智慧感知與決策

B073012003 鄧書桓

Task1:

i. Code:

Load. py:

根據已有的 rgb, depth sensor 新增兩個 sensor，分別是 BEV_rgb_sensor_spec(用於看 BEV 的 rgb 畫面)和 BEV_depth_sensor_spec(用來看 BEV 的深度畫面)，並讓這兩個位於 agent 前方 1.3 公尺的位置，以達成同步儲存 front view 與 bev view 的要求，詳細過程將在以下 code 進行解說:

```
# BEV_RGB
BEV_rgb_sensor_spec = habitat_sim.CameraSensorSpec()
BEV_rgb_sensor_spec.uuid = "BEV_color_sensor"
BEV_rgb_sensor_spec.sensor_type = habitat_sim.SensorType.COLOR
BEV_rgb_sensor_spec.resolution = [settings["height"], settings["width"]]
#使BEV位置位於front sensor前方1.3m
BEV_rgb_sensor_spec.position = [0.0, settings["sensor_height"], -1.3]
#改變角度使sensor朝下
BEV_rgb_sensor_spec.orientation = [
    -(90 * np.pi/180),
    0.0,
    0.0,
]
BEV_rgb_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE
```

```
# depth snesor
BEV_depth_sensor_spec = habitat_sim.CameraSensorSpec()
BEV_depth_sensor_spec.uuid = "BEV_depth_sensor"
BEV_depth_sensor_spec.sensor_type = habitat_sim.SensorType.DEPTH
BEV_depth_sensor_spec.resolution = [settings["height"], settings["width"]]
# 與上述同理(1.3m)
BEV_depth_sensor_spec.position = [0.0, settings["sensor_height"], -1.3]
# 角度也是
BEV_depth_sensor_spec.orientation = [
    -(90 * np.pi/180),
    0.0,
    0.0,
]
BEV_depth_sensor_spec.sensor_subtype = habitat_sim.SensorSubType.PINHOLE
```

新增指令 p 鍵，用於同時儲存 bev 與 front view 的 rgb_img。

```
elif keystroke == ord(SAVE_BEV_AND_FRONT_IMAGE):
    cv2.imwrite('./images_for_projection/BEV_img.png', BEV_image)
    cv2.imwrite('./images_for_projection/front_img.png', front_image)
    print("action: save front image and BEV image")
    print(posit)
```

Bev.py:

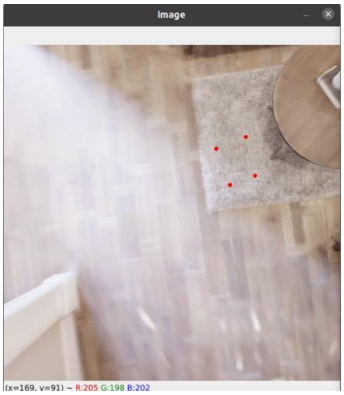
根據老師上課交的 pinhole camera model 原理來實現，但這邊並沒有使用 camera projection matrix 來實現，而是先讓每個點的 u, v 轉成 x,y,z 後再經過轉移矩陣相乘，得到 front view 下的 x, y ,z，之後再轉回 u,v，以此來完成 projection，詳細過程將在以下 code 進行解說:

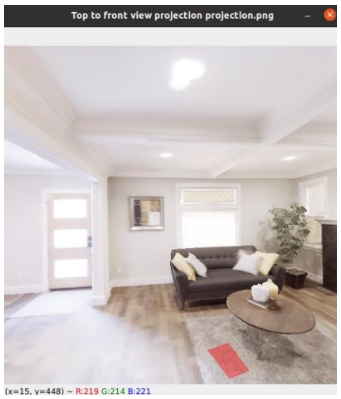
```
def top_to_front(self, theta=0, phi=0, gamma=0, dx=0, dy=0, dz=0, fov=np.pi/2):
    # 由 fov 與 img 大小得出 focal
    self.focal = (self.height/2 * cot(fov/2))
    # 計算後得到的 transformation matrix，對 x 轉 90 度,z 平移-1.3
    trans_mat = [[1, 0, 0, 0],
                 [0, 0, -1, 0],
                 [0, 1, 0, -1.3],
                 [0, 0, 0, 1]
                 ]
    print("points are : ", points)
    num = len(points)
    # 將 front img 上所選取的 pixel 點 * Z/f 得到 front view 的 x y
    wpoints_BEV = np.multiply(
        (np.array(points)-256).tolist(), (1.5/self.focal))
    wpoints_BEV_mat = [[1.5 for col in range(num)] for row in range(4)]
    # 將得到 n 組的 x, y 與固定的 z 變成 4*n 的行式，以便之後矩陣相乘(與轉換矩陣)
    for i in range(num):
        wpoints_BEV_mat[3][i] = 1
        for j in range(2):
            wpoints_BEV_mat[j][i] = wpoints_BEV[i][j]
    print("points in BEV world view (matrix) is : ", wpoints_BEV_mat)
    # 矩陣相乘後得到位於 bev view 下各點的 x, y, z
    wpoints_front_mat = np.dot(trans_mat, wpoints_BEV_mat)
    print("points in front world view (matrix) is : ", wpoints_front_mat)
    new_pixels = [[0 for col in range(2)] for row in range(num)]
    # 再依序將 x, y ,z 轉成 bew view 下的 u, v (*f/Z)
    for i in range(num):
        for j in range(2):
            new_pixels[i][j] = int(wpoints_front_mat[j][i] *
                                   (self.focal / wpoints_front_mat[2][i]))
            # 加負號的原因為定義的 x,y 方向與 u,v 方向相同，故需加負號
            if j == 0:
                new_pixels[i][j] = -new_pixels[i][j]
    new_pixels = (np.array(new_pixels)+256).tolist()
```

```
    print("points in front pixel view (matrix) are : ", new_pixels)
    return new_pixels
```

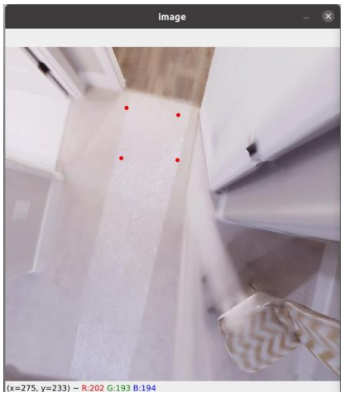
ii. Result and Discussion

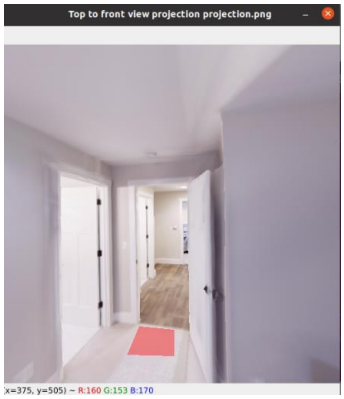
Floor1:





Floor2:





跟據上方的 4 張圖可知，由 bev 投影到 front view 非常成功，但若所點選的位置差太多或跑到牆上，將會投影失敗，我認為是因為這個只是單存圖案頂點的投影，若要成功的話，需要將點連成的線進行 sampling。

Task2:

i. Code:

先在 load.py 將所要的 posit 位置與 img、depth 資料存下來。

```
#將位置存到 txt
def store_posit(posit_data, posit):
    #使用 a 的方式於 txt 檔寫入資料
    with open("posit_data.txt", "a") as f:
      #將資料以 space 區隔，方便之後區分 x,y,z(因為存入 txt 檔的內容為 strig)
      f.write(str(posit[0]) + " " + str(posit[1]) +
              " " + str(posit[2]) + " " + "\n")
```

```
# 儲存所要的 rgb 與 depth 資訊(這邊僅以重要片段 code 表示)
elif keystroke == ord(SAVE_MANY_FRONT_IMG): #當按下 r 鍵時執行
        #將不同的照片的位置利用數字編碼，以便之後儲存照片
        path_multi_rgb.append(os.path.join(
            path_pkg, 'front_rgb_img' + '_' + str(count_rgb) + '.png'))
        path_multi_depth.append(os.path.join(
            path_pkg, 'front_depth_img' + '_' + str(count_depth) + '.png'))
        #儲存各個位置的照片
        cv2.imwrite(path_multi_rgb[count_rgb-1], front_image)
        cv2.imwrite(path_multi_depth[count_depth-1], Depth_image)
        print("action: save {} th front_rgb_image".format(count_rgb))
        print("action: save {} th front_depth_img".format(count_depth))
        count_rgb = count_rgb + 1
        count_depth = count_depth + 1
        store_posit(posit_data, posit)
        print(posit)
```

接著於 reconstruct 先將所儲存的位置、rgb、depth 資料引入，並使用 depth_image_to_point_cloud 將 2D 資料轉成 3D 資料，並以 pcd 檔的形式儲存，詳細過程將在以下 code 進行解說:

```
# 將資料存入後經此 function 將 2D 轉成 3D
def depth_image_to_point_cloud(path_img, pcd_path):
    focal = (512/2 * cot(np.pi/2/2))
    # 用於儲存有幾筆資料
    count = int(len([name for name in os.listdir(path_img)
                     if os.path.isfile(os.path.join(path_img, name))])/2)
    print("There are {} pcds need to be made".format(count))
    for num in range(count):
        # 讀取資料
        img = cv2.imread(path_img + 'front_rgb_img_' + str(num+1) + '.png', 1)
        depth_img = cv2.imread(
            path_img + 'front_depth_img_' + str(num+1) + '.png', 1)
        pixel_rgb = []
        pixel_xyz = []
        # 將資料由 2D 轉成 3D，將各個 pixel 的 u,v 乘上 z/f 以得到以那張照片為 frame 的 x,y,z
        # RGB 則是正規化到 0~1
        for i in range(512):
            for j in range(512):
```

```
                #限制 y 的大小，以此來去除天花板
                if (((i-256) * depth_img[i, j, 1]/focal / 25.5) > -0.65):
                    #注意這邊輸入的是 b,g,r 而非 r,g,b
                    pixel_rgb.append([img[i, j, 2]/255,
                                      img[i, j, 1]/255, img[i, j, 0]/255])
                    pixel_xyz.append([(j-256) * depth_img[i, j, 2]/focal / 25.5,
                                      (i-256) * depth_img[i, j, 1]/focal / 25.5, depth_img[i, j,
0] / 25.5])
        # 以下將資料以點雲方式呈現
        # 初始化點雲
        pcd = o3d.geometry.PointCloud()
        # 將位置與 rgb 寫入
        pcd.points = o3d.utility.Vector3dVector(pixel_xyz)
        pcd.colors = o3d.utility.Vector3dVector(pixel_rgb)
        # 將所轉換成的 pcd 儲存至指定資料夾
        o3d.io.write_point_cloud(
            pcd_path + "picture" + str(num+1) + ".pcd", pcd)
        print("{} th pcd is made.............{}%".format(
            num+1, int((num+1)/count * 100)))
    return count
```

將儲存好的 pcd 經過 local_icp_algorith 函式進行疊圖，想法為將第一張 pcd 視為 target pcd，將第二張 pcd 視為 source pcd，並將此疊圖到 target pcd 上，而結果將被視為新的 target pcd 並存入 target_final(用來儲存所有相對於 target pcd 的疊圖 pcd)，接著便將新的 pcd 視為新的 source pcd，並與新的 target pcd 進行疊圖，以此類推……;此外，在實現過程中還會使用到其他 function，詳細作用會在以下 code 中進行解說。另外，會在疊圖過程中將各個 transformation 記錄下來，並利用它進行 estimation 點的轉換。想法為各點於原 frame 下是[0, 0, 0]，將此與相對應的 transformation 相乘，以此轉換到 base frame 下，這樣才能做之後 estimation 線的實現。

Icp:

```
#整合後的 icp
def local_icp_algorithm(camera_points, colors):
    target_final = []
    #將第一張圖視為 target dcp，並存入 target_final(用來儲存所有相對於 target pcd 的疊圖 pcd)
    x = o3d.io.read_point_cloud('./pcd_data/picture1.pcd')
    target_final.append(x)
    target_tp = x
    #開始疊圖
    for num_con in range(count-1):
```

```
        # 自己設的 voxel_size
        voxel_size = 0.05
        #將資料做預處理
        source, target, source_down, target_down, source_fpfh, target_fpfh =
prepare_dataset(target_tp,
                                                                        voxel
_size, num_con)
        #將資料做 global registration
        result_ransac = execute_global_registration(source_down, target_down,
                                                    source_fpfh, target_fpfh,
                                                    voxel_size)
        #將資料做 point-to-plane ICP(用來細化點雲，可使點雲看起來更完整)
         #注意:這邊是使用 source_down 與 target_down 當作輸入(與 tutorial 不同)，這樣可以使點雲降維，進
而使疊圖更加快速
        result_icp = refine_registration(source_down, target_down, source_fpfh, target_fpfh,
                                         voxel_size, result_ransac)
        #用來做 source dcp 最終的轉換疊圖，並將結果存入 terget_final(用來儲存所有相對於 target pcd 的
疊圖 pcd)
        target_tp = get_new_dcp(source, target_final,
                                result_icp.transformation, num_con)
        #用來儲存 estimate 點的位置與之後要連線線的顏色
        store_relative_pos(colors, camera_points, result_icp.transformation)


    #全部做完後可視化成果
    o3d.visualization.draw_geometries(target_final)
    #將每個疊完圖的成果儲存至 final_pcd
    final_pcd = o3d.geometry.PointCloud()
    for i in range(count):
        final_pcd = final_pcd + target_final[i]
    print("reconstruct by ICP is done!!!!")
    return final_pcd
```

使用到的其他 function:

```
# 將點雲做前處理(使 pcd 降維)
def preprocess_point_cloud(pcd, voxel_size):
      #將點雲降維，這樣可以使點雲降維，進而使疊圖更加快速
    pcd_down = pcd.voxel_down_sample(voxel_size)
    radius_normal = voxel_size * 2
    pcd_down.estimate_normals(
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_normal, max_nn=30))
```

```
    radius_feature = voxel_size * 5
    pcd_fpfh = o3d.pipelines.registration.compute_fpfh_feature(
        pcd_down,
        o3d.geometry.KDTreeSearchParamHybrid(radius=radius_feature, max_nn=100))
    return pcd_down, pcd_fpfh

# 準備資料(提取 feature-FPFH)
def prepare_dataset(target, voxel_size, num_con):
    source = o3d.io.read_point_cloud(
        './pcd_data/picture' + str(num_con+2) + '.pcd')
    # 尋找 source 點雲與 target 點雲的法向向量(以便做 feature mapping)
    source.estimate_normals()
    target.estimate_normals()
    # 初始化轉移矩陣(沒啥功能)
    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]])
    # 將 source pcd 轉換到 target pcd frame
    source.transform(trans_init)
    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

# 使用 RANSAC 做 registration(於附近 33 維 FPFH 特徵空間中的最近鄰來檢測的)
def execute_global_registration(source_down, target_down, source_fpfh,
                                target_fpfh, voxel_size):
    distance_threshold = voxel_size * 1.5
    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
```

```
# 這裡做 point-to-plane ICP(用來細化點雲，可使點雲看起來更完整)
def refine_registration(source, target, source_fpfh, target_fpfh, voxel_size, result_ransac):
    distance_threshold = voxel_size * 0.4
    result = o3d.pipelines.registration.registration_icp(
        source, target, distance_threshold, result_ransac.transformation,
        o3d.pipelines.registration.TransformationEstimationPointToPlane())
    return result

#用來做 source dcp 最終的轉換疊圖，並將結果存入 terget_final(用來儲存所有相對於 target pcd 的疊圖 pcd)
def get_new_dcp(source, target_final, transformation, num_con):
    print("epoch :{}.............{}%".format(
        num_con+1, int((num_con+1)/(count-1) * 100)))
    source.transform(transformation)
    target_final.append(source)
    return source

#用來儲存 estimate 點的位置與之後要連線線的顏色
def store_relative_pos(colors, camera_points, transformation):
    #最後加上 1 是方便做矩陣乘法
    temp = np.array([0, 0, 0, 1])
    #將每個位置的點與該 trasformation 相乘，以轉乘 target pcd 的 frame
    temp = transformation @ temp
    temp = temp[0:3].tolist()
    camera_points.append(temp)
    colors.append([1, 0, 0])
```

藉由上述的 function，可得到 estimation 的點，並藉由 open3d 內建的 function 來實現，詳細過程將在以下 code 進行解說:

```
#初始化 estimate 點的位置與之後要連線線的顏色與標記要將哪幾個點相連
def initial_estimate():
    camera_points = []
    camera_points.append([0, 0, 0])
    colors = []
    lines = []
    #將 1 與 2 相連,3 與 4 相連...依此類推
    for i in range(count-1):
        temp_line = [i, i+1]
        lines.append(temp_line)
    return camera_points, colors, lines
```

```
#製造 estimate 的線
line_set = o3d.geometry.LineSet()
#將 estimate 的點存入 o3d.geometry.LineSet
line_set.points = o3d.utility.Vector3dVector(camera_points)
#將 estimate 的線存入 o3d.geometry.LineSet
line_set.lines = o3d.utility.Vector2iVector(lines)
#將 estimate 的顏色存入 o3d.geometry.LineSet
line_set.colors = o3d.utility.Vector3dVector(colors)
#將 estimate 與先前疊圖的 pcd 一同呈現
o3d.visualization.draw_geometries([final_pcd, line_set])
# o3d.io.write_point_cloud("./estimate_pcd.pcd", final_pcd)
print("estimate is done!!!!")
```

之後式 ground truth 的實現，首先將 load.py 儲存的位置資料(.txt)寫入並調整型態(變成 float)，接下來使用自製的 ground_truth()函示計算 ground truth 的點與儲存之後要連線線的顏色與標記要將哪幾個點相連。其中，將第一筆資料當成 base，並使每筆資料與 base 相減，以達成相對於第一張圖的座標。最後將此計算過後的 point 輸入至 open3d 內建的 function 來實現，詳細過程將在以下 code 進行解說:

```
#將 load.py 儲存的位置(.txt)寫入並調整型態(變成 float)
def get_posit():
    posit_data = []
    #開啟檔案
    with open("posit_data.txt") as f:
        for line in f.readlines():
            #以空白當成分割線
            line = line.split(' ')
            #保存所需資料
            line = line[0:3]
            posit_data.append(line)
    #轉成 float 型態
    for i in range(count):
        for j in range(3):
            posit_data[i][j] = float(posit_data[i][j])
    return posit_data

#用來計算 ground truth 的點與儲存之後要連線線的顏色與標記要將哪幾個點相連
def ground_truth():
    #將 load.py 儲存的位置(.txt)寫入並調整型態(變成 float)
    posit_data = get_posit()
```

```
    #把第一筆資料當成 base
    base = [posit_data[0][0], posit_data[0][1], posit_data[0][2]]
    ground_truth_posit = []
    ground_truth_color = [[0, 0, 0]]
    for i in range(count):
        #將每筆資料與 base 相減，以達成相對於第一張圖的座標
        gt_temp = [posit_data[i][k] - base[k] for k in range(3)]
        #因為我們是往-z 方向前進，故 z 須加個-號
        gt_temp[2] = -gt_temp[2]
        ground_truth_posit.append(gt_temp)
        ground_truth_color.append([0, 0, 0])
    print(ground_truth_posit)
    return ground_truth_posit, ground_truth_color
#製造 ground truth 的線
ground_truth_posit, ground_truth_color = ground_truth()
line_gt = o3d.geometry.LineSet()
#將 ground truth 的點存入 o3d.geometry.LineSet
line_gt.points = o3d.utility.Vector3dVector(ground_truth_posit)
#將 ground truth 的線存入 o3d.geometry.LineSet
line_gt.lines = o3d.utility.Vector2iVector(lines)
#將 ground truth 的顏色存入 o3d.geometry.LineSet
line_gt.colors = o3d.utility.Vector3dVector(ground_truth_color)
#將 ground truth, estimate 與先前疊圖的 pcd 一同呈現
o3d.visualization.draw_geometries([final_pcd, line_gt, line_set])
# o3d.io.write_point_cloud("./ground_truth.pcd", final_pcd)
print("ground_truth is done!!!!")
```

最後是 L1 distance 的呈現，首先藉由自製的 err_count()函式將各個點 ground truth 與 estimate 的 x, y, z 相減，接著讓他們平方開根號以實現 L1 distance。詳細將在以下 code 表示:

```
# 計算 ground truth 與 estimate 的 L2 distance
def error_count():
    err = 0
    for i in range(count):
        # 使第 i 個點 ground truth 與 estimate 的 x, y, z 相減
        error_temp = [ground_truth_posit[i][k] - camera_points[i][k]
                      for k in range(3)]
        #將 x, y, z 差的值做平方開更號
        err = err + math.sqrt(error_temp[0] **
```

```
                            2 + error_temp[1]**2 + error_temp[2]**2)
    err = err / count
    return err
# 計算 ground truth 與 estimate 的 L2 distance
L2_distance = error_count()
print("The L2 distance between estimated camera poses and groundtruth camera poses
is :{}".format(L2_distance))
```

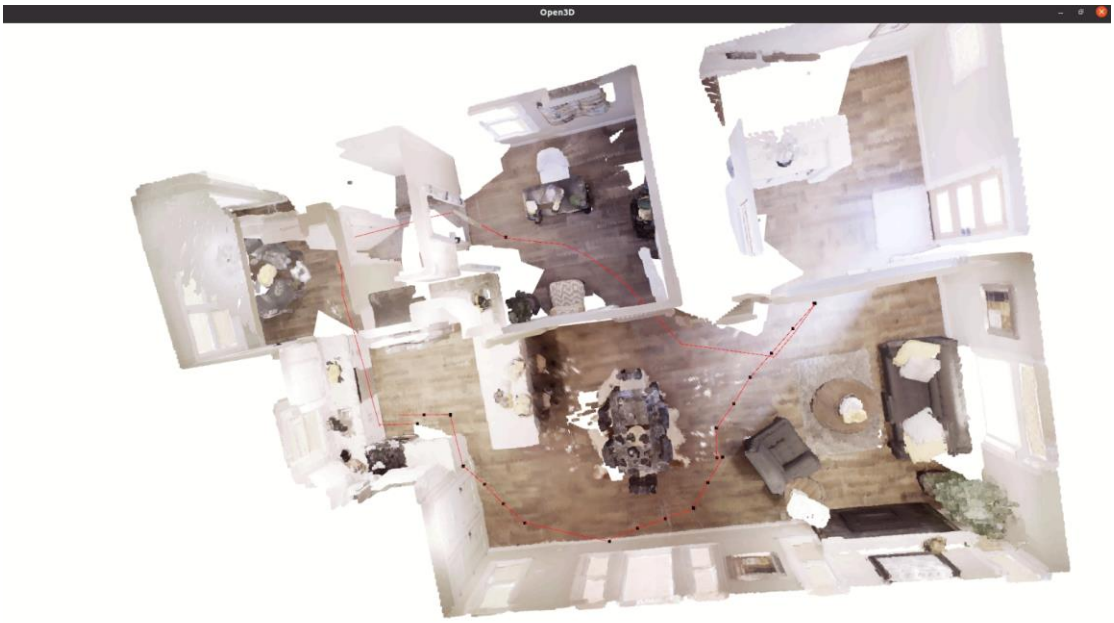
ii. Result and Discussion

Reesult:

一樓:

重建圖:

Estimation:



Ground truth:

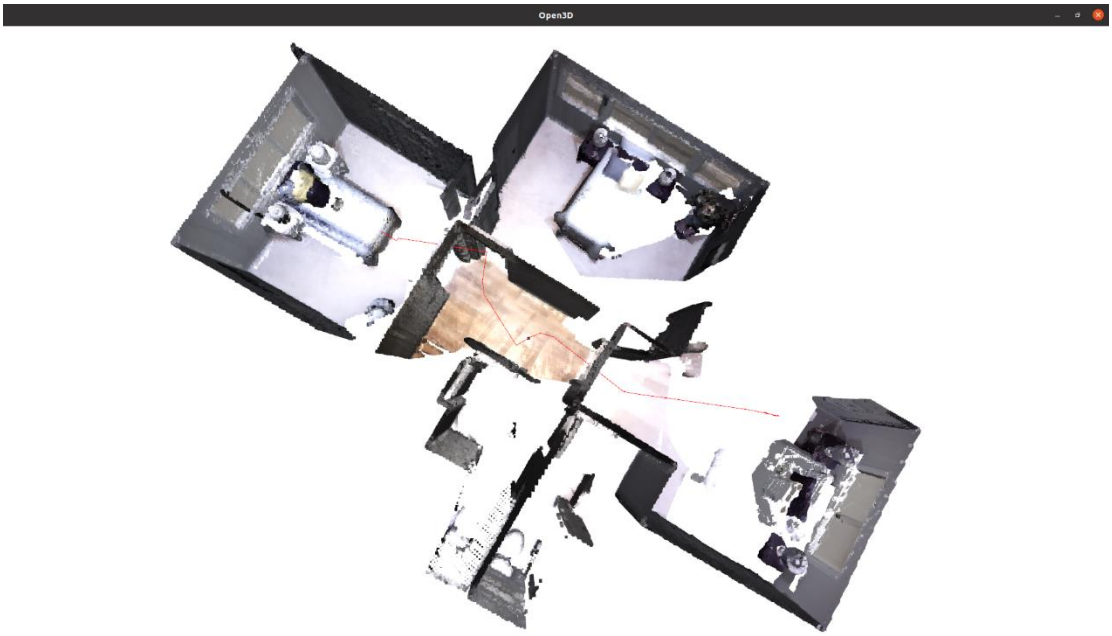
L1 distance:

```
The L2 distance between estimated camera poses and groundtruth camera poses is
:0.3384324190121891
(habitat) widden@widden-desktop:~/NCTU/Perception_and_Decision_Making/hw1$ pyth
```

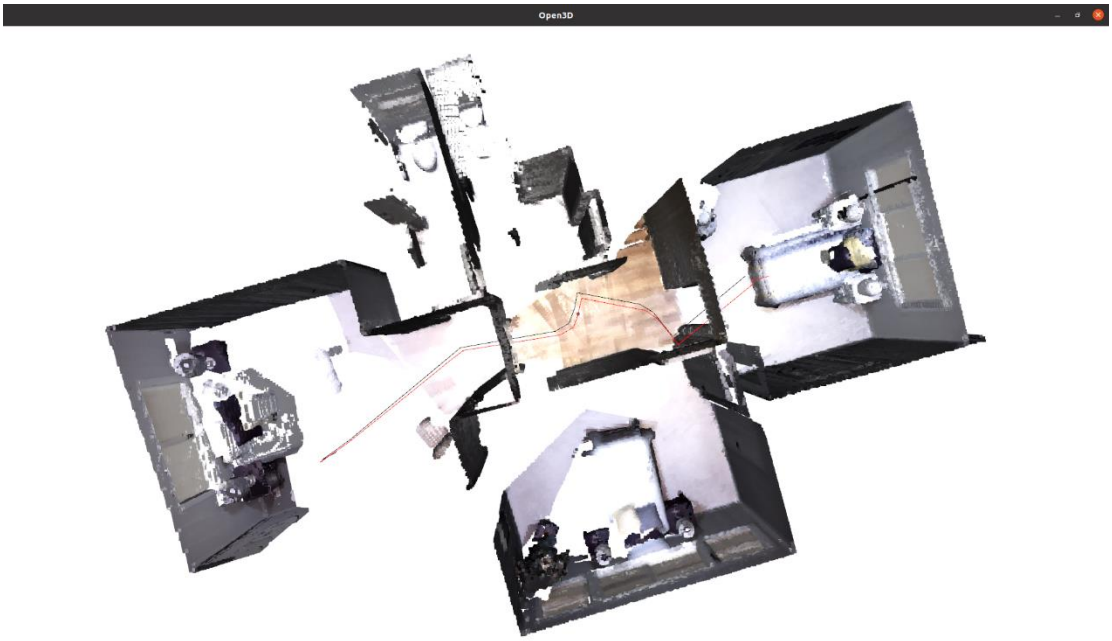
二樓:

重建圖

Estimate:



Ground truth:



L1 distance:

```
The L2 distance between estimated camera poses and groundtruth camera poses is
:0.10309426904603942
(habitat) widden@widden-desktop:~/NCTU/Perception_and_Decision_Making/hw1$
```

Discussion:

在進行二樓圖的重建時，若取樣點過多、太過深入某間廁所或是原地自轉太多次，重建就會失敗(疊的很奇怪)或是電腦當機，我認為是因為二樓的空間太小了，它的 feature mapping 會計算太多次才能成功疊出來抑或是疊失敗。也是因為上述原因，在做二樓重建時，我盡可能不做過多轉彎與深入房間，因此，得到的重建圖才會不完整。

另外，在 L1 distance 的部分一二樓都差不多，從上面的結果圖來看也是如此，estimation 與 ground truth 的路徑幾乎疊在一起，可見 open3d 疊圖的效果非常好。

最後，在做自己的 icp 時我瘋狂出錯，疊出來的圖看起來超級醜，L1 distance 也差超級多，希望助教能夠公布如何實做出 icp，讓我能看看究竟是哪邊出錯了。另外，也謝謝助教在 teams 上幾乎秒回的速度，讓我在做功課時不會卡太久。