ROS

Stage and TF

Agenda

- Stage:
  - Stage simulator
  - Reading laser sensor data
  - Writing a simple walker
- TF:
  - ROS transformation system
  - Writing a tf broadcaster
  - Writing a tf listener

ROS Stage Simulator

- http://wiki.ros.org/simulator_stage
- A 2D simulator that provides a virtual world populated by mobile robots, along with various objects for the robots to sense and manipulate

Run Stage with an existing world file

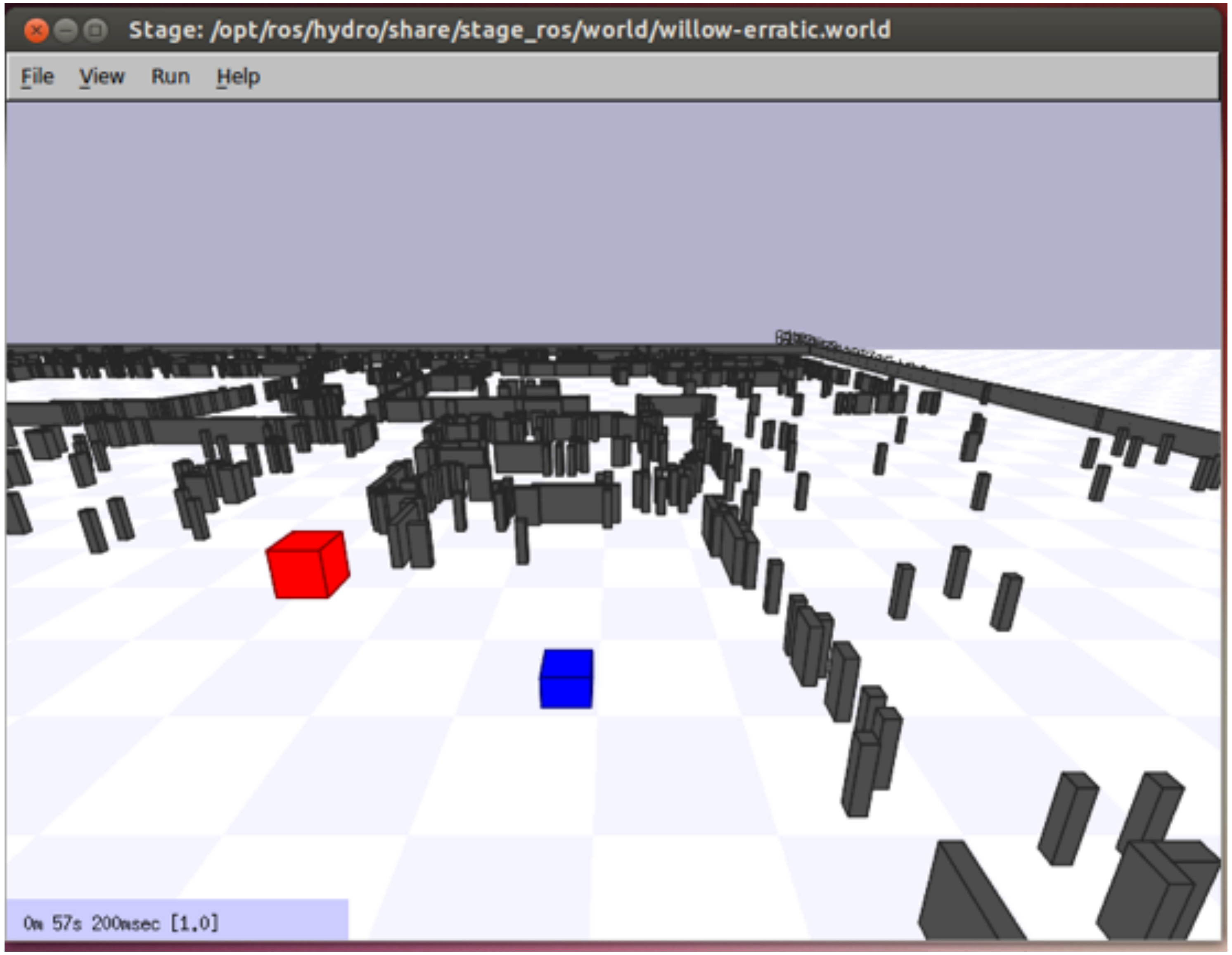
- Stage is already installed with ROS Indigo
- Stage ships with some example world files, including one that puts an Erratic-like robot in a Willow Garage-like environment
- To run it with an existing world file

```
rosrun stage_ros stageros `rospack find stage_ros`/world/willow-erratic.world
```

- Browsing the stage window should show up 2 little squares: a red square which is a red box and a blue square which is the erratic robot

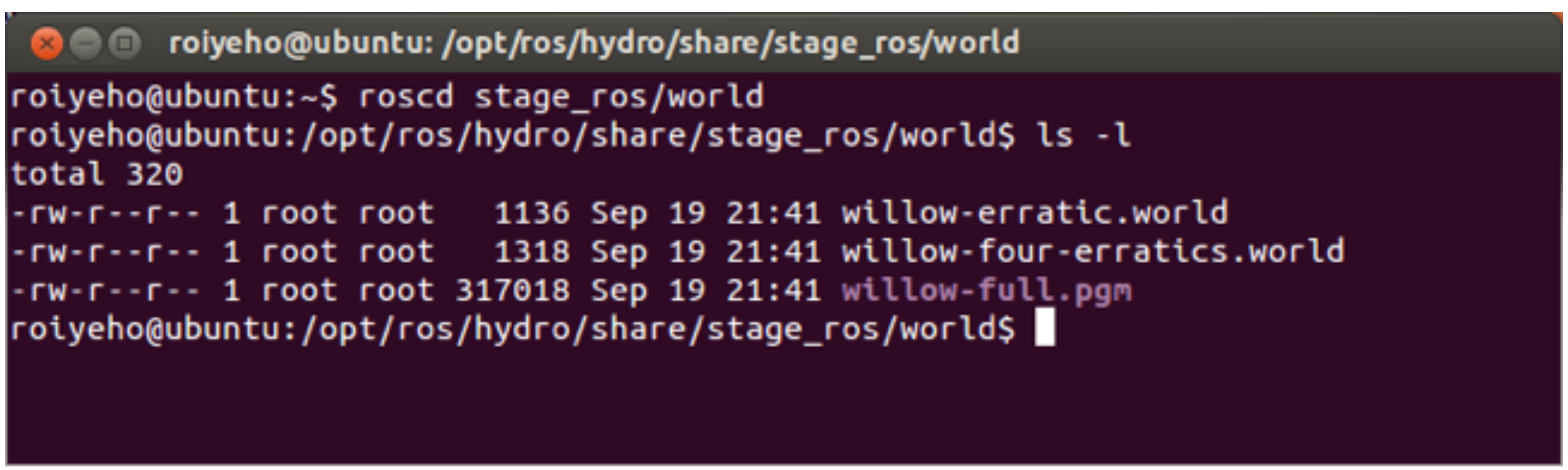
Run Stage with an existing world file

- Click on the stage window and press R to see the perspective view.



Stage World Files

- The world file is a description of the world that Stage must simulate.
- It describes robots, sensors, actuators, moveable and immovable objects.
- Sample world files can be found at the /world subdirectory in ros_stage package



World File Format

- The basic syntactic features of the world file format: comments, entities and properties
- The define statement can be used to define new types of entities.
    - define myrobot position (player() laser())
- Entities have properties, indicated using name value pairs
    - myrobot (name "robot1" port 6665 pose [1 1 0])
        - This entry creates a position device named "robot1" attached to port 6665, with initial position (1, 1) and orientation of 0

Stage World File Example

```
define block model
(
  size [0.5 0.5 0.75]
  gui_nose 0
)

define topurg ranger
(
  sensor(
    range_max 30.0
    fov 270.25
    samples 1081
  )
  # generic model properties
  color "black"
  size [ 0.05 0.05 0.1 ]
)

define pr2 position
(
  size [0.65 0.65 0.25]
  origin [-0.05 0 0 0]
  gui_nose 1
  drive "omni"
  topurg(pose [ 0.275 0.000 0 0.000 ])
)
```

Stage World File Example

```
define floorplan model
(
  # sombre, sensible, artistic
  color "gray30"

  # most maps will need a bounding box
  boundary 1

  gui_nose 0
  gui_grid 0

  gui_outline 0
  gripper_return 0
  fiducial_return 0
  ranger_return 1
)

# set the resolution of the underlying raytrace model in meters
resolution 0.02

interval_sim 100  # simulation timestep in milliseconds
window
(
  size [ 745.000 448.000 ]
  rotate [ 0.000 -1.560 ]
  scale 18.806
)
```

Stage World File Example

```
# load an environment bitmap
floorplan
(
  name "willow"
  bitmap "../maps/willow-full-0.05.pgm"
  size [58.25 47.25 1.0]
  pose [ -23.625 29.125 0 90.000 ]
)

# throw in a robot
pr2( pose [ -28.610 13.562 0 99.786 ] name "pr2" color "blue")
block( pose [ -25.062 12.909 0 180.000 ] color "red")
block( pose [ -25.062 12.909 0 180.000 ] color "red")
block( pose [ -25.062 12.909 0 180.000 ] color "red“)
```

Move the robot around

- Now run teleop_twist_keyboard:

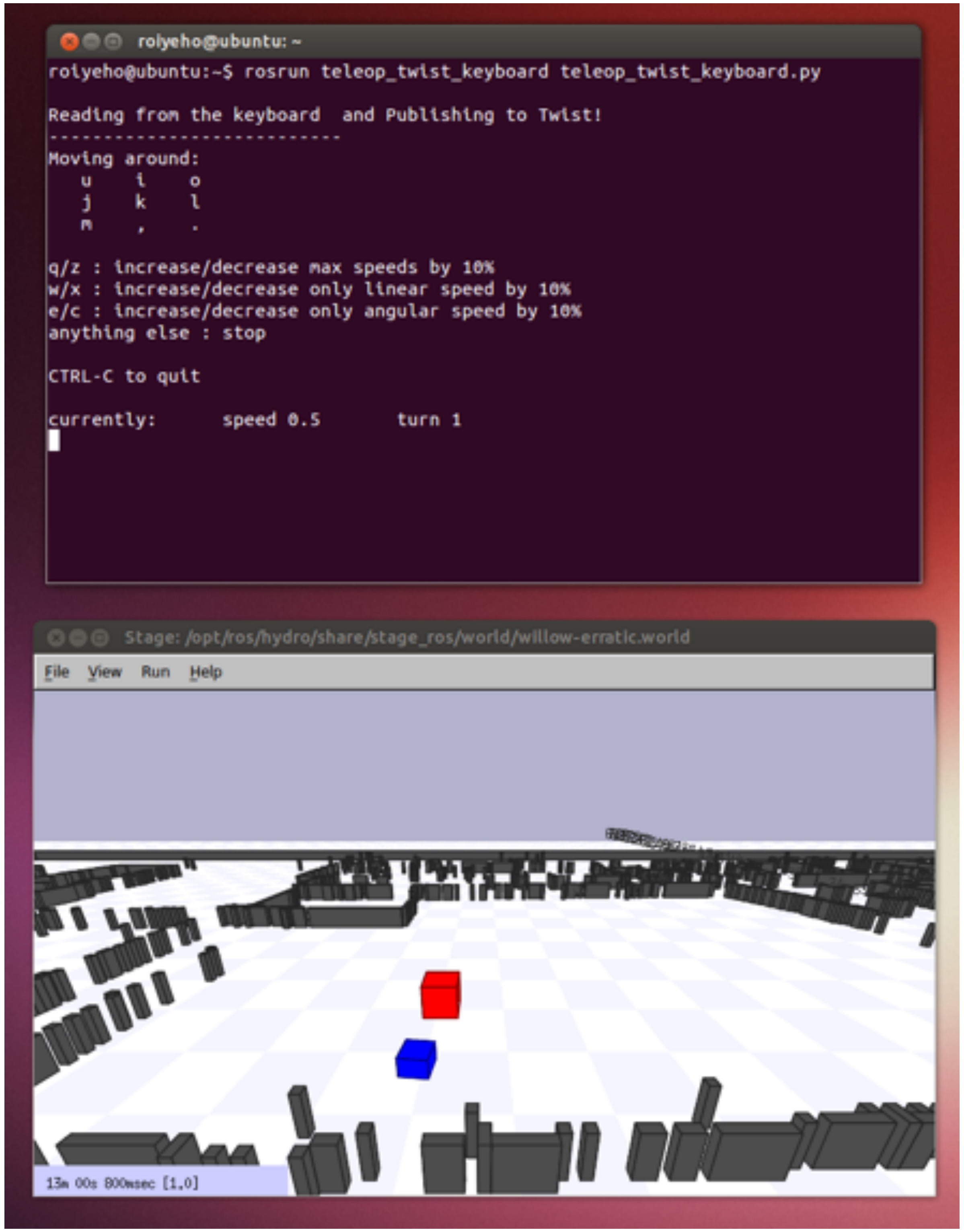
```
$ sudo apt-get install ros-indigo-teleop-twist-keyboard
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

- You should see console output that gives you the key-to-control mapping, something like this:

```
Reading from keyboard
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

anything else : stop
---------------------------
```

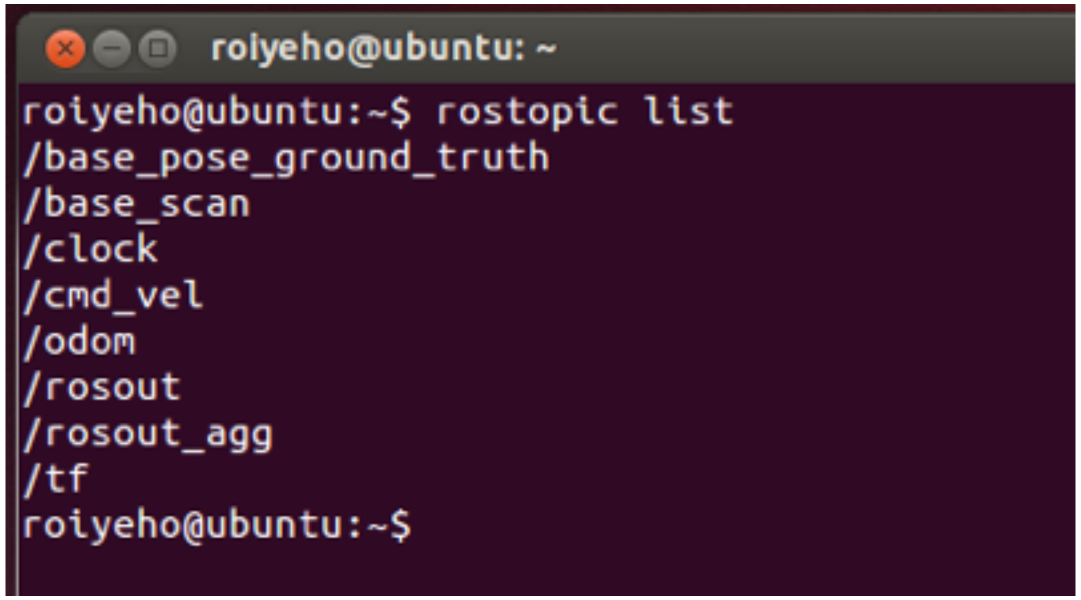
- Hold down any of those keys to drive the robot. E.g., to drive forward, hold down the i key.

Move the robot around



Stage Published Topics

- The stage simulator publishes several topics
- See what topics are available using rostopic list



- You can use rostopic echo -n 1 to look at an instance of the data on one of the topics

Odometry Messages

roiyeho@ubuntu: ~

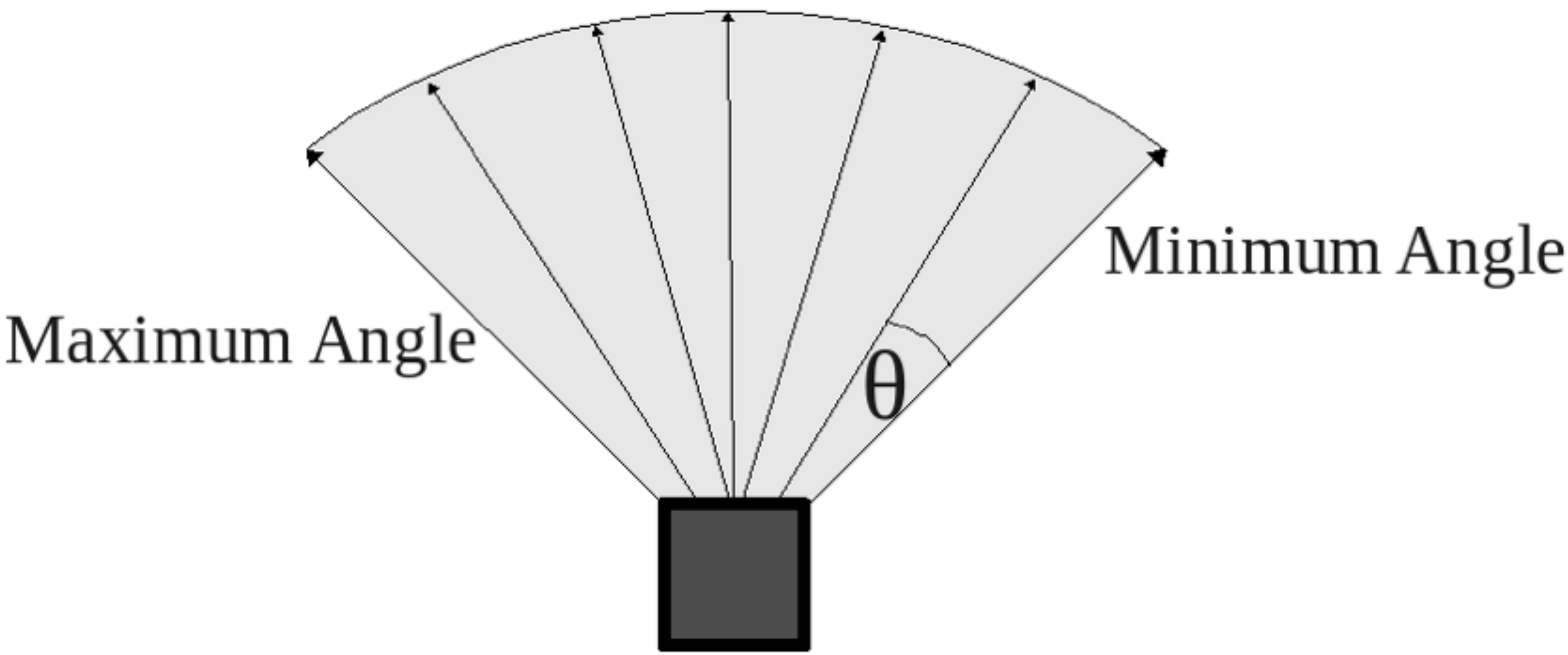
```
roiyeho@ubuntu:~$ rostopic echo /odom -n 1
header:
  seq: 11485
  stamp:
    secs: 1148
    nsecs: 600000000
  frame_id: odom
child_frame_id: ''
pose:
  pose:
    position:
      x: 1.16596142952
      y: -0.133586349782
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.389418342309
      w: 0.921060994003
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
```

A Stopper Node

- We will create a node called **stopper** that will make the robot move forward until it detects an obstacle in front of it
- We will use the laser sensor data to achieve this
- Create a new package called my_stage

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg my_stage std_msgs rospy roscpp
```

Laser Scanner



Hokuyo Laser

- A common laser sensor used in robotics

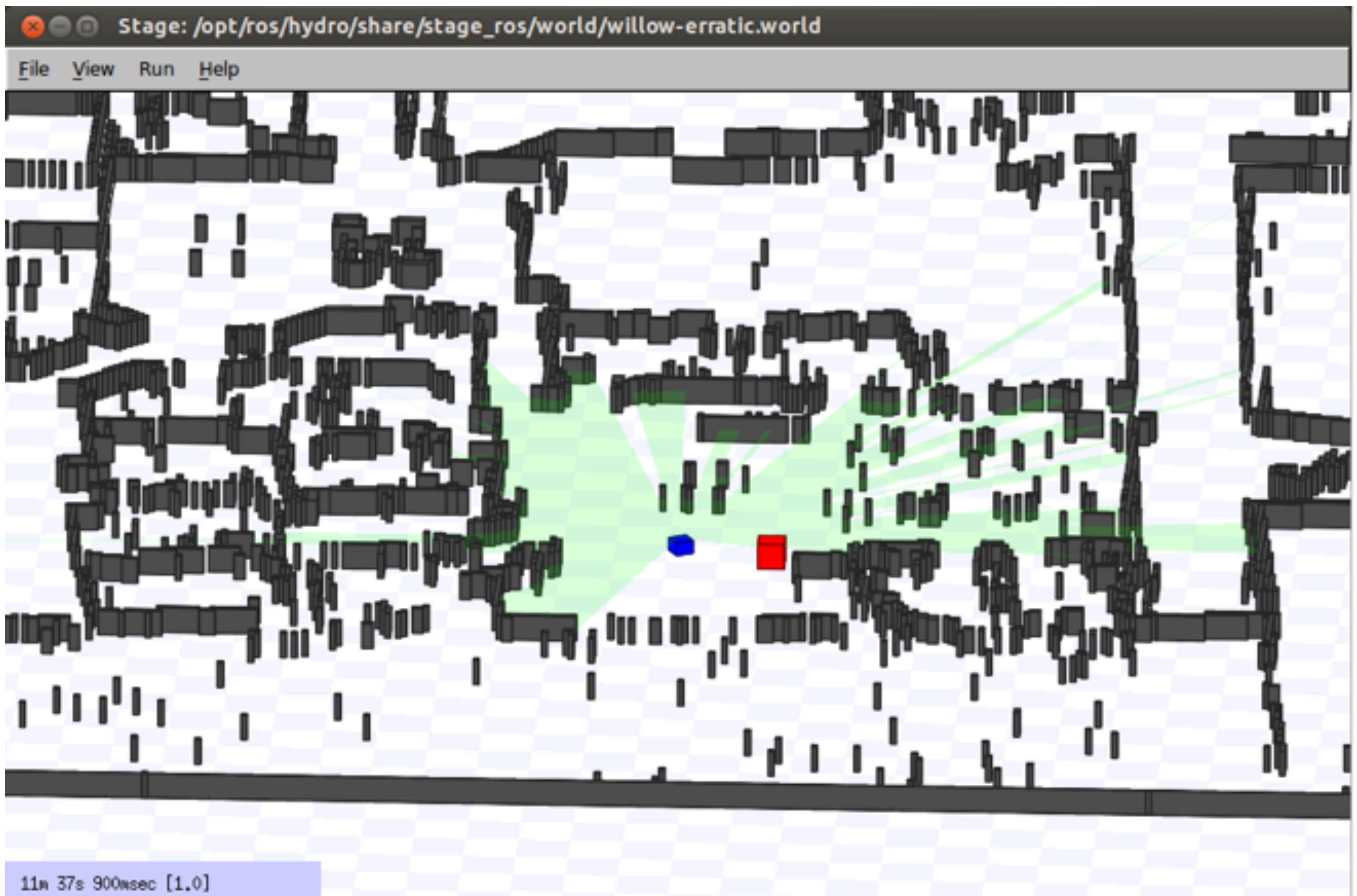
  http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html

Hokuyo Laser

Model No.	URG-04LX
Power source	5VDC ± 5%*1
Current consumption	500mA or less(800mA when start-up)
Measuring area	60 to 4095mm(white paper with 70mm□)
	240°
Accuracy	60 to 1,000mm : ± 10mm, 1,000 to 4,095mm : 1% of measurement
Repeatability	60 to 1,000mm : ± 10mm
Angular resolution	Step angle : approx. 0.36° (360° /1,024 steps)
Light source	Semiconductor laser diode(λ=785nm), Laser safety class 1(IEC60825-1, 21 CFR 1040.10 & 1040.11)
Scanning time	100ms/scan
Noise	25dB or less
Interface	USB, RS-232C(19.2k, 57.6k, 115.2k, 250k, 500k, 750kbps), NPN open-collector(synchronous output of optical scanner : 1 pce)
Communication specifications	Exclusive command(SCIP Ver.1.1 or Ver.2.0)*2
Ambient temperature/humidity	-10 to +50 degrees C, 85% or less(Not condensing, not icing)
Vibration resistance	10 to 55Hz, double amplitude 1.5mm Each 2 hour in X, Y and Z directions
Impact resistance	$196m/s^2$, Each 10 time in X, Y and Z directions
Weight	Approx. 160g
Accessory	Cable for power･communication/input･output(1.5m) 1 pce, D-sub connector with 9 pins 1 pce*3

Laser Scan Data

- In Stage Click D (or choose View > Data) to see the laser data on the map



Laser Scan Data

- Laser data is published to the topic **/base_scan**
- The message type that used to send information of the laser is sensor_msgs/LaserScan
- You can see the structure of the message using

```
$rosmsg show sensor_msgs/LaserScan
```

- Note that Stage produces perfect laser scans
  - Real robots and real lasers exhibit noise that Stage isn't simulating

LaserScan Message

- http://docs.ros.org/api/sensor_msgs/html/msg/LaserScan.html

```
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame_id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be
discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```

LaserScan Message

- Example of a laser scan message from Stage simulator:

```
roiyeho@ubuntu: ~
---
header:
  seq: 1594
  stamp:
    secs: 159
    nsecs: 500000000
  frame_id: base_laser_link
angle_min: -2.35837626457
angle_max: 2.35837626457
angle_increment: 0.00436736317351
time_increment: 0.0
scan_time: 0.0
range_min: 0.0
range_max: 30.0
ranges: [2.427844524383545, 2.42826247215271, 2.4287266731262207, 2.4292376041412354, 2.429795026779175, 2.430398941
040039, 2.4310495853424072, 2.4317471981048584, 2.4324913024902344, 2.4332826137542725, 2.4341206550598145, 2.435005
6648254395, 2.4359381198883057, 2.436917543411255, 2.437944173812866, 2.439018487930298, 2.4401402473449707, 2.44130
94520568848, 2.4425265789031982, 2.443791389465332, 2.4451043605804443, 2.446465253829956, 2.4478745460510254, 2.449
3319988250732, 2.450838088989258, 2.452392816543579, 2.453996419906616, 2.455648899078369, 2.457350492477417, 2.4591
01438522339, 2.460901975631714, 2.462752103805542, 2.4646518230438232, 2.466601848602295, 2.468601942062378, 2.47065
23418426514, 2.4727535247802734, 2.474905490875244, 2.4771084785461426, 2.479362726211548, 2.481668472290039, 2.4840
259552001953, 2.4864354133605957, 2.4888970851898193, 2.4914112091064453, 2.4939777851104736, 2.4965975284576416, 2.
4992706775665283, 2.5019969940185547, 2.504777193069458, 2.5076115131378174, 2.510500192642212, 2.5134434700012207,
2.516441822052002, 2.5194954872131348, 2.5226047039031982, 2.5257697105407715, 2.5289909839630127, 2.53226900100708,
 2.5356037616729736, 2.5389959812164307, 2.542445659637451, 2.5459535121917725, 2.5495197772979736, 2.55314469337463
4, 2.5568289756774902, 2.560572624206543, 2.56437611579895, 2.568240165710449, 2.572165012359619, 2.57615113258361 8,
 2.5801987648010254, 2.584308624267578, 2.5884809494018555, 2.5927164554595947, 2.597015380859375, 2.601378202438354
5, 2.6058056354522705, 2.610297918319702, 2.6148557662963867, 2.6194796562194824, 2.6241698265075684, 2.628927230834
961, 2.6337523460388184, 2.63478422164917, 2.6436073780059814, 2.6486384868621826, 2.6537396907806396, 3.44798207283
02, 3.4547808170318604, 3.461672306060791, 3.4686577320098877, 3.4757378101348877, 3.4829134941101074, 3.49018549919
1284, 3.4975550174713135, 3.5050225257873535, 3.5125889778137207, 3.5202558040618896, 3.5280232429504395, 3.53589296
3409424, 3.543865442276001, 3.5519418716430664, 3.5601232051849365, 3.568410634994507, 3.5768051147460938, 3.5853075
```

A Stopper Node

- In QtCreator under the package's src directory add a new class named Stopper
  - This will create Stopper.h and Stopper.cpp
- Add a source file run_stopper.cpp
  - This will contain the main function

Stopper.h

```
#include "ros/ros.h"
#include "sensor_msgs/LaserScan.h"

class Stopper {
public:
    // Tunable parameters
    const static double FORWARD_SPEED_MPS = 0.5;
    const static double MIN_SCAN_ANGLE_RAD = -30.0/180*M_PI;
    const static double MAX_SCAN_ANGLE_RAD = +30.0/180*M_PI;
    const static float MIN_PROXIMITY_RANGE_M = 0.5; // Should be smaller than
sensor_msgs::LaserScan::range_max

    Stopper();
    void startMoving();

private:
    ros::NodeHandle node;
    ros::Publisher commandPub; // Publisher to the robot's velocity command topic
    ros::Subscriber laserSub; // Subscriber to the robot's laser scan topic
    bool keepMoving; // Indicates whether the robot should continue moving

    void moveForward();
    void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan);
};
```

Stopper.cpp (1)

```
#include "Stopper.h"
#include "geometry_msgs/Twist.h"

Stopper::Stopper()
{
    keepMoving = true;

    // Advertise a new publisher for the simulated robot's velocity command topic
    commandPub = node.advertise<geometry_msgs::Twist>("cmd_vel", 10);

    // Subscribe to the simulated robot's laser scan topic
    laserSub = node.subscribe("base_scan", 1, &Stopper::scanCallback, this);
}

// Send a velocity command
void Stopper::moveForward() {
    geometry_msgs::Twist msg; // The default constructor will set all commands to 0
    msg.linear.x = FORWARD_SPEED_MPS;
    commandPub.publish(msg);
};
```

Stopper.cpp (2)

```
// Process the incoming laser scan message
void Stopper::scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan)
{
    // Find the closest range between the defined minimum and maximum angles
    int minIndex = ceil((MIN_SCAN_ANGLE_RAD - scan->angle_min) / scan-
>angle_increment);
    int maxIndex = floor((MAX_SCAN_ANGLE_RAD - scan->angle_min) / scan-
>angle_increment);

    float closestRange = scan->ranges[minIndex];
    for (int currIndex = minIndex + 1; currIndex <= maxIndex; currIndex++) {
        if (scan->ranges[currIndex] < closestRange) {
            closestRange = scan->ranges[currIndex];
        }
    }

    ROS_INFO_STREAM("Closest range: " << closestRange);

    if (closestRange < MIN_PROXIMITY_RANGE_M) {
        ROS_INFO("Stop!");
        keepMoving = false;
    }
}
```

Stopper.cpp (3)

```
void Stopper::startMoving()
{
    ros::Rate rate(10);
    ROS_INFO("Start moving");

    // Keep spinning loop until user presses Ctrl+C or the robot got too close to an obstacle
    while (ros::ok() && keepMoving) {
        moveForward();
        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming
messages
        rate.sleep();
    }
}
```

run_stopper.cpp

```cpp
#include "Stopper.h"

int main(int argc, char **argv) {
    // Initiate new ROS node named "stopper"
    ros::init(argc, argv, "stopper");

    // Create new stopper object
    Stopper stopper;

    // Start the movement
    stopper.startMoving();

    return 0;
};
```

CMakeLists.txt

- Edit the following lines in CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_stage)

...

## Declare a cpp executable
add_executable(stopper src/Stopper.cpp src/run_stopper.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(stopper ${catkin_LIBRARIES})
```

Stopper Output

Problems Tasks Console Properties Call Graph

```
<terminated> stopper Configuration [C/C++ Application] /home/roiyeho/catkin_ws/devel/lib/my_stage/stopper (10/25/13, 3:35 AM)
□[0m[ INFO] [1382661340.576015273, 18109.700000000]: Closest range: 0.760001□[0m
□[0m[ INFO] [1382661340.667596025, 18109.800000000]: Closest range: 0.740001□[0m
□[0m[ INFO] [1382661340.768342773, 18109.900000000]: Closest range: 0.720001□[0m
□[0m[ INFO] [1382661340.867396332, 18110.000000000]: Closest range: 0.680001□[0m
□[0m[ INFO] [1382661340.966313085, 18110.100000000]: Closest range: 0.680001□[0m
□[0m[ INFO] [1382661341.067020022, 18110.200000000]: Closest range: 0.660001□[0m
□[0m[ INFO] [1382661341.172239501, 18110.300000000]: Closest range: 0.640001□[0m
□[0m[ INFO] [1382661341.269401730, 18110.400000000]: Closest range: 0.620001□[0m
□[0m[ INFO] [1382661341.371178013, 18110.500000000]: Closest range: 0.600001□[0m
□[0m[ INFO] [1382661341.465327863, 18110.600000000]: Closest range: 0.580001□[0m
□[0m[ INFO] [1382661341.565325407, 18110.700000000]: Closest range: 0.540001□[0m
□[0m[ INFO] [1382661341.668388784, 18110.800000000]: Closest range: 0.520001□[0m
□[0m[ INFO] [1382661341.771353545, 18110.900000000]: Closest range: 0.500001□[0m
□[0m[ INFO] [1382661341.868324993, 18111.000000000]: Closest range: 0.500001□[0m
□[0m[ INFO] [1382661341.967389002, 18111.100000000]: Closest range: 0.480001□[0m
□[0m[ INFO] [1382661341.967489121, 18111.100000000]: Stop!□[0m
```

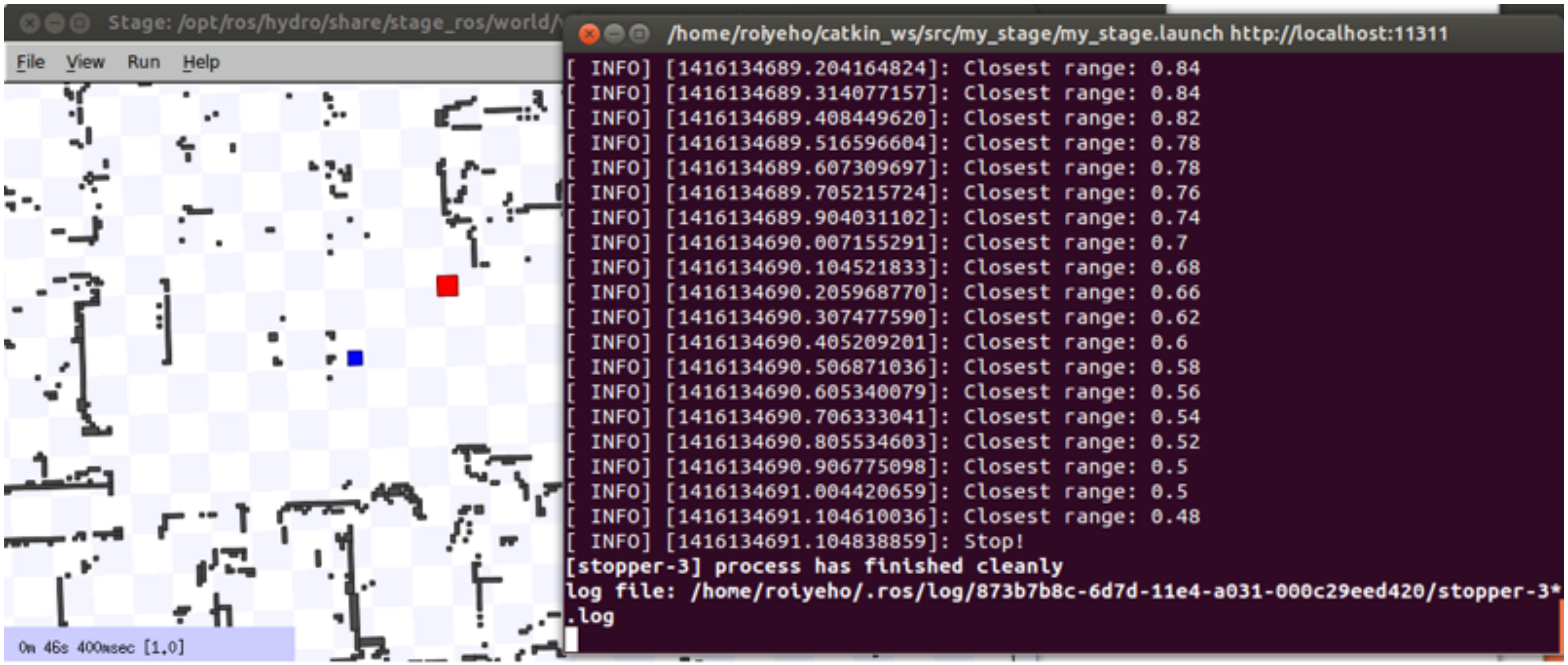
Launch File

- Launch file for launching both the Stage simulator and the stopper node:

```
<launch>
  <node name="stage" pkg="stage_ros" type="stageros" args="$(find
stage_ros)/world/willow-erratic.world"/>
  <node name="stopper" pkg="my_stage" type="stopper"
output="screen"/>
</launch>
```

```
$ roslaunch my_stage my_stage.launch
```

Launch File



Stage: /opt/ros/hydro/share/stage_ros/world/
File View Run Help
0m 46s 400msec [1.0]
/home/roiyeho/catkin_ws/src/my_stage/my_stage.launch http://localhost:11311
[INFO] [1416134689.204164824]: Closest range: 0.84
[INFO] [1416134689.314077157]: Closest range: 0.84
[INFO] [1416134689.408449620]: Closest range: 0.82
[INFO] [1416134689.516596604]: Closest range: 0.78
[INFO] [1416134689.607309697]: Closest range: 0.78
[INFO] [1416134689.705215724]: Closest range: 0.76
[INFO] [1416134689.904031102]: Closest range: 0.74
[INFO] [1416134690.007155291]: Closest range: 0.7
[INFO] [1416134690.104521833]: Closest range: 0.68
[INFO] [1416134690.205968770]: Closest range: 0.66
[INFO] [1416134690.307477590]: Closest range: 0.62
[INFO] [1416134690.405209201]: Closest range: 0.6
[INFO] [1416134690.506871036]: Closest range: 0.58
[INFO] [1416134690.605340079]: Closest range: 0.56
[INFO] [1416134690.706333041]: Closest range: 0.54
[INFO] [1416134690.805534603]: Closest range: 0.52
[INFO] [1416134690.906775098]: Closest range: 0.5
[INFO] [1416134691.004420659]: Closest range: 0.5
[INFO] [1416134691.104610036]: Closest range: 0.48
[INFO] [1416134691.104838859]: Stop!
[stopper-3] process has finished cleanly
log file: /home/roiyeho/.ros/log/873b7b8c-6d7d-11e4-a031-000c29eed420/stopper-3*.log

Agenda

- ROS transformation system
- Writing a tf broadcaster
- Writing a tf listener

What is tf?

- A robotic system typically has many coordinate frames that change over **time**, such as a world frame, base frame, gripper frame, head frame, etc.
- tf is a transformation system that allows making computations in one frame and then transforming them to another at any desired point in time
- tf allows you to ask questions like:
  - What is the current pose of the base frame of the robot in the map frame?
  - What is the pose of the object in my gripper relative to my base?
  - Where was the head frame relative to the world frame, 5 seconds ago?

Values of tf

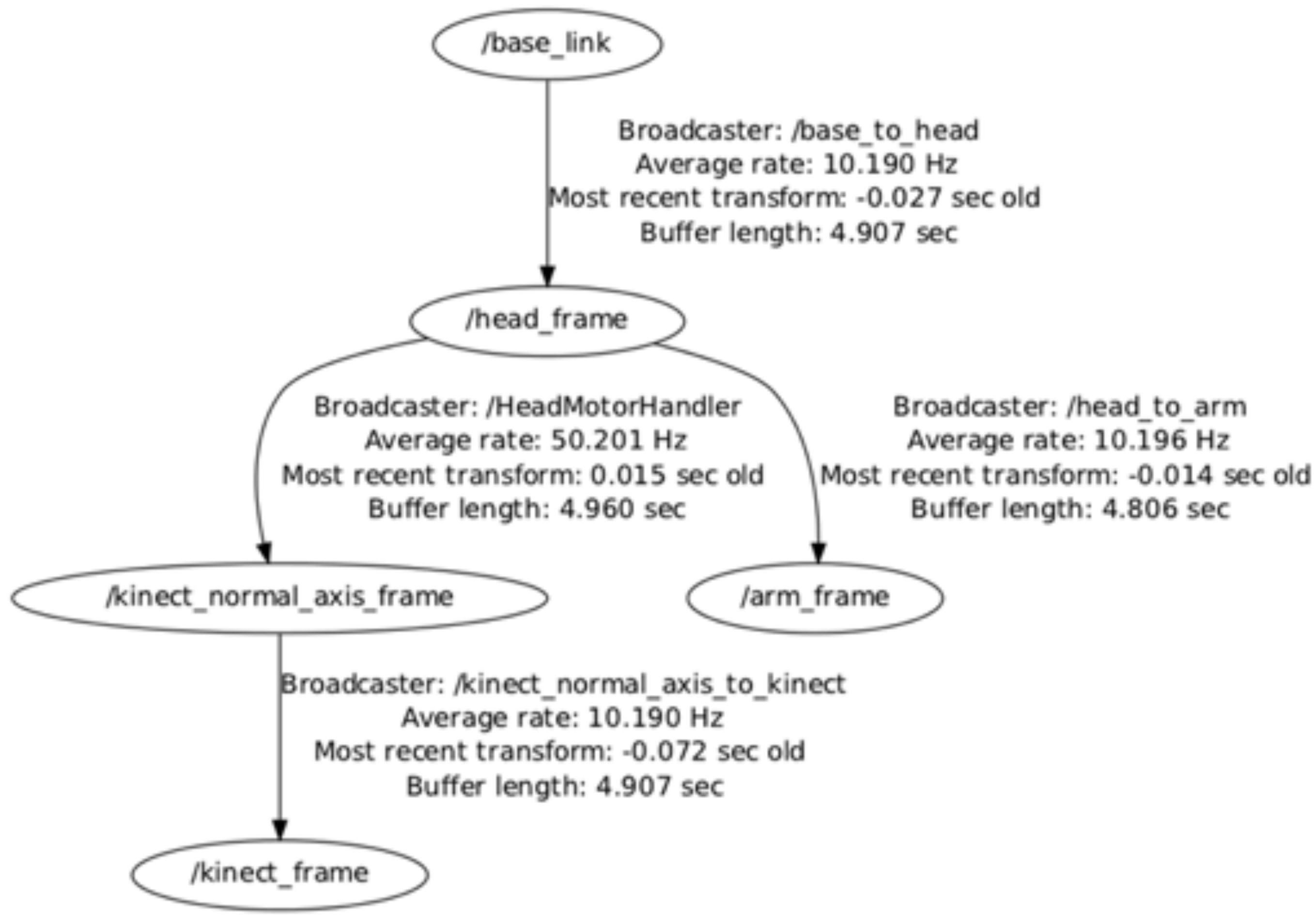
- No data loss when transforming multiple times
- No computational cost of intermediate data transformations between coordinate frames
- The user does not need to worry about which frame their data started
- Information about past locations is also stored and accessible (after local recording was started)

tf Nodes

- There are two types of tf nodes:
  - **Publishers** – publish transforms between coordinate frames on /tf
  - **Listeners** – listen to /tf and cache all data heard up to cache limit
- tf is distributed - there is no central source of tf information

Transform Tree

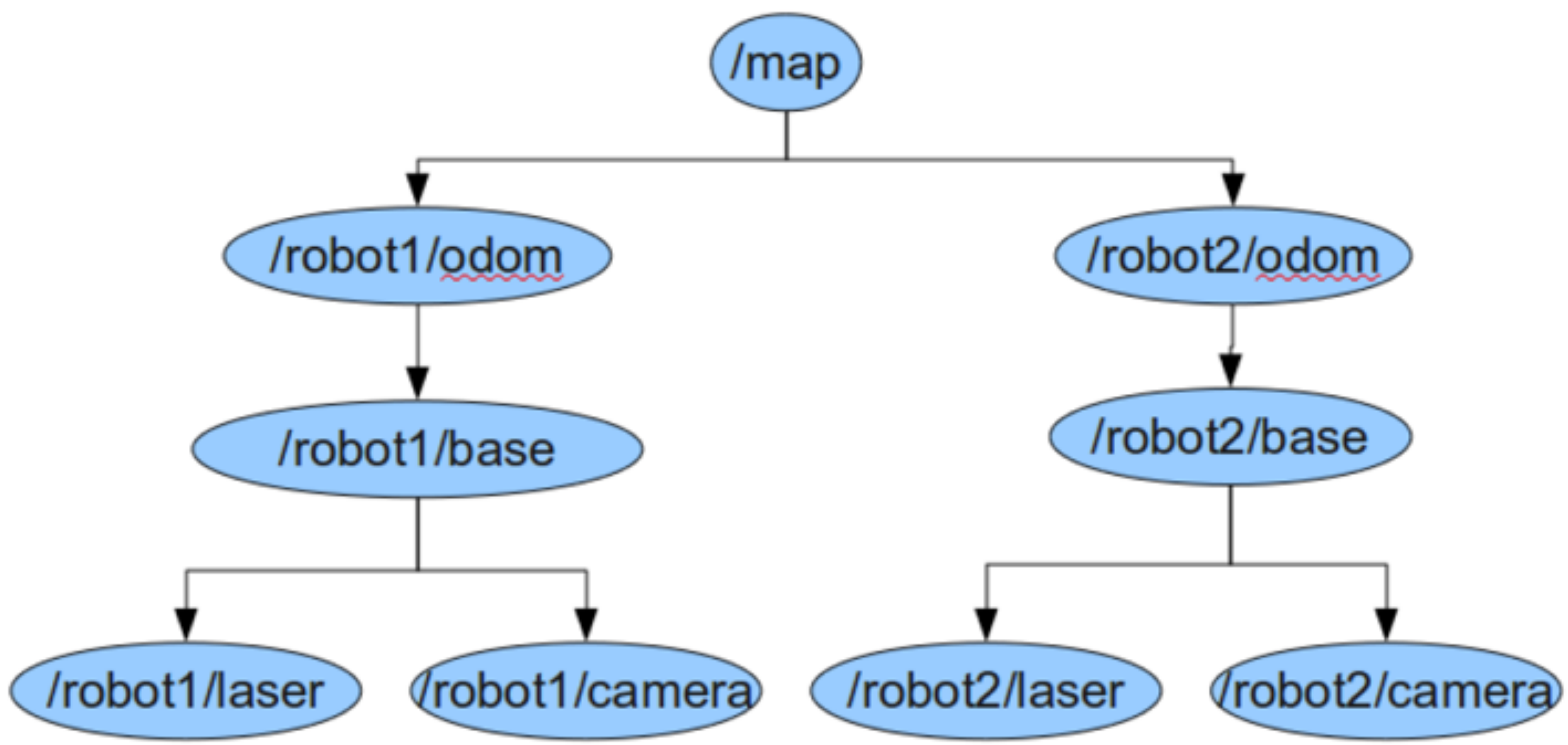
- TF builds a tree of transforms between frames



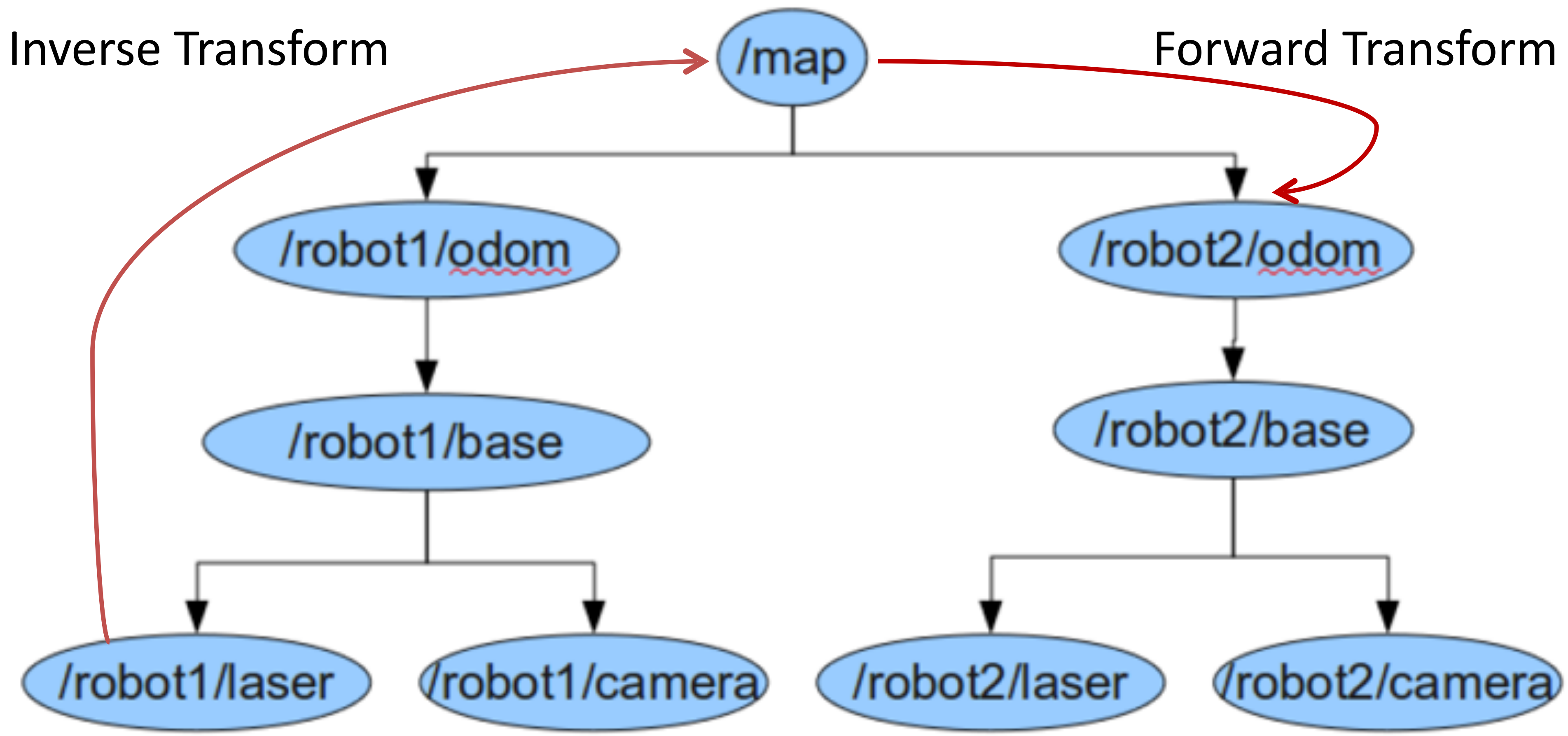
- Can support multiple disconnected trees
- Transforms only work within the same tree

How does this work?

- Given the following TF tree, let's say we want robot2 to navigate based on the laser data coming from robot1



How does this work?



tf Demo

- Launch the turtle_tf_demo by typing:

```
$ roslaunch turtle_tf turtle_tf_demo.launch
```

- In another terminal run the turtle_tf_listener

```
$ rosrun turtle_tf turtle_tf_listener
```

- Now you should see a window with two turtles where one follows the other

- You can drive the center turtle around in the turtlesim using the keyboard arrow keys

```
$ rosrun turtlesim turtle_teleop_key
```

tf Demo

tf Demo

- This demo is using the tf library to create three coordinate frames: a world frame, a turtle1 frame, and a turtle2 frame.
- It uses a **tf broadcaster** to publish the turtle coordinate frames and a **tf listener** to compute the difference in the turtle frames and move one turtle to follow the other

tf Command-line Tools

- view_frames: visualizes the full tree of coordinate transforms
- tf_monitor: monitors transforms between frames
- tf_echo: prints specified transform to screen
- roswtf: with the tfwtf plugin, helps you track down problems with tf
- static_transform_publisher is a command line tool for sending static transforms

view_frames

- view_frames creates a diagram of the frames being broadcast by tf over ROS

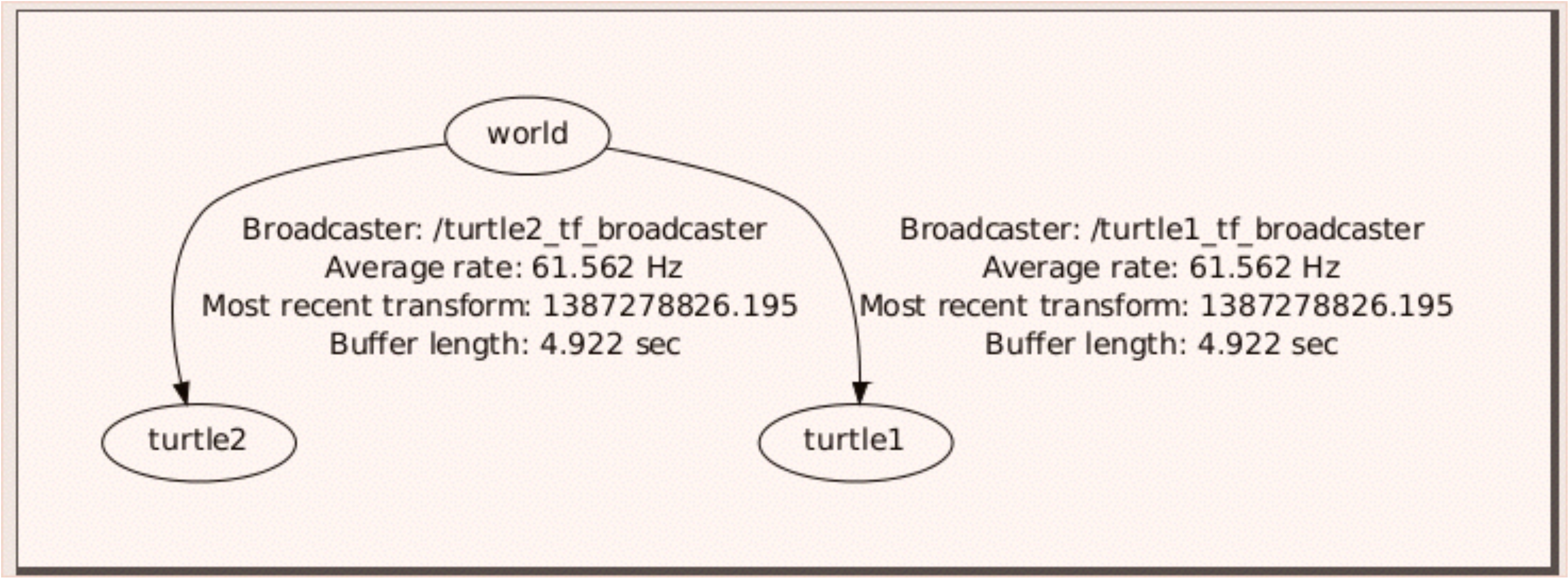
```
roiyeho@ubuntu: ~
roiyeho@ubuntu:~$ view_frames
Please use the package local script inside tf not the global one, this is deprec
ated.
Running [rosrun tf view_frames] for you
Listening to /tf for 5.000000 seconds
Done Listening
dot - graphviz version 2.26.3 (20100126.1600)

Detected dot version 2.26.3
frames.pdf generated
roiyeho@ubuntu:~$
```

- Here a tf listener is listening to the frames that are being broadcast over ROS and drawing a tree of how the frames are connected

view_frames

- To view the tree:

```
$ evince frames.pdf
```



tf_echo

- tf_echo reports the transform between any two frames broadcast over ROS
- Usage:

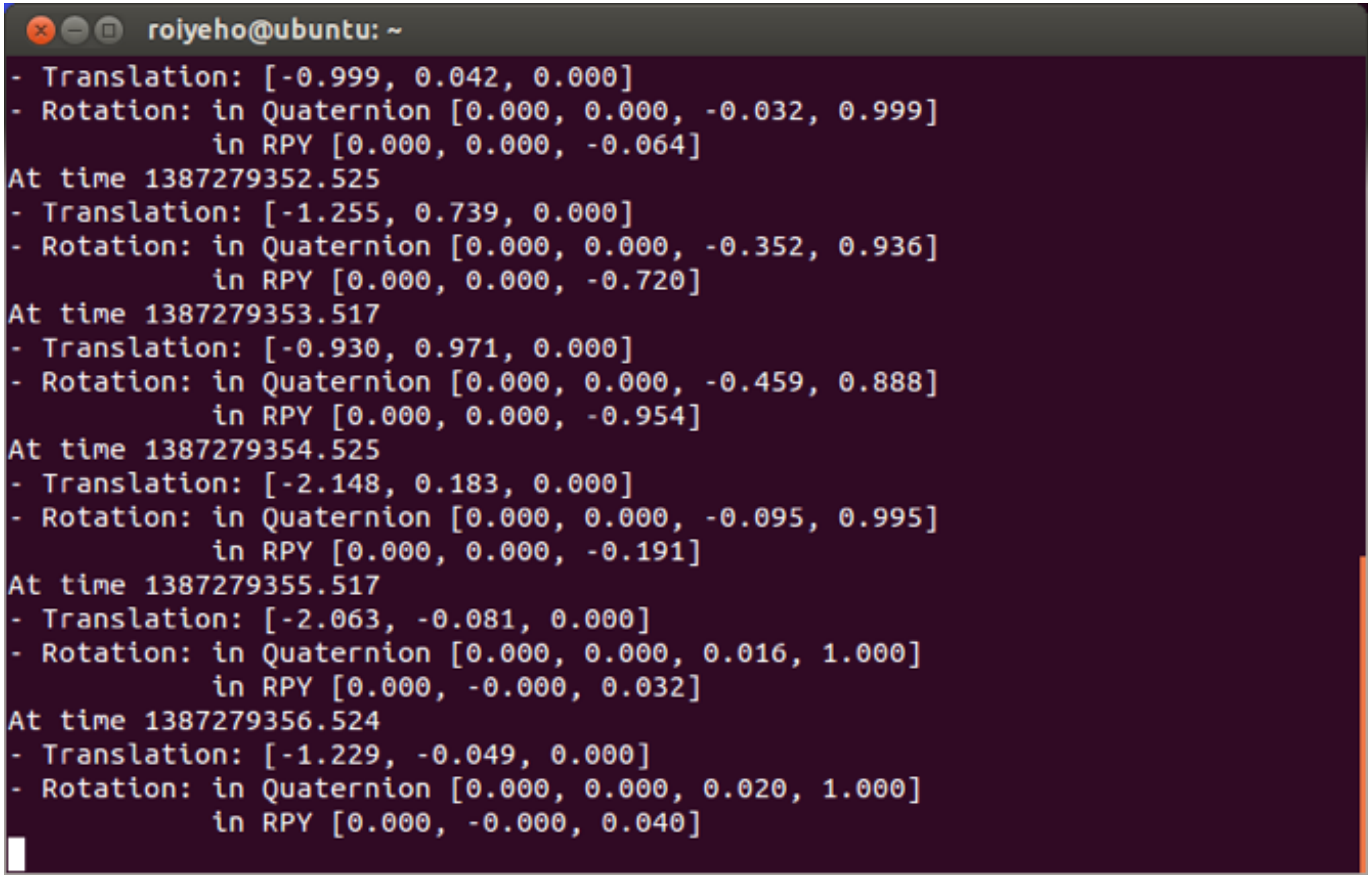
```
$ rosrun tf tf_echo [reference_frame] [target_frame]
```

- Let's look at the transform of the turtle2 frame with respect to turtle1 frame which is equivalent to: $T_{turtle1\_turtle2} = T_{turtle1\_world} * T_{world\_turtle2}$

```
$ rosrun tf tf_echo turtle1 turtle2
```

tf_echo

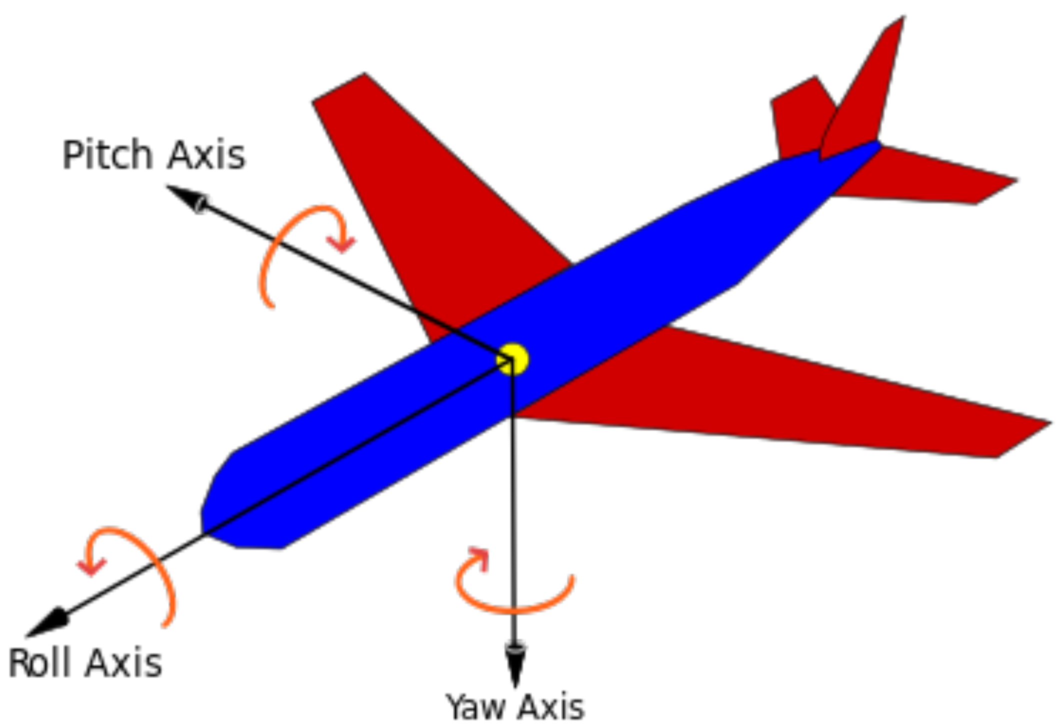
- As you drive your turtle around you will see the transform change as the two turtles move relative to each other

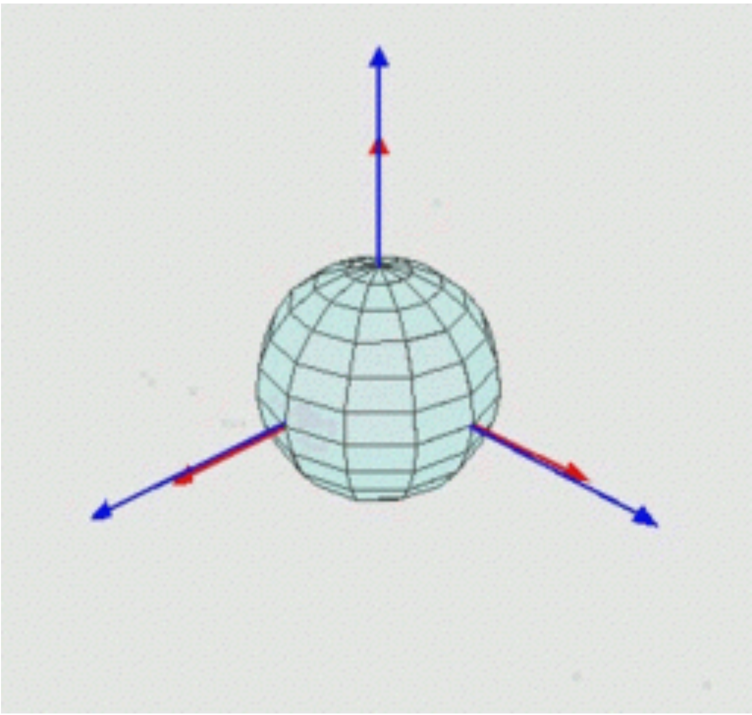



```
- Translation: [-0.999, 0.042, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.032, 0.999]
            in RPY [0.000, 0.000, -0.064]
At time 1387279352.525
- Translation: [-1.255, 0.739, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.352, 0.936]
            in RPY [0.000, 0.000, -0.720]
At time 1387279353.517
- Translation: [-0.930, 0.971, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.459, 0.888]
            in RPY [0.000, 0.000, -0.954]
At time 1387279354.525
- Translation: [-2.148, 0.183, 0.000]
- Rotation: in Quaternion [0.000, 0.000, -0.095, 0.995]
            in RPY [0.000, 0.000, -0.191]
At time 1387279355.517
- Translation: [-2.063, -0.081, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.016, 1.000]
            in RPY [0.000, -0.000, 0.032]
At time 1387279356.524
- Translation: [-1.229, -0.049, 0.000]
- Rotation: in Quaternion [0.000, 0.000, 0.020, 1.000]
            in RPY [0.000, -0.000, 0.040]
```

Rotation Representation

- There are many ways to represent rotations:
  - Euler angles yaw, pitch, and roll about Z, Y, X axes respectively
  - Rotation matrix
  - Quaternions





Quaternions

- In mathematics, quaternions are a number system that extends the complex numbers
- The fundamental formula for quaternion multiplication (Hamilton, 1843):

$$i2 = j2 = k2 = ijk = -1$$

- Quaternions find uses in both theoretical and applied mathematics, in particular for calculations involving 3D rotations such as in computers graphics and computer vision

Quaternions and Spatial Rotation

- Any rotation in 3D can be represented as a combination of a vector u (the Euler axis) and a scalar θ (the rotation angle)
- A rotation with an angle of rotation θ around the axis defined by the unit vector

$$\vec{u} = (u_x, u_y, u_z) = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$$

is represented by

$$\mathbf{q} = e^{\frac{1}{2}\theta(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\tfrac{1}{2}\theta + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\sin\tfrac{1}{2}\theta$$

Quaternions and Spatial Rotation

- Quaternions give a simple way to encode this axis–angle representation in 4 numbers
- Can apply the corresponding rotation to a position vector using a simple formula
  - http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
- Advantages of using quaternions:
  - Nonsingular representation
    - there are 24 different possibilities to specify Euler angles
  - More compact (and faster) than matrices.

tf_monitor

- Print information about the current coordinate transform tree to console

```
$ rosrun tf tf_monitor
```

```
roiyeho@ubuntu: ~
Frames:
Frame: turtle1 published by /turtle1_tf_broadcaster Average Delay: 0.000634343 Max Delay:
 0.00218446
Frame: turtle2 published by /turtle2_tf_broadcaster Average Delay: 0.000537036 Max Delay:
 0.00199225

All Broadcasters:
Node: /turtle1_tf_broadcaster 62.8423 Hz, Average Delay: 0.000634343 Max Delay: 0.0021844
6
Node: /turtle2_tf_broadcaster 62.8577 Hz, Average Delay: 0.000537036 Max Delay: 0.0019922
5



RESULTS: for all Frames

Frames:
Frame: turtle1 published by /turtle1_tf_broadcaster Average Delay: 0.000616328 Max Delay:
 0.00218446
Frame: turtle2 published by /turtle2_tf_broadcaster Average Delay: 0.000519976 Max Delay:
 0.00199225

All Broadcasters:
Node: /turtle1_tf_broadcaster 62.8271 Hz, Average Delay: 0.000616328 Max Delay: 0.0021844
6
Node: /turtle2_tf_broadcaster 62.8339 Hz, Average Delay: 0.000519976 Max Delay: 0.0019922
5
```

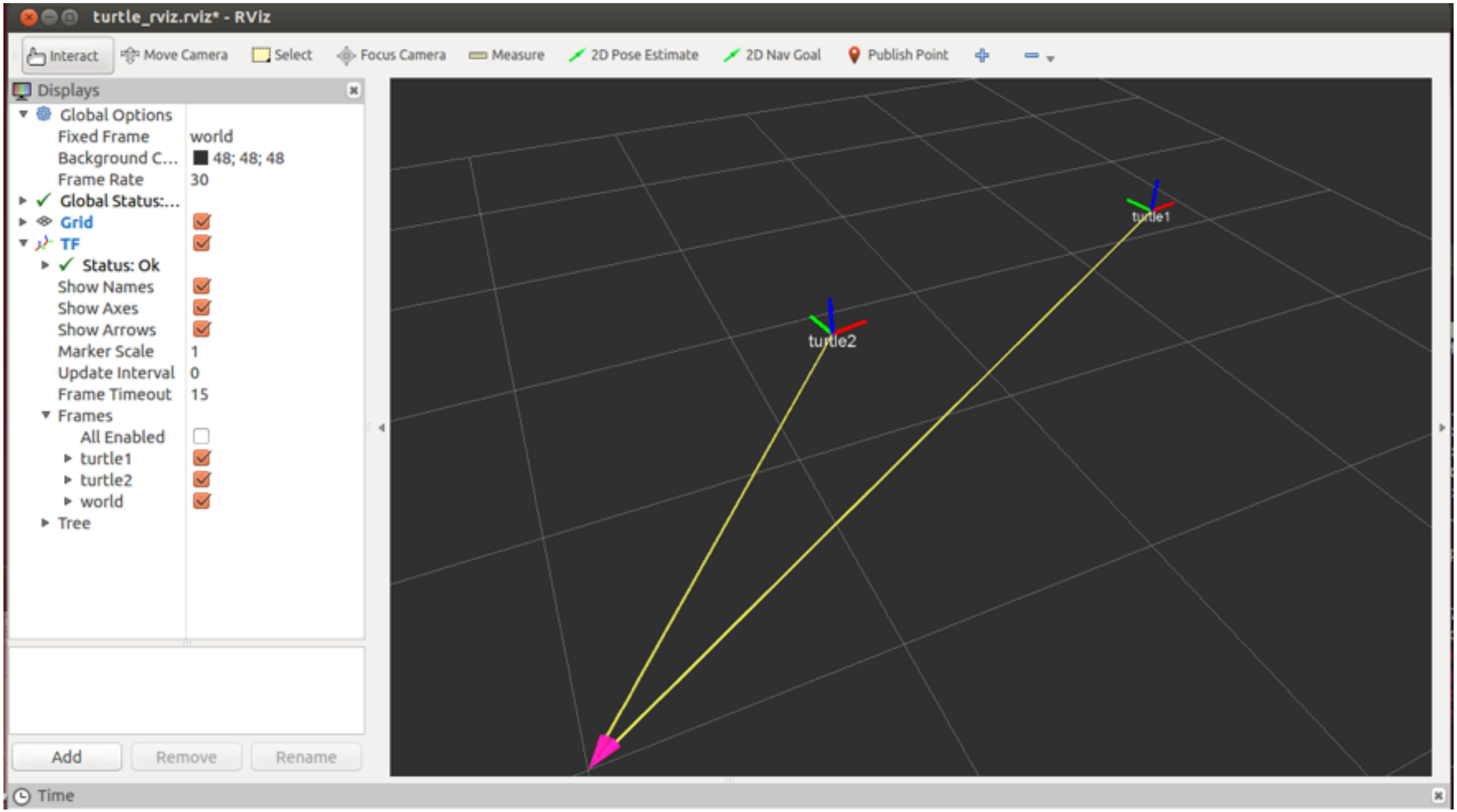
rviz and tf

- Let's look at our turtle frames using rviz
- Let's start rviz with the turtle_tf configuration file using the -d option for rviz:

```
$ rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz
```

- In the side bar you will see the frames broadcast by tf.
- Note that the fixed frame is /world
  - The fixed frame is assumed not to be moving over time
- As you drive the turtle around you will see the frames move in rviz.

rviz and tf



turtle_rviz.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
world
Background C...
48; 48; 48
Frame Rate
30
Global Status:...
Grid
TF
Status: Ok
Show Names
Show Axes
Show Arrows
Marker Scale
1
Update Interval
0
Frame Timeout
15
Frames
All Enabled
turtle1
turtle2
world
Tree
Add
Remove
Rename
Time
turtle1
turtle2

Broadcasting Transforms

- A tf broadcaster sends out the relative pose of coordinate frames to the rest of the system
- A system can have many broadcasters, each provides information about a different part of the robot
- We will now write the code to reproduce the tf demo

Writing a tf broadcaster

- First create a new package called tf_demo that depends on tf, roscpp, rospy and turtlesim

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg tf_demo tf roscpp rospy turtlesim
```

- Build the package by calling catkin_make
- Open the package in Eclipse and add a new source file called tf_broadcaster.cpp

tf_broadcaster.cpp (1)

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>

std::string turtle_name;

void poseCallback(const turtlesim::PoseConstPtr& msg)
{
    static tf::TransformBroadcaster br;
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(msg->x, msg->y, 0.0));
    tf::Quaternion quaternion;
    transform.setRotation(tf::createQuaternionFromYaw(msg->theta));
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world",
turtle_name));
}
```

tf_broadcaster.cpp (2)

```
int main(int argc, char** argv){
    ros::init(argc, argv, "my_tf_broadcaster");
    if (argc != 2) {
      ROS_ERROR("need turtle name as argument");
      return -1;
    };
    turtle_name = argv[1];

    ros::NodeHandle node;
    ros::Subscriber sub = node.subscribe(turtle_name + "/pose", 10, &poseCallback);

    ros::spin();
    return 0;
};
```

Sending Transforms

```
br.sendTransform(tf::StampedTransform(transform,
ros::Time::now(), "world", turtle_name));
```

- Sending a transform with a TransformBroadcaster requires 4 arguments:
  - The transform object
  - A timestamp, usually we can just stamp it with the current time, ros::Time::now()
  - The name of the parent frame of the link we're creating, in this case "world"
  - The name of the child frame of the link we're creating, in this case this is the name of the turtle itself

Running the Broadcaster

- Create tf_demo.launch in the /launch subfolder

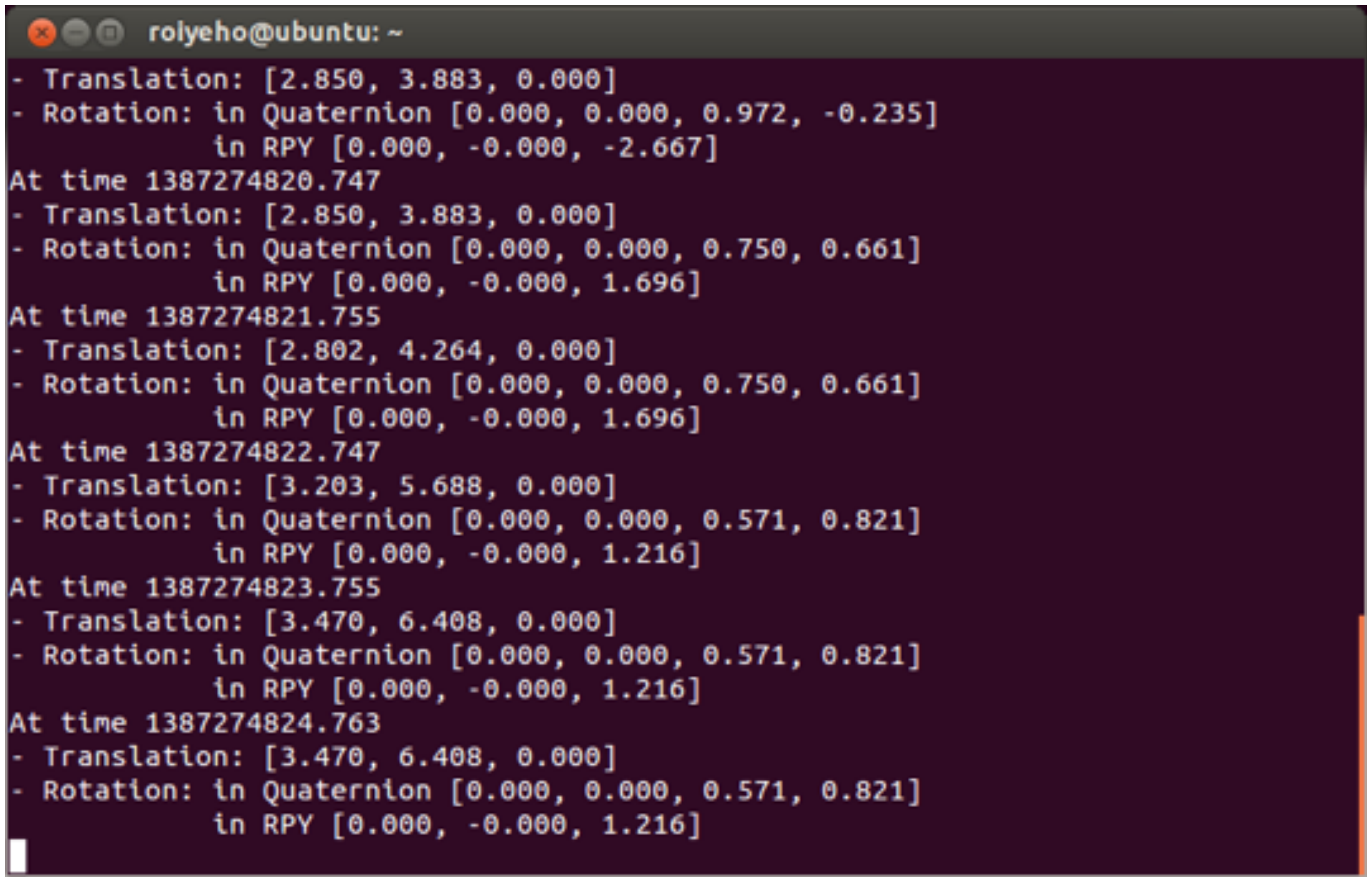
```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <!-- tf broadcaster node -->
  <node pkg="tf_demo" type="turtle_tf_broadcaster"
      args="/turtle1" name="turtle1_tf_broadcaster" />
 </launch>
```

```
$ cd ~/catkin_ws/src
$ roslaunch tf_demo tf_demo.launch
```

Checking the Results

- Use the **tf_echo** tool to check if the turtle pose is actually getting broadcast to tf:

```
$ rosrun tf tf_echo /world /turtle1
```



Writing a tf listener

- A tf listener receives and buffers all coordinate frames that are broadcasted in the system, and queries for specific transforms between frames
- Next we'll create a tf listener that will listen to the transformations coming from the tf broadcaster
- Add tf_listener.cpp to your project with the following code

tf_listener.cpp (1)

```cpp
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <turtlesim/Spawn.h>
#include <geometry_msgs/Twist.h>

int main(int argc, char** argv) {
    ros::init(argc, argv, "my_tf_listener");

    ros::NodeHandle node;

    ros::service::waitForService("spawn");
    ros::ServiceClient add_turtle =
       node.serviceClient<turtlesim::Spawn>("spawn");
    turtlesim::Spawn srv;
    add_turtle.call(srv);

    ros::Publisher turtle_vel =
       node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);

    tf::TransformListener listener;
    ros::Rate rate(10.0);
```

tf_listener.cpp (2)

```
    while (node.ok()) {
        tf::StampedTransform transform;
        try {
            listener.waitForTransform("/turtle2", "/turtle1", ros::Time(0),
ros::Duration(10.0));
            listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0),
transform);
        } catch (tf::TransformException ex) {
            ROS_ERROR("%s",ex.what());
        }

        geometry_msgs::Twist vel_msg;
        vel_msg.angular.z = 4 * atan2(transform.getOrigin().y(),
                                      transform.getOrigin().x());
        vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                       pow(transform.getOrigin().y(), 2));

        turtle_vel.publish(vel_msg);

        rate.sleep();
    }
    return 0;
};
```

Creating a TransformListener

- To use the TransformListener, we need to include the tf/transform_listener.h header file.
- Once the listener is created, it starts receiving tf transformations over the wire, and buffers them for up to 10 seconds.
- The TransformListener object should be scoped to persist otherwise its cache will be unable to fill and almost every query will fail.
    - A common method is to make the TransformListener object a member variable of a class

Core Methods of TransformListener

- LookupTransform
  - Get the transform between two coordinate frames
- WaitForTransform
  - Block until timeout or transform is available
- CanTransform
  - Test if a transform is possible between to coordinate frames

lookupTransform

```
listener.lookupTransform("/turtle2", "/turtle1",
ros::Time(0), transform);
```

- To query the listener for a specific transformation, you need to pass 4 arguments:
  - We want the transform from this frame ...
  - ... to this frame.
  - The time at which we want to transform. Providing ros::Time(0) will get us the latest available transform.
  - The object in which we store the resulting transform.

Running the Listener

- Add the following lines to CMakeLists.txt

```
add_executable(tf_listener src/tf_listener.cpp)
target_link_libraries(tf_listener
  ${catkin_LIBRARIES}
)
```

Launch File

- Add the following lines to tf_demo.launch

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <!-- tf broadcaster node -->
  <node pkg="tf_demo" type="tf_broadcaster"
      args="/turtle1" name="turtle1_tf_broadcaster" />

  <!-- Second broadcaster node -->
  <node pkg="tf_demo" type="tf_broadcaster"
      args="/turtle2" name="turtle2_tf_broadcaster" />

  <!-- tf listener node -->
  <node pkg="tf_demo" type="tf_listener" name="listener" />

 </launch>
```

Check the Results

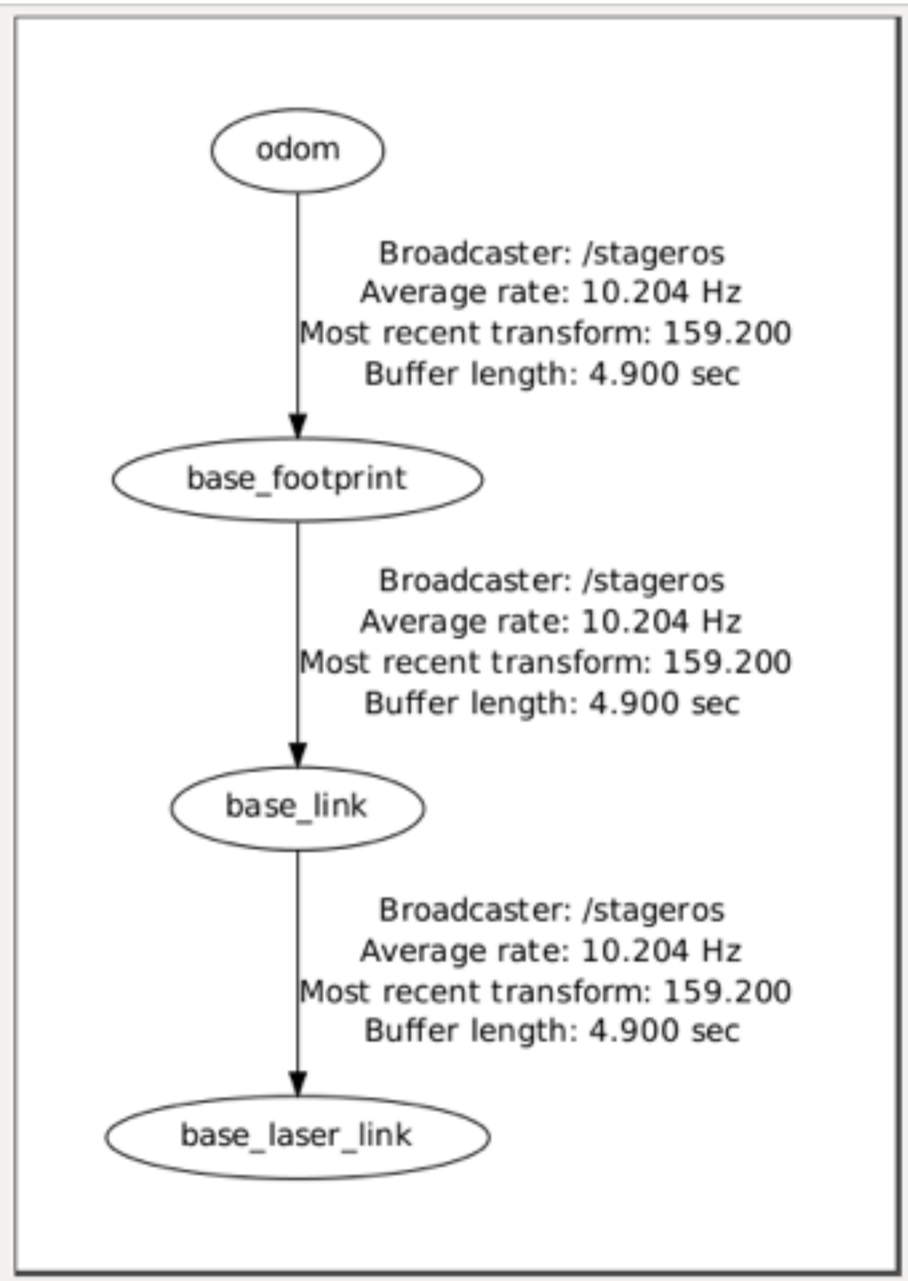
- To see if things work, simply drive around the first turtle using the arrow keys (make sure your terminal window is active, not your simulator window), and you'll see the second turtle following the first one!

lookupTransform Query Examples

- Compute the position of an observed ball in the target frame at the target time assuming it was stationary in the fixed frame
  - lookupTransform(ball_frame, ball_time, target_frame, target_time, fixed_frame, result_transform)
- Compute how far the robot moved between t = 1 and t = 2 in the map frame
  - lookupTransform(robot_frame, t = 1,robot_frame, t = 2, map_frame, result_transform)

Stage TF Frames

- Stage publishes the following TF frames:



- **odom** – The self consistent coordinate frame using the odometry measurements only
- **base_footprint** – The base of the robot at zero height above the ground
- **base_link** – The base link of the robot, placed at the rotational center of the robot
- **base_laser_link** – the location of the laser sensor

Stage TF Frames

- These transformations move relative to the /odom frame
- If we display the robot model in RViz and set the fixed frame to the /odom frame, the robot's position will reflect where the robot "thinks" it is relative to its starting position
- However the robot’s position will not be displayed correctly in relation to the map

Find Robot Location

- You can use tf to determine the robot's current location in the world
- To get robot's location in its own coordinate frame create a TF listener from the /base_footprint to the /odom frame

robot_location.cpp (1)

```
#include <ros/ros.h>
#include <tf/transform_listener.h>

using namespace std;

int main(int argc, char** argv){
    ros::init(argc, argv, "robot_location");
    ros::NodeHandle node;

    tf::TransformListener listener;
    ros::Rate rate(2.0);
    listener.waitForTransform("/base_footprint", "/odom", ros::Time(0),
ros::Duration(10.0));
```

robot_location.cpp (2)

```
    while (ros::ok()){
        tf::StampedTransform transform;
        try {
            listener.lookupTransform("/base_footprint", "/odom", ros::Time(0), transform);
            double x = transform.getOrigin().x();
            double y = transform.getOrigin().y();
            cout << "Current position: (" << x << "," << y << ")" << endl;
        } catch (tf::TransformException &ex) {
            ROS_ERROR("%s",ex.what());
        }
        rate.sleep();
    }

    return 0;
}
```

Static Transform Publisher

- The map → odom transform is published by gmapping
- In case you don't want to use gmapping, you can publish a static transform between these frames to have the robot's position in the map's frame
- To publish a static transformation between the /map and the /odom frames add the following to the launch file:

```
<launch>
  <!-- Publish a static transformation between /map and /odom -->
  <node name="tf" pkg="tf" type="static_transform_publisher" args="-11.28 23.27 0 0
0 0 /odom /map 100" />
</launch>
```

ROS Clock

- Normally, the ROS client libraries will use your computer's system clock as a time source, also known as the "wall-clock“
- When you are running a simulation or playing back logged data, is often desirable to instead have the system use a simulated clock so that you can have accelerated, slowed, or stepped control over your system's perceived time
- To support this, the ROS client libraries can listen to the /clock topic that is used to publish "simulation time"

ROS Clock

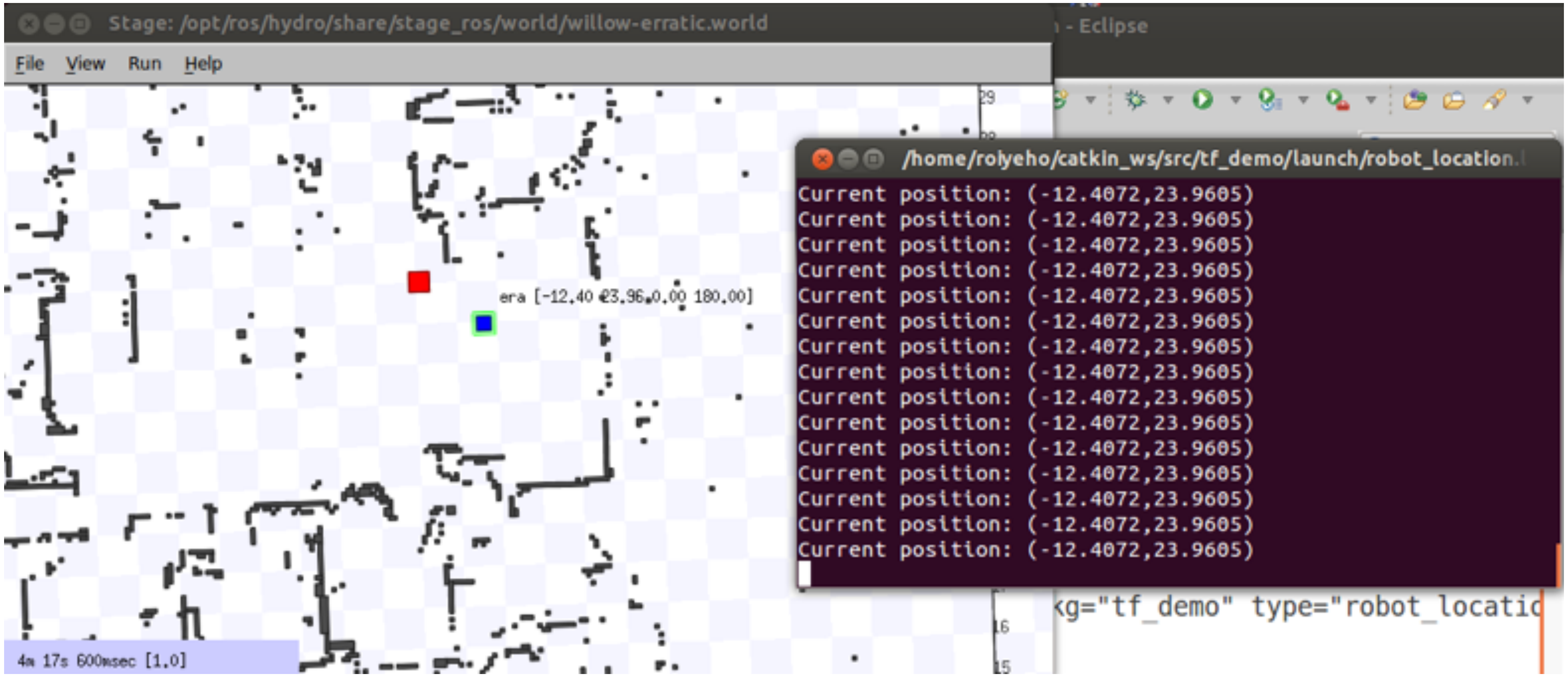
- In order for a ROS node to use simulation time according to the /clock topic, the /use_sim_time parameter must be set to true before the node is initialized
- If the /use_sim_time parameter is set, the ROS Time API will return time=0 until it has received a value from the /clocktopic
- Then, the time will only be updated on receipt of a message from the /clock topic, and will stay constant between updates

Launch File

```
<launch>
  <param name="/use_sim_time" value="true"/>

  <!-- Run stage -->
  <node name="stage" pkg="stage_ros" type="stageros" args="$(find stage_ros)/world/willow-erratic.world"/>

  <!-- Publish a static transformation between /odom and /map -->
  <node name="tf" pkg="tf" type="static_transform_publisher" args="-11.28 23.27 0 0 0 0 /odom /map 100" />

  <!– Run node -->
  <node name="robot_location" pkg="tf_demo" type="robot_location" output="screen" />

</launch>
```

Find Robot Location

- You can now change the TF listener to listen to the transform from the /base_footprint frame to the /map frame to get the robot's location in the map's frame



Watch the TF Frames in rviz

- Type: rosrun rviz rviz
- Add the TF display
- Make sure the Fixed Frame is set to /map

Watch the TF Frames in rviz