Perception and Decision Making in Intelligent Systems
A Semantic Mapping Framework

I. Introduction

In this work, a semantic segmentation model is trained to predict the labels of all object in the environment, and reconstruct the 3D semantic map of the first and second floor scene in the apartment_0 environment of Replica dataset. The focus revolves around the integration of point cloud data, ground truth information, and advanced techniques to improve the quality and accuracy of 3D mapping. A pivotal component of my approach is a custom voxel down-sampling algorithm designed to optimize the processing of point clouds. This work stems from the need to overcome domain shift challenges, improve performance, and enhance the accuracy of 3D models derived from diverse datasets.

II. Semantic Model Training

Code and Configuration:

Prior to the training phase, execute the "data_generator.py" script to direct the Habitat agent in the collection of essential data. This data collection process involves obtaining RGB, depth, and semantic images from the Replica dataset. Two distinct datasets are prepared for this purpose. One dataset comprises data from 10 different scenes within the Replica dataset, offering diversity and broader coverage. In contrast, the second dataset exclusively focuses on a single scene, specifically Apartment 0. This data collection step lays the foundation for subsequent training and experimentation in the work. Then create the .odgt files for training process.

In this work, the hardware configuration is: (1) CPU : Intel 8700 (2) GPU : Nvidia GeForce GTX 1060 6GB. The training environments are (1) pytorch version 1.13.0 (2) Cuda 11.7 (3) Cudnn 8.6.0. The first dataset, encompassing a diverse range of scenes (10 in total), consists of a total of 13,000 training images. In contrast, the second dataset focuses solely on Apartment 0 and contains 1,300 training images. This dataset offers a more concentrated exploration of a single scene.

The training configuration is shown as below:

(1) model_apartment_0 :

```
DATASET:
  imgMaxSize: 525
  imgSizes: (300, 375, 450, 525, 600)
  list_train: ./data/dataset_2_training.odgt
```

```yaml
  # list_val: ./data/dataset_2_validation.odgt
  list_val: ./data/second_floor_validation.odgt
  num_class: 101
  padding_constant: 32
  random_flip: True
  root_dataset: ""
  segm_downsampling_rate: 4
DIR: ckpt/model_apartment_0
MODEL:
  arch_decoder: c1
  arch_encoder: hrnetv2
  fc_dim: 720
  weights_decoder:
  weights_encoder:
TEST:
  batch_size: 1
  checkpoint: epoch_30.pth
  result: ./
TRAIN:
  batch_size_per_gpu: 4
  beta1: 0.9
  deep_sup_scale: 0.4
  disp_iter: 20
  epoch_iters: 320
  fix_bn: False
  lr_decoder: 0.02
  lr_encoder: 0.02
  lr_pow: 0.9
  num_epoch: 30
  optim: SGD
  seed: 304
  start_epoch: 0
  weight_decay: 0.0001
  workers: 16
VAL:
  batch_size: 1
  checkpoint: epoch_30.pth
  visualize: False
```

(2) model_others:

```
DATASET:
  imgMaxSize: 525
  imgSizes: (300, 375, 450, 525, 600)
  list_train: ./data/dataset_1_training.odgt
  list_val: ./data/second_floor_validation.odgt
  num_class: 101
  padding_constant: 32
  random_flip: True
  root_dataset: ""
  segm_downsampling_rate: 4
DIR: ckpt/model_others
MODEL:
  arch_decoder: c1
  arch_encoder: hrnetv2
  fc_dim: 720
  weights_decoder:
  weights_encoder:
TEST:
  batch_size: 1
  checkpoint: epoch_30.pth
  # epoch 19 is well
  result: ./
TRAIN:
  batch_size_per_gpu: 2
  beta1: 0.9
  deep_sup_scale: 0.4
  disp_iter: 20
  epoch_iters: 5000
  fix_bn: False
  lr_decoder: 0.02
  lr_encoder: 0.02
  lr_pow: 0.9
  num_epoch: 30
  optim: SGD
  seed: 304
  start_epoch: 0
  weight_decay: 0.0001
  workers: 16
VAL:
  batch_size: 1
  checkpoint: epoch_30.pth
  visualize: False
```

List_tran and List_val is set as the odgt file that contains the data we intent to use in both model. Since this work is using “color101.mat”, the num_class is set to 101. The images in this work are all 512 x 512, so the imgMaxSize parameter is set to 525.

Considering the constraints of our hardware, which utilized a single GPU, the batch size per GPU was established at 2, allowing for efficient memory utilization. Concurrently, we allocated 12 workers to expedite data loading. The training regimen extended across 30 epochs, as a more extensive number of epochs might risk overfitting.

As for the training process, the number of iterations was determined based on the number of training images within the dataset. In the dataset comprising 10 distinct scenes, a total of 13,000 training images were available. With a batch size of 2 images per GPU, a theoretical 6,500 iterations might have been anticipated. However, due to limitations in CUDA memory, the iteration count was set to 5,000, ensuring smooth training operations.

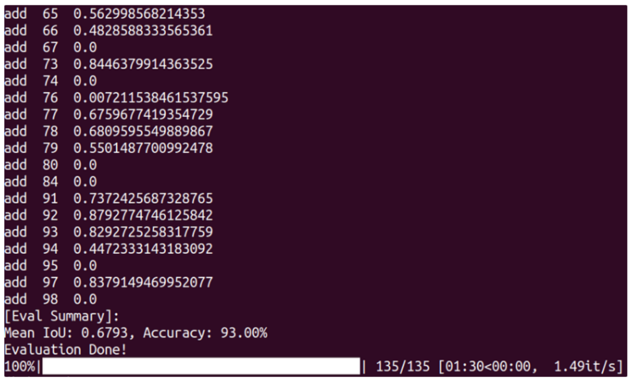
Conversely, when training exclusively on data collected from Apartment 0, which featured 1,300 images, the iteration count was adjusted to 650 iterations. These parameters were meticulously selected to balance performance and resource efficiency in our experiments.

Result and Discussion:

The conducted experiments involving Model_apartment_0 and Model_others revealed notable performance disparities that can be attributed to various factors, including the dataset collection and data diffusion processes. Both models were trained to perform scene segmentation tasks using data generated from the "apartment_0" scene. However, the stark contrast in their performance underscores the crucial role of training on a scene-specific dataset.

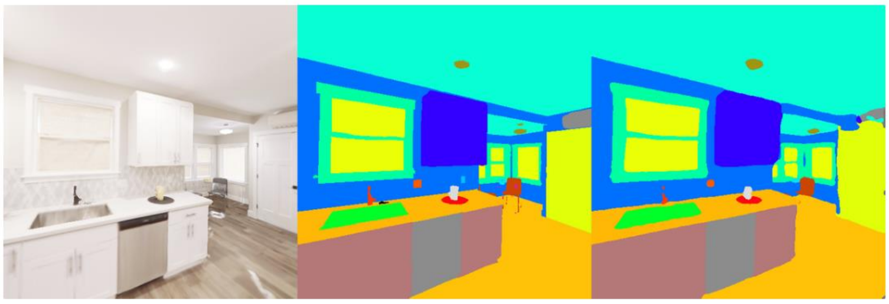
Model_apartment_0 Performance:

For model “ model_apartment_0 ”, the accuracy of the model converges to 99.65%, and the loss converges to 0.000769. For first floor, the evaluation data are mean IoU: 0.6793, and the predict accuracy is 93.00 %



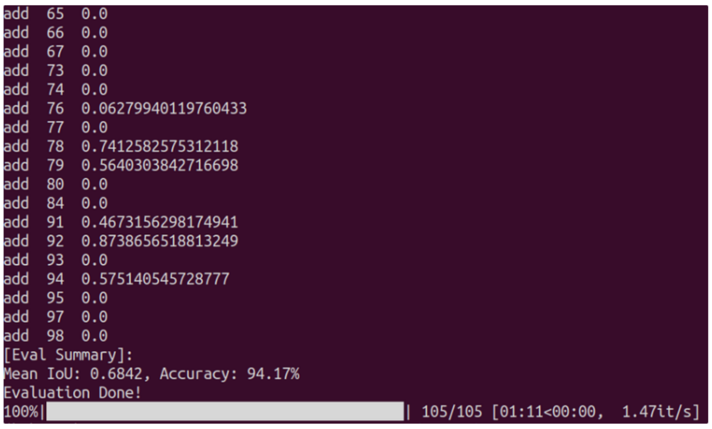
```
add  65  0.562998568214353
add  66  0.4828588333565361
add  67  0.0
add  73  0.8446379914363525
add  74  0.0
add  76  0.007211538461537595
add  77  0.6759677419354729
add  78  0.6809595549889867
add  79  0.5501487700992478
add  80  0.0
add  84  0.0
add  91  0.7372425687328765
add  92  0.8792774746125842
add  93  0.8292725258317759
add  94  0.4472333143183092
add  95  0.0
add  97  0.8379149469952077
add  98  0.0
[Eval Summary]:
Mean IoU: 0.6793, Accuracy: 93.00%
Evaluation Done!
100%|████████████████████████████████| 135/135 [01:30<00:00,  1.49it/s]
```

Fig(1). The terminal result of the evaluation



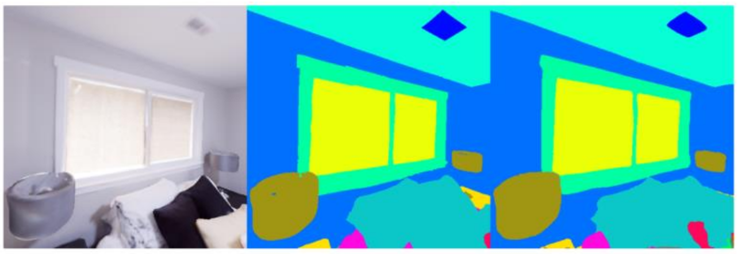
Fig(2). The result comparison of the model, from left to right are the RGB images, the ground truth segmentation image, and the model predicted image

As for the second floor, the evaluation data are mean IoU: 0.6482, and the predict accuracy is 94.17 %



```
add  65  0.0
add  66  0.0
add  67  0.0
add  73  0.0
add  74  0.0
add  76  0.06279940119760433
add  77  0.0
add  78  0.7412582575312118
add  79  0.5640303842716698
add  80  0.0
add  84  0.0
add  91  0.4673156298174941
add  92  0.8738656518813249
add  93  0.0
add  94  0.575140545728777
add  95  0.0
add  97  0.0
add  98  0.0
[Eval Summary]:
Mean IoU: 0.6842, Accuracy: 94.17%
Evaluation Done!
100%|████████████████████████████████| 105/105 [01:11<00:00,  1.47it/s]
```

Fig(3). The terminal result of the evaluation



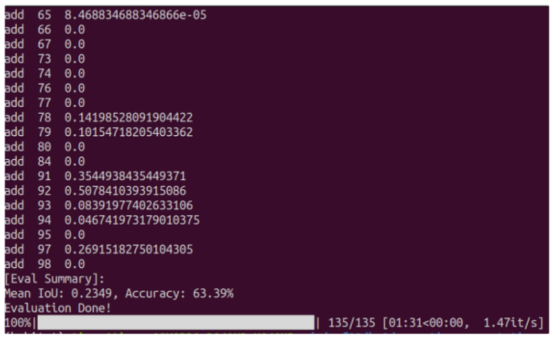
Fig(4). The result comparison of the model, from left to right are the RGB images, the ground truth segmentation image, and the model predicted image

Model_apartment_0 exhibited impressive results in its evaluation on both the first and second floor scenes. The model achieved a Mean IoU of 0.6793 and an accuracy of 93.00% for the first floor. Furthermore, for the second floor, it displayed remarkable performance with a Mean IoU of 0.6842 and an accuracy of 94.17%.

These outcomes indicate that Model_apartment_0 effectively learned and generalized the specific features and characteristics of the "apartment_0" scene, thereby showcasing a comprehensive understanding of both scenarios.

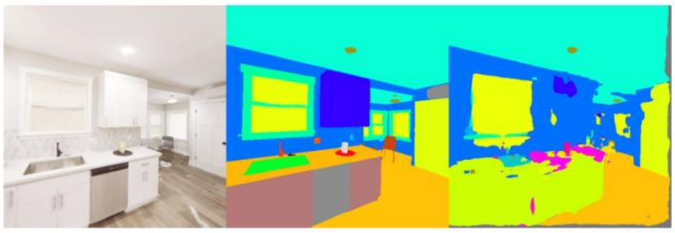
Model_others Performance:

For model " model_apartment_0 ", the accuracy of the model converges to 99.44%, and the loss converges to 0.00165. For first floor, the evaluation data are mean IoU: 0.2349, and the predict accuracy is 63.39 %



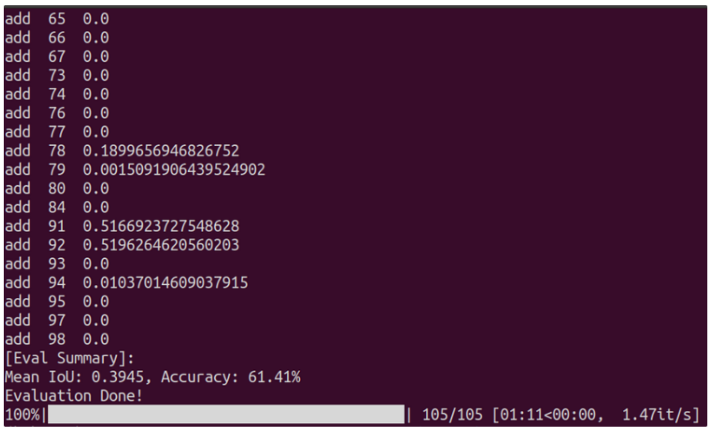
```
add  65  8.468834688346866e-05
add  66  0.0
add  67  0.0
add  73  0.0
add  74  0.0
add  76  0.0
add  77  0.0
add  78  0.14198528091904422
add  79  0.10154718205403362
add  80  0.0
add  84  0.0
add  91  0.3544938435449371
add  92  0.5078410393915086
add  93  0.08391977402633106
add  94  0.046741973179010375
add  95  0.0
add  97  0.26915182750104305
add  98  0.0
[Eval Summary]:
Mean IoU: 0.2349, Accuracy: 63.39%
Evaluation Done!
100%|████████████████████████████████| 135/135 [01:31<00:00,  1.47it/s]
```

Fig(5). The terminal result of the evaluation



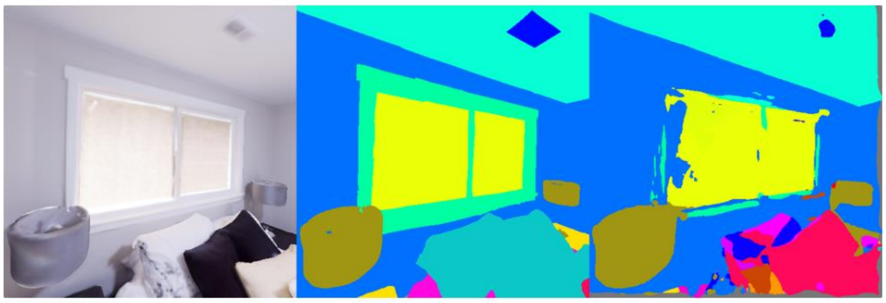
Fig(6). The result comparison of the model

As for the second floor, the evaluation data are mean IoU: 0.3945, and the predict accuracy is 61.41 %



```
add  65  0.0
add  66  0.0
add  67  0.0
add  73  0.0
add  74  0.0
add  76  0.0
add  77  0.0
add  78  0.1899656946826752
add  79  0.0015091906439524902
add  80  0.0
add  84  0.0
add  91  0.5166923727548628
add  92  0.5196264620560203
add  93  0.0
add  94  0.01037014609037915
add  95  0.0
add  97  0.0
add  98  0.0
[Eval Summary]:
Mean IoU: 0.3945, Accuracy: 61.41%
Evaluation Done!
100%|████████████████████████████████| 105/105 [01:11<00:00,  1.47it/s]
```

Fig(7). The terminal result of the evaluation



Fig(8). The result comparison of the model, from left to right are the RGB images, the ground truth segmentation image, and the model predicted image

In contrast, Model_others, trained on data diffused across multiple scenes, did not demonstrate comparable performance. The model exhibited a remarkable convergence of accuracy to 99.44% and a low loss of 0.00165 during training. However, when evaluated on the first floor and second floor scenes, its Mean IoU and accuracy metrics fell considerably short. For the first floor, it achieved a Mean IoU of 0.2349 and an accuracy of 63.39%, and Mean IoU of 0.3945 and accuracy of 61.41% for second floor.

III. 3D Semantic Map Reconstruction

Code:

In this work, the goal is to reconstruction a 3D Semantic Map, using the code from the last work as basics. The difference compare to last work is the voxelize process.

Custom Voxelize function:

In semantic segmentation, the semantic point cloud’s voxelize should be modified due to each voxel could contain more than one labels. With the open3d’s function, the color is averaged. To avoid that, implement a custom voxelize function to deal with this issue. The function code is shown as below:

```python
def custom_voxel_down(pcd, voxel_size):
    xyz_points = np.asarray(pcd.points)
    xyz_colors = np.asarray(pcd.colors)

    # determine each point is belongs to what voxel
    voxel_indices = ((xyz_points / voxel_size) - (xyz_points.min(0) /
voxel_size)).astype(int)

    # create voxel point cloud
    voxel_point_cloud = {}
    voxel_colors = {}
    for i in range(len(xyz_points)):
        voxel_index = tuple(voxel_indices[i])
        if voxel_index not in voxel_point_cloud:
            voxel_point_cloud[voxel_index] = []
            voxel_colors[voxel_index] = []
        voxel_point_cloud[voxel_index].append(xyz_points[i])
        voxel_colors[voxel_index].append(tuple(xyz_colors[i]))

    downsampled_points = []
    downsampled_colors = []
    for voxel_index, points in voxel_point_cloud.items():
        colors = voxel_colors[voxel_index]
        major_color = max(set(colors), key=colors.count)
        downsampled_points.append(np.mean(points, axis=0))
        downsampled_colors.append(np.array(major_color))

    downsampled_pcd = o3d.geometry.PointCloud()
    downsampled_pcd.points =
o3d.utility.Vector3dVector(downsampled_points)
    downsampled_pcd.colors =
o3d.utility.Vector3dVector(downsampled_colors)
```

```
    # o3d.visualization.draw_geometries([downsampled_pcd])
    return downsampled_pcd
```

This function is an implementation of point cloud voxel down-sampling. Its primary purpose is to down-sample the input point cloud data based on a specified voxel size, reducing the number of points while retaining sufficient detail for further processing:

(1) Extract the point positions and colors from the input point cloud data pcd into two separate NumPy arrays, xyz_points and xyz_colors.
(2) Calculate which voxel each point belongs to. This is achieved by dividing the position of each point by the voxel size and subtracting the minimum position, then converting it to an integer. This determines which voxel each point falls into.
(3) Create a dictionary voxel_point_cloud to store the point cloud data for each voxel, and simultaneously create a dictionary voxel_colors to store the corresponding color information. For each point, it is added to the voxel_point_cloud of the respective voxel while retaining its color information
(4) For each voxel, select the dominant color (the color that appears most frequently) as the representative color. This is achieved by counting the colors within each voxel.
(5) Create a new point cloud downsampled_pcd, adding the representative positions of each voxel to it, while coloring each point with the representative color.
(6) Return the downsampled downsampled_pcd.

In summary, this function transforms high-density point cloud data into a sparse voxel representation and simplifies and compresses the data using representative colors. It can help reduce computational burden and improve efficiency, while preserving enough information for further processing, making it useful for handling large point cloud datasets.

3d_semantic_map.py:

For the reconstruct code, the only difference from last work is the custom voxel down function, the remain are still the same. That is, in preprocess_point_cloud(pcd, voxel_size): Using the implemented voxelize function to down sampled point cloud.

```
import numpy as np
import open3d as o3d
import argparse
```

```python
import os
import copy
import time
import math
from sklearn.neighbors import NearestNeighbors
import cv2

def depth_image_to_point_cloud(rgb, depth):

    # Camera instrinsics
    principal_point = [256, 256]
    focal_length = np.tan(np.deg2rad(90/2)) * 256

    # Read the rgb images and the depth images
    image = np.asarray(o3d.io.read_image(rgb))
    depth = np.asarray(o3d.io.read_image(depth))
    rgb_colors = (image / 255).reshape(-1, 3)
    pixel_coords = np.indices((512, 512)).reshape(2, -1)
    d = depth.reshape(-1) / 1000 # real depth unit is m
    f = focal_length

    x = (principal_point[0] - pixel_coords[1]) * d / f
    y = (principal_point[1] - pixel_coords[0]) * d / f
    z = d

    xyz_points = np.vstack([x, y, z]).T

    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(xyz_points)
    pcd.colors = o3d.utility.Vector3dVector(rgb_colors)

    #calculate camera point
    camera_pcd = o3d.geometry.PointCloud()
    camera_pcd.points.append([0,0,0])
    camera_pcd.colors.append([1,0,0]) # set the color to red

    return pcd, camera_pcd
```

```python
def custom_voxel_down(pcd, voxel_size):
    xyz_points = np.asarray(pcd.points)
    xyz_colors = np.asarray(pcd.colors)

    # determine each point is belongs to what voxel
    voxel_indices = ((xyz_points / voxel_size) - (xyz_points.min(0) /
voxel_size)).astype(int)

    # create voxel point cloud
    voxel_point_cloud = {}
    voxel_colors = {}
    for i in range(len(xyz_points)):
        voxel_index = tuple(voxel_indices[i])
        if voxel_index not in voxel_point_cloud:
            voxel_point_cloud[voxel_index] = []
            voxel_colors[voxel_index] = []
        voxel_point_cloud[voxel_index].append(xyz_points[i])
        voxel_colors[voxel_index].append(tuple(xyz_colors[i]))

    downsampled_points = []
    downsampled_colors = []
    for voxel_index, points in voxel_point_cloud.items():
        colors = voxel_colors[voxel_index]
        major_color = max(set(colors), key=colors.count)
        downsampled_points.append(np.mean(points, axis=0))
        downsampled_colors.append(np.array(major_color))

    downsampled_pcd = o3d.geometry.PointCloud()
    downsampled_pcd.points =
o3d.utility.Vector3dVector(downsampled_points)
    downsampled_pcd.colors =
o3d.utility.Vector3dVector(downsampled_colors)

    # o3d.visualization.draw_geometries([downsampled_pcd])
    return downsampled_pcd

def preprocess_point_cloud(pcd, voxel_size):
    pcd_down = pcd.voxel_down_sample(voxel_size=voxel_size)
    # pcd_down = custom_voxel_down(pcd,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

def draw_registration_result(source, target, transformation):
    source_temp = copy.deepcopy(source)
    target_temp = copy.deepcopy(target)
    source_temp.paint_uniform_color([1, 0.706, 0])
    target_temp.paint_uniform_color([0, 0.651, 0.929])
    source_temp.transform(transformation)
    o3d.visualization.draw_geometries([source_temp, target_temp])

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_matchin
g(
        source_down, target_down, source_fpfh, target_fpfh, True,
        distance_threshold,
        o3d.pipelines.registration.TransformationEstimationPointToPoint(
False),
        3, [
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnEdgeL
ength(
                0.9),
            o3d.pipelines.registration.CorrespondenceCheckerBasedOnDista
nce(
                distance_threshold)
        ], o3d.pipelines.registration.RANSACConvergenceCriteria(100000,
0.999))
    return result

def execute_fast_global_registration(source_down, target_down,
source_fpfh,
                                     target_fpfh, voxel_size):
    distance_threshold = voxel_size * 0.5
    result =
o3d.pipelines.registration.registration_fgr_based_on_feature_matching(
        source_down, target_down, source_fpfh, target_fpfh,
        o3d.pipelines.registration.FastGlobalRegistrationOption(
            maximum_correspondence_distance=distance_threshold))
    return result

def local_icp_algorithm(source_down, target_down, trans_init,
voxel_size):
    distance_threshold = voxel_size * 0.4
    result = o3d.pipelines.registration.registration_icp(
        source_down, target_down, distance_threshold, trans_init,
        o3d.pipelines.registration.TransformationEstimationPointToPlane(
))
    return result

'''
Start Implementing tht local_icp_algorithm
'''


def best_fit_transform(source_points, target_points):

    # Calculates the least-squares best-fit transform between
corresponding 3D points source -> target
    assert np.shape(source_points) == np.shape(target_points)

    # number of dimensions
    m = np.shape(source_points)[1]

    # translate points to their centroids
    centroid_A = np.mean(source_points, axis=0)
    centroid_B = np.mean(target_points, axis=0)
    AA = source_points - centroid_A
    BB = target_points - centroid_B

    # rotation matrix
    W = np.dot(BB.T, AA)
    U, S, VT = np.linalg.svd(W)
    rotation = np.dot(U, VT)

    # special reflection case
    if np.linalg.det(rotation) < 0:
        VT[m-1,:] *= -1
        rotation = np.dot(U, VT)

    # translation
    translation = centroid_B.T - np.dot(rotation, centroid_A.T)

    # homogeneous transformation
    transformation = np.identity(m+1)
    transformation[:m, :m],transformation[:m, m] = rotation,translation

    return transformation, rotation, translation

def nearest_neighbor(source_points, target_points):

    # Find the nearest (Euclidean) neighbor in target points for each
point in source points
    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(target_points)

    # Euclidean distance (errors) of the nearest neighbor and the
nearest neighbor
    distances, indices = neigh.kneighbors(source_points,
return_distance=True)
```

```
    valid = distances < np.median(distances)*0.8
    return distances[valid].ravel(), 
indices[valid].ravel(),valid.ravel()

def implemented_local_icp_algorithm(source_down, target_down, 
trans_init=None,  max_iterations=100000 ,tolerance=0.000005):
    # the user may tuns the parameter

    source_points = np.asarray(source_down.points)
    target_points = np.asarray(target_down.points)
    m = np.shape(source_points)[1]

    # make points homogeneous, copy them so as to maintain the originals
    src = np.ones((m+1,source_points.shape[0]))
    dst = np.ones((m+1,target_points.shape[0]))
    src[:m,:] = np.copy(source_points.T)
    dst[:m,:] = np.copy(target_points.T)

    # apply the initial pose estimation
    if trans_init is not None:
        src = np.dot(trans_init, src)
    prev_error = 0

    # main part
    for i in range(max_iterations):
        # find the nearest neighbours between the current source and 
destination points
        distances, indices, valid = nearest_neighbor(src[0:m,:].T, 
dst[0:m,:].T)

        # compute the transformation between the current source and 
nearest destination points
        transformation,_,_ = best_fit_transform(src[0:m,valid].T, 
dst[0:m,indices].T)

        # update the current source
        src = np.dot(transformation, src)
```

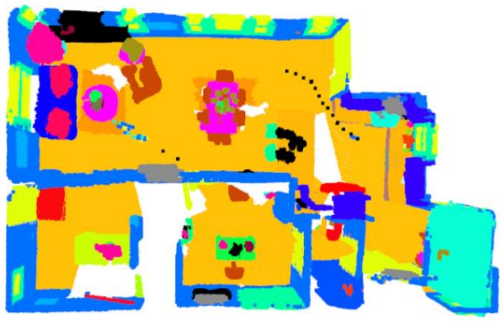
```python
        # check error
        mean_error = np.sum(distances) / distances.size
        if abs(prev_error-mean_error) < tolerance:
            break
        prev_error = mean_error

    # calculcate final tranformation
    transformation,_,_ = best_fit_transform(source_points, src[0:m,:].T)

    return transformation

def read_pose(args):

    # setting the file path
    file_path = args.data_root + "/GT_pose.npy"
    poses = np.load(file_path)
    print(len(poses))

    # transforming to real points with unit:m
    if args.floor == 1 :
        # mm to m / rw => 10 / 0.25
        x = -poses[:, 0]/ 40
        y = poses[:, 1]/ 40
        z = -poses[:, 2]/ 40
    elif args.floor == 2:
        # 10 / rw => 10 / 0.25
        x = -poses[:, 0] / 40 -0.00582
        y = ( poses[:, 1] / 40 ) - 0.07313
        z = -poses[:, 2]/ 40 -0.03

    xyz_points = np.vstack([x,y,z]).T

    gt_pose_pcd = o3d.geometry.PointCloud()
    gt_pose_pcd.points = o3d.utility.Vector3dVector(xyz_points)
    gt_pose_pcd.paint_uniform_color([0,0,0]) #set color to black

    gt_lines = []
    for i in range(len(xyz_points) - 1):
        gt_lines.append([i, i + 1])

    gt_line_set = o3d.geometry.LineSet()
    gt_line_set.points = o3d.utility.Vector3dVector(xyz_points)
    gt_line_set.lines = o3d.utility.Vector2iVector(gt_lines)

    return gt_pose_pcd, gt_line_set

def reconstruct(args):

    # config
    voxel_size = 0.00225
    point_cloud = o3d.geometry.PointCloud()
    estimate_camera_cloud = o3d.geometry.PointCloud()
    data_folder_path = args.data_root
    rgb_images = os.listdir(os.path.join(data_folder_path,
"result_model_3/"))
    # rgb_images = os.listdir(os.path.join(data_folder_path,
"gt_result/"))
    depth_images = os.listdir(os.path.join(data_folder_path, "depth/"))
    if args.floor == 1:
        print("Start reconstructing the first floor...")
    if args.floor == 2:
        print("Start reconstructing the second floor...")
    reconstruct_start = time.time()
    print("Numbers of images is %d" % len(rgb_images))

    # temps
    pcd = []
    camera_pcd = []
    pcd_down = []
    pcd_transformed = [] # contain the pcd transformed to the main axis
    fpfh = []

    for i in range(1,len(rgb_images)):
    # for i in range(1,10):
        if i == 1:
            # target point cloud
            rgb_principal = data_folder_path + "/result_model_3/%d.png"
% i
            # rgb_principal = data_folder_path + "/gt_result/%d.png" % i
            depth_principal = data_folder_path + "/depth/%d.png" % i
            print("Principal picture is set as picture %d." % i)
            pcd.append(depth_image_to_point_cloud(rgb_principal,
depth_principal)[0]) # pcd[i-1]
            camera_pcd.append(depth_image_to_point_cloud(rgb_principal,
depth_principal)[1])
            pcd_down.append(preprocess_point_cloud(pcd[i-
1],voxel_size)[0]) # pcd_down[i-1]
            fpfh.append(preprocess_point_cloud(pcd[i-1],voxel_size)[1])
# fpfh[i-1]
            pcd_transformed.append(pcd_down[i-1])
        else:
            # get source point cloud
            rgb= data_folder_path + "/result_model_3/%d.png" % i
            # rgb= data_folder_path + "/gt_result/%d.png" % i
            depth = data_folder_path + "/depth/%d.png" % i
            print("------------------------------------")
            print("dealing with picture %d..." % i)
            pcd.append(depth_image_to_point_cloud(rgb, depth)[0]) #
pcd[i-1]
            camera_pcd.append(depth_image_to_point_cloud(rgb, depth)[1])
            pcd_down.append(preprocess_point_cloud(pcd[i-
1],voxel_size)[0]) # pcd_down[i-1]
            fpfh.append(preprocess_point_cloud(pcd[i-1],voxel_size)[1])
# target_fpfh[i-1]

            # Global registeration
            global_start = time.time()
            result_ransac = execute_fast_global_registration(pcd_down[i-
1], pcd_transformed[i-2],
                                                          fpfh[i-1],
fpfh[i-2], voxel_size)
            print("Global registeration took %.3f sec." % (time.time() -
global_start))
```

```
            # ICP
            icp_start = time.time()
            if args.version == 'open3d':
                print("Using open3d's icp...")
                result_icp = local_icp_algorithm(pcd_down[i-1],
pcd_transformed[i-2],
                                                result_ransac.transformat
ion, voxel_size)
                transformation = result_icp.transformation #
transformation of i to i-1
            elif args.version == 'my_icp':
                print("Using the implemented icp...")
                result_icp = implemented_local_icp_algorithm(pcd_down[i-
1], pcd_transformed[i-2],
                                                result_ransac.transformat
ion)
                transformation = result_icp # transformation of i to i-1
                # draw_registration_result(pcd_down[i-
1],pcd_transformed[i-2],transformation)
            print("ICP took %.3f sec.\n" % (time.time() - icp_start))

            # transformation to 1st camera axis
            # transformation = result_icp.transformation #
transformation of i to i-1
            pcd_transformed.append(pcd_down[i-
1].transform(transformation))
            camera_pcd[i-1] = camera_pcd[i-1].transform(transformation)

    for pcd in pcd_transformed:
        point_cloud += pcd

    for pcd in camera_pcd:
        estimate_camera_cloud += pcd

    estimate_lines = []
    for i in range(len(estimate_camera_cloud.points) - 1):
        estimate_lines.append([i, i + 1])
    estimate_line_set = o3d.geometry.LineSet()
```

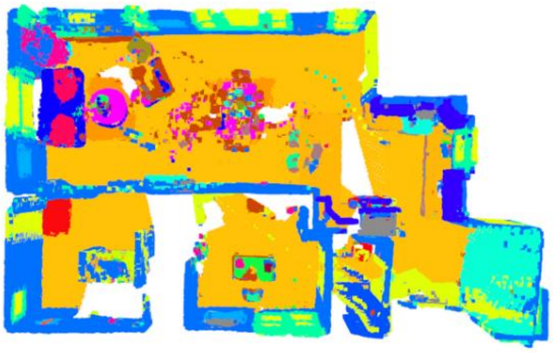
```python
    estimate_line_set.points =
o3d.utility.Vector3dVector(estimate_camera_cloud.points)
    estimate_line_set.lines = o3d.utility.Vector2iVector(estimate_lines)
    estimate_line_set.paint_uniform_color([1, 0, 0])

    # filter the ceiling
    xyz_points = np.asarray(point_cloud.points)
    colors = np.asarray(point_cloud.colors)
    if args.floor == 1:
        threshold_y = 0.0135
    elif args.floor == 2:
        if args.version == 'open3d':
            threshold_y = 0.0115
        elif args.version == 'my_icp':
            threshold_y = 0.009
    filtered_xyz_points = xyz_points[xyz_points[:, 1] <= threshold_y]
    filtered_colors = colors[xyz_points[:, 1] <= threshold_y]
    point_cloud.points = o3d.utility.Vector3dVector(filtered_xyz_points)
    point_cloud.colors = o3d.utility.Vector3dVector(filtered_colors)

    gt_pose_cloud, gt_line_set = read_pose(args)
    print("------------------------------------")
    print("3D reconstruction took %.3f sec." % (time.time() -
reconstruct_start))
    return point_cloud, gt_pose_cloud, gt_line_set,
estimate_camera_cloud, estimate_line_set
    # return point_cloud, estimate_camera_cloud, estimate_line_set

def calculate_mean_l2_distance(gt_pos_pcd, estimate_camera_cloud):
    sum = 0
    for i in range(len(estimate_camera_cloud.points)):
        x = gt_pos_pcd.points[i][0]-estimate_camera_cloud.points[i][0]
        y = gt_pos_pcd.points[i][1]-estimate_camera_cloud.points[i][1]
        z = gt_pos_pcd.points[i][2]-estimate_camera_cloud.points[i][2]
        sum += math.sqrt(x**2 + y**2 + z**2)
    return sum/len(estimate_camera_cloud.points)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-f', '--floor', type=int, default=1)
    parser.add_argument('-v', '--version', type=str, default='my_icp',
help='open3d')
    parser.add_argument('--data_root', type=str,
default='data_collection/first_floor/')
    args = parser.parse_args()

    if args.floor == 1:
        args.data_root = "data_collection/first_floor/"
    elif args.floor == 2:
        args.data_root = "data_collection/second_floor/"

    # Output result point cloud and estimated camera pose
    result_pcd, gt_pos_pcd, line_set, estimate_camera_cloud,
estimate_line_set= reconstruct(args)

    # Calculate and print L2 distance
    print("Mean L2 distance:", calculate_mean_l2_distance(gt_pos_pcd,
estimate_camera_cloud))

    # Visualize result
    #
o3d.visualization.draw_geometries([result_pcd,gt_pos_pcd,line_set,estim
ate_camera_cloud, estimate_line_set])
    o3d.visualization.draw_geometries([result_pcd])
    o3d.io.write_point_cloud('first_floor_model_3_2.pcd',result_pcd)
    # result = custom_voxel_down(result_pcd, 0.0023)
    # o3d.io.write_point_cloud('first_floor_model_3_2.pcd',result)
    print("3D reconstruction finished.")
```

Result and Discussion:

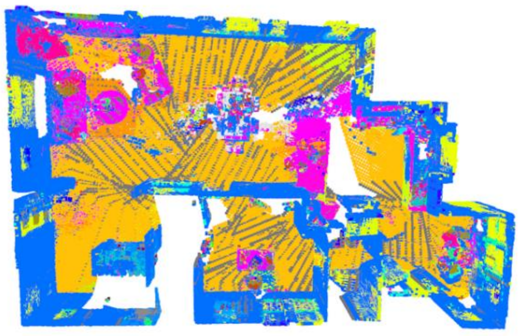
First Floor:



Fig(9). First floor ground truth semantic map

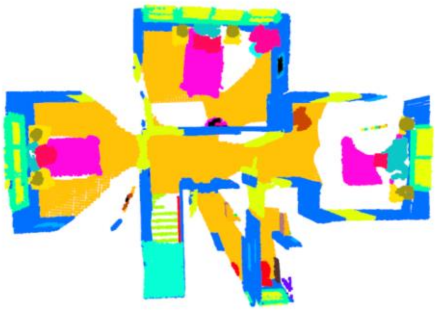


Fig(10). First floor semantic map train on model_apartment_0

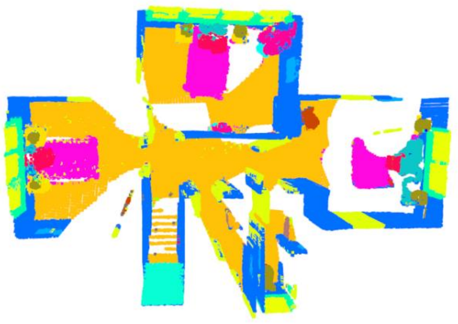


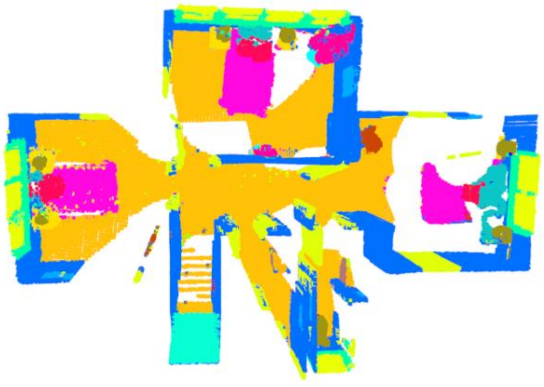
Fig(11). First floor semantic map train on model_others

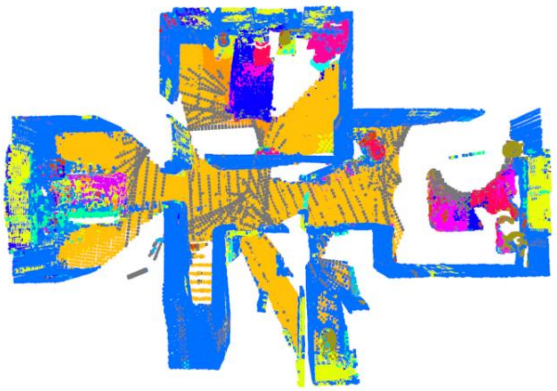
Second Floor:



Fig(12). First floor ground truth semantic map


Fig(13). Second floor ground truth semantic map


Fig(14). Second floor semantic map train on model_apartment_0


Fig(15). Second floor semantic map train on model_others

Discussion:

In the context of 3D mapping for the first and second floors, we employed two different models: "model_apartment_0" and "model_others." Our objective was to assess their performance and effectiveness in reconstructing 3D semantic maps.

The key differentiator between the two models lies in the specificity of the training dataset. "Model_apartment_0" was exclusively trained on the "apartment_0" scene, allowing it to excel in reconstructing semantic maps tailored to that environment. In contrast, "model_others" was exposed to a more diverse range of scenes, potentially causing it to struggle with capturing the unique characteristics of each floor.

One of the evident challenges faced by the "model_others" was the presence of noise and artifacts in the 3D semantic maps. These issues included scattered points and "outliers," which could be attributed to the model's difficulty in coping with the inherent complexity and diversity of multiple scenes. The training's broader scope appears to have diluted its ability to generate clean and accurate semantic maps.

IV. Reference resources

(1) Semantic Segmentation - https://github.com/CSAILVision/semantic-segmentation-pytorch

V. Ckpt folder link

(1) Ckpt folder share link: https://drive.google.com/drive/folders/1sIP8_SW8VgBwEH7qAzY8Vczi8KmDgAn0?usp=drive_link