ROS Perception in 5 Days

Chapter 4: Face Detection and Tracking

- Summary -

Estimated time to completion: **2 hours**

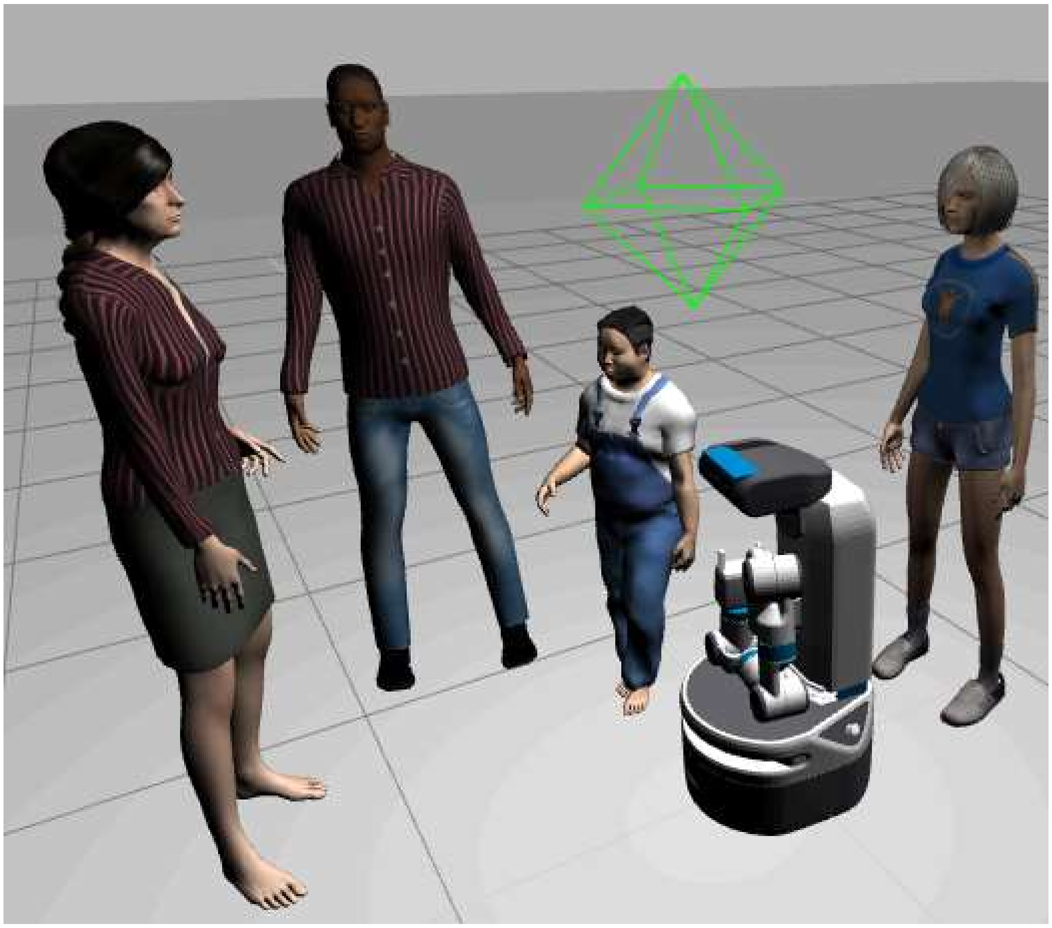
In this chapter, you will learn how to detect human faces and locate them in 3D. Specifically, you will learn about:

- How to detect and locate a human face in 3D space in ROS

- End of Summary -

In [ ]:

```
```



Another vital skill that a robot requires is the ability to work alongside humans. This means that the robot has to detect when a human is around, address the human by looking them in the eye, and recognize them.

There are three primary skills concerning humans that are essential for a robot:

- **Detect faces**: This means that it can detect a human face and address it for talking or giving feedback. The robot can't know who it is; it just knows that it's a human face. This is important for emotion reading, listening to orders, or just giving particular feedback necessary for Human-Robot Interaction (HRI).
- **Recognize faces**: This skill allows the robot to know who it is talking to. This is vital for security purposes or just to understand how to address each individual.
- **Track people**: The robot has to track the movements of humans, follow the one it needs to follow, and filter the others without colliding with them.

These three primary skills will be addressed in the subsequent chapters.

In this chapter, you will learn how to detect faces.

4.1 Face Detection in ROS

You will use the face_detector (http://wiki.ros.org/face_detector#Example_with_robots) package.

4.2 Starting the Face Detector Server

Start the face detector server. For this, you will have to create a new package named **my_face_detector**.

▶ Execute in WebShell #1

In [ ]:
```
roscd;cd ../src
catkin_create_pkg my_face_detector rospy
cd my_face_detector
mkdir launch
touch launch/face_detection_tc.launch
```

Inside the launch folder, create a launch file named **face_detection_tc.launch**, for example. Here you have the contents of that file:

face_detection_tc.launch

In [ ]:

```
<launch>

    <arg name="camera" default="head_camera" />
    <arg name="depth_ns" default="depth_registered" />
    <arg name="image_topic" default="image_raw" />
    <arg name="depth_topic" default="image_raw" />
    <arg name="fixed_frame" default="head_camera_rgb_optical_frame" />
    <arg name="rgb_ns" default="rgb" />


    <node pkg="face_detector" type="face_detector" name="face_detector" output
        <remap from="camera" to="$(arg camera)" />
        <remap from="image_topic" to="$(arg image_topic)" />
        <remap from="depth_topic" to="$(arg depth_topic)" />
        <remap from="depth_ns" to="$(arg depth_ns)" />
        <remap from="rgb_ns" to="$(arg rgb_ns)" />
        <param name="fixed_frame" type="string" value="$(arg fixed_frame)" />

        <param name="classifier_name" type="string" value="frontalface" />
        <rosparam command="load" file="$(find face_detector)/param/classifier.
        <param name="classifier_reliability" type="double" value="0.9"/>
        <param name="do_continuous" type="bool" value="true" />
        <param name="do_publish_faces_of_unknown_size" type="bool" value="fals
        <param name="do_display" type="bool" value="false" />
        <param name="use_rgbd" type="bool" value="true" />
        <param name="approximate_sync" type="bool" value="true" />
    </node>

</launch>
```

There are many things to comment on here:

In [ ]:

```
<arg name="camera" default="head_camera" />
<arg name="depth_ns" default="depth_registered" />
<arg name="image_topic" default="image_raw" />
<arg name="depth_topic" default="image_raw" />
<arg name="fixed_frame" default="head_camera_rgb_optical_frame" />
<arg name="rgb_ns" default="rgb" />
```

The setting of the arguments is vital for this to work. It's divided into basic elements of the topics to use all of the topics you have from the point-cloud camera.

Face detection uses two types of data:

- RGB image: This is published, in this case, in the topic **/head_camera/rgb/image_raw**
- Depth image data: This is published in the topic **/head_camera/depth_registered/image_raw**

These topics, then, have to be divided into the corresponding arguments, resulting in setting them the way mentioned.
The **fixed_frame** selected is the one where the PCL camera is. In this case, it's the **head_camera_rgb_optical_frame.** Normally, it's always the optical frame of the robot that is selected.

Note that you don't need to start the opening server because you are using a simulation. It's already done for you. In the case of the real robot, you will probably have to start it.

In [ ]:

```
<!--include file="$(find openni_launch)/launch/openni.launch"/-->
<!--
<node name="$(anon dynparam)" pkg="dynamic_reconfigure" type="dynparam" args="
<param name="depth_registration" type="bool" value="true" />
</node>
-->
```

Let's start the face_detector:

In [ ]:

```
<node pkg="face_detector" type="face_detector" name="face_detector" output="sc
    <remap from="camera" to="$(arg camera)" />
    <remap from="image_topic" to="$(arg image_topic)" />
    <remap from="depth_topic" to="$(arg depth_topic)" />
    <remap from="depth_ns" to="$(arg depth_ns)" />
    <remap from="rgb_ns" to="$(arg rgb_ns)" />
    <param name="fixed_frame" type="string" value="$(arg fixed_frame)" />

    <param name="classifier_name" type="string" value="frontalface" />
    <rosparam command="load" file="$(find face_detector)/param/classifier.yaml
    <param name="classifier_reliability" type="double" value="0.9"/>
    <param name="do_continuous" type="bool" value="true" />
    <param name="do_publish_faces_of_unknown_size" type="bool" value="false" /
    <param name="do_display" type="bool" value="false" />
    <param name="use_rgbd" type="bool" value="true" />
    <param name="approximate_sync" type="bool" value="true" />
</node>
```

Set the arguments, and then you set some other parameters. You can leave them as they are to detect faces that are facing forward. For more details, refer to http://wiki.ros.org/face_detector (http://wiki.ros.org/face_detector).

Let's launch it and see what happens:

▶ Execute in WebShell #1

In [ ]:
```
cd ~/catkin_ws
source devel/setup.bash
rospack profile
roslaunch my_face_detector face_detection_tc.launch
```

As you have likely seen, although you launch the system, there is no detection. There is nothing really published. That's because the server will only publish if there is a client connected to it. This is common in well-designed servers to avoid overflooding the ROS system with data no one is listening to.

In [ ]:
```
[ INFO] [1501668161.272087967, 446.303000000]: You must subscribe to one of
```

The next step is creating a client to trigger the face_detection.

4.3 Face Detector Client

First, add the line below to the basic **face_detection_tc.launch** to launch your client:

In [ ]:
```
<!-- The  face_detector needs for someone to subscribe to publish data and det
<node pkg="my_face_detector" type="face_detector_client.py" name="face_detecti
</node>
```

In the **my_face_detector** package, create a Python file named **face_detector_client.py**.

▶ Execute in WebShell #1

In [ ]:
```
roscd my_face_detector
mkdir scripts
touch scripts/face_detector_client.py
chmod +x scripts/*.py
```

face_detector_client.py

In [ ]:

```
#!/usr/bin/env python

import rospy
from people_msgs.msg import PositionMeasurementArray

# Move base using navigation stack
class FaceDetectClient(object):

    def __init__(self):
        self.face_detect_subs = rospy.Subscriber(   "/face_detector/people_tra
                                                    PositionMeasurementArray,
                                                    self.face_detect_subs_call



        self.pos_mesurement_array = PositionMeasurementArray()

    def face_detect_subs_callback(self,msg):
        self.pos_mesurement_array = msg


def Face_DetectionClient_Start():
    # Create a node
    rospy.init_node("face_detection_client_start_node")

    # Make sure sim time is working
    while not rospy.Time.now():
        pass

    face_detector_client = FaceDetectClient()

    rospy.spin()

if __name__ == "__main__":
    Face_DetectionClient_Start()
```

As you can see, there is no mystery here. Subscribe to the topic, **/face_detector/people_tracker_measurements_array**, and the face detecting topics will start publishing data.

Let's have a look at this data:

In [ ]:

```
user catkin_ws $ rosmsg show people_msgs/PositionMeasurementArray
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
people_msgs/PositionMeasurement[] people
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  string name
  string object_id
  geometry_msgs/Point pos
    float64 x
    float64 y
    float64 z
  float64 reliability
  float64[9] covariance
  byte initialization
float32[] cooccurrence
```

You should get the position of every face detected, with an ID for each one. This is perfect for tracking many people at the same time.

Visualize the Face Detections

As you may have already guessed, in perception, visualizing the detection data is crucial if the robot is comprehending what's going on around it. That's why you have to use RViz and special markers to indicate where the face detections are made and when they stop.

For this, you will need to copy a pre-made RViz config file that we provide below:

▶ Execute in WebShell #1

In [ ]:

```
roscd face_detection_tc
cp -r rviz_config /home/user/catkin_ws/src/my_face_detector/rviz_config
```

Now, relaunch **face_detection_tc.launch** with the client added:

▶ Execute in WebShell #1

In [ ]:

```
cd ~/catkin_ws
source devel/setup.bash
rospack profile
roslaunch my_face_detector face_detection_tc.launch
```

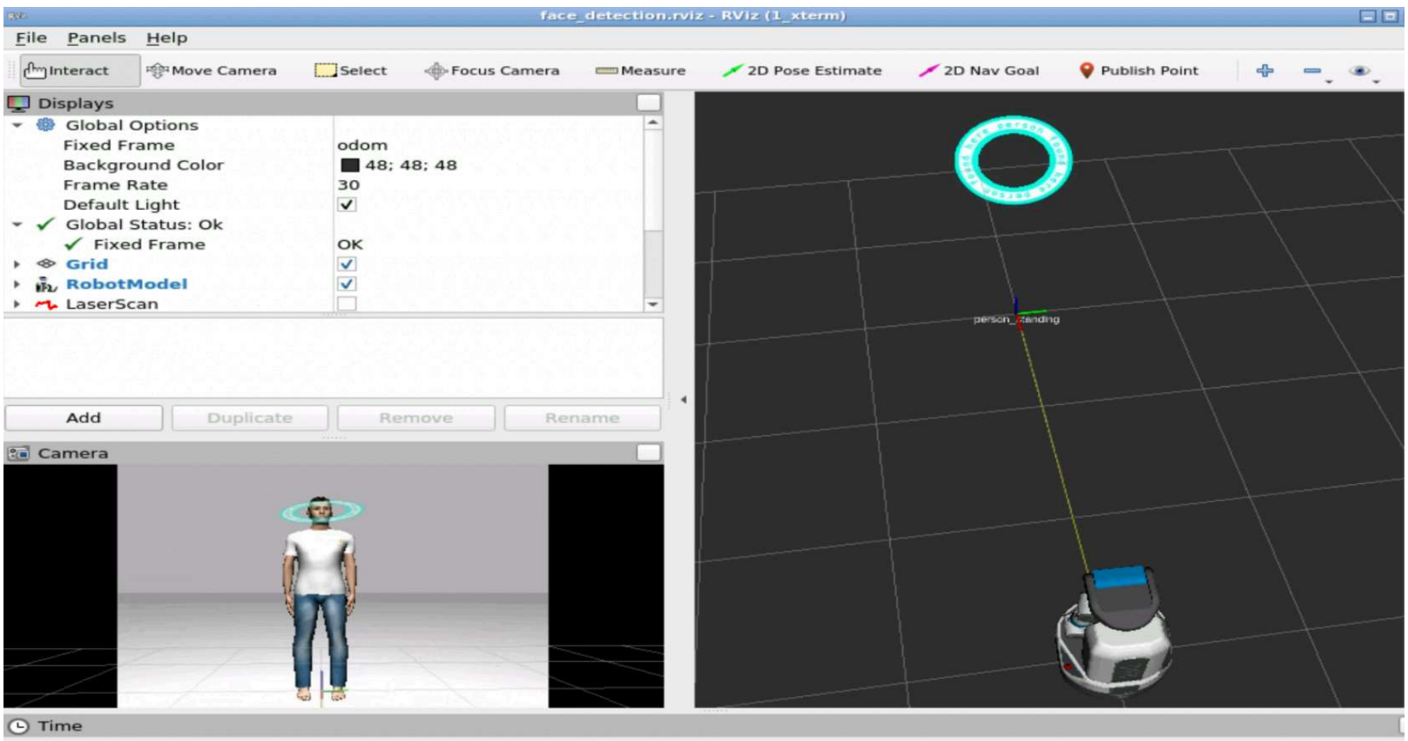
Launch RViz and **open the new RViz config file**. You should see something similar to the image below:

▶ Execute in WebShell #2

In [ ]:

```
rviz
```

NOTE: To work correctly, you need to have the launch file executed in **ExerciseU4-2**, running and working as expected.



This topic representation *(**/face_detector/people_tracker_measurements_array**)* is giving you the position of the face detected. As you can see, it is placed more or less where the person frame TF is.

Let's have a look at the topics read for getting this information so that you can reproduce this anywhere:

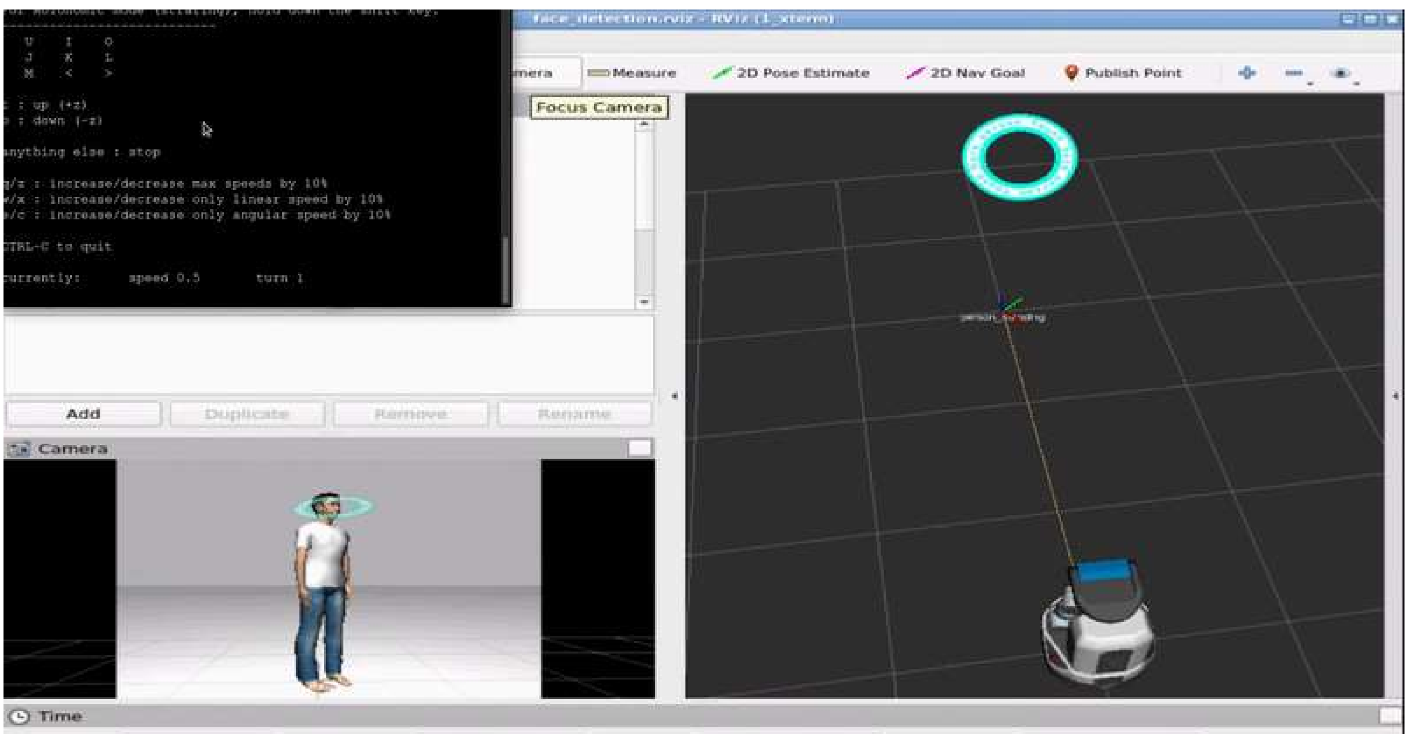
- The first thing to note is that this is not your run-of-the-mill RViz marker. This is because it's using the [jsk_rviz_plugins (http://jsk-visualization.readthedocs.io/en/latest/jsk_rviz_plugins/index.html)](http://jsk-visualization.readthedocs.io/en/latest/jsk_rviz_plugins/index.html) package. You will have to install this package if you want to use it outside the Ignite Academy or RDS environments. These markers give a lot of functionality to RViz, and you could spend a whole course learning all the places where this can be applied. You can have a look at the [ROS RViz Advanced Markers (https://www.robotigniteacademy.com/en/course/ros-rviz-advanced-markers/details/)](https://www.robotigniteacademy.com/en/course/ros-rviz-advanced-markers/details/) course to learn more about RViz markers.
- The LaserScan and the PointCloud2 data are disconnected to avoid overflooding the PC because PCL consumes quite a lot of resources. But you can activate them by simply checking the box.
- **PeoplePositionMeasurementsArray**: This is the blue circle drawn around the position of the face detected. It's the primary data we are looking for.
- Camera: Just the RGB camera, as a reference, where RViz superimposes the PeoplePositionMeasurementsArray data.

Now test how the **face_detector** performs by moving the person using basic keyboard commands:

Execute in WebShell #2

▶ **Execute in WebShell #3**

In [ ]:
```
roslaunch person_sim move_person_standing.launch
```



Project

Select the project unit for this course. You can now do the exercise for detection of faces.

END Project