智慧感知與決策

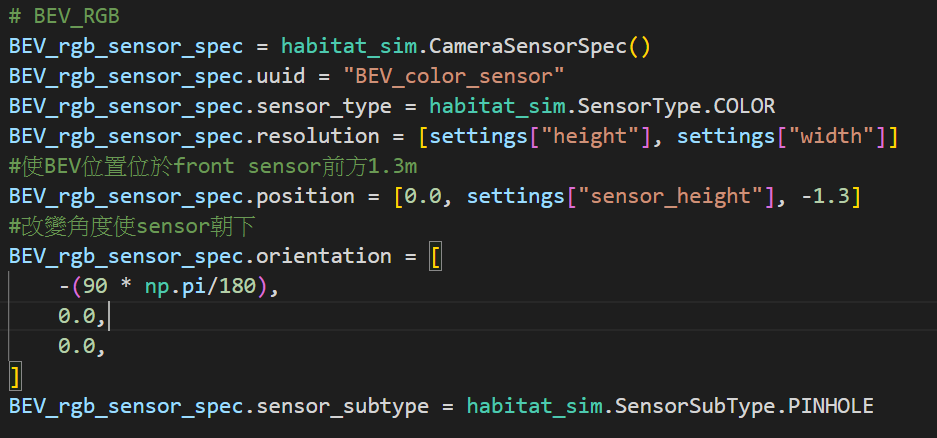
B073012003鄧書桓

**Task1:**

1. **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:



一張含有 文字, 螢幕, 螢幕擷取畫面, 銀色 的圖片

自動產生的描述

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

一張含有 文字 的圖片

自動產生的描述

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]

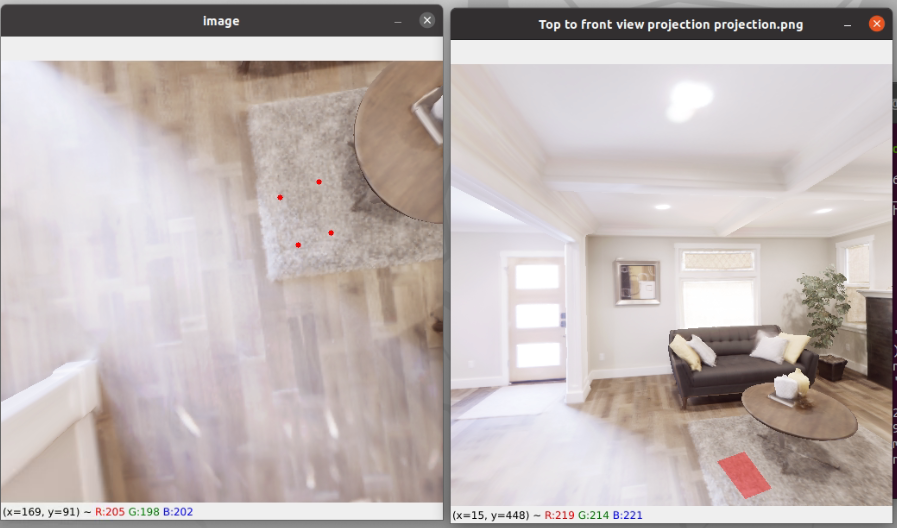
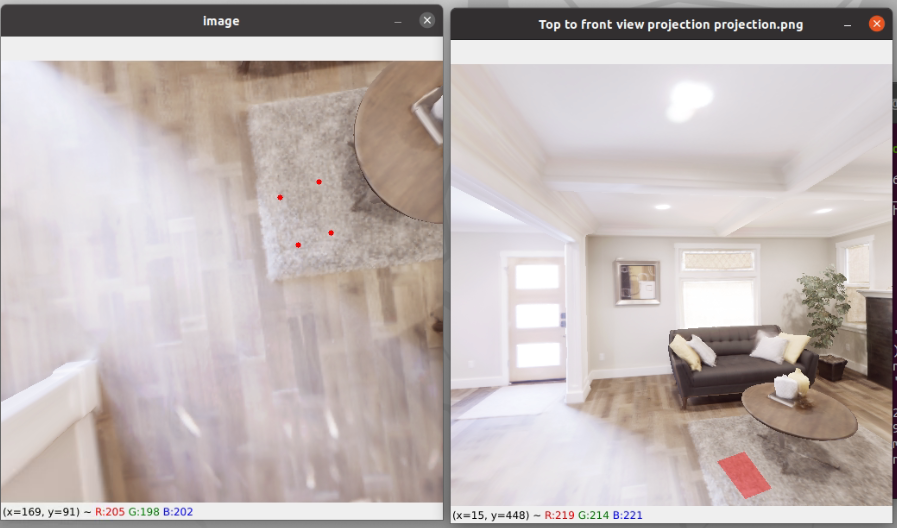
        new\_pixels = (np.array(new\_pixels)+256).tolist()

        print("points in front pixel view (matrix) are : ", new\_pixels)

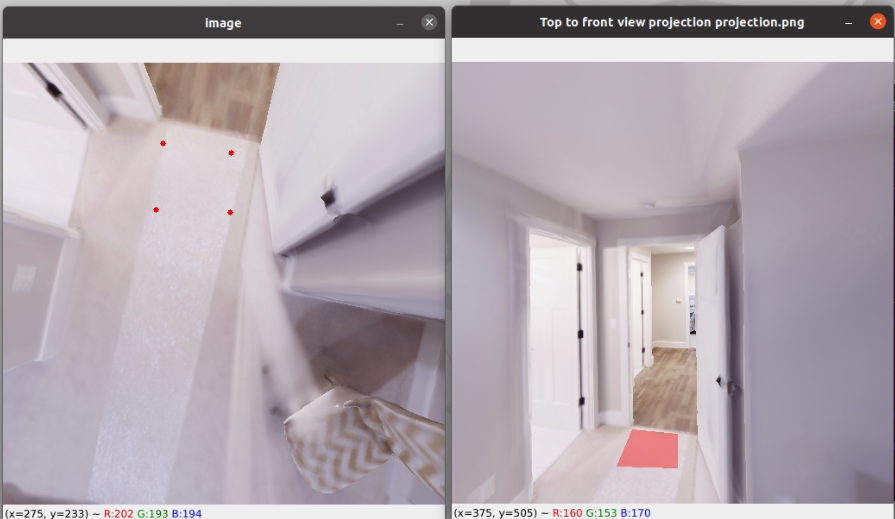
        return new\_pixels

1. **Result and Discussion**

Floor1:

Floor2:

 一張含有 文字, 室內, 牆, 家電用品 的圖片

自動產生的描述

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:**

1. **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()

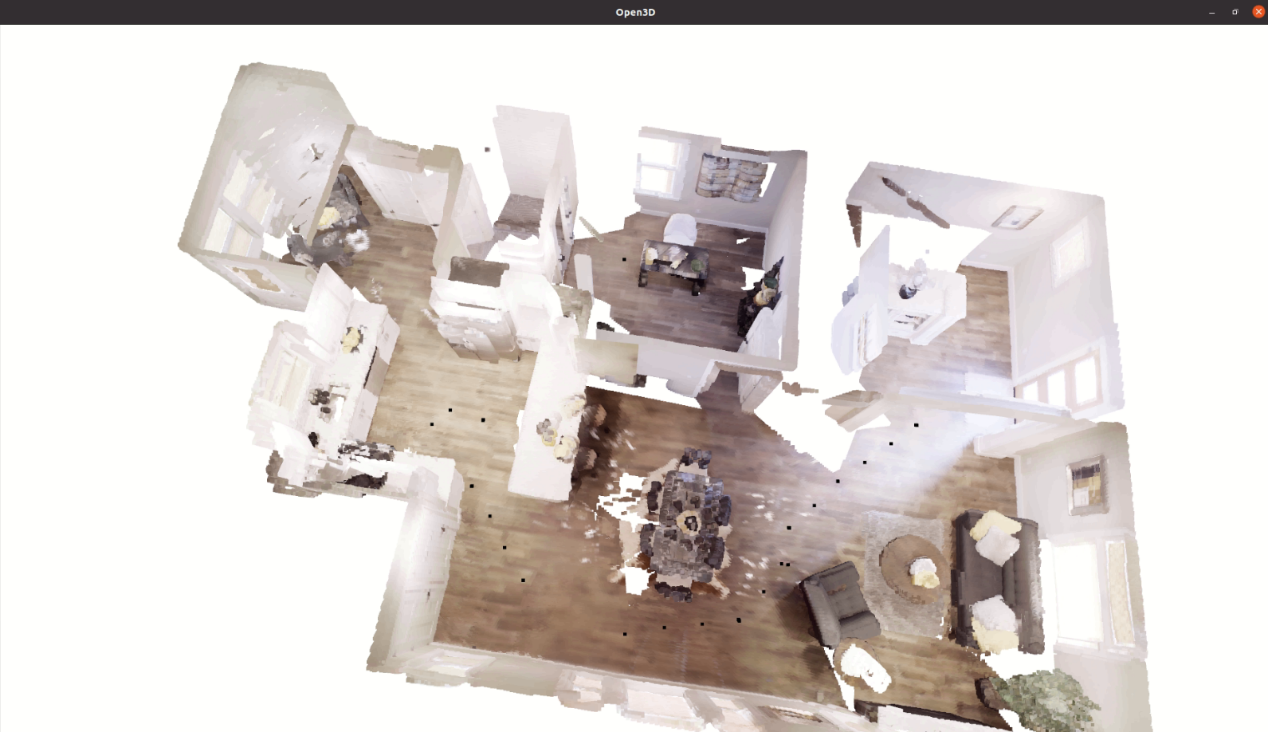
print("The L2 distance between estimated camera poses and groundtruth camera poses is :{}".format(L2\_distance))

**ii. Result and Discussion**

Reesult:

一樓:

重建圖:



Estimation:



Ground truth:



L1 distance:

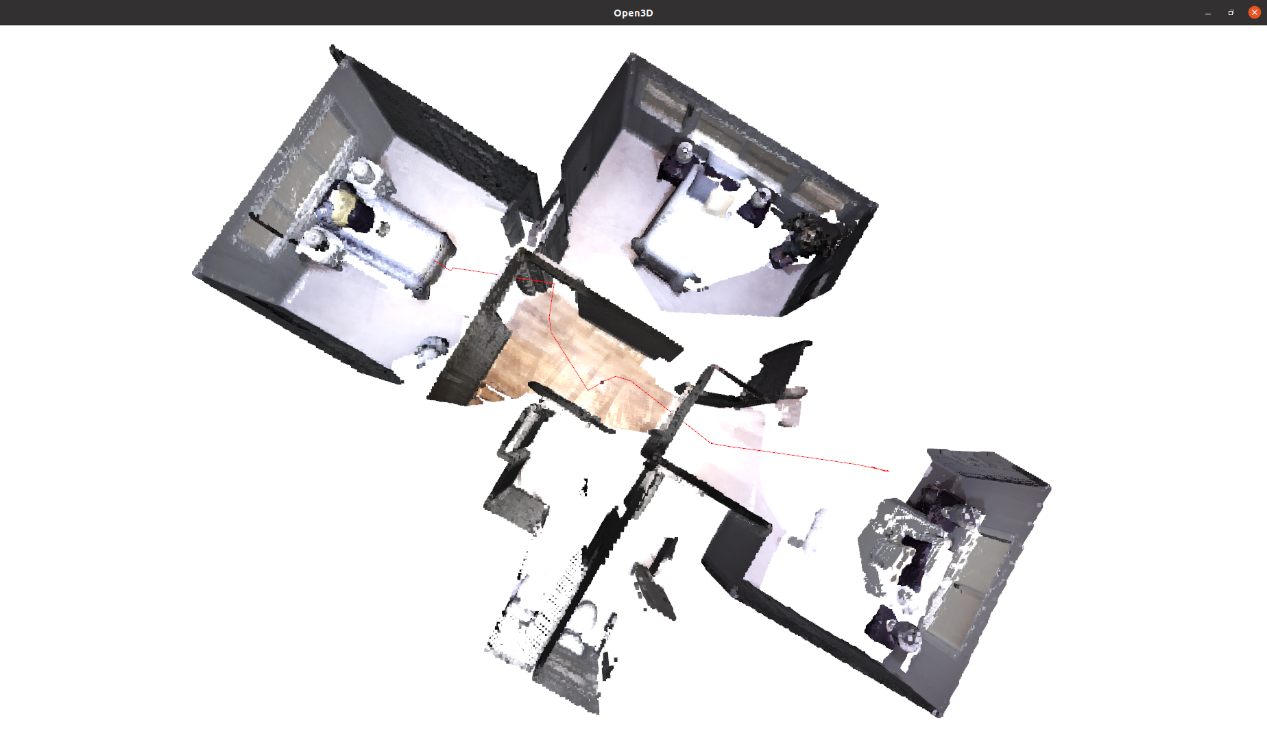


二樓:

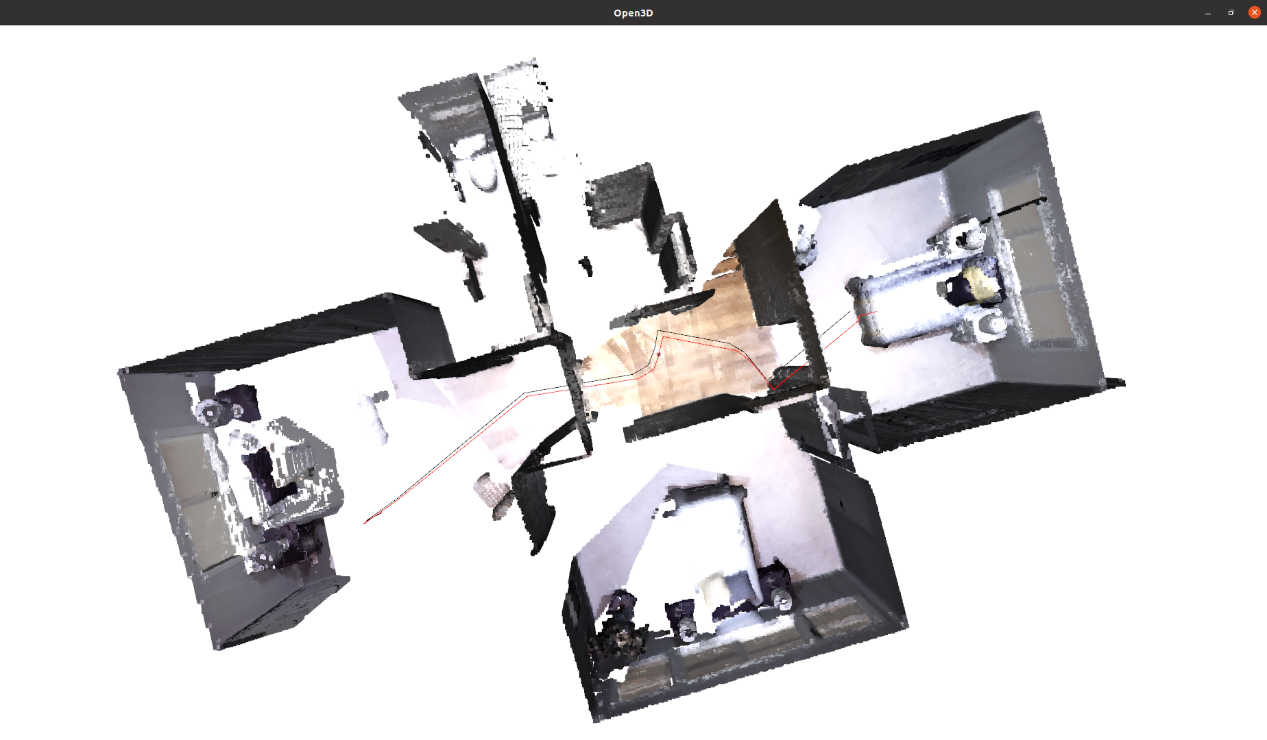
重建圖



Estimate:



Ground truth:



L1 distance:



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。