ROS Perception in 5 Days

PROJECT Hexapod Perception

- Summary -

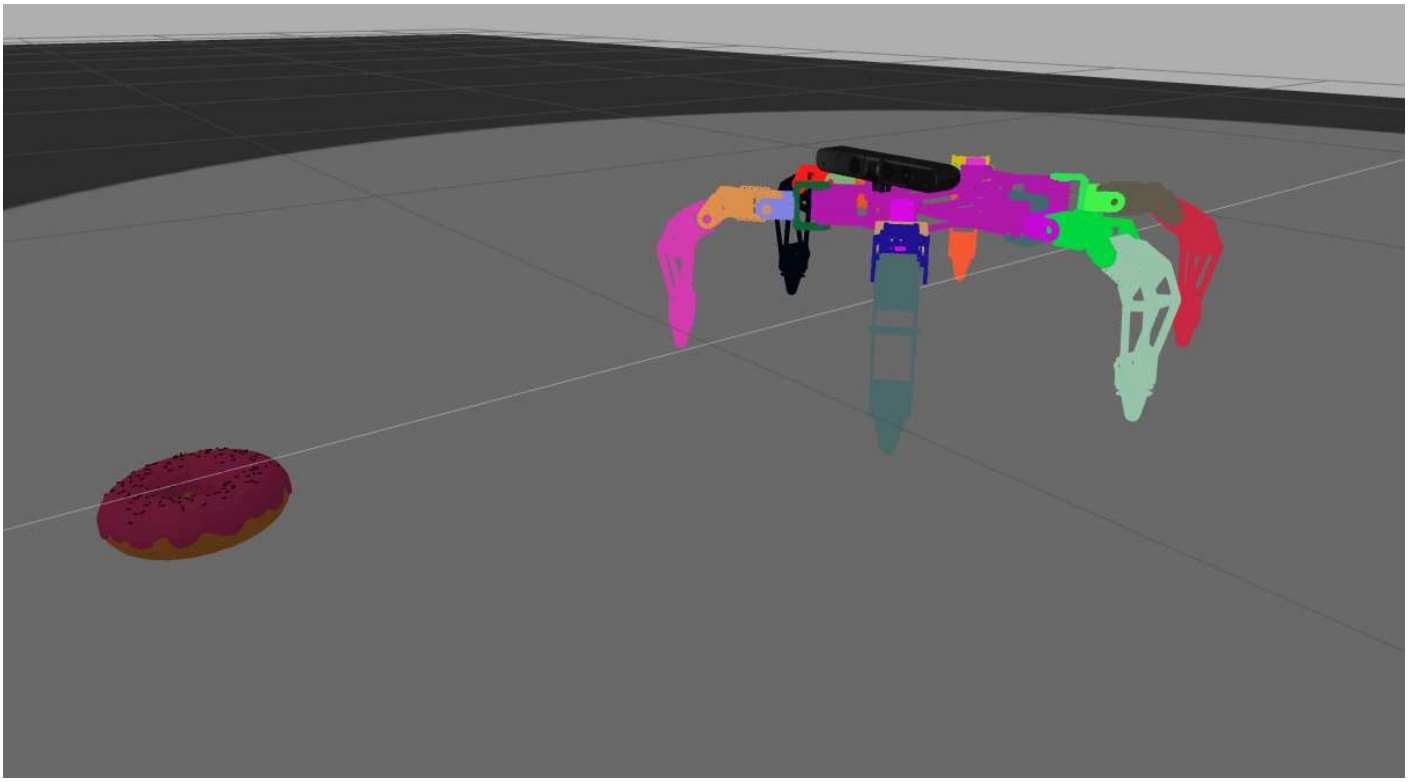
Estimated time to completion: **6 hours**

In this project, you will apply the knowledge acquired throughout this course.

You will be asked to apply the following concepts:

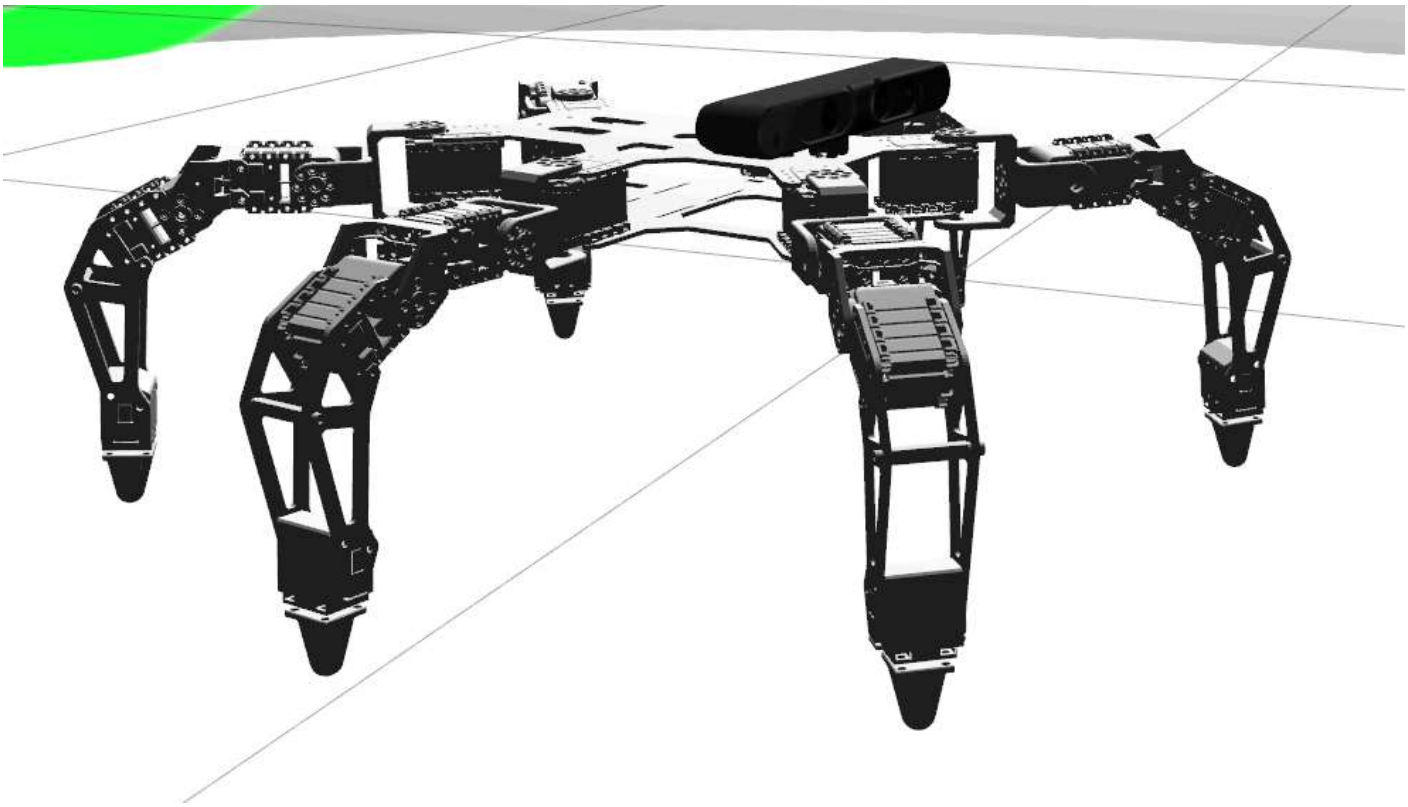
- How to recognize an object based on its histogram
- How to recognize a persons face
- How to detect a persons face
- How to use YOLO to recognize an object

- End of Summary -



In this project you can practice what you have learned in this course with a Hexapod robot with an Xtion RGB-D sensor.

Your Own PhantomX Hexapod

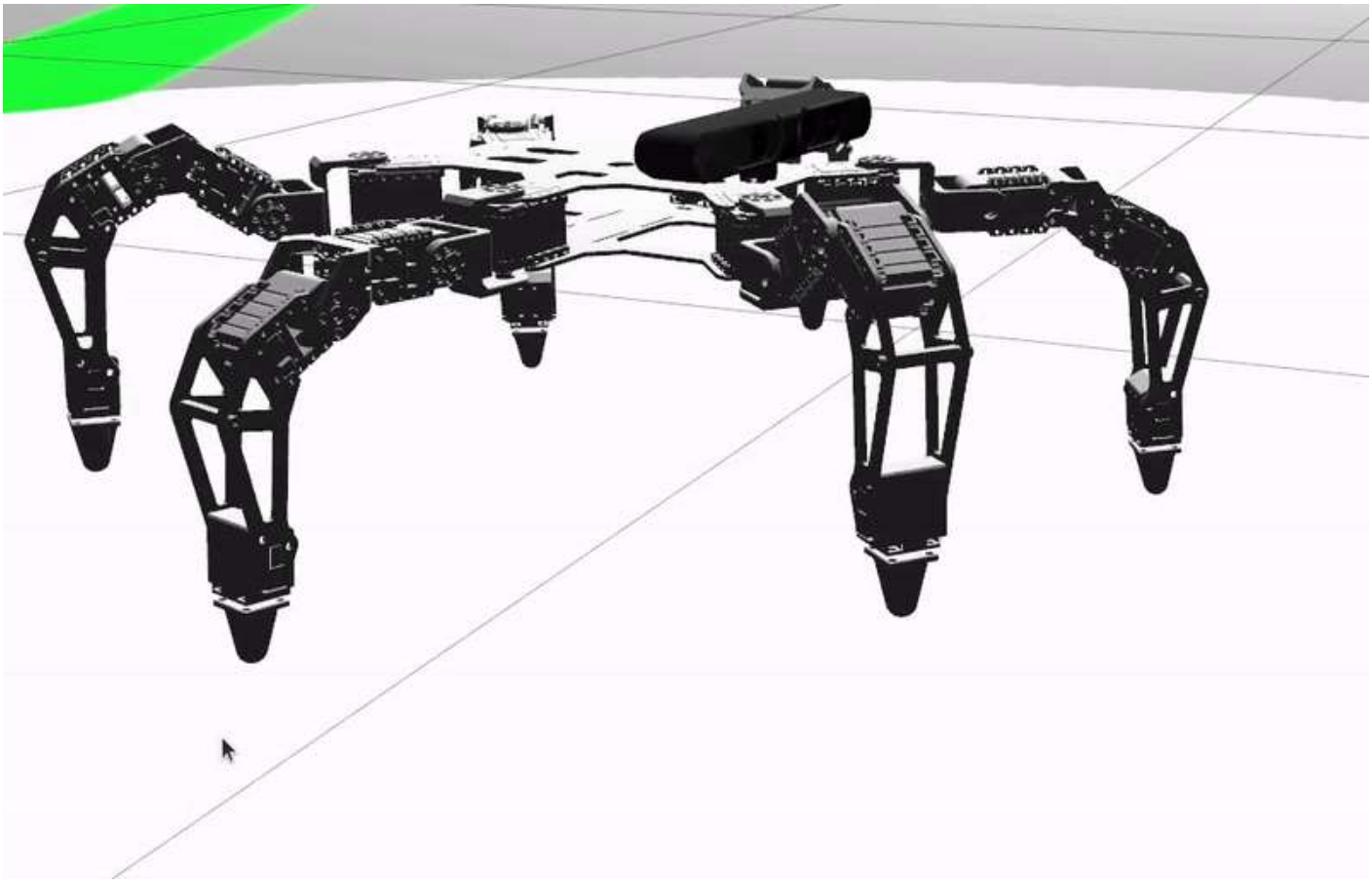


PhantomC Hexapod is a robot kit created by TrossenRobotics (https://www.trossenrobotics.com). It's a versatile Hexapod and perfect for what we intend, a moving robot for doing perception.

To move the simulated Hexapod, publish a **Twist** message in the topic **/phantomx/cmd_vel**.

▶ Execute in Shell #1

In [ ]:
```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/phantomx/cmd_v
```



- This simulated Hexapod has a full Xtion point-cloud sensor. This will allow you to have **RGB images, depth images, and point-cloud data**. This is vital for object detection and people tracking.

- You will have the following main image data topics, plus their different transformations:

In [ ]:
```
# Depth Camera Info
/camera/depth_registered/camera_info
# 2D depth image
/camera/depth_registered/image_raw
# Point Cloud
/camera/depth_registered/points

# RGB camera info
/camera/rgb/camera_info
# RGB camera raw image
/camera/rgb/image_raw
# RGB camera compressed ( faster to process )
/camera/rgb/image_raw/compressed
```

Here you have a brief description of what each one gives as output:

RGB, Depth Images and Point-Cloud

camera_info

The camera info topics give information about all the parameters of the real or simulated camera:

Execute in WebShell #1

```
In [ ]: rostopic info /camera/rgb/camera_info
```

This will give you the message type **sensor_msgs/CameraInfo**. Although you could use the command **rosmsg show** to learn the variables inside the topic, you will get a better explanation of all the variables if you do the following:

Execute in WebShell #1

```
In [ ]: roscd sensor_msgs/msg
```

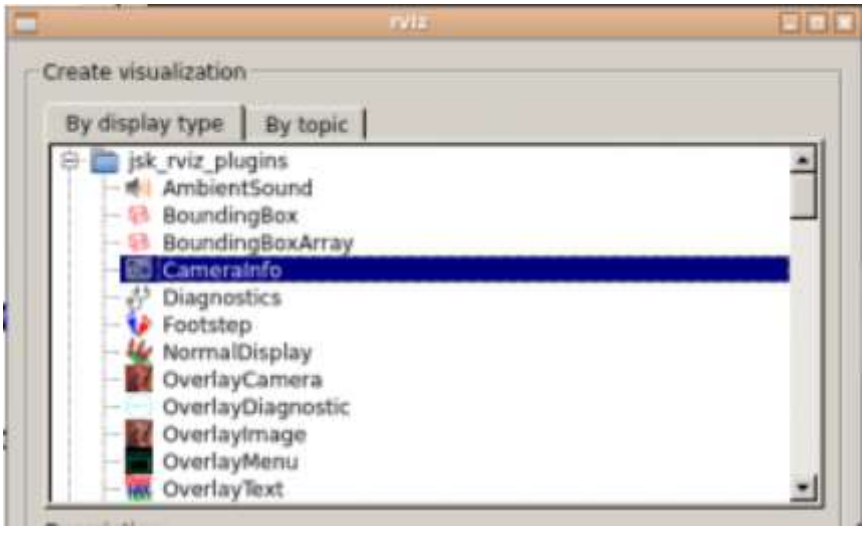
```
In [ ]: vim CameraInfo.msg
```

Doing this will give you a detailed explanation of what each variable is.

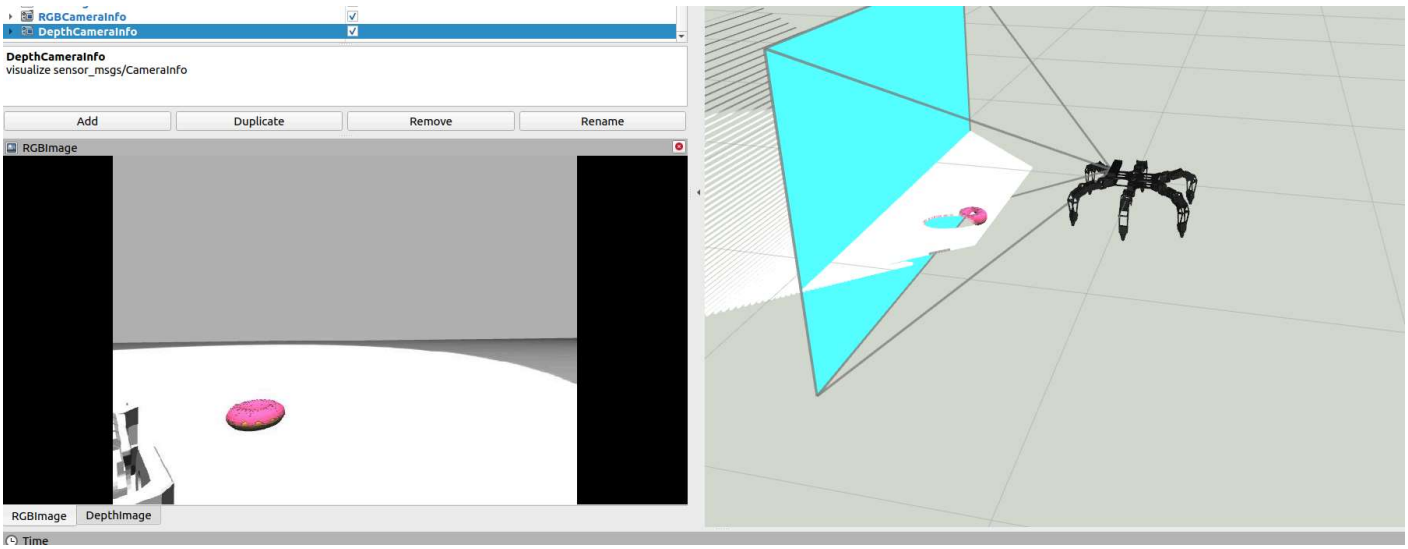
This topic can be used in RViz to draw the optical parameters for where the camera is pointing.

To do so, add the element **CameraInfo** to RViz.

CameraInfo element:



CameraInfo for both depth image and RGB cameras (they have different topics, but the same info):



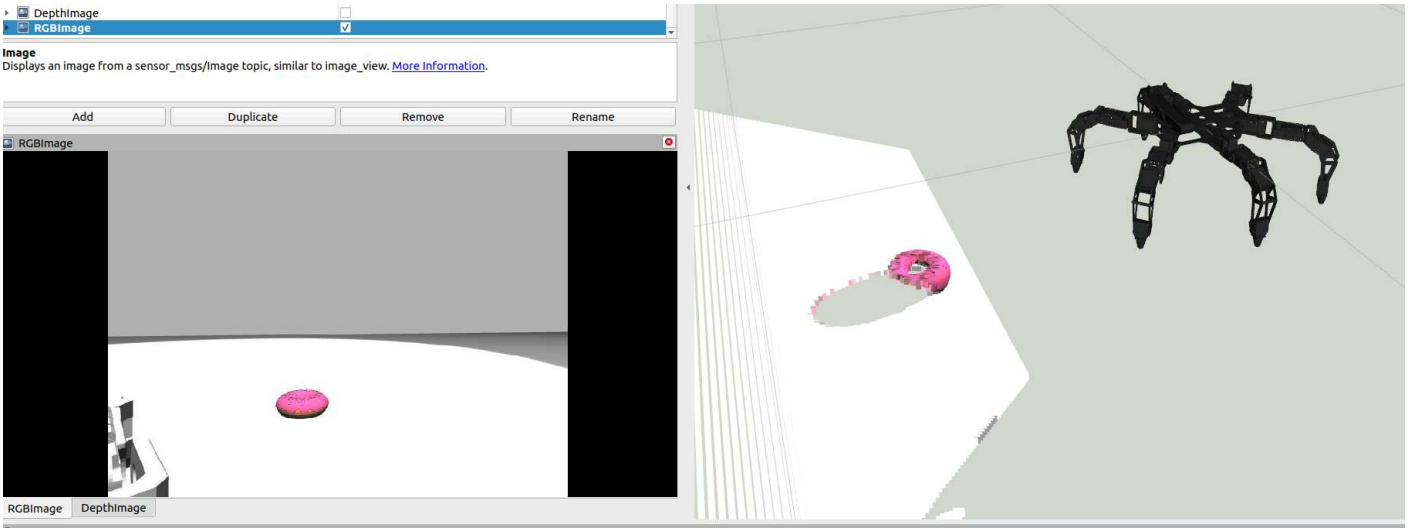
This is a very useful in perception to know if the robot should be seeing the object or not. You can use any CameraInfo topics:

In [ ]:
```
/camera/rgb/camera_info
/camera/depth_registered/camera_info
```

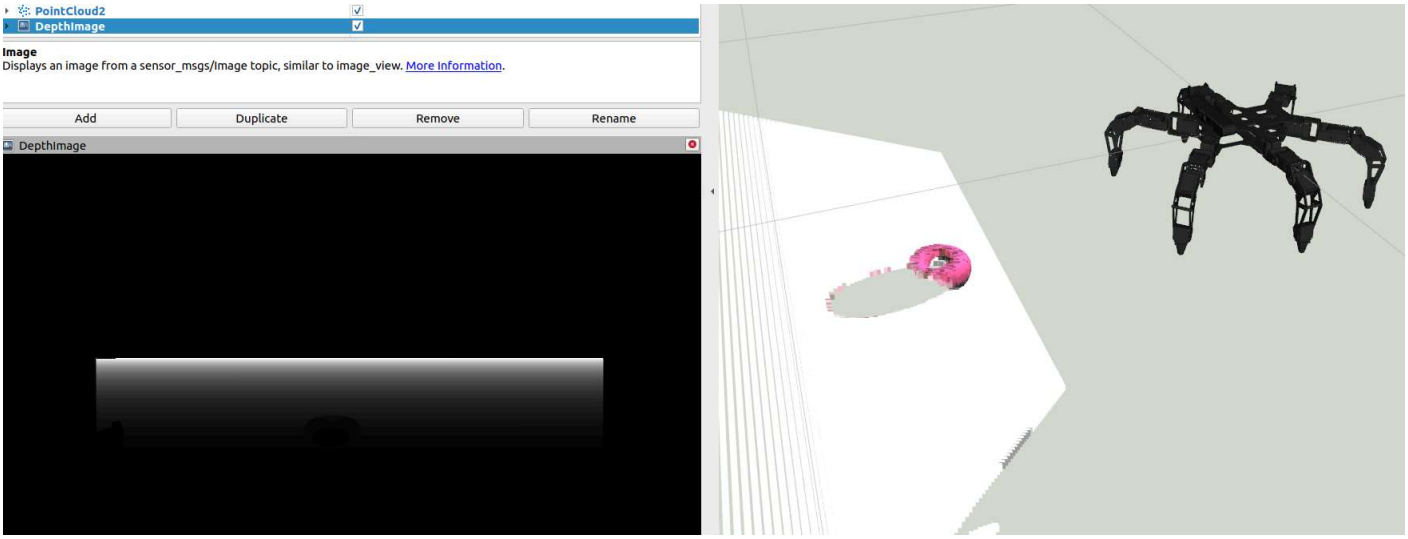
Image and Point-Cloud

Below you can see a comparison of how the RGB, depth image, and point-cloud cameras see the girl, Olive:

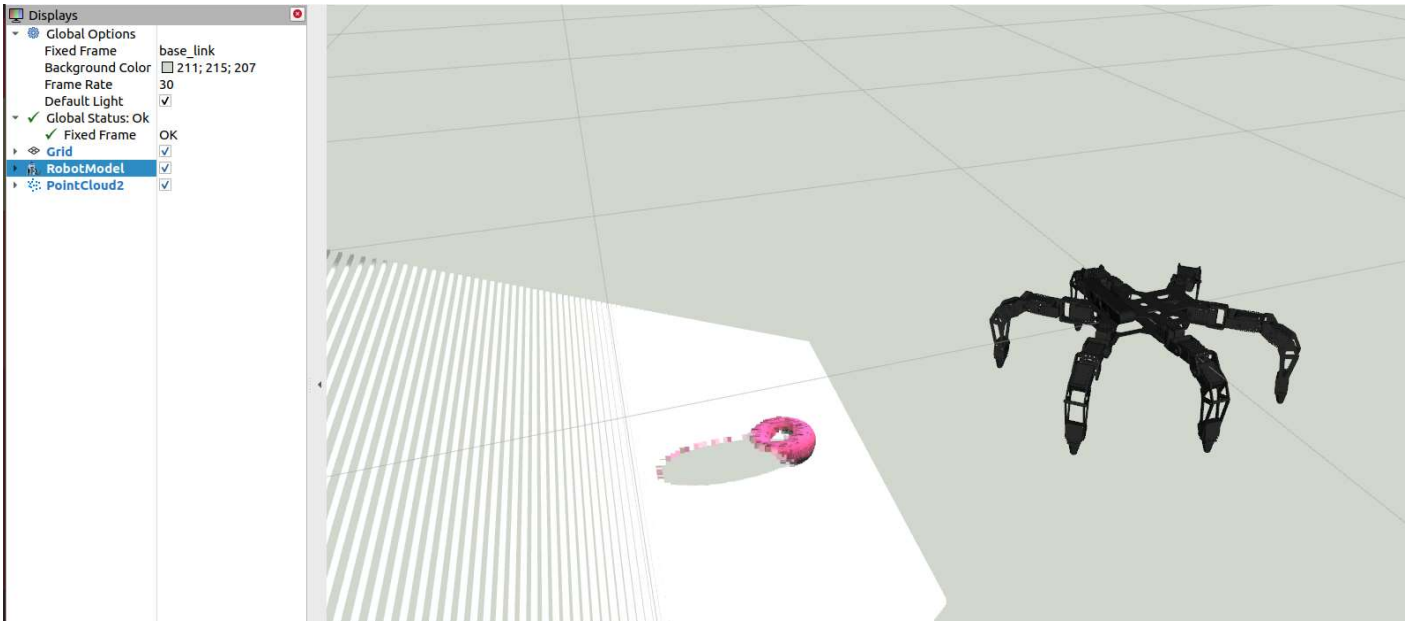
RGB:



Depth-2D-camera:

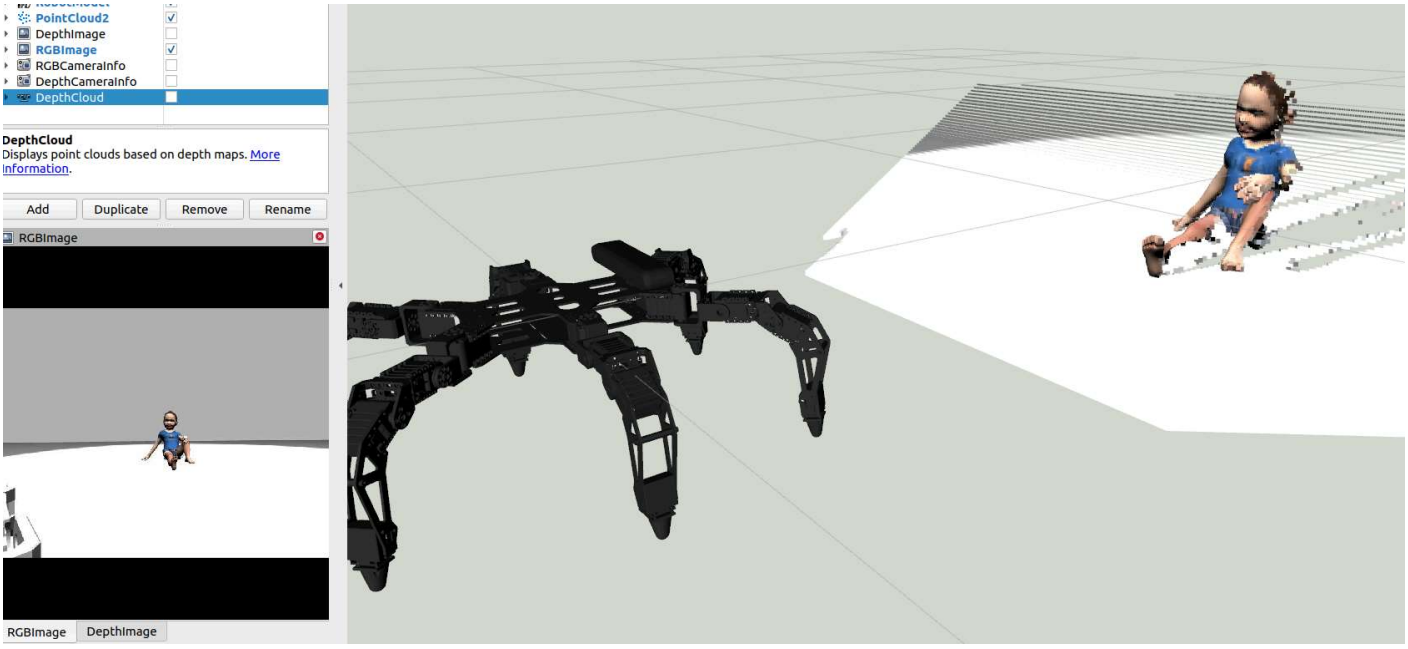


Point-Cloud camera:

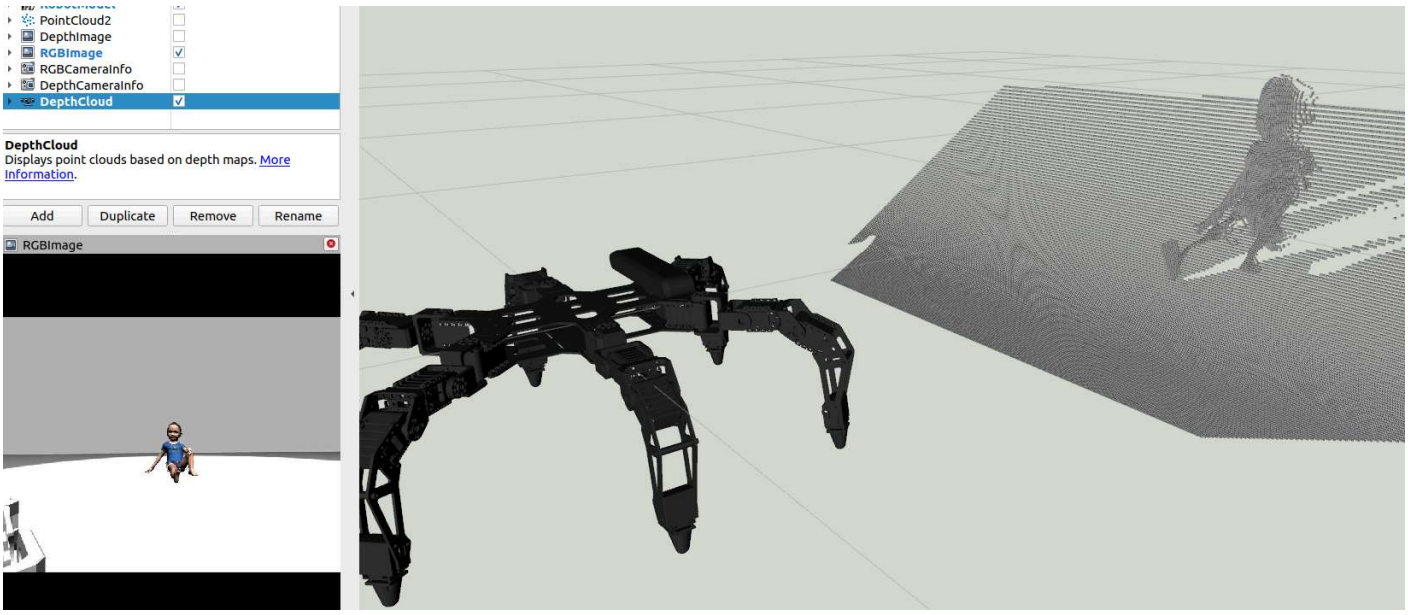


You can visualise the point-clouds in many ways. Below are two examples, one with the PointCloud2 RViz element, and the other with the DepthCloud element.

PointCloud2:



DepthCloud:

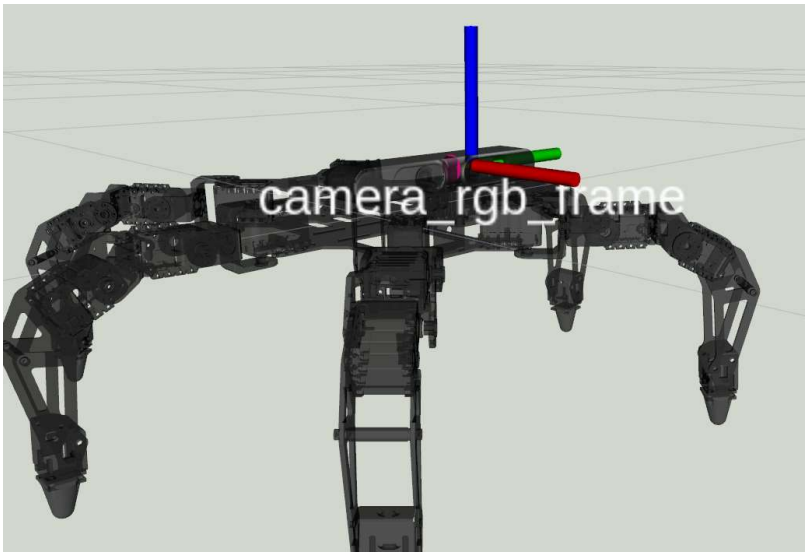


The Camera Optical Frame Problem

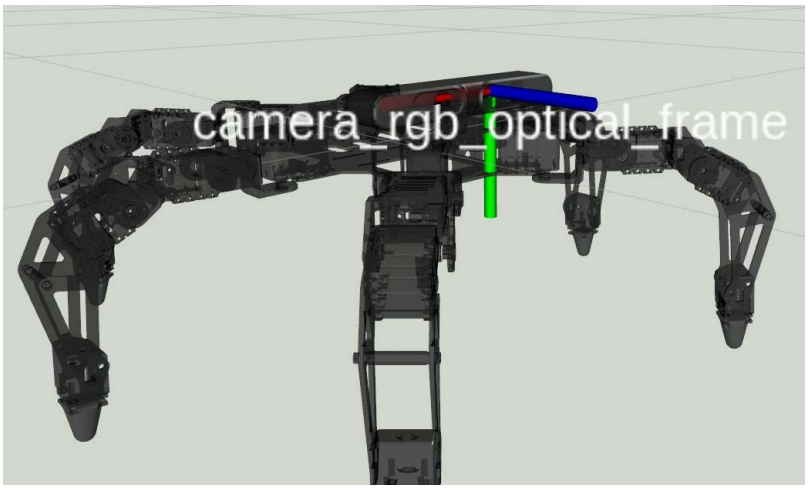
Finally, a topic that is a widespread problem in perception is the reference frames for the sensors. Commonly, the frame of the place where the sensor is mounted is inverted from the sensor frame. This is because the sensor readings might be inversed. To visualize data coherently, the sensor frames are reversed. This could also be the case if the processing algorithm, for some reason, outputs data inversely. For all this, you have to be careful to see how the reference frames are mounted, and data is visualized in RViz.

Here you have how the camera_frame (the physical frame where the camera is mounted) and the optical_frame (the frame used by the sensor) are oriented. They are inverted. This is because the sensor, in this case, works that way.

RGB camera frame:

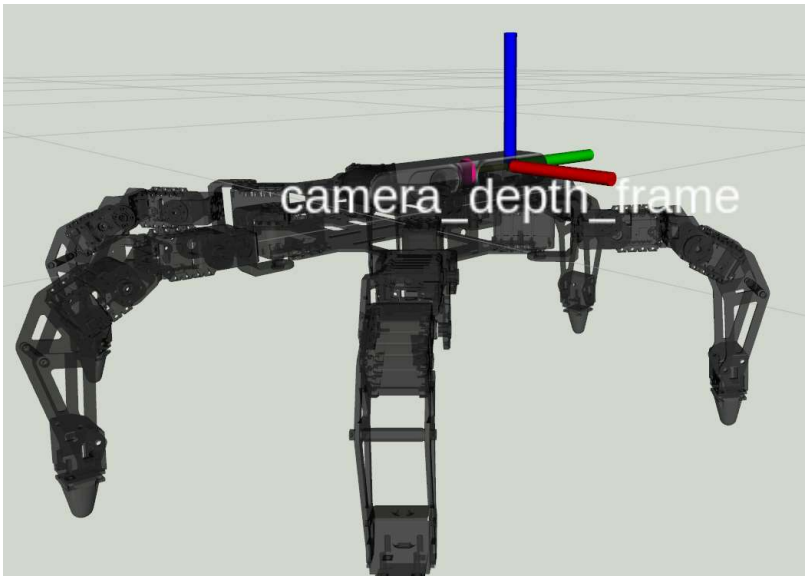


RGB optical frame:

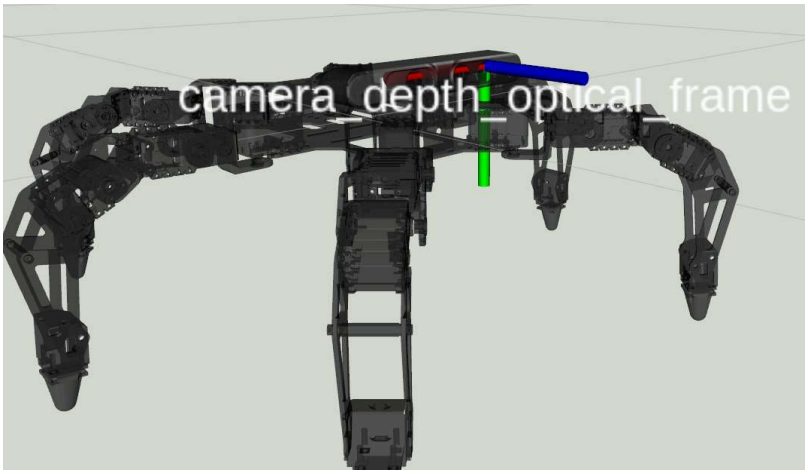


Note that the **optical frames are inverted**, and that RGB and depth image cameras are displaced because they are two different sensors.

Depth image camera frame:



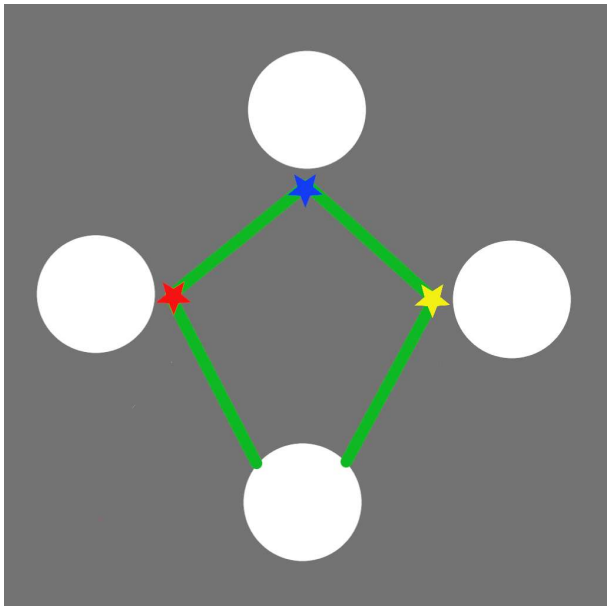
Depth image optical frame:



Elements of the Simulated World

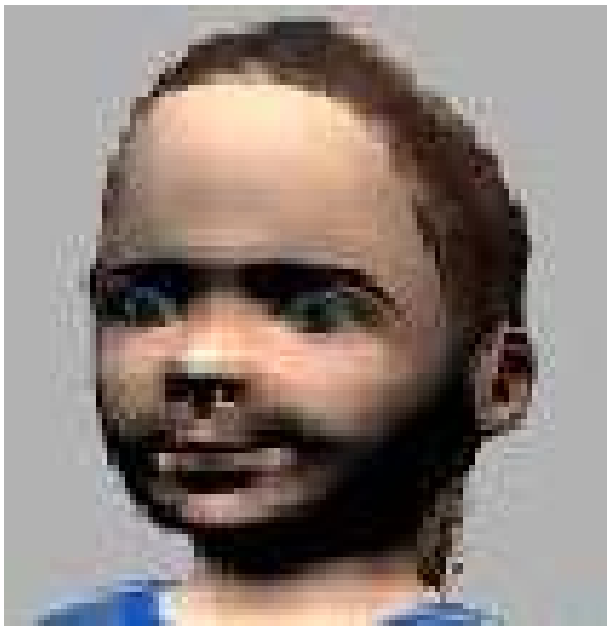
In the world in which Aibo is spawned, the following elements are used for the various tasks to be performed in this project:

- Line following map: The floor has patterns painted on it. They will be used for the line following tasks and navigation. For example, to find the Aibo Bone, you will have to follow the white lines and the colored stars to find it with perception. You will start on the black star and work your way from there.

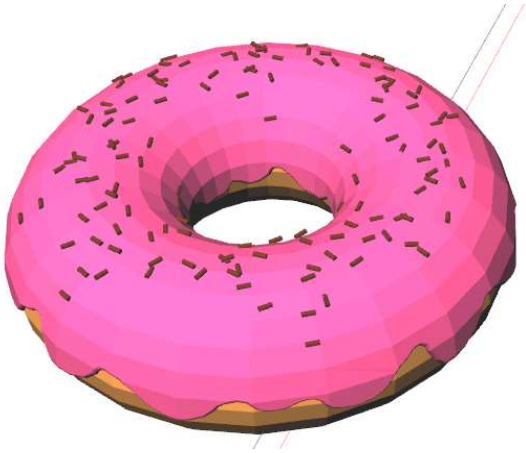


- Martian: The martian moving in circles

- Olive: This girl will be used for the face detection and recognition



- Pink donut: for object recognition



Project Exercises

- WARNING -

GdkError:

When you launch this for the first time you might get the follwoinf error.

DON'T PANIC. Its happens sometimes.

Just relaunch the command again and it should work without any more issues

```
(YOLO V3:1266): Gdk-ERROR **: 08:07:06.547: The program 'YOLO V3' received an X Window System error.
This probably reflects a bug in the program.
The error was 'BadAccess (attempt to access private resource denied)'.
  (Details: serial 253 error_code 10 request_code 130 (MIT-SHM) minor_code 1)
  (Note to programmers: normally, X errors are reported asynchronously;
   that is, you will receive the error a while after causing it.
   To debug your program, run it with the GDK_SYNCHRONIZE environment
   variable to change this behavior. You can then get a meaningful
   backtrace from your debugger if you break on the gdk_x_error() function.)
```

- End WARNING -

It's time to get to work.

Below you have a list of proposed exercises to practice what you have learned about Basic Perception with ROS.

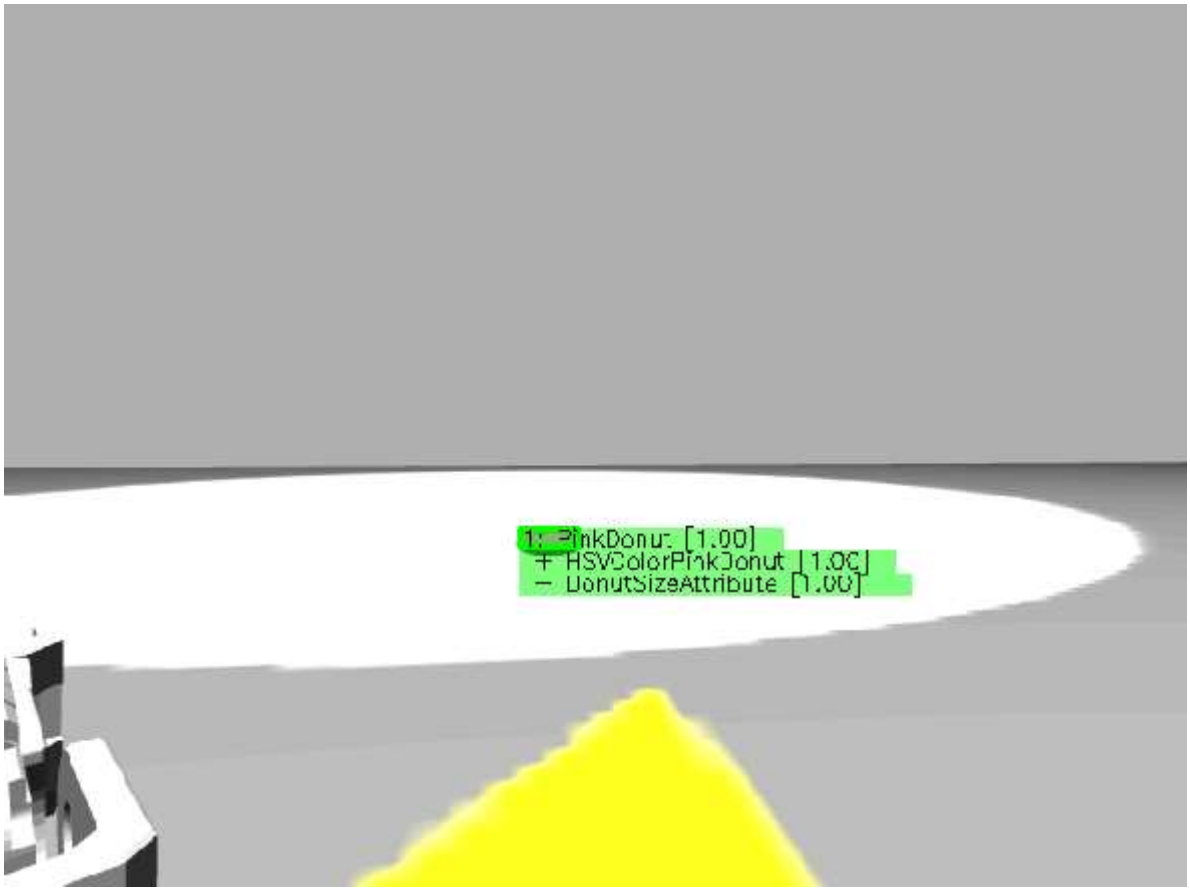
To help you find the solutions afterward, if needed, create them in a package called **my_hexapod_perception**.

P-1 Basic Object Recognition

Create a launch file named **hexapod_object_detection.launch**.

You have to be able to detect two different objects:

- The pink donut: this was done in previous chapters, but you will have to adjust the HSV because the camera and the world have changed, and therefore the lighting conditions and other colors in the scene will change the result. You should get a clean detection, **WITHOUT false positives due to the floor colors (some are pinkish)**, like the image below:



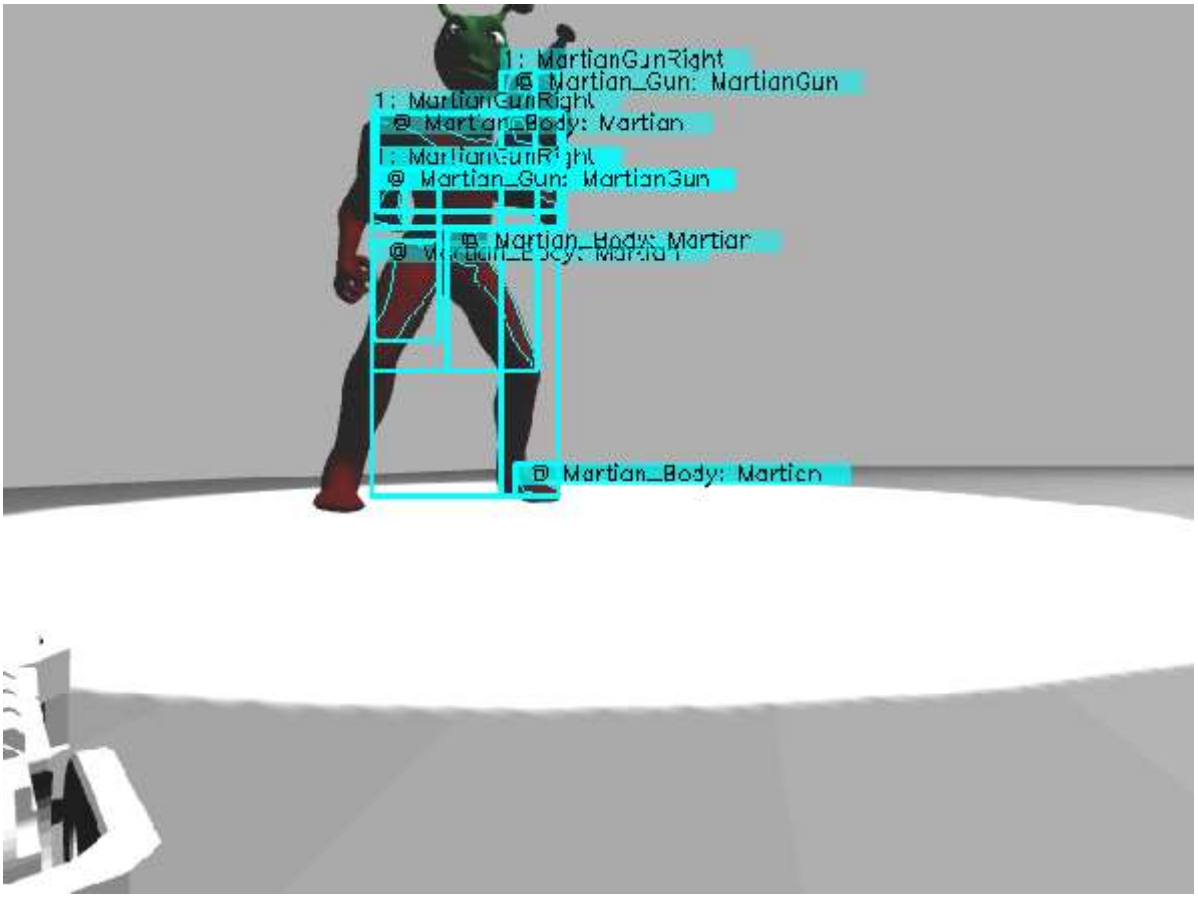
- The martian **WITH** the gun: This detection has two parts.

The first part is to generate the histograms for the **martian** and another for the **martian gun**. Then create **simple objects** that can detect each of them simultaneously. **The second part** is to make a combined detection **complex object** that detects the **martian with The GUN on the RIGHT**. Below you have an example of how to do the complex object:

In [ ]:

```
<RelationLib>

    <RelationShip Type="SpaceRight" Name="right"/>

</RelationLib>


<ComplexObjectBase>

    <ComplexObject ID="1" Name="MartianGunRight">
        <SimpleObject Class="Martian" InnerName="Martian_Body"/>
        <SimpleObject Class="MartianGun" InnerName="Martian_Gun"/>

        <Relation Obj1="Martian_Body" Obj2="Martian_Gun" Relationship="right"/
    </ComplexObject>


</ComplexObjectBase>
```

You should see something similar to the image below:



P-2 Follow a Line Exercise

Create a launch file called **hexapod_follow_line.launch** based on the **line_objective.py** exercise in Chapter 3, Exercise No. **3.3.2**. It should do the same thing:

- Through a topic, it has to change its behavior.
- Follow the yellow path with the command **green_right**. Follow it, turning left if the command is **green_left**.
- If stated **red**, **blue**, or **yellow**, it should follow the green path until it finds the start it was looking for. Then it will stop and finish the program.

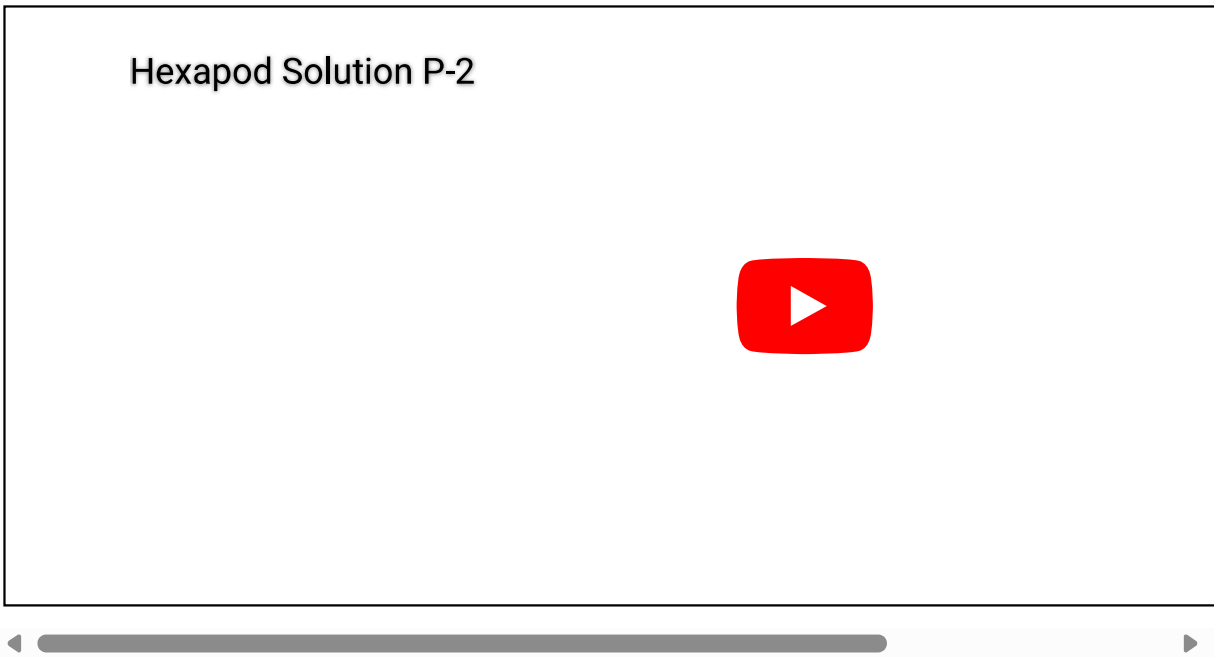
You will have to take into account the following:

- The Hexapod behaves differently from the TurtleBot, so it will probably be necessary to create some states when to turn and when to go forward. Also, it's more sensitive to high-speed values, so don't exceed **0.4** any with linear or angular speed.
- The colors have changed; the line is now **green**, so be careful with that.
- Also, you will need to adjust the HSV values because the environment has changed. If not, you might get false positives.

So you should have something similar to below (select the cell and press RUN in the notebook if the video doesn't show):

In [1]:
```
from IPython.display import YouTubeVideo

YouTubeVideo('nq_U1YfkXW8', width=800, height=300)
```

Out[1]:



P-3 Detect Face and Recognition

Create a launch file named **detect_person.launch**.

In this exercise you will need to:

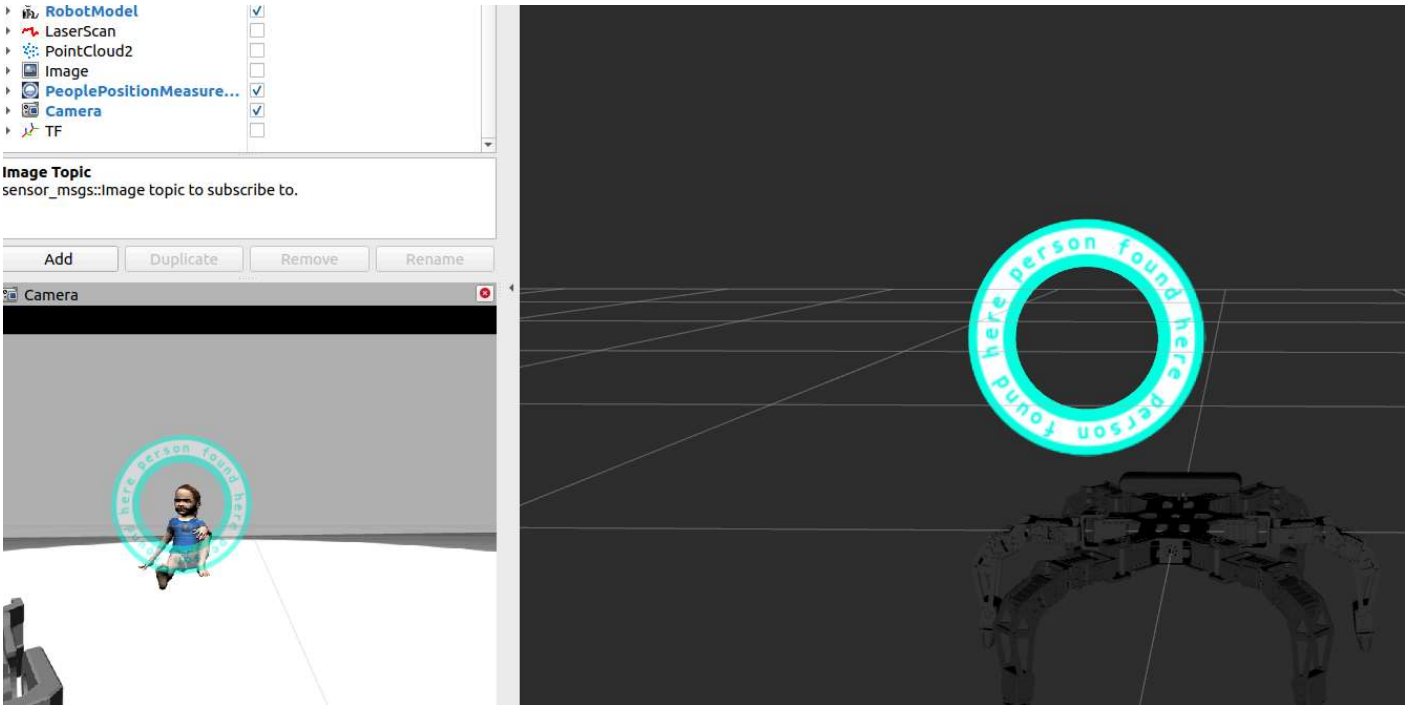
- Be able to detect a human face
- Recognize who it is

Olive:

Note that the image of the face is much smaller than the ones that you have used. This means that in the face detection, you **SHOULD NOT scale the image down for processing because** you won't make detections.

The image is small because the camera in this **HEXAPOD** has a lower resolution, and also, we can't get as close as the Fetch Robot did to the people for better quality images.

You should get something similar to the image below for the face detection:



You should get something similar to the image below for the face recognition:

Congratulations! You finished the Perception Project.

We would be delighted to know what different strategies you developed to complete the project. If you are interested in share, tweet it to us at @_TheConstruct_(). We're excited to see how you did!