Integration of the camera depth image

3EY4 - Lab 10 - Bonus

AV#23

Danial Noori Zadeh

400367734

Hassan Al Masmoum

400330500

Kevin Le

400385350

April 9, 2024

Table of Contents

Problem Identification

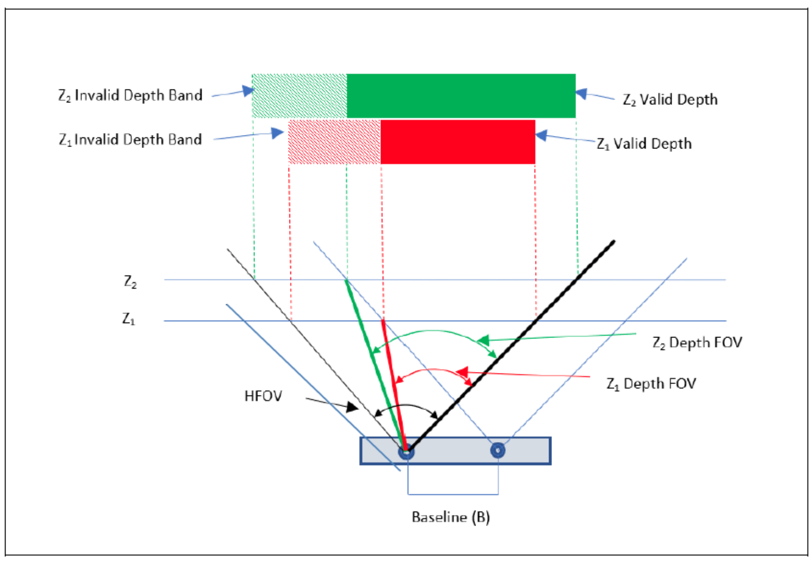
We were tasked with integrating the depth cloud data published by the camera with the LiDAR to detect obstacles that are not on the plane of the LiDAR. Implied in this problem is the requirement of efficiency and rate matching between the two sources of information (LiDAR and depth camera). A mismatch of sampling time or the resolution of can cause mis-operation of the self-driving algorithm.

Solution Overview

The camera publishes two topics that are key to our solution. One is the camera info that publishes information needed to convert depth (d) and pixel location (u, v) to xyz coordinates and later to r, θ polar coordinate to refine the LiDAR published ranges. The second important message topic is /camera/depth/image_rect_raw that contains the actual depth readings. We use CVBridge library to convert the image raw data to a matrix, but we were unable to pip install pyrealsense2 python wrapper to extract the xyz data, so, we manually computed them from the camera intrinsic data.

The solution uses a range of interest (ROI) over both u and v axis of the image pixels to reduce the unnecessary calculations. Another method improve efficiency is to filter the extracted xyz points based on a x and y window to avoid reading ground points and detection of objects that are out of the trajectory of the car. The results are confirmed by publishing the refined scan ranges and the performance was significant improved by optimizing the ROI (from 6 Hz publish rate to 12 Hz).

Figure 4-1. Depth Field of View to Depth Map illustration



Note:

1. As the scene's distance from the depth module increases, the invalid depth band decreases in the overall depth image. Overall depth image is invalid depth band plus valid depth map.

Design steps

Step 1 – launching the camera node & listening to the camera messages.

We used the rs_camera.launch file provided by the camera to realsense2 package, enabled the IR for extra precision and reduced the publish rate to 15 fps. However, it is recommended by the realsense documentation that both the depth and RGB camera operate at the same frequency and as high as possible, and if needed apply “decimation” filter to bring down the rate. We have tThe launch file is called in the self-driving package under experiment.launch as shown below.

```
av23@av23-desktop:~$ roslaunch realsense2_camera
demo_pointcloud.launch        rs_camera.launch                  rs_multiple_devices.launc
demo_t265.launch              rs_d400_and_t265.launch           rs_rgbd.launch
opensource_tracking.launch    rs_d435_camera_with_model.launch  rs_rtabmap.launch
rs_aligned_depth.launch       rs_from_file.launch               rs_t265.launch
av23@av23-desktop:~$ roslaunch realsense2_camera
```

```
<!-- Launch the camera node -->
<include file="$(find realsense2_camera)/launch/rs_camera.launch">
  <arg name="depth_width"        default="848"/>
  <arg name="depth_height"       default="480"/>
  <arg name="enable_depth"       default="true"/>
  <arg name="depth_fps"          value = "15"/>

</include>
```

The important camera depth data are published the highlighted topics, which we set up in the initialization of the navigation.py file as shown below. We need to import **Image** and **CameraInfo** datatypes from sensors.msg to launch these publishers and subscribers. The “refined_lidar_ranges” is the new topic published to debug the corrected ranges from the program.

```
        rospy.Subscriber(depth_image_topic, Image, self.imageDepthCallback, queue_size=1)
        rospy.Subscriber(depth_info_topic, CameraInfo, self.imageDepthInfoCallback, queue_size=1)
        # Publisher for refined data
        self.refined_lidar_pub = rospy.Publisher("refined_lidar_ranges", LaserScan, queue_size=5)

```

```
av23@av23-desktop:~$ rostopic list
/camera/color/camera_info
/camera/color/image_raw
/camera/color/metadata
/camera/depth/camera_info
/camera/depth/image_rect_raw
/camera/depth/metadata
```

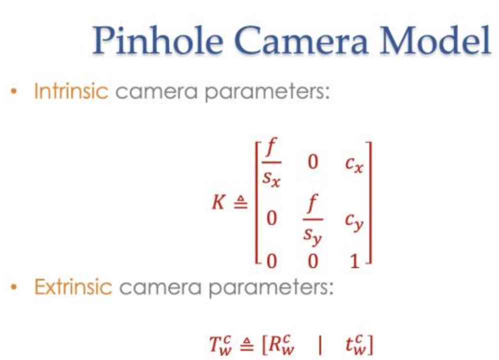
Step 2 – collecting camera information data

Looking into a sample message from the camera_info topic we can see the following fields which contain many useful information such intrinsic matrix (K) of the camera.

```
av23@av23-desktop:~$ rostopic echo -n 1  /camera/depth/camera_info
header:
  seq: 6381
  stamp:
    secs: 1711601916
    nsecs:  53215504
  frame_id: "camera_depth_optical_frame"
height: 480
width: 848
distortion_model: "plumb_bob"
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [421.70062255859375, 0.0, 425.8759765625, 0.0, 421.70062255859375, 239.4052734375, 0.0, 0.0, 1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [421.70062255859375, 0.0, 425.8759765625, 0.0, 0.0, 421.70062255859375, 239.4052734375, 0.0, 0.0, 0.0, 1.0, 0.0]
binning_x: 0
binning_y: 0
roi:
  x_offset: 0
  y_offset: 0
  height: 0
  width: 0
  do_rectify: False
---
```

we then extracted the important information from the K field and stored them globally for the deprojection in the imageDepthCallback function.

```python
    def imageDepthInfoCallback(self, cameraInfo):
        try:
            self.intrinsics = True
            self.img_width = cameraInfo.width
            self.img_height = cameraInfo.height
            self.ppx = cameraInfo.K[2]
            self.ppy = cameraInfo.K[5]
            self.fx = cameraInfo.K[0]
            self.fy = cameraInfo.K[4]
            #print("CALIBERATION: found camera Info\n")
            #print(self.fx,', ', self.fy,', ', self.ppx, ', ', self.ppy, '\n')

        except CvBridgeError as e:
            print(e)
            return

```



Step 3 – receive and de-projecting camera depth data.

The xyz coordinates of the points in the camera depth image can be computed from the depth and pixel location (u and v). We used a reduced ROI to improve the efficiency of the deprojection process. We filtered out the ground points by filter the points based on their y-value. These settings can be changed in the params.yaml file.

$$z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$u = \frac{f_x}{z}x + c_x \Rightarrow x = \frac{z}{f_x}[u - c_x]$$

$$v = \frac{f_y}{z}y + c_y \Rightarrow y = \frac{z}{f_y}[v - c_y]$$

$$z = z \qquad z = z$$

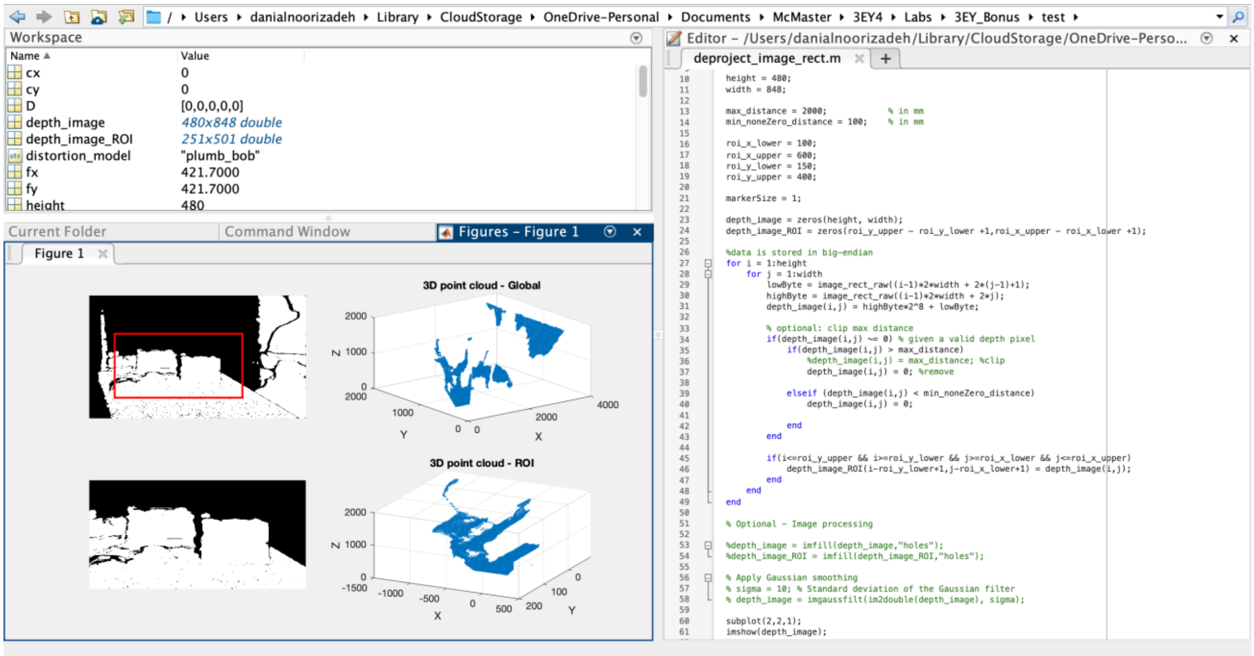
```yaml
# Camera Parameters
use_camera: 1     # use the camera depth data (1) else 0 -> do not
min_depth: 100    # mm
max_depth: 5000   # mm
cam_baselink_offset: 270 #mm

# the following are defined in number of pixels
image_width:  848
image_height: 480

roi_u_lower: 250
roi_u_upper: 548
roi_v_lower: 150
roi_v_upper: 470

y_cam_max: 100  #mm
y_cam_min: -150 #mm
x_cam_max: 600  #mm
x_cam_min: -600 #mm
```

```python
# start of camera callback functions
    def imageDepthCallback(self, data):

        try:
            cv_image = self.bridge.imgmsg_to_cv2(data, data.encoding)
            # height, width = cv_image.shape
            if self.intrinsics:
                # Convert all pixels to points within the region of interest (ROI)
                v_range = np.arange(self.roi_v_lower, self.roi_v_upper + 1)
                u_range = np.arange(self.roi_u_lower, self.roi_u_upper + 1)
                u_grid, v_grid = np.meshgrid(u_range, v_range)

                depths = cv_image[v_grid, u_grid]
                depths[depths > self.max_depth] = 0
                depths[depths < self.min_depth] = 0
                depths = depths.astype(float)

                # deproject_depth_to_point
                x = (depths / self.fx) * (u_grid - self.ppx)
                y = (depths / self.fy) * (v_grid - self.ppy)
                z = depths

                # filter the points out of the range
                mask = (y < self.y_cam_max) & (y > self.y_cam_min) & (z > 0) & (z < self.max_depth) & (x<self.x_cam_max) & (x>self.x_cam_min)
                x = x[mask]
                #y = y[mask] # we don't need the y component anymore
                z = z[mask]
                z = z + self.cam_baselink_offset

                x = x.astype(float)
                #y = y.astype(float)
                z = z.astype(float)

                # Calculate the polar coordinate in the camera frame from the point in the xz plane
                self.cam_ranges = np.sqrt((x)**2 + (z)**2)  # Euclidean distance
                self.cam_angles = np.arctan2(z, x)          # Angle in radians

                #OPTIONAL: Store xyz coordinate
                #self.depth_points = np.stack([x,y,z], axis=-1)

        except CvBridgeError as e:…
        except Exception as e:…
```

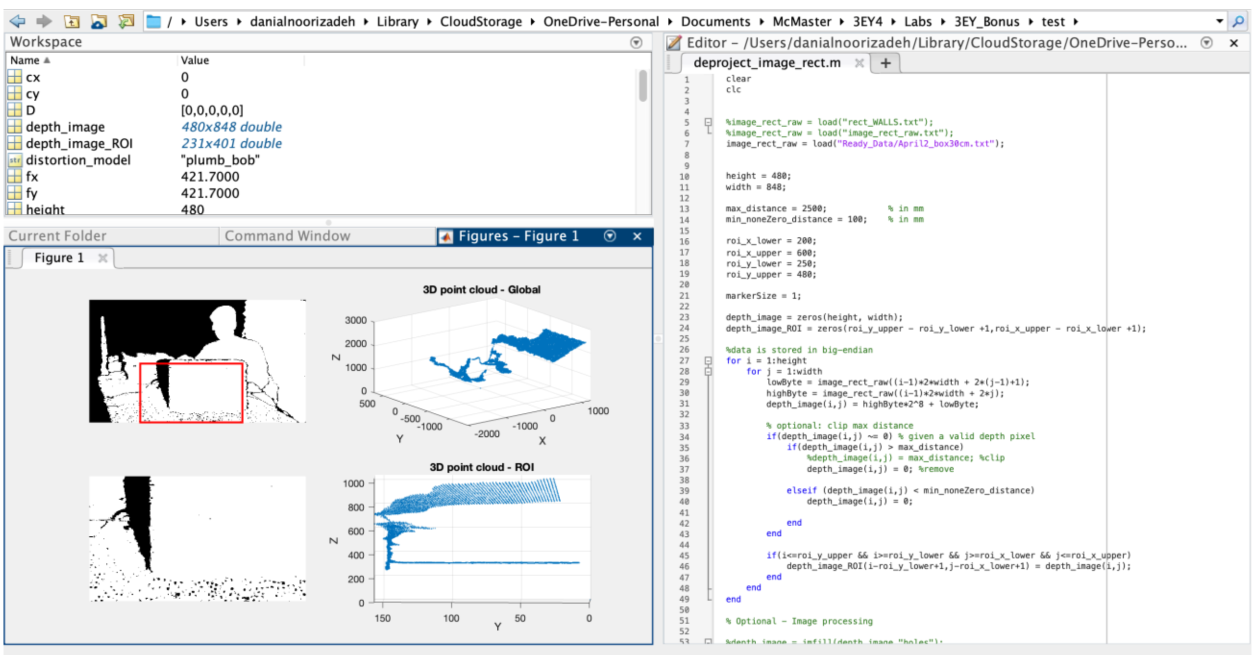
The z value is compensated by self.cam_baselink_offset to translate the points into the LiDAR frame and then the points are expresses in polar coordinates. an important step before conversion to polar using numpy square root and inverse tangent is that the arrays have to be set as float type otherwise the function will exhibit erroneous behavior.

Verification of deprojection algorithm in MATLAB

To verify the correctness of the deprojection function we made a MATLAB script that converts the image message to an image matrix, iterates over the depth points within the ROI and then computes the xyz coordinates. We used experimental data collected by placing a block that is out of the sight of the LiDAR at different distances away from the AEV. We monitored the output point cloud and tuned the ROI and y limits.



The red square displays the ROI setting. The advantage of performing the test in MATLAB is that we can use image processing tools such as hole filling filters to improve the readings.



Step 4 – Integration into LiDAR callback function

The LiDAR data needs to be refined before any pre-processing or navigation algorithm is run on it. We would only integrate the camera info if the global use_camera variable is set to true, the camera intrinsic data is collected and the cam_ranges have been extracted. We then pass the LiDAR ranges, camera ranges and corresponding angles to the refine_lidar method and create a new LaserScan message for the refined ranges.

```
    def lidar_callback(self, data):

        ranges = data.ranges

        t = rospy.Time.from_sec(time.time())
        self.current_time = t.to_sec()
        dt = self.current_time - self.prev_time

        self.prev_time = self.current_time
        sec_len= int(self.heading_beam_angle/data.angle_increment)

        if (self.use_camera and self.intrinsics and (self.cam_ranges is not None) and (self.cam_angles is not None)):
            # refine the ranges
            ranges = self.refine_lidar(ranges, self.cam_ranges, self.cam_angles)

            # Create a new LaserScan message for the refined ranges
            refined_ranges_msg = LaserScan()
            refined_ranges_msg.header.stamp = rospy.Time.now()
            refined_ranges_msg.header.frame_id = data.header.frame_id  # Assuming the frame ID is the same as the original LiDAR data
            refined_ranges_msg.angle_min = data.angle_min
            refined_ranges_msg.angle_max = data.angle_max
            refined_ranges_msg.angle_increment = data.angle_increment
            refined_ranges_msg.time_increment = data.time_increment
            refined_ranges_msg.scan_time = data.scan_time
            refined_ranges_msg.range_min = data.range_min
            refined_ranges_msg.range_max = data.range_max
            refined_ranges_msg.ranges = ranges

            # Publish the refined ranges
            self.refined_lidar_pub.publish(refined_ranges_msg)


        proc_ranges, mod_ranges = self.preprocess_lidar(ranges)


        if self.drive_state == "normal":⋯

        elif self.drive_state == "backup":⋯

        else:⋯

        # Publish to driver topic
        drive_msg = AckermannDriveStamped()
```

We first cast the ranges from LiDAR to a list, so it becomes mutable. We then find the closest lidar index that the cam_angles correspond to by dividing the angle by the angle increment of the LiDAR. It's important we add pi/2 to the angles to compensate for the LiDAR x axis being 90 degrees rotated with respect to the camera frame. We have already compensated for the translation between the two frames by adding the offset to the z-values.

The program will then iterate over the ranges in the camera, access the corresponding angle in the LiDAR frame and only overwrite the ranges if the camera suggests a smaller value. The corrected_point is a debugging variable used to store the number of corrections made to the LiDAR ranges.

```python
    # Refine LiDAR Data
    def refine_lidar(self, ranges_LiDAR, cam_ranges, cam_angles):

        # Convert ranges_LiDAR tuple to a list
        refined_ranges = list(ranges_LiDAR)

        # Convert camera angles to LiDAR indices
        lidar_indices = np.round((cam_angles + np.pi/2.0)/ self.ls_ang_inc).astype(int)

        corrected_points = 0
        # Iterate through each LiDAR index and compare the ranges
        for i in range(len(lidar_indices)):
            # Check if the camera range is smaller than the LiDAR range at the corresponding index
            if float(cam_ranges[i]/1000.0) < refined_ranges[lidar_indices[i]]:
                # Update the LiDAR range with the camera range
                refined_ranges[lidar_indices[i]] = float(cam_ranges[i]/1000.0)
                corrected_points = corrected_points + 1

        #print("corrected ", corrected_points, " times\n")
        # OPTIONAL: Convert the refined_ranges list back to a tuple if needed
        #refined_ranges = tuple(refined_ranges)

        return refined_ranges
```

Testing & Verification

Static test

The published refined LiDAR was compared to the original LiDAR in rviz. We further verify the corrected points are coming from the camera's depth cloud by enabling the depth cloud topic of the camera in rviz.

AEV in action

The vehicle is able to avoid stationary objects assuming they are already in the camera's line of sight; however, due to the difference in timing between LiDAR and the refined camera info the vehicle cannot avoid the objects that it did not have enough time to recognize.