Tesk1

a. Code:

```python
import cv2
import numpy as np

points = []

class Projection(object):
    def __init__(self, image_path, points):
        """
            :param points: Selected pixels on top view (BEV) image
        """
        if type(image_path) != str:
            self.image = image_path
        else:
            self.image = cv2.imread(image_path)
        self.height, self.width, _ = self.image.shape
        self.points = points

    def top_to_front(self, theta=0, phi=0, gamma=0, dx=0, dz=0, fov=90):
        """
            Project the top view pixels to the front view pixels.
            :return: New pixels on perspective (front) view image
        """
        bev_pixels = []
        new_pixels = []

        principal_point = [self.width // 2, self.height // 2]
        f = (self.width / 2) * (1 / np.tan(np.deg2rad(fov / 2)))

        for point in self.points:
            pixel_x, pixel_y = principal_point[0] - point[0], principal_point[1] - point[1]
            bev_pixels.append([pixel_x, pixel_y, 1])

        bev_points = []
        for bev_pixel in bev_pixels:
            bev_point = [2.5 * bev_pixel[0] / f, 2.5 * bev_pixel[1] / f, 2.5]
            bev_points.append(bev_point)

        bev_points = np.array(bev_points)

        theta_rad = np.deg2rad(theta)  # Convert theta to radians
        c = np.cos(theta_rad)
        s = np.sin(theta_rad)
```

```python
        transformation_matrix = np.array([
            [1, 0, 0, dx],
            [0, c, -s, 1.5],
            [0, s, c, dz],
            [0, 0, 0, 1]
        ])
        # Apply transformation
        front_points = np.dot(transformation_matrix, np.hstack((bev_points, np.ones((bev_points.shape[0], 1)))).T).T[:, :3]

        for front_point in front_points:
            if front_point[2] != 0:  # Prevent division by zero
                new_pixel_x = principal_point[0] - int(front_point[0] / front_point[2] * f)
                new_pixel_y = principal_point[1] - int(front_point[1] / front_point[2] * f)
                new_pixels.append([new_pixel_x, new_pixel_y])

        return new_pixels

    def show_image(self, new_pixels, img_name='projection_result.png', color=(0, 0, 255), alpha=0.4):
        """
            Show the pro                      e selected area on perspective (front) view image.
                  (function) fillPoly: Any
        """
        new_image = cv2.fillPoly(self.image.copy(), [np.array(new_pixels)], color)
        new_image = cv2.addWeighted(new_image, alpha, self.image, (1 - alpha), 0)

        cv2.imshow(f'Top to front view projection {img_name}', new_image)
        cv2.imwrite(img_name, new_image)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        return new_image

def click_event(event, x, y, flags, params):
    # checking for left mouse clicks
    if event == cv2.EVENT_LBUTTONDOWN:
        print(x, ' ', y)
        points.append([x, y])
        cv2.circle(img, (x, y), 3, (0, 0, 255), -1)
        cv2.imshow('image', img)
```

```python
if __name__ == "__main__":
    front_rgb = "bev_data/front2.png"
    top_rgb = "bev_data/bev2.png"

    # click the pixels on window
    img = cv2.imread(top_rgb, 1)
    cv2.imshow('image', img)
    cv2.setMouseCallback('image', click_event)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    projection = Projection(front_rgb, points)
    new_pixels = projection.top_to_front(theta=90, dx=0, dz=0)
    projection.show_image(new_pixels)
```

b. Result and Discussion:

1. Import the required libraries

cv2：OpenCV，for image processing.

numpy：NumPy，used for mathematical calculations and matrix operations.

2. Define global variables points

Initialize an empty points list to store the pixels selected by the user.

3. Projection

Read the image file and obtain the size and pixel information of the image.

4. top_to_front　：Project top view pixels to front view

Calculate principal point

```python
principal_point = [self.width // 2, self.height // 2]
```

Calculate focal length

```python
f = (self.width / 2) * (1 / np.tan(np.deg2rad(fov / 2)))
```

To convert the pixel points in the top view into pixel coordinates relative to the main point:

```
pixel_x, pixel_y = principal_point[0] - point[0], principal_point[1] - point[1]
```

Convert top view pixels to 3D points with a depth of 2.5:

```
bev_point = [2.5 * bev_pixel[0] / f, 2.5 * bev_pixel[1] / f, 2.5]
```

Transforming the 3D bird eye view points to 3D front viewpoints by multiplying the transformation matrix：

```
transformation_matrix = np.array([
    [1, 0, 0, dx],
    [0, cos(theta), -sin(theta), 1.5],
    [0, sin(theta), cos(theta), dz],
    [0, 0, 0, 1]
])
```

Map the 3D points to 2D coordinates on the plane based on the focal length f and depth and calculate the new pixel position:

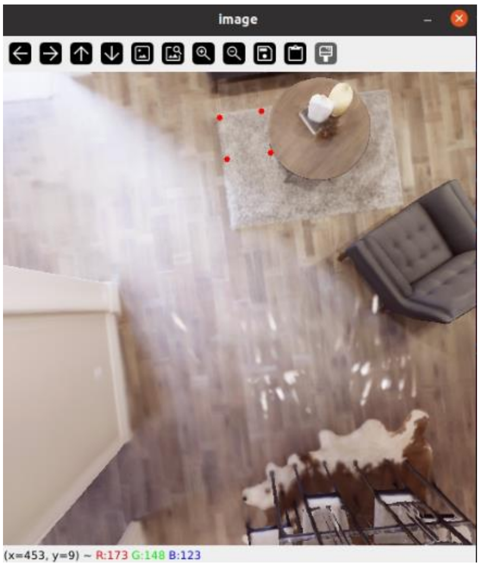
```
new_pixel_x = principal_point[0] - int(front_point[0] / front_point[2] * f)
new_pixel_y = principal_point[1] - int(front_point[1] / front_point[2] * f)
```

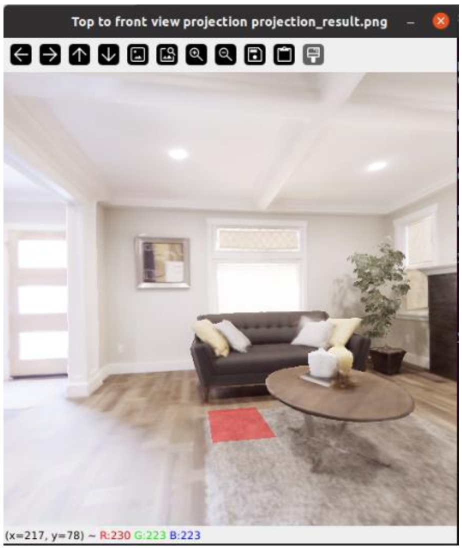
5. show_image：

Displays an image of the results and saves the results as a picture file.

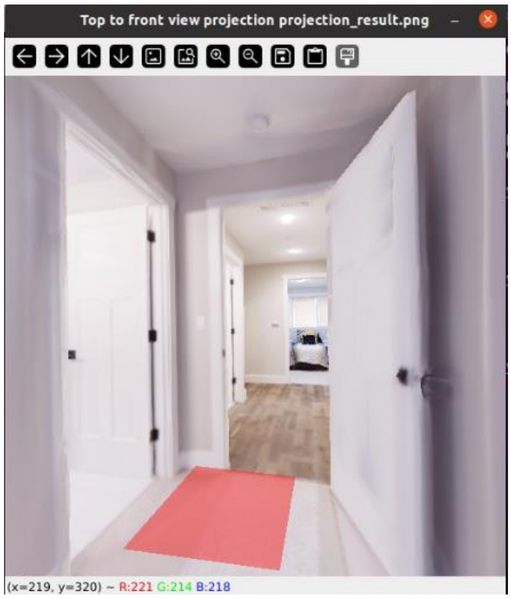
6. click_event：

When the user clicks on the image, the click location is recorded and a red dot is drawn.





The result of the first pair of images



The result of the second pair of images

參考資料

Tesk2:

```
import numpy as np
import open3d as o3d
from sklearn.neighbors import NearestNeighbors
import argparse
import os
import copy
import time
import math

def depth_image_to_point_cloud(rgb, depth):
    # Camera intrinsics
    principal_point = np.array([256, 256])
    f = (256) * (1 / np.tan(np.deg2rad(90 / 2)))

    images = [rgb, depth]
    processed_images = []

    for image_path in images:
        if image_path == rgb:
            image = np.asarray(o3d.io.read_image(image_path)) / 255  # Normalize RGB values
            processed_images.append(image)
        else:
            depth_image = np.asarray(o3d.io.read_image(image_path)) / 1000  # Convert depth to meters
            processed_images.append(depth_image)

    image, depth = processed_images

    # Generate pixel coordinates
    height, width = depth.shape
    pixel_coords = np.mgrid[0:height, 0:width]  # Create a grid of pixel coordinates

    # Flatten depth and pixel coordinates for processing
    d = depth.flatten()
    x = (principal_point[0] - pixel_coords[1].flatten()) * d / f
    y = (principal_point[1] - pixel_coords[0].flatten()) * d / f
    z = d

    # Stack coordinates into a single array
    xyz_points = np.array((x, y, z)).reshape(3, -1).T

    # Create point cloud
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(xyz_points)
    pcd.colors = o3d.utility.Vector3dVector(image.reshape(-1, 3))
```

```python
    # Create point cloud
    pcd = o3d.geometry.PointCloud()
    pcd.points = o3d.utility.Vector3dVector(xyz_points)
    pcd.colors = o3d.utility.Vector3dVector(image.reshape(-1, 3))

    # Create camera point
    camera_pcd = o3d.geometry.PointCloud()
    camera_pcd.points = o3d.utility.Vector3dVector([[0, 0, 0]])
    camera_pcd.colors = o3d.utility.Vector3dVector([[1, 0, 0]])  # Set the color to red

    return pcd, camera_pcd

def preprocess_point_cloud(pcd, voxel_size):

    pcd_down = pcd.voxel_down_sample(voxel_size=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 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
```

```python
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

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 compute_best_fit_transform(src_pts, tgt_pts):
    # Ensure that the dimensions of source and target points match
    assert src_pts.shape == tgt_pts.shape, "Source and target point shapes do not match."

    # Get the number of dimensions
    dimensions = src_pts.shape[1]

    # Calculate the centroid of each set of points
    mean_src = np.mean(src_pts, axis=0)
    mean_tgt = np.mean(tgt_pts, axis=0)

    # Shift the points to their respective centroids
    shifted_src = np.empty_like(src_pts)
    shifted_tgt = np.empty_like(tgt_pts)

    for i in range(src_pts.shape[0]):
        shifted_src[i] = src_pts[i] - mean_src
        shifted_tgt[i] = tgt_pts[i] - mean_tgt

    # Compute the covariance matrix
    covariance_matrix = np.dot(shifted_tgt.T, shifted_src)

    # Use Singular Value Decomposition to get the rotation matrix
    U, singular_values, Vt = np.linalg.svd(covariance_matrix)
    rotation_matrix = np.dot(U, Vt)

    # Handle the case of reflection
    if np.linalg.det(rotation_matrix) < 0:
        Vt[dimensions - 1, :] *= -1
        rotation_matrix = np.dot(U, Vt)
```

```python
    # Compute the translation vector
    translation_vector = mean_tgt - np.dot(rotation_matrix, mean_src)

    # Construct the homogeneous transformation matrix
    transform_matrix = np.zeros((dimensions + 1, dimensions + 1))
    for d in range(dimensions + 1):
        transform_matrix[d, d] = 1  # Set the diagonal to 1
    transform_matrix[:dimensions, :dimensions] = rotation_matrix
    transform_matrix[:dimensions, dimensions] = translation_vector

    return transform_matrix, rotation_matrix, translation_vector

def find_nearest_neighbor(src_points, dst_points):
    # Initialize the nearest neighbor searcher
    neighbor_search = NearestNeighbors(n_neighbors=1)
    neighbor_search.fit(dst_points)

    # Compute the nearest neighbor distances and indices for each source point
    distances, indices = neighbor_search.kneighbors(src_points, return_distance=True)

    # Select valid neighbors whose distances are less than 80% of the median
    threshold = np.median(distances) * 0.8
    valid_neighbors = distances < threshold

    return distances[valid_neighbors].ravel(), indices[valid_neighbors].ravel(), valid_neighbors.ravel()

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

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

    # Create homogeneous coordinates and copy original points
    src = np.ones((m + 1, source_points.shape[0]))
    dst = np.ones((m + 1, target_points.shape[0]))
    src[0:m, :] = np.copy(source_points.T)
    dst[0:m, :] = np.copy(target_points.T)

    # Initial pose estimation
    if trans_init is not None:
        src = np.dot(trans_init, src)

    prev_error = float('inf')  # Initialize previous error to a high value
```

```python
    # Main iteration loop
    for i in range(max_iterations):
        # Find nearest neighbors between current source and target points
        distances, indices, valid = find_nearest_neighbor(src[0:m, :].T, dst[0:m, :].T)

        # Compute the best-fit transformation between the source and the target
        transformation, _, _ = compute_best_fit_transform(src[0:m, valid].T, dst[0:m, indices].T)

        # Update the source points with the computed transformation
        src = np.dot(transformation, src)

        # Calculate the mean error to check for convergence
        mean_error = np.sum(distances) / np.count_nonzero(distances)

        # Check for convergence based on tolerance
        if np.abs(prev_error - mean_error) < tolerance:
            break
        prev_error = mean_error

    # Final transformation calculation
    transformation, _, _ = compute_best_fit_transform(source_points, src[0:m, :].T)

    return transformation

def accumulate_point_clouds(base_cloud, additional_clouds):
    for cloud in additional_clouds:
        # Add points from the new point cloud to the base point cloud
        base_cloud += cloud
    return base_cloud

def load_pose_data(args):
    # Set the file path
    path_to_file = f"{args.data_root}/GT_pose.npy"
    pose_data = np.load(path_to_file)

    # Convert the data into actual coordinates (units: meters)
    if args.floor == 1:
        # Convert millimeters to meters, rw => 10 / 0.25
        x_coords = -pose_data[:, 0] / 40
        y_coords = pose_data[:, 1] / 40
        z_coords = -pose_data[:, 2] / 40
    elif args.floor == 2:
```

```
    # Convert the data into actual coordinates (units: meters)
    if args.floor == 1:
        # Convert millimeters to meters, rw => 10 / 0.25
        x_coords = -pose_data[:, 0] / 40
        y_coords = pose_data[:, 1] / 40
        z_coords = -pose_data[:, 2] / 40
    elif args.floor == 2:
        # Convert the coordinates to meters, rw => 10 / 0.25
        x_coords = -pose_data[:, 0] / 40 - 0.00582
        y_coords = (pose_data[:, 1] / 40) - 0.07313
        z_coords = -pose_data[:, 2] / 40 - 0.03

    # Stack the coordinate points
    point_cloud_data = np.vstack((x_coords, y_coords, z_coords)).T

    # Create a point cloud object
    ground_truth_pcd = o3d.geometry.PointCloud()
    ground_truth_pcd.points = o3d.utility.Vector3dVector(point_cloud_data)
    ground_truth_pcd.paint_uniform_color([0, 0, 0])  # Set the color to black

    # Create line segments connecting points
    line_segments = [[i, i + 1] for i in range(len(point_cloud_data) - 1)]

    # Create a line set object
    ground_truth_lines = o3d.geometry.LineSet()
    ground_truth_lines.points = o3d.utility.Vector3dVector(point_cloud_data)
    ground_truth_lines.lines = o3d.utility.Vector2iVector(line_segments)

    return ground_truth_pcd, ground_truth_lines

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, "rgb/"))
    depth_images = os.listdir(os.path.join(data_folder_path, "depth/"))
```

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

for index in range(1, len(rgb_images)):
# Get the file path of RGB and depth images
    rgb_image_path = os.path.join(data_folder_path, "rgb/%d.png" % index)
    depth_image_path = os.path.join(data_folder_path, "depth/%d.png" % index)

    if index == 1:
        print("The principal picture is set as picture %d." % index)
        principal_pcd = depth_image_to_point_cloud(rgb_image_path, depth_image_path)
        pcd.append(principal_pcd[0])  # pcd[index-1]
        camera_pcd.append(principal_pcd[1])

        principal_pcd_down = preprocess_point_cloud(pcd[index - 1], voxel_size)
        pcd_down.appe                                                   1]
        fpfh.append(p  (function) def depth_image_to_point_cloud(
        pcd_transform      rgb: Any,
    else:                  depth: Any
        print("Proces  ) -> tuple
        source_pcd = depth_image_to_point_cloud(rgb_image_path, depth_image_path)
        pcd.append(source_pcd[0])  # pcd[index-1]
        camera_pcd.append(source_pcd[1])

        source_pcd_down = preprocess_point_cloud(pcd[index - 1], voxel_size)
        pcd_down.append(source_pcd_down[0])  # pcd_down[index-1]
        fpfh.append(source_pcd_down[1])  # target_fpfh[index-1]

        # Perform global registration
        global_registration_start_time = time.time()
        global_registration_result = execute_global_registration(pcd_down[index - 1], pcd_transformed[index - 2],
                                                                 fpfh[index - 1], fpfh[index - 2], voxel_size)
        print("Global registration took %.3f seconds." % (time.time() - global_registration_start_time))
```

```python
        #ICP
        icp_start_time = time.time()
        if args.version == 'open3d':
            icp_result = local_icp_algorithm(pcd_down[index - 1], pcd_transformed[index - 2],
                                             global_registration_result.transformation, voxel_size)
            transformation_matrix = icp_result.transformation  # transformation of index to index-1
        elif args.version == 'my_icp':
            icp_result = my_icp(pcd_down[index - 1], pcd_transformed[index - 2],
                                global_registration_result.transformation)
            transformation_matrix = icp_result  # transformation of index to index-1
            # draw_registration_result(pcd_down[index - 1], pcd_transformed[index - 2], transformation_matrix)
        print("ICP took %.3f seconds.\n" % (time.time() - icp_start_time))

        #Convert the point cloud to the coordinate system of the first camera
        pcd_transformed.append(pcd_down[index - 1].transform(transformation_matrix))
        camera_pcd[index - 1] = camera_pcd[index - 1].transform(transformation_matrix)

# Merge the transformed point clouds
point_cloud = accumulate_point_clouds(point_cloud, pcd_transformed)

# Merge the camera point clouds
estimate_camera_cloud = accumulate_point_clouds(estimate_camera_cloud, camera_pcd)

estimate_camera_cloud.colors[0] = [0,0,0]
estimate_lines = []
for i in range(len(estimate_camera_cloud.points) - 1):
    estimate_lines.append([i, i + 1])
estimate_line_set = o3d.geometry.LineSet()
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 = 0.01
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)
```

```python
    gt_pose_cloud, gt_line_set = load_pose_data(args)
    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

def calculate_mean_l2_distance(gt_pos_pcd, estimate_camera_cloud):
    total_distance = 0
    num_points = len(estimate_camera_cloud.points)

    # Iterate over each point and calculate the distance
    for index in range(num_points):
        # Extract the respective coordinates
        gt_x = gt_pos_pcd.points[index][0]
        gt_y = gt_pos_pcd.points[index][1]
        gt_z = gt_pos_pcd.points[index][2]

        est_x = estimate_camera_cloud.points[index][0]
        est_y = estimate_camera_cloud.points[index][1]
        est_z = estimate_camera_cloud.points[index][2]

        # Compute the coordinate differences
        delta_x = gt_x - est_x
        delta_y = gt_y - est_y
        delta_z = gt_z - est_z

        # Calculate the Euclidean distance
        distance = (delta_x ** 2 + delta_y ** 2 + delta_z ** 2) ** 0.5
        total_distance += distance

    # Return the average distance
    average_distance = total_distance / num_points if num_points > 0 else 0
    return average_distance

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 or my_icp')
    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/"
```

```python
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 or my_icp')
    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/"

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

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

    # TODO: Visualize result
    o3d.visualization.draw_geometries([result_pcd,gt_pos_pcd,line_set,estimate_camera_cloud, estimate_line_set])
```

The my_icp function implements a custom Iterative Closest Point algorithm.

- Input Parameters
  - `source_down`: Downsampled source point cloud.
  - `target_down`: Downsampled target point cloud.
  - `trans_init`: Initial transformation matrix (optional).
  - `max_iterations`: Maximum number of iterations for the algorithm.
  - `tolerance`: Convergence criterion based on mean error.
- Initialization
  - The source and target point clouds are converted to NumPy arrays.
  - Homogeneous coordinates are created for both clouds by adding an additional row of ones.
- Pose Estimation
  - If an initial transformation is provided, it is applied to the source points.
- Main Iteration Loop
  - A loop runs for a maximum of `max_iterations`, during which:
    - The nearest neighbors are found using the `find_nearest_neighbor` function.
    - The best-fit transformation between the source and target points is computed using `compute_best_fit_transform`.
    - The source points are updated using the computed transformation.
    - The mean error is calculated to monitor convergence.
    - The loop exits if the change in mean error is less than the specified tolerance.
- Final Transformation
  - The final transformation matrix is computed between the original source points and the transformed source points.
  - The function returns this transformation matrix.

The reconstruct function orchestrates the 3D reconstruction process using the point clouds generated from RGB and depth images.

- **Configuration and Initialization**
    - The function begins by setting parameters such as voxel size and initializing point cloud objects.
    - It reads the RGB and depth images from the specified directory based on the input arguments.
- **Processing Images**
    - A loop iterates over each image index:
        - For the first image, it converts the RGB and depth images to a point cloud.
        - For subsequent images, it processes the images and computes global registration.
        - The ICP algorithm (either Open3D's or the custom `my_icp`) is applied to refine the alignment between point clouds.
- **Accumulating Point Clouds**
    - Transformed point clouds are merged into a single point cloud object for visualization.
- **Filtering Ceiling Points**
    - Based on the specified floor, points above a certain threshold are removed to filter out the ceiling.
- **Loading Ground Truth Data**
    - Ground truth pose data is loaded and visualized as a line set.
- **Return Values**
    - The function returns the reconstructed point cloud, ground truth point cloud, line set, estimated camera cloud, and estimated line set.

a. What’s the meaning of extrinsic matrix and intrinsic matrix?

The **intrinsic matrix** refers to the parameters of a camera that define its internal characteristics. It includes focal length, principal point, and skew. The intrinsic matrix transforms 3D points in the camera coordinate system to 2D points in the image plane. It is usually represented as:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where fxf_xfx and fyf_yfy are the focal lengths in terms of pixels, sss is the skew coefficient, and (cx,cy)(c_x, c_y)(cx,cy) is the principal point.

On the other hand, the **extrinsic matrix** represents the camera's position and orientation in the world coordinate system. It defines the transformation from world coordinates to camera coordinates, combining rotation and translation. The extrinsic matrix can be expressed as:

$$E = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$

where RRR is the rotation matrix and ttt is the translation vector. Together, the intrinsic and extrinsic matrices allow the conversion between 3D world points and their corresponding 2D image points.

b. Have you ever tried to do ICP alignment without global registration, i.e., RANSAC? How’ s the performance? Explain the reason. (Hint: The limitation of ICP alignment)

Yes, I have attempted ICP alignment without global registration methods like RANSAC. The performance of this approach was suboptimal, particularly in scenarios where there were significant initial misalignments or outliers in the point cloud data.

The limitation of ICP (Iterative Closest Point) lies in its sensitivity to the initial alignment and outliers. ICP minimizes the distance between corresponding points, assuming that the initial guess is close to the true alignment. Without global registration to provide a better initial estimate or to remove outliers, ICP can easily converge to a local minimum, leading to poor alignment results. Additionally, the presence of noise and outliers can significantly distort the matching process, resulting in inaccurate transformations and misaligned point clouds.

c. Describe the tricks you apply to improve your ICP alignment.

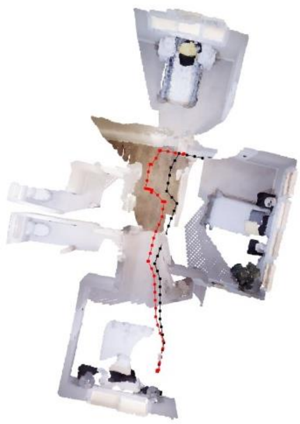
To improve ICP alignment, I have applied several strategies:

1. **Preprocessing the Point Cloud**: I filter out noise and outliers using techniques such as voxel grid filtering or statistical outlier removal. This helps in reducing the influence of erroneous points during the ICP process.
2. **Choosing Good Initial Estimates**: By utilizing global registration techniques like RANSAC or using feature-based matching, I can obtain a better initial guess for the transformation, which significantly enhances the convergence of ICP.
3. **Multi-resolution Approach**: I employ a multi-resolution strategy, starting with a downsampled version of the point clouds for the initial alignment and then refining the alignment on higher-resolution data. This approach allows ICP to converge faster and more accurately.
4. **Adaptive Point Selection**: Instead of using all points, I selectively use the most informative points based on their distance to the surface or normals. This can lead to better convergence as it reduces computational load and focuses on more relevant areas of the point cloud.
5. **Iterative Refinement**: I perform multiple iterations of ICP with different settings, adjusting parameters such as the distance threshold for point matching to improve alignment iteratively.

Open3D 1F

my_icp 1F

Open3D 2F



my_icp 2F