Homework 1 Report

Name: 謝元碩
Student ID: 311512015

Task1. BEV projection

i. Code Design

(1) load.py

- **降低 sensor 高度以縮小誤差**：將 sim_settings 的"sensor_height"調整為 1

```
sim_settings = {
    "scene": test_scene,  # Scene path
    "default_agent": 0,  # Index of the default agent
    "sensor_height": 1,  # Height of sensors in meters, relative to the agent
    "width": 512,  # Spatial resolution of the observations
    "height": 512,
    "sensor_pitch": 0,  # sensor pitch (x rotation in rads)
}
```

- **在 make_simple_cfg 函式中加入 bev sensor**：將 sensor orientation 的 roll 項調整為順時針旋轉 90 度(俯視)，sensor position 的最後一項改為-1.5(平移座標)，其餘 sensor 不變

```
# a BEV sensor, to the agent
bev_sensor_spec = habitat_sim.CameraSensorSpec()
bev_sensor_spec.uuid = "BEV_sensor"
bev_sensor_spec.sensor_type = habitat_sim.SensorType.COLOR
bev_sensor_spec.resolution = [settings["height"], settings["width"]]
bev_sensor_spec.position = [0.0, settings["sensor_height"], -1.5]
bev_sensor_spec.orientation = [
    -np.pi/2,
    0.0,
    0.0,
]
bev_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE
```

- 在 navigateAndSee 函式中，加入顯示 BEV 視角的畫面。函式最後需**回傳 front view image(color sensor)、BEV view image(BEV sensor)**，如此可在使用鍵盤的過程中儲存移動時觀測到的影像

```
def navigateAndSee(action=""):
    if action in action_names :
        observations = sim.step(action)
        #print("action: ", action)

        cv2.imshow("RGB", transform_rgb_bgr(observations["color_sensor"]))
        cv2.imshow("BEV", transform_rgb_bgr(observations["BEV_sensor"]))
        #cv2.imshow("depth", transform_depth(observations["depth_sensor"]))
        #cv2.imshow("semantic", transform_semantic(observations["semantic_sensor"]))
        agent_state = agent.get_state()
        sensor_state = agent_state.sensor_states['color_sensor']
        print("camera pose: x y z rw rx ry rz")
        print(sensor_state.position[0],sensor_state.position[1],
              sensor_state.position[2],  sensor_state.rotation.w,
              sensor_state.rotation.x, sensor_state.rotation.y, sensor_state.rotation.z)

        return transform_rgb_bgr(observations["color_sensor"]),transform_rgb_bgr(observations["BEV_sensor"]),
    transform_depth(observations["depth_sensor"]), sensor_state.position[0],sensor_state.position[1],
    sensor_state.position[2]
```

- 加入鍵盤按鍵 q 和 e 的條件：按下 q 將儲存 front view 的圖像，按下 e 將儲存 BEV view 的圖像。s[0]即對應到 navigateAndSee 函式回傳的第 1 項變數(front view image)，s[1]同樣對應函式回傳的第 2 項變數(BEV view image)

```
while True:
```

⋮

```
    keystroke = cv2.waitKey(0)
    if keystroke == ord(FORWARD_KEY):
        action = "move_forward"
        s = navigateAndSee(action)
        print("action: FORWARD")
        save_count = save_count + 1
    elif keystroke == ord(LEFT_KEY):
        action = "turn_left"
        s = navigateAndSee(action)
        print("action: LEFT")
        save_count = save_count + 1
    elif keystroke == ord(RIGHT_KEY):
        action = "turn_right"
        s = navigateAndSee(action)
        print("action: RIGHT")
        save_count = save_count + 1
    elif keystroke == ord(FINISH):
        print("action: FINISH")
        break
    elif keystroke == ord(SAVE_FRONT):  #get Front_view image
        cv2.imwrite('Front_view.png',s[0])
        print("Save front image.")
    elif keystroke == ord(SAVE_BEV):    #get BEV_view image
        cv2.imwrite('BEV_view.png',s[1])
        print("Save BEV image.")
    else:
        print("INVALID KEY")
        continue
```

```
SAVE_FRONT="q"
SAVE_BEV="e"
```

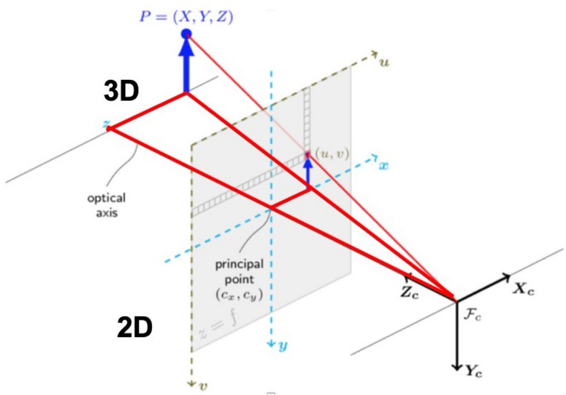
(2) bev.py

- 在 click_event 函式中，當滑鼠左鍵被觸發，將會累加按下的次數，同時記錄點座標

```
global count
# checking for left mouse clicks
if event == cv2.EVENT_LBUTTONDOWN:

    count = count + 1

    #print(x, ' ', y)
    points.append([x, y])
    font = cv2.FONT_HERSHEY_SIMPLEX
    #cv2.putText(img, str(x) + ',' + str(y), (x+5, y+5), font, 0.5, (0, 0, 255), 1)
    cv2.circle(img, (x, y), 3, (0, 0, 255), -1)
    cv2.imshow('image', img)
```

- 這裡將說明我如何完成 top_to_front projection。在 2D image 與 3D coordinate system 的轉換中，使用到 Pinhole Camera Model，公式及示意圖如下：

$$\frac{u}{f} = \frac{X}{Z} \ , \ \frac{v}{f} = \frac{Y}{Z}$$



首先在 2D 轉 3D，由於數值需配合上圖三角形比例關係，u、v 需先扣掉 width/2、height/2，始能換算 x 跟 y，此步驟的 z 為 BEV sensor 的預設高度 1；3D 轉成 2D 的部分，同樣先把經轉移矩陣換算過的 xyz 帶入公式求出 u、v，再將 width/2、height/2 加回來，成為 front view image 的座標，如此就能順利完成轉換。f 為使用 FOV 公式求得，公式如下：

$$f = \frac{width}{2} \cot \frac{FOV}{2} = 256$$

轉移矩陣的部分，我們需將 bev sensor 的坐標系轉換到 front view sensor 的坐標系。以自己定義的 bev view 及 front view 坐標系，平移只有 z 軸部分，　為-1.5，轉動為對 x 軸順時針轉 90 度，其所得轉移矩陣如下：

$$
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

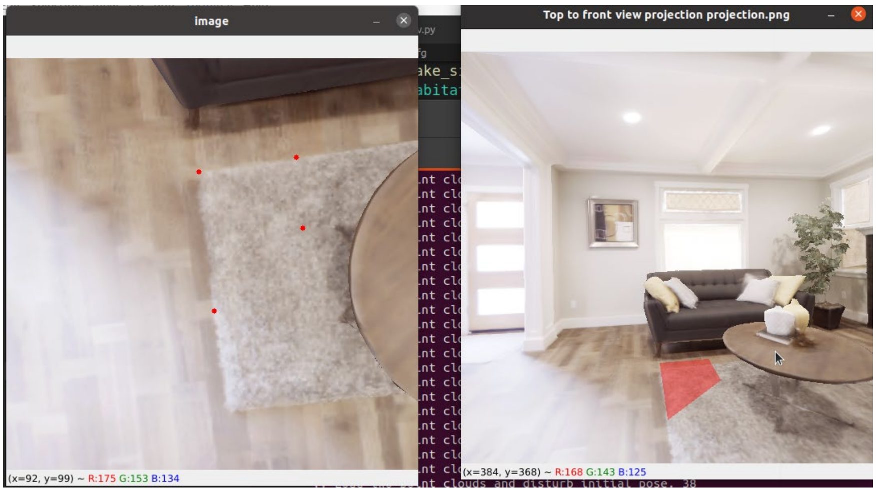
接著在程式中使用此轉移矩陣對 bev 座標做轉換，並轉成 list 型態以便取值做運算。

經上述三個步驟的轉換，即產生經 projection 後的新 uv 座標。只要將儲存坐標系的 list 回傳，bev projection 即大功告成。程式碼如下：

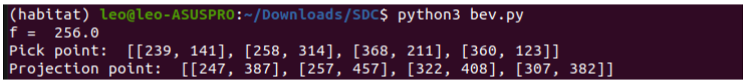
```
#######Projection Algorithm#######
global count
f = float(self.width/2*(1/math.tan(fov/180*math.pi/2)))
T = np.array([[1,0,0,0],[0,0,-1,0],[0,1,0,-1.5],[0,0,0,1]])

for i in range(0,count):
    #2D to 3D transformation of bev image
    x_bev = Z*(points[i][0]-256)/f
    y_bev = Z*(points[i][1]-256)/f

    #coordinate transformation, bev to front view
    xyz_front = np.dot(T, [[x_bev],[y_bev],[Z],[1]])
    xyz_front = np.transpose(xyz_front).tolist()

    #3D to 2D transformation of front view image
    u_front = int(f*xyz_front[0][0]/xyz_front[0][2]*(-1)+256)
    v_front = int(f*xyz_front[0][1]/xyz_front[0][2]+256)

    #save points
    uv_front.append([u_front, v_front])

###print the result###
print("f = ", round(f, 2))
print("Pick point: ", points)
print("Projection point: ", uv_front)

new_pixels = uv_front

return new_pixels
```

ii. Result

- Example 1:

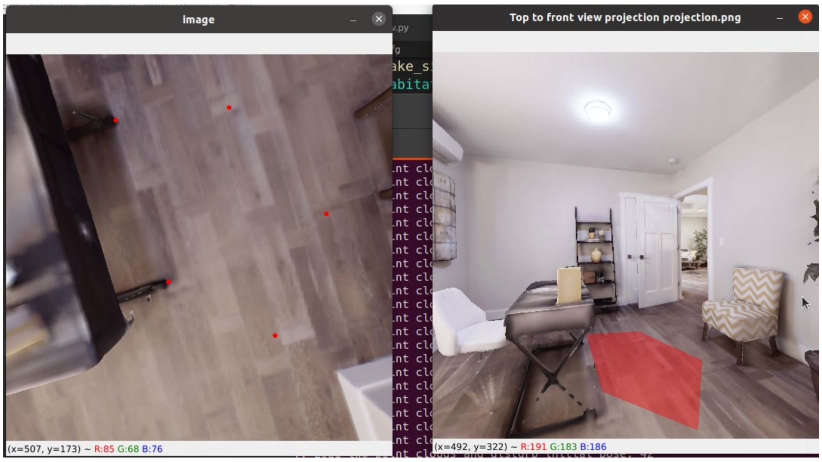


Output screen:

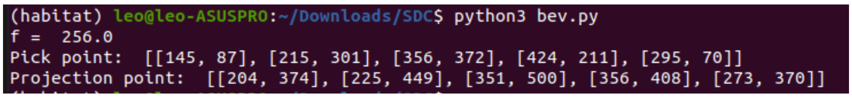


```
(habitat) leo@leo-ASUSPRO:~/Downloads/SDC$ python3 bev.py
f =  256.0
Pick point:  [[239, 141], [258, 314], [368, 211], [360, 123]]
Projection point:  [[247, 387], [257, 457], [322, 408], [307, 382]]
```

- Example 2:



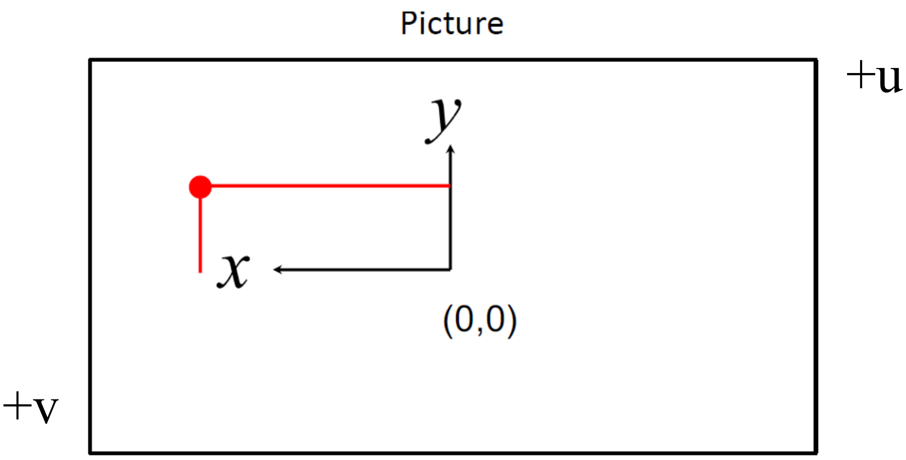
Output screen:



```
(habitat) leo@leo-ASUSPRO:~/Downloads/SDC$ python3 bev.py
f =  256.0
Pick point:  [[145, 87], [215, 301], [356, 372], [424, 211], [295, 70]]
Projection point:  [[204, 374], [225, 449], [351, 500], [356, 408], [273, 370]]
```

iii. Discussion

思考演算法的過程中，我發現若相機中心與影像介面的坐標系設定不同(uv 軸與 xy 軸的相對朝向)，2D 與 3D 座標間的轉換容易導致 xy 軸方向有正負號的差別，如果沒注意到，將會使 projection 結果出錯。



在我定義的坐標系上做轉換時，當進行 front view image 的 3D 轉 2D 時，x 軸換算中的 Pinhole Camera Model 部分需先多乘上一個-1，再去+256 才會正確(如程式碼)，而這也是滿值得討論與深思的。

Task2. 3D Scene Reconstruction

i. Code Design

(1) load.py

- **儲存在模擬環境行走時所顯示的 rgb image 及 depth image**：在這個 task 中，load.py 最主要的功能，就是儲存所有行走過程看見的 rgb 及 depth 圖像，這些圖片檔將會用來進行 3D 重建。由於電腦性能限制，我設定為每走 2 步記錄一次，資料量才不會太過龐大。
- 另外也同時**記錄當下位置的卡式座標(xyz)**，以畫出 ground truth trajectory
- 最後**更新儲存圖片的總次數**到 count.txt，這個數值將會在 reconstruct.py 中讀入做使用

```
#每行走2步，會儲存一次rgb及depth圖像，同時記錄當下座標及更新圖像張數
if save_count%2 == 0:

    count = count + 1
    save_count = 0

    #儲存rgb及depth圖像
    cv2.imwrite('img/rgb/rgb' + str(count) + '.png',s[0])
    cv2.imwrite('img/depth/depth' + str(count) + '.png',s[2])
    print("Save image " + str(count))

    #記錄當下座標記錄當下座標
    with open('ground_xyz.txt', 'a') as outfile:
        outfile.write(str(s[3]) + ' ' + str(s[4]) + ' ' + str(s[5]) + ' \n')

    #更新圖像張數
    with open('count.txt', 'w') as outfile:
        outfile.write(str(count))
```

(2) reconstruct.py

- 宣告將會用到的兩種 count 全域變數(註解如圖)

```
final_count = 1 #counting point_cloud reconstruction
img2pcd_count = 1 #counting image_to_point_cloud
voxel_size = 0.05  # means 5cm for this dataset
```

- 函式 part 1：建立點雲

  在 get_pointcloud 函式中，會讀入兩張分別為 rgb 及 depth 資訊的圖片檔，利用圖檔的資訊，做 2D 轉 3D 的座標轉換，並將轉換後的卡式座標及 rgb 資料分別存到 save_pixel 和 save_rgb。

  接著宣告一個 PointCloud，把前述 data 存入點雲中，最後將點雲存成 pcd 檔。

```python
#####depth_image_to_point_cloud#####
save_rgb = []
save_pixel = []
def get_pointcloud(img2pcd_count):

    save_rgb.clear()
    save_pixel.clear()

    img_rgb = cv2.imread('img/rgb/rgb' + str(img2pcd_count) + '.png')
    img_depth = cv2.imread('img/depth/depth' + str(img2pcd_count) + '.png')

    fov = 90
    f = float(512/2*(1/math.tan(fov/180*math.pi/2)))

    for i in range(0,512):
        for j in range(0,512):
            x = (img_depth[i,j][0]/25.5)*(j-256)/f
            y = (img_depth[i,j][0]/25.5)*(i-256)/f

            if y>(-0.6):  #去除天花板
                save_pixel.append([x,y,img_depth[i,j][0]/25.5])
                save_rgb.append([img_rgb[i][j][2]/255,img_rgb[i][j][1]/255,img_rgb[i][j][0]/255])

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(save_pixel)
    pcd.colors = o3d.utility.Vector3dVector(save_rgb)
    o3d.io.write_point_cloud('pcd/' + str(img2pcd_count) +  '.pcd',pcd)
    print("Save Point Cloud " + str(img2pcd_count))
```

- 函式 part 2：點雲前置作業

  prepare_dataset 函式先對 source 點雲做初始座標轉換，接著將 source PointCloud 和 target PoitCloud 丟入 preprocess_point_cloud 函式，最後回傳初始點雲、降維後的點雲及經 fpfh_feature 計算後的點雲

  preprocess_point_cloud 函式如前述所言，主要作點雲降維和 fpfh 特徵計算

```python
#####初始化點雲#####
def prepare_dataset(voxel_size,source,target):

    print(":: Load two point clouds and disturb initial pose. " + str(final_count))

    demo_icp_pcds = o3d.data.DemoICPPointClouds()
    # source.estimate_normals()
    # target.estimate_normals()
    trans_init = np.asarray([[0.0, 0.0, 1.0, 0.0], [1.0, 0.0, 0.0, 0.0],
                             [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 0.0, 1.0]])
    source.transform(trans_init)
    #draw_registration_result(source, target, np.identity(4))
    source_down, source_fpfh = preprocess_point_cloud(source, voxel_size)
    target_down, target_fpfh = preprocess_point_cloud(target, voxel_size)
    return source, target, source_down, target_down, source_fpfh, target_fpfh


#####點雲前處理#####
def preprocess_point_cloud(pcd, voxel_size):
    #print(":: Downsample with a voxel size %.3f." % voxel_size)
    pcd_down = pcd.voxel_down_sample(voxel_size)

    radius_normal = voxel_size * 2
    #print(":: Estimate normal with search radius %.3f." % radius_normal)
    #加上法向量(後面point to plane ICP會用到)
    pcd_down.estimate_normals(
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))

    radius_feature = voxel_size * 5
    #print(":: Compute FPFH feature with search radius %.3f." % radius_feature)
    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd_down,
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100))
    return pcd_down, pcd_fpfh
```

- 函式 part3：執行 ransac registration

  此函式對 source、target 以 feature matching 的方法，進行點對點的 ransac 校正

```python
#####global registration#####
def execute_global_registration(source_down, target_down, source_fpfh,
                                target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5
    #print(":: RANSAC registration on downsampled point clouds.")
    #print("   Since the downsampling voxel size is %.3f," % voxel_size)
    #print("   we use a liberal distance threshold %.3f." % distance_threshold)
    result = o3d.pipelines.registration.registration_ransac_based_on_feature_matching(
        source_down, target_down, source_fpfh, target_fpfh, True,
        distance_threshold,
        o3d.pipelines.registration.TransformationEstimationPointToPoint(False),
        3, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeLength(
                0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDistance(
                distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(100000, 0.999))
    return result
```

- 函式 part4：做 open3d 的 ICP 配準

  這裡使用 open3d 內建的 ICP 演算法，對兩份點雲做 point to plane 的配準

```python
#####Local refinement, 這裡做point-to-plane ICP#####
def refine_registration(source, target, source_fpfh, target_fpfh, voxel_size, result_ransac):
    distance_threshold = voxel_size * 0.4
    #print(":: Point-to-plane ICP registration is applied on original point")
    #print("   clouds to refine the alignment. This time we use a strict")
    #print("   distance threshold %.3f." % distance_threshold)
    result = o3d.pipelines.registration.registration_icp(
        source, target, distance_threshold, result_ransac.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return result
```

- 函式 part5：大量點雲座標轉換

  此函式整合前述 part2~4 的函式，以無窮迴圈不斷讀入新的點雲，每次皆使用 ICP 配準後得到轉移矩陣 T，將 source 點雲轉移到 target 點雲的座標系，轉移後利用 list 存到 final 中，如此可得到完整的地圖點雲資料。每一次使用的 target 需為前一次轉移後的 source 資料，這樣地圖才會保持完整。

  另外，我將 estimated trajectory 的計算也放在這裡。estimated trajectory 的計算方式，同樣使用 ICP 配準產生的轉移矩陣，對每一個點雲的「原點」做轉換，並一一存到 list 中。透過 ICP 產生出來的座標，即代表估測的座標。

```python
#####整合過的獨立ICP function#####
def local_icp(voxel_size, total_count):

    global final_count
    final = []

    estimate_point = [[0,0,0]]
    estimate_color = []
    origin = np.array([[0],[0],[0],[1]])

    while True:

        source = o3d.io.read_point_cloud('pcd/' + str(final_count+1) + '.pcd')
        if final_count == 1:
            target = o3d.io.read_point_cloud('pcd/1.pcd')

        source, target, source_down, target_down, source_fpfh, target_fpfh = prepare_dataset(voxel_size,source,target)
        result_ransac = execute_global_registration(source_down, target_down,source_fpfh, target_fpfh,voxel_size)
        result_icp = refine_registration(source_down, target_down, source_fpfh, target_fpfh,voxel_size, result_ransac)
        T = result_icp.transformation

        # 此部份為自己寫的ICP,目前還有錯誤未解決
        # source_array = np.asarray(source_down.points)
        # target_array = np.asarray(target_down.points)
        # T = icp(source_array, target_array, result_ransac.transformation, times=20, threshold=0.001)

        if final_count == 1:
            final.append(target_down)

        target = new_target(source_down,T)
        final.append(source_down)
```

```python
    #make estimated trajectory
    origin_trans = np.dot(T,origin)
    origin_trans = np.transpose(origin_trans).tolist()
    estimate_point.append([origin_trans[0][0], origin_trans[0][1], origin_trans[0][2]])
    estimate_color.append([255,0,0])
    estimate_lines = [[i,i+1] for i in range(total_count-1)]

    if final_count == total_count-1:
        break

    final_count = final_count + 1

return final, estimate_point, estimate_color, estimate_lines
```

```python
#####將轉移後的source存到下一次使用的target#####
def new_target(source_down,T):
    return source_down.transform(T)
```

- 函式 part6：ground truth trajectory
  此函式直接將 load.py 中所存的每組 xyz 座標存入 list，以用來生成路徑

```python
#####ground_truth_trajectory#####
def ground_truth_trajectory(total_count):

    gt_count = 0
    x_list = []
    y_list = []
    z_list = []
    gt_point = []
    gt_color = []
    with open('ground_xyz.txt', 'r') as infile:
        for line in infile.readlines():
            s = line.split(' ')
            x = s[0:1]
            y = s[1:2]
            z = s[2:3]
            x_list.append(float(x[0]))
            y_list.append(float(y[0]))
            z_list.append(float(z[0]))
    infile.close()

    while gt_count < total_count:

        a = x_list[0]
        b = y_list[0]
        c = z_list[0]
        for i in range(len(x_list)):
            gt_point.append([x_list[i]-a, y_list[i]-b, -(z_list[i]-c)])

        gt_color.append([0,0,0])
        gt_lines = [[i,i+1] for i in range(total_count-1)]

        gt_count = gt_count + 1

    return gt_point, gt_color, gt_lines
```

- 函式 part7：計算 estimated trajectory 與 ground truth trajectory 路徑平均誤差
  此函式將上述兩種路徑的每個對應點取歐式距離後，取平均值

```python
#####distance between ground truth trajectory and estimated trajectory####
def calculate_distance(total_count):
    cal_count = 0
    dis = 0
    while cal_count < total_count:
        x = gt_point[cal_count][0] - estimate_point[cal_count][0]
        y = gt_point[cal_count][1] - estimate_point[cal_count][1]
        z = gt_point[cal_count][2] - estimate_point[cal_count][2]
        dis = dis + math.sqrt(x**2+y**2+z**2)
        cal_count = cal_count +1

    mean_dis = dis/total_count
    print("Mean distance: " + str(mean_dis))
```

- 函式 part8：自己寫的 ICP(未完成)

  依照講義的說明及步驟，試著寫出 ICP 演算法的雛型，目前有三組函式：使用 SVD 計算轉移矩陣、尋找鄰近點和 ICP 演算法。但目前執行結果仍是失敗的。

```python
##########################ICP algorithm(still has error)##########################
#find T with SVD
def best_fit_transform(source, target):

    dimension = 3

    # 將所有點扣掉質心
    source_center = np.mean(source, axis=0)
    target_center = np.mean(target, axis=0)
    source_new = source - source_center
    target_new = target - target_center

    # rotation matrix
    W = np.dot(source_new.T, target_new)
    U, S, VT = np.linalg.svd(W)
    R = np.dot(VT.T, U.T)

    # add translation and rotation into T
    t = target_new.T - np.dot(R,source_new.T)
    T = np.identity(dimension+1)
    T[:dimension, :dimension] = R
    T[:dimension, dimension] = t

    return T, R, t

#尋找鄰近點(source to target)
def nearest_neighbor(source, target):

    neighbor = NearestNeighbors(n_neighbors=1)
    neighbor.fit(target)
    distance, find = neighbor.kneighbors(source, return_distance=True)
    return distance.ravel(), find.ravel()
```

```python
#icp algorithm function
def icp(source, target, ransac_T, times, threshold):

    dimension = 3

    source_new = np.ones((dimension+1,source.shape[0]))
    target_new = np.ones((dimension+1,target.shape[0]))
    source_new[:dimension,:] = np.copy(source.T)
    target_new[:dimension,:] = np.copy(target.T)

    source_new = np.dot(ransac_T, source_new)

    old_error = 0
    for i in range(times):
        #尋找source和target的鄰近點
        distance, find = nearest_neighbor(source_new[:dimension,:].T, target_new[:dimension,:].T)

        # 求此迭代產生的T
        T,_,_ = best_fit_transform(source_new[:dimension,:].T, target_new[:dimension,find].T)

        # 更新source
        source_new = np.dot(T, source_new)

        # 誤差判斷，小於閥值就停止尋找
        mean_error = np.mean(distance)
        if np.abs(old_error - mean_error) < threshold:
            break
        old_error = mean_error

    # 計算經迭代後產生的T
    T,_,_ = best_fit_transform(source, source_new[:dimension,:].T)

    return T
##########################ICP algorithm(still has error)##########################
```

- 主函式

  在主函式中，首先將所有圖檔轉成點雲資料，接著進行 ICP 配準及重建，此時會回傳轉移過後的所有點雲資料 final，以及 estimated point。

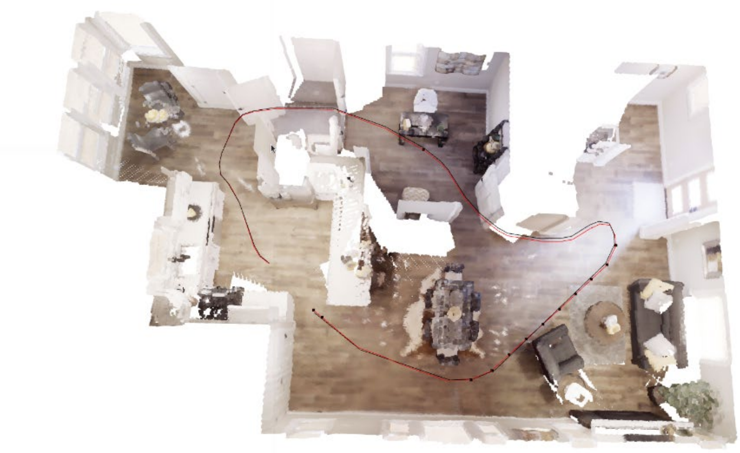
  後續會執行 ground truth trajectory 的函式以取得移動座標，並將兩種路徑宣告成 LineSet 格式，再計算平均誤差並輸出在終端機中。

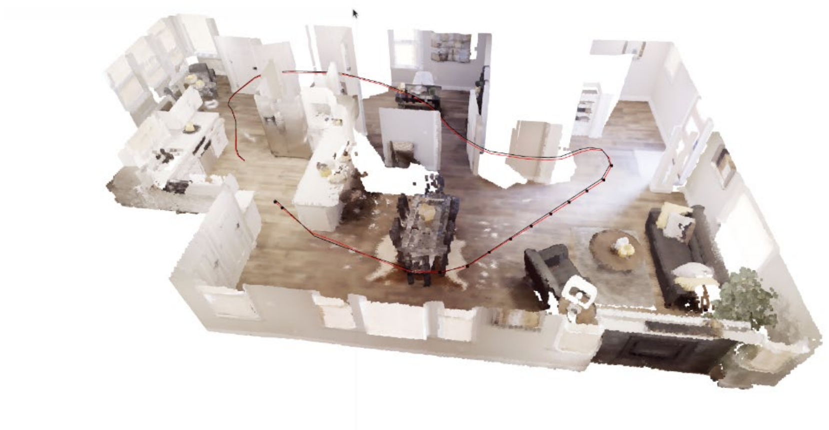
  最後宣告點雲、把資料丟入並視覺化，完整的 3D 地圖、兩種路徑即大功告成。

```python
#讀取圖片張數
with open('count.txt', 'r') as infile:
    total_count = int(infile.read())

#生成點雲
while img2pcd_count <= total_count:
    get_pointcloud(img2pcd_count)
    img2pcd_count = img2pcd_count+1

#執行ICP重建函式(包含estimated trajectory)
final, estimate_point, estimate_color, estimate_lines = local_icp(voxel_size, total_count)

#建立estimated trajectory
estimate = o3d.geometry.LineSet()
estimate.lines = o3d.utility.Vector2iVector(estimate_lines)
estimate.colors = o3d.utility.Vector3dVector(estimate_color)
estimate.points = o3d.utility.Vector3dVector(estimate_point)

#建立ground truth trajectory
gt_point, gt_color, gt_lines = ground_truth_trajectory(total_count)
ground = o3d.geometry.LineSet()
ground.lines = o3d.utility.Vector2iVector(gt_lines)
ground.colors = o3d.utility.Vector3dVector(gt_color)
ground.points = o3d.utility.Vector3dVector(gt_point)

#計算兩種路徑之平均誤差
calculate_distance(total_count)

#建立3D點雲資料
final_pcd = o3d.geometry.PointCloud()
for i in range(len(final)):
    final_pcd = final_pcd + final[i]

#視覺化所有點雲資料及存檔
o3d.visualization.draw_geometries([final_pcd, estimate, ground])
o3d.io.write_point_cloud('final.pcd',final_pcd)
```

ii. Result

- Floor 1



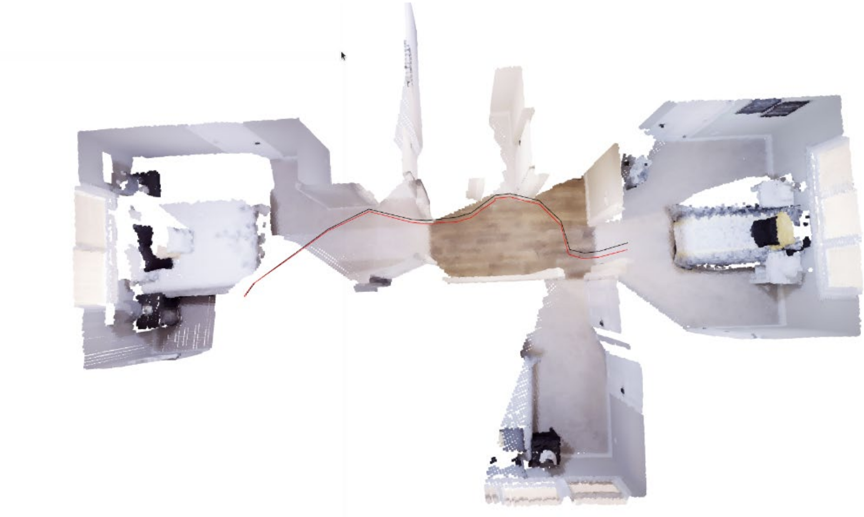


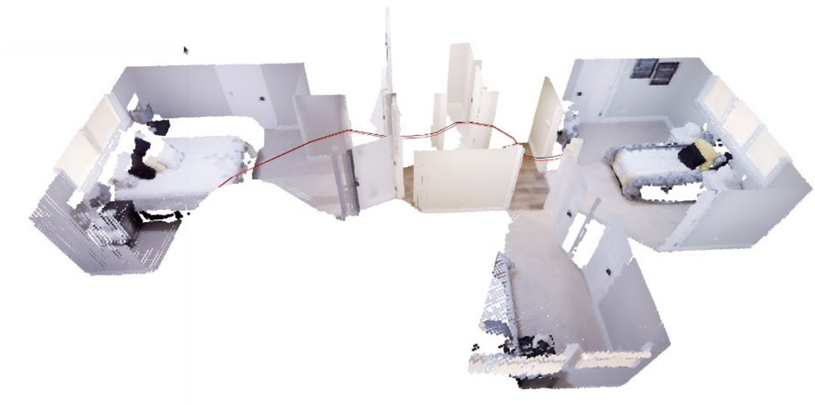
Output screen:

Include Mean L2 distance between ground truth and estimated trajectory

```
:: Load two point clouds and disturb initial pose. 74
:: Load two point clouds and disturb initial pose. 75
:: Load two point clouds and disturb initial pose. 76
Mean distance: 0.047678354876298165
```

- Floor 2(因旋轉太多易重建失敗，故 data size 獲取較少)





Output screen:

Include Mean L2 distance between ground truth and estimated trajectory

```
:: Load two point clouds and disturb initial pose. 40
:: Load two point clouds and disturb initial pose. 41
:: Load two point clouds and disturb initial pose. 42
Mean distance: 0.0627859069802546
```

iii. Discussion

重建二樓時，由於二樓空間皆為單一出入口，行走時皆須旋轉較大的角度才能離開房間，多次且大量的旋轉，導致建圖失敗率極高，代表經過 ICP 得到的轉移矩陣在此狀況易出錯，如何去改善針對此狀況的 ICP 是一個很好的議題。

ground truth and estimated trajectory 兩種路徑事實上並非重疊，不同的配準演算法會影響路徑的正確率。行走過程若產生誤差，似乎會發生誤差越來越大的狀況，無法再次收斂，導致路徑平均誤差增大。如何讓規畫路徑具強健性，值得討論。