2nd HRI Summer School

From a video stream to attention assessment

August 25, 2015

Description

Assessing in real-time the focus of attention of a human interacting with a robot is essential to understand implicit references in a dialogue ("robot, take that!"), to measure engagement (is the human "with me"?) or to detect outright problems (why is this human staring at me since 20 minutes?)

With a regular camera, a bit of math and a good face detector, we can actually estimate pretty accurately and in real-time the 3D head pose of surrounding humans. Combined with some frames' magic, this lets us assess what the human is looking at: his/her ***Visual Focus of Attention*** or VFOA.

Objectives

During the workshop, we will:

- use a state-of-art open-source face detector (coming for the `dlib` library) and a PnP algorithm (from OpenCV) to match a 3D template of a head onto a 2D camera stream,
- export the head pose as a ROS tf frame and visualize the face in RViz,
- write a dedicated ROS node that computes what is seen by the human at a given time,
- test the system in a pre-recorded scenario,

Preliminary steps

We will first make sure everything works well on your computer: `dlib`, `opencv`, your compiler,...

> This assumes, of course, that you have downloaded and installed all the workshop's prerequisites. Otherwise, head to `https://hrisummerschool15.wordpress.com/summer-school-programme/workshops/` to get the list of prerequisites and quickly install them.

Go to `https://github.com/severin-lemaignan/attention-assessment-workshop/` and download the repo as a zip file or clone it with `git`. Extract the content somewhere: this will be our `$PROJECT_ROOT` directory.

Then recursively copy the content of `templates/intro/` to the root directory (`$PROJECT_ROOT`):

```
$ cd $PROJECT_ROOT
$ cp -R templates/intro/* .
```

Copy as well `dlib`'s face model (called `shape_predictor_68_face_landmarks.dat`) to `$PROJECT_ROOT/share` (if you do not have it yet, you can download it from `http://dlib.net/files/shape_predictor_68_face_landmarks.dat.bz2` and then unzip it – attention: 61MB download!)

You should now be able to compile and run the initial template:

```
$ mkdir build && cd build
$ cmake -DDLIB_PATH=<path to dlib> ..
$ make
```

If everything is installed correctly on your system, this should compile without error, and generate a binary called `head_detection`.

> If the linking fails with an error related to `dlib`, open `$DLIB_PATH/dlib/CMakeLists.txt` and add `add_definitions(-fPIC)` around line 20. Then, recompile.

Run it by passing as argument the `dlib`'s trained face model that you copied in `share/`. For instance:

```
$ ./head_detection ../share/shape_predictor_68_face_landmarks.dat
```

This should display the video stream of your webcam and, for each detected face, the contours of the face and the IDs of the 68 features recognized by the model. This is very similar to the face detection example that comes with `dlib`.

As you may have seen, this code is organized into a library (`libhead_pose_estimation.so`) and a small executable linked to this library. We will keep this structure for the remaining of the workshop: all the code related the the 3D pose estimation of the head goes to the library to keep the main executables and ROS nodes simple and small.

Part I

From a 2D face to 3D head pose estimation

To estimate the head pose, we need to fit the 3D model of a reference head to the 2D face(s) on video stream:

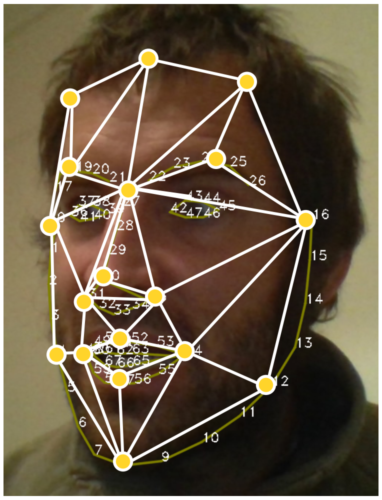


Figure 1: Fitting a 3D model onto a 2D face

Once this is achieved, the translation and rotation of the 3D model (a 4×4 transformation matrix) gives us an estimate of the real head pose of the person ***relative to the camera***.

We actually do not need to fit (***back-project***) many points of the 3D model onto the 2D image to compute this transformation. Theoretically, 3 points are enough. To make it more robust, we can however fit more points.

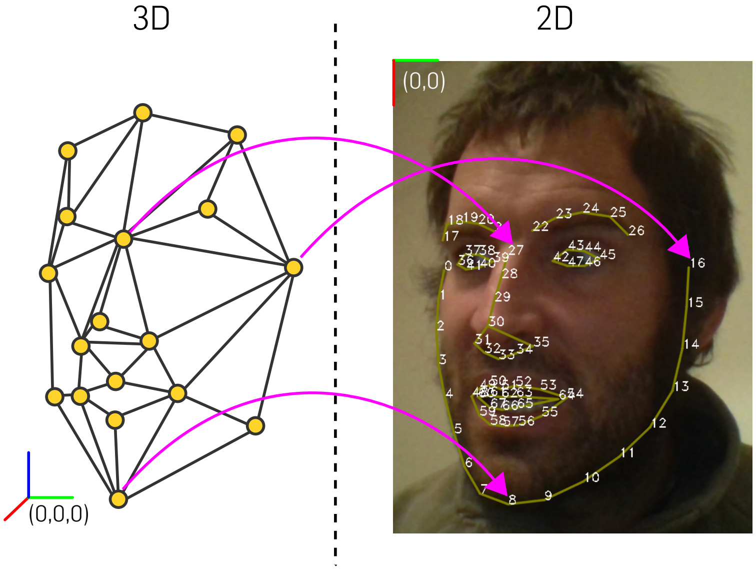


Figure 2: Establishing 2D-3D correspondances

The *fitting* operation is performed with the *PnP* algorithm: it takes 2D-3D correspondances (pairs of points: one on the 3D model and the corresponding one in 2D, figure 2) and iteratively computes the transformation that would project the 3D points onto the corresponding 2D points.

Step 1 – Choosing good correspondances

The first step is to choose good pairs of 2D-3D points. Select 6 to 8 2D points (take their IDs from figure 3), and store the corresponding 3D coordinates.

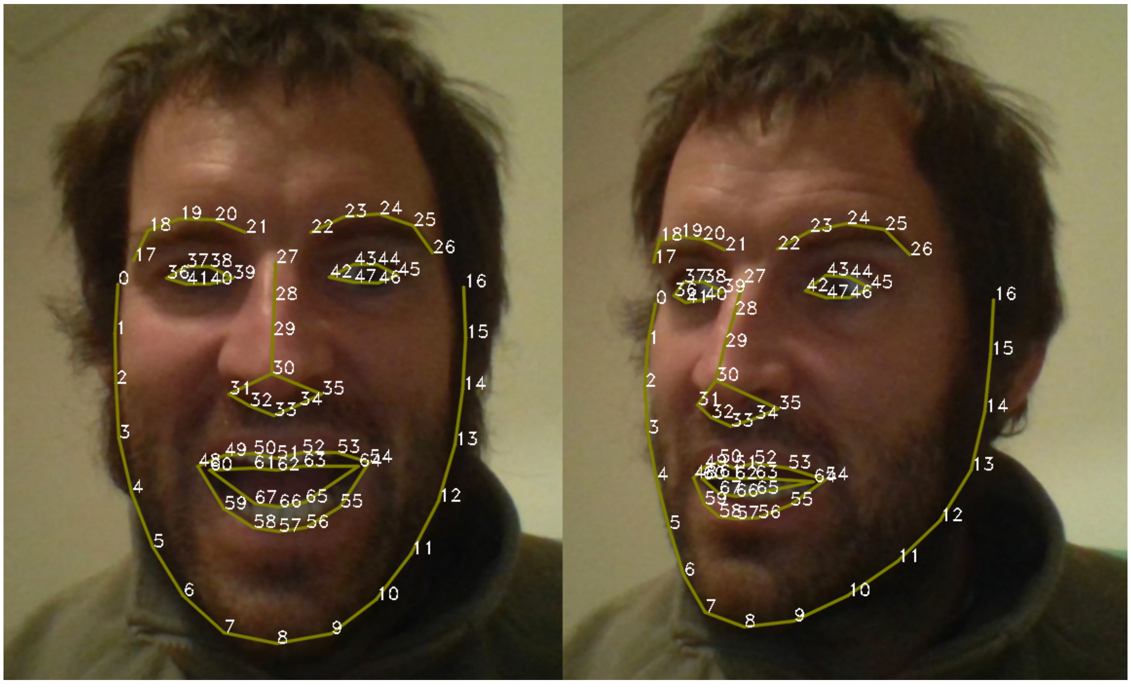
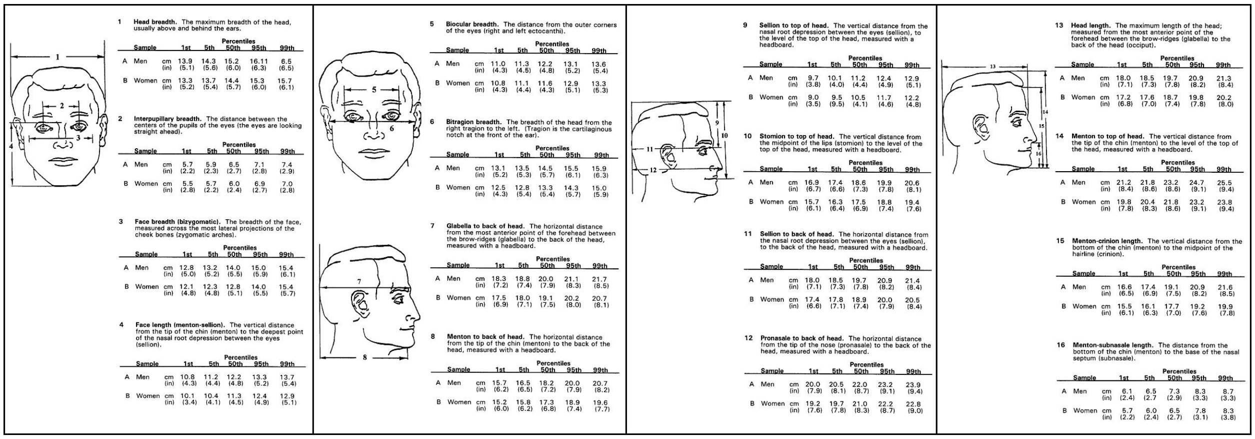


Figure 3: The 68 features detected by `dlib`.

To find the 3D coordinates, decide of an origin on the face (for instance, the nose, or possibly better, the ***sellion*** which lays between the two eyes), and compute the position of the points relative to the nose. You can get the anthropometry of an average adult head from figure 4 (that comes from this wikipedia page: `https://en.wikipedia.org/wiki/Human_head`).

Set X pointing forward and Z pointing up. Use millimeters.

You can simply store them in the library header (`head_pose_estimation.hpp`) as respectively an enumaration and constants:


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

1 **Head breadth.** The maximum breadth of the head, usually above and behind the ears.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	13.9	14.3	15.2	16.11	6.5
		(in)	(5.1)	(5.6)	(6.0)	(6.3)	(6.5)
B	Women	cm	13.3	13.7	14.4	15.3	15.7
		(in)	(5.2)	(5.4)	(5.7)	(6.0)	(6.1)

2 **Interpupillary breadth.** The distance between the centers of the pupils of the eyes (the eyes are looking straight ahead).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	5.7	5.9	6.5	7.1	7.4
		(in)	(2.2)	(2.3)	(2.7)	(2.8)	(2.9)
B	Women	cm	5.5	5.7	6.0	6.9	7.0
		(in)	(2.8)	(2.2)	(2.4)	(2.7)	(2.8)

3 **Face breadth (bizygomatic).** The breadth of the face, measured across the most lateral projections of the cheek bones (zygomatic arches).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	12.8	13.2	14.0	15.0	15.4
		(in)	(5.0)	(5.2)	(5.5)	(5.9)	(6.1)
B	Women	cm	12.1	12.3	12.8	14.0	15.4
		(in)	(4.8)	(4.8)	(5.1)	(5.5)	(5.7)

4 **Face length (menton-sellion).** The vertical distance from the tip of the chin (menton) to the deepest point of the nasal root depression between the eyes (sellion).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	10.8	11.2	12.2	13.3	13.7
		(in)	(4.3)	(4.4)	(4.8)	(5.2)	(5.4)
B	Women	cm	10.1	10.4	11.3	12.4	12.9
		(in)	(3.4)	(4.1)	(4.5)	(4.9)	(5.1)

5 **Biocular breadth.** The distance from the outer corners of the eyes (right and left ectocanthi).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	11.0	11.3	12.2	13.1	13.6
		(in)	(4.3)	(4.5)	(4.8)	(5.2)	(5.4)
B	Women	cm	10.8	11.1	11.6	12.9	13.3
		(in)	(4.3)	(4.4)	(4.3)	(5.1)	(5.3)

6 **Bitragion breadth.** The breadth of the head from the right tragion to the left. (Tragion is the cartilaginous notch at the front of the ear).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	13.1	13.5	14.5	15.5	15.9
		(in)	(5.2)	(5.3)	(5.7)	(6.1)	(6.3)
B	Women	cm	12.5	12.8	13.3	14.3	15.0
		(in)	(4.3)	(5.4)	(5.4)	(5.7)	(5.9)

7 **Glabella to back of head.** The horizontal distance from the most anterior point of the forehead between the brow-ridges (glabella) to the back of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	18.3	18.8	20.0	21.1	21.7
		(in)	(7.2)	(7.4)	(7.9)	(8.3)	(8.5)
B	Women	cm	17.5	18.0	19.1	20.2	20.7
		(in)	(6.9)	(7.1)	(7.5)	(8.0)	(8.1)

8 **Menton to back of head.** The horizontal distance from the tip of the chin (menton) to the back of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	15.7	16.5	18.2	20.0	20.7
		(in)	(6.2)	(6.5)	(7.2)	(7.9)	(8.2)
B	Women	cm	15.2	15.8	17.3	18.9	19.6
		(in)	(6.0)	(6.2)	(6.8)	(7.4)	(7.7)

9 **Sellion to top of head.** The vertical distance from the nasal root depression between the eyes (sellion), to the level of the top of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	9.7	10.1	11.2	12.4	12.9
		(in)	(3.8)	(4.0)	(4.4)	(4.9)	(5.1)
B	Women	cm	9.0	9.5	10.5	11.7	12.2
		(in)	(3.5)	(9.5)	(4.1)	(4.6)	(4.8)

10 **Stomion to top of head.** The vertical distance from the midpoint of the lips (stomion) to the level of the top of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	16.9	17.4	18.6	19.9	20.6
		(in)	(6.7)	(6.6)	(7.3)	(7.8)	(8.1)
B	Women	cm	15.7	16.3	17.5	18.8	19.4
		(in)	(6.1)	(6.4)	(6.9)	(7.4)	(7.6)

11 **Sellion to back of head.** The horizontal distance from the nasal root depression between the eyes (sellion), to the back of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	18.0	18.5	19.7	20.9	21.4
		(in)	(7.1)	(7.3)	(7.8)	(8.2)	(8.4)
B	Women	cm	17.4	17.8	18.9	20.0	20.5
		(in)	(6.6)	(7.1)	(7.4)	(7.9)	(8.4)

12 **Pronasale to back of head.** The horizontal distance from the tip of the nose (pronasale) to the back of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	20.0	20.5	22.0	23.2	23.9
		(in)	(7.9)	(8.1)	(8.7)	(9.1)	(9.4)
B	Women	cm	19.2	19.7	21.0	22.2	22.8
		(in)	(7.6)	(7.8)	(8.3)	(8.7)	(9.0)

13 **Head length.** The maximum length of the head; measured from the most anterior point of the forehead between the brow-ridges (glabella) to the back of the head (occiput).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	18.0	18.5	19.7	20.9	21.3
		(in)	(7.1)	(7.3)	(7.8)	(8.2)	(8.4)
B	Women	cm	17.2	17.6	18.7	19.8	20.2
		(in)	(6.8)	(7.0)	(7.4)	(7.8)	(8.0)

14 **Menton to top of head.** The vertical distance from the tip of the chin (menton) to the level of the top of the head, measured with a headboard.

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	21.2	21.8	23.2	24.7	25.5
		(in)	(8.4)	(8.6)	(8.6)	(9.1)	(9.4)
B	Women	cm	19.8	20.4	21.8	23.2	23.8
		(in)	(7.8)	(8.3)	(8.6)	(9.1)	(9.4)

15 **Menton-crinion length.** The vertical distance from the bottom of the chin (menton) to the midpoint of the hairline (crinion).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	16.6	17.4	19.1	20.9	21.6
		(in)	(6.5)	(6.9)	(7.5)	(8.2)	(8.5)
B	Women	cm	15.5	16.1	17.7	19.2	19.9
		(in)	(6.1)	(6.3)	(7.0)	(7.6)	(7.8)

16 **Menton-subnasale length.** The distance from the bottom of the chin (menton) to the base of the nasal septum (subnasale).

					Percentiles		
Sample			1st	5th	50th	95th	99th
A	Men	cm	6.1	6.5	7.3	8.3	8.7
		(in)	(2.4)	(2.7	(2.9)	(3.3)	(3.3)
B	Women	cm	5.7	6.0	6.5	7.8	8.3
		(in)	(2.2)	(2.4)	(2.7)	(3.1)	(3.8)

Figure 4: Adult human physical characteristics of the head.

```
// Facial features index of interesting points
enum FACIAL_FEATURE {
    NOSE=30,
    RIGHT_EYE=36,
    LEFT_EYE=45,
    // Complete that!
};

// 3D coordinates of these points, assuming here that the nose is the origin
const static cv::Point3f NOSE_3D(0., 0., 0.);
const static cv::Point3f RIGHT_EYE_3D(-41., -65.5, 43.);
// Complete that!
```

Step 2 – From 2D to 3D: back-projection with solvePnP

OpenCV's `solvePnP` method implements the PnP algorithm.

To call it, you need to pass:

- a vector of 3D points
- a vector of the corresponding 2D points
- and a projection matrix

Using the code provided below, modify `src/head_pose_estimation.cpp` to create a new method `cv::Matx44d HeadPoseEstimation::pose(size_t face_idx)` that takes the index of one of the detected faces, call `cv::solvePnP` from it, and finally returns the transformation:

```
cv::Matx44d HeadPoseEstimation::pose(size_t face_idx)
{

    // Very simple projection matrix: we hardcode the focal length
    // to a reasonable value for webcams and the optical center is
    // simply the center of a 640x480px image.
    cv::Mat projectionMat = cv::Mat::zeros(3,3,CV_32F);
    cv::Matx33f projection = projectionMat;
    projection(0,0) = 500; // focal length (x)
    projection(1,1) = 500; // focal length (y)
    projection(0,2) = 320; // optical center (x)
    projection(1,2) = 240; // optical center (y)
    projection(2,2) = 1;

    std::vector<Point3f> head_points_3d;

    // TODO: fill the vector

    std::vector<Point2f> head_points_2d;

    // TODO: fill the vector

    Mat rvec, tvec;

    // TODO: call solvePnP to find the 3D pose of our head

    // transforms the rotation vector rvec into a rotation matrix
    Matx33d rotation;
    Rodrigues(rvec, rotation);


    // create the 4x4 transformation matrix (using meters for the translations)
    Matx44d pose = {
        rotation(0,0),    rotation(0,1),    rotation(0,2),    tvec.at<double>(0)/1000,
        rotation(1,0),    rotation(1,1),    rotation(1,2),    tvec.at<double>(1)/1000,
        rotation(2,0),    rotation(2,1),    rotation(2,2),    tvec.at<double>(2)/1000,
                    0,                0,                0,                            1
    };

    return pose;
}

/** Small helper method to extract the 2D coordinates of a facial feature
 */
Point2f HeadPoseEstimation::coordsOf(size_t face_idx, FACIAL_FEATURE feature)
{
    return toCv(shapes[face_idx].part(feature));
}
```

Step 3 – Display the head pose

Using the provided code, overlay the 3 unit vectors X, Y, Z on top of the face to show the 3D head pose.

```
// creates the 3 unit vectors
std::vector<Point3f> axes;
axes.push_back(Point3f(0,0,0));
axes.push_back(Point3f(50,0,0));
axes.push_back(Point3f(0,50,0));
axes.push_back(Point3f(0,0,50));
std::vector<Point2f> projected_axes;

// the opposite of solvePnP: it simply projects 3D points onto a 2D plan
// Use here the rvec and tvec returned by solvePnP
projectPoints(axes, rvec, tvec, projection, noArray(), projected_axes);

// displays the vectors as well as the coordinates
line(_debug, projected_axes[0], projected_axes[3], Scalar(255,0,0),2,CV_AA);
line(_debug, projected_axes[0], projected_axes[2], Scalar(0,255,0),2,CV_AA);
line(_debug, projected_axes[0], projected_axes[1], Scalar(0,0,255),2,CV_AA);

putText(_debug, "(" + to_string(int(pose(0,3) * 100)) + "cm, "
                    + to_string(int(pose(1,3) * 100)) + "cm, "
                    + to_string(int(pose(2,3) * 100)) + "cm)",
        coordsOf(face_idx, NOSE),
        FONT_HERSHEY_DUPLEX, 0.5, Scalar(0,0,255),2);
```

Modify `head_detection.cpp` to call `HeadPoseEstimation::pose`.

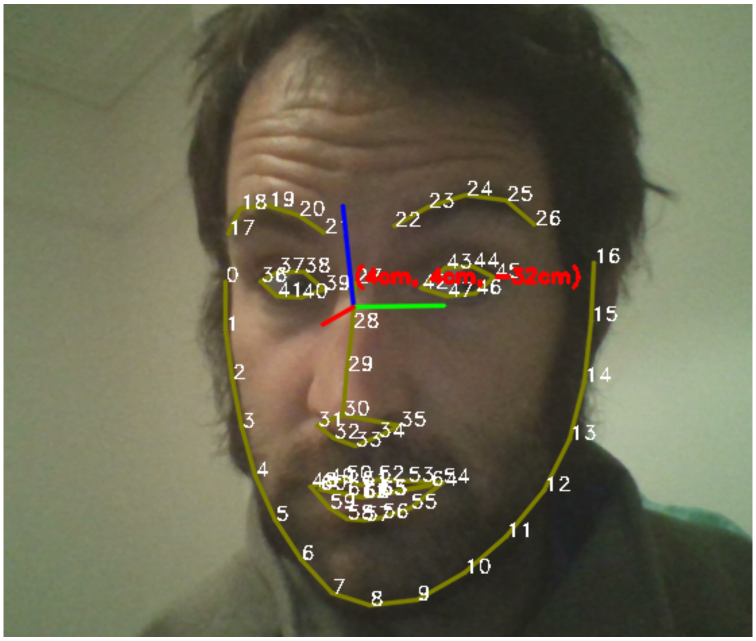


Figure 5: Estimated head pose

You should now obtain something similar to figure 5. We have estimated in real-time the 6D pose of the head using a monocular camera, in about 100 lines of code, and this even works with multiple persons at the same time. Not bad!

Part II

Field of attention

Step 1 – Publishing the head pose as a *tf* frame

The first step of the second part is to “ROSify” the head pose estimation: in our case, it means publishing the head pose as a *tf* frame.

Start by copying the templates from `templates/part2` to `$PROJECT_ROOT`, compile them, and install them.

```
$ cd $PROJECT_ROOT
$ cp -R templates/part2/* .
$ cd build
$ make && make install
```

If ROS is correctly set-up on your system, you should be able to launch the `head_pose_estimator` ROS node with:

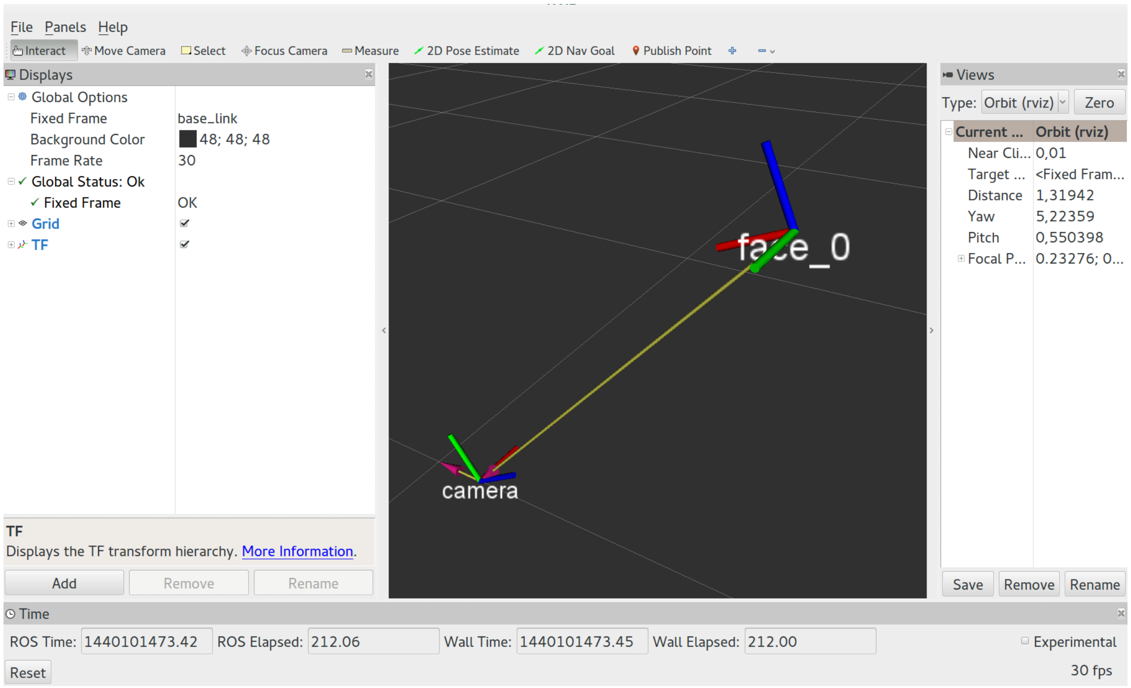
```
$ roslaunch attention_assessment attention_assessment.launch
```

The `head_pose_estimator` node needs a video stream as input. If you look at the launch file (`attention_assessment.launch`), you see that the launch file tries to launch `gscam`. This node allows to stream the webcam or movies as a ROS image topic. Please download and install `gscam` from `https://github.com/ros-drivers/gscam`. Note that this node relies on GStreamer: on Ubuntu, you need to install the package `libgstreamer-plugins-base0.10-dev`.

Once you are able to run the node, complete it to actually broadcast one *tf* frame per detected face.

You can use the following code to convert a `cv::Matx44d` transformation matrix as returned by `HeadPoseEstimation::pose` into a `tf::StampedTransform`:

```
// we assume 'face_idx' to be the index of the currently processed face, and
// 'pose' the corresponding 4x4 transformation matrix

tf::Transform face_pose;

// the frame orientation of the camera follows the classical camera
// convention (Z forward)
auto z = -pose(2,3);

face_pose.setOrigin(tf::Vector3(pose(0,3),
                                pose(1,3),
                                z) );

tf::Quaternion qrot;
tf::Matrix3x3 mrot(
        pose(0,0), pose(0,1), pose(0,2),
        pose(1,0), pose(1,1), pose(1,2),
        pose(2,0), pose(2,1), pose(2,2));
mrot.getRotation(qrot);
face_pose.setRotation(qrot);

auto tf_pose = tf::StampedTransform(face_pose,
                                    ros::Time::now(),
                                    cameraFrame,
                                    "face_" + to_string(face_idx)));
```



Global Options
Fixed Frame base_link
Background Color ■ 48; 48; 48
Frame Rate 30
✓ Global Status: Ok
✓ Fixed Frame OK
Grid ✓
TF ✓

TF
Displays the TF transform hierarchy. More Information.

Type: Orbit (rviz)

Current ...	Orbit (rviz)
Near Cli...	0,01
Target ...	<Fixed Fram...
Distance	1,31942
Yaw	5,22359
Pitch	0,550398
Focal P...	0.23276; 0...

ROS Time: 1440101473.42 ROS Elapsed: 212.06 Wall Time: 1440101473.45 Wall Elapsed: 212.00

30 fps

Figure 6: Screenshot of RViz displaying the 6D pose of a single face.

Once complete, launch your node and start RViz. You should see something similar to figure 6.

Step 2 – Visualization of the field of view and field of attention

The next step consists in computing the field of view (***FoV***) and field of attention (***FoA***) of the person. We model them as cones spanning from the head's origin.

> Attention!
>
> This is a simplified model: obviously, one may move his/her eye balls – hence rotating the field of view – without moving the head. However, this approximation work well enough in many situation. For a more accurate measurement of the direction of the field of view, one may resort to eye-tracking techniques.

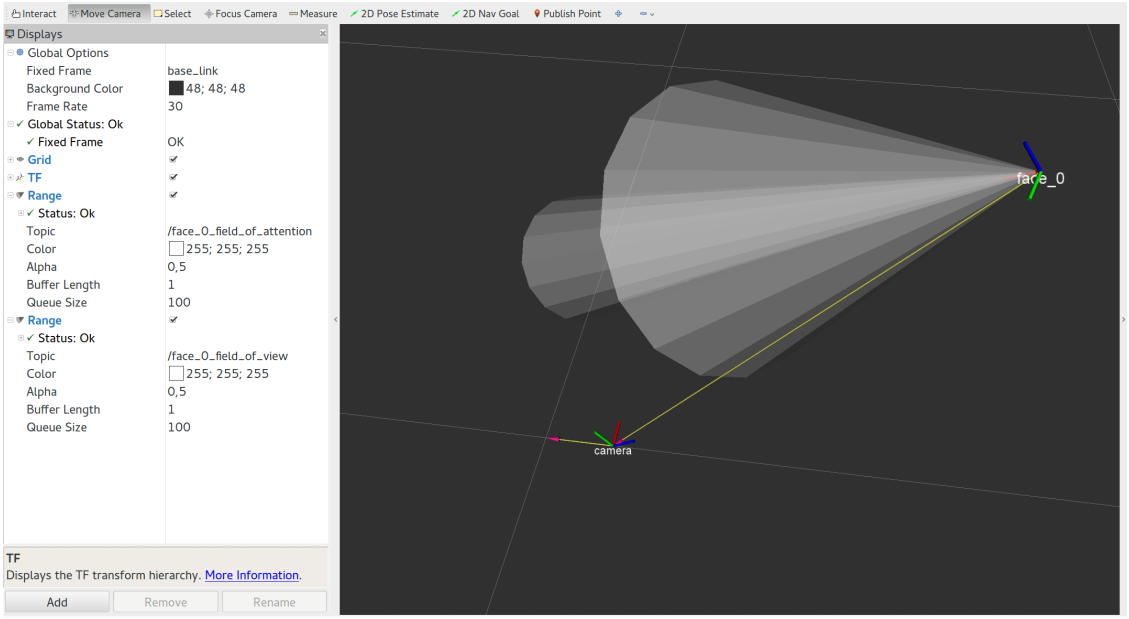
> From now on, we will only consider the **first** detected person, corresponding to the frame `face_0`. Support for multiple people is left as an exercise to the reader :-)

A convenient way to represent the fields of view and fields of attention in RViz is to simulate a range sensor, which is displayed in RViz as a cone.

Modify `ros_node.cpp` to publish the fields of view and field of attention as range sensors, and check you can see them in RViz (figure 7).

You can use the following code snippets to create the publishers (one for the field of view, one for the field of attention) and to actually publish the range sensor messages:

```
// To be added to the node's constructor. Do not forget to add the required
// declarations in the ros_node.hpp header

// Initialization of the FoV and FoA messages and publishing
fov_pub = rosNode.advertise<sensor_msgs::Range>("face_0_field_of_view", 1);
foa_pub = rosNode.advertise<sensor_msgs::Range>("face_0_field_of_attention", 1);

// the Range messages are actually almost constant! We can define them in the
// constructor.
fov.radiation_type = sensor_msgs::Range::INFRARED;
fov.field_of_view = 50. / 180 * M_PI; // radians
fov.min_range = 0;
fov.max_range = 10;
fov.range = 0.5; // meters
```



Global Options	
Fixed Frame	base_link
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	☑
TF	☑
Range	☑
✓ Status: Ok	
Topic	/face_0_field_of_attention
Color	□ 255; 255; 255
Alpha	0,5
Buffer Length	1
Queue Size	100
Range	☑
✓ Status: Ok	
Topic	/face_0_field_of_view
Color	□ 255; 255; 255
Alpha	0,5
Buffer Length	1
Queue Size	100

TF
Displays the TF transform hierarchy. More Information.

Figure 7: Fields of view (large) and attention (narrow) displayed in RViz.

```
foa.radiation_type = sensor_msgs::Range::INFRARED;
foa.field_of_view = 15. / 180 * M_PI; // radians
foa.min_range = 0;
foa.max_range = 10;
foa.range = 1; // meters
```

```
// To be added to the main loop

fov.header.stamp = ros::Time::now();
fov.header.frame_id = "face_0";
fov_pub.publish(fov);

foa.header.stamp = ros::Time::now();
foa.header.frame_id = "face_0";
foa_pub.publish(foa);
```

Step 3 – Who is in the field of attention?

The next step consists in computing who/what is in the field of attention of the person. It simply requires to compute which *tf* frames are within the field of attention.

We implement this computation as a new, independent ROS node, written in Python.

Copy the provided Python template to create a new ROS node:

```
$ cd $PROJECT_ROOT
$ cp -R templates/part2-python .
$ cd build
$ make install
```

This adds a new node called `current_focus`. It has been added to the launch file, so start it by simply calling:

```
$ roslaunch attention_assessment attention_assessment.launch
```

It does not do anything useful for now: complete it to publish on the topic `current_focus` the list of ***tf*** frames that are currently in field of attention of the person.

Part III

Attention assessment

Step 1 – Plotting attention focus

Copy the following Python template for a new ROS node that listen to `current_focus` and plot in real-time the current focus(es):

```
$ cd $PROJECT_ROOT
$ cp -R templates/part3-python .
$ cd build
$ make install
```

This node requires `matplotlib`. Install the package `python-matplotlib` if you do not have it yet.

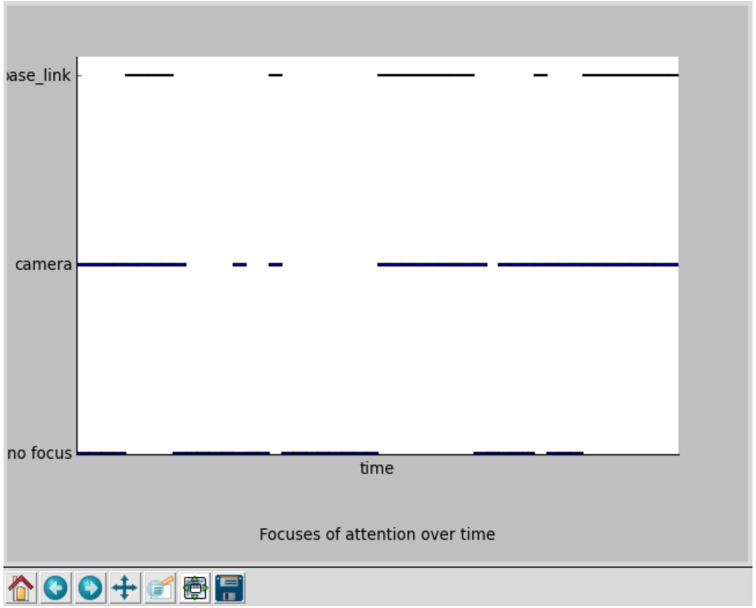


Figure 8: Focuses of attention over time: the lines indicate when the person was looking onto each of the frames.

Complete the node `focus_analysis.py` as needed to plot the frames currently on focus. When running this node, it should look similar to figure 8.

Step 2 – Real-world setup

In a typical experiment, with one or several tasks at hand, the focus of attention is expected to change over time.

To estimate if the user is currently engaged into the task, we therefore need to compare, at a given time, if the current focus of attention matches the ***expected*** focus of attention.

Figure 9 represents an example of a child-robot face-to-face interaction. Depending on the phase of the interaction, the child would have to look either to the robot, to one of the tablets, or to the experimenter. The different possible focuses are depicted in red.

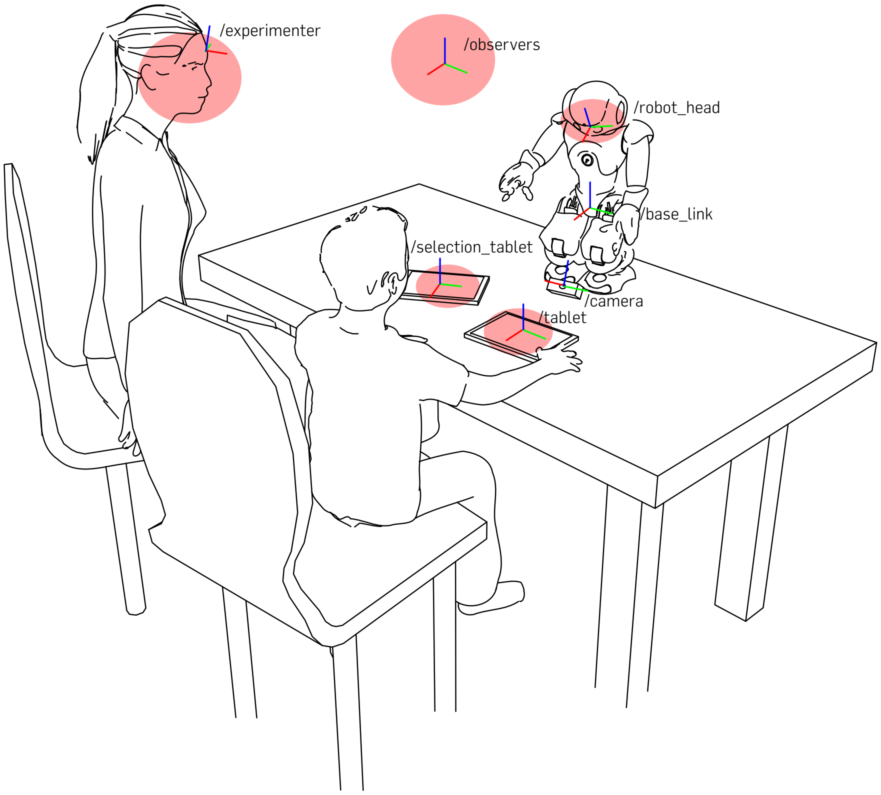


Figure 9: Typical child-robot face-to-face interaction. In red, the different possible focuses of attention: the expected current focus of the child changes over the task's duration.

Using a real recorded experiment and the log of that experiment (which provides the expected focuses of attention over time), we will next try to assess the level of attention of a child during an interaction with a robot.

Figure 9 represents the experimental setup. We will assume that the tablets, the robot and the experimenter did not move during the experiment.

First, we need to publish all the useful frames. Open `launch/attention_assement.launch` and edit it to publish the following static frames:

```
<node name="tablet_transform" pkg="tf" type="static_transform_publisher"
                                               args="0.3 0 0 0 0 0 /base_link /tablet 100"/>
<node name="selection_tablet_transform" pkg="tf" type="static_transform_publisher"
                                  args="0.3 -0.3 0 0 0 0 /base_link /selection_tablet 100"/>
<node name="experimenter_transform" pkg="tf" type="static_transform_publisher"
                                      args="0.5 -1 0.7 0 0 0 /base_link /experimenter 100"/>
<node name="observers_transform" pkg="tf" type="static_transform_publisher"
                                       args="-2.7 -0.5 0.7 0 0 0 /base_link /observer 100"/>
<node name="base_link_transform" pkg="tf" type="static_transform_publisher"
                                                    args="0 0 0 0 0 0 /map /base_link 100"/>
<node name="robot_transform" pkg="tf" type="static_transform_publisher"
                                           args="0 0 0.5 0 0 0 /base_link /robot_head 100"/>
```

Test that the focus tracking and plotting works well with this more complex environment.

Then, download the video from `http://goo.gl/KmA2Qn` (4.4MB), and modify the `gscam` invokation in the launch file to open it instead of the webcam:

```
<param name="gscam_config"
  value="filesrc location=<video path> ! decodebin2 name=dec ! queue ! ffmpegcolorspace"/>
```

Run again the project: this time, the head pose of the child on the video should be tracked and his focus of attention should be plotted.

If you want to see the video stream to ensure the head is correctly tracked, either modify `head_detection.cpp` to open the vide instead of the webcam, or modify the head pose estimation library to display the image. For instance, adding `imshow("headdetection", _debug);waitKey(10);` at the end of `HeadPoseEstimation::pose` in `head_pose_estimation.cpp` will do.

Step 3 – Expected focus of attention

To assess the engagement of the user in the interaction, we want to compare the actual focuses of attention to the expected ones.

The file `share/interaction.log` contains a simplified interaction log, similar to what could be produced by the robot's execution manager.

The file `nodes/log_parsing.py` contains two functions to parse this log file and convert it to a dictionary associating the expected focuses of attention with their times.

If you run it, it should prints in 'real-time' the expected focus of attention.

```
$ cd $PROJECT_ROOT/nodes
$ python log_parsing.py
```

Adapt `focus_analysis.py` to read the log file in parallel to the child's measured focus. Create a new line on the plot that indicates when the ***expected focus*** and the ***actual focus*** match.

Step 4 – Level of attention

Finally, compute a (naive) level of attention by incrementing a variable at every step if the expected/actual focuses match, and decrementing it otherwise. Plot it along the other values.