CSIE5429 3D Computer Vision with Deep Learning Applications

Homework 2 Report

Name: 高榮浩 **ID:** R12922127

- **Problem 1: 2D-3D Matching**
  - **Q1-1 Solving Camera Pose Estimation with 2D-3D Correspondence Matching and PnP Algorithms**
    - **Implementation**

I'm employing the Direct Linear Transformation (DLT) method in conjunction with the RANdom SAmple Consensus (RANSAC) technique. I made a deliberate choice not to use the P3P (Perspective-3-Point) method due to its complex algorithm and challenging implementation. After numerous attempts, I opted for the DLT method, which shares similarities with the approach covered in our initial homework assignment and offers a more straightforward implementation.

In my implementation, I've identified several key points. Firstly, for feature matching, I've selected the FlannBasedMatcher over the BFMatcher. This decision stems from the fact that the FlannBasedMatcher demonstrates superior speed compared to the BFMatcher. Secondly, when determining a good fit for each iteration in RANSAC, I not only verify if the number of inliers exceeds the defined threshold, but also ensure that the mean of the distances between the predicted and actual points for each data pair is smaller than before.

◆ Pseudo Code

```
function RANSAC:
    # Set the parameters
    let s be the minimum number of points needed
    let N be the number of iterations
    let d be the threshold for a good fit
    let T be the minimum nearby points for a model to be considered a good fit

    # Perform RANSAC iterations
    Repeat N times:
        # Randomly select a sample of s point pairs from the dataset
        Draw s random point pairs from the data
        # Fit a model to the selected point pairs using DLT
        Fit a model to the selected pairs using the DLT method

        # Test the model with data points outside the sample
        For each data point pairs:
            # Calculate the distance between the predicted point and the actual point
            Compute the distance between the predicted point and the actual point
            # Check if the distance is within the defined threshold (d)
            If the distance is less than d, consider the point as an inlier

        # Check if there are T or more inliers (good fits)
        If there are T or more inliers, the model is considered a good fit

    # Use the best-fit model from the iterations, based on the fitting error
    Select the model with the lowest fitting error as the best-fit model
```

```
function DLT:
    Undistort the 2D points

    # Define intrinsic matrix parameters
    let f_x, f_y, c_x, c_y be the parameters of the intrinsic matrix

    # Build the matrix A from at least 6 points correspondences (x, y, z) <-> (u, v)
    A = [[x * f_x, y * f_x, z * f_x, f_x, 0, 0, 0, 0, x * c_x - u * x, y * c_x - u * y, z * c_x - u * z, c_x - u], [0, 0, 0, 0, x * f_y, y * f_y, z * f_y, f_y, x * c_y - v * x, y * c_y - v * y, z * c_y - v * z, c_y - v], ...]

    # Obtain the Singular Value Decomposition (SVD) of A: A = U1 @ S1 @ Vh1
    Obtain the SVD of A: A = U1 @ S1 @ Vh1

    # Extract the last column of Vh1 as x_bar
    let x_bar be Vh1[-1]

    # Form the rotation matrix R_bar from x_bar
    R_bar = [[x_bar[0], x_bar[1], x_bar[2]], [x_bar[4], x_bar[5], x_bar[6]], [x_bar[8], x_bar[9], x_bar[10]]

    # Obtain the SVD of R_bar: R_bar = U2 @ S2 @ Vh2
    Obtain the SVD of R_bar: R_bar = U2 @ S2 @ Vh2

    # Extract the rotation matrix R from U2 and Vh2
    R = U2 @ Vh2

    # Calculate the scaling factor beta
    beta = 3 / trace of S2

    # Check the sign of beta * (x * x_bar[8] + y * x_bar[9] + z * x_bar[10] + x_bar[11])
    if beta * (x * x_bar[8] + y * x_bar[9] + z * x_bar[10] + x_bar[11]) < 0:
        # Reverse the rotation matrix and negate beta
        R = -R
        beta = -beta

    # Calculate the translation vector t
    t = [[beta * x_bar[3]], [beta * x_bar[7]], [beta * x_bar[11]]]
```

■ Q1-2 Evaluating Pose Estimation Accuracy: Calculating Median Pose Error and Discussion

◆ Notation

- s: the minimum number of points needed
- d: the threshold for a good fit
- Note: When we express "s = 1%," it signifies that 's' equates to 1% of the total number of data point pairs.

◆ Best Result

```
Rotation Error: 0.0016616088974186205
Translation Error: 0.005580167958663298 5
```

◆ Rotation Error

	s = 6	*s = 1%*
d = 10	0.00306	0.00187
d = 15	0.00302	**0.00166**
d = 20	0.00317	0.00204

◆ Translation Error

	s = 6	*s = 1%*
d = 10	0.01158	0.00829
d = 15	0.01126	**0.00558**
d = 20	0.01290	0.00782

◆ Discussion

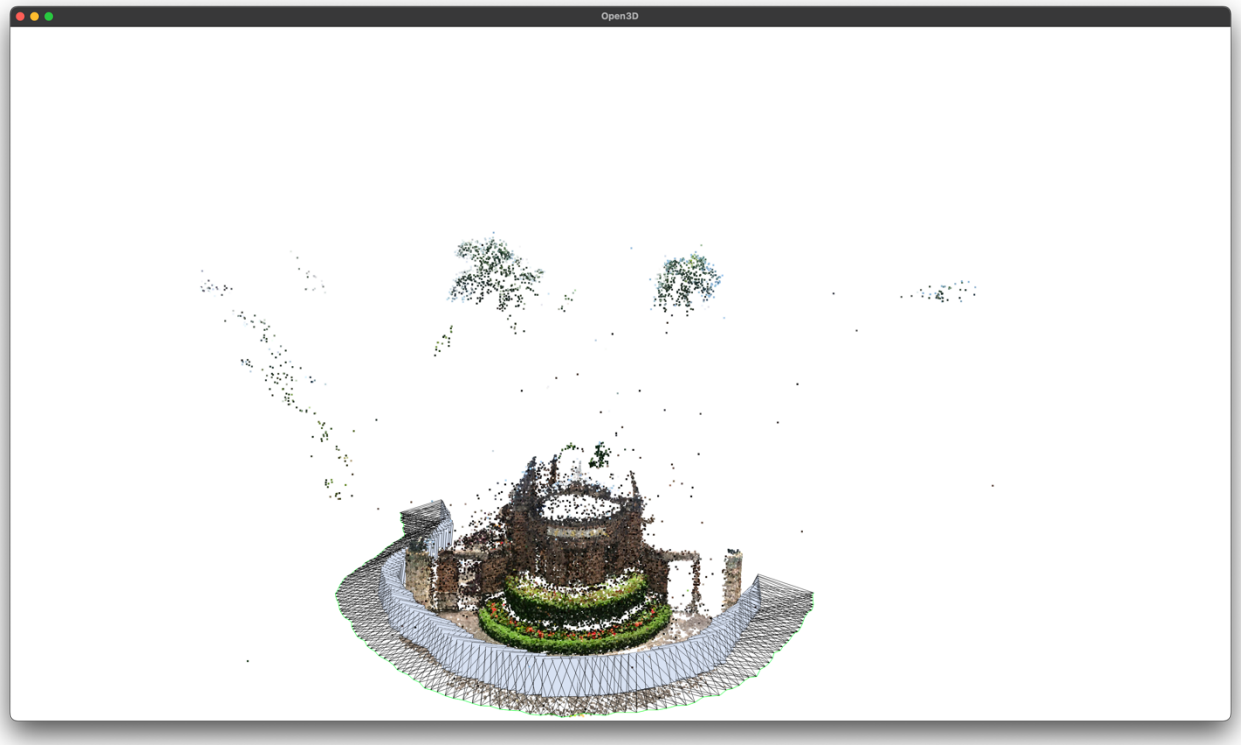
Here are the hyperparameters I've set for the process:

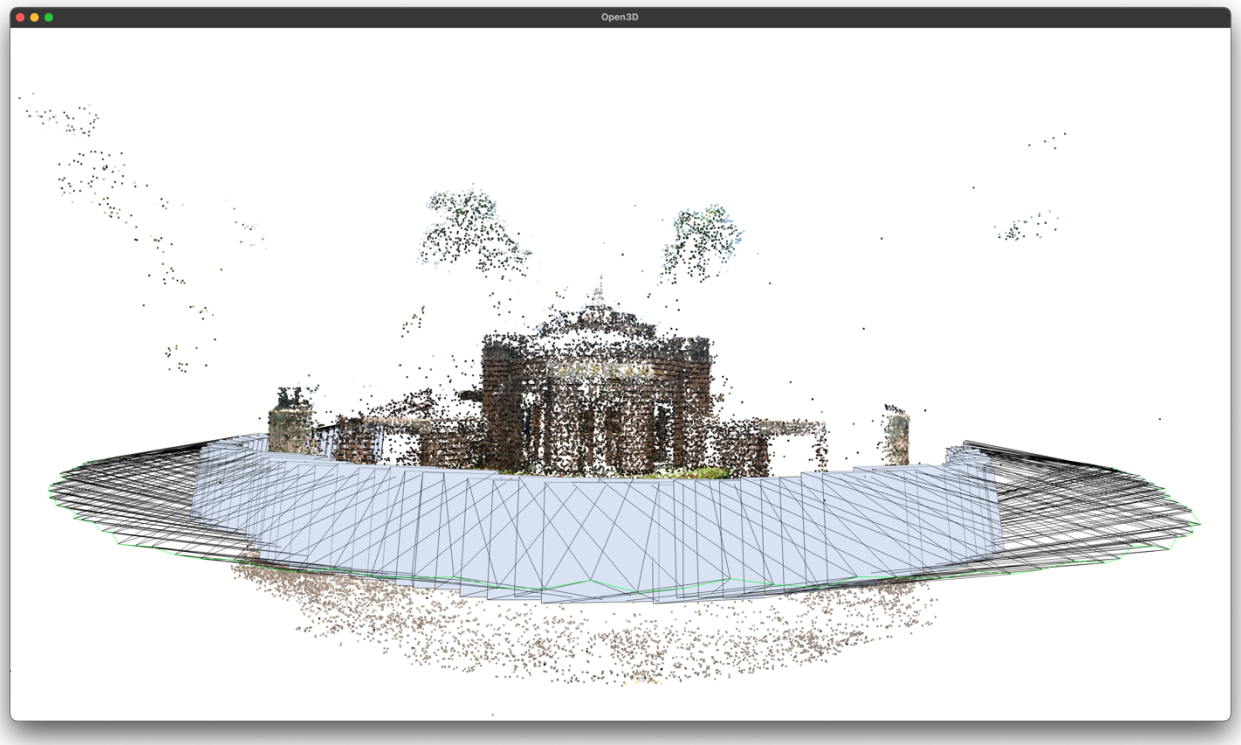
- 's' represents the minimum number of points required, and it is defined as 1% of the total number of data point pairs. It's worth noting that I experimented with higher percentages, which resulted in lower errors. However, increasing 's' also led to longer computation times. In the end, I opted for 1% as a balance between accuracy and efficiency.
- 'N' signifies the number of iterations and is set to 100.
- 'd' is the threshold for a good fit and is established at 15. This value was determined to be the best result based on prior experimentation.
- 'T' stands for the minimum number of nearby points for a model to be considered a good fit. It is set to half of the total number of data point pairs.

It's important to note that I haven't utilized any training data in this implementation. In the future, using training data to fine-tune these hyperparameters and potentially reduce errors could be a promising avenue for improvement.

■ **Q1-3 Visualizing Camera Poses and Trajectory with 3D Point Clouds Using Open3D**

◆ **Result**





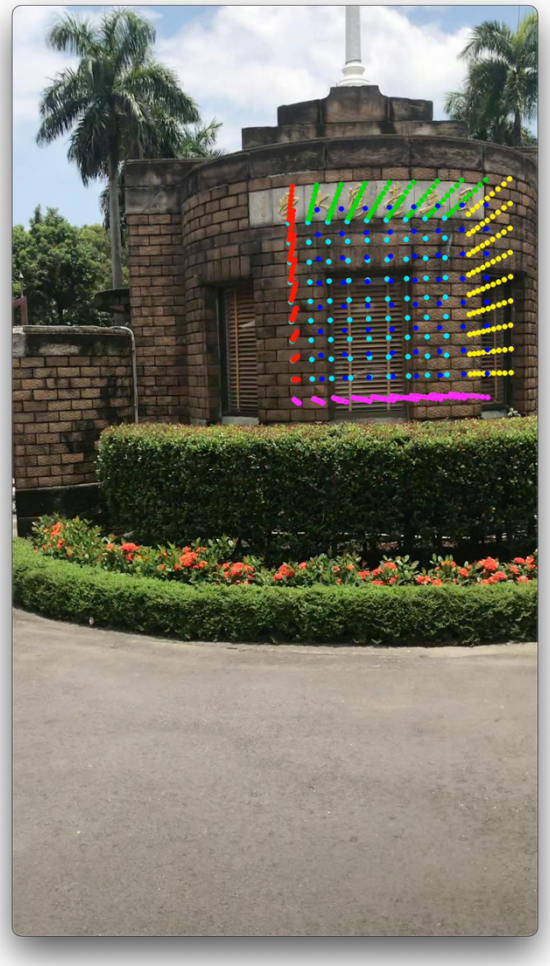
◆ Explanation and Discussion

In each camera pose, we have a total of 8 lines to consider, along with the necessity to draw 1 quadrangle. The process begins by initially recording all data points. Subsequently, we define which points will form the lines, and we specify the vertices required to create a triangle. Finally, we render the point cloud, lines, and quadrangles. It's important to note that the quadrangle is constructed from two triangles by cutting it along its diagonal. Despite this, it manifests two distinct faces in 3D space. To prevent any transparency issues in either face, we build the quadrangle using four triangles: two within the same face and the other two within another face.

● Problem 2: Augmented Reality

■ Q2-1 Creating Augmented Reality with Virtual Cube in Validation Image Sequences Using Painter's Algorithm

◆ Result



◆ Implementation

First, we manually select the cube's position within the point cloud. Then, for the 8 vertices we've chosen, we determine the points on its 6 surfaces that we need to reference in the output AR video. For each validation image, we follow a similar process to problem 1, where we estimate the camera's pose and calculate the 3D position of the camera's apex. From there, we calculate the distances from every point that needs to be referenced in the output AR video to the apex position and sort them from farthest to nearest.

Subsequently, for each point we need to reference in the output AR video, we transform its 3D position into a 2D position on the image by multiplying it with the intrinsic and extrinsic matrices we previously estimated. We then check whether the point lies within the boundaries of the image. If it does, we draw a circle on the image. We repeat this process for each validation image, and the output AR video is generated by combining these images. It's worth noting that we've implemented the painter's algorithm, where we initially sort the 3D distances between the points to be drawn and the apex of the camera's pose before drawing them on the validation image in order. This can be demonstrated in the captured frames of the output AR video.

- **YouTube Link**
  https://youtu.be/fBk_blK2JQ0

- **Execution**
  - **Command**
    - **Problem 1: 2D-3D Matching**
      `python 2d-3d_matching.py`

    - **Problem 2: Augmented Reality**
      `python augmented_reality.py`

  - **Environment / Package**
    - `python==3.8.18`
    - `scipy==1.10.1`
    - `pandas==2.0.3`
    - `numpy==1.24.3`
    - `opencv-python==4.8.1.78`
    - `open3d==0.17.0`
    - `tqdm==4.66.1`