Homework 4: Structure from Motion

I-Sheng Fang
The Master's Degree Program in Robotics
National Yang Ming Chiao Tung University
isfang.gdr09g@nctu.edu.tw
Implementation, Discussion

Hsiang-Chun Yang
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw
Implementation, Experiment, Discussion.

Yu-Lin Yeh
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
s995503@gmail.com
Experiment, Discussion.

Abstract

In this assignment, we implement the Structure from Motion (SfM). The main procedures of our work are getting the interest points, finding out correspondence, estimating the fundamental matrix, finding out the most appropriate solution, and applying triangulation to get 3D points. Besides using the images from TAs, we are also asked to try the images taken by our own camera.

1 Introduction

Three-dimensional reconstruction is a popular field in recent years. In this homework, we implement Structure from Motion (SfM)[1]. SfM reconstructs the 3D scene by some useful methods, such as SIFT[2], RANSAC[3], and bundle adjustment[4]. Through these algorithms, we can find out the camera relation by predicting the second camera matrix and reconstruct the 3D points by 2D image pairs.

2 Implementation Procedure

2.1 Feature extraction/matching

First, we need to find the correspondence across two images. Here, we use SIFT provided by OpenCV to extract the key points and their descriptors. Then, we search for the best match key point pairs between two images by their ratio distance. The implementation of feature extraction and matching are as follows.

```
def sift(img):
    sift_descriptor = cv2.SIFT_create()
    kp, des = sift_descriptor.detectAndCompute(img, None)
    return kp, des

def matching(des1, des2):
    matches = []
    for i in range(des1.shape[0]):
```

```
        dis = [(norm(des1[i]-des2[j]), j, i) for j in range(des2.shape[0])]
        dis = sorted(dis, key=lambda x: x[0])
        dms = []
        for t in range(2):
            dm = cv2.DMatch(
                    _distance=dis[t][0],
                    _trainIdx=dis[t][1],
                    _queryIdx=dis[t][2])
            dms.append(dm)
        matches.append(dms)
    return matches

def match_feature(img1, kp1, des1, img2, kp2, des2, ratio, results_dir):
    matches = matching(des1, des2)
    good_match = []
    for m, n in matches:
        if m.distance/n.distance < ratio:
            good_match.append(m)
    match_kp1 = []
    match_kp2 = []
    for m in good_match:
        match_kp1.append(kp1[m.queryIdx].pt)
        match_kp2.append(kp2[m.trainIdx].pt)
    match_kp1 = np.array(match_kp1)
    match_kp2 = np.array(match_kp2)
    return match_kp1, match_kp2
```

2.2 Fundamental matrix

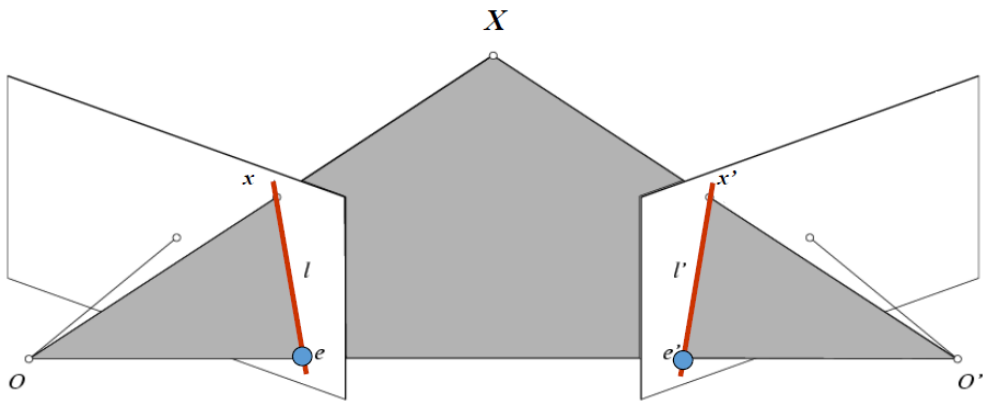


Figure 1

As the Figure 1 shown, given a fundamental matrix F, the epipolar line l in left image can be obtained by $F^T x'$ and the epipolar line l' in right image can be obtained by Fx. The fundamental matrix F can be solved by the following equation.

$$x^T F x' = 0 \tag{1}$$

The equation can be further expanded to the following form.

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf31 + yf_{32} + f_{33} = 0 \tag{2}$$

Before solving the fundamental matrix, we have to normalize the image coordinates x and x' to eliminate the orders of magnitude difference. With the help of normalization, we can have a more accurate fundamental matrix. Here, we use 8-point algorithm to find out the fundamental matrix. Each time, we sample 8 pairs of match key points and form the following equation.

$$\begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_8x_8 & x'_8y_8 & x'_8 & y'_8x_8 & y'_8y_8 & y'_8 & x_8 & y_8 & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = 0 \tag{3}$$

Since this equation has the form $Ax = 0$, we can solve it by applying singular value decomposition (SVD). Notice that, after solving the fundamental matrix, we have to denormalize it since we have normalized the image coordinates before. The implementation of 8-points algorithm and normalization are as follows. We also apply RANSAC algorithm to improve the quality of fundamental matrix.

```
def normalize_ext(pts):
    center = np.mean(pts, axis=0)
    scale = np.sqrt(2) / norm(center-pts, axis=1).mean()
    T = np.array([
            [scale, 0, -scale*center[0]],
            [0, scale, -scale*center[1]],
            [0, 0, 1]
        ])
    pts_ext = np.hstack((pts, np.ones((pts.shape[0], 1))))
    norm_pts_ext = (T @ pts_ext.T).T
    return norm_pts_ext, T

def run_8_point(match_kp1, match_kp2):
    assert match_kp1.shape[0]==match_kp2.shape[0] and match_kp1.shape[0]==8
    norm_kp1, T1 = normalize_ext(match_kp1)
    norm_kp2, T2 = normalize_ext(match_kp2)
    A = []
    for i in range(8):
        A.append([
            norm_kp2[i,0] * norm_kp1[i,0],
            norm_kp2[i,0] * norm_kp1[i,1],
            norm_kp2[i,0] * norm_kp1[i,2],
            norm_kp2[i,1] * norm_kp1[i,0],
            norm_kp2[i,1] * norm_kp1[i,1],
            norm_kp2[i,1] * norm_kp1[i,2],
            norm_kp2[i,2] * norm_kp1[i,0],
            norm_kp2[i,2] * norm_kp1[i,1],
            norm_kp2[i,2] * norm_kp1[i,2]
```

```
        ])
    A = np.array(A)

    U, S, V = svd(A)
    F = V[-1].reshape(3, 3)

    U, S, V = svd(F)
    S[2] = 0
    F = U @ np.diag(S) @ V

    F = T2.T @ F @ T1
    F /= F[2,2]
    return F
```

2.3 Epipolar line

We can use $F^T x'$ compute the epipolar line in the first image, where F is the fundamental matrix and x' is the image coordinates of the second image. In contrast, the epipolar line in the second image can be computed with Fx. Each line $ax + by + c = 0$ will be presented in the form of (a, b, c). The implementation of computing and drawing epipolar line are as follows.

```
def compute_epipolar_line(F, pts):
    pts_ext = np.hstack((pts, np.ones((pts.shape[0], 1))))
    lines = (F @ pts_ext.T).T
    n = np.sqrt(np.sum(lines[:,:2]**2, axis=1)).reshape(-1, 1)
    return lines / n * -1

def drawlines(img1, pts1, img2, pts2, lines):
    w = img1.shape[1]
    new_img1 = np.array(img1)
    new_img2 = np.array(img2)
    pts1 = pts1.astype(np.int32)
    pts2 = pts2.astype(np.int32)
    for coef, pt1, pt2 in zip(lines, pts1, pts2):
        color = tuple(np.random.randint(0, 255, 3).tolist())
        x0, y0 = map(int, [0, -coef[2]/coef[1]])
        x1, y1 = map(int, [w, -(coef[2]+coef[0]*w)/coef[1]])
        new_img1 = cv2.line(new_img1, (x0, y0), (x1, y1), color, 1)
        new_img1 = cv2.circle(new_img1, tuple(pt1[0:2]), 5, color, -1)
        new_img2 = cv2.circle(new_img2, tuple(pt2[0:2]), 5, color, -1)
    return new_img1, new_img2
```

2.4 Essential matrix

We use the camera matrix K and fundimantal martrix F to get the essential matrix E with the following equation.

$$E = K_1^T F K_2 \tag{4}$$

```
def essential_matrix(K1, K2, F):
    E = K1.T @ F @ K2
    U,S,V = np.linalg.svd(E)
    m = (S[0]+S[1]) / 2
    E = U @ np.diag((m, m, 0)) @ V
    return E
```

2.5 Solution of second camera matrix

After we get the essential matrix E, the second camera matrix P_2 could have four possible solutions. We first use SVD to get U and V. The essential matrix E has two part $[R|t]$. The possible rotation matrix R could be UWV or UW^TV and the translation vector could be $\pm u_3$.

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

And the four possible solutions of the second camera matrix are following equations.

$$\begin{aligned} P_2 &= [UWV|u_3] \\ P_2 &= [UWV|-u_3] \\ P_2 &= [UW^TV|u_3] \\ P_2 &= [UW^TV|-u_3] \end{aligned}$$

```
def four_possible_solution_of_second_camera_matrix(E):
    U, S, V = np.linalg.svd(E)
    if np.linalg.det(U @ V):
        V = -V
    W = np.array([[0,-1, 0],
                  [1, 0, 0],
                  [0, 0, 1]])
    R = U @ W @ V
    t = U[:,2:]
    P2_0 = np.hstack((R, t))
    P2_1 = np.hstack((R, -t))
    R = U @ W.T @ V
 stack((R, t))
    P2_3 = np.hstack((R, -t))

    return [P2_0, P2_1, P2_2, P2_3]
```

2.6 Triangulation

After we get the second camera matrix, we can use triangulation to predict the reconstruct 3D points. We choose the second camera matrix P_2 which can reconstruct most in front points as the most possible second camera matrix P_2.

For triangulation, we need key point pair x_1, x_2 and the camera matrices P_1, P_2 first.

$$x_1 = w \begin{bmatrix} u_1 \\ v_1 \\ 1 \end{bmatrix}, x_2 = w \begin{bmatrix} u_2 \\ v_2 \\ 1 \end{bmatrix}$$

$$P_1 = [p_{11} \quad p_{12} \quad p_{13}], P_2 = [p_{21} \quad p_{22} \quad p_{23}]$$

Second, we create the matrix A.

$$A = \begin{bmatrix} u_1 p_{13} - p_{11} \\ v_1 p_{13} - p_{12} \\ u_2 p_{23} - p_{21} \\ v_2 p_{23} - p_{22} \end{bmatrix}$$

Then, we decomposition A with SVD, i.e. $A = USV$. The reconstruct points X are $V(:, end)$.

Now, the problem will be how to know the point is in front of camera or not. We know the camera center $C = -R^T t$. We only need to compute the $(X - C)\dot{R}_3^T$. If this value is positive, this point X is in front of camera.

```
def triangulation(x1, x2, P1, P2):
    pred_pt = np.ones((x1.shape[0], 4))
    C = P2[:,:3].T @ P2[:,3:]
    infront_num = 0
    for i in range(x1.shape[0]):
        A = np.asarray([
            (x1[i,0] * P1[2,:].T - P1[0,:].T),
            (x1[i,1] * P1[2,:].T - P1[1,:].T),
            (x2[i,0] * P2[2,:].T - P2[0,:].T),
            (x2[i,1] * P2[2,:].T - P2[1,:].T)
        ])
        U, S, V = np.linalg.svd(A)
        pred_pt_i = V[-1] / V[-1,3]
        pred_pt[i,:] = pred_pt_i
        if np.dot((pred_pt_i[:3]-C.reshape(-1)), P2[2,:3]) > 0:
            infront_num += 1
    return pred_pt, infront_num
```

3 Experimental Result

We test our implementation with the images provided by TA and we shot. Figure 2 shows the key points results with each images. Figure 3 shows the epipolar lines results with each images. Figure 4 shows the feature matching results with each image pairs. Figure 5 shows the final structure form the motion results as 3D reconstruction points, 3D mesh and the 3D model.

4 Discussion

We discover that the images taken by our camera cannot construct the ideal 3D model every time. In other words, the final result cannot be reproduced. Combining the possible reason, we thought the main reason is the randomness in the SfM procedure. When we estimate the the fundamental matrix across images, we sample 8 pairs each time and compute the fundamental matrix on them. The failure case is in Figure 6

We also discover that the background removal is important in SfM. As the Figure 7 shown, without removing background, SIFT will extract key points in the background and cause miss matching.

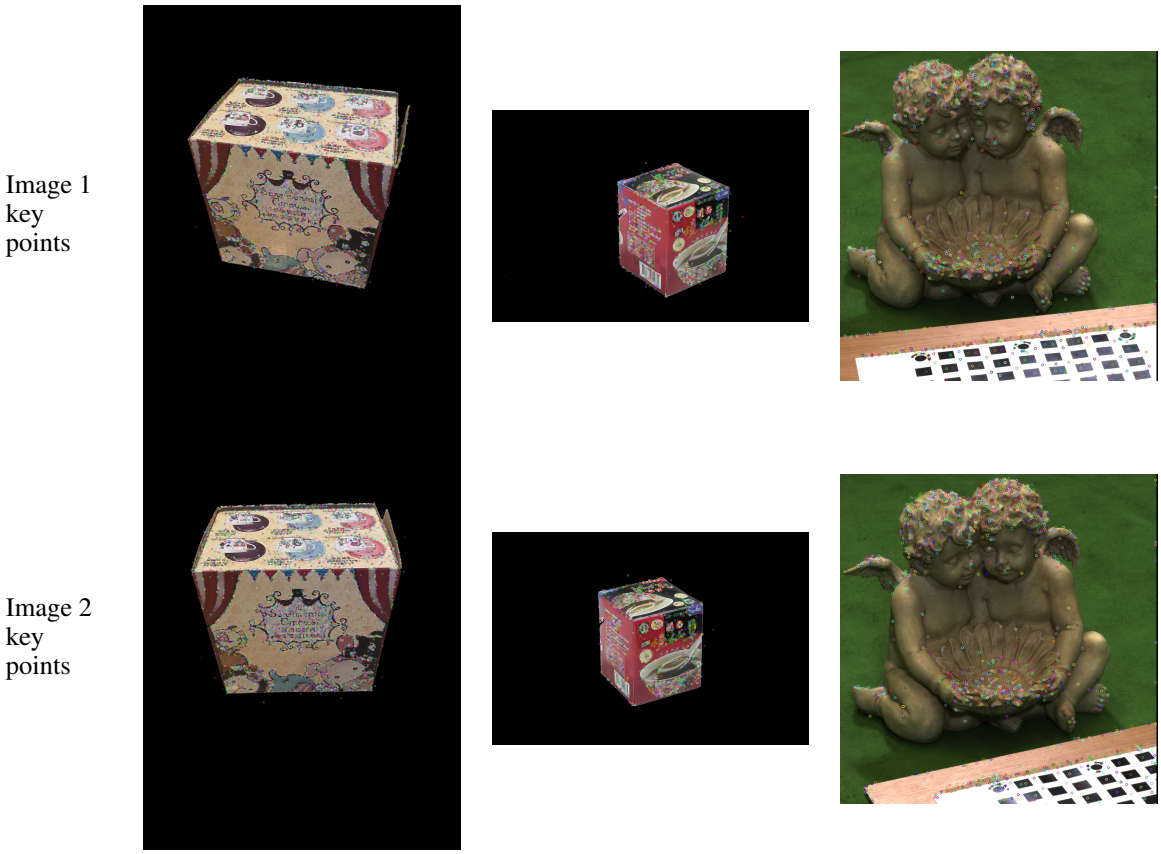


Figure 2: The key points of each image

5 Conclusion

Through the analysis of experiment results, we compare the final 3D models reconstructed by different image pairs. We conclude that there are many reasons which will affect the results, such as image background and the way of estimating the fundamental matrix. In conclusion, this experience will inspire us for SfM research in the future.

References

[1] X. Jianxiong & O. Andrew &T. Antonion (2013) Sun3d: A database of big spaces reconstructed using sfm and object labels *Proceedings of the IEEE international conference on computer vision*

[2] P. Xunyu &L. Siwei (2010) Detecting image region duplication using SIFT features *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*

[3] K.Bernd &G.Andreas &L. Henning (2010) Visual odometry based on stereo image sequences with RANSAC-based outlier rejection scheme *2010 ieee intelligent vehicles symposium*

[4] F. Pascal (2000) Regularized bundle-adjustment to model heads from image sequences without calibration data*International Journal of Computer Vision*

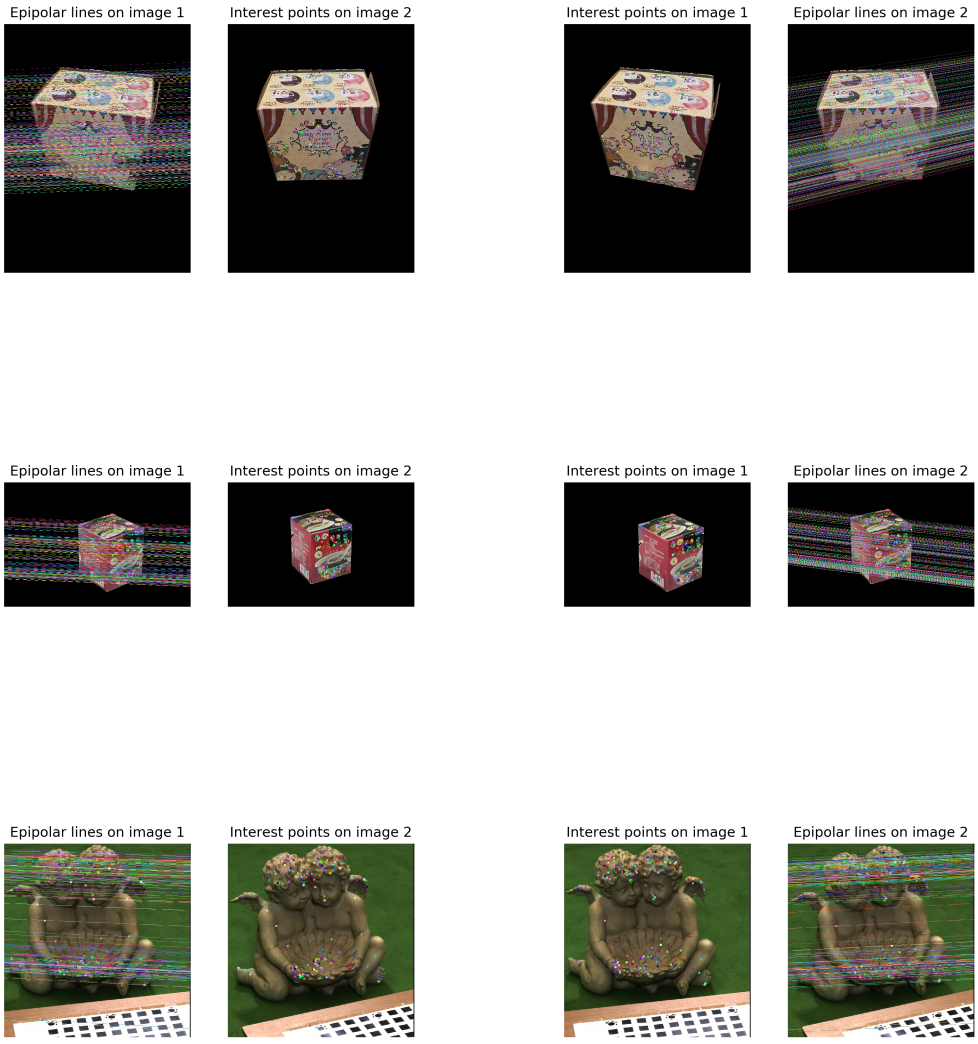


Image 1 Image 2

Figure 3: The epipolar lines of each image pairs

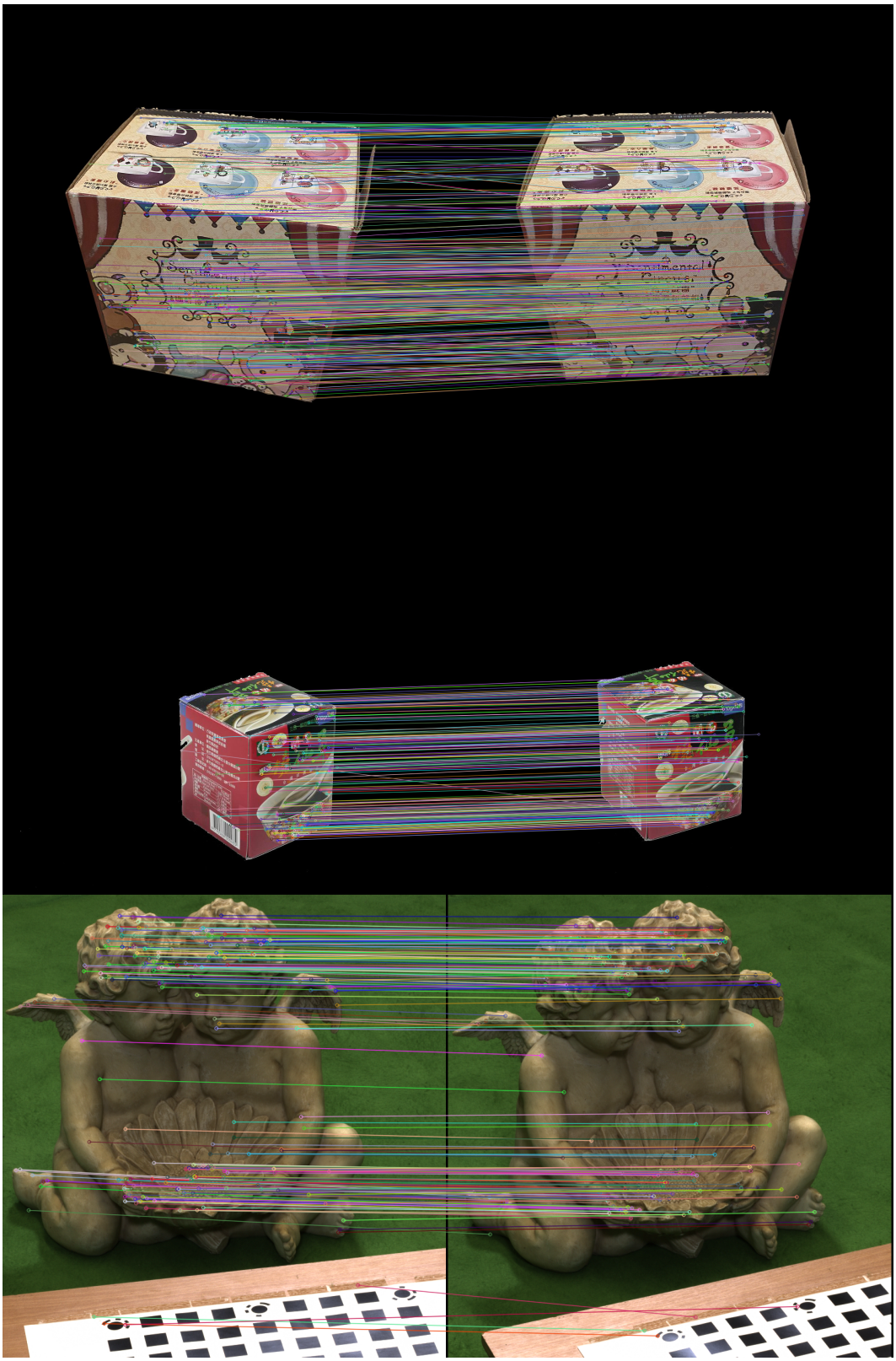


Figure 4: The feature matching results of each image pairs.

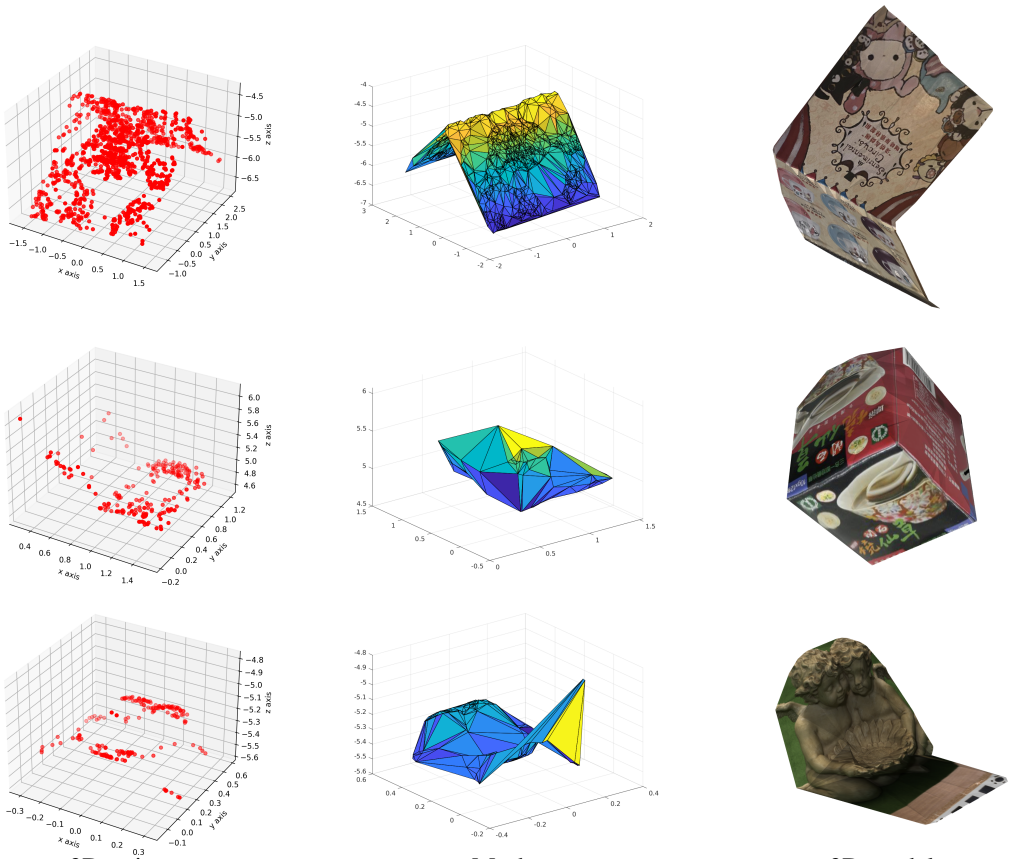


Figure 5: The structure from motion results of each image pairs

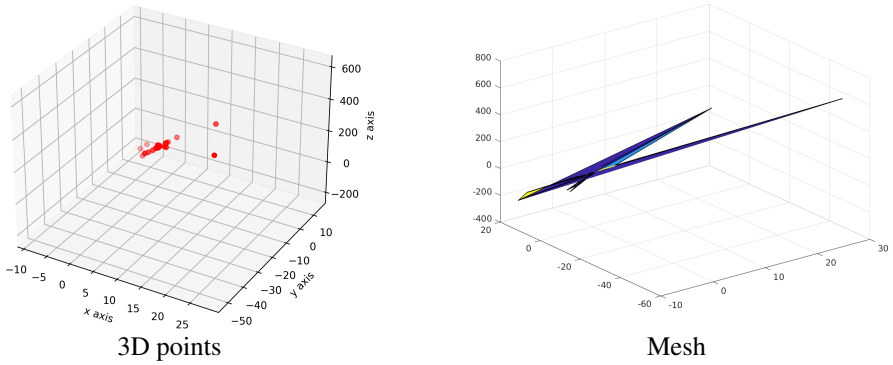
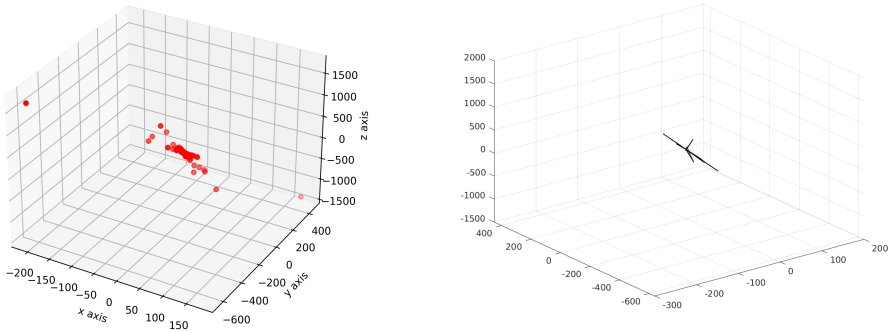


Figure 6: The failure case of statue images



3D points Mesh

Figure 7: The structure motion result without image background removal