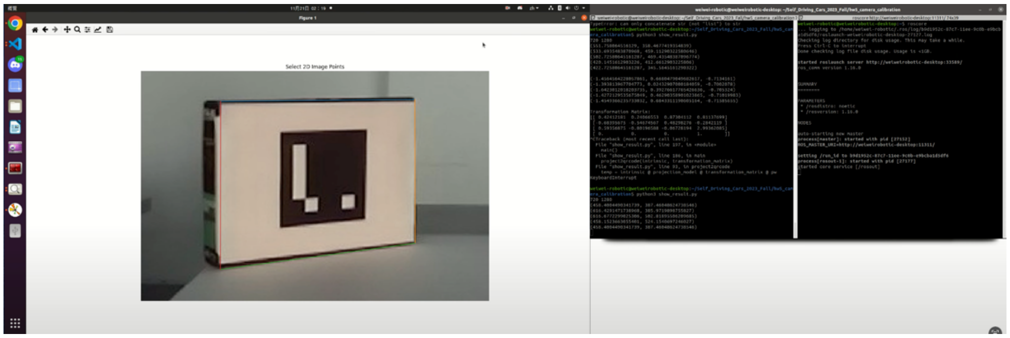
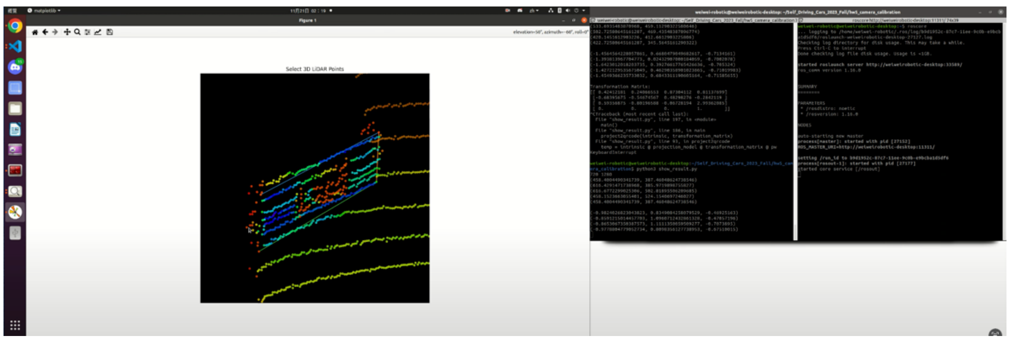
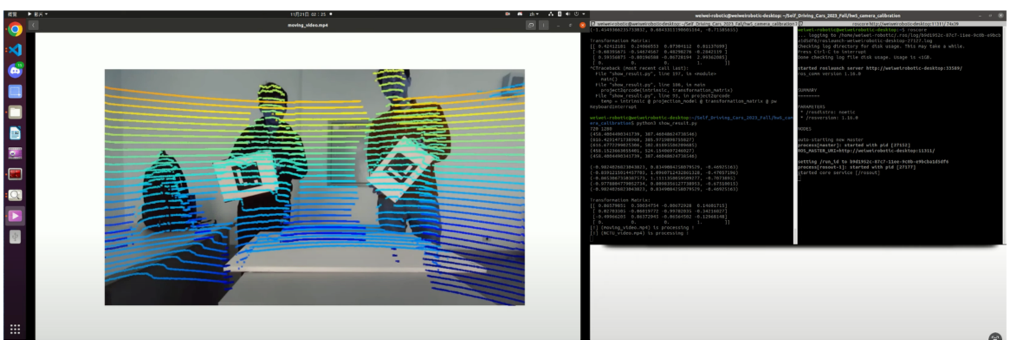
Self Driving Cars 2023-Fall
Homework 5
312605001 機器人碩士 歐庭維

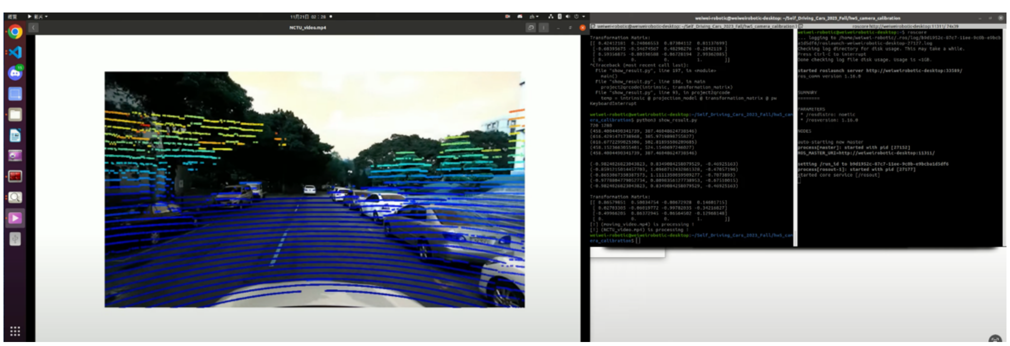
1. screenshot (or video) of running code along with your personal information

- screenshot









- video link : https://youtu.be/TTbifdly0Fw

2. Briefly explain your code

- Part 1 : subscribe_rosbag.py

After trying various approaches, I noticed a problem: the timestamps of the lidar and camera don't match up. The "lidar_callback" updates at a slower rate than the "camera_callback". So, I threw the "camera_callback" into the "lidar_callback" to fix it. I only save the file when both callbacks happen at the same time.

```
class MySubscriber():
    def __init__(self):
        rospy.Subscriber("/points", PointCloud2, self.lidar_callback, queue_size=None)
        rospy.Subscriber("/left/image/compressed", CompressedImage, self.camera_callback,
queue_size=None)

    def lidar_callback(self, msg):
        global file_name
        # Check the header
        # print(msg.header, '\n')

        timestamp = str(msg.header.stamp.secs) + "{:09d}".format(msg.header.stamp.nsecs)
        pointcloud = read_point_from_msg(msg)
        np.save(output_root_lidar + str(file_name) + '.npy', pointcloud)

        # Call camera callback after processing lidar data
        self.camera_callback(msg)

    def camera_callback(self, msg):
        global file_name
        # Check the header
        # print(msg.header, '\n')

        timestamp = str(msg.header.stamp.secs) + "{:09d}".format(msg.header.stamp.nsecs)
        img = CvBridge().compressed_imgmsg_to_cv2(msg)

        # Check if the image is not empty and has a valid size
        if img is not None and img.size > 0:
            # Check the type of the image
            print("Image type:", img.dtype)

            cv2.imwrite(output_root_camera + str(file_name) + '.jpg', img, [cv2.IMWRITE_JPEG_QUALITY,
90])
            file_name += 1
        else:
            print("Warning: Empty or invalid image received.")
```

- Part 2 show_result.py

  - 2-1 find the transformation matrix

    - Set the camera intrinsic parameters

```
    focal_length = 698.939
    intrinsic = np.array([[focal_length, 0, width/2],
                          [0, focal_length, height/2],
                          [0, 0, 1]])
```

    - Set the distortion coefficients

```
dist = np.zeros(5)
```

    - Get 2D and 3D coordinates by clicking points on the image and lidar data

```
    uv_coordinates = click.click_points_2D(image)
    world_coordinates = click.click_points_3D(lidar)
```

- Solve PnP using RANSAC algorithm

* Intrinsic

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z}\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \Rightarrow Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} P_w \Rightarrow Z P_{uv} = K P_w$$

* Extrinsics

$$Z P_{u,v} = K \begin{pmatrix} n & o & a & p \end{pmatrix} P_w$$

$$\hookrightarrow Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \begin{pmatrix} n & o & a & p \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

to find the extrinsic ⇒ solve PnP problem

the known parameter

```
rvec, tvec, _ = cv2.solvePnPRansac(world_coordinates, uv_coordinates, intrinsic, dist)
```

- Refine the solution using Levenberg-Marquardt optimization

```
rvec, tvec = cv2.solvePnPRefineLM(world_coordinates, uv_coordinates, intrinsic, dist, rvec, tvec)
```

- Convert rotation vector to rotation matrix & create the transformation matrix

```
    R, _ = cv2.Rodrigues(rvec)

    transformation_matrix = np.column_stack((R, tvec))
    transformation_matrix = np.vstack((transformation_matrix, [0, 0, 0, 1]))
```

- 2-2 Visualization of the 3D lidar points projected onto the image

  In this step, I Visualize the 3D lidar points projected onto the image to check the transformation result.

  - Since the intrinsic is a 3x3 matrix, to align with matrix operations, I designed a "Projection model matrix" to convert it into a 3x4 matrix.

```
    projection_model = np.array([[1, 0, 0, 0],
                                 [0, 1, 0, 0],
                                 [0, 0, 1, 0]])
```

  - Project lidar points onto the image

    In this step, the variable ‘temp’ does not represent the actual UV coordinates after the transformation.

* Intrinsic

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z}\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \Rightarrow Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} P_w \Rightarrow Z P_{uv} = K P_w$$

* Extrinsics

$$Z P_{u,v} = K \begin{pmatrix} n & o & a & p \end{pmatrix} P_w$$

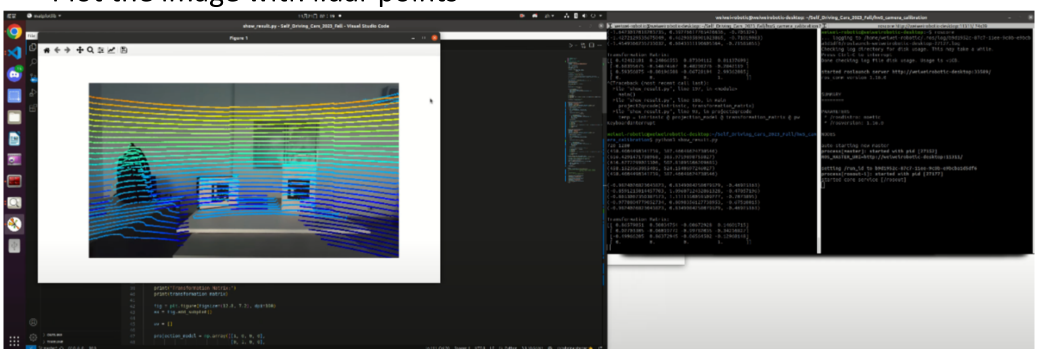
$$\hookrightarrow \underbrace{Z}_{\text{scaling factor}}\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = K \begin{pmatrix} n & o & a & p \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

$$K\begin{pmatrix} n & o & a & p \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

let $c = 1 \Rightarrow Z = \frac{1}{c} \Rightarrow \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{c}\begin{pmatrix} a \\ b \\ c \end{pmatrix}$

```
    for j in range(len(lidar)):
        pw = np.concatenate([lidar[j, :3], [1]])
        temp = ((intrinsic @ projection_model) @ transformation_matrix) @ pw.T
        scales = 1/temp[2]
        temp = temp * scales
        uv.append(temp)
```

- Plot the image with lidar points



- 2-3 Result of overlaying lidar on the camera in ‘moving.bag’ & ‘NCTU.bag’.
  - Process each frame step by step and save the overlaid images to a folder.

```
    for frame in range(loop_len):
        img = cv2.imread(camera_folder + camera_files[frame])
        pointcloud = np.load(lidar_folder + lidar_files[frame])

        uv = []

        projection_model = np.array([[1, 0, 0, 0],
                                     [0, 1, 0, 0],
                                     [0, 0, 1, 0]])

        for j in range(len(pointcloud)):
            pw = np.concatenate([pointcloud[j, :3], [1]])
            temp = intrinsic @ projection_model @ transformation_matrix @ pw
            scales = 1 / temp[2]
            temp = temp * scales
            uv.append(temp)

        uv = np.array(uv).T

        result_file = os.path.join(output_folder, f'{frame}.png')

        plt.figure(figsize=(12.8, 7.2), dpi=100)
        plt.imshow(img)
        plt.xlim(0, 1280)
        plt.ylim(720, 0)
        plt.scatter(uv[0, :], uv[1, :], c=pointcloud[:, 2], marker=',', s=10,
edgecolors='none', alpha=0.7, cmap='jet')
        plt.axis('off')

        plt.savefig(result_file)
        plt.close()
```

- Convert the results in the folder into a .mp4 file.

```
def images_to_video(input_folder, output_video_path, fps=15):
    image_files = sorted(os.listdir(input_folder), key=lambda x:
int(x.split('.')[0]))

    # Assuming all images have the same resolution as the first image
    first_image = cv2.imread(os.path.join(input_folder, image_files[0]))
    height, width, layers = first_image.shape

    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    video = cv2.VideoWriter(output_video_path, fourcc, fps, (width, height))

    for image_file in image_files:
        image_path = os.path.join(input_folder, image_file)
        img = cv2.imread(image_path)
        video.write(img)

    video.release()
    cv2.destroyAllWindows()
```

3. Discussion

   a. Explain why we need to use sdc-calibration.bag for calibration

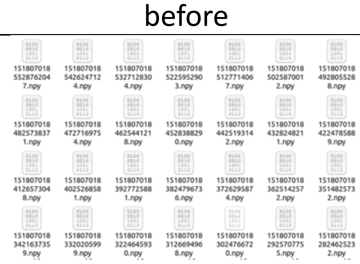
   In real world hardware setups, different sensors are positioned at various locations. However, for effective algorithms, achieving a unified reference coordinate system is important. This requires accurate calibration of the spatial relationship between diverse sensors, such as LiDAR and camera. The ‘sdc-calibration.bag’ file contains 3D depth information from LiDAR and 2D images from the camera. Through calibration with marked objects, we can establish the relationship between the ‘camera_frame’ and ‘lidar_frame.’ The calibrated results enhance a basic 2D image with depth information, providing valuable real-time distance information when the camera detects objects.

   b. what issues may arise if we use sdc-moving.bag or sdc-NCTU.bag?

   (a) update rate issue among sensors
   Due to the different update rates among sensors, when overlaying camera and LiDAR, it is necessary to perform temporal processing. This ensures that information from the same timestamp is correctly aligned during the overlay.

   (b) naming issue after subscribing.
   the version provided by the TA uses the timestamp as the file name after subscribing. However , this approach can lead to difficulties in subsequent processing. Therefore, I adjusted the file naming method to solve this issue.

before

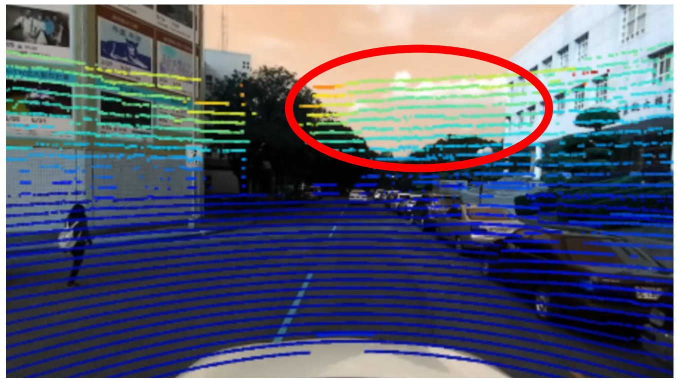


after

(c) overlay errors caused by file selection issue

Because Python’s default sorting uses binary, I encountered overlay errors when initially sorting file names. As a result, I had to design my own sorting method to address the overlay issues.

```
camera_files = sorted(os.listdir(camera_folder), key=lambda x: int(x.split('.')[0]))
lidar_files = sorted(os.listdir(lidar_folder), key=lambda x: int(x.split('.')[0]))
```



(d) Calibration issues.

Because the quality of correction directly affects the alignment results, having multiple calibration points during correction might improve the outcome. My approach is to initially display the result of the first correction. If the calibration is not satisfactory, I recalibrate again.

(e) The issue of inaccurate colors in the camera

The sky's color looks different. I think it's because OpenCV and ROS use different color orders. When going from RGB to BGR without proper handling, it causes this issue. In image recognition, it's a big problem. For example, in recognizing traffic lights, not handling it right can lead to mistakes in identifying the signals

Rviz	video