Lab Guide
RGBD Imaging
Content Description

The following document describes a RGBD example in either python or MATLAB software environments utilizing the virtual QCar.

Prior to starting the example please go to the **Cityscape Lite** workspace and run the **qlabs_setup_applications.py** python script to configure the virtual world.

MATLAB

In this example, we will capture images from the Intel RealSense's RGB and Depth cameras. After thresholding the RGB image for a red stop sign, and extracting the sign's coordinates, the distance to the sign will be extracted from the corresponding coordinates in the depth image.

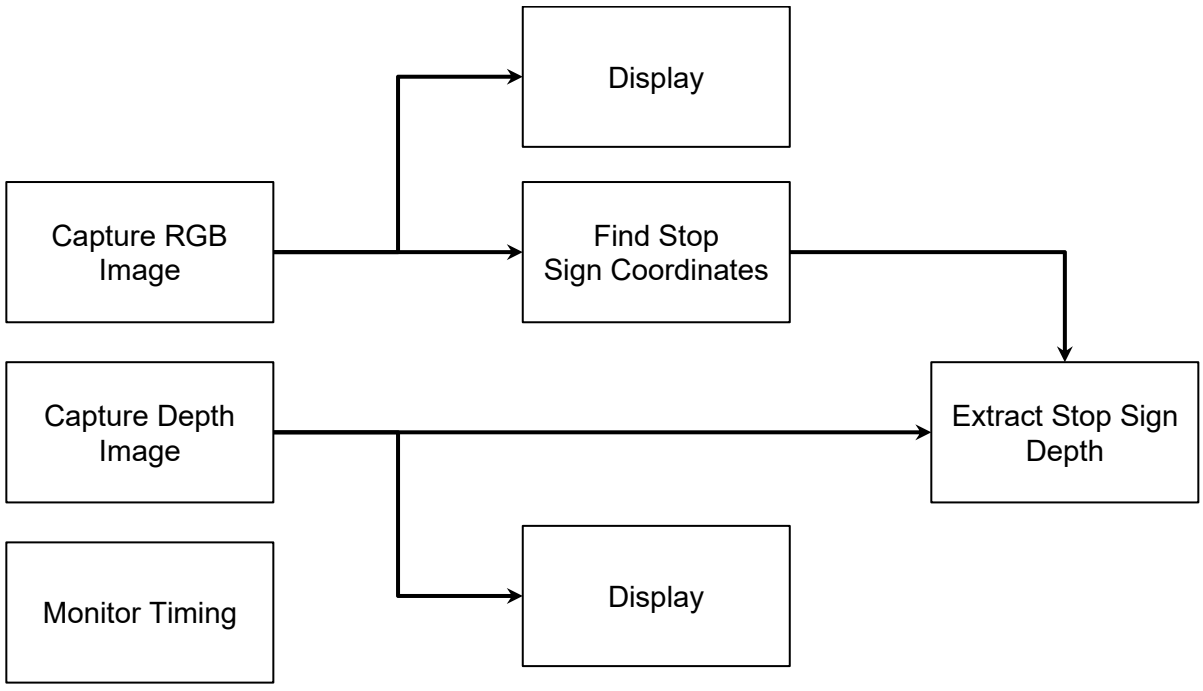


Figure 1. Component diagram

In addition, a timing module will be monitoring the entire application's performance. The Simulink implementation is displayed in Figure 2 below.

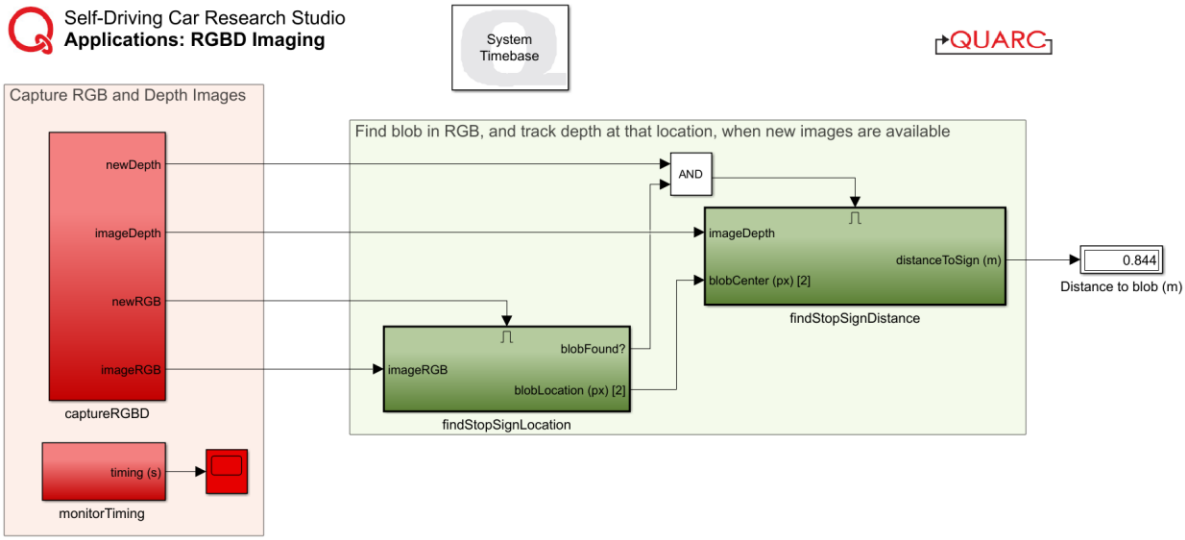
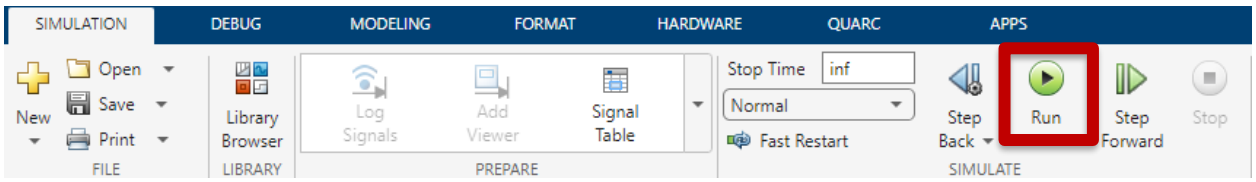


Figure 2. Simulink implementation of RGBD Imaging

Running the Example

To run examples for virtual QCar please go to the **SIMULATION** tab in the ribbon interface and click on the Run icon.



As lighting conditions and other objects in the scene may vary, you may need to adjust/tune the thresholding parameters inside the **findStopSignLocation** module. See the support documentation on **Image Color Spaces** and **Image Thresholding** for more information on this. Tune the saturation and value parameters until the binary image only displays the stop sign. The output in the 3 **Video Display** blocks should look those in Figure 3, which shows the raw RGB output, a binary output after thresholding, and the depth output.

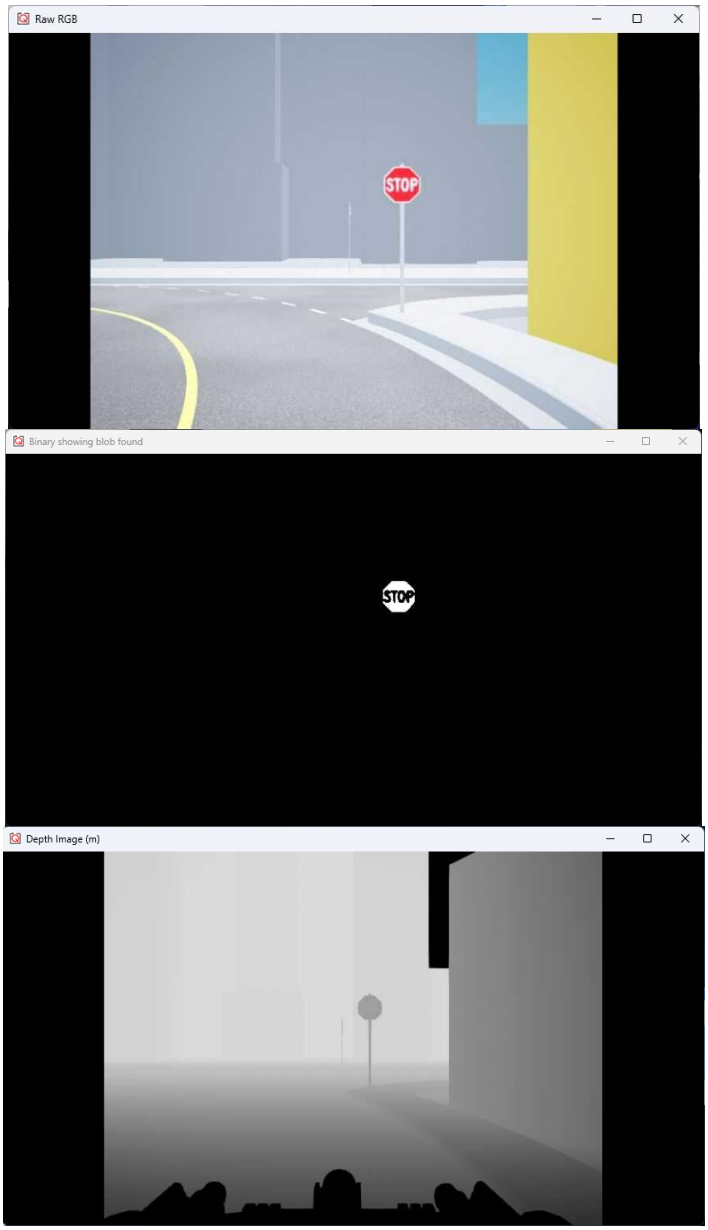
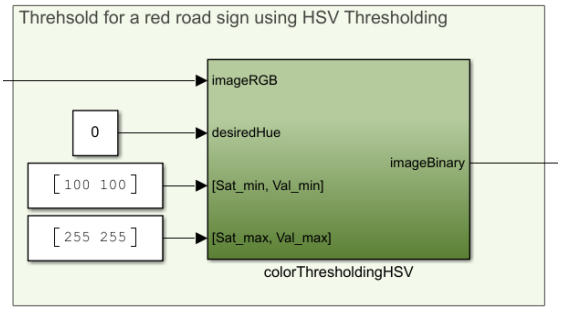


Figure 3. RGBD outputs showing RGB image (top), bitonal thresholding image (middle) and depth image (bottom)

Details

1. Capturing nothing but the Stop Sign

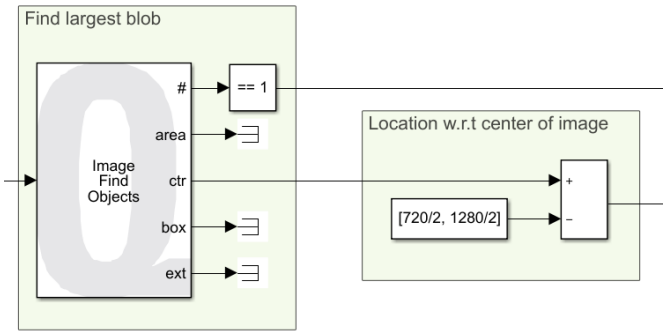


First, we pass the RGB image as **imageRGB** to the **colorThresholdingHSV** module inside the **findStopSignLocation** subsystem. This converts it to the HSV color space, decoupling the color itself from its intensity and lightness/darkness. subsystem using an **Image Transform** block.

The stop sign is red, which corresponds to a hue of **0**. Once we threshold the HSV image with suitable saturation and value parameters, the **imageBinary** output shows all the pixels that fall within our color search region.

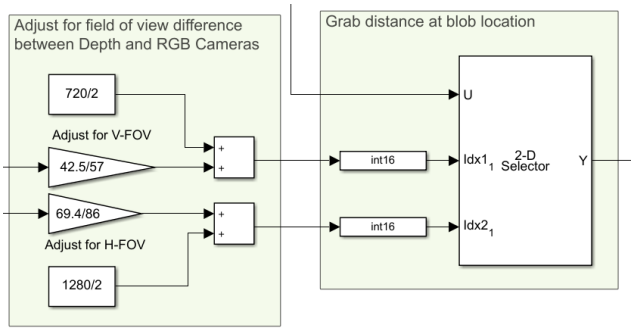
2. Finding the location of this blob in image coordinates

Within the **findStopSignLocation** subsystem, we use the **Image Find Objects** block to find the (row, col) image coordinates of the largest blob in the **imageBinary** input. We are only interested in the center of this blob, available at the **ctr** output. Note that this is w.r.t a coordinate frame attached at pixel location (1, 1). We need the location with respect to the center of the image, which is handled by a simple subtraction.



3. Estimating distance to the blob from the depth image

Although the RGB and Depth images are both captured at a 1280 x 720 resolution, the depth camera has a higher field of view, and hence the coordinates from step 2 must be adjusted before extracting the depth. We account for the field of view differences and then use a selector to extract the distance in the depth image at the adjusted coordinates.



4. Performance considerations

   To improve performance, we only execute the **findStopSignLocation** subsystem when **imageRGB** is new, through the means of an enabled subsystem. Therefore, the **blobFound?** output is high (**1**) if and only if a blob is found AND a new RGB image was available. Next, the **findStopSignDistance** subsystem is executed only if the **imageDepth** input is new AND **blobFound?** is true.

   Overall, the distance to the stop sign is only calculated if three conditions (new RGB image, new Depth image as well as a blob actually found) are met simultaneously, improving performance. Also note that the **Video Display** blocks are placed within the enabled subsystems.

Python

In this example, we will capture images from the Intel Realsense's RGB and Depth cameras. After thresholding the Depth image based on a minimum and maximum distance, a filtered binary mask is applied to RGB Image to only show the image within that range.

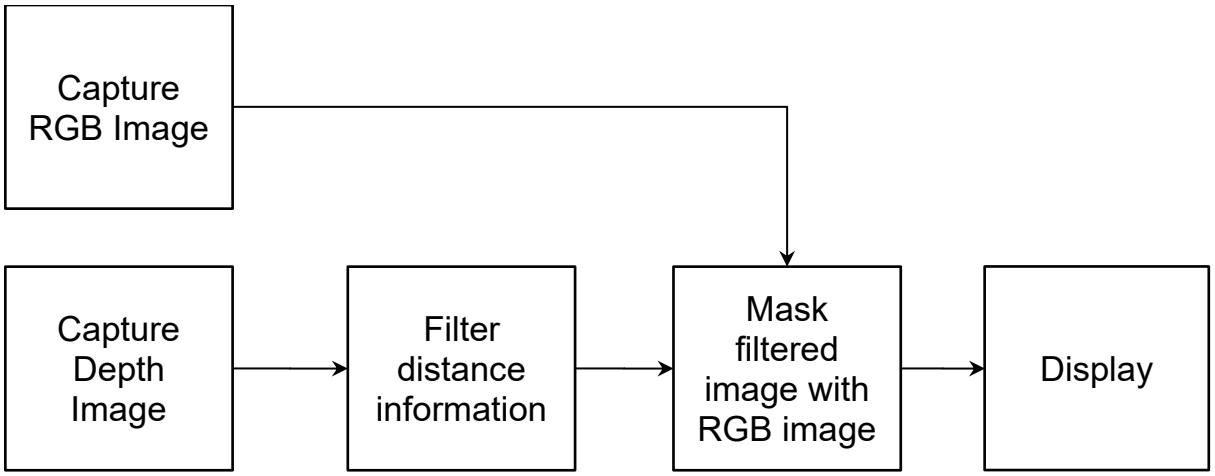


Figure 4. Component diagram

Running the Example

1. Check User Manual – Software Python for details on deploying python applications. Run the **task_lane_following.py** example on your local machine.

2. In this example, users only need to define the image frame size and the maximum & minimum distance for the depth image filtering. After executing the script on the QCar, the results should look like Figure. 5. (Another QCar is sitting in the front as an object.)
   Note: The minimum and maximum thresholds you apply should be based on real world distances assuming you had a physical QCar.

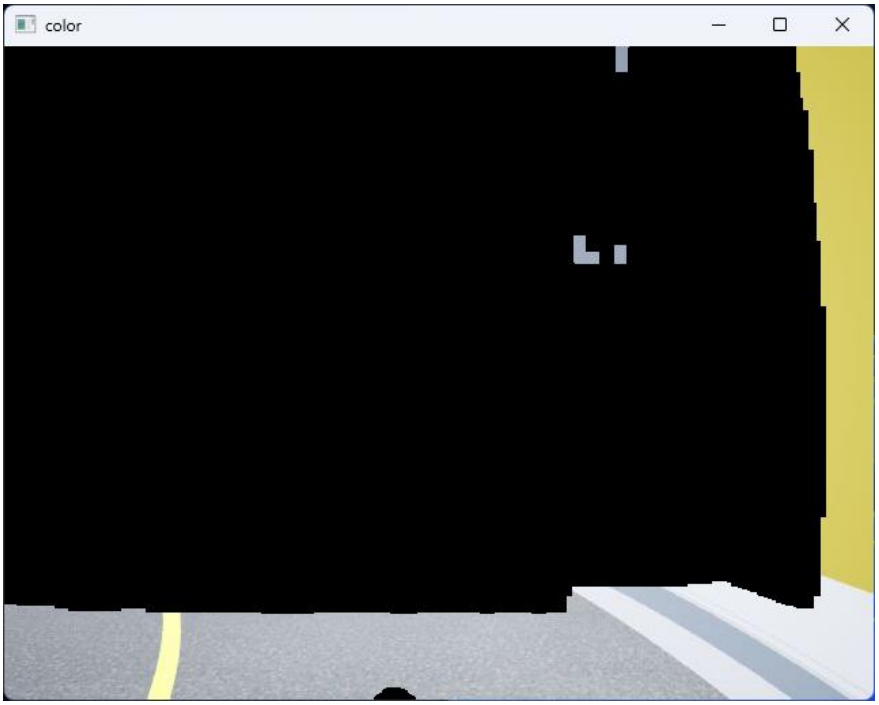


Figure 5. RGBD outputs showing the filtered image based on distance thresholds

Details

1. Depth image thresholding

   binary_thresholding() (that is provided with the **hal.utilities.image_processing** module) automatically detects 3-color or grayscale images and correspondingly thresholds them using the bounds provided. In this example, it thresholds based on the maximum and minimum distance that users define. The result is a binary image (numpy array) that is **1** within the threshold bounds and **0** elsewhere.

```
binaryNow = ImageProcessing.binary_thresholding(myCam1.imageBufferDepthPX,
                                                minPixel,maxPixel).astype(np.uint8)
```

2. Clean up the noise from the RGB-D sensor

   To clean up the noisy depth image, two filters are applied. This first consists of a **temporal difference** filter to remove single frame random noise. By calculating the difference between the current and previous frames, and removing those pixels in the current frame, we remove noise between frames. For a new pixel to get registered, it must last at least 2 frames. This introduces a maximum delay of **sample_time** in the system, which is set to 1/30 s. Next, a **spatial closing filter** (morphological dilation and erosion sequence) is applied to fill holes and clean up edges.

```
binaryClean = ImageProcessing.image_filtering_close(cv2.bitwise_and(
cv2.bitwise_not(np.abs(binaryNow - binaryBefore)/255), binaryNow/255 ),
dilate=3, erode=1, total=1)
```

3. Image correction between the depth camera and RGB camera and mask

   The field of view is different on both cameras and there is also a physical distance between them. The first line handles this transformation. This adjusted mask is then applied to the RGB image.

```
binaryClean = cv2.resize(binaryClean[20:420, 60:600],
                                 (640,480)).astype(np.uint8)*255
#Apply the binaryClean mask to the RGB image captured, and then display
it.
maskedRGB = cv2.bitwise_and(src1=myCam1.imageBufferRGB,
                            src2=myCam1.imageBufferRGB,
                           mask=binaryClean)
```