# Computer Vision Homework 4 - Structure from Motion

**Group 15 - 313553014 廖怡誠、313553013 李品妤、312553007 林芷萱**

## 1. Introduction

Structure from Motion (SfM) is a computer vision technique for reconstructing 3D structures from a set of 2D images taken from different angles. It identifies key-points in the images, matches them across overlapping views, and estimates both the camera parameters and the 3D geometry of the scene. Through processes like feature matching, camera pose estimation, and triangulation, SfM produces either sparse point clouds or detailed 3D models.

SfM's versatility in producing accurate 3D reconstructions from simple images has made it an essential, cost-effective, and accessible tool across research, industry, and creative fields. Its ability to generate detailed models continues to drive innovation and expand its applications in diverse areas, solidifying its value as a powerful method for 3D reconstruction.

In this report, we introduce our methods for implementing SfM and conduct several experiments to evaluate the impact of various factors. These issues will be discussed in the discussion section.

## 2. Implementation Procedure

In this section, we describe our implementation of Structure from Motion (SfM) step by step.

### 2.1. Find out correspondences across the images

To identify points in the input images that correspond to the same positions in the real-world coordinate system, we use a feature detection and matching approach. First, we apply a feature detector to extract the key-point positions and their local features in each image. Then, we match the key-points between the images. Let $F(I)$ represent the feature points extracted from image $I$. We calculate the Euclidean distance between each key-point in image $I$ and the key-points in image $J$, then filter the appropriate key-point pairs as matching points. For example, given a key-point in image $I$, we identify the first-nearest and second-nearest key-points in image $J$ based on their Euclidean distances. If the ratio of the first-nearest distance to the second-nearest distance is less than a predefined threshold, the pair is considered a matching point. As the threshold decreases, fewer matching points are identified.

In our implementation, we use the Scale-Invariant Feature Transform (SIFT) to extract key-point information and match them using the Brute-Force Matcher from OpenCV. Finally, we identify matching points using a threshold of 0.7 and create a corresponding array of these points.

```python
""" step1. find out correspondence across images """
def extract_keypoints_and_features(self):

    sift_detector = cv2.SIFT_create()
    gray_imag1 = cv2.cvtColor(self.img1, cv2.COLOR_BGR2GRAY)
    gray_imag2 = cv2.cvtColor(self.img2, cv2.COLOR_BGR2GRAY)
    # find keypoints of images
    keypts1, des1 = sift_detector.detectAndCompute(gray_imag1, None)
    keypts2, des2 = sift_detector.detectAndCompute(gray_imag2, None)
    # create matcher with Euclidean ditance calculating
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    # matches = bf.match(des1, des2)
    # good_matches = sorted(matches, key=lambda x: x.distance)

    # find the first-nearest and second nearest point
    matches = bf.knnMatch(des1, des2, k=2)
    good_matches = []
    ratio_threshold = 0.7
    for m, n in matches:
        if (m.distance / n.distance) < ratio_threshold :
            good_matches.append(m)
    # get the point pair from the good matches list, then create the points1 and points2.
    pts1 = np.array([keypts1[m.queryIdx].pt for m in good_matches])
    pts2 = np.array([keypts2[m.trainIdx].pt for m in good_matches])

    return pts1, pts2
```

## 2.2. Estimate the fundamental matrix

We apply the normalized 8-point algorithm to estimate the fundamental matrix between two images. Fundamental matrix is a geometric relation which lets one coordination from focus C transform to the other coordination from focus C'. Next, we will give a detailed description.

### Step 1. Normalization of Points

To improve numerical stability, the first step is to normalize the corresponding points from the two images. Assume the point sets from the two images are $\{x_i = [x_i, y_i, 1]^T\}$ and $\{x_i' = [x_i', y_i', 1]^T\}$. Each set of points is transformed by a normalization matrix $T$ (for the first image) and $T'$ (for the second image). The transformation centers the points at the origin and scales their average distance to $\sqrt{2}$. The normalization matrix is defined as:

$$T = \begin{bmatrix} s & 0 & -s \cdot x_{mean} \\ 0 & s & -s \cdot y_{mean} \\ 0 & 0 & 1 \end{bmatrix}, s = \frac{\sqrt{2}}{\text{average distance}}$$

where $(x_{mean}, y_{mean})$ is the centroid of the points, and $s$ is the scaling factor. After normalization, the points are transformed to $\hat{x}_i = Tx_i$ and $\hat{x}_i' = T'x_i'$ respectively.

```python
    # step2. estimate the fundamental matrix across images (normalized 8 points)
    def __normalize_pts(self, pts):
        """Normalize points to have mean = 0 and average distance = sqrt(2)."""
        centroid = np.mean(pts, axis=0)
        shifted_points = pts - centroid
        mean_dist = np.mean(np.sqrt(np.sum(shifted_points**2, axis=1)))
        scale = np.sqrt(2) / mean_dist
        T = np.array([[scale, 0, -scale * centroid[0]],
                      [0, scale, -scale * centroid[1]],
                      [0, 0, 1]])
        normalized_points = (T @ np.column_stack((pts, np.ones(len(pts))))).T).T
        return normalized_points[:, :2], T
```

## Step 2. Constructing Matrix A

Based on the Fundamental Matrix constraint $\hat{x}'^T F \hat{x} = 0$, we have the equation: $Af = 0$ where $A$ is a $n \times 9$ matrix (with $n$ being the number of point correspondences). Each row of $A$ is constructed as: $A_i = [x_i'x_i, y_i'x_i, x_i', x_i'y_i, y_i'y_i, y_i', x_i, y_i, 1]$. The vector $f$ is the flattened representation of the Fundamental Matrix $F$.

```python
    def __construct_matrix_A(self, pts1, pts2):
        """Construct matrix A for the 8-point algorithm."""
        A = []
        for (x1, y1), (x2, y2) in zip(pts1, pts2):
            A.append([x1 * x2, y1 * x2, x2, x1 * y2, y1 * y2, y2, x1, y1, 1])
        return np.array(A)
```

## Step 3. Solving for F using SVD

Under ideal conditions, the solution to $Af = 0$ satisfies $\|Af\| = 0$. To find this solution, Singular Value Decomposition (SVD) is applied to $A$:

$$A = U\Sigma V^T$$

Here, $\Sigma$ is the diagonal matrix of singular values. The last column of $V^T$(corresponding to the smallest singular value) provides the solution $f$. Reshaping $f$ into a $3 \times 3$ matrix gives the initial estimate of the Fundamental Matrix $F$.

## Step 4. Enforcing Rank-2 Constraint

The computed $F$ may not have a rank of 2 due to numerical errors. To enforce the rank-2 constraint, $F$ is decomposed using SVD:

$$F = U\Sigma V^T$$

The smallest singular value in $\Sigma = [s_1, s_2, s_3]$ is set to zero ($s_3 = 0$), and $F$ is reconstructed as:

$$F = U \cdot diag(s_1, s_2, 0) \cdot V^T$$

This ensures that $F$ has a rank of 2.

### Step 5. Denormalization

The final step is to map the normalized Fundamental Matrix $F$ back to the original coordinate system using the inverse of the normalization matrices $T$ and $T'$:

$$F = T'^{T} \cdot F \cdot T$$

To ensure the Fundamental Matrix $F$ has a consistent scale, it is often normalized such that $F_{33} = 1$. This $F$, which is in the original coordinate space, describes the epipolar geometry between the two images.

```
63      def estimate_fundamental_matrix(self, pts1, pts2):
64          # Normalize points
65          pts1_norm, T1 = self.__normalize_pts(pts1)
66          pts2_norm, T2 = self.__normalize_pts(pts2)
67
68          # Construct A matrix
69          A = self.__construct_matrix_A(pts1_norm, pts2_norm)
70
71          # Solve for F using SVD
72          _, _, Vt = np.linalg.svd(A)
73          F = Vt[-1].reshape(3, 3)
74
75          # Enforce rank-2 constraint
76          U, S, Vt = np.linalg.svd(F)
77          S[-1] = 0
78          F = U @ np.diag(S) @ Vt
79
80          # Denormalize
81          F = T2.T @ F @ T1
82
83          return F / F[2, 2]
```

## 2.3. Draw the interest points and epipolar lines

First, using the fundamental matrix FFF and the matching points from the second image (pts2), we calculate the epipolar lines in the first image using the OpenCV function cv2.computeCorrespondEpilines. These epipolar lines are represented by the equation $ax + b + c = 0$. To determine the endpoints of these lines within the image, we compute their intersections with the left and right borders of the image. Specifically, for an image of width $w$, when $x = 0$, the left endpoint is given by $y_0 = -c/b$. Similarly, when $x = w$, the right endpoint is calculated as $y_1 = -(c + a \cdot w)$. With these two endpoints, the epipolar line can be drawn across the image. Next, the epipolar lines are plotted on the first image, and the corresponding feature points from $pts_1$ are highlighted using cv2.circle. At the same time, the matching points from the second image ($pts_2$) are marked similarly to visualize the correspondences.

```python
# step3. draw the interest points on you found in step.1 in one image and the corresponding epipolar lines in another
def draw_epipolar_lines(self, F, pts1, pts2):

    imag1 = self.img1
    imag2 = self.img2

    lines = cv2.computeCorrespondEpilines(pts2.reshape(-1, 1, 2), 2, F)
    lines = lines.reshape(-1, 3)

    h, w = imag1.shape[:2]
    for r, pt1, pt2 in zip(lines, pts1, pts2):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, -r[2] / r[1]])
        x1, y1 = map(int, [w, -(r[2] + r[0] * w) / r[1]])
        imag1 = cv2.line(imag1, (x0, y0), (x1, y1), color, 1)
        imag1 = cv2.circle(imag1, tuple(map(int, pt1)), 5, color, -1)
        imag2 = cv2.circle(imag2, tuple(map(int, pt2)), 5, color, -1)

    plt.figure(figsize=(10, 5))
    plt.subplot(121), plt.imshow(cv2.cvtColor(imag1, cv2.COLOR_BGR2RGB))
    plt.title('Epipolar Lines on Image 1')
    plt.subplot(122), plt.imshow(cv2.cvtColor(imag2, cv2.COLOR_BGR2RGB))
    plt.title('Epipolar Lines on Image 2')
```

## 2.4. Estimated essential matrix and get 4 possible solutions

The Essential Matrix $E$ is a special case of the Fundamental Matrix $F$, where the camera intrinsic parameters $K$ are known. The relationship between $F$ and $E$ is given by:

$$E = K^T F K$$

where $K$ is the intrinsic calibration matrix of the camera and $E$ encodes the relative rotation and translation between two camera views.

To ensure the Essential Matrix satisfies its theoretical constraint, i.e., it must have two singular values equal and the third equal to zero, we decompose $E$ with SVD. Then replacing the diagonal matrix $\Sigma$ with a corrected matrix:

$$\Sigma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Finally, reconstructing the essential matrix using the corrected singular values.

Based on constraint of essential matrix, we know there are four possible combinations of (R, t) which are rotation matrix and translation matrix. Therefore, we decompose E with SVD:

$$E = U \Sigma V^T.$$

To ensure proper orientation, the determinant of $U$ and $V^T$ is checked. If $\det(U)$ or $\det(V^T)$ is negative, their signs are flipped to ensure they represent valid rotations. From the SVD, two possible rotations matrices ($R_1$ and $R_2$) are derived using a predefined matrix $W$:

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These are calculated as $R_1 = UWV^T$ and $R_2 = UW^TV^T$. Similarly, the translation vector $t$ is extracted as the third column of $U$. Since $t$ represents a direction, it is determined up to a sign, resulting in four possible combinations: $(R_1, t), (R_1, -t), (R_2, t),$ $and$ $(R_2, -t)$.

```python
    # step4. get 4 possible solutions of essential matrix from fundamental matrix
    def get_essential_matrix(self, K, F):
        # computte essential matrix
        E = K.T @ F @ K
        # decompose E using SVD
        U, S, Vt = np.linalg.svd(E)
        # Correct signular values to enforces rank-2 constraint
        new_S = np.diag([1, 1, 0])
        E = U @ new_S @ Vt
        return E

    def __decompose_essential_matrix(self, E):
        """ Decompose Essential Matrix into possible rotations and translation. """
        U, _, Vt = np.linalg.svd(E)

        # Ensure proper orientation
        # if np.linalg.det(U) < 0:
        #     U *= -1
        # if np.linalg.det(Vt) < 0:
        #     Vt *= -1

        # Define W matrix
        W = np.array([
            [0, -1, 0],
            [1,  0, 0],
            [0,  0, 1]
        ])

        # Compute possible rotations
        R1 = U @ W @ Vt
        R2 = U @ W.T @ Vt

        # Ensure rotations are valid (det(R) = 1)
        if np.linalg.det(R1) < 0:
            R1 *= -1
        if np.linalg.det(R2) < 0:
            R2 *= -1

        # Compute translation vector
        t = U[:, 2]  # Third column of U
        return R1, R2, t
```

## 2.5. Find out the best solution with triangulation

To select the correct combination of $(R, t)$, a test is performed based on the triangulation of 3D points and the positive depth constraint. First, the projection matrix for the first camera is defined as $P_1 = K_1[I|0]$, where $K_1$ is the intrinsic matrix, $I$ is the identity matrix, and the zero vector represents no translation. For the second camera, the projection matrix is computed for each candidate solution as $P_2 = K_2[R|t]$.

The triangulation process reconstructs 3D points using corresponding 2D points ($pts_1$ and $pts_2$) from the two views and the projection matrices $P_1$ and $P_2$. The cv2.triangulatePoints function is used to obtain the 3D points in homogeneous coordinates, which are then converted to inhomogeneous coordinates by dividing by the fourth (homogenous) component. The positive depth constraint ensures that the reconstructed 3D points lie in front of the cameras. The solution with the maximum number of points having positive depth is considered the correct one.

```python
    # step5. find out the most appropriate solution of essential matrix
    def find_appropriate_essential_matrix(self, E, pts1, pts2, K1, K2):
        """ Determine the correct (R, t) solution from Essential Matrix. """
        R1, R2, t = self.__decompose_essential_matrix(E)
        solutions = [(R1, t), (R1, -t), (R2, t), (R2, -t)]

        # Projection matrices for the first camera (identity rotation, no translation)
        P1 = K1 @ np.hstack((np.eye(3), np.zeros((3, 1))))

        # Test all four combinations
        best_solution = None
        max_positive_depth = 0

        for R, t in solutions:
            # Projection matrix for the second camera
            P2 = K2 @ np.hstack((R, t.reshape(-1, 1)))
            pts1_h = cv2.convertPointsToHomogeneous(pts1).reshape(-1, 3).T  # 2D point to homo
            pts2_h = cv2.convertPointsToHomogeneous(pts2).reshape(-1, 3).T  # 2D point to homo

            # Triangulate points
            points_4d_h = cv2.triangulatePoints(P1, P2, pts1_h[:2], pts2_h[:2])
            # Convert to inhomogeneous coordinates
            points_3d = points_4d_h[:3, :] / points_4d_h[3, :]
            # Count points with positive depth
            positive_depth = np.sum(points_3d[2, :] > 0)

            if positive_depth > max_positive_depth:
                max_positive_depth = positive_depth
                best_solution = (R, t)

        return best_solution
```

## 2.6. Apply triangulation to get 3D points

This part is the same as **section 2.5**, because finding the appropriate rotation and translation also requires calculating the 3D points. In addition, because it is needed to generate a 3D model, we additionally output the 2D projection and camera matrix.

```python
    # setp6. apply triangulation to get 3D points
    def get_3d_points(self, R, t, K, pts1, pts2):
        # Projection matrices for the first camera (identity rotation, no translation)
        P1 = np.hstack((np.eye(3), np.zeros((3, 1))))
        # Intrinsic matrix for both cameras
        P1 = K @ P1
        # Projection matrix for the second camera
        P2 = K @ np.hstack((R, t.reshape(-1, 1)))
        pts1_h = cv2.convertPointsToHomogeneous(pts1).reshape(-1, 3).T  # 2D point to homo
        pts2_h = cv2.convertPointsToHomogeneous(pts2).reshape(-1, 3).T  # 2D point to homo
        # Triangulate points
        points_4d_h = cv2.triangulatePoints(P1, P2, pts1_h[:2], pts2_h[:2])
        # Convert to inhomogeneous coordinates
        points_3d = points_4d_h[:3, :] / points_4d_h[3, :]
        P = points_3d.T
        # 2D projections
        projected_points_h = P2 @ np.column_stack((P, np.ones(P.shape[0]))).T
        p_img2 = (projected_points_h[:2, :] / projected_points_h[2, :]).T
        return P, p_img2, P2
```

# 3. Experimental Result

In this section, we will present and discuss our experimental results for Structure from Motion.

## 3.1. Different algorithm of fundamental matrix

First, we need to discuss the algorithm used to compute the fundamental matrix. According to the requirements set by the teaching assistant, we used the **Normalized 8-Point Algorithm**. However, as shown in the epipolar line diagram (**Figure 1**), the lines appear to diverge, which is noticeably different from our expectation that each line should be parallel. This discrepancy leads to suboptimal image results. Detailed outcomes can be found in **Figure 2**, which displays the 3D model, **Figure 3**, which shows the mesh diagram, and **Figure 4** is texture mesh diagrams. Therefore, in subsequent experiments, we switched to using **RANSAC** to compute the fundamental matrix. This method is more robust and precise compared to the Normalized 8-Point Algorithm, as it is less affected by complex line structures.

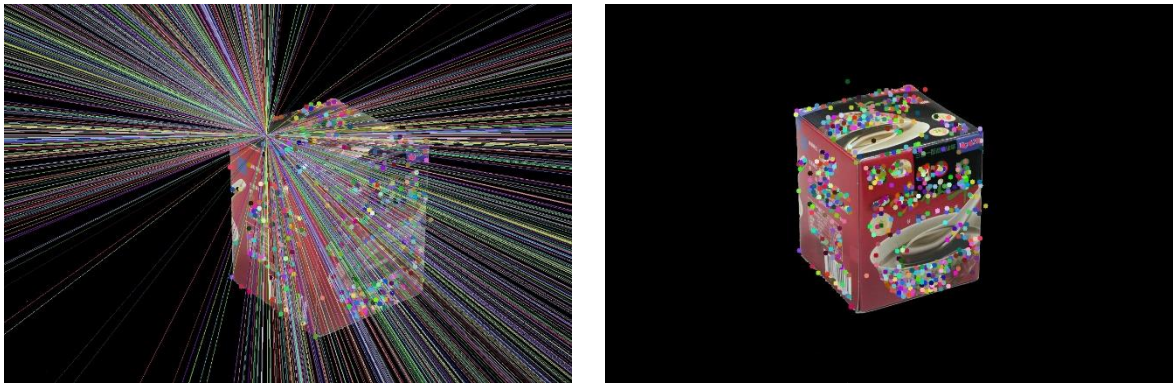

**Figure 1. The result of epipolar line**

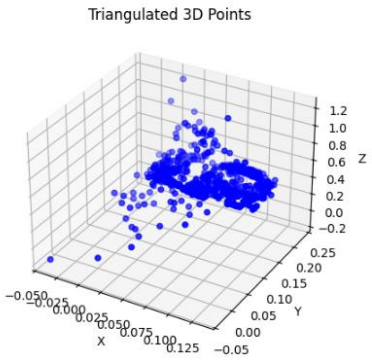


**Figure 2. The result of 3D model**

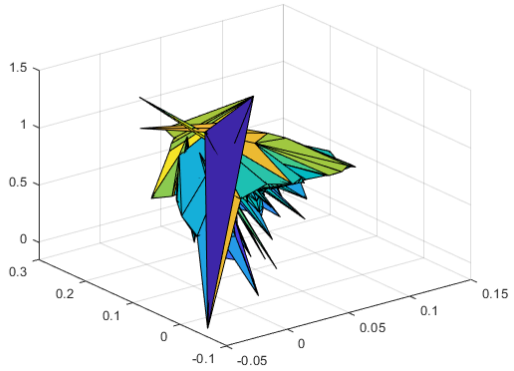

**Figure 3. The result of mesh**

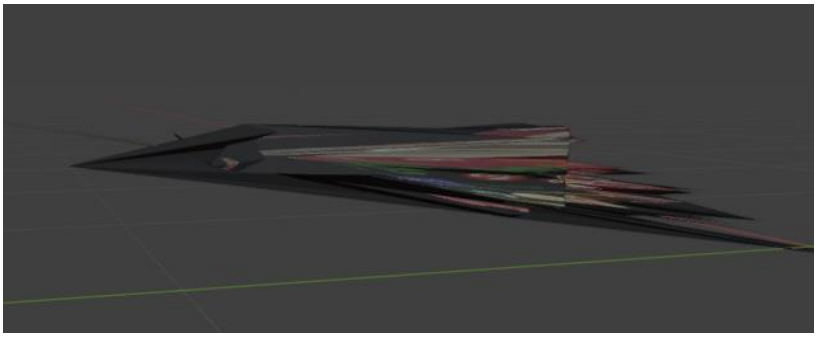

**Figure 4. The result of texture mesh.**

## 3.2. Result of TA Data - Mesona1 & Mesona2

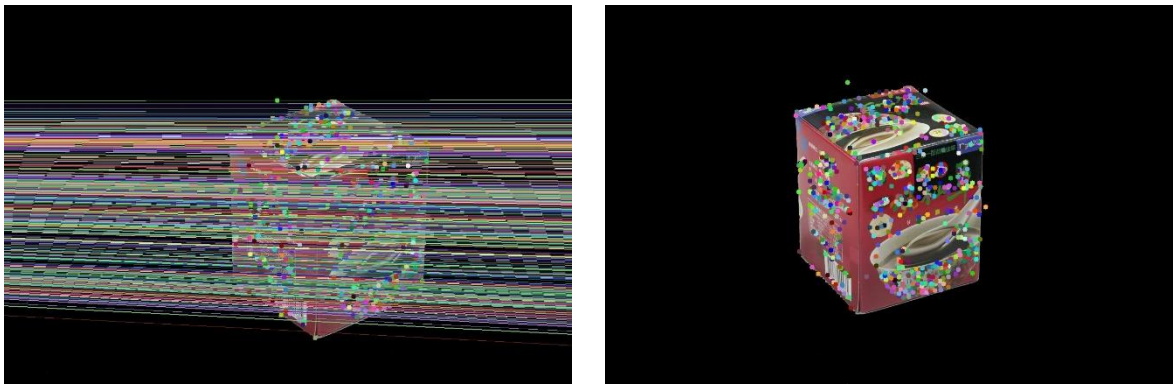

**Figure 5. The result of epipolar line**

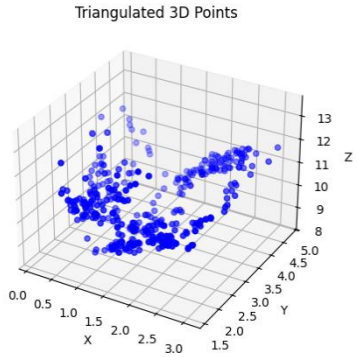


**Figure 6. The result of 3D model**

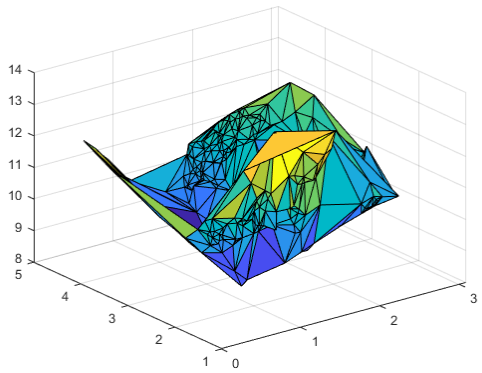

**Figure 7. The result of mesh**

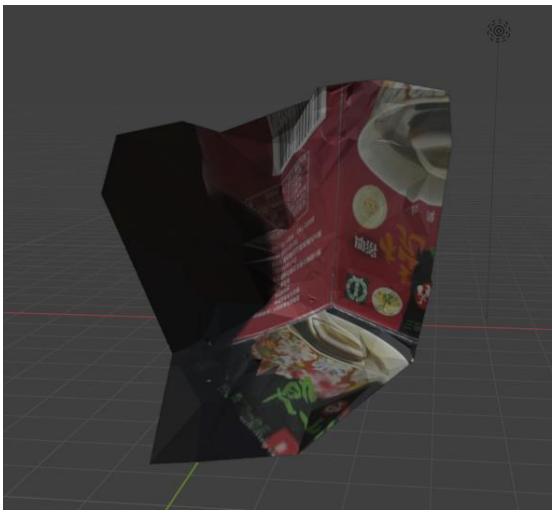

**Figure 8. The result of texture mesh.**

## 3.3. Result of TA Data - Statue1 & Statue2

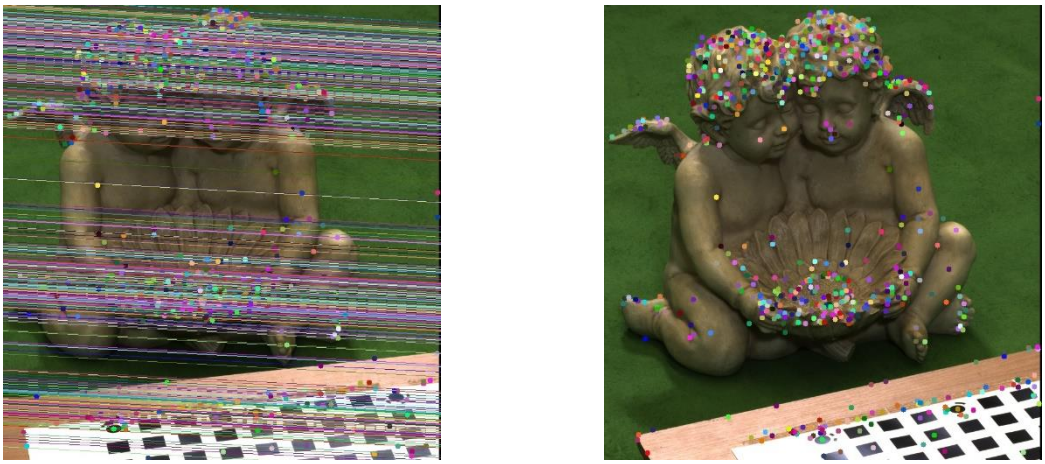

**Figure 9. The result of epipolar line**

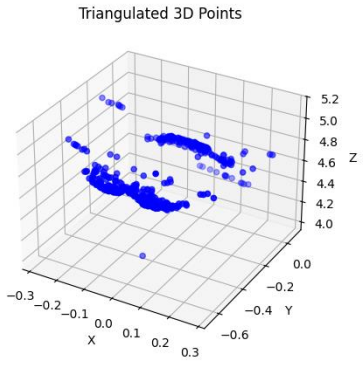


**Figure 10. The result of 3D model**

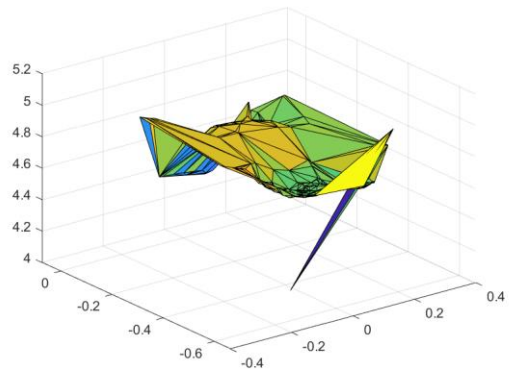

**Figure 11. The result of mesh**

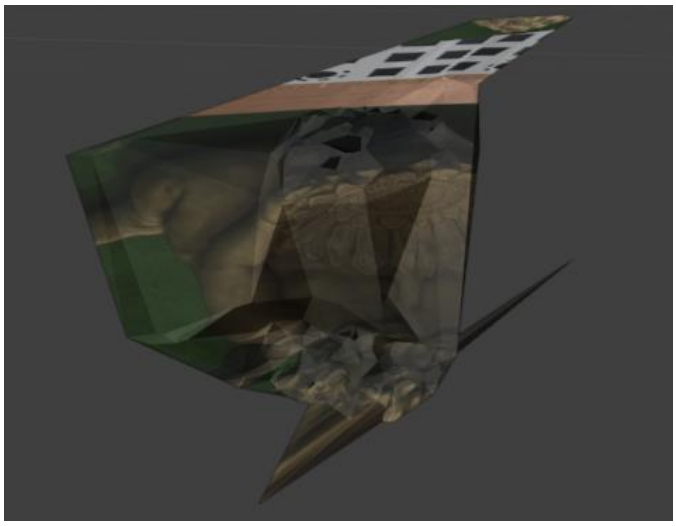

**Figure 12. The result of texture mesh.**

## 3.4. Result of Our Data – Trash1 & Trash2

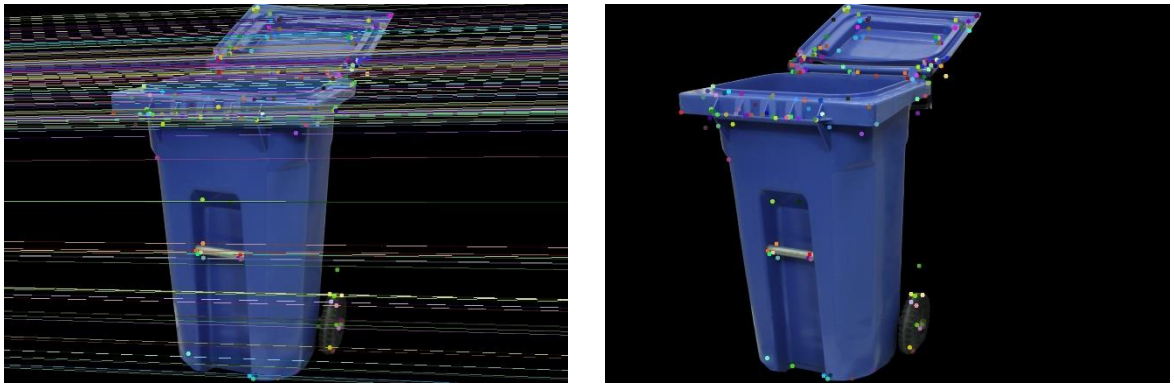

**Figure 13. The result of epipolar line**

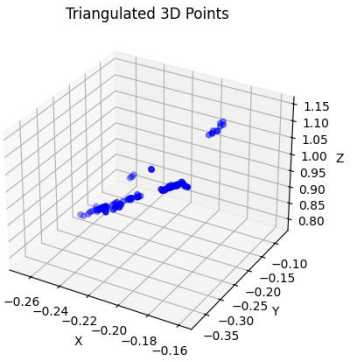


**Figure 14. The result of 3D model**

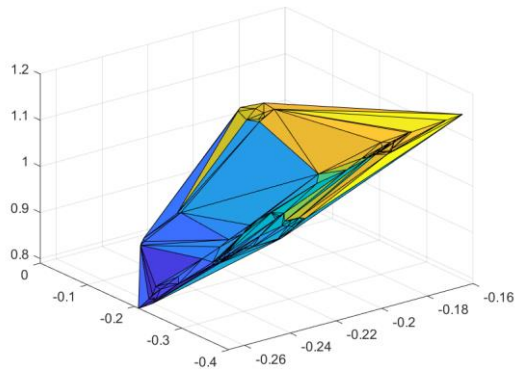


**Figure 15. The result of mesh**

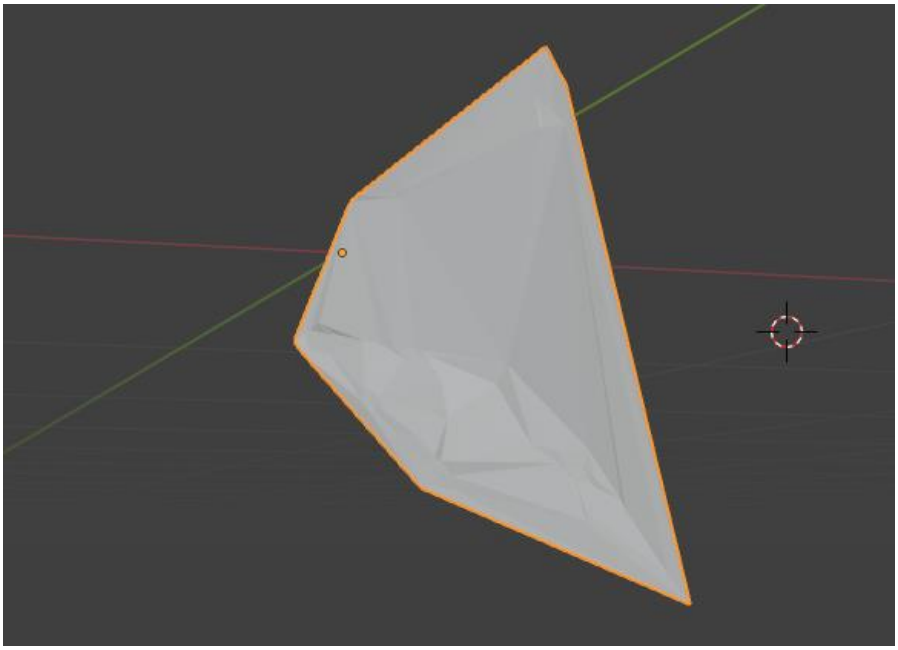

**Figure 16. The result of texture mesh.**

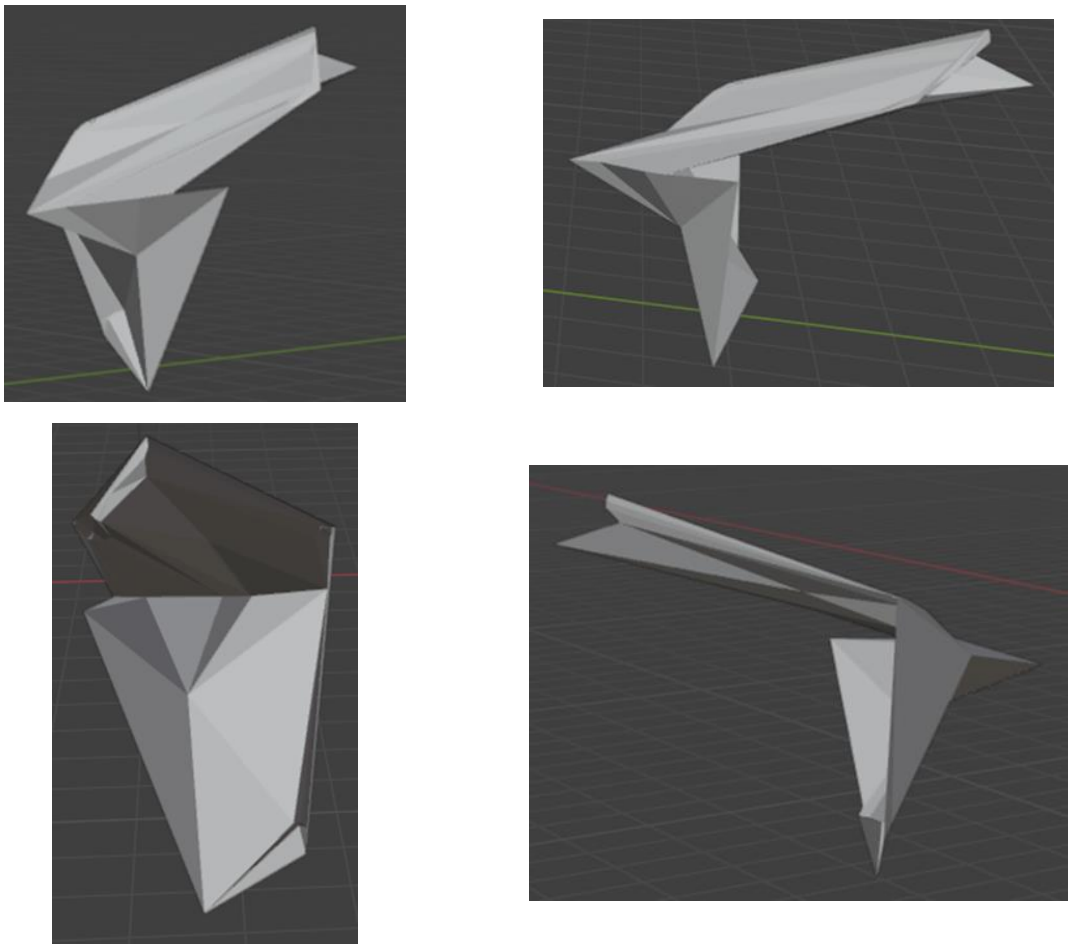

**Figure 17. The result of different viewing angles of the 3D model**

The experimental results indicate that the final 3D model is incomplete, primarily due to the provided image data capturing only a portion of the object rather than its full structure. This limitation prevents the reconstruction process from accurately representing the object in its entirety, leading to suboptimal results. The lack of comprehensive visual data, especially from angles that are not covered by the current dataset, significantly hinders the ability to create a precise and complete 3D model.

Therefore, providing additional images taken from various perspectives, including those that encompass all critical angles and details of the object, would likely enhance the reconstruction process. This improvement would ensure a more accurate representation of the object's geometry and contribute to the successful creation of a fully complete 3D model.

Besides, the suboptimal quality of the 3D model generated from our data can be attributed to the significantly fewer interest points in comparison to TA's data. This scarcity of interest points results in fewer epipolar lines, which negatively impacts the accuracy and robustness of the 3D reconstruction. Furthermore, the object in our dataset is smooth and uniform in color, lacking sufficient texture and distinctive features. This further constrains feature detection and matching, ultimately leading to a 3D model that performs worse than TA's.

# 4. Discussion

## 4.1. The Impact of RANSAC Threshold on Accurate Fundamental Matrix Estimation

During the implementation of Structure from Motion, we noticed that the epipolar lines did not align with the matching points in the stereo images. The failure case is in **Figure 18**. This issue significantly impacted the accuracy of the estimated fundamental matrix ($F$). We addressed the problem by analyzing the condition:

$$error = |x' F x| < threshold$$

If the threshold was set too high, many outliers to be classified as inliers, leading to an incorrect estimation of $F$. To resolve the issue, we adjusted the RANSAC threshold dynamically for each image. This ensured that the threshold was appropriately tuned to filter out outliers while retaining valid inliers, resulting in a more accurate estimation of $F$.

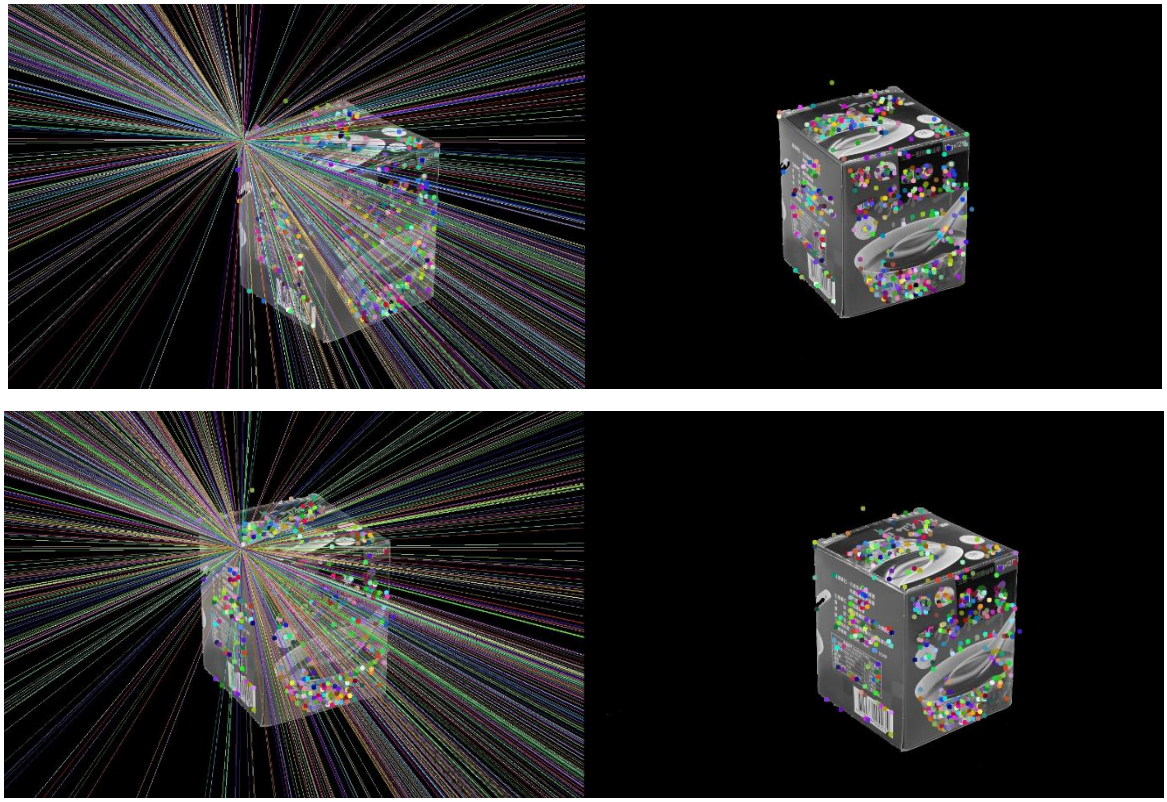

**Figure 18. The failure case of epipolar lines**

## 4.1. Evaluation of Feature Matching Methods in SfM

We tested different matching methods — BFMatcher, FLANN, KAZE, and AKAZE. BFMatcher showed stable and high inlier ratios across all datasets and RANSAC thresholds. On the other hand, KAZE and AKAZE performed well on certain datasets but lacked overall consistency. FLANN was more consistent than KAZE and AKAZE but still did not perform as well as BFMatcher.

Based on these results, BFMatcher is the most practical and reliable choice for feature matching in SfM pipelines. It works well across different datasets and stays accurate under changing RANSAC thresholds, making it good for estimating the fundamental matrix F. KAZE and AKAZE can perform better in some specific cases, but they depend too much on dataset-specific settings. Therefore, in section 2.1, we use the Brute-Force Matcher.

| | **Inlier Ratio (ratio0.75, threshold0.5)** | | |
|---|---|---|---|
| | **Mesona** | **Statue** | **Our Data** |
| **BFMatcher** | 0.05 | 0.04 | 0.03 |
| **FLANN** | 0.02 | 0.03 | 0.03 |
| **KAZE** | 0.04 | 0.01 | 0.01 |
| **AKAZE** | 0.02 | 0.03 | 0.02 |

| | Inlier Ratio (ratio0.75, threshold0.05) | | |
|---|---|---|---|
| | **Mesona** | **Statue** | **Our Data** |
| **BFMatcher** | 0.08 | 0.05 | 0.03 |
| **FLANN** | 0.05 | 0.04 | 0.05 |
| **KAZE** | 0.07 | 0.02 | 0.01 |
| **AKAZE** | 0.08 | 0.02 | 0.02 |

## 5. Conclusion

In this homework, we explored the implementation and evaluation of the Structure from Motion. Our result demonstrated the quality and completeness of the generated models were heavily influenced by the quality of the input data and the presence of distinctive features in the images. In the discussion, we found the importance of choosing an appropriate threshold. Setting the threshold too high or too low resulted in worse result. Additionally, the experimental results highlighted the necessity of diverse and detailed image data to achieve complete and accurate 3D reconstructions. Sparse and featureless datasets led to limited interest points, negatively impacting the accuracy of epipolar geometry and 3D point triangulation. Implementing SfM ourselves has been both an engaging challenge and a valuable learning experience, as it provided us with deeper insights into the complexities of 3D reconstruction and the importance of each step in the pipeline.

## 6. Work Assignment Plan

| | |
|---|---|
| 廖怡誠 | 1. implement code<br>2. write the report - the **Implement Procedure** and **Experimental Result** part |
| 李品妤 | 1. implement code<br>2. write the report - the **Discussion** part, **Experimental Result** part and **Conclusion** part |
| 林芷萱 | 1. implement code<br>2. write the report - the **Introduction** part and **Experimental Result** part |

## 7. Command

These are our command. If you need to know more detailed input parameters, you can see the corresponding code.

| Our implementation |
| --- |
| python main.py [--data1] [--data2] [--img_name] [--fundamental_matrix_algorithm] |

```
parse = argparse.ArgumentParser()
parse.add_argument('-d1', '--data1', default= "./my_data/trash1.png", type= str, help= 'path of image1')
parse.add_argument('-d2', '--data2', default= "./my_data/trash2.png", type= str, help= 'path of image2')
parse.add_argument('-N', '--img_name', default= "trash", type= str,
                   choices= ["Mesona", "Statue", "trash"], help= 'calibration parameter')
parse.add_argument('-F', '--fundamental_matrix_algorithm', default= "RANSAC",
                  type= str, choices=["RANSAC", "nor_point8"], help= 'fundamental_matrix_algorithm')
```

## 8. File description

- **main.py:** The main code we mainly implement.
- **compare_feature_matching.py**: The code is for experiment.
- **checkVisble.m**: TA provided.
- **obj_main.m**: TA provided.
- **generate_model.m**: The code called obj_main.m and read the parameter.
- **data:** TA provided.
- **my_data:** our data.
- **result:** our experiment results contain images, obj files, and mtl files.