Lab Guide
Object Detection

Background

The objective of the object detection lab guide is to explore different methods of object detection and classification, as well as using detection results for post-processing. Prior to starting this lab guide, please review the following concept reviews:

- Concept Review – Color Spaces
- Concept Review – Image Filters

Considerations for this lab guide:

Object detection requires good understanding of image processing techniques. You can refer to Concept Review – Color Spaces for the fundamental composition of camera outputs, and Concept Review – Image Filters for the image cleanup techniques.

This lab will guide you through the 2 pipelines shown below:

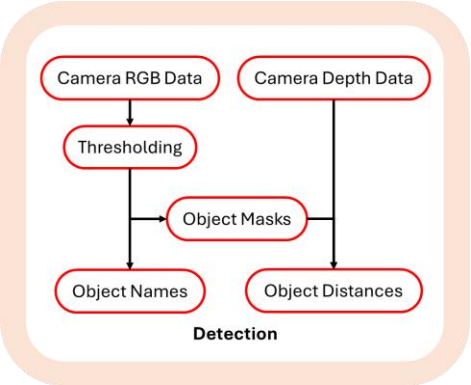

Figure 1: Detection

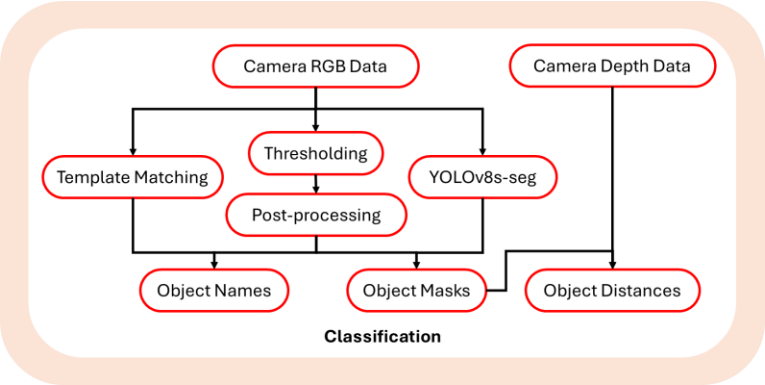

Figure 2: Classification

- **Detection,** in a narrow sense, only entails determining whether an object is in the image. Students will be guided to detect objects through color-based methods.
- **Classification** introduces multiple methods to students, from simple geometry-based methods to more advanced neural network-based methods.

Skills Activity - Setup

Physical setup -

If you're using the physical car, you will need a physical stop sign, a yield sign, and a traffic light, or photos of these objects to show in front of the camera. You can place the QCar2 on the tabletop as this lab is entirely static.

Review the User Manual – Connectivity for establishing a network connection and transferring files to the QCar2. Transfer **object_detection.py** to QCar2. It is recommended that you connect to the QCar via Remote Desktop for interfacing with this Skills Activity and use VS code on the QCar to further develop the scripts. If not already installed, refer to User Manual – Software Python for instructions to install VS code.

After completing each section, run your code using the following command in the terminal:

```
python object_detection.py
```

Virtual setup -

If you're doing this skills activity virtually, open Quanser Interactive Lab, navigate to the "Self-Driving Car Studio" and select "Cityscape". Once the virtual world is loaded, execute **virtual_road_signage.py** to define and spawn the virtual signs and QCar2.

```
python virtual_road_signage.py
```

Follow the prompts in the terminal to specify the configuration of the skills activity.

You will also be able to move and rotate the virtual QCar2 and change the weather to adjust the views and the lighting conditions of the traffic signs. To do so, follow the prompts from the script **virtual_road_signage.py**.

After completing each section, run your code using the following command in a separate terminal:

```
python object_detection.py
```

Skills Activity – Detection

The objective of this section is to apply image processing techniques to detect traffic lights and stop signs. Then, perform post-processing to extract actionable data from detection results.

Step 1 – RGB Image Thresholding

Before using color to detect objects, it is crucial to determine the RGB values of the desired color. In the "Experiment Configuration" region, set ***task*** = "threshold" and ***mode*** = "rgb". To help the process of determining the RGB values, a tool is partially developed. Navigate to the **color_thresholding()** method (right click and select "Go to Definition" or find it in **qcar_function.py** in hal/content directory). Complete **SECTION A.1** to generate a binary mask using lower/upper RGB bounds.

```
        # =================  SECTION A.1 - Thresholding Mask  =================
        mask = np.zeros(img.shape[:2],np.uint8)
        return mask
        # =================         End of SECTION A.1         =================
```

Then, navigate to the **mask_img()** method, complete **SECTION A.2** to apply the mask.

```
        # =================    SECTION A.2 - Mask Image     =================
        img_thresh = np.zeros(img.shape,np.uint8)
        return img_thresh
        # =================       End of SECTION A.2        =================
```

Results:

Start by setting all "high" threshold to 255 to show the unfiltered image. You can hover your mouse over the image to see the RGB values of the pixels. Adjust the RGB sliders in the "Image" window until the target object (stop sign or the traffic light) is clearly segmented as shown in Figure 3 for physical setup and Figure 4 for virtual setup. The segmented camera feed does not need to be perfect at this point, the detection result will be improved in later sections.

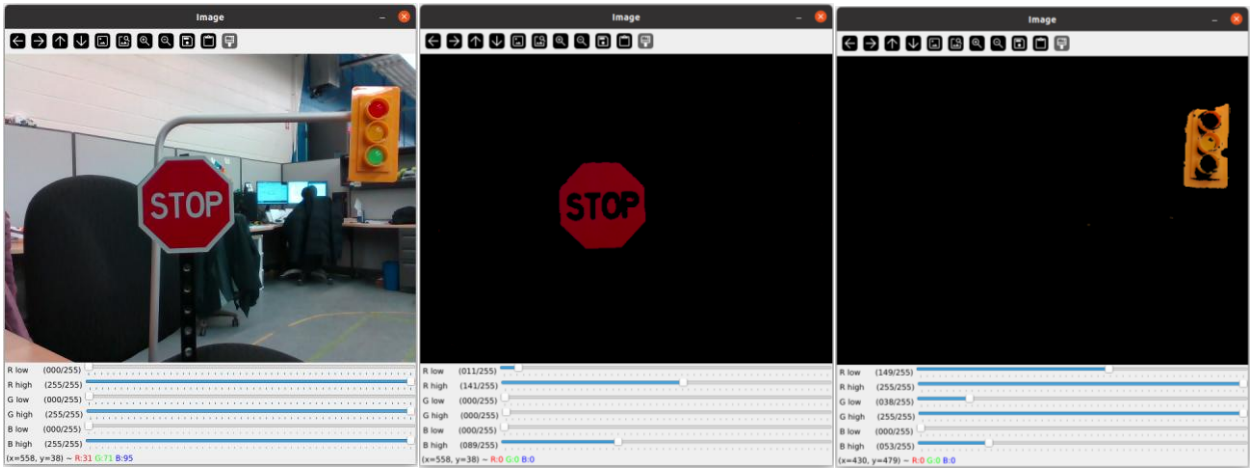

Figure 3. Raw camera feed (left), segmenting red (middle), segmenting yellow (right).

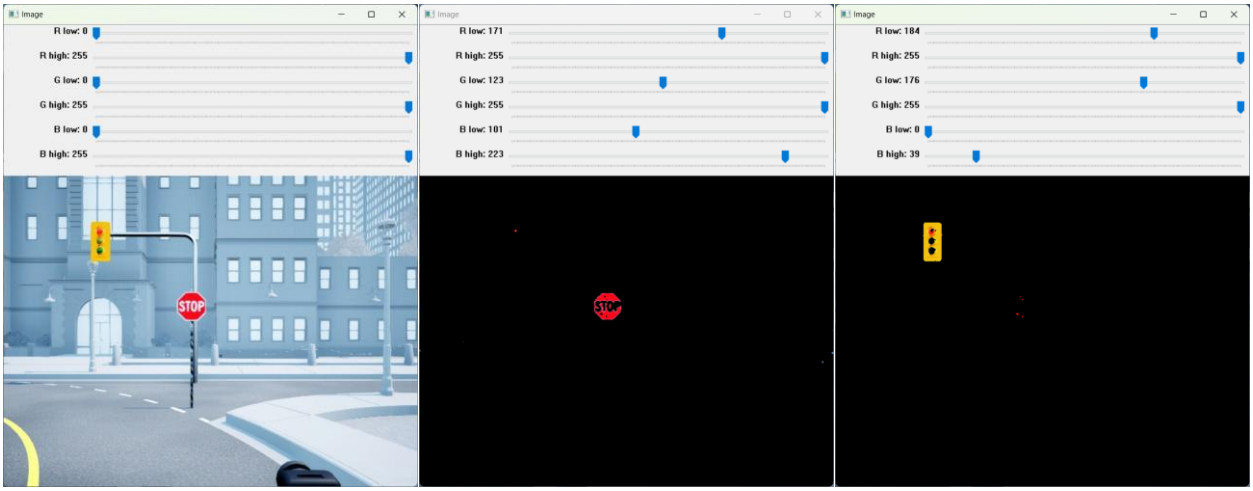

Figure 4. Virtual raw camera feed (left), segmenting red (middle), segmenting yellow (right).

Step 2– HSV Image Thresholding

The RGB color space combines color and brightness information, making it sensitive to lighting variations. To observe the impact of lighting variations, create a shadow or turn of the light after segmenting a target color for physical setup or change the weather for virtual setup. What do you notice? To address this, the HSV color space can be used, since color (Hue) and brightness (Value) is already decoupled, making it more robust under dynamics lighting conditions. Set ***mode*** = "hsv" in the "Experiment Configuration" region of **object_detection.py**. In **SECTION A**, add a line to convert the RGB image to HSV before using it for thresholding.

Results:

Adjust the HSV sliders in the "Image" window until the target object is segmented as shown in Figure 5 for physical setup or Figure 6 for virtual setup. Record the HSV bounds for traffic light (yellow) and stop sign (red). These values will be used in the next step.

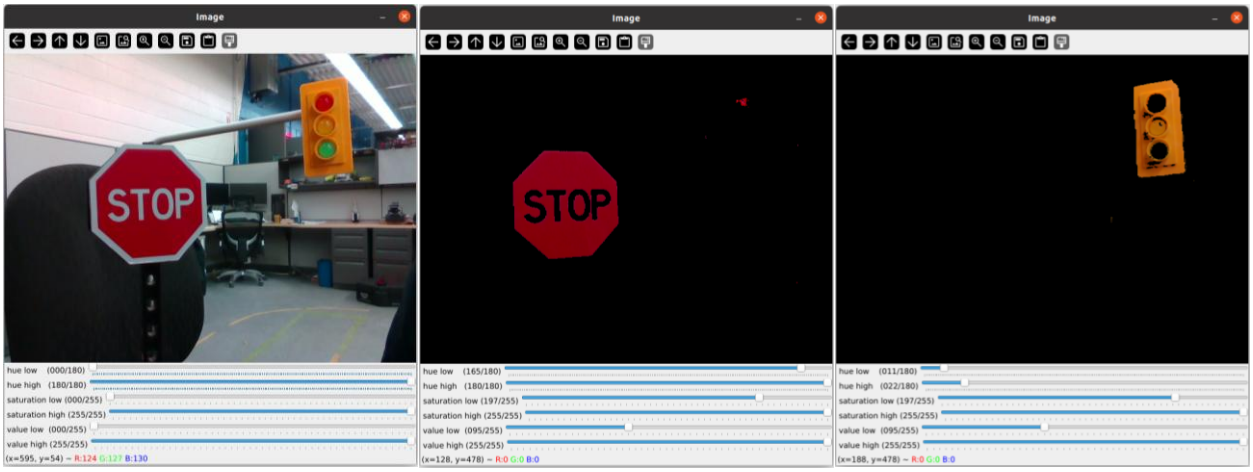

Figure 5. Raw camera feed (left), isolating red (middle), isolating yellow (right).

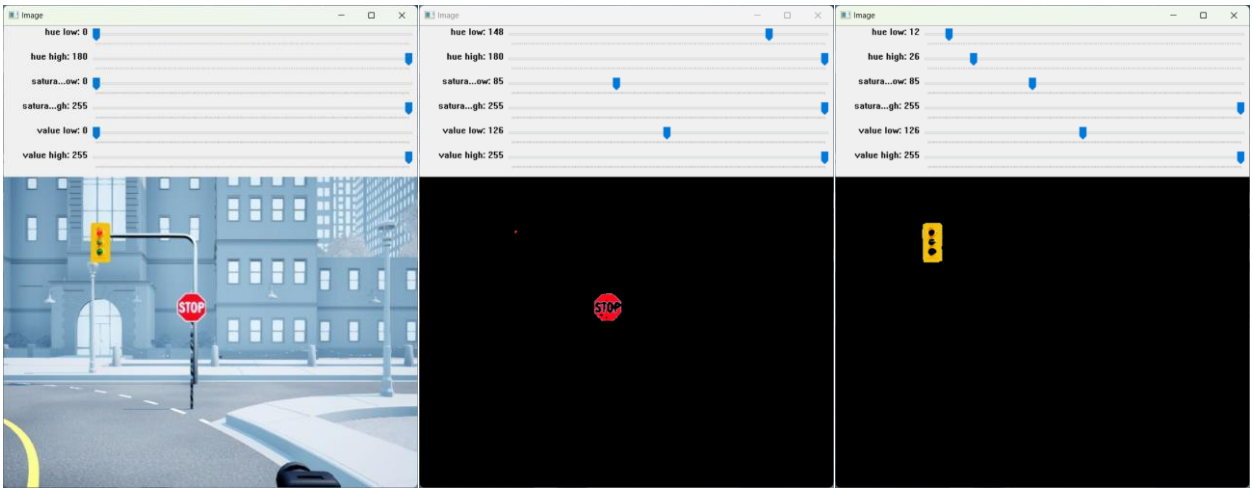

Figure 6. Virtual raw camera feed (left), isolating red (middle), isolating yellow (right).

Step 3– Color-based Object Detection

At this point you should have collected 2 sets of HSV values for traffic light and stop sign. Set *task* = "detect" in the "Experiment Configuration" region. In **SECTION B**, find the **obj_detect()** method and navigate to its definition. This method returns 3 lists containing information of detected objects. Complete **SECTION B.1** with recorded HSV bounds and ensure detection results are appended to appropriate lists.

```
        # ==================    SECTION B.1 - Object Detection    ================
            traffic_bounds=[(0, 0, 0), (255, 255, 255)]
            stop_sign_bounds=[(0, 0, 0), (255, 255, 255)]
            detectedMask.append(np.zeros(img.shape[:2],np.uint8))
            detectedName.append('')
            detectedBbox.append((0, 0, 0, 0))
```

```
        return detectedMask,detectedName,detectedBbox
    # =================          End of SECTION B.1          =================
```

Results:

A cv2 windows labeled “Image” will open. If the output is in the correct format, the camera feed will be annotated with detection results as shown in Figure 7 for physical setup or Figure 8 for virtual setup.

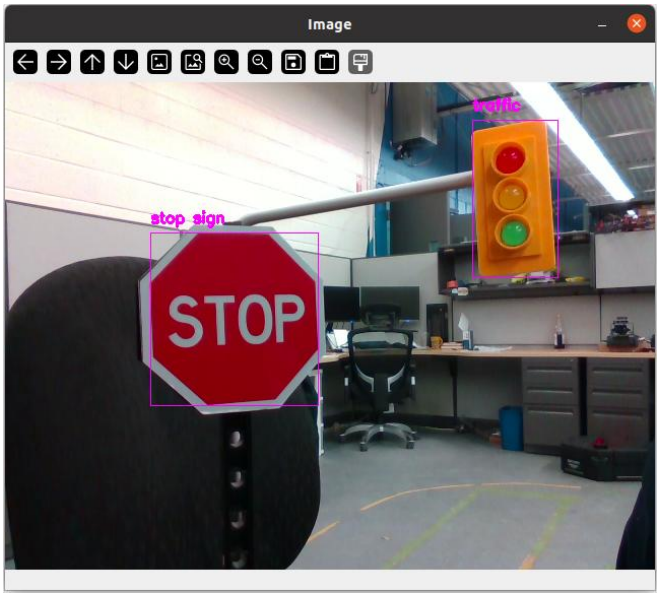

Figure 7. Detected stop sign and traffic light with annotation.

Figure 8. Detected stop sign and traffic light with annotation in virtual world.

Step 4– Distance Estimation

In contrast to the RGB or HSV values contained in the pixels of a colored image, each pixel in the depth image contains the distance value in meter. The object masks can now be combined with the depth image to estimate real-world distances to the objects, which is important for navigation tasks. Find the **find_distance()** method in **SECTION C**, navigate to its definition, and complete **SECTION C.1**.

```
        # ================   SECTION C.1 - Distance Estimation   ================
        detectedDist.append(0)
        return detectedDist
        # ================          End of SECTION C.1          ================
```

Results:

A cv2 windows labeled “Image” will open. If the output distances are in the corrected format, the distances to the objects will be added to the annotation as shown in Figure 9 for physical setup and Figure 10 for virtual setup.

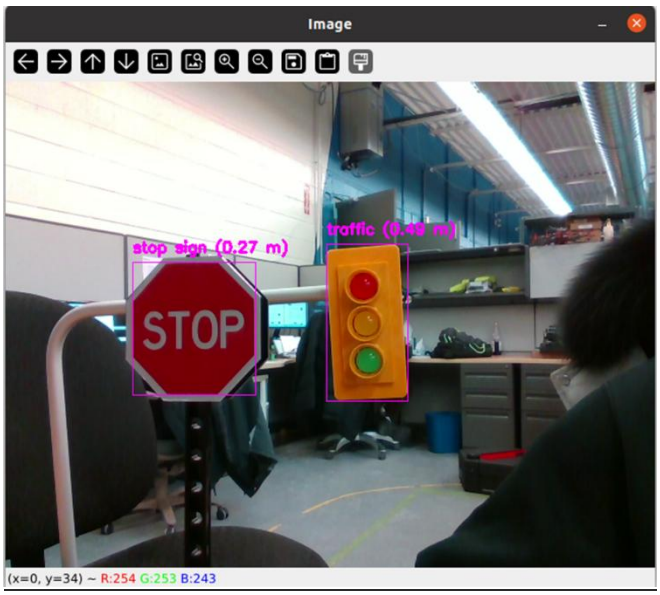

Figure 9. Stop sign and traffic light with estimated distances.

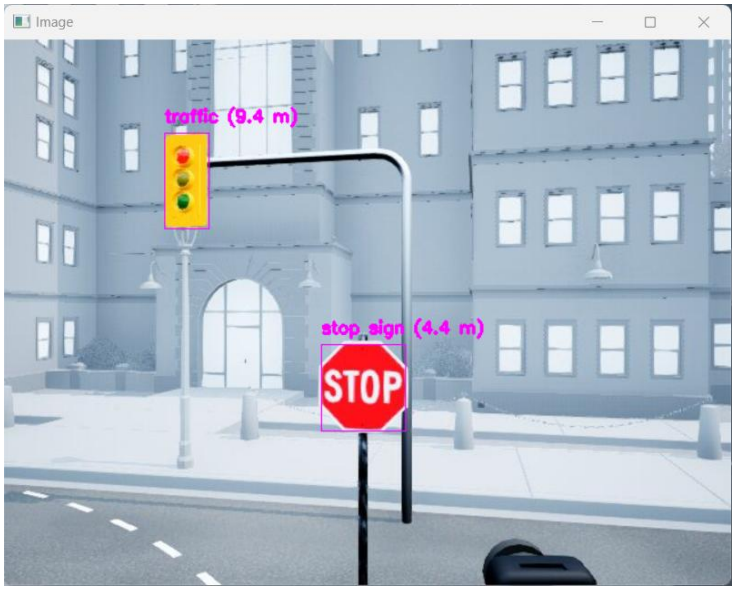

Figure 10. Stop sign and traffic light with estimated distances in the virtual world.

Skills Activity – Classification

The objective of this sections is to explore methods to classify objects with identical colors.

Step 1– Shape-based Classification

When a stop sign and a yield sign are both in the camera feed, using color alone is insufficient for object detection. Therefore, we might need to use something else to tell them apart. In this section, we will explore using geometric features to differentiate between the 2 objects. Set *task* = "classify" and *mode* = "shape" in the "Experiment Configuration" region. Navigate to the **obj_detect()** method and complete **SECTION B.2**. This method should largely be the same as SECTION B.1, with additional steps to distinguish between stop signs and yield signs based on the shapes of the red blobs.

```
        # ============  SECTION B.2 - Shape-based Classification   ===========
            yellow_bounds=[(0, 0, 0), (255, 255, 255)]
            red_bounds=[(0, 0, 0), (255, 255, 255)]
            detectedMask.append(np.zeros(img.shape[:2],np.uint8))
            detectedName.append('')
            detectedBbox.append((0, 0, 0, 0))
            return detectedMask,detectedName,detectedBbox
        # ============              End of SECTION B.2               ===========
```

Results:

A cv2 windows labeled "Image" will open and annotated camera feed will be displayed as shown in Figure 11 for physical setup and Figure 12 for virtual setup. You can use this annotated feed to validate your classification algorithm.

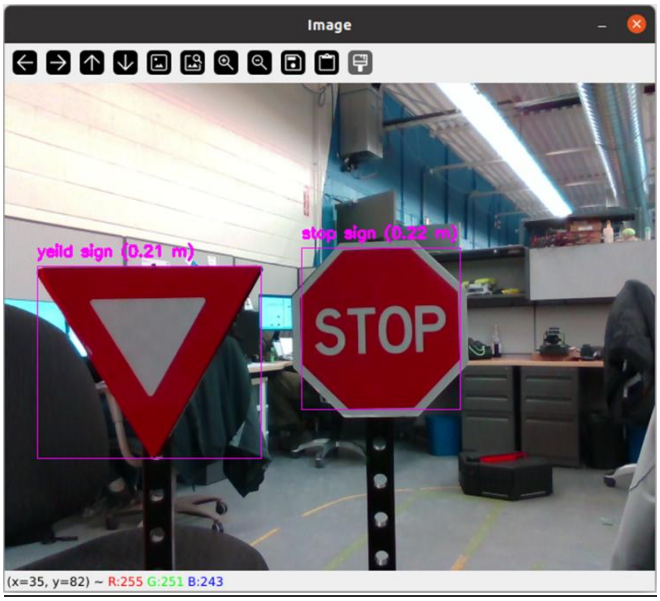

Figure 11. Detected stop sign and yield sign using color and shape.

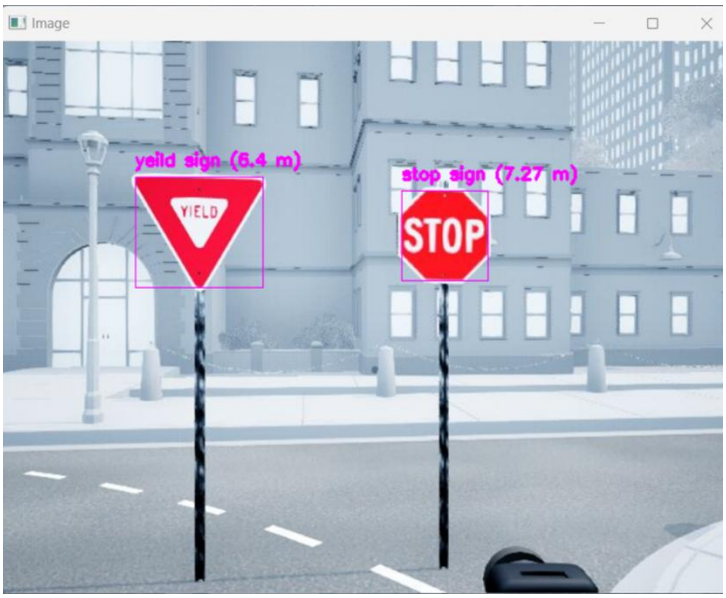

Figure 12. Detected stop sign and yield sign using color and shape in virtual world.

Step 2– Template Matching

The color and shape of an object are discriminative features. While feature-based classification often requires manually defining value ranges for each feature (e.g. color thresholds, geometric properties) to differentiate objects, this process becomes increasingly tedious as more features are added. Template matching offers an alternative approach that bypasses manual feature extraction and threshold tuning. Instead, this method relies on predefined templates—reference images of target objects. To generate the templates, comment out the **annotate()** method in the visualization region, run the code and take snippets directly in the "Image" window. Uncomment **annotate()** when done.

Set *mode* = "template" in the "Experiment Configuration" region. Navigate to **obj_detect()** and complete **SECTION B.3** to implement the **matchTemplate()** method from OpenCV for classification.

```
        # ============   SECTION B.3 - Template-based Detection    ============
            stop_path = 'path/to/your/stop.png'
            yield_path = 'path/to/your/yield.png'
            detectedMask.append(np.zeros(img.shape[:2],np.uint8))
            detectedName.append('')
            detectedBbox.append((0, 0, 0, 0))
            return detectedMask,detectedName,detectedBbox
        # ============              End of SECTION B.3              ============
```

Results:

A cv2 windows labeled "Image" will open and annotated camera feed will be displayed as shown in Figure 13 for physical setup and Figure 14 for virtual setup. How is the performance of template matching in comparison to shape-based classification? Move and rotate the signs, what do you observe? What can you say about the strengths and limitations of template matching?

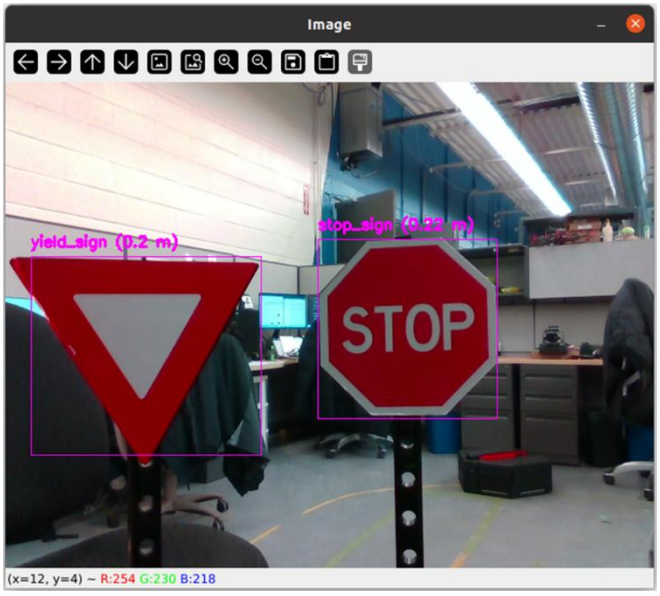

Figure 13. Detected stop sign and yield sign using template matching.

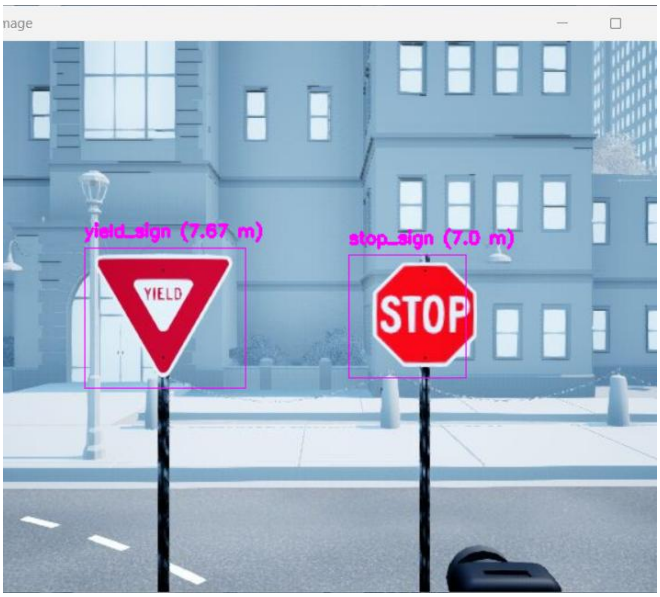

Figure 14. Detected stop sign and yield sign using template matching virtual world.

Step 3– YOLOv8 Segmentation Model

With the advancement in GPU technology, running complex machine learning algorithms in real-time becomes increasingly feasible. QCar2 leverages this progress with one of the most powerful edge-computing processors available, enabling state-of-the-art classification models to operate at over 20 Hz. In this section, we will use YOLO for classification and process its prediction results. Internet connection may be required if the QCar is running the YOLO library for the first time, as the pre-trained model will need to be downloaded. In addition, first time loading may take up to 20 minutes as the pre-training model is being optimized for the hardware. However, YOLO is currently **not supported** with the virtual setup.

Set ***mode*** = "yolo" in the "Experiment Configuration" region. Navigate to **obj_detect()** and complete **SECTION B.4** to process prediction results into formats that are compatible with the rest of the scripts.

```
        # ===============  SECTION B.4 - YOLO Segmentation Model  ===============
            imgProcessed = self.yolo.pre_process(img)
            results = self.yolo.predict(imgProcessed)
            detectedMask.append(np.zeros(img.shape[:2],np.uint8))
            detectedName.append('')
            detectedBbox.append((0, 0, 0, 0))
            return detectedMask,detectedName,detectedBbox
        # =================          End of SECTION B.4          =================
```

Results:

A cv2 windows labeled "Image" will open. If the output distances are in the corrected format, the distances to the objects will be added to the annotation as shown in Figure 15.

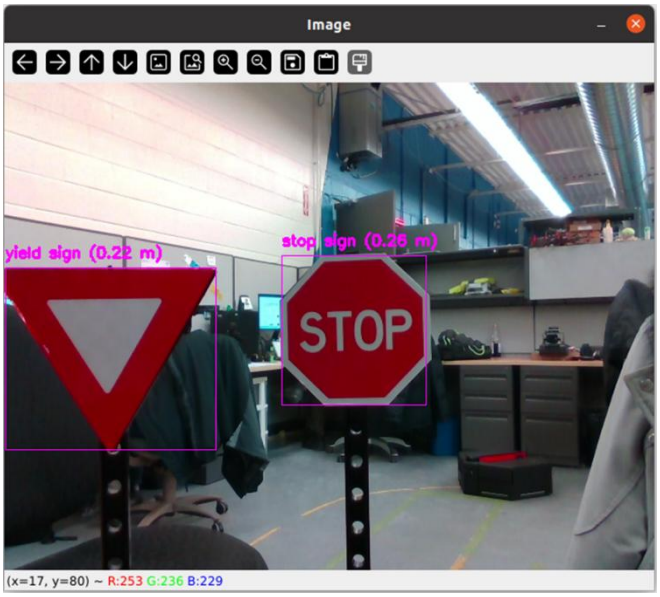

Figure 15. Detected stop sign and yield sign using YOLO.