n** axis (similarly, **t** and **b** axes) with respect to frame A by measuring the x, y and z components along the respective axes of frame A. We can then state the position and orientation of frame B with respect to frame A as:

$$^{A}\boldsymbol{T}_{\boldsymbol{B}}=\boldsymbol{T}_{\boldsymbol{B/A}}=\begin{bmatrix} n_{x/A} & t_{x/A} & b_{x/A} & p_{x/A} \\ n_{y/A} & t_{y/A} & b_{y/A} & p_{y/A} \\ n_{z/A} & t_{z/A} & b_{z/A} & p_{z/A} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can refer to the above matrix as "frame B with respect to frame A."

Having labelled frames "A" and "B", providing values for the elements of $^{A}\boldsymbol{T}_{\boldsymbol{B}}$ fully specifies the position and orientation of frame B with respect to frame A.

Coordinate-frame transformations: In addition to providing a means to explicitly declare the position and orientation of a frame (with respect to some named frame, e.g. frame B with respect to frame A), **T** matrices can also be interpreted as operators. For example, if we know the position and orientation of frame C with respect to frame B, i.e. $^{B}\boldsymbol{T}_{\boldsymbol{C}}$, and if we also know the position and orientation of frame B with respect to frame A, $^{A}\boldsymbol{T}_{\boldsymbol{B}}$, then we can compute the position and orientation of frame C with respect to frame A as follows:

$$^{A}\boldsymbol{T}_{\boldsymbol{c}}={}^{A}\boldsymbol{T}_{\boldsymbol{B}}\,{}^{B}\boldsymbol{T}_{\boldsymbol{C}}$$

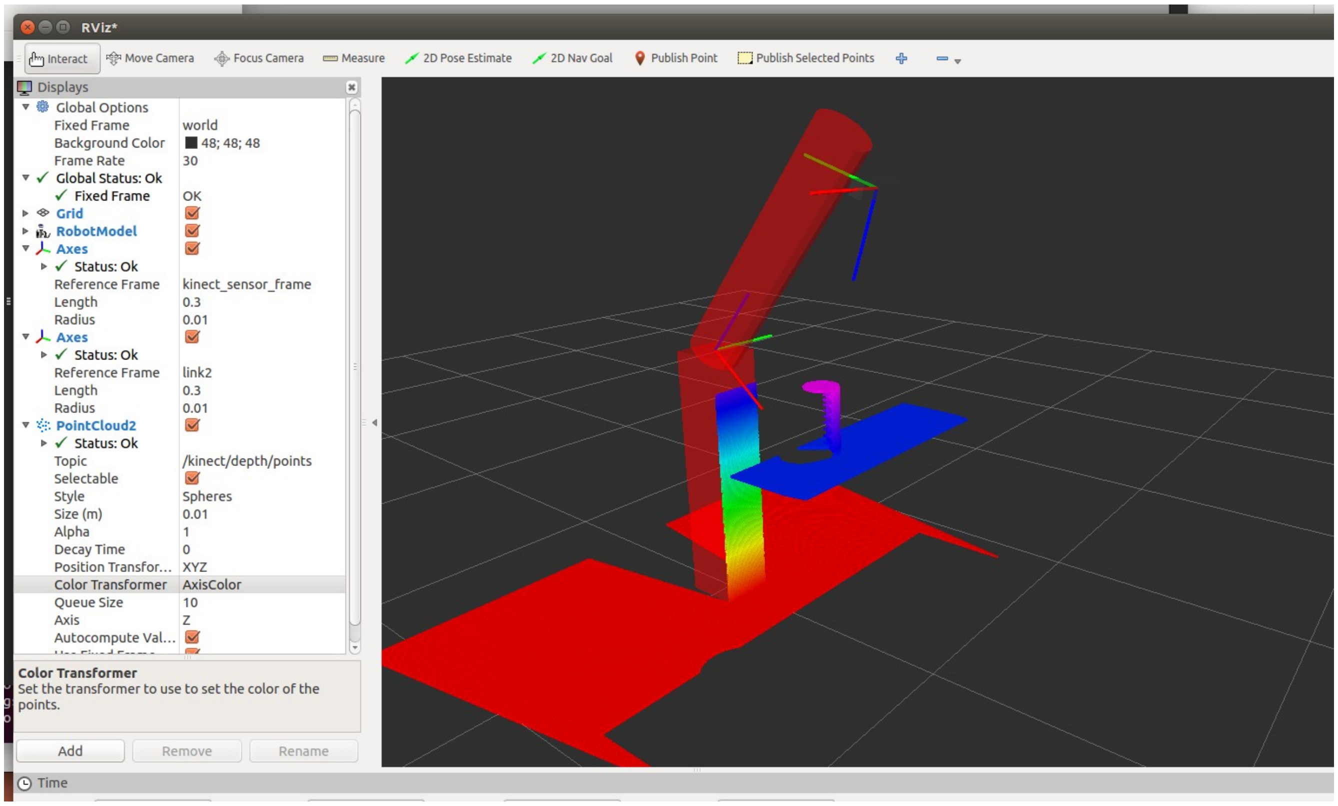
That is, a simple matrix multiplication yields the desired transform. This process can be extended, e.g.:

$$^{A}\boldsymbol{T}_{\boldsymbol{F}}={}^{A}\boldsymbol{T}_{\boldsymbol{B}}\,{}^{B}\boldsymbol{T}_{\boldsymbol{C}}\,{}^{C}\boldsymbol{T}_{\boldsymbol{D}}\,{}^{D}\boldsymbol{T}_{\boldsymbol{E}}\,{}^{E}\boldsymbol{T}_{\boldsymbol{F}}$$

In the above, the 4x4 on the left-hand side can be interpreted column by column. For example, the fourth column, rows 1 through 3, are the coordinates of the origin of frame “F” as measured with respect to frame “A.”

By using the notation of prefix superscripts and post subscripts, there is a visual mnemonic to aid logical compatibility. The super and sub-scripts act like “Lego” blocks, such that a subscript of a leading **T** matrix must match the pre-superscript of the trailing **T** matrix. Following this convention helps to keep transform operations consistent.

Assignment of frames to robot links: Robot arm kinematics depends on assignment of a unique frame to each solid body (link). From our earlier minimal robot description, we saw the following Rviz display, which includes “axes” displays of the kinect sensor frame and the link-2 frame, which is reiterated below:

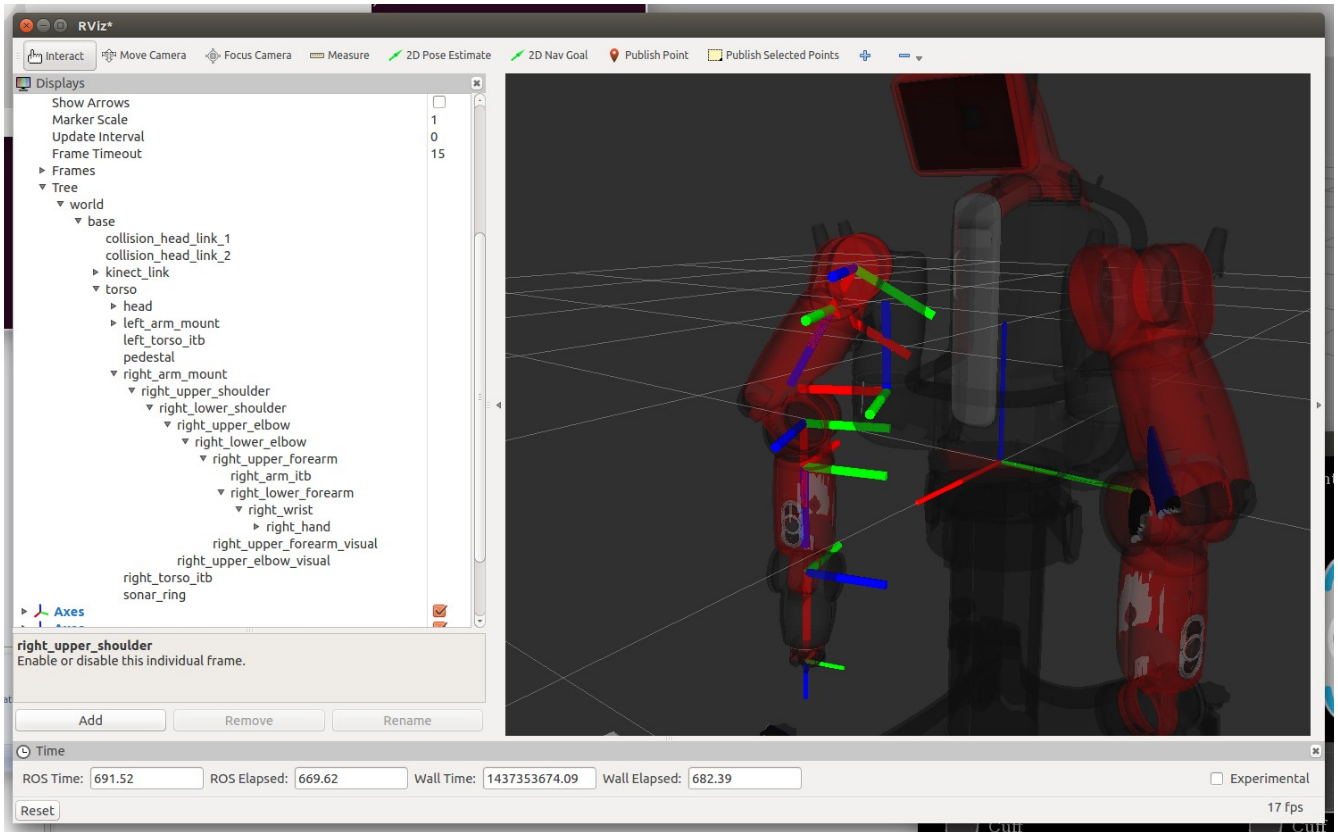


Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
Axes	✓
✓ Status: Ok	
Reference Frame	kinect_sensor_frame
Length	0.3
Radius	0.01
Axes	✓
✓ Status: Ok	
Reference Frame	link2
Length	0.3
Radius	0.01
PointCloud2	✓
✓ Status: Ok	
Topic	/kinect/depth/points
Selectable	✓
Style	Spheres
Size (m)	0.01
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	AxisColor
Queue Size	10
Axis	Z
Autocompute Val...	✓

Color Transformer
Set the transformer to use to set the color of the points.

In the above view, we see that the link2 coordinate frame has its z (blue) axis pointing through the axis of symmetry of the cylindrical link 2. Also, the y (green) axis of the link-2 frame points through the axis of rotation of the joint between links 1 and 2. As link2 moves, the link2 z-axis will move with link 2, continuing to point through link-2's axis of symmetry.

A more complex (and more realistic) example is shown below for the “Baxter” robot. The “torso” frame has its z (blue) axis pointing up through Baxter's main body. This frame is stationary, unless Baxter is repositioned with respect to the world frame. Baxter's right arm has 7 degrees of freedom (similar to a human arm). Each degree of freedom is a rotational joint. Coordinate frames are assigned conveniently with respect to the joint axes. In the figure below, each frame's z (blue) axis is coincident with a joint rotational axis. Although URDF specifications allow for frames to be defined more generally, it is conventional to choose frame z axes to be coincident with joint axes.



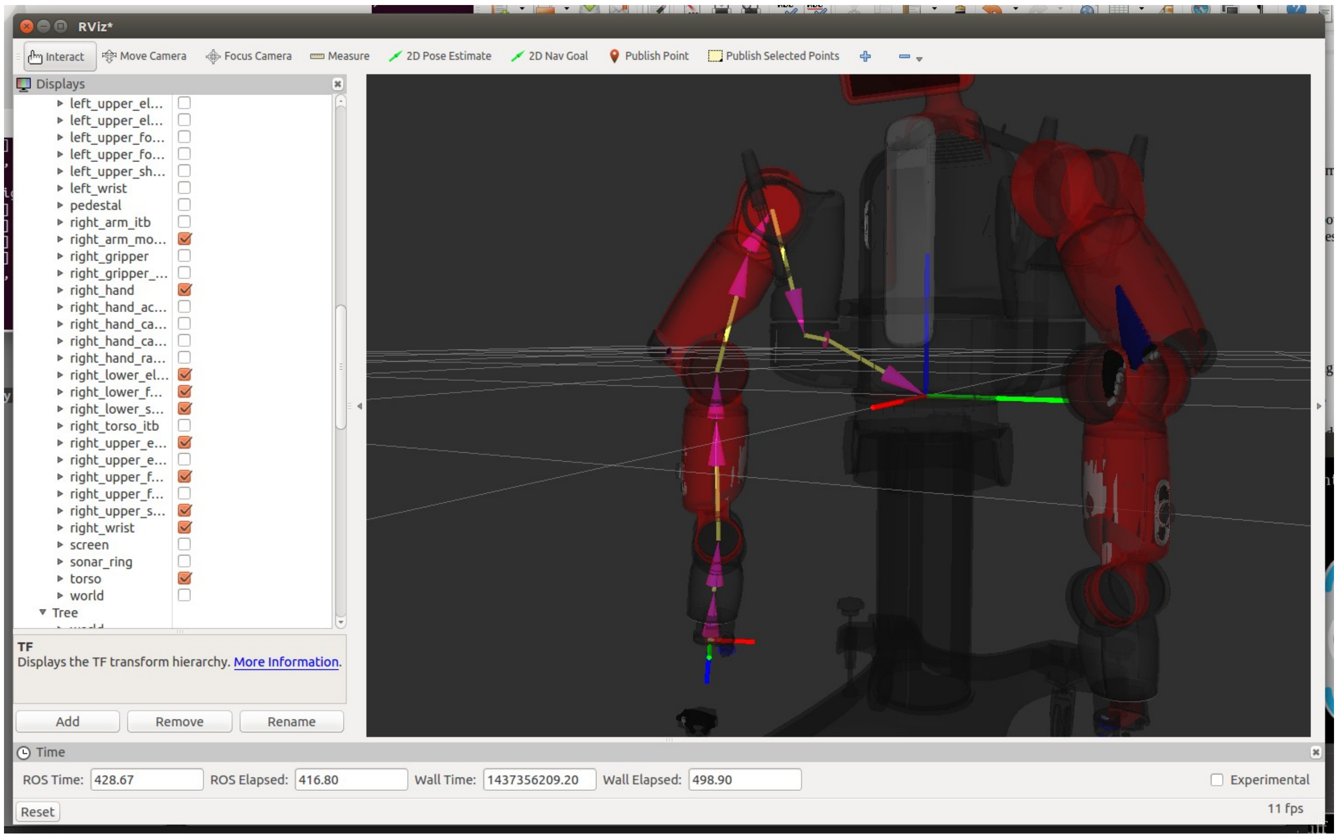
```
Show Arrows          [ ]
Marker Scale         1
Update Interval      0
Frame Timeout        15
▸ Frames
▾ Tree
  ▾ world
    ▾ base
        collision_head_link_1
        collision_head_link_2
      ▸ kinect_link
      ▾ torso
        ▸ head
        ▸ left_arm_mount
          left_torso_itb
          pedestal
        ▾ right_arm_mount
          ▾ right_upper_shoulder
            ▾ right_lower_shoulder
              ▾ right_upper_elbow
                ▾ right_lower_elbow
                  ▾ right_upper_forearm
                      right_arm_itb
                    ▾ right_lower_forearm
                      ▾ right_wrist
                        ▸ right_hand
                    right_upper_forearm_visual
                right_upper_elbow_visual
          right_torso_itb
          sonar_ring
```

right_upper_shoulder
Enable or disable this individual frame.

ROS Time: 691.52 ROS Elapsed: 669.62 Wall Time: 1437353674.09 Wall Elapsed: 682.39

17 fps

The right-arm frames are related to each other through a sequence of parent/child relationships from the torso out to the hand, as illustrated below (with arrows pointing from child to parent).



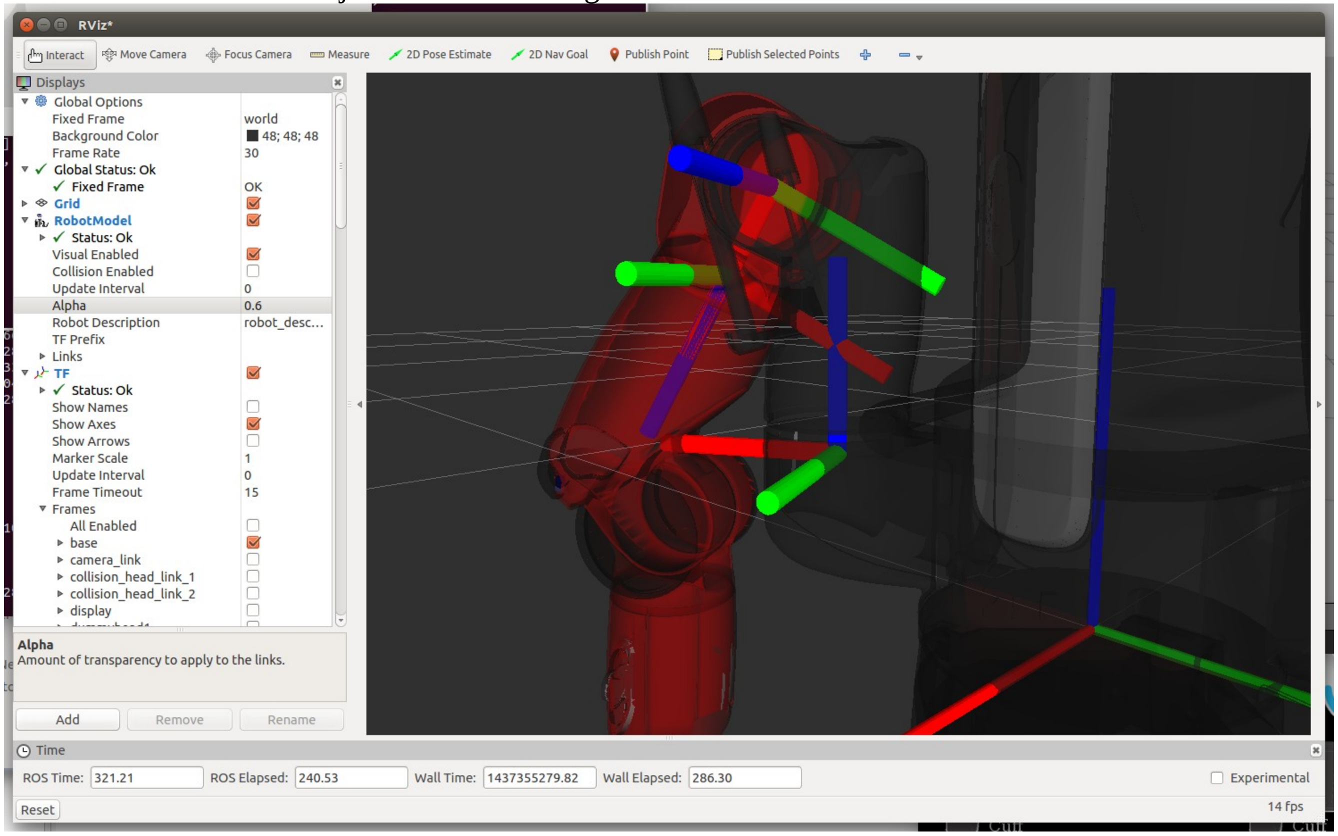
```
▸ left_upper_el…     [ ]
▸ left_upper_el…     [ ]
▸ left_upper_fo…     [ ]
▸ left_upper_fo…     [ ]
▸ left_upper_sh…     [ ]
▸ left_wrist         [ ]
▸ pedestal           [ ]
▸ right_arm_itb      [ ]
▸ right_arm_mo…      [✓]
▸ right_gripper      [ ]
▸ right_gripper_…    [ ]
▸ right_hand         [✓]
▸ right_hand_ac…     [ ]
▸ right_hand_ca…     [ ]
▸ right_hand_ca…     [ ]
▸ right_hand_ra…     [ ]
▸ right_lower_el…    [✓]
▸ right_lower_f…     [✓]
▸ right_lower_s…     [✓]
▸ right_torso_itb    [ ]
▸ right_upper_e…     [✓]
▸ right_upper_e…     [ ]
▸ right_upper_f…     [✓]
▸ right_upper_f…     [ ]
▸ right_upper_s…     [✓]
▸ right_wrist        [✓]
▸ screen             [ ]
▸ sonar_ring         [ ]
▸ torso              [✓]
▸ world              [ ]
▾ Tree
```

TF
Displays the TF transform hierarchy. More Information.

ROS Time: 428.67 ROS Elapsed: 416.80 Wall Time: 1437356209.20 Wall Elapsed: 498.90

11 fps

A zoom on the first three joints of Baxter's right arm is below:



```
Global Options
  Fixed Frame            world
  Background Color       48; 48; 48
  Frame Rate             30
✓ Global Status: Ok
  ✓ Fixed Frame          OK
Grid                     ✓
RobotModel               ✓
  ✓ Status: Ok
  Visual Enabled         ✓
  Collision Enabled      ☐
  Update Interval        0
  Alpha                  0.6
  Robot Description      robot_desc...
  TF Prefix
  Links
TF                       ✓
  ✓ Status: Ok
  Show Names             ☐
  Show Axes              ✓
  Show Arrows            ☐
  Marker Scale           1
  Update Interval        0
  Frame Timeout          15
  Frames
    All Enabled          ☐
    base                 ✓
    camera_link          ☐
    collision_head_link_1 ☐
    collision_head_link_2 ☐
    display              ☐

Alpha
Amount of transparency to apply to the links.

ROS Time: 321.21   ROS Elapsed: 240.53   Wall Time: 1437355279.82   Wall Elapsed: 286.30   ☐ Experimental
14 fps
```

The first shoulder frame, named “right_upper_shoulder,” can move with respect to the torso frame. This motion is constrained to rotation about the vertical, blue axis of the right_upper_shoulder frame. The rotation of this frame depends on the joint angle called “right_s0”. As the angle of right_s0 changes, the red and green axes of the right_upper_shoulder frame will rotate, but the origin of this frame will remain constant, and the z-axis will remain vertical (parallel to the torso-frame z axis).

Although rotation of joint right_s0 changes frame right_upper_shoulder by only a pure rotation about the right_upper_shoulder z-axis, such motion has a more significant effect on all distal (child) frames of the right arm. All of these frames experience both a rotation and a translation of their respective origins.

The next frame (immediate child of right_upper_shoulder) is (arbitrarily) named the “right_lower_shoulder” frame. This frame has a horizontal z axis, and this z axis is coincident with the rotation axis of joint “right_s1”.   Movement of joint right_s1 causes the frame right_lower_shoulder_frame to rotate about its z axis—while this frames origin remains stationary. Motion of the right_s1 joint has no effect on the parent frame, right_upper_shoulder.

The third frame illustrated is the “right_upper_elbow” frame, with corresponding joint “right_e0.” Motion of joint right_e0 causes rotation of frame right_upper_elbow about its z axis, while this frame's origin remains stationary. The right_upper_elbow frame z-axis is conceptually equivalent to the major axis of a human-arm humerus (upper arm) bone.

Motion of right_e0 has no influence on the upstream (parent) frames right_upper_shoulder and right_lower_shoulder. This pattern continues sequentially through to the final frame in the chain—

typically a frame defined conveniently on a tool or gripper. A frame is defined for each "link" in the URDF robot description, and joints are defined that relate a "parent" frame to a "child" frame. A coordinate-frame transform exists between each parent/child pair, and this transform is a function of the joint displacement (e.g. joint angle) for that joint that relates the parent link to the child link. By multiplying together the sequence of such transforms, one can compute the position and orientation of the final (e.g. gripper) frame with respect to the base (e.g. torso) frame.

Transforms in ROS: Although 4x4 matrices are the historical approach to coordinate-frame transformations, implementations in ROS are slightly different. The "tf" class (see http://wiki.ros.org/tf) defines datatypes tf::Quaternion, tf::Vector3, tf::Point, tf::Pose and tf::Transform —as well as the "stamped" versions of these, and the special case "tf::StampedTransform."

Internally, a "tf::Transform" contains separate representations for frame origin and orientation. Member functions enable requesting the origin or the orientation, either as a quaternion (via member function getRotation()) or a 3x3 matrix (via member function getBasis()). There is a direct mapping between a 4x4 homogeneous transformation matrix and components of a tf::Transform. Further, although a tf::Transform object is not a matrix, the operator "*" is nonetheless defined, such that Transform objects tf1 and tf2, corresponding to R1 and R2 4x4 matrices, can be multiplied such that tf3= tf1*tf2 is defined, and tf3 is the equivalent tf representation of R3, where R3 = R1*R2.

A tf::StampedTransform object additionally contains an optional time stamp and a pair of strings identifying the child frame id and the parent frame id. Equivalently, such objects incorporate the super and sub-scripts of the 4x4 matrices introduced earlier. This information can be exploited in computations to keep track of transforms between named frames.

While these ROS datatypes are logical enough, the "tf" package can be confusing, given the broad functionality that it contributes. Tf publishers transmit transforms on the topic "/tf". Within ROS, the topic "/tf" carries messages of type tf2_msgs/TFMessage. There can be (and typically are) many publishers to this topic. Each publisher expresses a transform relationship, describing a named "child" frame with respect to a named "parent" frame. Each such transform message carries the equivalent information of a 4x4 homogeneous transformation matrix, in addition to named parent and child frame id's. Transform messages can also carry a history of prior transforms (each with a corresponding time stamp).

User-written ROS nodes can publish transforms to the tf topic, announcing important spatial relationships. We have seen this already with the "robot_state_publisher" node, which operates on published joint-angle data (joint_states), computes corresponding transformations (with information from the robot description URDF) and publishes resulting transforms. In simpler cases, the mount position/orientation of a sensor on a robot can be published on the /tf topic as a "static" transform.

If a robot is mobile, it can be important to know where the robot is in the world—known as "localization." Computation of localization relative to a map can be broadcast by posting the result equivalent to ${}^{map}\boldsymbol{T}_{robot}$ to the tf topic (using the corresponding tf message type).

A robot arm can publish successive link transforms on the tf topic. When sensor data is published, it should use a header that contains the name of the sensor's frame. This information, together with tf messages, allows one to transform the sensor data to any frame of interest (e.g., world frame or robot frame).

A user node can instantiate a "transformListener" object (from the tf library, see http://wiki.ros.org/tf/Tutorials/Writing a tf listener), which is very useful for computing coordinate transforms. The transformListener subscribes to the tf topic, receives messages relating child and parent frames (with random arrival times) and assembles this information into a connected chain (using the most recent available pair-wise transforms). By maintaining this chain, the transformListener is capable of responding to inquiries regarding coordinate-frame transforms (e.g., where is my right index fingertip with respect to my nose).

Transforms in Eigen: The "Eigen" library (see http://eigen.tuxfamily.org) is a useful library for efficient linear-algebra computations. The Transform class in Eigen includes "Affine transformations", including Affine3f (single-precision floating point) and Affine3d (double-precision floating point), which are equivalent to transforms 3-D. These datatypes include "linear" (a 3x3 rotation matrix) and "translation" (3x1 displacement vector) components, and thus they are suitable for storing transform information. The "*" operator is defined for objects of type Eigen::Affine, such that two affine objects multiplied together yield another affine object, where the rotation and translation components of the result are equivalent to multiplying 4x4 homogeneous transformation matrices. Affine objects have a variety of member functions, including means for inversion and extraction of rotation information in different representations. Eigen affine objects do not, however, contain time stamps nor parent/child frame id's.

A look inside robot_state_publisher: The node "robot_state_publisher" subscribes to the topic "joint_states", which carries messages of type "sensor_msgs/JointState." It consults the robot URDF from the parameter server, combines this information with published joint angles, deduces the resulting coordinate frames of each robot link, and publishes these on the topic "tf." For our minimal robot, we used robot_state_publisher to compute and publish corresponding link transforms. (Only link2 had a variable transform for this minimal example). We can "spoof" robot_state_publisher to illustrate its operation.

The package "example_tf_publisher" contains

```
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 463
        nsecs: 75000000
      frame_id: link1
    child_frame_id: link2
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 1.0
      rotation:
        x: 0.0
        y: -0.580646118311
        z: 0.0
```

w: 0.814156057087