ROS Perception in 5 Days

Chapter 1: Vision Basics in ROS Part 1 - Follow a Red Ball

- Summary -

Estimated time to completion: **3 hours**

In this chapter, you will learn how to create a basic blob tracker. Specifically, you will learn about:

- How to visualize an RGB camera in ROS.
- How to move through topics as a robot.
- How to create a blob detector.
- How to get the color encoding for the color that you want to track.
- How to use blob detection data and move the RGB camera to follow the blob.

- End of Summary -

The most common sensor in robotics is the humble RGB camera. It's used for everything from basic color tracking (blob tracking) to artificial intelligence (AI) autonomous driving. So getting to know this basic perception sensor and how to use it in ROS is of the upmost importance.

In this unit, you will use cameras in ROS and use **OpenCV** for blob tracking in a very crude but effective way. In Chapter 2, you will go deeper into how blob tracking is done and how the image is processed.

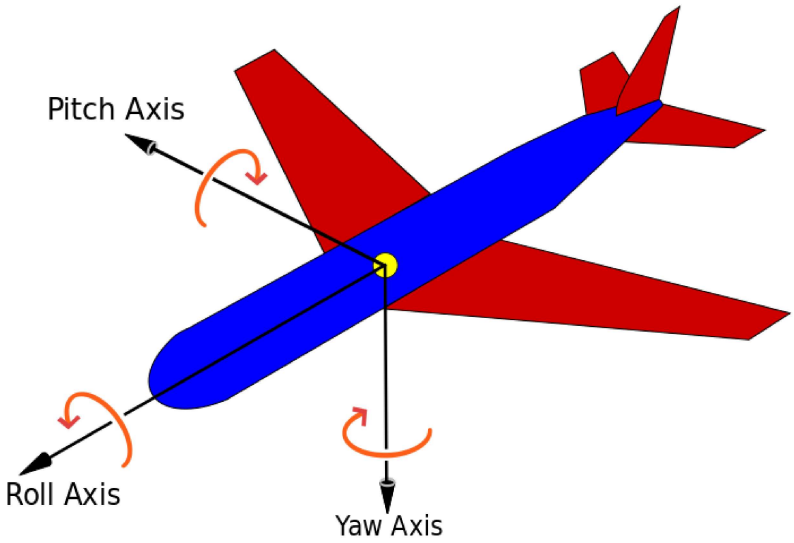
If you want to learn more about OpenCV, we recommend you take the OpenCV Basics Course (https://app.theconstructsim.com/#/Course/65).

1.1 The First Image from a Robot

1.2 Roll, Pitch, and Yaw

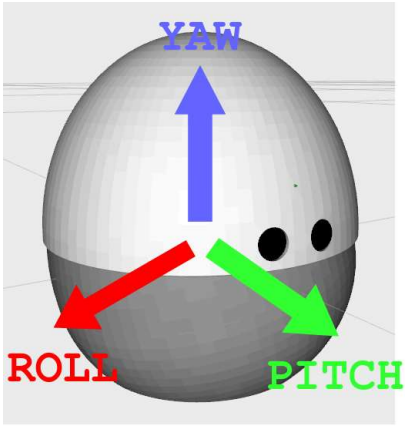
Let's get to work! In the image above, you see Mira in a room with a red cricket ball (red-harrow-robot).

Mira is a 3 degrees-of-freedom robot that turns its head in a **Roll-Pitch-Yaw** movement, which is very easy for camera movement. It's the perfect robot for this introduction to image.

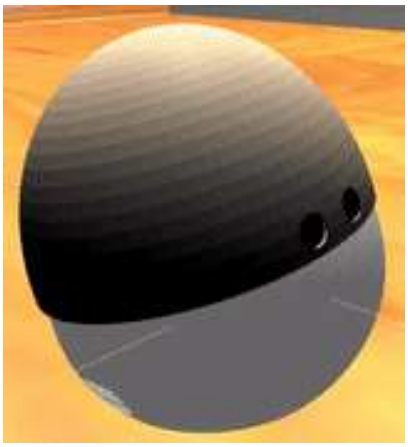


Auawise (//commons.wikimedia.org/wiki/User:Auawise) derivative work: Jrvz (//commons.wikimedia.org/w/index.php?title=User:Jrvz&action=edit&redlink=1) (talk (//commons.wikimedia.org/w/index.php?title=User_talk:Jrvz&action=edit&redlink=1)) - Yaw_Axis.svg (//commons.wikimedia.org/wiki/File:Yaw_Axis.svg), CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0), Link (https://commons.wikimedia.org/w/index.php?curid=9441238)

For Mira, the axes are slightly different, more in the fashion of robotics than aerospace (where they are inverted):



Roll Axis Movement:



Roll Axis Movement

Pitch Axis Movement:

Pitch Axis Movement

Yaw Axis Movement:

Yaw Axis Movement

You will also have at your disposal a script for autonomously moving the red-haro-robot around. So, let's move it.

▶ Execute in WebShell #1

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

You can move the ball using the following basic keyboard commands.

i	Move Forwards
,	Move Backwards
j	Turn Left
l	Turn Right
k	Stop
u	Move Forwards + Turning Left
o	Move Forwards + Turning Right
m	Move Backwards + Turning Left
.	Move Backwards + Turning Right
t	Move Up
g	Stop Going Up/Down
b	Move Down
q	Increase Speed
z	Decrease Speed

Now, you are going to see what Mira sees. You will use a ROS graphical tool named ***rqt_image_view*** that allows you to see what the camera in the robot is publishing.

To open the tool, type the following:

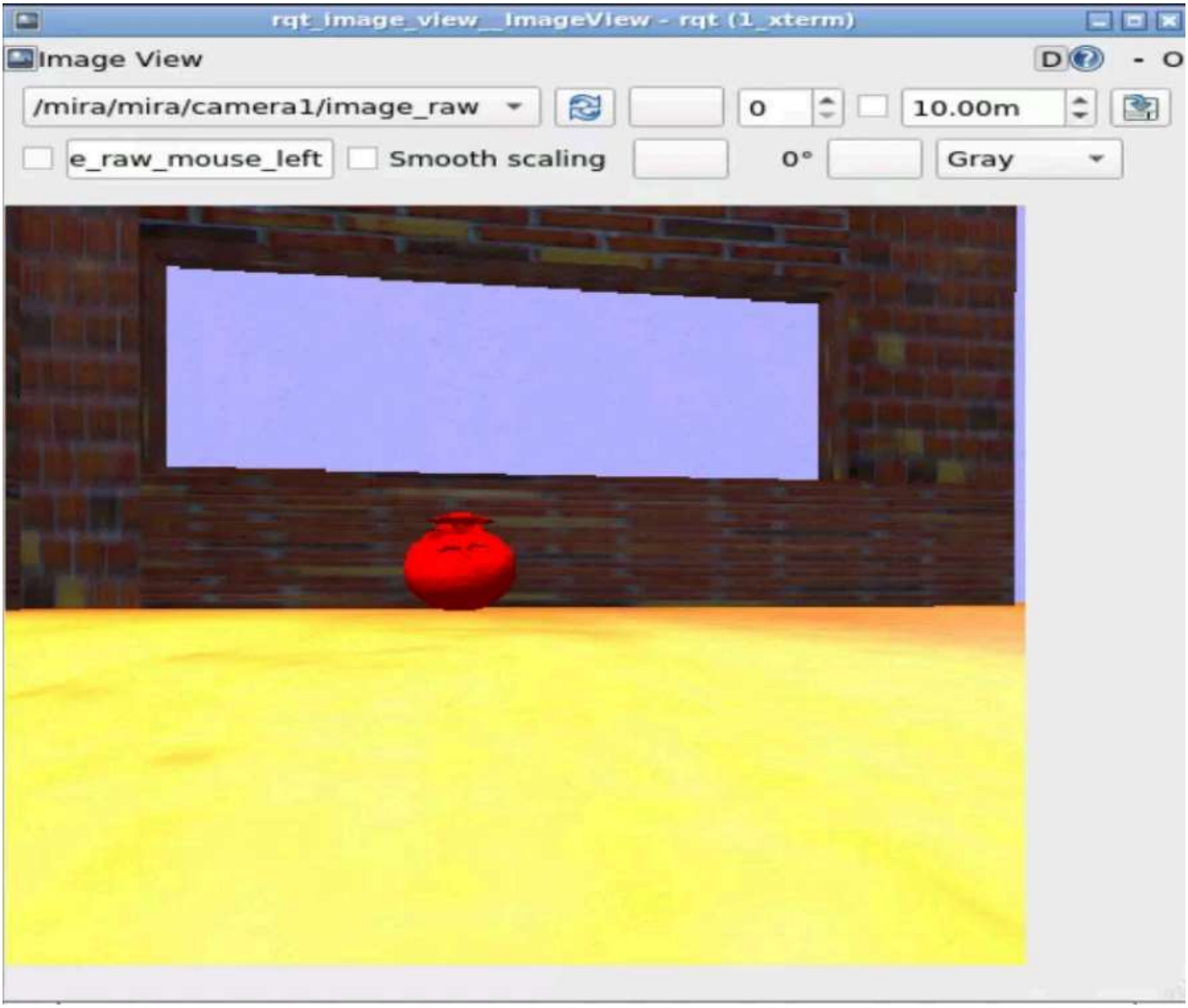
▶ Execute in WebShell #1

In [ ]:
```
rosrun rqt_image_view rqt_image_view
```

Now, open the Graphical Interface by clicking on the Graphical Interface icon, same as the image below:

A window for the ***rqt_image_view*** application should appear on your screen.

In the application, select the ***/mira/mira/camera1/image_raw*** image topic and wait a few seconds until the image feed is established. You should see something similar to the image below.



- Exercise 1.1 -

Great! Now practice moving the ball around by trying to make it appear in Mira's view. Once Mira can see the ball, you can close the WebShell #2 program.

- END Exercise 1.1 -

1.3 Color Encoding

Now, you're going to create a program that tracks blobs of color in an image. Blobs are areas in an image with a similar color encoding. The first step is to get a color encoding that defines the object you want to be tracked. Let's do that with the red-haro-robot, shall we?

To obtain the color encoding, we're going to use a Python script already installed that receives images from the camera and allows you to move the sliders around to get the values of the color encoding needed.

There are **TWO different color encodings**:

- RGB: It encodes based on the combination of Red-Green-Blue values that go from 0-255
- HSV: It encodes based on Hue-Saturation-Value with values between 0-255.

We will use HSV here because it tends to be more robust against changes in lighting conditions.

Launch the following command in a terminal and go to the Graphical Interface tab:

▶ Execute in WebShell #1

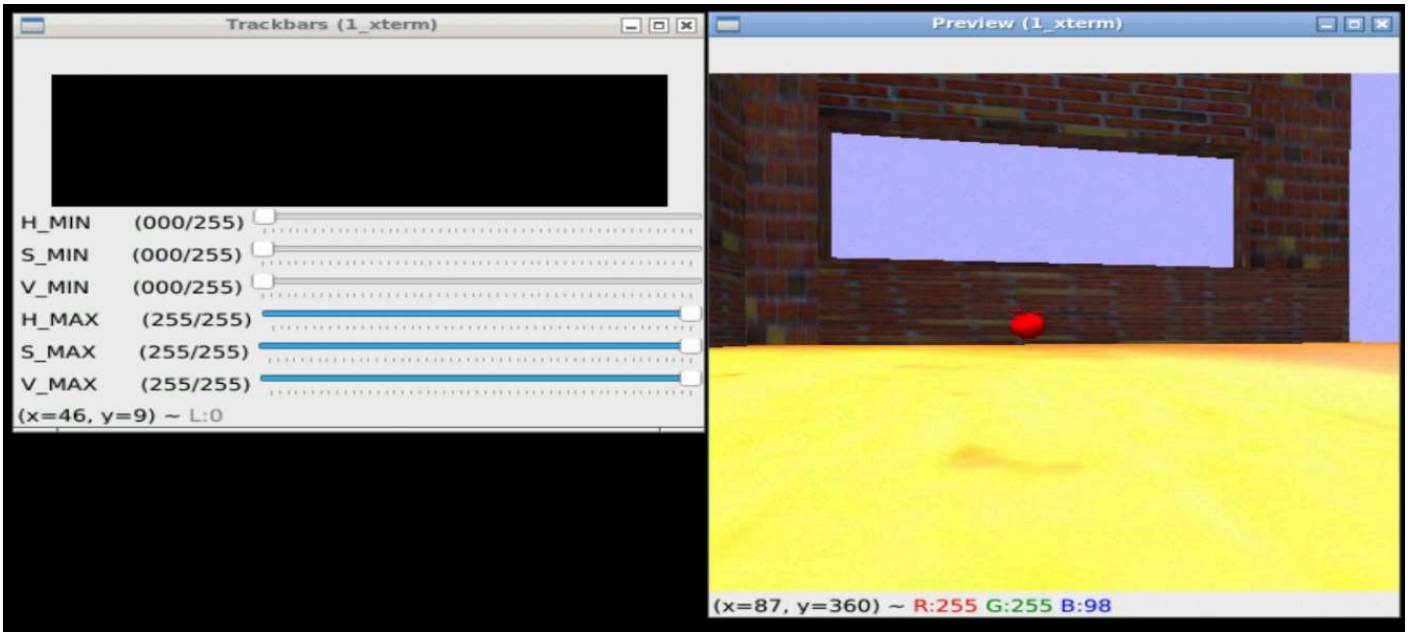
- WARNING -

This launch can take up to 45 seconds to appear in the graphical tools. So **DON'T PANIC if it doens't appear immediately**.

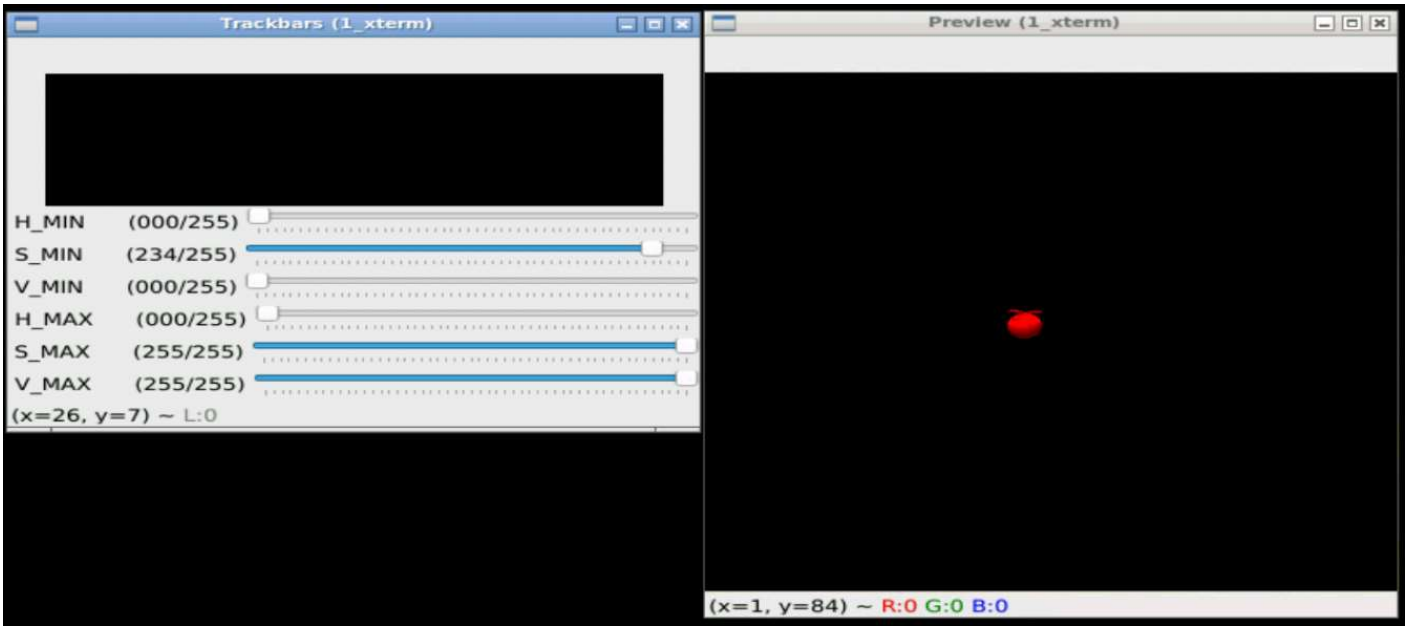
- End WARNING -

In [ ]:
```
rosrun blob_tracking_v2 range_detector.py --filter HSV --preview
```

This will start a GUI similar to the following:



Now you have to move the sliders until the only thing previewed is the red-haro-robot. Take as reference this result:



The values that work best should be similar to the values below:

- H_MIN = 0
- S_MIN = 234
- V_MIN = 0
- H_MAX = 0
- S_MAX = 255
- V_MAX = 255

1.4 Create A Blob Tracking Package

Now, let's create a package to launch all of the required software to track the red-haro-robot.

- First, create a new package named ***my_blob_tracking_pkg***, which depends on *rospy*.
- Inside that package, we will create the needed scripts to make it work insid ethe `scripts` folder.

▶ Execute in WebShell #1

In [ ]:

```
cd ~/catkin_ws/src
catkin_create_pkg my_blob_tracking_pkg rospy cv_bridge image_transport sensor_
cd ~/catkin_ws/
catkin_make
source devel/setup.bash
rospack profile
```

1.5 Start Blob Tracking with OpenCV

To track a blob, we need the following scripts:

- Access to the RGB camera images: **mira_sensors.py**
- A blob detector that detects blobs in the images: **blob_detector.py**
- A blob tracker that moves Mira's head: **mira_follow_blob.py**

▶ Execute in WebShell #1

In [ ]:

```
roscd my_blob_tracking_pkg
mkdir scripts;cd scripts
# We create empty files
touch mira_sensors.py
touch blob_detector.py
touch mira_follow_blob.py
# We make all the python scripts executable
chmod +x *.py
```

mira_sensors.py

Fill in the file with the following code, and after, let's comment on specific elements. If there are some Python or ROS concepts that we don't explain here, it's because a basic understanding of Python and ROS are a prerequisite for this course.

- mira_sensors.py -

In [ ]:

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

import sys
import rospy
import cv2
import numpy as np
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image


class MiraSensors(object):

    def __init__(self, show_raw_image = False):

        self._show_raw_image = show_raw_image
        self.bridge_object = CvBridge()
        self.camera_topic = "/mira/mira/camera1/image_raw"
        self._check_cv_image_ready()
        self.image_sub = rospy.Subscriber(self.camera_topic,Image,self.camera_


    def _check_cv_image_ready(self):
        self.cv_image = None
        while self.cv_image is None and not rospy.is_shutdown():
            try:
                raw_cv_image = rospy.wait_for_message("/mira/mira/camera1/imag
                self.cv_image = self.bridge_object.imgmsg_to_cv2(raw_cv_image,
                rospy.logdebug("Current "+self.camera_topic+" READY=>")

            except:
                rospy.logerr("Current "+self.camera_topic+" not ready yet, ret
        return self.cv_image

    def camera_callback(self,data):

        try:
            # We select bgr8 because its the OpneCV encoding by default
            self.cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_enc
        except CvBridgeError as e:
            print(e)

        if self._show_raw_image:
            cv2.imshow("Image window", self.cv_image)
            cv2.waitKey(1)
```

```
    def get_image(self):
        return self.cv_image




def main():
    mira_sensors_object = MiraSensors()
    rospy.init_node('mira_sensors_node', anonymous=True)
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")
    cv2.destroyAllWindows()

if __name__ == '__main__':
    main()
```

- End mira_sensors.py -

In [ ]:
```
self.camera_topic = "/mira/mira/camera1/image_raw"
self._check_cv_image_ready()
self.image_sub = rospy.Subscriber(self.camera_topic,Image,self.camera_callback
```

We are subscribing to the same image topic that you visualized at the start of this unit. We also check its publishing before continuing.

In [ ]:
```
try:
    # We select bgr8 because its the OpneCV encoding by default
    self.cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="b
except CvBridgeError as e:
    print(e)
```

We use **cv_bridge**, a ROS package that allows you to convert **ROS Image Messages** into **OpenCV** objects. This opens ROS programs to use OpenCV for anything you want. We then save the latest image so that you can have access to it at any time during this class.

In [ ]:
```
def get_image(self):
    return self.cv_image
```

This is the method used to access the latest image on the camera.

- blob_detector.py -

In [ ]:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import rospy
import cv2
import numpy as np
from mira_sensors import MiraSensors
from geometry_msgs.msg import Point


class BlobTracker(object):

    def __init__(self):
        self.point_blob_topic = "/blob/point_blob"
        # This publisher  uses Point message to publish
        # x,y: x,y relative poses of the center of the blob detected relative
        # z: size of the blob detected
        self.pub_blob = rospy.Publisher(self.point_blob_topic,  Point, queue_s


    def blob_detect(self,
                    image,                  #-- The frame (cv standard)
                    hsv_min,                #-- minimum threshold of the hsv f
                    hsv_max,                #-- maximum threshold of the hsv f
                    blur=0,                 #-- blur value (default 0)
                    blob_params=None,       #-- blob parameters (default None)
                    search_window=None,     #-- window where to search as [x_m
                    imshow=False
                ):
        """
        Blob detecting function: returns keypoints and mask
        return keypoints, reversemask
        """

        #- Blur image to remove noise
        if blur > 0:
            image    = cv2.blur(image, (blur, blur))
            #- Show result
            if imshow:
                cv2.imshow("Blur", image)
                cv2.waitKey(0)

        #- Search window
        if search_window is None: search_window = [0.0, 0.0, 1.0, 1.0]
```

```
        #- Convert image from BGR to HSV
        hsv     = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

        #- Apply HSV threshold
        mask    = cv2.inRange(hsv,hsv_min, hsv_max)

        #- Show HSV Mask
        if imshow:
            cv2.imshow("HSV Mask", mask)

        #- dilate makes the in range areas larger
        mask = cv2.dilate(mask, None, iterations=2)
        #- Show HSV Mask
        if imshow:
            cv2.imshow("Dilate Mask", mask)
            cv2.waitKey(0)

        mask = cv2.erode(mask, None, iterations=2)

        #- Show dilate/erode mask
        if imshow:
            cv2.imshow("Erode Mask", mask)
            cv2.waitKey(0)

        #- Cut the image using the search mask
        mask = self.apply_search_window(mask, search_window)

        if imshow:
            cv2.imshow("Searching Mask", mask)
            cv2.waitKey(0)

        #- build default blob detection parameters, if none have been provided
        if blob_params is None:
            # Set up the SimpleBlobdetector with default parameters.
            params = cv2.SimpleBlobDetector_Params()

            # Change thresholds
            params.minThreshold = 0
            params.maxThreshold = 100

            # Filter by Area.
            params.filterByArea = True
            params.minArea = 30
            params.maxArea = 20000

            # Filter by Circularity
```

```
            params.filterByCircularity = False
            params.minCircularity = 0.1

            # Filter by Convexity
            params.filterByConvexity = False
            params.minConvexity = 0.5

            # Filter by Inertia
            params.filterByInertia =True
            params.minInertiaRatio = 0.5

        else:
            params = blob_params

        #- Apply blob detection
        detector = cv2.SimpleBlobDetector_create(params)

        # Reverse the mask: blobs are black on white
        reversemask = 255-mask

        if imshow:
            cv2.imshow("Reverse Mask", reversemask)
            cv2.waitKey(0)

        keypoints = detector.detect(reversemask)

        return keypoints, reversemask


    def draw_keypoints(self,
                       image,                   #-- Input image
                       keypoints,               #-- CV keypoints
                       line_color=(0,255,0),    #-- line's color (b,g,r)
                       imshow=False             #-- show the result
                       ):
        """
        Draw detected blobs: returns the image
        return(im_with_keypoints)
        """

        #-- Draw detected blobs as red circles.
        #-- cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS ensures the size of the
        im_with_keypoints = cv2.drawKeypoints(image, keypoints, np.array([]),

        if imshow:
            # Show keypoints
```

```
            cv2.imshow("Keypoints", im_with_keypoints)

        return(im_with_keypoints)


    def draw_window(self,
                    image,              #- Input image
                    window_adim,        #- window in adimensional units
                    color=(255,0,0),    #- line's color
                    line=5,             #- line's thickness
                    imshow=False        #- show the image
                ):
        """
        Draw search window: returns the image
        return(image)
        """

        rows = image.shape[0]
        cols = image.shape[1]

        x_min_px    = int(cols*window_adim[0])
        y_min_px    = int(rows*window_adim[1])
        x_max_px    = int(cols*window_adim[2])
        y_max_px    = int(rows*window_adim[3])

        #-- Draw a rectangle from top left to bottom right corner
        image = cv2.rectangle(image,(x_min_px,y_min_px),(x_max_px,y_max_px),cc

        if imshow:
            # Show keypoints
            cv2.imshow("Keypoints", image)

        return(image)


    def draw_frame(self,
                image,
                dimension=0.3,      #- dimension relative to frame size
                line=2              #- line's thickness
        ):
        """
        Draw X Y frame
        return : image
        """

        rows = image.shape[0]
```

```
        cols = image.shape[1]
        size = min([rows, cols])
        center_x = int(cols/2.0)
        center_y = int(rows/2.0)

        line_length = int(size*dimension)

        #-- X
        image = cv2.line(image, (center_x, center_y), (center_x+line_length, c
        #-- Y
        image = cv2.line(image, (center_x, center_y), (center_x, center_y+line

        return (image)


    def apply_search_window(self, image, window_adim=[0.0, 0.0, 1.0, 1.0]):
        """
        Apply search window
        return: image
        """
        rows = image.shape[0]
        cols = image.shape[1]
        x_min_px    = int(cols*window_adim[0])
        y_min_px    = int(rows*window_adim[1])
        x_max_px    = int(cols*window_adim[2])
        y_max_px    = int(rows*window_adim[3])

        #--- Initialize the mask as a black image
        mask = np.zeros(image.shape,np.uint8)

        #--- Copy the pixels from the original image corresponding to the wina
        mask[y_min_px:y_max_px,x_min_px:x_max_px] = image[y_min_px:y_max_px,x_

        #--- return the mask
        return(mask)


    def blur_outside(self, image, blur=5, window_adim=[0.0, 0.0, 1.0, 1.0]):
        """
        Apply a blur to the outside search region
        """
        rows = image.shape[0]
        cols = image.shape[1]
        x_min_px    = int(cols*window_adim[0])
        y_min_px    = int(rows*window_adim[1])
        x_max_px    = int(cols*window_adim[2])
```

```
        y_max_px    = int(rows*window_adim[3])

        # Initialize the mask as a black image
        mask    = cv2.blur(image, (blur, blur))

        # Copy the pixels from the original image corresponding to the window
        mask[y_min_px:y_max_px,x_min_px:x_max_px] = image[y_min_px:y_max_px,x_

        return(mask)


    def get_blob_relative_position(self, image, keyPoint):
        """
        Obtain the camera relative frame coordinate of one single keypoint
        return(x,y)
        """
        rows = float(image.shape[0])
        cols = float(image.shape[1])
        # print(rows, cols)
        center_x    = 0.5*cols
        center_y    = 0.5*rows
        # print(center_x)
        x = (keyPoint.pt[0] - center_x)/(center_x)
        y = (keyPoint.pt[1] - center_y)/(center_y)
        return x,y

    def publish_blob(self, x, y ,size):
        blob_point = Point()
        blob_point.x = x
        blob_point.y = y
        blob_point.z = size
        self.pub_blob.publish(blob_point)


if __name__=="__main__":

    rospy.init_node("blob_detector_node", log_level=rospy.DEBUG)
    mira_sensors_obj = MiraSensors()
    cv_image = mira_sensors_obj.get_image()

    blob_detector_object = BlobTracker()

    # HSV limits for RED Haro
    hsv_min = (0,234,0)
    hsv_max = (0, 255, 255)
```

```
    # We define the detection area [x_min, y_min, x_max, y_max] adimensional (
    window = [0.0, 0.0, 1.0, 0.9]

    while not rospy.is_shutdown():
        # Get most recent Image
        cv_image = mira_sensors_obj.get_image()

        # Detect blobs
        keypoints, _ = blob_detector_object.blob_detect(cv_image, hsv_min, hsv
                                    blob_params=None, search_window=window, im
        # Draw window where we make detections
        cv_image = blob_detector_object.draw_window(cv_image, window)


        for keypoint in keypoints:
            x , y = blob_detector_object.get_blob_relative_position(cv_image,
            blob_size =  keypoint.size
            blob_detector_object.publish_blob(x,y,blob_size)

        # Draw Detection
        blob_detector_object.draw_keypoints(cv_image, keypoints, imshow=True)

        #-- press q to quit
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break


    rospy.logwarn("Shutting down")
    cv2.destroyAllWindows()
```

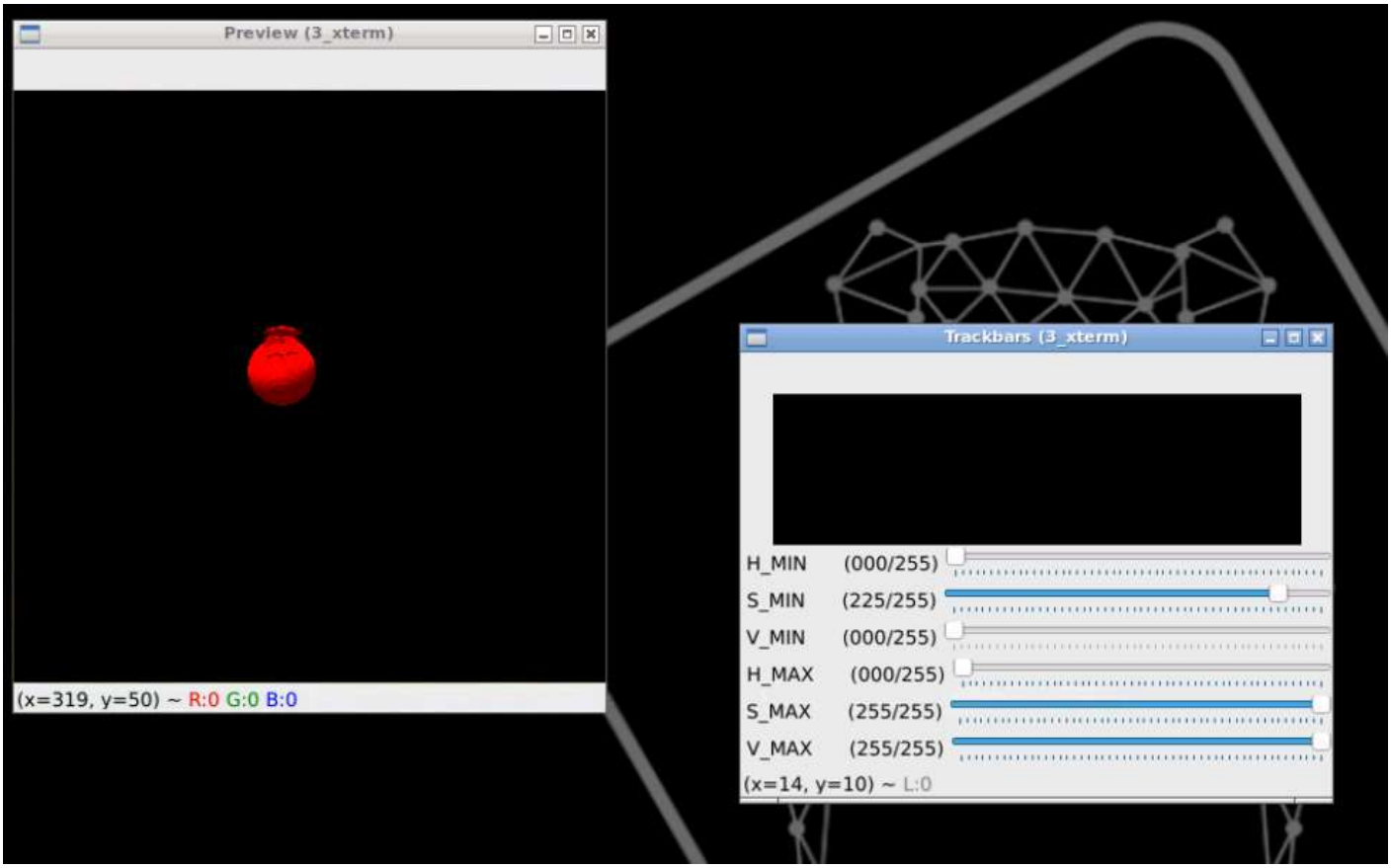
- End blob_detector.py -

Publish the blob detections into the topic **/blob/point_blob**. We won't go into the OpenCV details. We will only cover how it can be used and adapted to different color blobs:

In [ ]:
```
mira_sensors_obj = MiraSensors()
cv_image = mira_sensors_obj.get_image()
```

Use the previous class created to access the images recorded by the RGB camera.

- Here is an example. The values can vary: the image has a minimum of 225, but we set it to 234. The important part is that it works for you.



In [ ]:

```
# HSV limits for RED Haro
hsv_min = (0,234,0)
hsv_max = (0, 255, 255)

# We define the detection area [x_min, y_min, x_max, y_max] adimensional (0.0
window = [0.0, 0.0, 1.0, 0.9]
```

Define the limits of the HSV encoding that we did previously to track the red-haro-robot. If you want to track another color, here is where you would change it.

We also define a window for the detections. This is normally used for performance purposes. It's using the same trick that the human eyes uses. The center of the eye has more definitions than the rest of the eye, which allows for faster processing. Here, it is the same. We have to process less image, and therefore we detect faster. This is vital in semi-real time that robotics needs.

In [ ]:
```
# Detect blobs
keypoints, _ = blob_detector_object.blob_detect(cv_image, hsv_min, hsv_max, bl
                                blob_params=None, search_window=window, imshow=Fal
# Draw window where we make detections
cv_image = blob_detector_object.draw_window(cv_image, window)


for keypoint in keypoints:
    x , y = blob_detector_object.get_blob_relative_position(cv_image, keypoint
    blob_size =  keypoint.size
    blob_detector_object.publish_blob(x,y,blob_size)

# Draw Detection
blob_detector_object.draw_keypoints(cv_image, keypoints, imshow=True)
```

We then make the blob detection and draw on the original image, the window, and the bounding box, for the detection of the blob.

In [ ]:
```
blob_detector_object.publish_blob(x,y,blob_size)
```

Publish the detections to the **/blob/point_blob** topic that the next script will use to move Mira's head around to follow the blob detected. The size of the blob is not used in these examples, but it could be used to have a good estimation of the distance from the object.

- mira_follow_blob.py -

In [ ]:

```
#!/usr/bin/env python
import time
import rospy
from math import pi, sin, cos, acos
import random
from std_msgs.msg import Float64
from sensor_msgs.msg import JointState
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Point

"""
Topics To Write on:
type: std_msgs/Float64
/mira/pitch_joint_position_controller/command
/mira/roll_joint_position_controller/command
/mira/yaw_joint_position_controller/command
"""

class MiraBlobFollower(object):

    def __init__(self, is_2D = True):

        rospy.loginfo("Mira Initialising Blob Follower...")

        self.move_rate = rospy.Rate(10)

        self._is_2D = is_2D
        self.acceptable_error = 0.2

        self.current_yaw = 0.0
        self.twist_obj = Twist()
        self.pub_mira_move = rospy.Publisher('/mira/commands/velocity',  Twist

        self.point_blob_topic = "/blob/point_blob"
        self._check_cv_image_ready()
        rospy.Subscriber(self.point_blob_topic, Point, self.point_blob_callbac

        rospy.loginfo("Mira Initialising Blob Follower...")


    def _check_cv_image_ready(self):
        self.point_blob = None
        while self.point_blob is None and not rospy.is_shutdown():
            try:
                self.point_blob = rospy.wait_for_message(self.point_blob_topic
```

```
                rospy.logdebug("Current "+self.point_blob_topic+" READY=>")

            except:
                rospy.logerr("Current "+self.point_blob_topic+" not ready yet,
        return self.point_blob


    def point_blob_callback(self, msg):

        if msg.x > self.acceptable_error:
            self.twist_obj.angular.z = -1.0
        elif msg.x < -1*self.acceptable_error:
            self.twist_obj.angular.z = 1.0
        else:
            self.twist_obj.angular.z = 0.0

        if msg.y > self.acceptable_error:
            self.twist_obj.angular.x = - 1.0
        elif msg.y < -1*self.acceptable_error:
            self.twist_obj.angular.x = 1.0
        else:
            self.twist_obj.angular.x = 0.0




    def loop(self):

        while not rospy.is_shutdown():

            self.pub_mira_move.publish(self.twist_obj)
            self.move_rate.sleep()


if __name__ == "__main__":
    rospy.init_node('mira_follow_blob_node', anonymous=True, log_level=rospy.D
    mira_jointmover_object = MiraBlobFollower()
    mira_jointmover_object.loop()
```

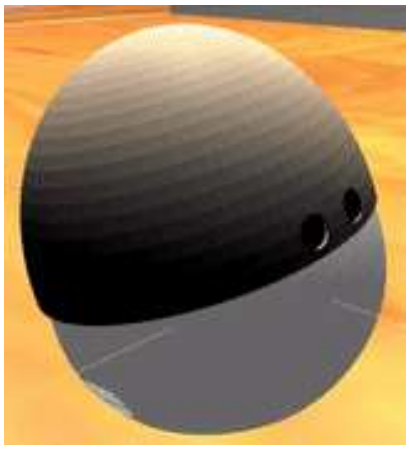
- End mira_follow_blob.py -

This last script gets the **blob detections** and **moves Mira's head** accordingly.

Mira has a topic named **/mira/commands/velocity**. Depending on what **Twist** message is published, the robot will move its head. In this case:

- Publish a speed in the **angular.x**: Moves the Roll Axis. Publish a Positive value: Moves head UP. Negative >> Down.
- Publish a speed in the **angular.z**: Moves the Yaw Axis. Publish a Positive value: Turn head left. Negative >> Turn Right.

Roll Axis Movement:



Roll Axis Movement

Yaw Axis Movement:

Yaw Axis Movement

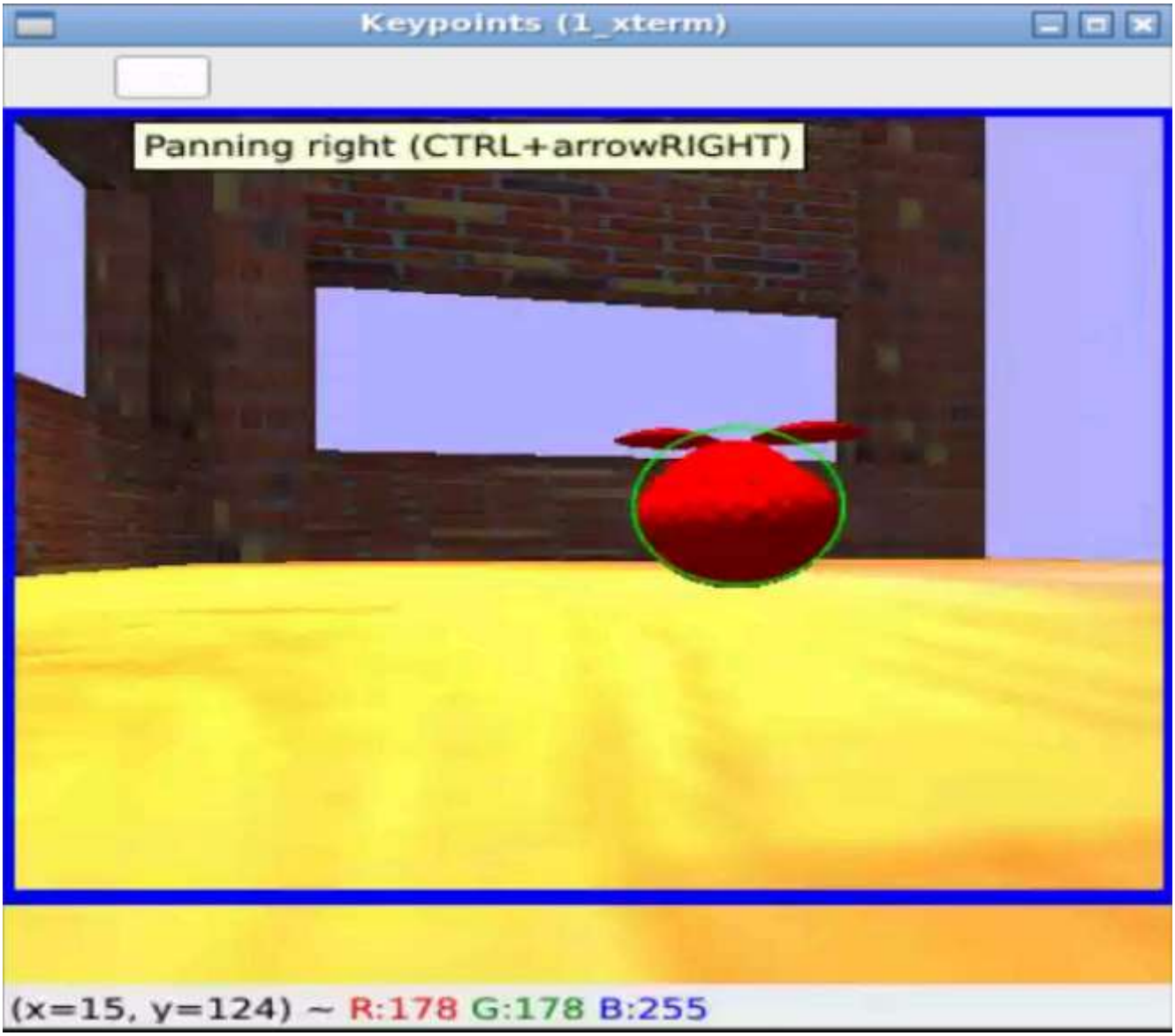
1.6 Launch and Test the Blob Tracker

Now it's time to see it in action. Launch the **blob_tracker.py**.

▶ Execute in WebShell #1

In [ ]:
```
rosrun my_blob_tracking_pkg blob_detector.py
```

Open the Graphical Interface by clicking on the Graphical Interface icon, so you can see the blob detection:



In a second terminal, launch the **teleop** for the **red-haro-robot**, and move it so that Mira can see it.

▶ Execute in WebShell #2

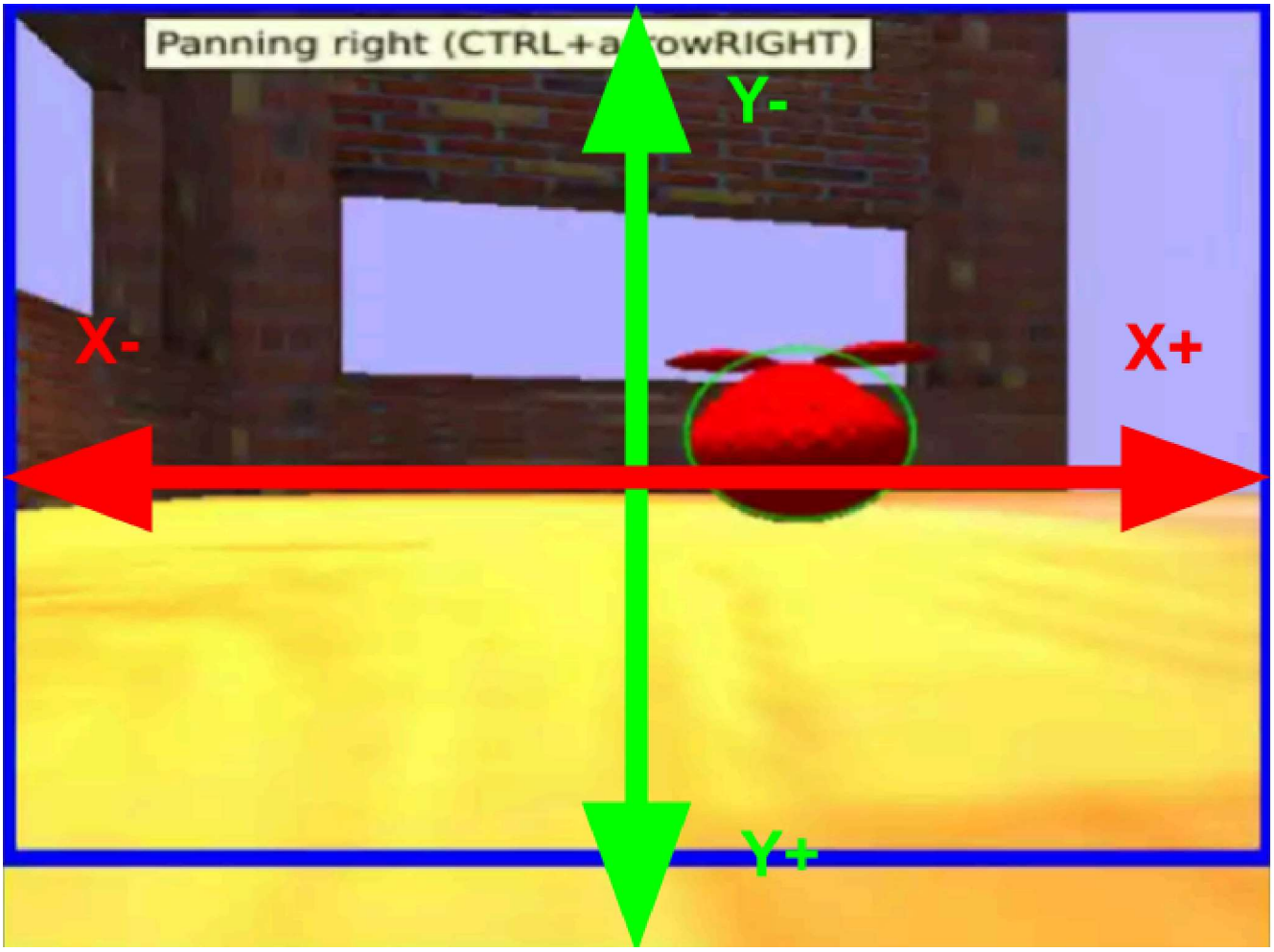
In [ ]:
```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/haro/cmd_vel
```

Check the blob topic info:

▶ Execute in WebShell #3

In [ ]:
```
rostopic list | grep /blob/point_blob
rostopic echo /blob/point_blob
```

See how the size decreases the bigger the distance. And see how x and y change based on the position. **Note** that the x and y position's values are relative to the **CENTER** of the image.

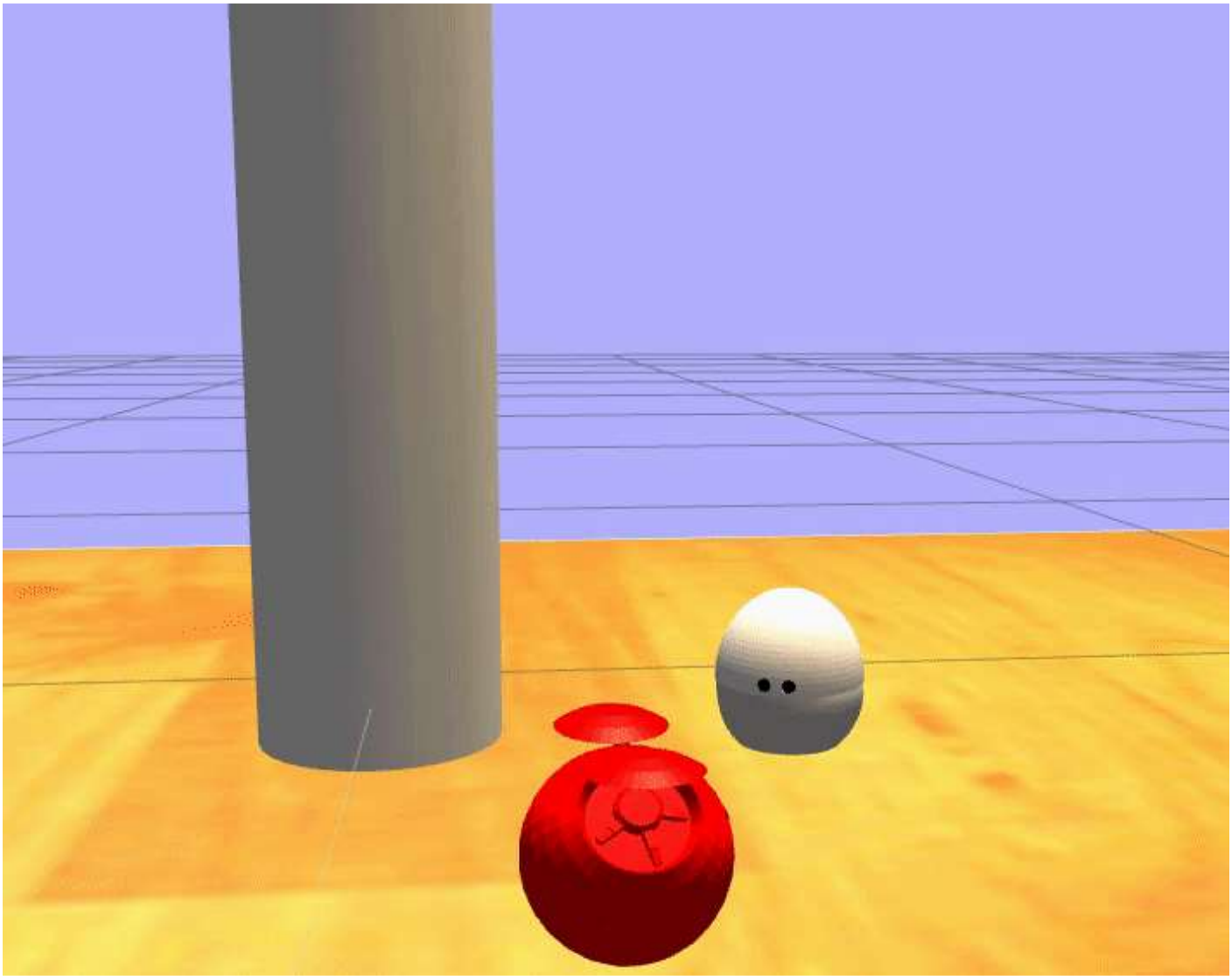


Great! Now let's run the **mira_follow_blob.py** and see how it performs:

▶ Execute in WebShell #3

In [ ]:
```
rosrun my_blob_tracking_pkg mira_follow_blob.py
```

Mira should now follow the red-haro-robot, left-right, up-down. Take into account that the robot has physical limits in the joints and cannot follow the red-haro-robot everywhere.



robotignite ACADEMY

Congratulations! You can now track anything with color. Continue to the next unit to go a little bit deeper in OpenCV into ROS and learn how to navigate following a line drawn on the ground.

Select the project unit for this course. You can now do the first project exercise.

- End Project -