CV HW4 Group 19 Report

0756079 陳冠聞, 0756138 黃伯凱

May 7, 2019

1 Introduction

Structure from motion (SfM) is a photogrammetric range imaging technique for estimating three-dimensional structures from two-dimensional image sequences that may be coupled with local motion signals.

In this homework, we will implement SfM in python. Except for the keypoints detection, camera calibration, matrix operation and image I/O are using the packages of OpenCV and NumPy, all the other functions are implemented by ourselves.

2 Implementation

The goal of SfM is to estimate 3-D models from a 2-D images sequences with motion signals. To be able to estimate the 3-D coordinate, we must have the **Fundamental Matrix** and **Essential Matrix**, then do the **triangulation**.

For the estimation of fundamental matrix, we have to have the correspondences across different image first. In this homework, we use the feature detector implemented in OpenCV, and do feature matching on the keypoints found by the detector to get the correspondences. After that, we can start to estimate fundamental matrix, and we explain the algorithm in detail below.

2.1 Estimation of Fundamental Matrix

2.1.1 What is Fundamental Matrix

Assume there are two images I and $I^{'}$. An image point x in the left image corresponds to epipolar line $l^{'}$ in right image. Fundamental matrix maps from a point in one image to a line in the other image. That is, $l^{'} = Fx$ and $l = F^{T}x^{'}$. Since x and $x^{'}$ correspond to the same 3-D point X, we have $x^{'T}Fx = 0$.

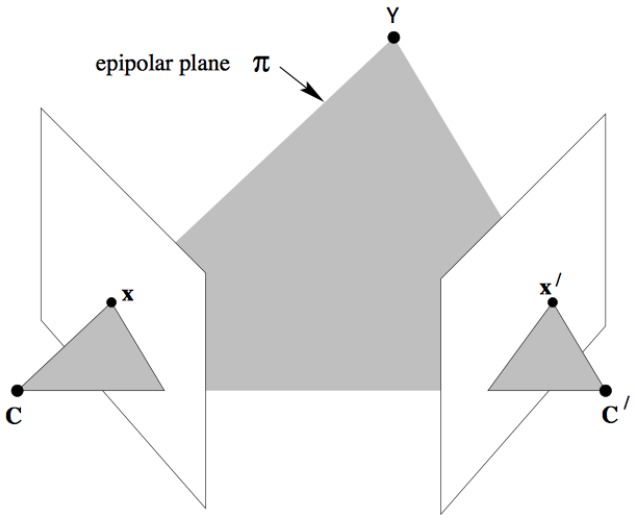


Figure 1: Epipolar Geometry

We can also explain it in math, let C and $C^{'}$ be two different cameras. Y is a 3D point, X and $X^{'}$ are the 3D coordinate represented by camera C and $C^{'}$ corresponding to Y respectively. x and $x^{'}$ are the pixel coordinate on each image corresponding to X and $X^{'}$. K and $K^{'}$ are the camera matrices. Clearly, the points $X, x, x^{'}, C, C^{'}$ are coplanar, it is a so-called **Epipolar Plane**.

Since X and $X^{'}$ are the coordinate of Y in the coordinate of both cameras, we know that

$$x = KX, \quad x^{'} = K^{'}X^{'} \tag{1}$$

The transformation of the coordinate system between these cameras can be represented by a **Rotation Matrix R** and a **Transformation Matrix t**

$$X^{'} = RX + t \tag{2}$$

From Eq.1 and Eq.2, we obtain

$$x^{'} = K^{'}[RX + t] \tag{3}$$

Multiply $K^{'-1}$ on both sides

$$K^{'-1}x^{'} = RX + t \tag{4}$$

To remove t from the equation, we apply cross product t on both sides. ($t \times t = 0$)

$$t \times (K^{'-1}x^{'}) = t \times (RX) + 0 \tag{5}$$

Due to the equivalence of vector cross product and matrix multiplication we mentioned in Appendix 7.1, we can change t into a 3×3 matrix and rewrite Eq.5 again :

$$[t_{\times}]K^{'-1}x^{'} = [t_{\times}]RX \tag{6}$$

For any vector $\vec{a}$ and $\vec{b}$, $\vec{a} \cdot (\vec{b} \times \vec{a}) = 0$. Let $\vec{a} = K^{'-1}x^{'} = RX$ and $\vec{b} = [t_{\times}]$, we obtain

$$\vec{a}^{T}[t_{\times}]RX = (K^{'-1}x^{'})^{T}[t_{\times}]RX \tag{7}$$

$$= x^{'T}K^{'-T}[t_{\times}]RX \tag{8}$$

$$= 0 \tag{9}$$

Finally, from Eq.1, we know $X = K^{-1}x$, so we can rewrite Eq.9

$$x^{'T}K^{'-T}[t_{\times}]RX = x^{'T}K^{'-T}[t_{\times}]RK^{-1}x = 0 \tag{10}$$

As a result, we obtain $x^{'T}K^{'-T}[t_{\times}]RK^{-1}x = 0$. Let $F = K^{'-T}[t_{\times}]RK^{-1}$ we got $x^{'T}Fx = 0$. F is the **Fundamental Matrix** and $x^{'T}Fx = 0$ is the **Epipolar Constraint**.

2.1.2 How to compute Fundamental Matrix

We can write Epipolar Constraint in matrix form:

$$\begin{bmatrix} x_i^{'} & y_i^{'} & 1 \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} = 0 \tag{11}$$

If we have N correspondences, we can write the Equation 11 to

$$A\vec{f} = \begin{bmatrix} x_1^{'}x_1 & x_1^{'}y_1 & x_1^{'} & y_1^{'}x_1 & y_1^{'}y_1 & y_1^{'} & x_1 & y_1 & 1 \\ x_2^{'}x_2 & x_2^{'}y_2 & x_2^{'} & y_2^{'}x_2 & y_2^{'}y_2 & y_2^{'} & x_2 & y_2 & 1 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... \\ x_N^{'}x_N & x_N^{'}y_N & x_N^{'} & y_N^{'}x_N & y_N^{'}y_N & y_N^{'} & x_N & y_N & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ ... \\ f_{33} \end{bmatrix} = \vec{0} \tag{12}$$

Because the rank of A is at most 8, so we can estimate the normalized $\vec{f}$ use normalized **8-points algorithm**, which is find 8 correspondences and normalize them, and use the **SVD decomposition** to get the least square solution of Equation 12. But since the rank of A is at most 8, we need to enforce the internal constraint by using **Rank enforcement**, which is to apply another SVD on F, then set the least significant singular value to zero and restore it back.

$$F = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^T \tag{13}$$

$$\tilde{F} = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \tag{14}$$

In addition, because the correspondences might contain noise, we estimate the fundamental matrix by using **RANSAC** algorithm, which remove the outliers by selecting the one with most inliers.

```
def compute_fundamental(x1_sample, x2_sample, T1, T2):
    A = []
    num_points = x1_sample.shape[1]
    for i in range(num_points):
        x1 = x1_sample[:, i]
        x2 = x2_sample[:, i]
        A.append([x1[0]*x2[0], x1[1]*x2[0], x1[2]*x2[0],
                  x1[0]*x2[1], x1[1]*x2[1], x1[2]*x2[1],
                  x1[0]*x2[2], x1[1]*x2[2], x1[2]*x2[2]])

    A = np.array(A)
    U,S,V = np.linalg.svd(A)
    F = V[-1].reshape(3,3)
    U,S,V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
    return F
```

Figure 2: Code Snippet for Fundamental Matrix

2.2 Epipolar Line

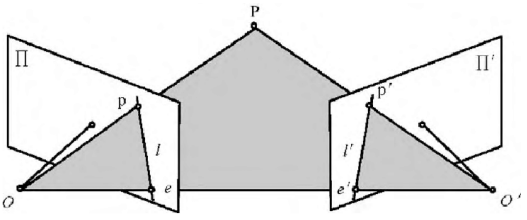


Figure 3: Epipolar Line

As we mentioned in last section, $l^{'} = Fx$ and $l = F^{T}x^{'}$. l and $l^{'}$ are the epipolar lines corresponding to x and $x^{'}$, while F is the fundamental matrix.

Epipolar line helps us understand the relative position between cameras and object. It is very easy to implement since we only have to multiply each 2D point by fundamental matrix.

```python
def plot_epipolar_line(img1, img2, x1, x2, F1, F2, imgName, num_lines=30):
    """
    Plot the epipole and epipolar line F*x=0 in an image.
    F is the fundamental matrix,
    x is a point in the other image.
    """
    def draw_lines(img, ax, pts, x, y, title):
        ax.axis('off'), ax.imshow(img), ax.set_title(title)
        for idx in range(num_lines):
            y_ = y[:, idx]
            crop = np.where((y_>=0) & (y_<img.shape[0]))
            y_ = y_[crop]
            x_ = x[crop]
            ax.plot(x_, y_, linewidth=1)
            ax.scatter(pts[0, idx], pts[1, idx], marker='x')

    def draw_points(img, ax, pts, title):
        ax.axis('off'), ax.imshow(img), ax.set_title(title)
        for idx in range(num_lines):
            ax.scatter(pts[0,idx], pts[1,idx], marker='x')

    m,n = img1.shape[:2]
    line1 = np.dot(F2,x2[:, :num_lines])
    line2 = np.dot(F1,x1[:, :num_lines])

    x = np.linspace(0,n,100)

    # au + bv + c = 0
    # v = (-au-c)/b
    y1 = np.array([(line1[0, :]*xx+line1[2, :])/(-line1[1, :]) for xx in x])
    y2 = np.array([(line2[0, :]*xx+line2[2, :])/(-line2[1, :]) for xx in x])

    f, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2,2, figsize=(15,10))
    draw_points(img1, ax1, x1, title='Left Image')
    draw_lines(img2, ax2, x2, x, y2, title='Right Image')
    draw_lines(img1, ax3, x1, x, y1, title='Left Image')
    draw_points(img2, ax4, x2, title='Right Image')
    plt.savefig(("../res/EpiploarLine_%s.png") %(imgName))
    plt.show()
```

Figure 4: Code Snippet for Epipolar Line

2.3 Estimate Essential Matrix

2.3.1 What is Essential Matrix

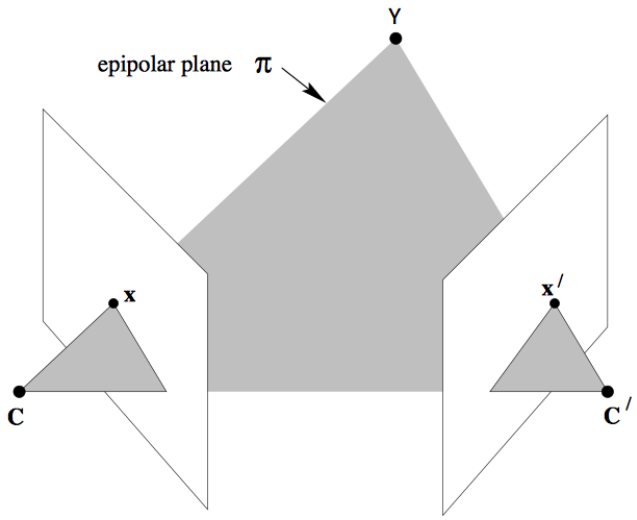


Figure 5: Epipolar Geometry

Since points $X, x, x^{'}, C, C^{'}$ are coplanar, we obtain the equation

$$\vec{Cx} \cdot (\vec{CC'} \times \vec{C'x'}) = 0 \tag{15}$$

let $\vec{CC'}$ be transformation vector $\vec{t}$ between camera C and $C^{'}$. Also, let $\vec{Cx}, \vec{C'x'}$ be homogenous vectors $\vec{p}$ and $\vec{p^{'}}$

$$\vec{Cx} = \vec{p} = [x, y, 1]^T_C, \quad \vec{C'x'} = \vec{p^{'}} = [x^{'}, y^{'}, 1]^T_{C'} \tag{16}$$

Notice that $\vec{p}$ and $\vec{p^{'}}$ are based on the coordinate system of camera C and $C^{'}$ respectively. If we want to express $\vec{p^{'}}$ in camera C's coordinate, we have to rotate it by the rotation matrix between C and $C^{'}$. So we have to multiply $\vec{p^{'}}$ by **rotation matrix R**. Now we can rewrite Eq.15 into :

$$\vec{p} \cdot (\vec{t} \times R\vec{p^{'}}) = 0 \tag{17}$$

In addition, due to the equivalence of vector cross product and matrix multiplication we mentioned in Appendix 7.1, we can change $\vec{t}$ into a 3×3 matrix

and rewrite Eq.17 again :

$$\vec{p} \cdot [t]_\times \cdot R\vec{p'} = 0 \tag{18}$$

Finally, let $E = [t]_\times R$, which E is the **Essential Matrix**.

2.3.2 How to compute Essential Matrix

According to Eq.10, we know $F = K'^{-T}[t_\times]RK^{-1}$. We can find out that Essential Matrix is quite similar to Fundamental Matrix. The relationship between F and E is

$$E = K'^{T}(K'^{-T}[t_\times]RK^{-1})K \tag{19}$$

$$= K'^{T}FK \tag{20}$$

That is, if we have camera matrix and Fundamental Matrix, we can simply compute the Essential Matrix.

```
def compute_essential(F, K1, K2):
    E = np.dot(K2.T, np.dot(F, K1))
    return E
```

Figure 6: Code Snippet for Essential Matrix

2.4 Estimate Extrinsic Matrix

2.4.1 Compute R and t

We need both intrinsic and extrinsic matrices of both cameras to reconstruct a 3D object. TA has given as the intinsic matrix, and we assume the extrinsic of Camera C is $[I \mid 0]$. Then we can use Essential Matrix to compute the second extrinsic since $E = [t_\times]R$.

$[t_\times]$ is rank 2 and R is a orthogonal matrix, so the Essential Matrix E is also rank 2. To maintain the rank, we apply SVD on E and make it to rank 2 by using matrices Z and W

$$Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{21}$$

$$\Sigma = ZW = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{22}$$

$$E = U\Sigma V^T = UZWV^T \tag{23}$$

U is orthogonal matrix, so $U^TU = I$.

$$E = UZU^TUWV^T \tag{24}$$

Let $S_1 = UZU^T$ and $R^{'} = UWV^T$, we can rewrite Eq.24

$$E = S_1R^{'} \tag{25}$$

Let's look as S_1 first

$$S_1 = UZU^T \tag{26}$$

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{12} & u_{22} & u_{32} \\ u_{13} & u_{23} & u_{33} \end{bmatrix} \tag{27}$$

$$= \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \begin{bmatrix} u_{12} & u_{22} & u_{32} \\ -u_{11} & -u_{21} & -u_{31} \\ 0 & 0 & 0 \end{bmatrix} \tag{28}$$

$$= \begin{bmatrix} 0 & u_{11}u_{22} - u_{12}u_{21} & u_{1132} - u_{12}u_{31} \\ u_{12}u_{21} - u_{11}u_{22} & 0 & u_{21}u_{32} - u_{22}u_{31} \\ u_{12}u_{31} - u_{11}u_{32} & u_{22}u_{31} - u_{21}u_{32} & 0 \end{bmatrix} \tag{29}$$

According to the property of orthogonal matrix : $u_3 = u_1 \times u_2$ (u_i is the i^{th} column of U). So we can rewrite S_1

$$S_1 = \begin{bmatrix} 0 & u_{33} & -u_{23} \\ -u_{33} & 0 & u_{13} \\ u_{23} & -u_{13} & 0 \end{bmatrix} \tag{30}$$

Look carefully at $[t_\times]$ and S_1, it is obvious that S_1 is totally same as $[t_\times]$

$$[t_\times] = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}, \quad S_1 = \begin{bmatrix} 0 & u_{33} & -u_{23} \\ -u_{33} & 0 & u_{13} \\ u_{23} & -u_{13} & 0 \end{bmatrix} \tag{31}$$

After comparing both matrices, we obtain t and restore it into a vector $\vec{t_1}$

$$\vec{t_1} = \begin{bmatrix} u_{13} \\ u_{23} \\ u_{33} \end{bmatrix} \tag{32}$$

Next, we find out $R^{'}$ is a orthogonal matrix

$$\begin{aligned} R^{'}R^{'T} &= (UWV^T)(VW^TU^T) && (33)\\ &= I && (34)\\ &= (VW^TU^T)(UWV^T) && (35)\\ &= R^{'T}R^{'} && (36) \end{aligned}$$

So we also obtain that $R^{'} = R$. As a result, **we can compute the extrinsic of camera $C^{'}$ by $\vec{t_1}$ and R.** (We will call this R as R_1 from now on)

2.4.2 Four Possible Extrinsic

However, if we change Σ into $-ZW^T$, E remains the same, but the result will be totally different.

$$E = U(-ZW^T)V^T = -(UZU^T)(UW^TV^T) \tag{37}$$

Let $S_2 = -UZU^T$, $R_2 = UW^TV^T$, and compare S_2 to $[t_\times]$

$$[t_\times] = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix}, \quad S_2 = \begin{bmatrix} 0 & -u_{33} & u_{23} \\ u_{33} & 0 & -u_{13} \\ -u_{23} & u_{13} & 0 \end{bmatrix} \tag{38}$$

The result of t is different, so we let this t be a new vector $\vec{t_2}$

$$\vec{t_2} = -\begin{bmatrix} u_{13} \\ u_{23} \\ u_{33} \end{bmatrix} \tag{39}$$

Since $E = UZWV^T = U(-ZW)V^T$, these four results : $\vec{t_1}$, $\vec{t_2}$, R_1, R_2 are all possible. So there will be four possible extrinsic matrix

$$[R_1 \mid t_1], \quad [R_1 \mid t_2], \quad [R_2 \mid t_1], \quad [R_2 \mid t_2] \tag{40}$$

```python
def compute_RT_from_essential(E):

    # make sure E is rank 2
    U,S,V = np.linalg.svd(E)
    e = (S[0]+S[1])/2

    E = np.dot(U, np.dot(np.diag([e,e,0]), V))

    Z = np.array([[0,1,0], [-1,0,0], [0,0,0]])
    W = np.array([[0,-1,0], [1,0,0], [0,0,1]])

    R1 = np.dot(U, np.dot(W,V))
    R2 = np.dot(U, np.dot(W.T,V))

    t1 = np.expand_dims(U[:,2], axis=1)
    t2 = np.expand_dims(-U[:,2], axis=1)

    return t1, t2, R1, R2
```

Figure 7: Code Snippet for Extrinsic Matrix

2.5 Triangulation

We obtain all the 2D coordinates in both images, projection matrices ($K[R \mid t]$) of both cameras, so we are ready to start to reconstruct the 3D object. The method to reproject 2D points to 3D world is called - **Triangulation**.

We will use new notation in this section, be careful not to confuse them with the notations in the previous sections. Let X be the 3D world coordinate, $\tilde{u}$ and $\tilde{u'}$ are the 2D coordinates in two images, P and $P^{'}$ are the projection matrices of two cameras. We know that

$$\tilde{u_i} = PX_i, \quad \tilde{u_i^{'}} = P^{'}X_i \tag{41}$$

Since any vector cross product itself will be zero, we can change Eq.41 to

$$u_i \times PX_i = 0, \quad u_i^{'} \times P^{'}X_i = 0 \tag{42}$$

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} \times \begin{bmatrix} p^{1T} \\ p^{2T} \\ p^{3T} \end{bmatrix} X_i = 0, \quad \begin{bmatrix} u_i^{'} \\ v_i^{'} \\ 1 \end{bmatrix} \times \begin{bmatrix} p^{'1T} \\ p^{'2T} \\ p^{'3T} \end{bmatrix} X_i = 0 \tag{43}$$

$$\begin{bmatrix} v_i p^{3T} - p^{2T} \\ p^{1T} - u_i p^{3T} \\ u_i p^{2T} - v_i p^{1T} \end{bmatrix} X_i = 0, \quad \begin{bmatrix} v_i^{'} p^{'3T} - p^{'2T} \\ p^{'1T} - u_i^{'} p^{'3T} \\ u_i^{'} p^{'2T} - v_i^{'} p^{'1T} \end{bmatrix} X_i = 0 \tag{44}$$

Since $u_i \times P$ is rank 2, so each perspective camera model gives rise to two equations on the three entries of X_i. We can reduce $u_i \times P$ into two rows.

$$\begin{bmatrix} v_i p^{3T} - p^{2T} \\ u_i p^{2T} - v_i p^{1T} \end{bmatrix} X_i = 0, \quad \begin{bmatrix} v_i^{'} p^{'3T} - p^{'2T} \\ u_i^{'} p^{'2T} - v_i^{'} p^{'1T} \end{bmatrix} X_i = 0 \tag{45}$$

We combine Eq.45 into one matrix multiplication

$$\begin{bmatrix} v_i p^{3T} - p^{2T} \\ u_i p^{2T} - v_i p^{1T} \\ v_i^{'} p^{'3T} - p^{'2T} \\ u_i^{'} p^{'2T} - v_i^{'} p^{'1T} \end{bmatrix} X_i = 0 \tag{46}$$

If we have N correspondences, we can write the Eq.46 as

$$A\vec{X_i} = \begin{bmatrix} v_1 p^{3T} - p^{2T} \\ u_1 p^{2T} - v_1 p^{1T} \\ v_1^{'} p^{'3T} - p^{'2T} \\ u_1^{'} p^{'2T} - v_1^{'} p^{'1T} \\ .. \\ .. \\ v_N p^{3T} - p^{2T} \\ u_N p^{2T} - v_N p^{1T} \\ v_N^{'} p^{'3T} - p^{'2T} \\ u_N^{'} p^{'2T} - v_N^{'} p^{'1T} \end{bmatrix} = 0 \tag{47}$$

So we use the SVD decomposition to get the least square solution of A in Eq.47, and the result will be our 3D points in the world coordinate.

```python
def linear_triangulation(pts1, pts2, P1, P2):

    # Initial (4*1) array, the first col is useless
    pts_3d = np.zeros((4,1))

    for i in range(pts1.shape[1]):
        A = np.array([
                pts1[0,i]*P1[2,:]-P1[0,:],
                pts1[1,i]*P1[2,:]-P1[1,:],
                pts2[0,i]*P2[2,:]-P2[0,:],
                pts2[1,i]*P2[2,:]-P2[1,:]
            ])
        U,S,V = np.linalg.svd(A)
        x = np.expand_dims((V[-1] / V[-1,3]), axis=1)
        pts_3d = np.hstack((pts_3d, x))

    return pts_3d[:, 1:]
```

Figure 8: Code Snippet for Triangulation

2.6 Choose the Correct Projection Matrix

Since there are four possible Projection Matrices for camera C', and there will be only one possibility that all 3D points are in front of 2 cameras. So we have to compute the forward vector of both cameras in order show the correct result of 3D reconstruction.

The rotation and transformation matrix we have got in the projection matrix is based on camera coordination. Since all the 3D points we have got is in world coordinate, so we have to change the camera center to the world coordinate and find its forward vector. We let $[X_{cam}, Y_{cam}, Z_{cam}]_W$ be the position of camera in the world coordinate.

The **camera position in the world coordinate** will be

$$\begin{bmatrix} X_{cam} \\ Y_{cam} \\ Z_{cam} \end{bmatrix}_W = R^T \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}_C - R^T t_c = -R^T t_c \tag{48}$$

The **forward vector of the camera** is the Z-axis in camera's coordinate

$$\begin{bmatrix}0\\0\\1\end{bmatrix}_C - \begin{bmatrix}0\\0\\0\end{bmatrix}_C = (R^T \begin{bmatrix}0\\0\\1\end{bmatrix}_C - R^T t_c) - \begin{bmatrix}X_{cam}\\Y_{cam}\\Z_{cam}\end{bmatrix}_W \tag{49}$$

$$= (R(3,:)^T - R^T t_c) - (-R^T t_c) \tag{50}$$

$$= R(3,:)^T \tag{51}$$

Then, we assume two vectors $\vec{V_1}$ and $\vec{V_2}$. $\vec{V_1}$ will be the vector from world center to a 3D point, and $\vec{V_2}$ will be the vector from world center to camera position.

$$\vec{V_1} = \begin{bmatrix}X_{pt}\\Y_{pt}\\Z_{pt}\end{bmatrix}_W - \begin{bmatrix}0\\0\\0\end{bmatrix}_W, \quad \vec{V_2} = \begin{bmatrix}X_{cam}\\Y_{cam}\\Z_{cam}\end{bmatrix}_W - \begin{bmatrix}0\\0\\0\end{bmatrix}_W \tag{52}$$

If the 3D point is in front of the camera, then the inner product of $\vec{V_1}$ and $\vec{V_2}$ will be greater than 0. So, apply $\vec{V_1} \cdot \vec{V_2}$ on all 3D points, and the one has the most points in front of both cameras will be the correct projection matrix.

```
# Calculate how many points are in front of camera : (X-CamCenter) dot (R[3, :].T)
res1 = np.dot(x[:3, 0].T, np.array([0,0,1]).T)

cam2_center = -np.dot(RT_comb[idx][0].T, RT_comb[idx][1])
res2 = np.dot((x[:3, 0] - cam2_center.T), np.expand_dims(RT_comb[idx][0][-1, :], axis=1))
if res1 > 0 and res2 > 0:
    count += 1
```

Figure 9: Code Snippet for choosing Projection Matrix

2.7 Construct 3D Model and Texture Mapping

Having all the 3D points in the world coordinate, we use matlab code TA provided to reconstruct the 3D model to generate .obj and .mtl files. Finally, we use Unity to map the texture to the model, and the result will be shown in next section. For more detail about our 3D model with texture, you may import *SfmTexture.unitypackage* to your own unity.

3 Result

3.1 Image Set 1

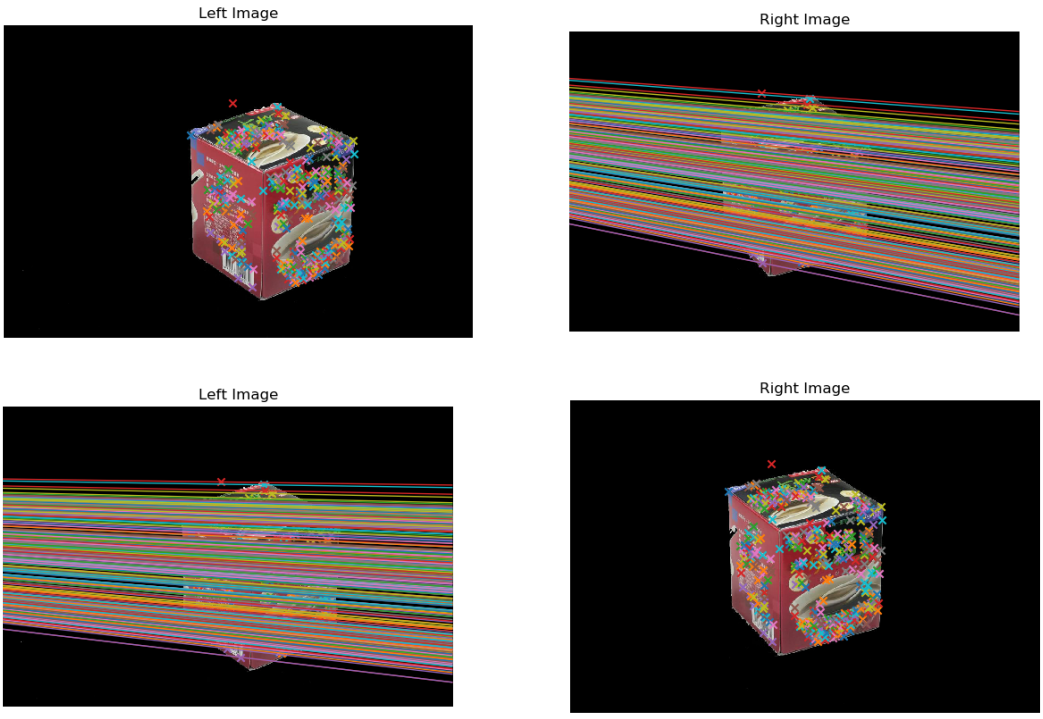


Figure 10: Epipolar Line of image set 1

```
Our Fundamental Matrix :
 [[-0.00000026 -0.0000054  -0.00022165]
 [ 0.00000446 -0.00000044  0.01714858]
 [-0.00033631 -0.01627834  1.        ]]
```

(a) Our Fundamental Matrix

```
OpenCV Fundamental Matrix :
 [[-0.00000092 -0.00001468  0.00352192]
 [ 0.00001398 -0.00000025  0.01193032]
 [-0.0030584  -0.01226006  1.        ]]
```

(b) OpenCV Fundamental Matrix

Figure 11: Fundamental Matrix of image set 1

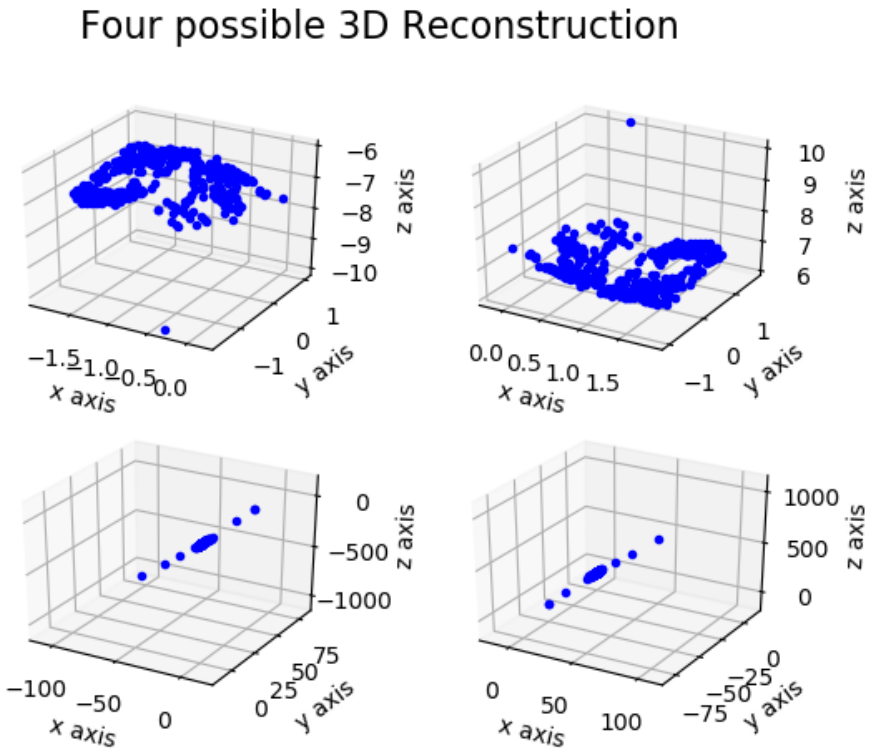
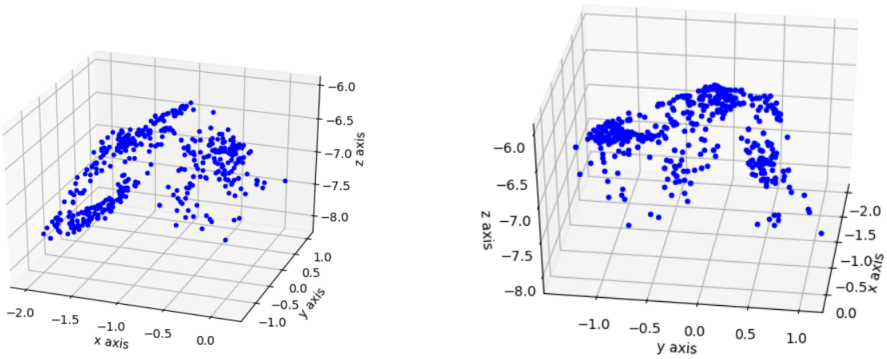


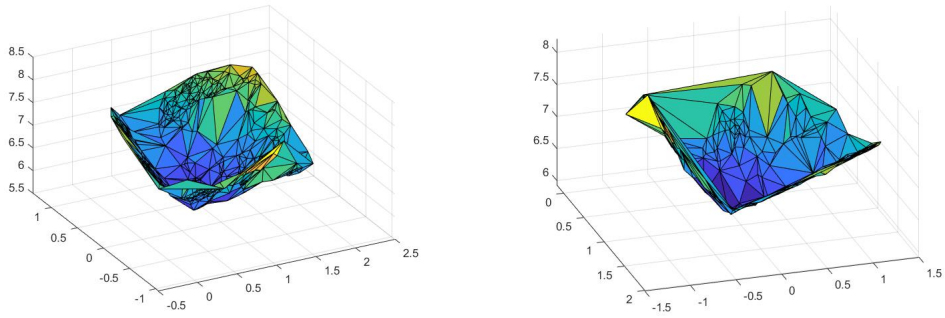
Figure 12: Four Possible Projection Matrix of image set 1



(a) Viewing Angle 1

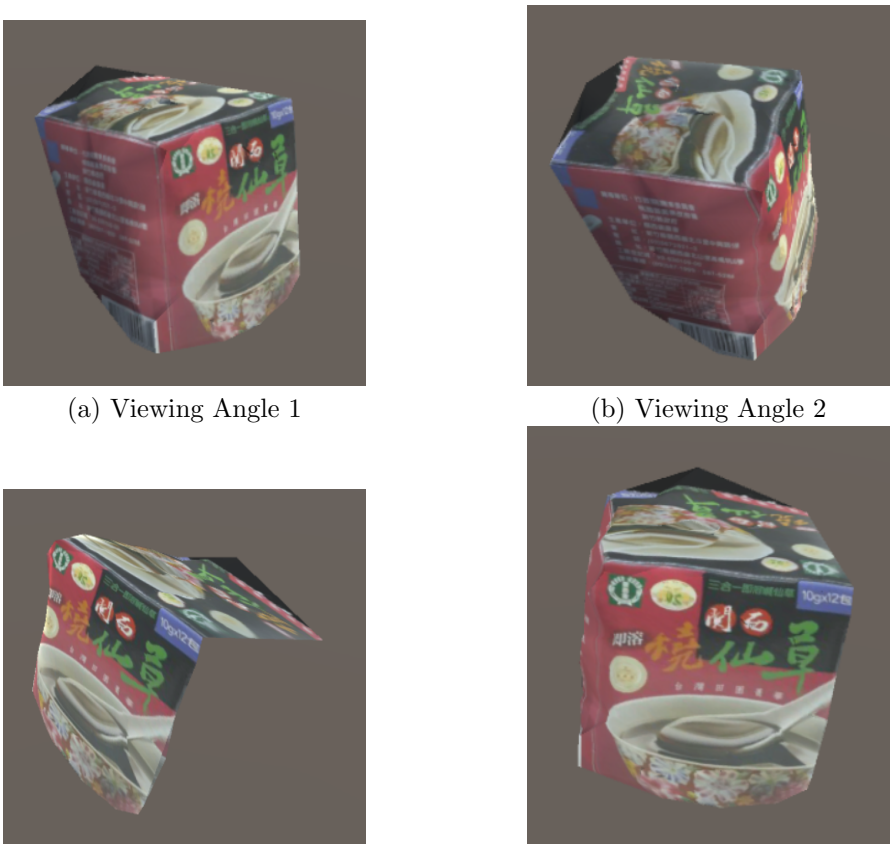
(b) Viewing Angle 2

Figure 13: Point Cloud by using the correct Projection Matrix



(a) Viewing Angle 1 (b) Viewing Angle 2

Figure 14: 3D Model of image set 1



(a) Viewing Angle 1 (b) Viewing Angle 2

(b) Viewing Angle 3 (b) Viewing Angle 4

Figure 15: Texture Mapping of image set 1

3.2 Image Set 2

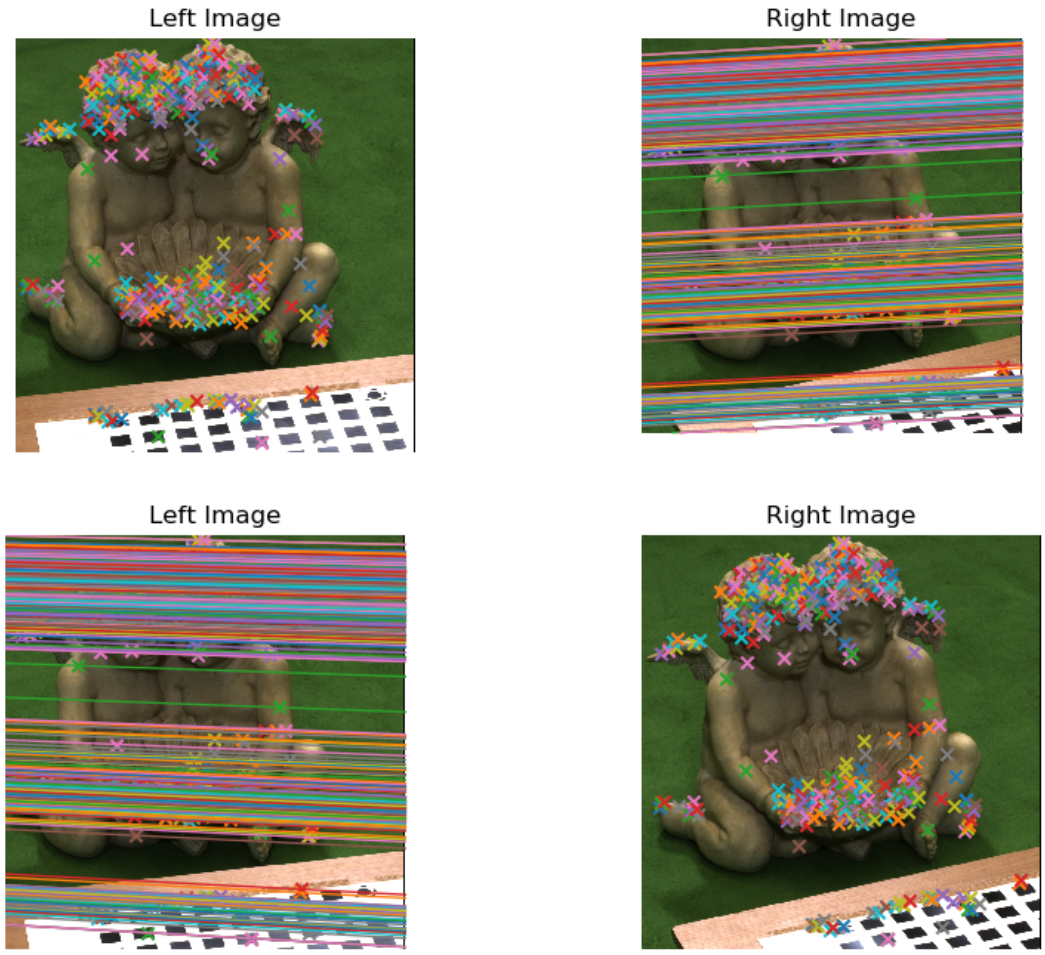


Figure 16: Epipolar Line of image set 2

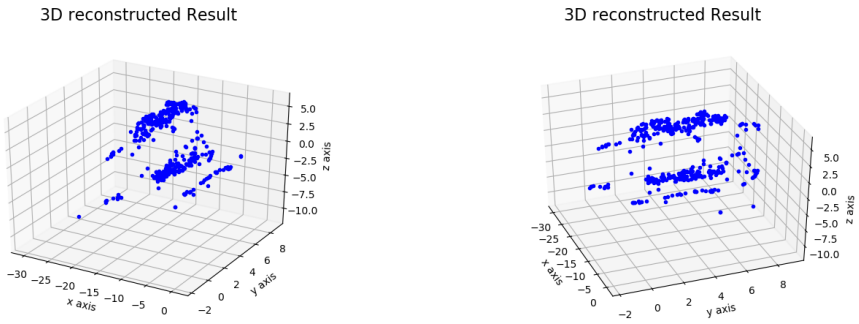
```
Our Fundamental Matrix :
 [[-0.00000006  0.00000011 -0.00151379]
 [-0.00000114 -0.00000006 -0.02780359]
 [-0.00059603  0.02850112  1.        ]]
```

(a) Our Fundamental Matrix

```
OpenCV Fundamental Matrix :
 [[ 0.00000034 -0.00000392  0.00131188]
 [ 0.00000311  0.00000033 -0.02867342]
 [-0.00370639  0.02907172  1.        ]]
```

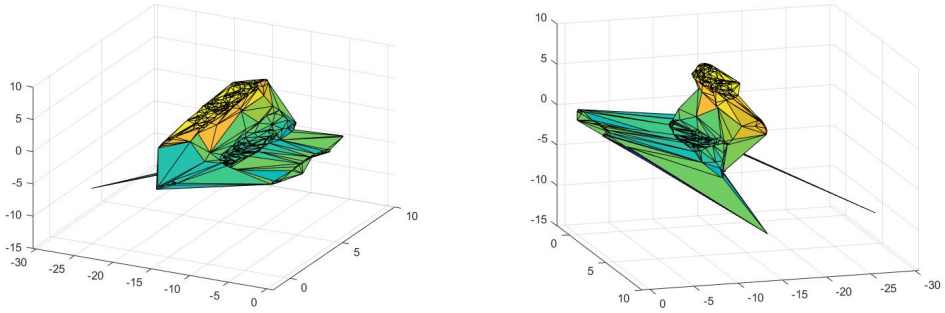
(b) OpenCV Fundamental Matrix

Figure 17: Fundamental Matrix of image set 2



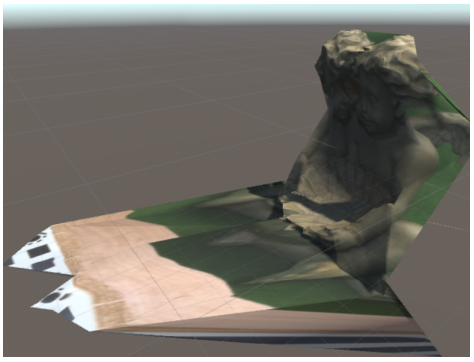
(a) Viewing Angle 1 (b) Viewing Angle 2

Figure 18: Point Cloud in different View Point

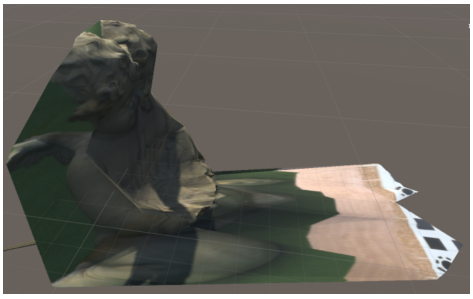


(a) Viewing Angle 1 (b) Viewing Angle 2

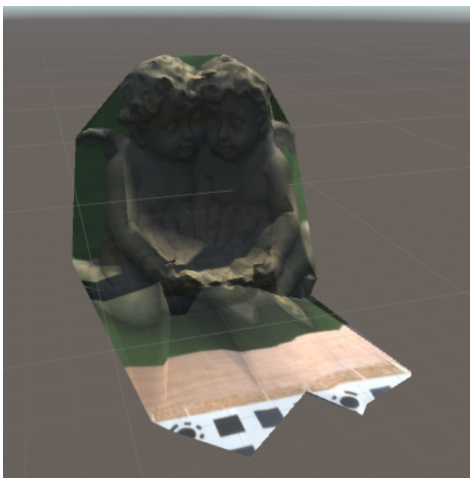
Figure 19: 3D Model of image set 2



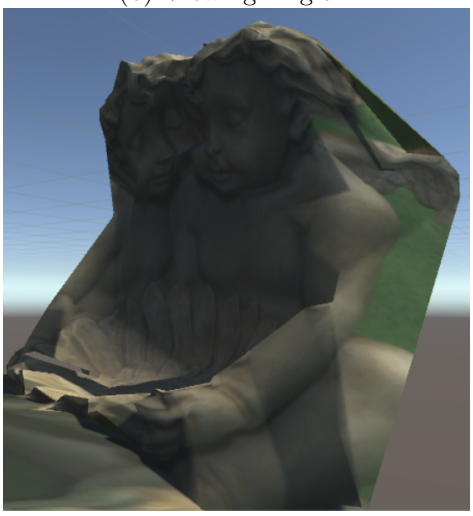
(a) Viewing Angle 1



(b) Viewing Angle 2



(b) Viewing Angle 3



(b) Viewing Angle 4

Figure 20: Texture Mapping of image set 2

3.3 Our Image

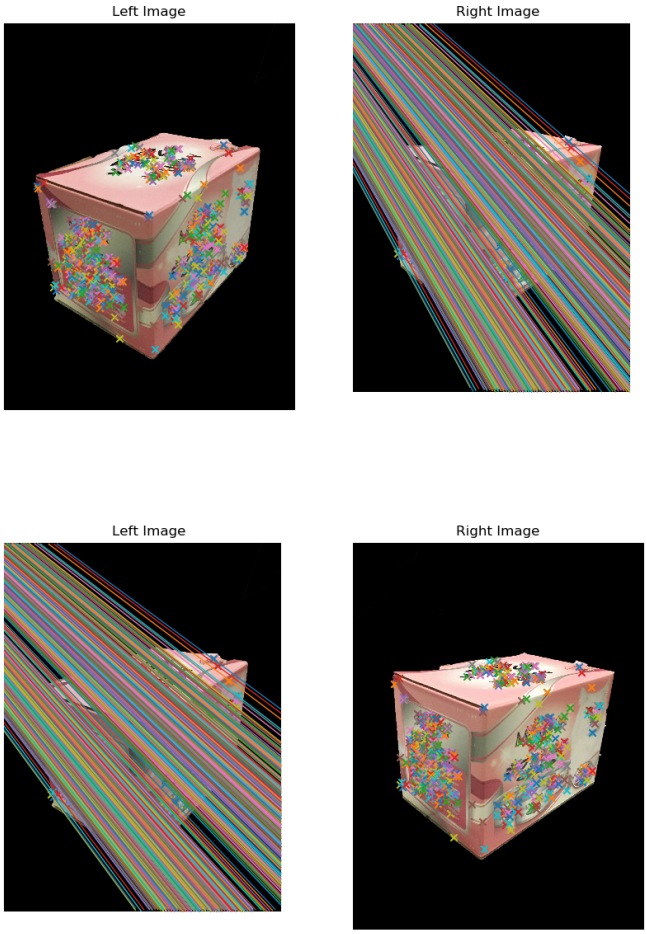


Figure 21: Epipolar Line of our image

```
Our Fundamental Matrix :
 [[ 0.00000001 -0.00000296 -0.00498467]
 [ 0.00000287 -0.00000051  0.00490776]
 [ 0.00421123 -0.00440095  1.        ]]
```

(a) Our Fundamental Matrix

```
OpenCV Fundamental Matrix :
 [[ 0.00000011 -0.00000236 -0.00495283]
 [ 0.00000224 -0.00000045  0.00475054]
 [ 0.00412031 -0.00432321  1.        ]]
```

(b) OpenCV Fundamental Matrix

Figure 22: Fundamental Matrix of our image

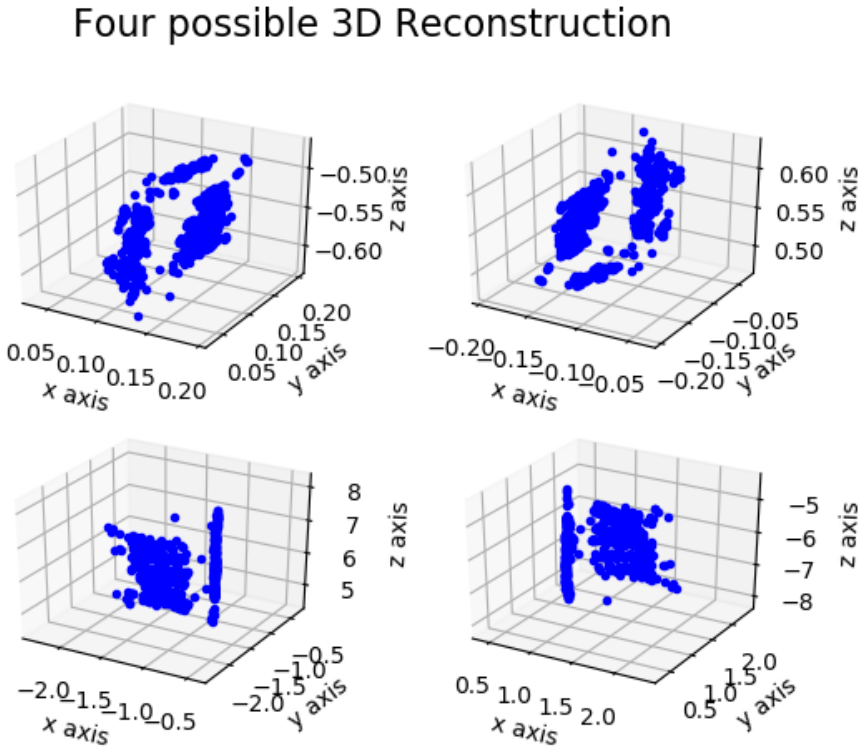
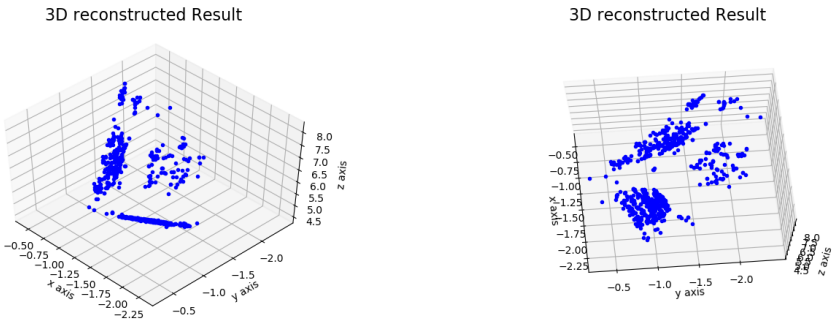


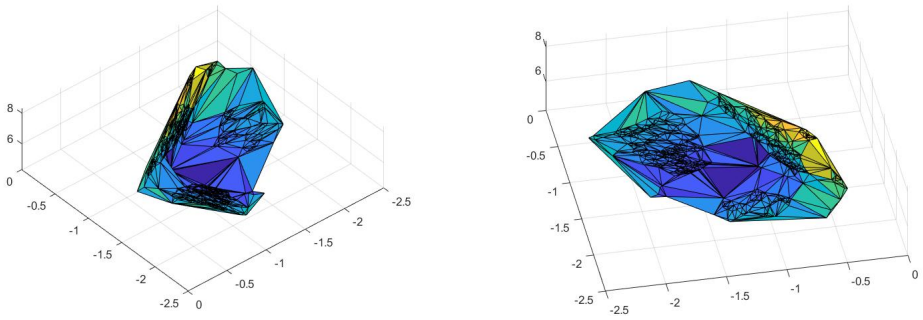
Figure 23: Four Possible Projection Matrix of our image



(a) Viewing Angle 1

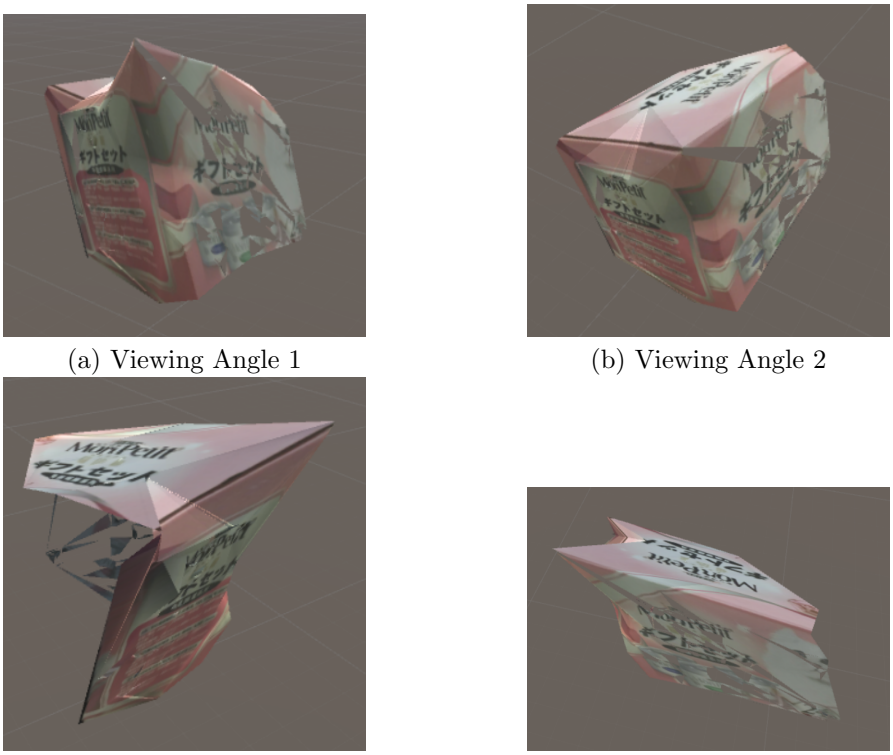
(b) Viewing Angle 2

Figure 24: Point Cloud by using the correct Projection Matrix



(a) Viewing Angle 1 (b) Viewing Angle 2

Figure 25: 3D Model of our image



(a) Viewing Angle 1 (b) Viewing Angle 2

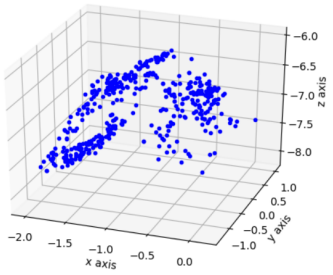
(b) Viewing Angle 3 (b) Viewing Angle 4

Figure 26: Texture Mapping of our image

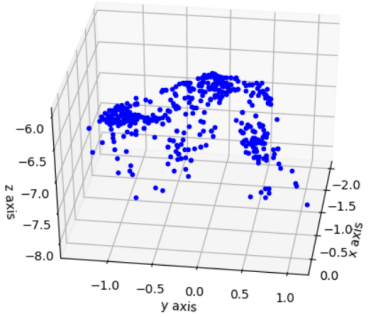
4 Discussion

4.1 SIFT v.s. SURF

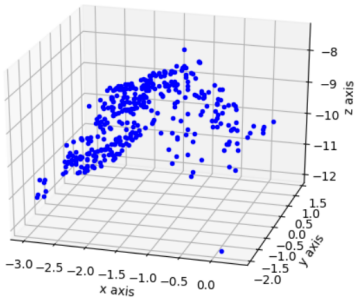
From the implementation above, we can see that the estimation fundamental matrix is based on the correspondences, and those correspondences are obtained from matching the keypoints in different image. It's obvious that the result will be different if we adopt different keypoint detector. To get the best result, we experiment with **SIFT** and **SURF** detector. From the result in Figure 27, we can see that the 3-D point cloud in SURF has more 3d point out of the shape of the box, and we think it might due to the fact that the keypoints found by SIFT is more distinctive while SURF is more repeatable. As a result, we condsier the result of SIFT is better and adopt it as our keypoint detector in SfM.



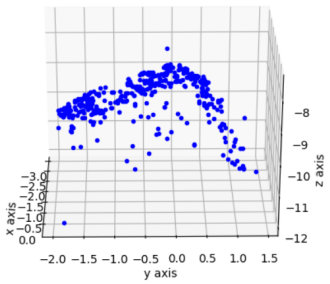
(a) SIFT Viewing Angle 1



(b) SIFT Viewing Angle 2



(c) SURF Viewing Angle 1



(d) SURF Viewing Angle 2

Figure 27: Point Cloud

4.2 Reprojection to World Coordinate

We've tried many images to test if our implementation really works. Figure 28 is one of our testing results.

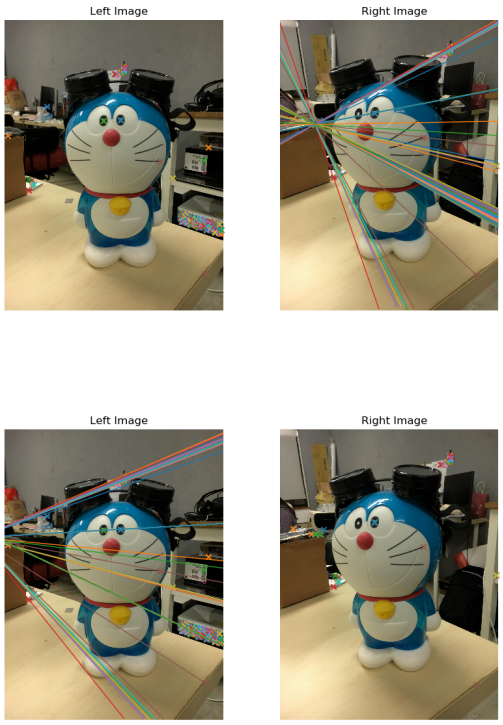


Figure 28: Epipolar Line of our image

We can see that there are nearly no points actually matching. The possible reason is that the object is reflectable, so it is difficult to find keypoints on it. But the main reason is that the background is too messy, so most of the keypoints are in the background. That is, we change our object to non-reflectable object, and also remove the background in the image.

Figure 29 is another testing result. We change the object to a **non-reflectable object**, and also **replace the background by a black image**.

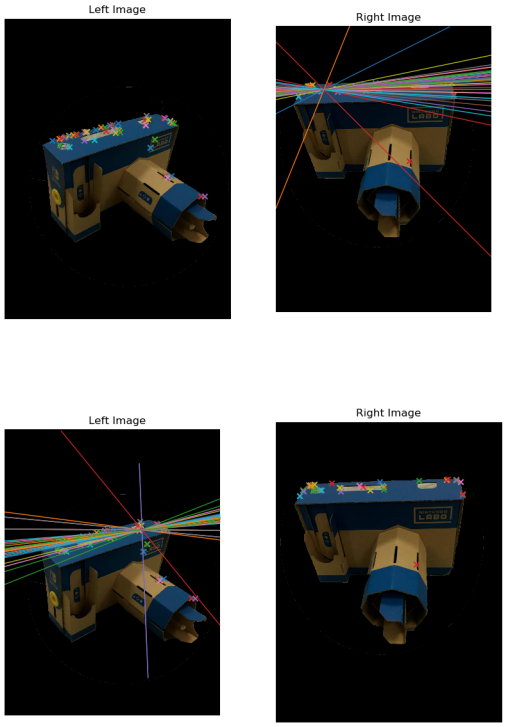


Figure 29: Epipolar Line of our image

However, this result is still not good enough. The keypoints are still not matching, and most part of the object is not detected by the detector. So we decide to change our object to a colorful object, and don't move too much between two images so that the detector may detect the keypoints more easily.

Finally, we choose a **colorful** and **non-reflective** object, **remove the background**, and **shift rarely** between two images. After all of the testing, the result finally looks well.

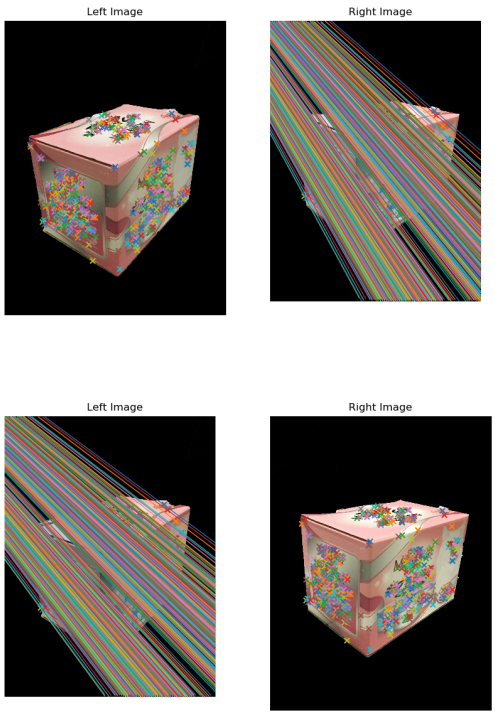


Figure 30: Epipolar Line of our image

As a result, this method does not work on every object and is highly correlated to the surroundings. The result may look totally different if the one of the factors we've chosen is different. So we consider it not so stable if we only use two images. Perhaps multiple images may work much better and is not so restricted.

5 Conclusion

In the homework, we implement the task of **Structure from Motion**. We implement the functions of **8-points algorithm** and **RANSAC** to estimate the fundamental matrix. Furthermore, we compare the difference between using SIFT and SURF as the keypoint detector. Essential matrix is computable after we obtain the fundamental matrix. Applying SVD on essential matrix, we obtain four different extrinsic matrices of the second camera. By checking if all the 3D points are in front of both cameras, we can find out the correct extrinsic matrix from four of them and use triangulation to reproject the 3D structure. We use the code provided by TA to draw the 3D model, and finally use Unity to map texture to the 3D model.

6 Work Assignment

0756079 陳冠聞: Implementation and report of detector, fundamental matrix. Comparison of SIFT and SURF.

0756138 黃伯凱: Rest of the implementation and report.

7 Appendix

7.1 Cross Product as a Matrix Multiplication

The cross product of a vector $\vec{a} = [a_1, a_2, a_3]^T$ with a vector $\vec{b} = [b_1, b_2, b_3]^T$

$$\vec{a} \times \vec{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} \tag{53}$$

The equation can be represented as a 3×3 matrix times vector $\vec{b}$. We rewrite $\vec{a}$ into a 3×3 matrix $[a_\times]$

$$[a_\times] = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \tag{54}$$

And now compute $[a_\times]\vec{b}$

$$[a_\times]\vec{b} = \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix} \tag{55}$$

The result is same as $\vec{a} \times \vec{b}$. So we can rewrite vector cross product into matrix multiplication by replacing $\vec{a}$ by a matrix $[a_\times]$.