ROS Perception in 5 Days

Chapter 3: Flat Surface and Object Detection

Estimated time to completion: 3 hours

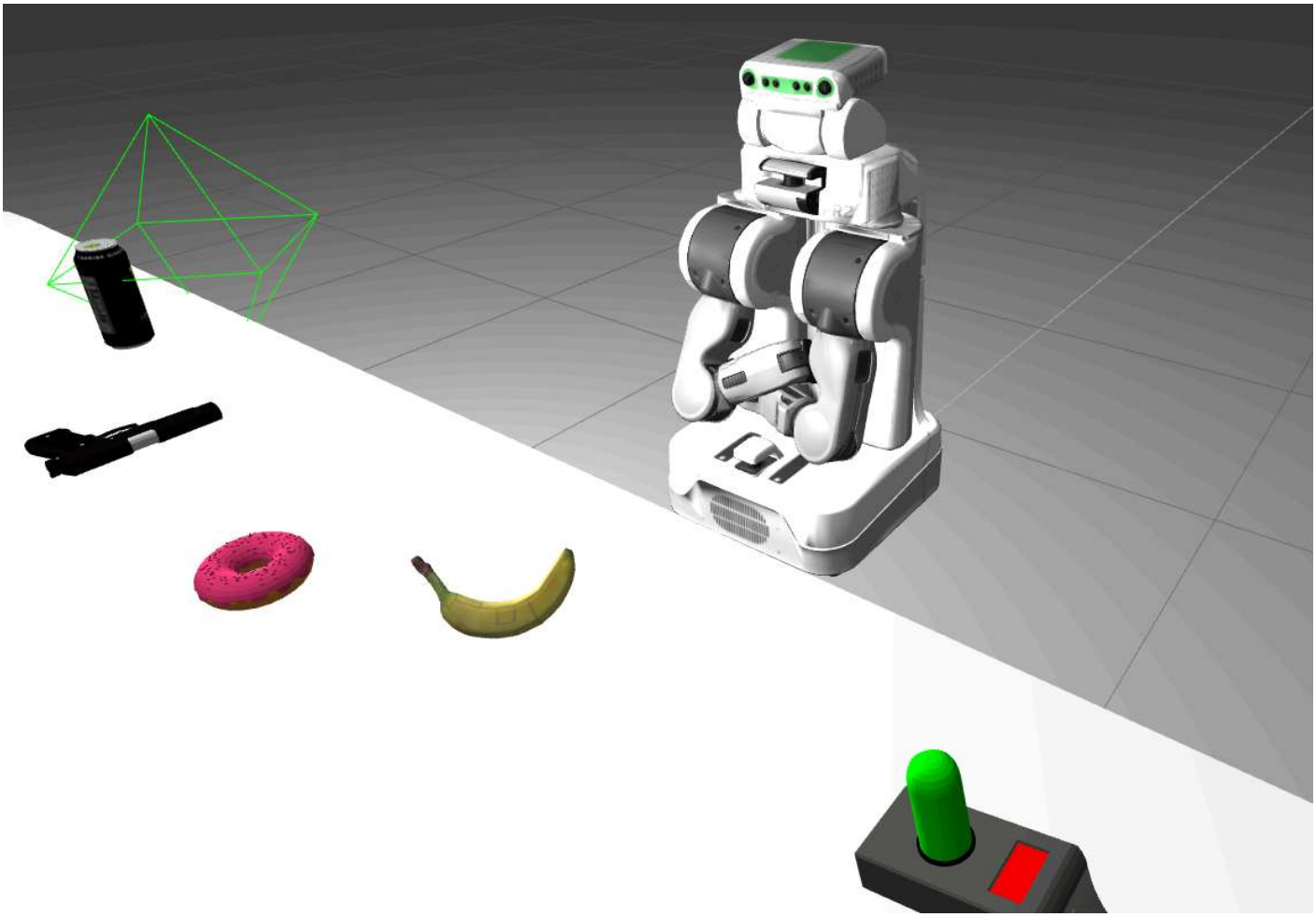
In this chapter, you will learn how to use perception skills to recognize objects. This allows you to create robots that can grasp objects and understand the world around them a little bit better. Specifically, you will master two primary skills. How to:

- **Recognize flat surfaces**: This skill allows the robot to detect places where objects typically are, like tables and shelves. It's the first step to searching for objects.
- **Recognize objects**: Once you know where to look, you have to recognize different objects in the scene and localize their position from your robot's location (frame).

3.1 Environment Introduction

You will be using the following environment:

- PR2 robot: The PR2 robot has been recovered from OSRF history for this course. It's a versatile robot that can move omnidirectionally, has two arms and a moving torso. It also has lasers and point-cloud cameras. It's a perfect candidate for object recognition and manipulation.
- Object selection on the table: You will find several objects on the table. They are to allow for variety in the detection. There is a gun among the objects because we want to explain how to detect a hazardous situation with an armed human.



You can move the PR2 robot (http://wiki.ros.org/Robots/PR2) using **keyboard teleoperation** and **joint-commands**, listed below for your use.

▶ Execute in WebShell #1

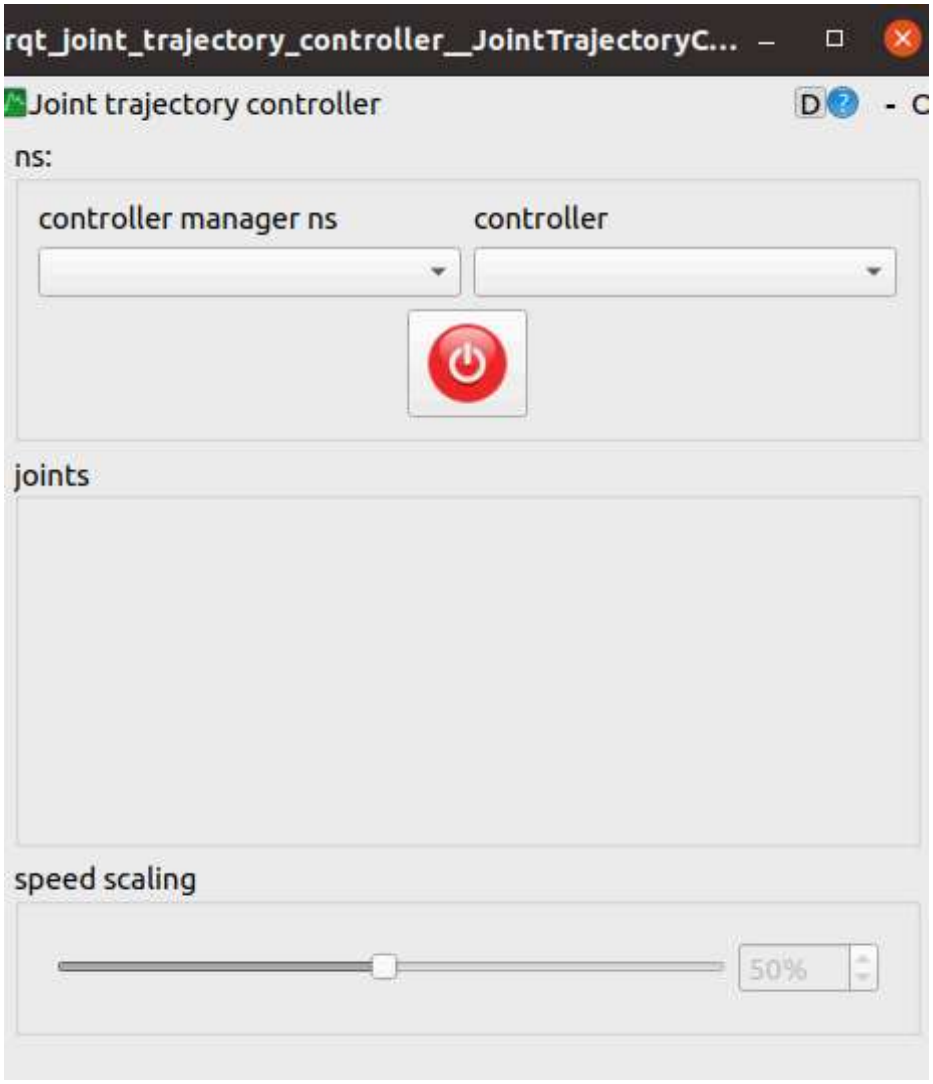
In [ ]:
```
roslaunch pr2_tc_teleop keyboard_teleop.launch
```

Increase speed to approximately **8.0** using the **Q KEY** to see the PR2 robot move.

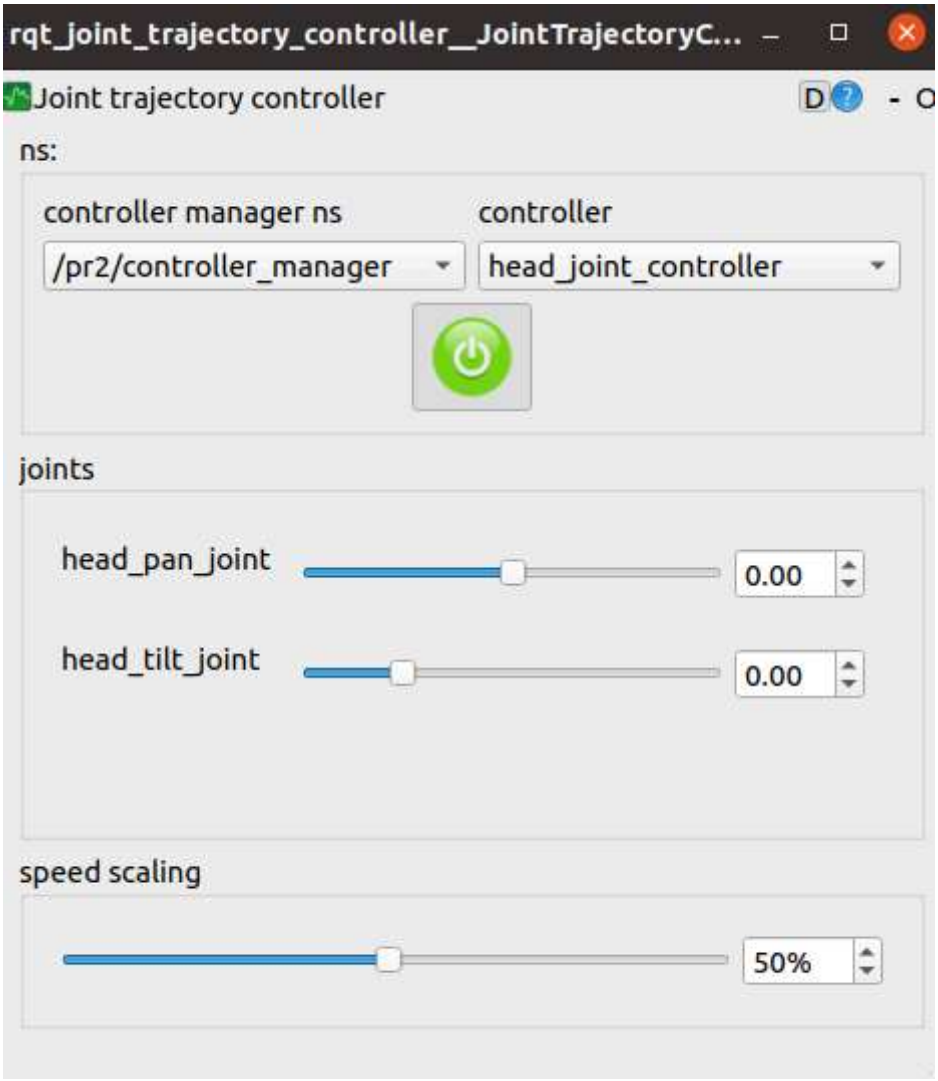
i	Move Forwards		SHIFT+i	Move Forwards
,	Move Backwards		SHIFT+,	Nothing
j	Turn Left		j	Strafe Left
l	Turn Right		l	Strafe Right
k	Stop		k	Stop
u	Move Forwards + Turning Left		SHIFT+u	Move Forwards + Strafe Left
o	Move Forwards + Turning Right		SHIFT+o	Move Forwards + Strafe Right
m	Move Backwards + Turning Left		SHIFT+m	Move Backwards + Strafe Left
.	Move Backwards + Turning Right		SHIFT+.	Move Backwards + Strafe Right
q	Increase Speed		q	Increase Speed
z	Decrease Speed		z	Decrease Speed

Click on the Graphical Interface icon and open the Graphical Tools to move the PR2 robot through a simple GUI of the robots joints:

You should see an image similar to the image below:



Select the **head_joint_controller** to move the head **pan** and **tilt**. This will prove useful as a check to ensure your object detection is working.



3.2 Flat Surface Detector

The first step in recognizing objects is knowing where these objects are located. You will use the surface_perception (https://github.com/jstnhuang/surface_perception) package to detect flat surfaces and represent the detections in RViz. The surface perception package also detects objects on a flat surface. It's an **excellent** ROS package and well made, taking into account that it hasn't been updated since ROS-indigo, and it works in **ROS-Noetic**.

The first step is to create an object recognition package:

▶ Execute in Shell #1

In [ ]:
```
cd /home/user/catkin_ws/src
catkin_create_pkg my_object_recognition_pkg rospy
cd my_object_recognition
mkdir launch
touch launch/surface_detection.launch
cd /home/user/catkin_ws
catkin_make
source devel/setup.bash
rospack profile
```

- surface_detection.launch -

In [ ]:
```
<?xml version="1.0"?>
<launch>
    <node name="surface_perception_node" pkg="surface_perception" type="demo"
        <remap from="cloud_in" to="/camera/depth_registered/points"/>
    </node>
</launch>
```

- surface_detection.launch -

This binary needs two elements to work:

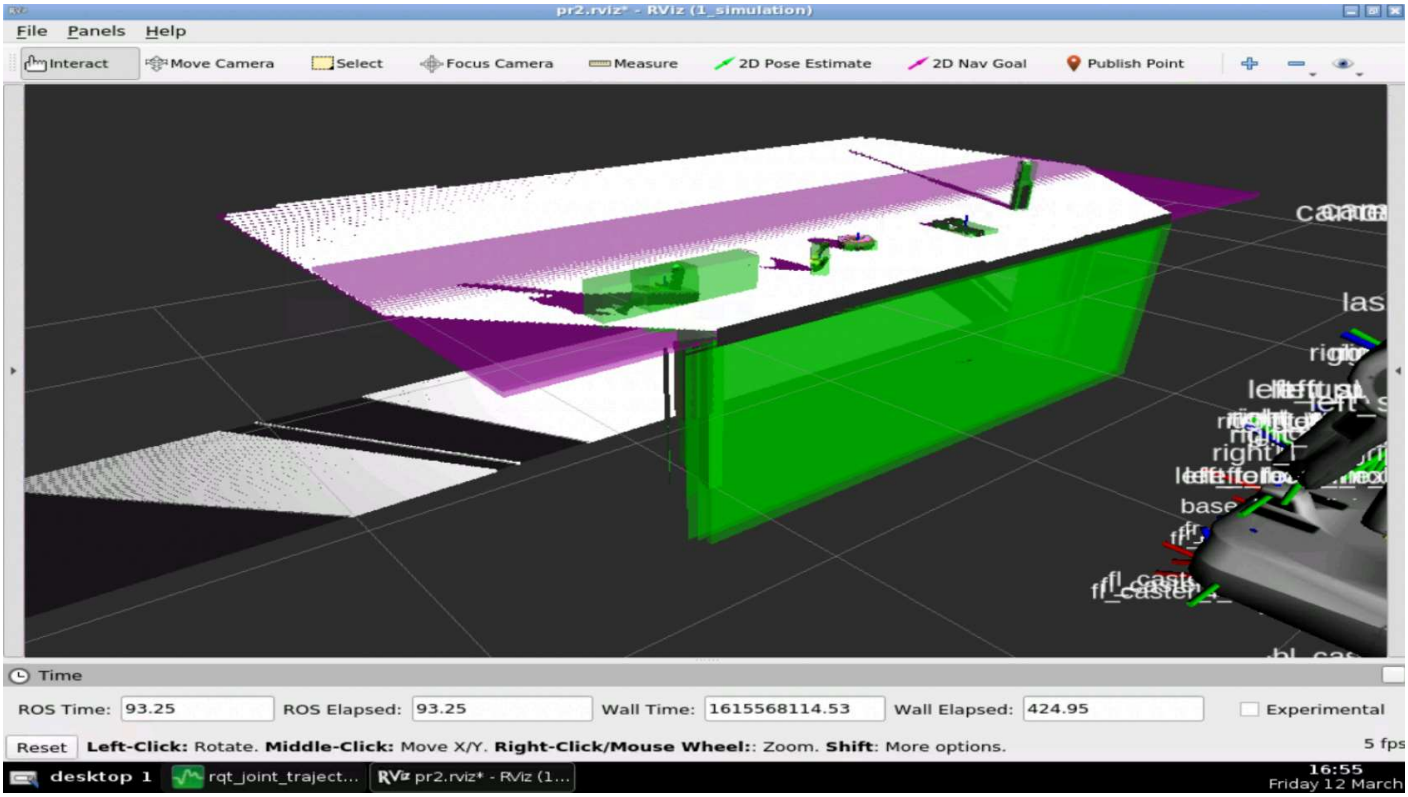
- Input the Tf frame in which we will base the detections. In this case, **base_link** will suffice.
- Remap the **cloud_in** to the topic where your robot is publishing the **point-cloud** from your depth sensor. In our case, this is **/camera/depth_registered/points**.

Let's launch the surface detection package and see what happens:

▶ Execute in Shell #1

In [ ]:
```
roslaunch my_object_recognition_pkg surface_detection.launch
```

You should now see an image similar to the image below:



If you don't see the markers, be sure that in the RViz panel, you see an element type **Marker** pointing to the topic **/surface_objects**:

You can see that with only this system, you can get:

- Markers representing the different horizontal surfaces detected by your robot. They are indicated in **purple markers**.
- Markers representing the objects detected. They are indicated in **green markers**.

- Exercise 3.2.1 -

- Create a Python script called **surface_data_extraction.py** that extracts the marker data generated by the surface_detection.launch and filters only the horizontal surface objects that correspond to the table height.
- The table height is approximately 0.8 meters, so you should filter surface objects around that height.
- You can differentiate the surface from the objects because their name is **surface_X** while the objects are **surface_X_object_Y_axes**.

Later, this data can be used to look for objects only around that region of space and used for grasping.

Remember to make the Python script executable; otherwise, ROS won't be able to execute it:

▶ Execute in Shell #1

In [ ]:
```
chmod +x surface_data_extraction.py
```

- Exercise 3.2.1 -

This code is very simple and self explanatory if you have completed the recommended prerequisite courses, **ROS Basics in 5 Days** and **ROS Python3 Basics**, or similar.

In [ ]:
```
    def update_table_height(self,new_table_height):
        self.table_height = new_table_height


    ...

    def look_for_table_surface(self, z_value):
        """

        """
        delta_min = z_value - self._error_height
        delta_max = z_value + self._error_height
        is_the_table = delta_min < self.table_height < delta_max

        return is_the_table
```

I want to comment on how we check that a detection is a table. We set a certain height value in **self.table_height** with the **update_table_height**, and then we confirm that the given z_value through **look_for_table_surface** is more or less the updated height.

By launching both the **surface detection** and this **surface_data_extraction.py**, you should get the table detections **only**.

▶ Execute in Shell #1

In [ ]:
```
cd /home/user/catkin_ws
source devel/setup.bash
rospack profile
# Now Launch
roslaunch my_object_recognition_pkg surface_detection.launch
```

▶ Execute in Shell #2

In [ ]:
```
cd /home/user/catkin_ws
source devel/setup.bash
rospack profile
# Now Launch
rosrun my_object_recognition_pkg surface_data_extraction.py
```

When the system detects a surface around the height given, you should get an output similar to the output below:

In [ ]:
```
[INFO] [1615891002.402422, 276.540000]: {'surface_1': position:
  x: 1.5979965925216675
  y: 1.1848139762878418
  z: 0.9700260162353516
orientation:
  x: 0.0
  y: 0.0
  z: 0.008941666223108768
  w: 0.9999600648880005}
```

3.3 Object Detection: Extended Object Detection

MoscowskyAnton (https://github.com/MoscowskyAnton) created this Extended Object Detection System (https://github.com/Extended-Object-Detection-ROS/extended_object_detection). The package tries to bring all the basic object and people recognition algorithms into a unified, comprehensive structure that allows for nesting detections.

It has fantastic documentation that you can check out at Wiki (https://github.com/Extended-Object-Detection-ROS/wiki_english/wiki).

These are some of the things you can do:

- Detect blobs
- Use Haar Cascades
- Use TensorFlow
- QR tracking
- Feature matching
- Basic motion detection

And much more. In this unit, we provide several examples and explain how you can combine them for creating your own detectors.

Object Detection: Simple/Complex Object Detection

This system's main idea is to use **.xml** files to define the different **attributes** you will use to detect objects. Then you combine these attributes to detect a specific **simple object**. Let's have a look at the example that we will use for this unit:

▶ Execute in Shell #1

In [ ]:
```
roscd my_object_recognition_pkg
mkdir -p config/object_base_example
roscd my_object_recognition_pkg/config/object_base_example
touch Simple_Gun.xml
```

Simple_Gun.xml

In [ ]:

```
<?xml version="1.0" ?>

<AttributeLib>
    <Attribute Name="HistColorPortalGun" Type="HistColor" Histogram="histogram
    <Attribute Name="PotalGunSize" Type="Size" MinAreaPc="0.00" MaxAreaPc="100
    <Attribute Name="NotFractal" Type="Size" MinAreaPc="0.5" MaxAreaPc="100"/>

    <Attribute Name="HSVColorBlackGun" Type="HSVColor" Hmin="0" Hmax="0" Smin=
    <Attribute Name="HaarGun" Type="HaarCascade" Cascade="gun_haar/classifier/
    <Attribute Name="MyBlobAttribute" Type="Blob" minThreshold="54" maxThreshc

</AttributeLib>

<SimpleObjectBase>

    <SimpleObject Name="PortalGun" ID="1">
        <Attribute Type="Detect">HistColorPortalGun</Attribute>
        <Attribute Type="Check">PotalGunSize</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="BlackGun" ID="2">
        <Attribute Type="Detect">HSVColorBlackGun</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="HaarBlackGun" ID="3" Mode="Hard" MergingPolicy="Union"
        <Attribute Type="Detect">HaarGun</Attribute>
        <Attribute Type="Detect">MyBlobAttribute</Attribute>
    </SimpleObject>

</SimpleObjectBase>


<RelationLib>



</RelationLib>


<ComplexObjectBase>


</ComplexObjectBase>
```

These **XML** files have **three main parts**:

- **Attributes**: Here, we define each detector. In this example, we define:
  - HistColorPortalGun: We use color histograms (that we will learn how to generate) for tracking specific color profiles (similar to blobs).
  - PortalGunSize: Of type **size** checks, the object's area is greater than (MinAreaPc *image area) and less than (MaxAreaPc* image area).
  - NotFractal: This is to avoid detecting very small or large blobs.
  - HSVColorBlackGun: Here, we are detecting not through histograms but through basic color.
  - HaarGun: We detect through a Haar Cascade for, in this case, gun morphology done by HaarGunGit (https://github.com/Saksham00799/opencv-gun-detection).
  - MyBlobAttribute: This is a blob detector in black and white. For more info on blog detectors, go to BlobDocs (https://github.com/Extended-Object-Detection-ROS/wiki_english/wiki/BlobAttribute).

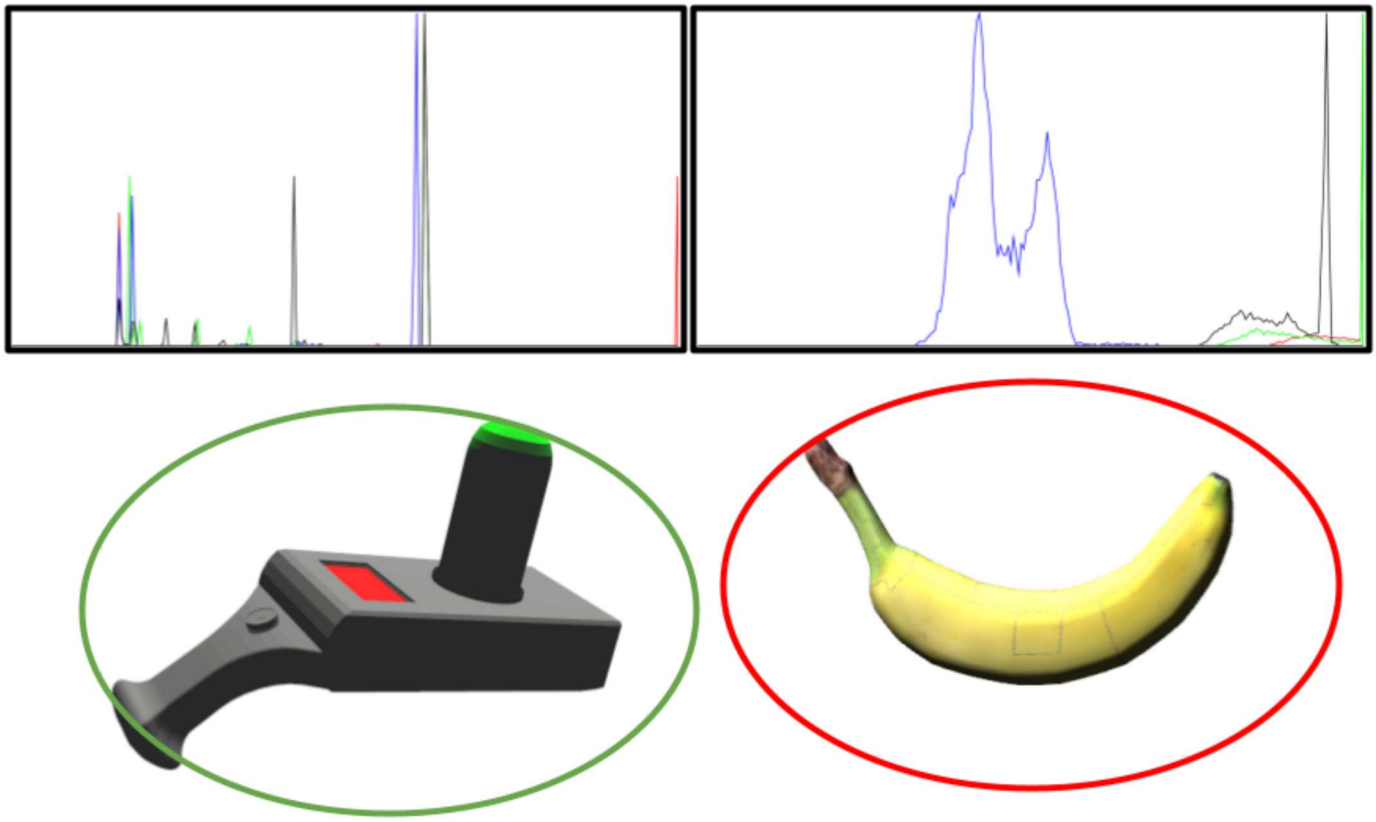
- **Simple objects**: Here, we define the main objects we want to detect, which attributes we use, and how we combine them. That's done through the **mode** (hard, soft). It's basically setting **AND** or **OR** conditions if one of the attributes detects something). And the merging policy DOCS (https://github.com/Extended-Object-Detection-ROS/wiki_english/wiki/MergingPolicySimpleObjects). This defines whether the bounding box for the attributes merges into the biggest bounding box, the union of all of them, or any other combination.

- **Relation.lib and complex objects**: These two tags go together because they define how complex objects are detected and built. Complex objects are not more than combinations of **two** or more **simple objects** with some spatial relation, like detecting **one inside of the other** or **one beside the other at a certain distance**. In this case, we are not defining any.

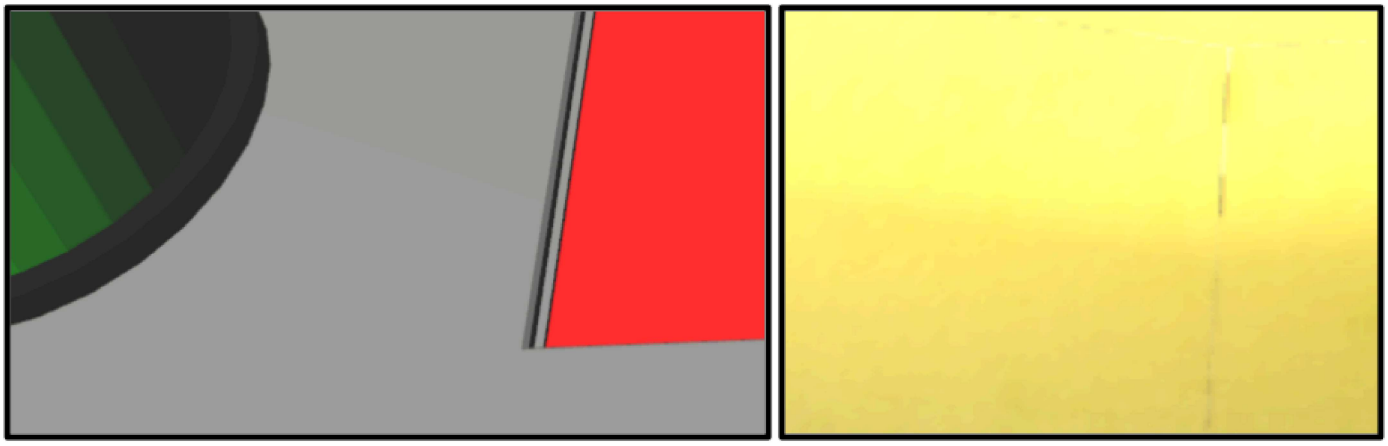
Simple Object: Portal Gun

We use **HistColorPortalGun** and **NotFractal**. How it works is **HistColorPortalGun** is set to **Detect** Type; therefore, we feed it the image data. If it detects something, then we pass it through the **check** of **NotFractal**. If it doesn't find any fractal, then the detection is okay. Otherwise, it's not valid.

In the next section, we will see how to generate all this, but here is an example of two objects that we will generate using a color histogram. You will see that it's very useful as an object **fingerprint**. As you can see, both objects have **very distinct** histograms, which makes their detection much easier.



Those graphs are the color histograms for each object. Please take into account that it's the histogram of the most **iconic region** of the objects. If we were to create the **histogram** of the images for the whole object, they would be very similar because white is predominant in both images. These are the images fed to the histogram generator:



Create the Histograms

We will use a tool that you can find inside the **Extended Object Detection package**.

▶ Execute in Shell #1

In [ ]:

```
roscd my_object_recognition_pkg/config/object_base_example
mkdir histograms
# And now we create histogram generator launch:
roscd my_object_recognition_pkg
touch launch/hist_color_params_collector_point.launch
```

- hist_color_params_collector_point.launch -

In [0]:

```
<launch>
    <arg name="output" value="screen"/>
    <arg name="hist_name" default="PortalGun"/>
    <arg name="hist_path" default="$(find my_object_recognition_pkg)/config/ob

    <node name="hist_color_params_collector_point_node" pkg="extended_object_c
        <param name="out_filename" value="$(arg hist_path)"/>
        <remap from="image_raw" to="/camera/rgb/image_raw"/>
    </node>


</launch>
```

- hist_color_params_collector_point.launch -

These are the variables that, depending on your robot and object, you will have to change:

- hist_name: The name of the histogram file you will generate.
- hist_path: Where you want the program to save your histogram file.
- /camera/rgb/image_raw: This is the topic for your RGB camera image topic. Depending on your robot, this can change.

Let's now launch it and generate the **histogram file** for the portal gun.

First, get as close as possible to the table and the object with the PR2 robot. Using your keyboard commands, **move the robot** closer to the table and lower the robot's head to better view the objects.

▶ Execute in WebShell #1

- WARNING -

To make itmove the linear speed has to be at least 10.0. So **DON'T PANIC if it doens't move**. Just press **Q** untill the linearspeed is around 10.0

- End WARNING -

In [ ]:
```
roslaunch pr2_tc_teleop keyboard_teleop.launch
```

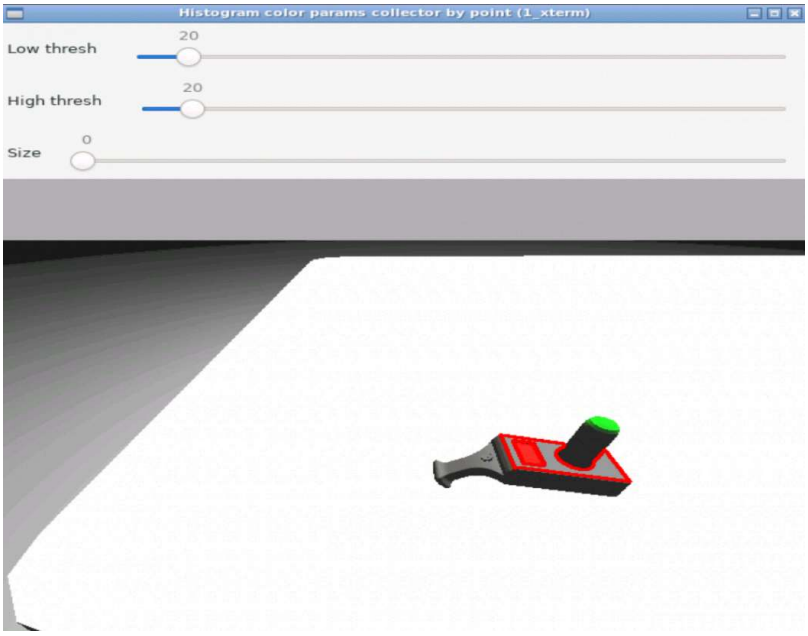
Now you can launch the **histogram generator GUI**:

▶ Execute in Shell #1

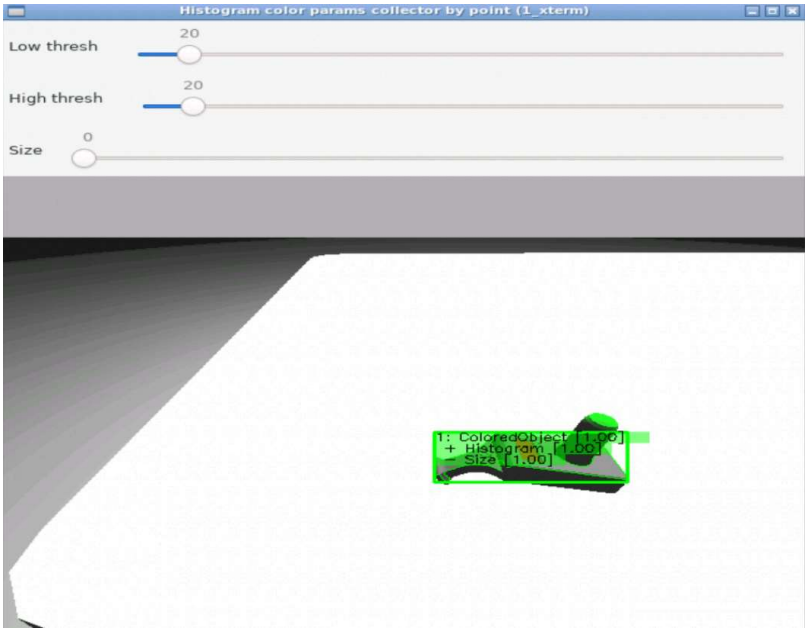
In [ ]:
```
# This variable set is because of current Academy system being remote. Locally
QT_X11_NO_MITSHM=1
echo $QT_X11_NO_MITSHM
roslaunch my_object_recognition_pkg hist_color_params_collector_point.launch
```

Click on the Graphical Interface icon to see the histogram GUI generator:

LEFT click on the most iconic color on the object. In this case, we clicked on the grey holder.



Once you see that it generated a contour that makes sense, **WITHOUT MOVING THE CAMERA**, **RIGHT click** on the image. This will fix the values and start detecting based on the histogram values. You know it worked when it turns to green, similar to the image below:



Don't worry about the **size or thesh** values. They are not used here.

If all went well, when you press the **ESC** key in **Graphical Tools**, when you have the **histogram point creator GUI window selected**, it should generate the **PortalGun.yaml** file and output these two lines.

Here it tells you what to place in the **SimpleObject.xml** file as attributes for detecting the PortalGun with histograms.

In [ ]:

```
<Attribute Name="MyHistColorAttribute" Type="HistColor" Histogram="/home/user/
<Attribute Name="MySizeAttribute" Type="Size" MinAreaPc="0.00" MaxAreaPc="100"
```

Now, let's test that it works using the **Simple_Gun.xml** file.

We need to check that the correct path for the **PortalGun Histogram** is set in **Simple_Gun.xml**. The paths can be:

- Relative: Relative to the path where you have the **Simple_Gun.xml**. So, in this case, **histograms/PortalGun.yaml**
- Absolute: **/home/user/catkin_ws/src/my_object_recognition_pkg/config/object_base_example/histograms/Portal**

Relative will always look cleaner, so let's use the **Relative** path.

We also will change the name of the **attribute** to **HistColorPortalGun**.

And we will also change the name of the size attribute to **PortalGunSize**.

You should then have something similar to below:

Simple_Gun.xml

In [ ]:

```
<?xml version="1.0" ?>

<AttributeLib>
    <Attribute Name="HistColorPortalGun" Type="HistColor" Histogram="histogram
    <Attribute Name="PotalGunSize" Type="Size" MinAreaPc="0.00" MaxAreaPc="100
    <Attribute Name="NotFractal" Type="Size" MinAreaPc="0.5" MaxAreaPc="100"/>

    <Attribute Name="HSVColorBlackGun" Type="HSVColor" Hmin="0" Hmax="0" Smin=
    <Attribute Name="HaarGun" Type="HaarCascade" Cascade="gun_haar/classifier/
    <Attribute Name="MyBlobAttribute" Type="Blob" minThreshold="54" maxThreshc

</AttributeLib>

<SimpleObjectBase>

    <SimpleObject Name="PortalGun" ID="1">
        <Attribute Type="Detect">HistColorPortalGun</Attribute>
        <Attribute Type="Check">PotalGunSize</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="BlackGun" ID="2">
        <Attribute Type="Detect">HSVColorBlackGun</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="HaarBlackGun" ID="3" Mode="Hard" MergingPolicy="Union"
        <Attribute Type="Detect">HaarGun</Attribute>
        <Attribute Type="Detect">MyBlobAttribute</Attribute>
    </SimpleObject>

</SimpleObjectBase>


<RelationLib>



</RelationLib>


<ComplexObjectBase>


</ComplexObjectBase>
```

And we create the launcher to use **Simple_Gun.xml**

▶ Execute in Shell #1

In [ ]:
```
roscd my_object_recognition_pkg
touch launch/gun_detection.launch
```

- gun_detection.launch -

In [ ]:
```
<launch>
    <arg name="output" default="screen"/>
    <arg name="objectBasePath" default="$(find my_object_recognition_pkg)/conf

    <node name="extended_object_detection" pkg="extended_object_detection" typ

        <param name="objectBasePath" value="$(arg objectBasePath)"/>
        <param name="videoProcessUpdateRate" value="5"/>
        <param name="screenOutput" value="false"/>
        <param name="publishImage" value="true"/>
        <param name="publishMarkers" value="true"/>
        <param name="subscribeDepth" value="false"/>
        <param name="maxContourPoints" value="-1"/>

        <rosparam param="selectedOnStartSimple">[1]</rosparam>
        <rosparam param="selectedOnStartComplex">[-1]</rosparam>

    </node>

</launch>
```

- gun_detection.launch -

Comments on several elements of the **gun_detection-launch** file that are worth noting:

In [0]:
```
<arg name="objectBasePath" default="$(find my_object_recognition_pkg)/config/c
```

Here we indicate the **Simple_Gun.xml** file where we will define all the simple objects and attributes.

In [ ]:
```
<rosparam param="selectedOnStartSimple">[1]</rosparam>
<rosparam param="selectedOnStartComplex">[-1]</rosparam>
```

Here we set which **simple objects** and **complex objects** will be activated and searched for in detection.

- []: If nothing is placed inside the arrays, **ALL** the simple or complex objects will be activated.
- [1,4,...,25]: Each number indicates the **ID** of the simple or complex object. For example, we are placing only **[1]**; this means that only the simple object with ID=1 will be active, in this case, **PortalGun**. **[-1]**: Means **NONE** will be activated.

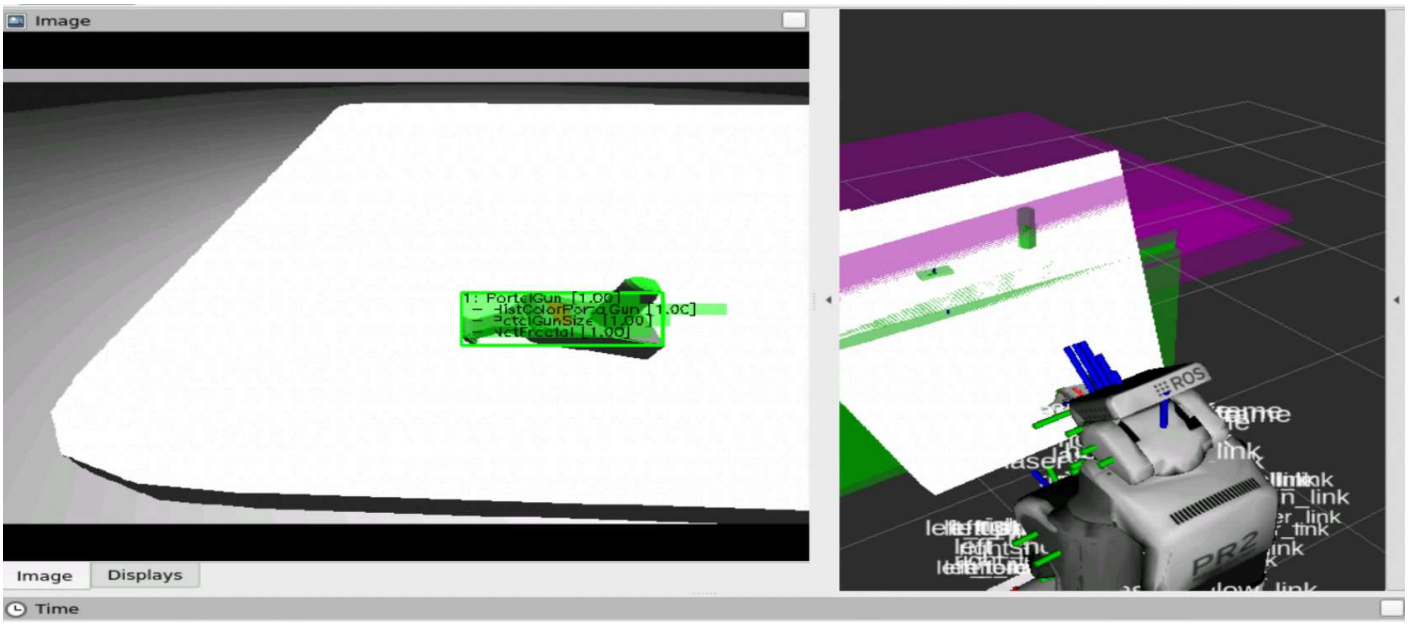
Launch it and have a look at the RViz:

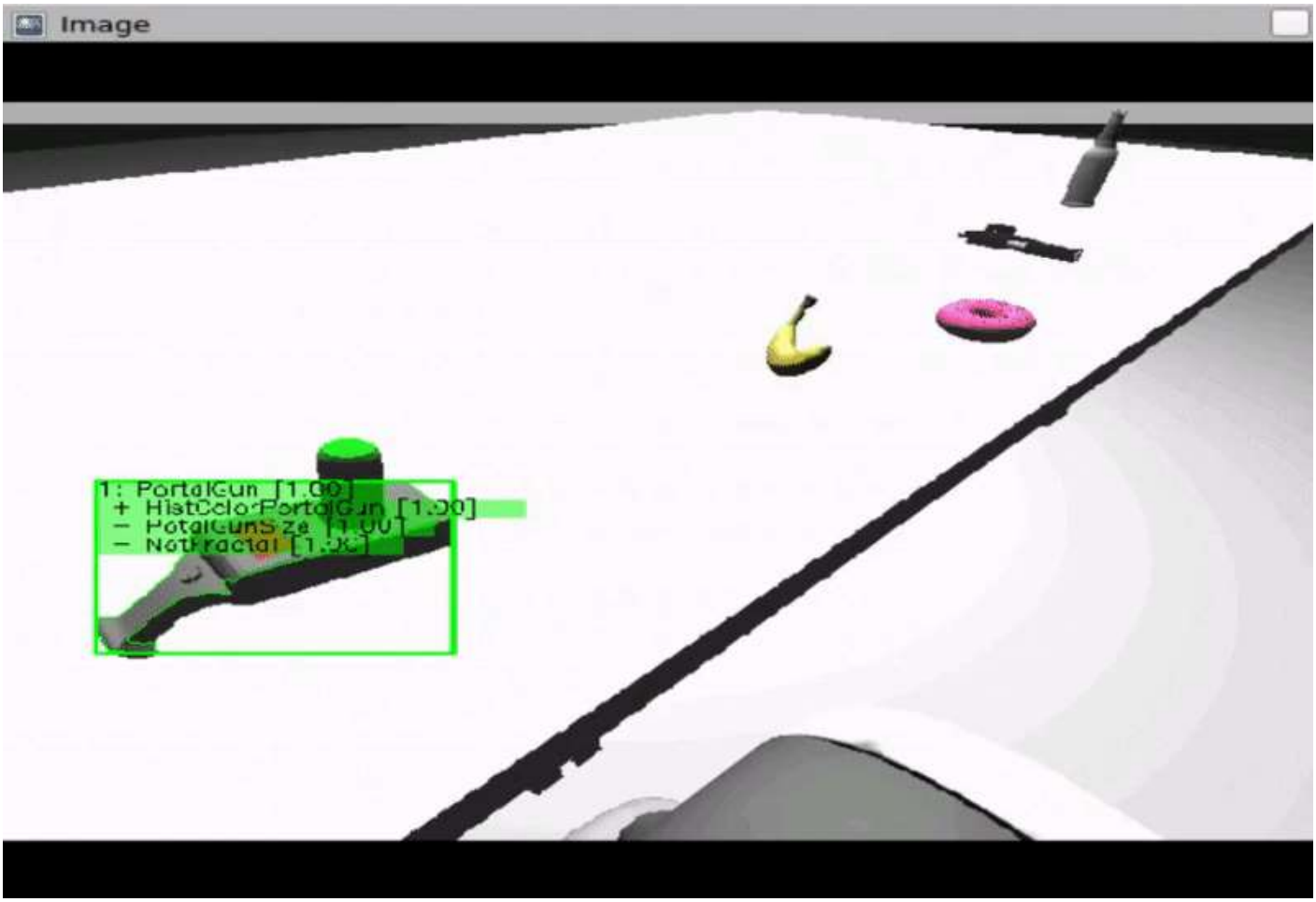
▶ Execute in Shell #1

In [ ]:
```
roslaunch my_object_recognition_pkg gun_detection.launch
```

Click on the Graphical Interface icon to see the histogram GUI generator. If your RViz doesn't appear as the one shown, you can view the RViz config file in this unit's solutions at the following location, **my_object_recognition_pkg/rviz/object_recognition.rviz**.

The detections are published in the topic /extended_object_detection/detected_image.





- Exercise 3.3.1 -

- Create a **histogram detector** for **bananas** based on the same steps we used for the **PortalGun**.
- Add the modifications to the **Simple_Gun.xml** and the **gun_detector.launch** to detect the **banana** and **PortalGun**.
- **Note**: You might have to remove the fractal attribute due to the shape of the banana.

- Exercise 3.3.1 -

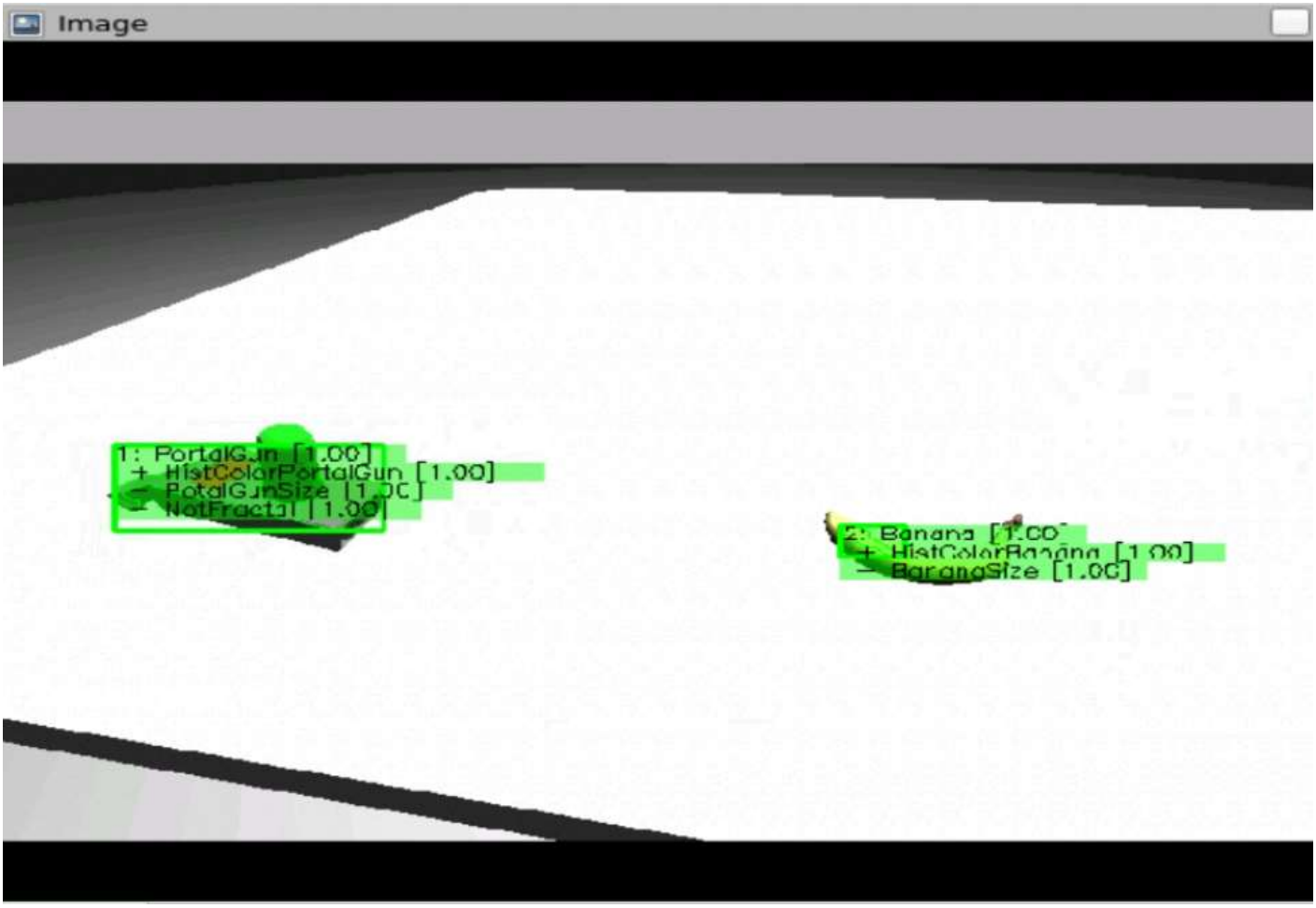
▶ Execute in Shell #1

In [ ]:
```
# This variable set is because of current Accademy system being remote. Locally
QT_X11_NO_MITSHM=1
echo $QT_X11_NO_MITSHM
roslaunch my_object_recognition_pkg hist_color_params_collector_point_banana.l
```

▶ Execute in Shell #1

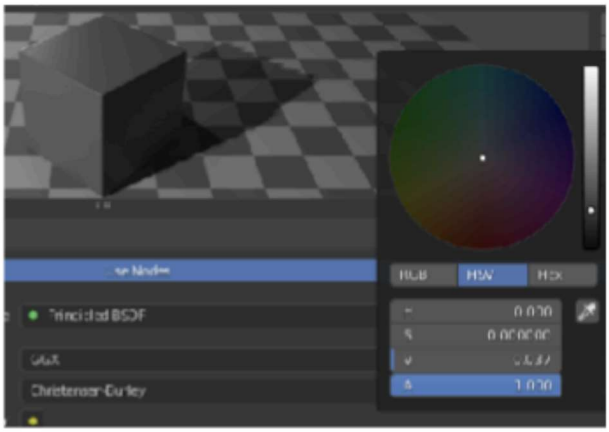
In [ ]:
```
roslaunch my_object_recognition_pkg gun_detection.launch
```

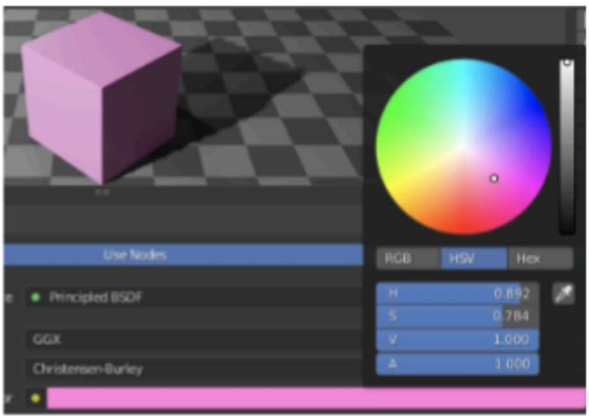
You should see something similar to the image below:



Simple Object: Black Gun

It works the same as the previous exercise but using **HSVColorBlackGun** attribute.





Create the HSV Detection

We will use a tool that you can find inside the **Extended Object Detection package**.

▶ Execute in Shell #1

In [ ]:
```
# We create a HSV sampler launch:
roscd my_object_recognition_pkg
touch launch/hsv_color_params_collector.launch
```

- hsv_color_params_collector.launch -

In [ ]:
```
<launch>
    <arg name="output" value="screen"/>

    <node name="hsv_color_params_collector_node" pkg="extended_object_detectic
        <remap from="image_raw" to="/camera/rgb/image_raw"/>
    </node>


</launch>
```

- hsv_color_params_collector.launch -

Start the launch file to get the values of HSV:

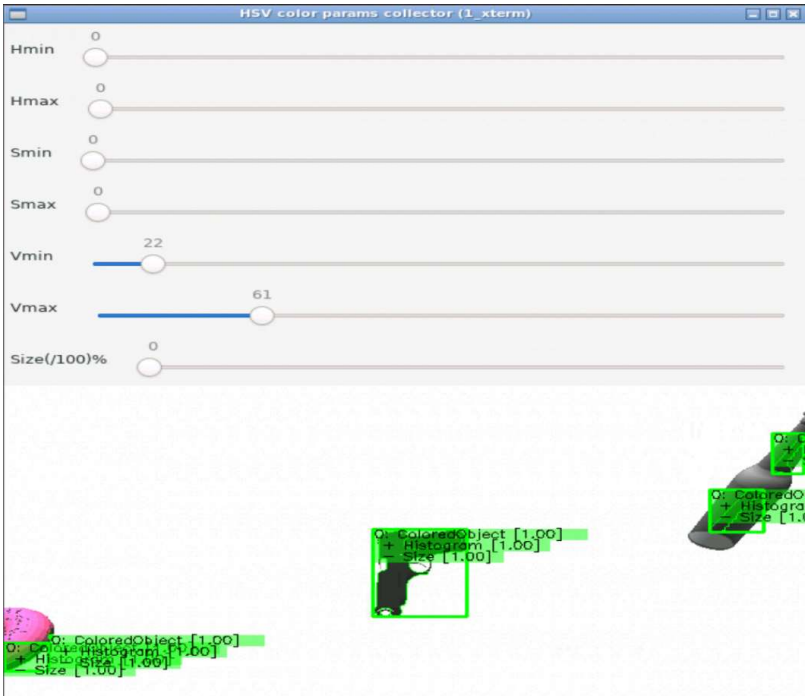
▶ Execute in Shell #1

In [ ]:
```
# This variable set is because of current Accademy system being remote. Locally
QT_X11_NO_MITSHM=1
echo $QT_X11_NO_MITSHM
roslaunch my_object_recognition_pkg hsv_color_params_collector.launch
```

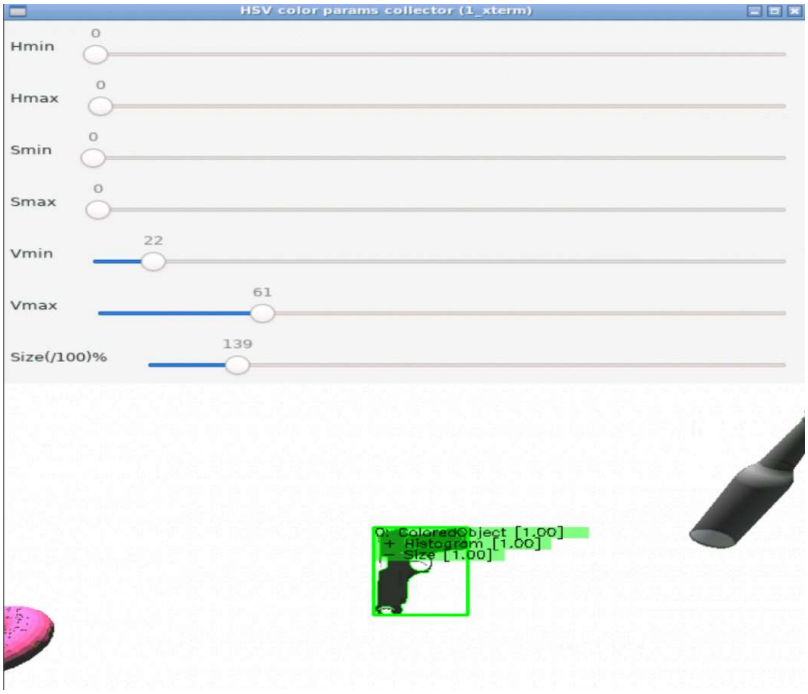
Open the Graphical Interface by clicking on the Graphical Interface icon to see the HSV GUI sampler:

As you can see, if we adjust only the **HSV** values, it will detect all the points where we have the **black-color of the gun**. Not only in the gun but in the bottle or the shadow of the donut.

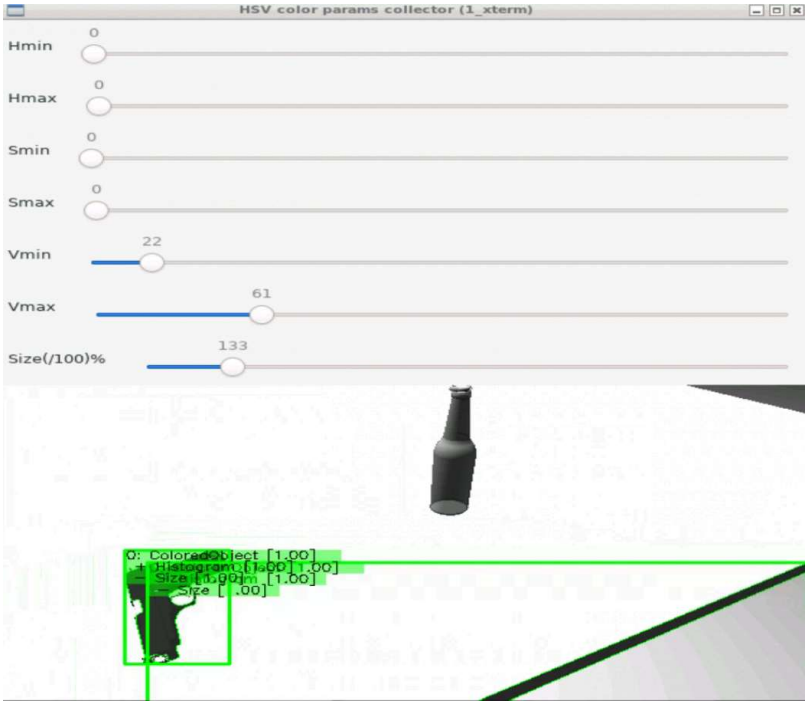
To get a preliminary idea of the values of HSV, we recommend that you use a web-based color picker, like ColorZilla (https://chrome.google.com/webstore/detail/colorzilla/bhlhnicpbhignbdhedgjhgdocnmhomnp?hl=es) or any browser add-on that allows you to get the HSV value of an image, and pick on the image of this notebook that shows the gun (several lines above this, for example).



Again, we will use the **size** attribute to detect areas that are significantly big to avoid this.



But this doesn't fix the issue that huge dark areas, like the table's side, will be detected as a gun also. To solve this, we will have to also **limit the maximum size**. But that has to be done in the **Simple_Gun.xml** attribute definition.



Once you have the values you are happy with, **press the ESC key** on the keyboard while the GUI is in focus, and the launch will terminate, outputting the **attributes** you need to add to **Simple_Gun.xml** to detect the gun with HSV.

In [ ]:

```
[ WARN] [1615908901.726154378, 4258.786000000]: YOUR ATTRIBUTES
<Attribute Name="MyHSVColorAttribute" Type="HSVColor" Hmin="0" Hmax="0" Smin="
<Attribute Name="MySizeAttribute" Type="Size" MinAreaPc="0.01" MaxAreaPc="100"
```

Simple_Gun.xml

We changed the **MaxAreaPc=20** in **BlackGunSizeAttribute** to avoid detecting the table's side.

In [ ]:

```
<?xml version="1.0" ?>

<AttributeLib>
    <Attribute Name="HistColorPortalGun" Type="HistColor" Histogram="histogram
    <Attribute Name="PotalGunSize" Type="Size" MinAreaPc="0.00" MaxAreaPc="100
    <Attribute Name="NotFractal" Type="Size" MinAreaPc="0.5" MaxAreaPc="100"/>

    <Attribute Name="HistColorBanana" Type="HistColor" Histogram="histograms/B
    <Attribute Name="BananaSize" Type="Size" MinAreaPc="0.05" MaxAreaPc="100"/


    <Attribute Name="HSVColorBlackGun" Type="HSVColor" Hmin="0" Hmax="0" Smin=
    <Attribute Name="BlackGunSizeAttribute" Type="Size" MinAreaPc="0.01" MaxAr

    <Attribute Name="HaarGun" Type="HaarCascade" Cascade="gun_haar/classifier/
    <Attribute Name="MyBlobAttribute" Type="Blob" minThreshold="54" maxThreshc

</AttributeLib>

<SimpleObjectBase>

    <SimpleObject Name="PortalGun" ID="1">
        <Attribute Type="Detect">HistColorPortalGun</Attribute>
        <Attribute Type="Check">PotalGunSize</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="Banana" ID="2">
        <Attribute Type="Detect">HistColorBanana</Attribute>
        <Attribute Type="Check">BananaSize</Attribute>

    </SimpleObject>

    <SimpleObject Name="BlackGun" ID="3">
        <Attribute Type="Detect">HSVColorBlackGun</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
        <Attribute Type="Check">BlackGunSizeAttribute</Attribute>
    </SimpleObject>

    <SimpleObject Name="HaarBlackGun" ID="4" Mode="Hard" MergingPolicy="Union"
        <Attribute Type="Detect">HaarGun</Attribute>
        <Attribute Type="Detect">MyBlobAttribute</Attribute>
    </SimpleObject>

</SimpleObjectBase>
```

```
<RelationLib>


</RelationLib>


<ComplexObjectBase>


</ComplexObjectBase>
```

gun_detection.launch

In [ ]:

```
<launch>
    <arg name="output" default="screen"/>
    <arg name="objectBasePath" default="$(find my_object_recognition_pkg)/conf

    <node name="extended_object_detection" pkg="extended_object_detection" typ

        <param name="objectBasePath" value="$(arg objectBasePath)"/>
        <param name="videoProcessUpdateRate" value="5"/>
        <param name="screenOutput" value="false"/>
        <param name="publishImage" value="true"/>
        <param name="publishMarkers" value="true"/>
        <param name="subscribeDepth" value="false"/>
        <param name="maxContourPoints" value="-1"/>

        <rosparam param="selectedOnStartSimple">[3]</rosparam>
        <rosparam param="selectedOnStartComplex">[-1]</rosparam>

    </node>

</launch>
```

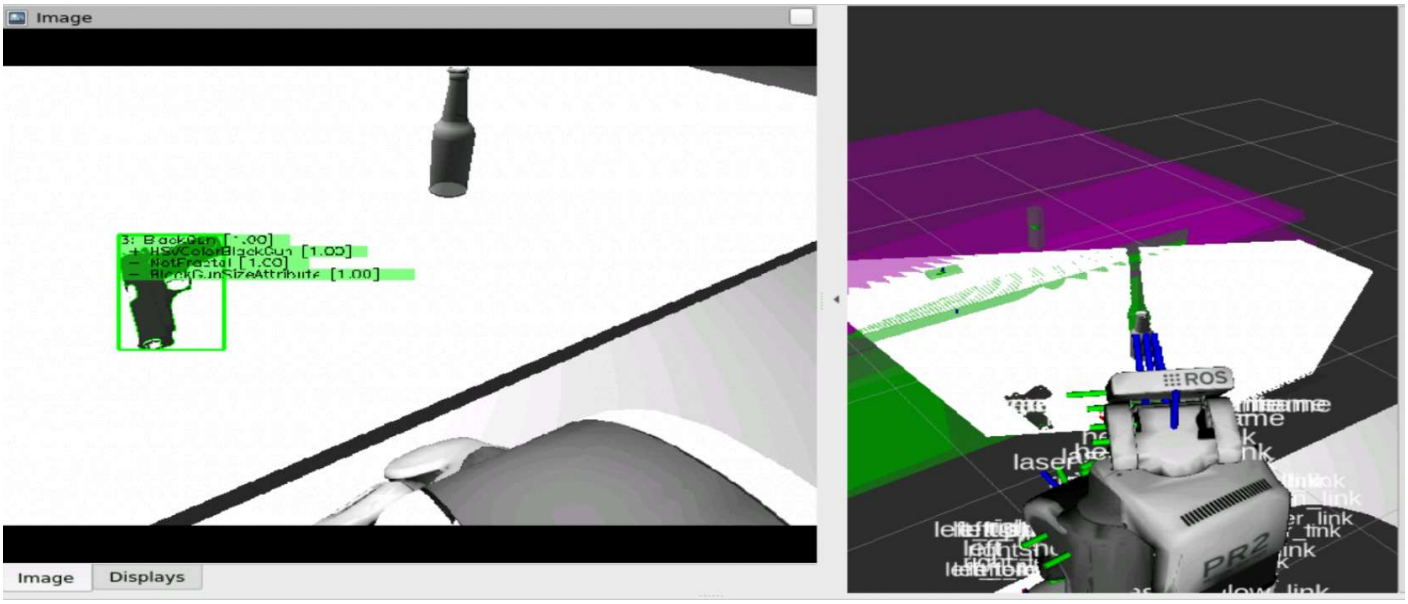
We only set the **ID=3** to clean up a bit.

▶ Execute in Shell #1

In [ ]:

```
roslaunch my_object_recognition_pkg gun_detection.launch
```

You should see that now, the table border is no longer detected. Of course, limiting the size of the detection has its drawbacks because if the robot gets **too close**, the gun will stop detecting it. You need to know your robot's normal working conditions to make these decisions.

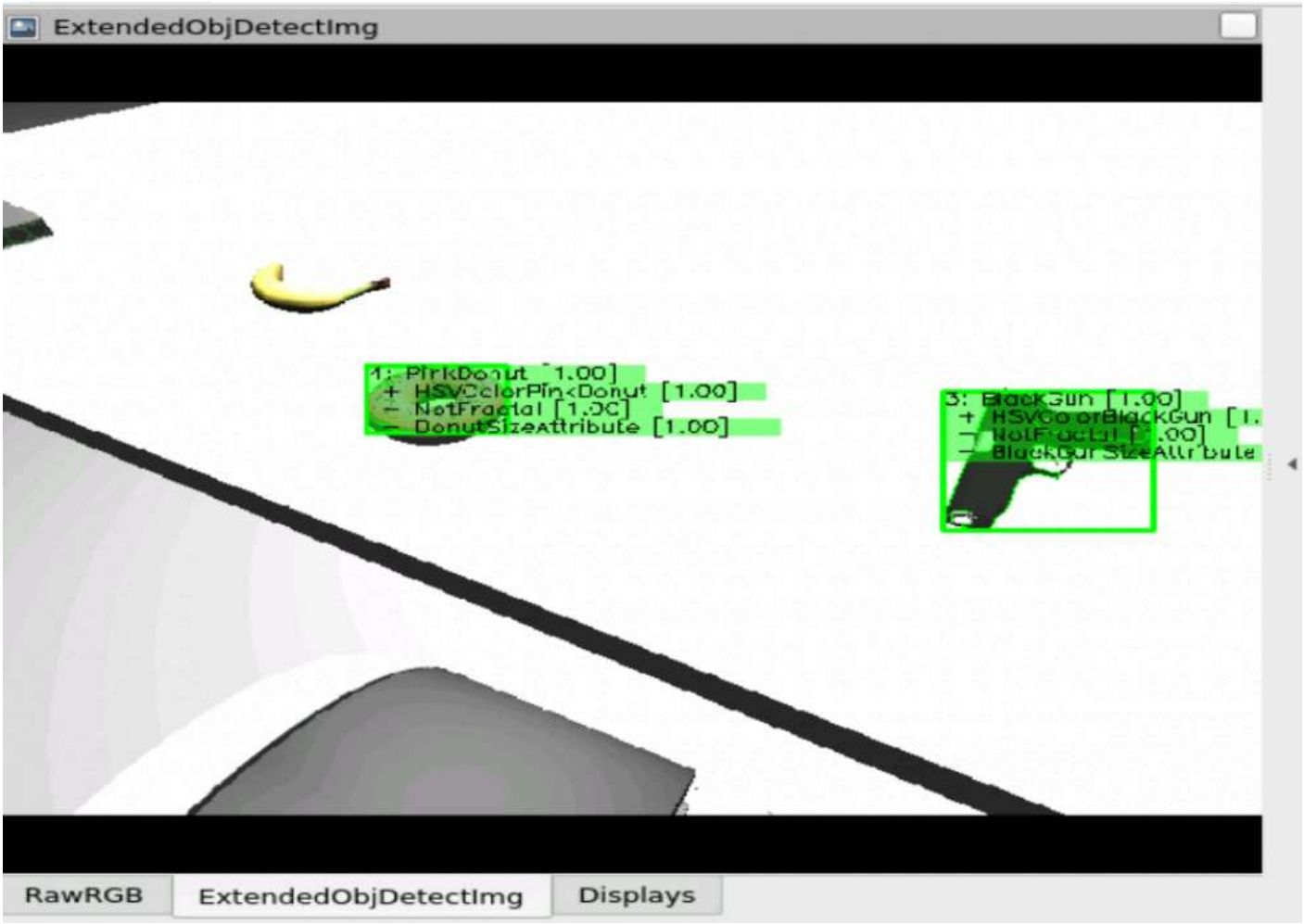


- Exercise 3.3.2 -

- Create an HSV detector for the **donut**.
- Add the modifications to the **Simple_Gun.xml** and the **gun_detector.launch** to detect the **gun** and **donut**.

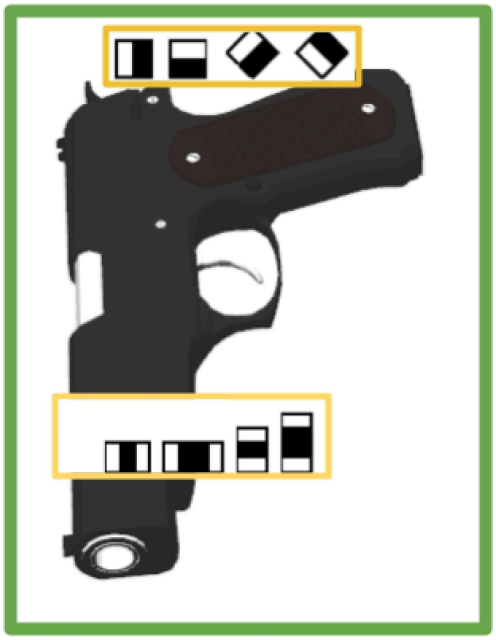
- Exercise 3.3.2 -

You should get something similar to the image below:



Simple Object: HaarBlackGun

Here, it uses the **HaarGun** attribute, and we also look for **blobs**. If **BOTH** detect something (HARD mode), we merge both bounding boxes into one. This **blob** detection was added to avoid the Haar Cascade detecting "guns" everywhere, for example, table sides, table legs, the horizon, or very small black artifacts.



Haar OK

Blob OK

Haar in a Nutshell

Basic Haar operation would be:

- We get an image and extract all the basic features (the black and white patterns shown in previous images).
- We do it in clusters or groups so that we go faster with big images.
- Then, we use Adaboost Training (https://en.wikipedia.org/wiki/AdaBoost) to select the features that give better results (detections of what we want).
- It uses cascade classifiers (machine learning essentially) to do the heavy lifting.

To learn the source of Haar, have a look at this paper: Paper Haar (https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf).

Use Haar

Download a **Haar classifier** already trained and created by Saksham00799, in this case, for detecting guns:

▶ Execute in Shell #1

In [ ]:
```
roscd my_object_recognition_pkg/config/object_base_example
git clone https://github.com/Saksham00799/opencv-gun-detection
mv opencv-gun-detection gun_haar
```

Modify the **Simple_Gun.xml** to find the **Haar** files:

Simple_Gun.xml

In [ ]:

```
<?xml version="1.0" ?>

<AttributeLib>
    <Attribute Name="HistColorPortalGun" Type="HistColor" Histogram="histogram
    <Attribute Name="PotalGunSize" Type="Size" MinAreaPc="0.00" MaxAreaPc="100
    <Attribute Name="NotFractal" Type="Size" MinAreaPc="0.5" MaxAreaPc="100"/>

    <Attribute Name="HistColorBanana" Type="HistColor" Histogram="histograms/B
    <Attribute Name="BananaSize" Type="Size" MinAreaPc="0.05" MaxAreaPc="100"/


    <Attribute Name="HSVColorBlackGun" Type="HSVColor" Hmin="0" Hmax="0" Smin=
    <Attribute Name="BlackGunSizeAttribute" Type="Size" MinAreaPc="0.01" MaxAr


    <Attribute Name="HSVColorPinkDonut" Type="HSVColor" Hmin="68" Hmax="179" S
    <Attribute Name="DonutSizeAttribute" Type="Size" MinAreaPc="0.00" MaxAreaP

    <Attribute Name="HaarGun" Type="HaarCascade" Cascade="gun_haar/classifier/
    <Attribute Name="MyBlobAttribute" Type="Blob" minThreshold="54" maxThresho

</AttributeLib>

<SimpleObjectBase>

    <SimpleObject Name="PortalGun" ID="1">
        <Attribute Type="Detect">HistColorPortalGun</Attribute>
        <Attribute Type="Check">PotalGunSize</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
    </SimpleObject>

    <SimpleObject Name="Banana" ID="2">
        <Attribute Type="Detect">HistColorBanana</Attribute>
        <Attribute Type="Check">BananaSize</Attribute>

    </SimpleObject>

    <SimpleObject Name="BlackGun" ID="3">
        <Attribute Type="Detect">HSVColorBlackGun</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
        <Attribute Type="Check">BlackGunSizeAttribute</Attribute>
    </SimpleObject>
```

```
    <SimpleObject Name="PinkDonut" ID="4">
        <Attribute Type="Detect">HSVColorPinkDonut</Attribute>
        <Attribute Type="Check">NotFractal</Attribute>
        <Attribute Type="Check">DonutSizeAttribute</Attribute>
    </SimpleObject>

    <SimpleObject Name="HaarBlackGun" ID="5" Mode="Hard" MergingPolicy="Union"
        <Attribute Type="Detect">HaarGun</Attribute>
        <Attribute Type="Detect">MyBlobAttribute</Attribute>
    </SimpleObject>

</SimpleObjectBase>


<RelationLib>



</RelationLib>


<ComplexObjectBase>


</ComplexObjectBase>
```

gun_detection.launch

In [ ]:
```
<launch>
    <arg name="output" default="screen"/>
    <arg name="objectBasePath" default="$(find my_object_recognition_pkg)/conf

    <node name="extended_object_detection" pkg="extended_object_detection" typ

        <param name="objectBasePath" value="$(arg objectBasePath)"/>
        <param name="videoProcessUpdateRate" value="5"/>
        <param name="screenOutput" value="false"/>
        <param name="publishImage" value="true"/>
        <param name="publishMarkers" value="true"/>
        <param name="subscribeDepth" value="false"/>
        <param name="maxContourPoints" value="-1"/>

        <rosparam param="selectedOnStartSimple">[5]</rosparam>
        <rosparam param="selectedOnStartComplex">[-1]</rosparam>

    </node>
```

Launch the detector and see how it performs:

▶ Execute in Shell #1

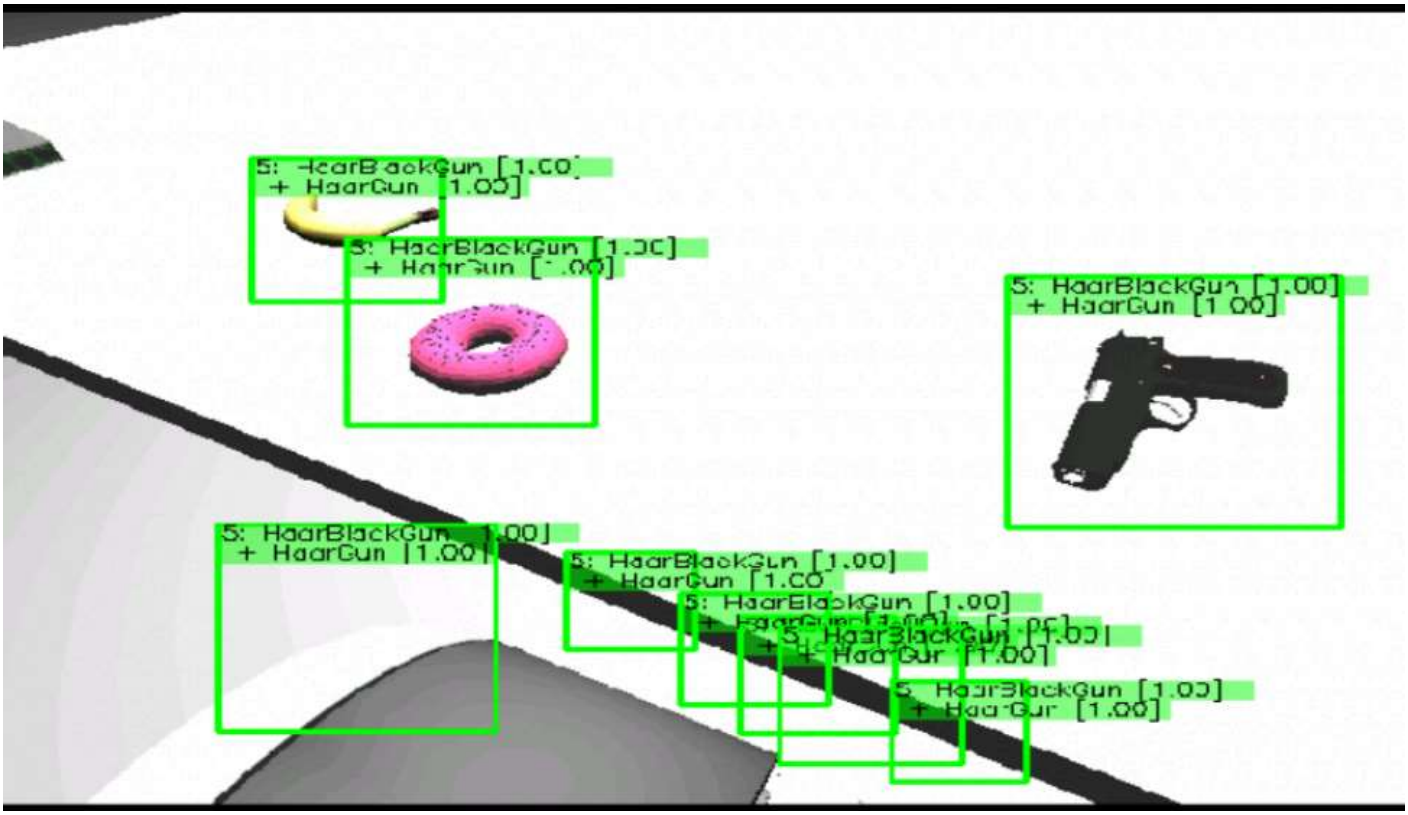
In [ ]:
```
roslaunch my_object_recognition_pkg gun_detection.launch
```

Open the Graphical Interface by clicking on the Graphical Interface icon to see the how the gun classifier performs:

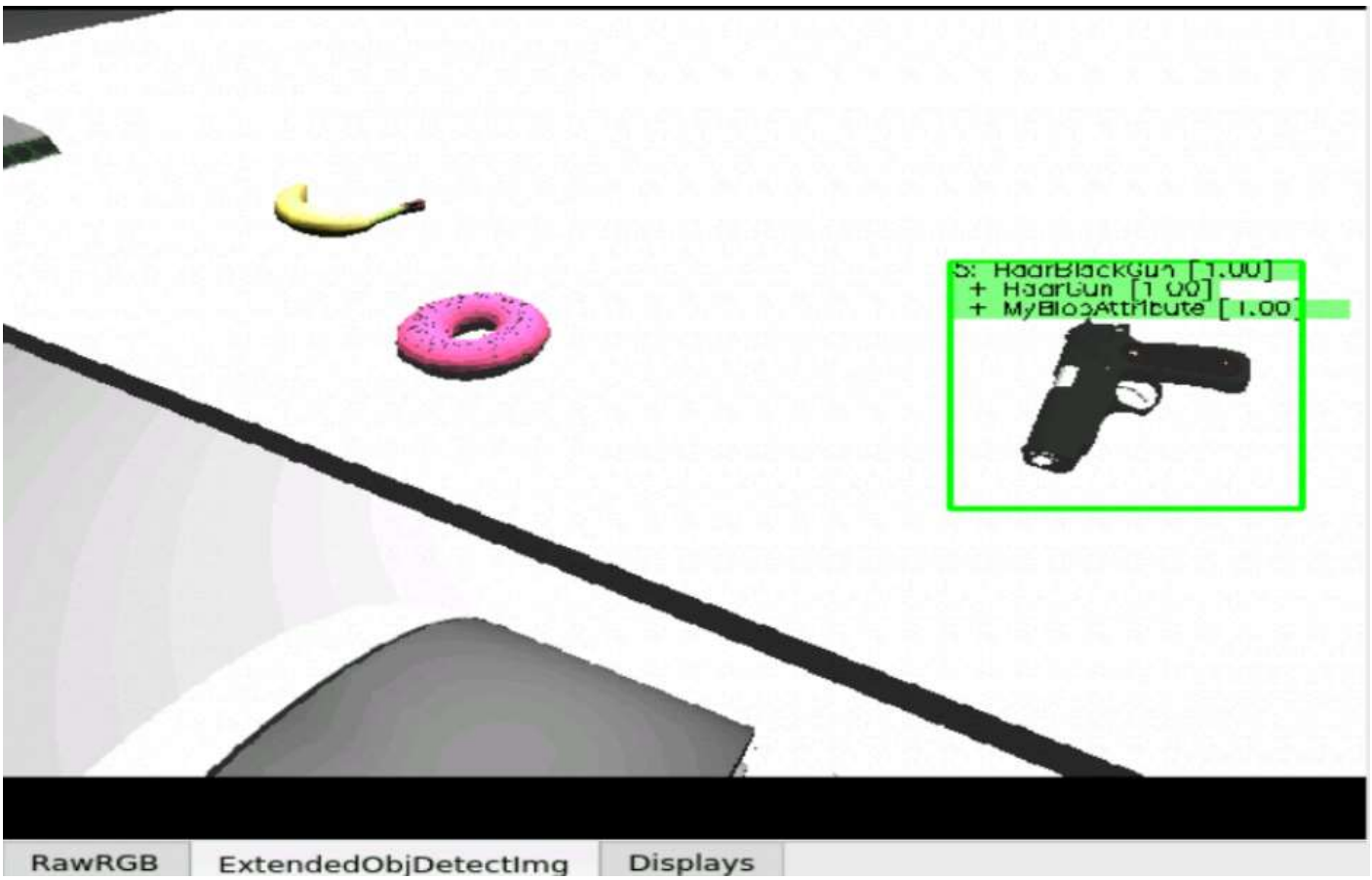
This is **WITHOUT** the attribute:

In [ ]:
```
<Attribute Type="Detect">MyBlobAttribute</Attribute>
```

As you can see, the **Haar** detector **sees** lots of possible guns on the **donut**, **banana**, and even on the **side of the table**.



That's why we add the **MyBlobAttribute attribute**. This limits the detections to only the ones that are big **black and white** blobs.



In [ ]:

```
```

- Exercise 3.3.3 -

- Download a **Haar** bottle classifier and see if it works with the monster_can drink.
- You can download it from https://github.com/jemgunay/bottle-classifier.git (https://github.com/jemgunay/bottle-classifier.git).

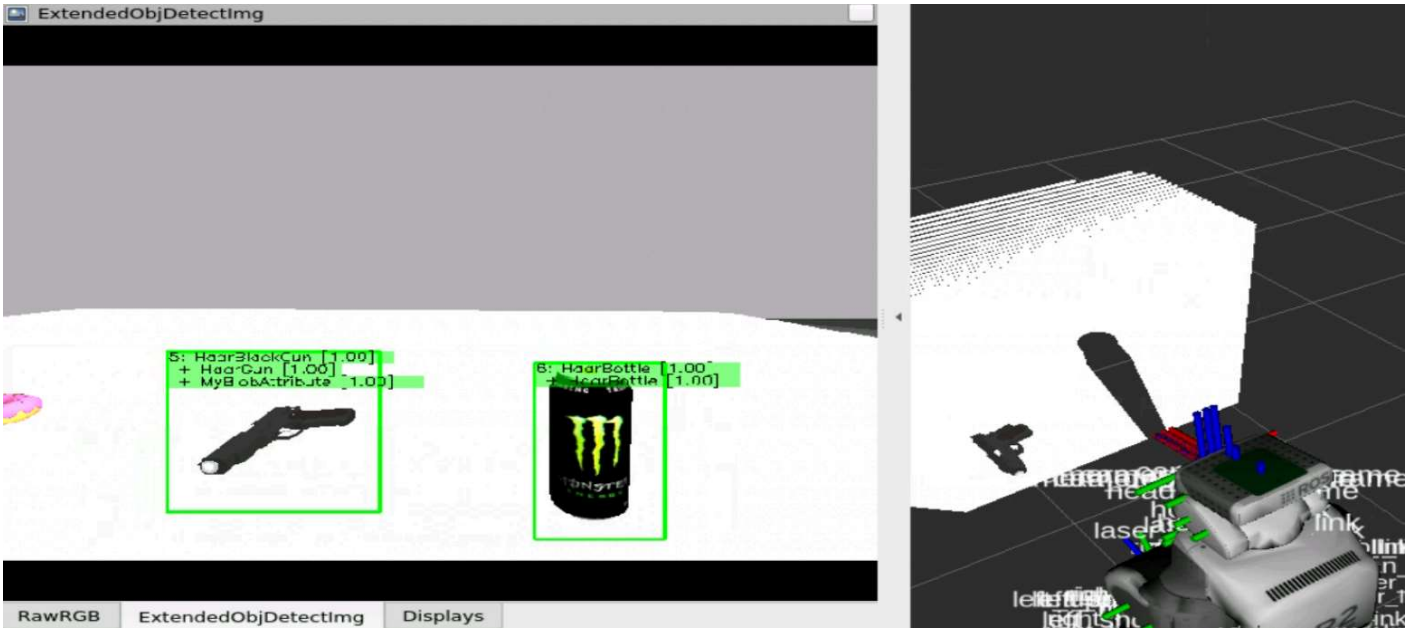
In [ ]:

```
roscd my_object_recognition_pkg/config/object_base_example
git clone https://github.com/jemgunay/bottle-classifier.git
```

- Also, note that the classifier works best when you are as close as possible and with the PR2 robot's torso lowered at maximum. Use the controller GUI to lower the torso and tilt the head a bit down for best performance.

- Exercise 3.3.3 -

Note, in this case, we are **NOT** using the **blob**. This is because it wasn't needed to make the detection and there are no false positives. If everything went okay, you should get something similar to the image below:



Complex Object:

In this unit, we won't use complex objects, but in the **project**, we will. It essentially is combining **simple objects** to detect more complex ones, for example, **person with gun**. But that is for the final project.

See below an example of how it could be done:

ArmedPerson.xml

In [ ]:

```
<?xml version="1.0" ?>

<AttributeLib>
    <Attribute Name="HaarGun" Type="HaarCascade" Cascade="gun_haar/classifier/
    <Attribute Name="MyBlobAttribute" Type="Blob" minThreshold="54" maxThreshc

    <Attribute Name="CnnPerson" Type="Dnn" framework="tensorflow" weights="ssc


</AttributeLib>

<SimpleObjectBase>

    <SimpleObject Name="HaarBlackGun" ID="3" Mode="Hard" MergingPolicy="Union"
        <Attribute Type="Detect">HaarGun</Attribute>
        <Attribute Type="Detect">MyBlobAttribute</Attribute>
    </SimpleObject>

    <SimpleObject Name="CnnPerson" ID="67">
        <Attribute Type="Detect">CnnPerson</Attribute>
    </SimpleObject>

</SimpleObjectBase>


<RelationLib>

        <RelationShip Type="SpaceIn" Name="in"/>

</RelationLib>


<ComplexObjectBase>

    <ComplexObject ID="10" Name="ArmedPerson">
        <SimpleObject Class="CnnPerson" InnerName="Person"/>
        <SimpleObject Class="HaarBlackGun" InnerName="Gun"/>

        <Relation Obj1="Gun" Obj2="Person" Relationship="in"/>

    </ComplexObject>
```

```
</ComplexObjectBase>
```

Congratulations! You now know how to recognize objects using histograms, color profiles, and Haar Cascades. In the next chapter, you will learn how to recognize and locate 3D space objects using YOLO Machine Learning.

Project

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

END Project